BCSE307L
Compiler Design

Language translators - Compilation process – Modules – Interfaces – Tools - Lexical Analysis - Syntax Analysis - Semantic analysis – Translation - Intermediate Code - Basic Blocks - Instruction selection - Liveness Analysis - Dataflow Analysis – Code Optimization - , LEX and YACC tools.

Objective

- To provide fundamental knowledge of various language translators.
- To make students familiar with lexical analysis and parsing techniques.
- To understand the various actions carried out in semantic analysis.
- To make the students get familiar with how the intermediate code is generated.
- To understand the principles of code optimization techniques and code generation.
- To provide foundation for study of high-performance compiler design.

Outcomes

- Apply the skills on devising, selecting, and using tools and techniques towards compiler design
- Develop language specifications using context free grammars (CFG).
- Apply the ideas, the techniques, and the knowledge acquired for the purpose of developing software systems.
- Constructing symbol tables and generating intermediate code.
- Obtain insights on compiler optimization and code generation

Syllabus

- **Module: 1 Introduction to Compilation and Lexical Analysis 7 hours**
  - Introduction to LLVM - Structure and Phases of a Compiler-Design Issues-Patterns Lexemes-Tokens-Attributes-Specification of Tokens-Extended Regular Expression-Regular expression to Deterministic Finite Automata (Direct method) - Lex - A Lexical Analyzer Generator
- **Module: 2 Syntax Analysis 8 hours**
  - Role of Parser- Parse Tree - Elimination of Ambiguity – Top Down Parsing – Recursive Descent Parsing - LL (1) Grammars – Shift Reduce Parsers- Operator Precedence Parsing - LR Parsers, Construction of SLR Parser Tables and Parsing- CLR Parsing- LALR Parsing
- **Module: 3 Semantic Analysis 5 hours**
  - Syntax Directed Definition – Evaluation Order - Applications of Syntax Directed Translation - Syntax Directed Translation Schemes - Implementation of L-attributed Syntax Directed Definition
- **Module: 4 Intermediate Code Generation 5 hours**
  - Variants of Syntax trees - Three Address Code- Types – Declarations - Procedures - Assignment Statements - Translation of Expressions - Control Flow - Back Patching- Switch Case Statements.

Cont..

- **Module: 5 Code Optimization 6 hours**
  - Loop optimizations- Principal Sources of Optimization -Introduction to Data Flow Analysis - Basic Blocks - Optimization of Basic Blocks - Peephole Optimization- The DAG Representation of Basic Blocks -Loops in Flow Graphs - Machine Independent Optimization Implementation of a naïve code generator for a virtual Machine-Security checking of virtual machine code
- **Module: 6 Code Generation 5 hours**
  - Issues in the design of a code generator- Target Machine- Next-Use Information – Register Allocation and Assignment- Runtime Organization- Activation Records.
- **Module: 7 Parallelism 7 hours**
  - Parallelization- Automatic Parallelization- Optimizations for Cache Locality and Vectorization- Domain Specific Languages-Compilation- Instruction Scheduling and Software Pipelining- Impact of Language Design and Architecture Evolution on Compilers Static Single Assignment
- **Module: 8 Contemporary Issues 2 hours**

Text Books & References

- **Text Book**
- A. V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman, Compilers: Principles, techniques, & tools, 2007, Second Edition, Pearson Education, Boston.

- **Reference Books**
- Watson, Des. A Practical Approach to Compiler Construction. Germany, Springer International Publishing, 2017

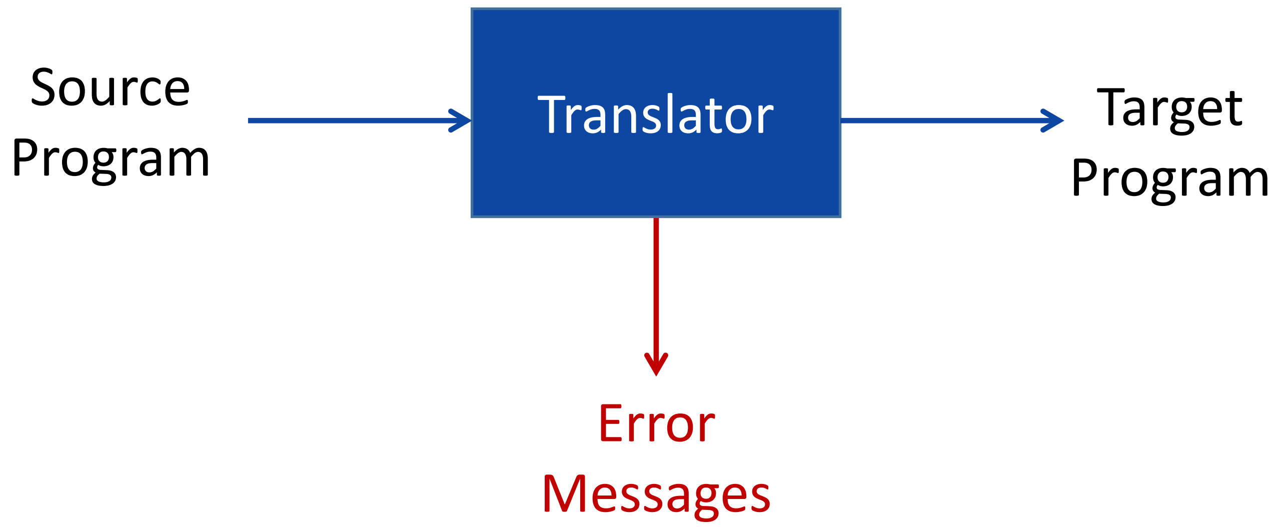
Teams Code:
G1 Slot:

Content - Module -1

- Introduction to Compilation And Lexical Analysis
  - Introduction to LLVM
  - Structure and Phases of a Compiler
  - Design Issues
  - Patterns Lexemes
  - Tokens-Attributes
  - Specification of Tokens
  - Extended Regular Expression
  - Regular expression to Deterministic Finite Automata (Direct method)
  - Lex - A Lexical Analyzer Generator

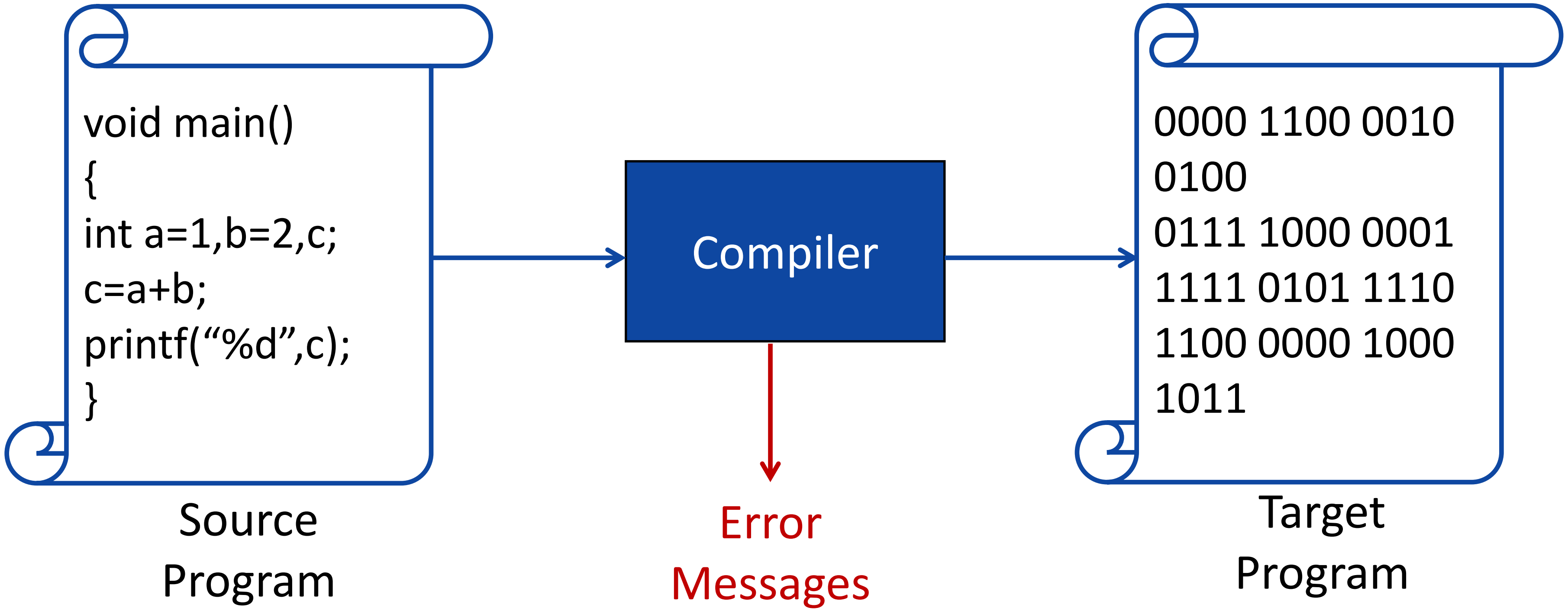
Translator

- A translator is a program that **takes one form of program as input** and **converts it into another form.**
- Types of translators are:
  1. Compiler
  2. Interpreter
  3. Assembler



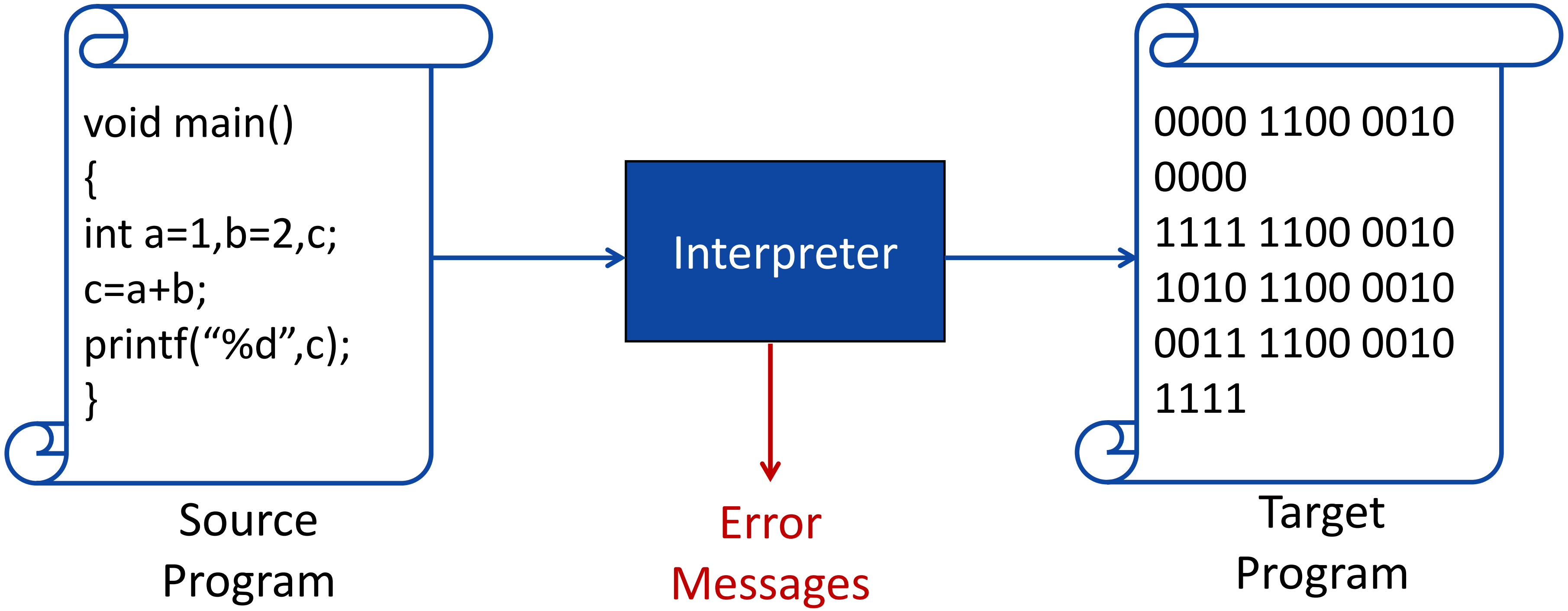
Compiler

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



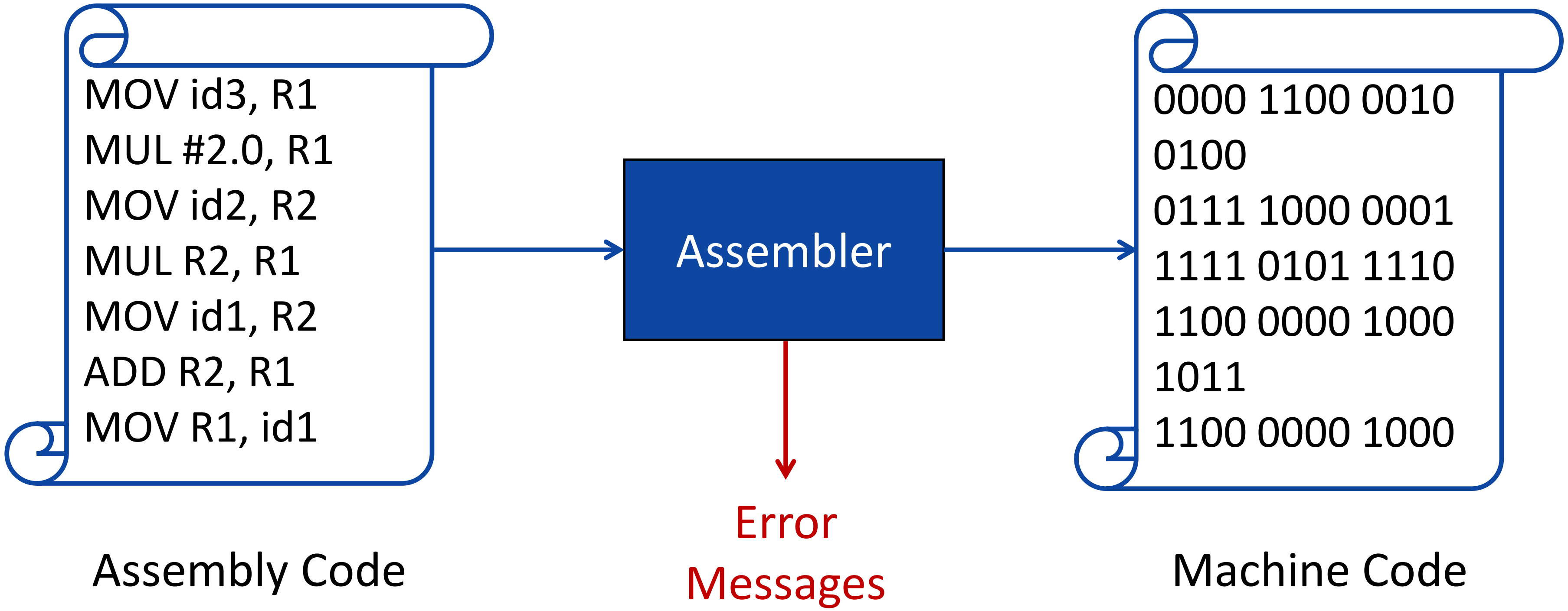
Interpreter

- Interpreter is also program that reads a program written in source language and translates it into an equivalent program in target language line by line



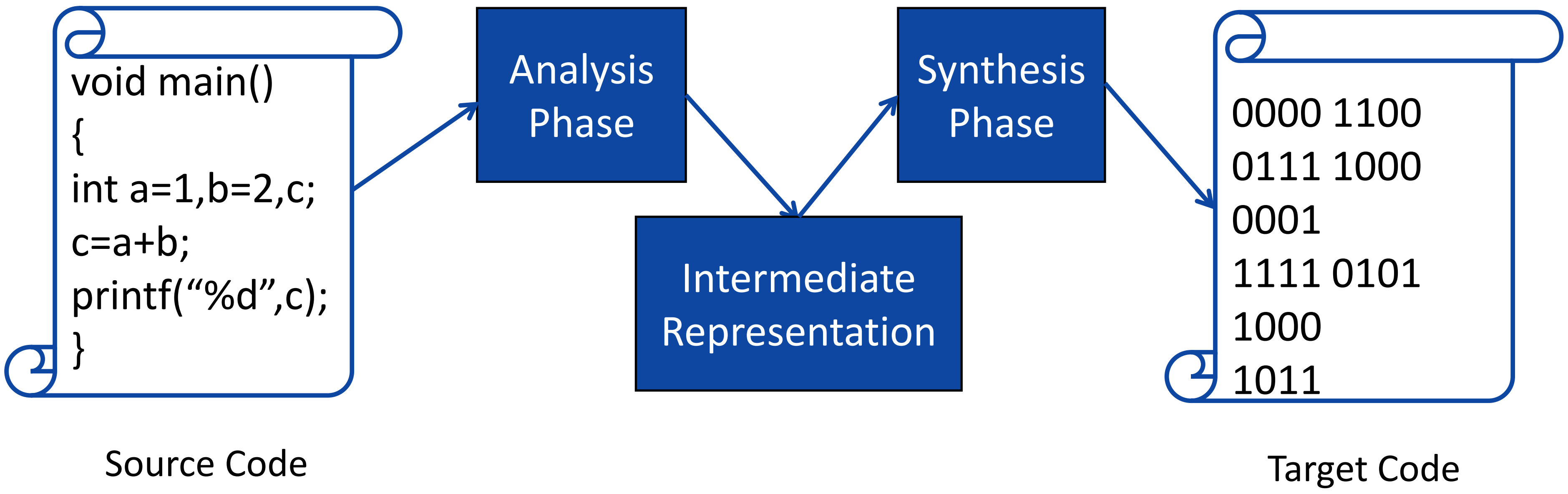
Assembler

- Assembler is a translator which takes the assembly code as an input and generates the machine code as an output.



Analysis Synthesis model of compilation

- **There are two parts of compilation.**
  1. Analysis Phase
  2. Synthesis Phase



Analysis phase & Synthesis phase

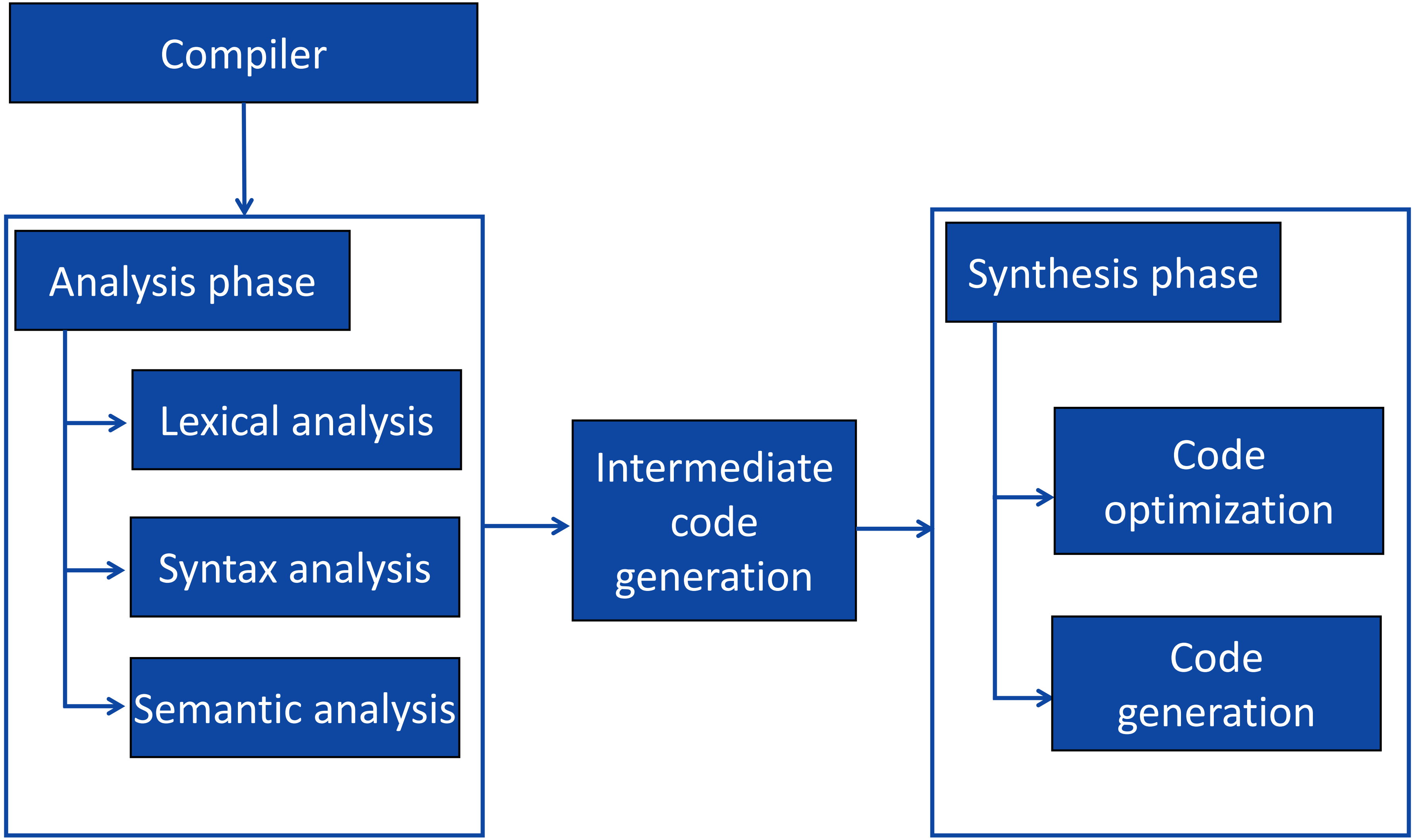
Analysis Phase

- Analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program.
- Analysis phase consists of three sub phases:
    1. Lexical analysis
    2. Syntax analysis
    3. Semantic analysis

Synthesis Phase

- The synthesis part constructs the desired target program from the intermediate representation.
- Synthesis phase consist of the following sub phases:
    1. Code optimization
    2. Code generation

Phases of compiler



Lexical analysis

- Lexical Analysis is also called **linear analysis** or **scanning.**
- Lexical Analyzer divides the given source statement into the **tokens**.
- Ex: Position = initial + rate * 60 would be grouped into the following tokens:

  Position (identifier)

  = (Assignment symbol)

  initial (identifier)

  + (Plus symbol)

  rate (identifier)

  * (Multiplication symbol)

  60 (Number)

Position = initial + rate*60

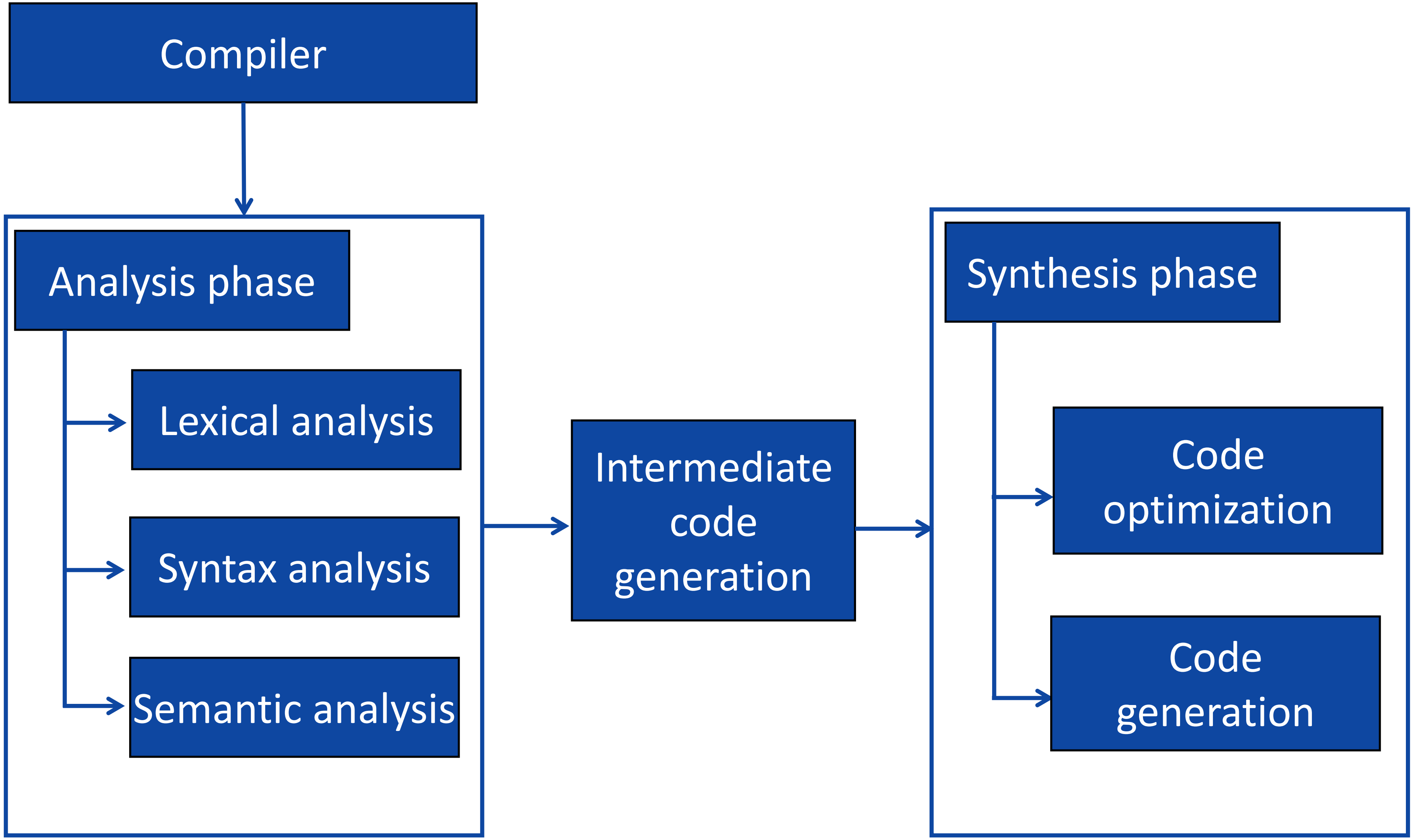
Lexical analysis

id1 = id2 + id3 * 60

Reads the stream of char making up the source program & group the char into meaningful sequences called lexeme.

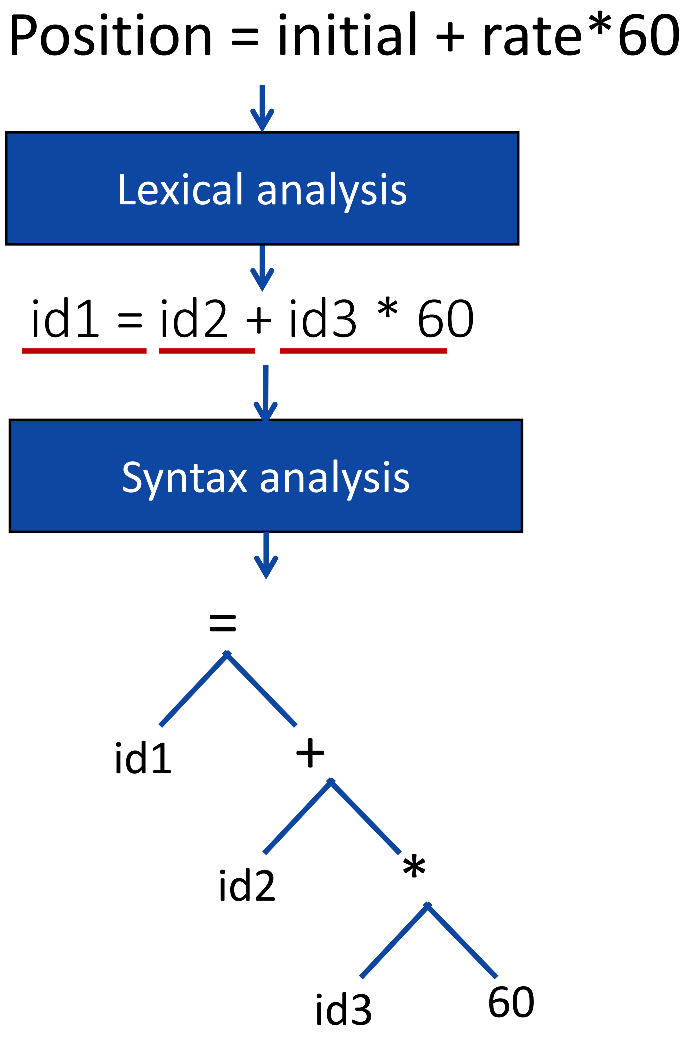
Lexical analyzer represents the lexeme in the form of tokens.

Phases of Compiler

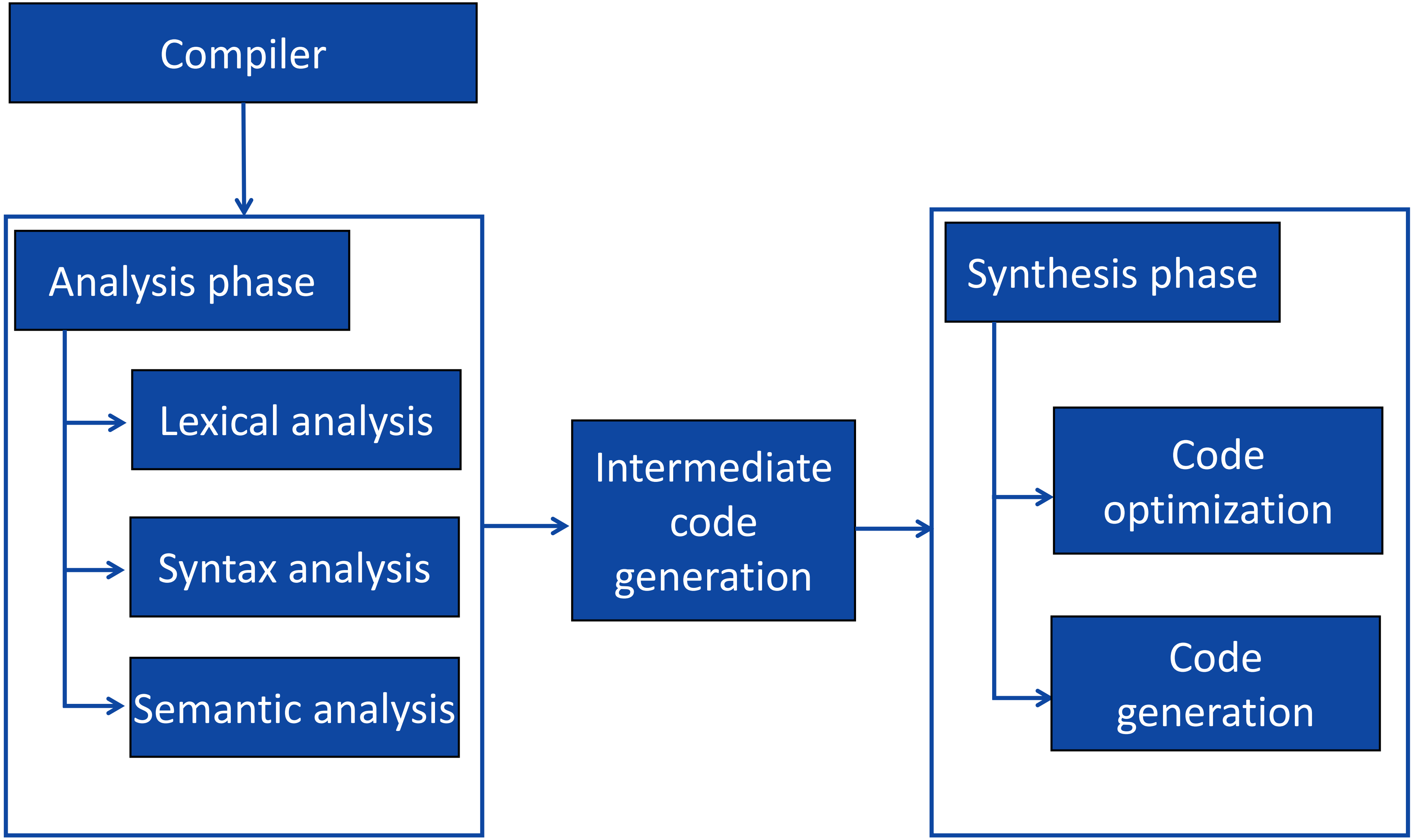


Syntax analysis

- Syntax Analysis is also called **Parsing** or **Hierarchical Analysis**.
- It takes token produced by lexical analyzer as Input & generates the parse tree.
- The syntax analyzer checks each line of the code and spots every tiny mistake.
- If code is error free then syntax analyzer generates the tree.



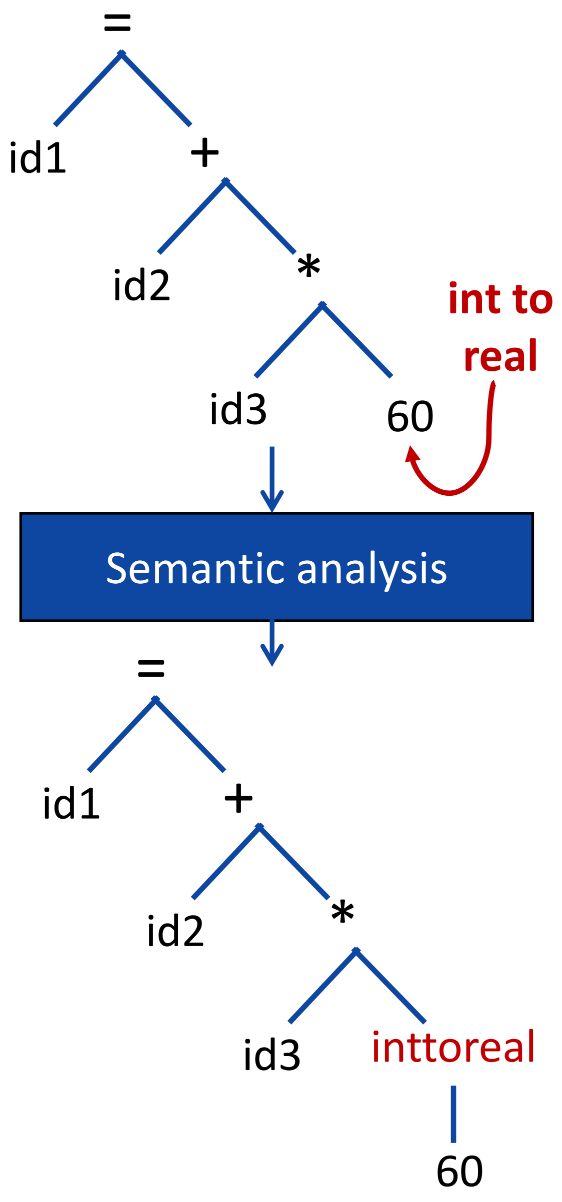
Phases of compiler



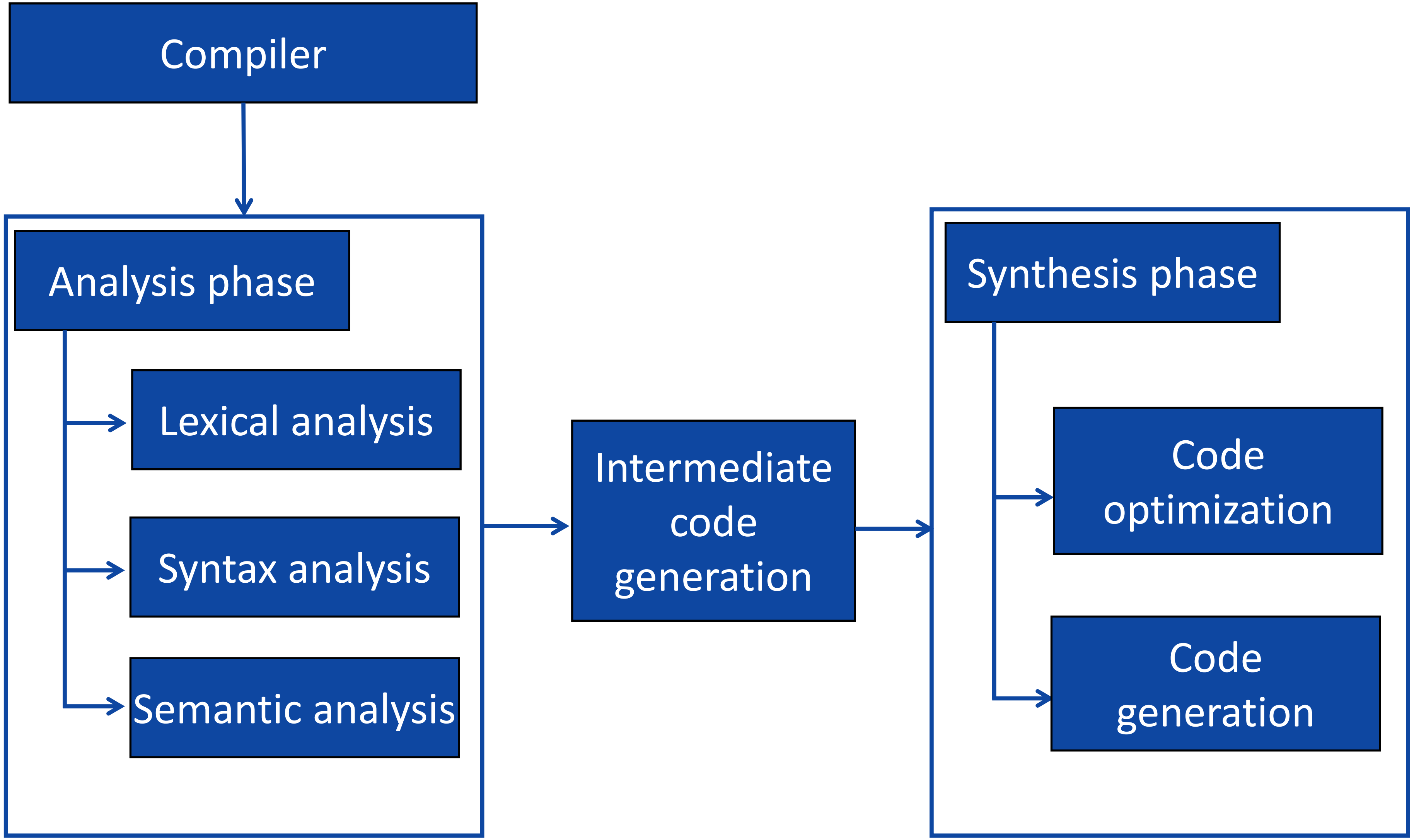
Semantic analysis

- Semantic analyzer determines the **meaning of a source string**.
- It performs following operations:
  1. matching of parenthesis in the expression.
  2. Matching of if..else statement.
  3. Performing arithmetic operation that are type compatible.
  4. Checking the scope of operation.

  *Note: Consider id1, id2 and id3 are real

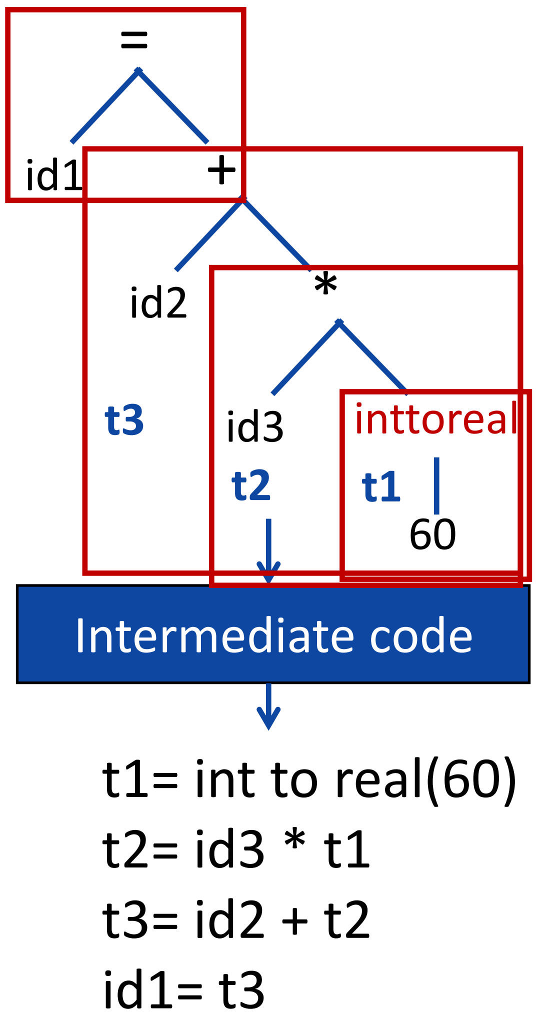


Phases of compiler

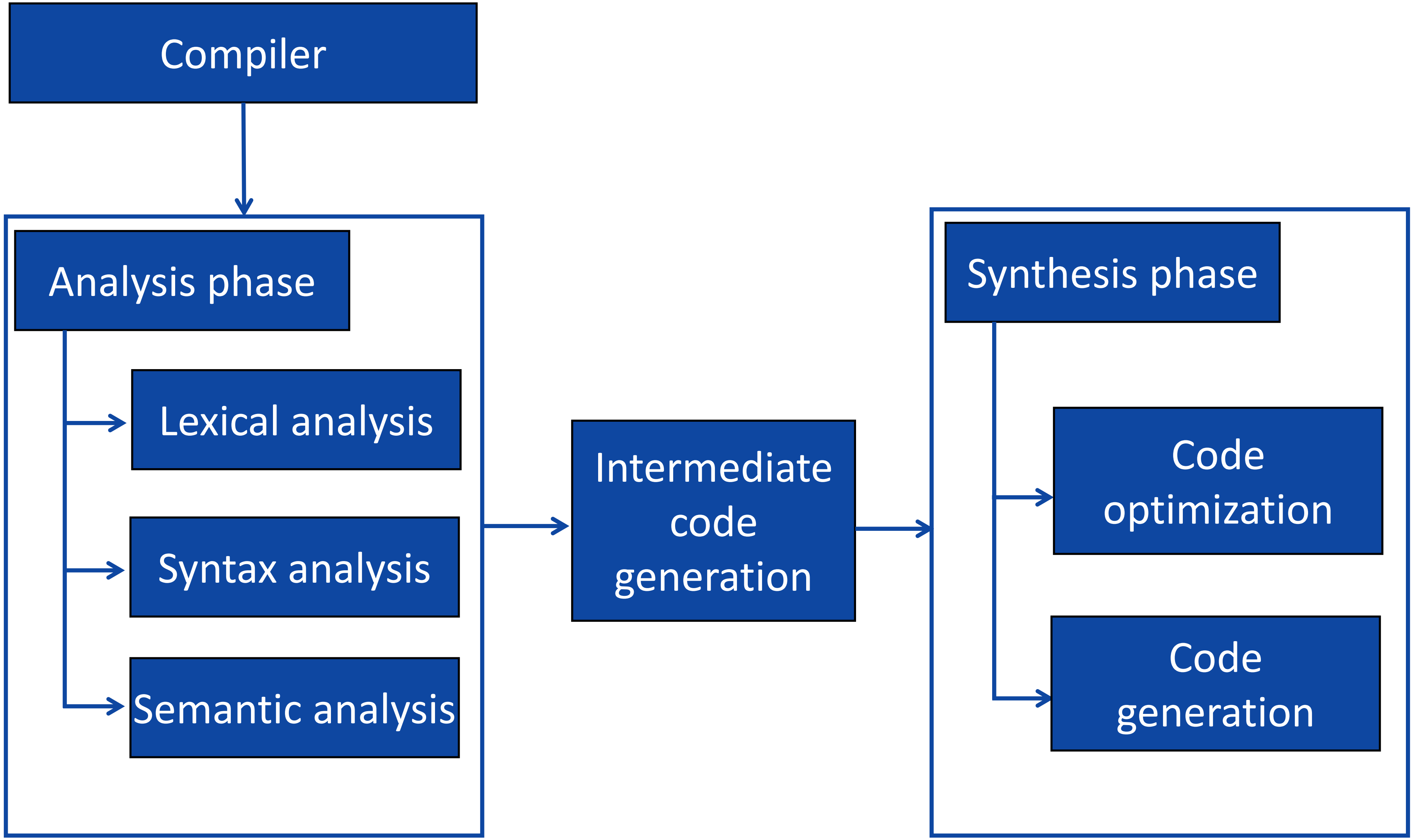


Intermediate code generator

- Two important properties of intermediate code :
  1. It should be **easy to produce**.
  2. **Easy to translate** into target program.
- Intermediate form can be represented using **"three address code".**
- Three address code consist of a sequence of instruction, each of which has **at most three** operands.

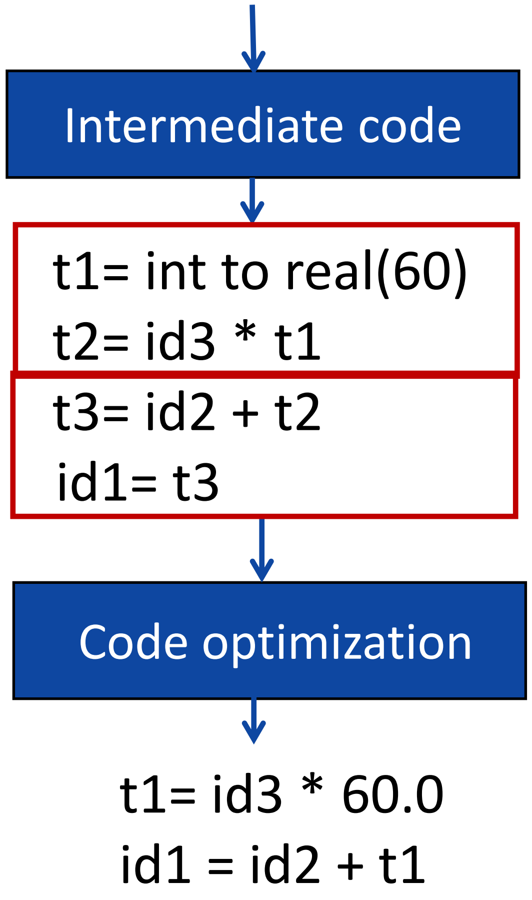


Phases of compiler

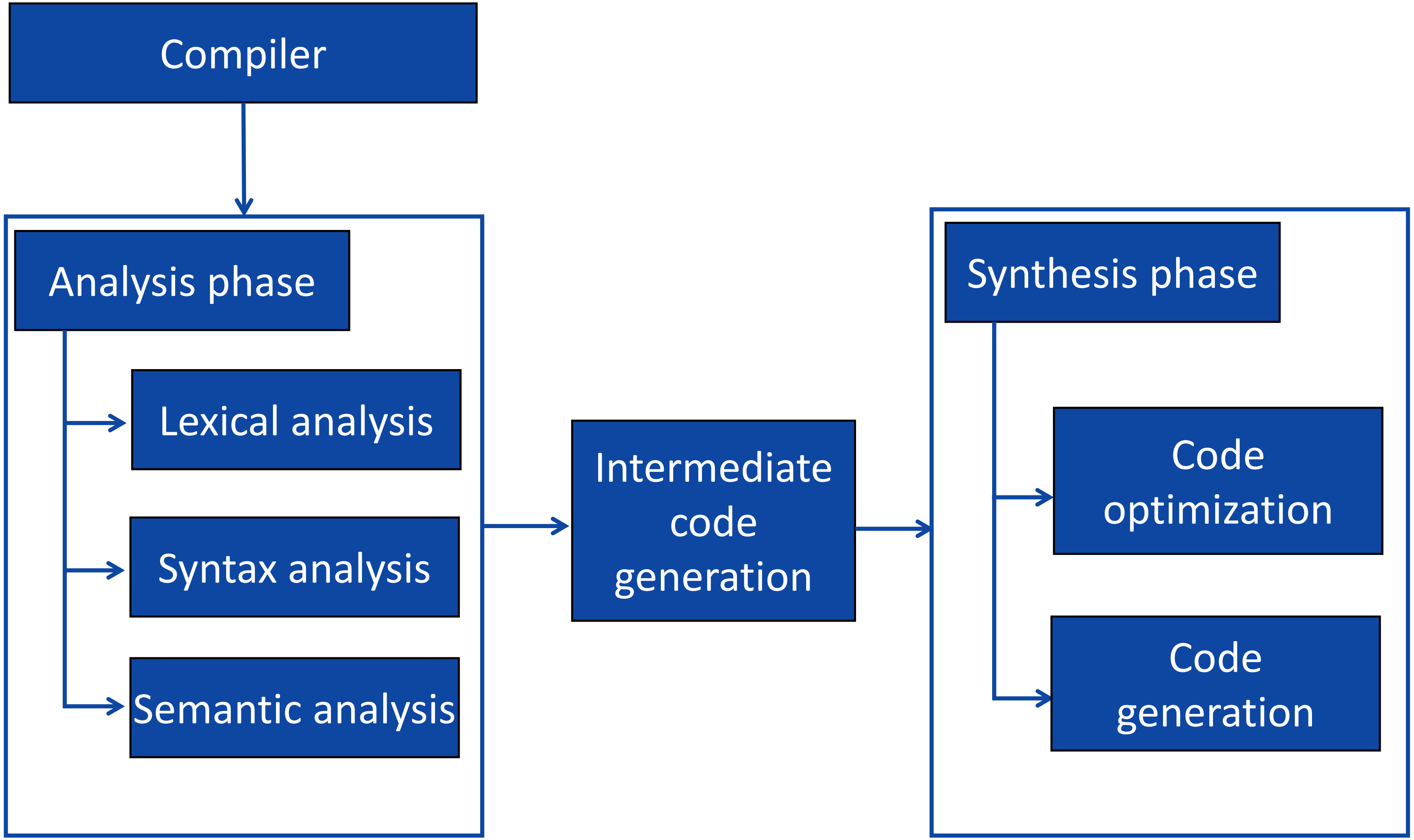


Code optimization

- It **improves** the intermediate code.
- This is necessary to have a **faster execution** of code or **less consumption of memory**.



Phases of compiler



Code generation

- The intermediate code instructions are **translated into sequence of machine instruction**.

↓

Code optimization

↓

t1= id3 * 60.0

id1 = id2 + t1

↓

Code generation

↓

MOV id3, R2
MUL #60.0, R2
MOV id2, R1
ADD R2,R1
MOV R1, id1

Id3→R2
Id2→R1

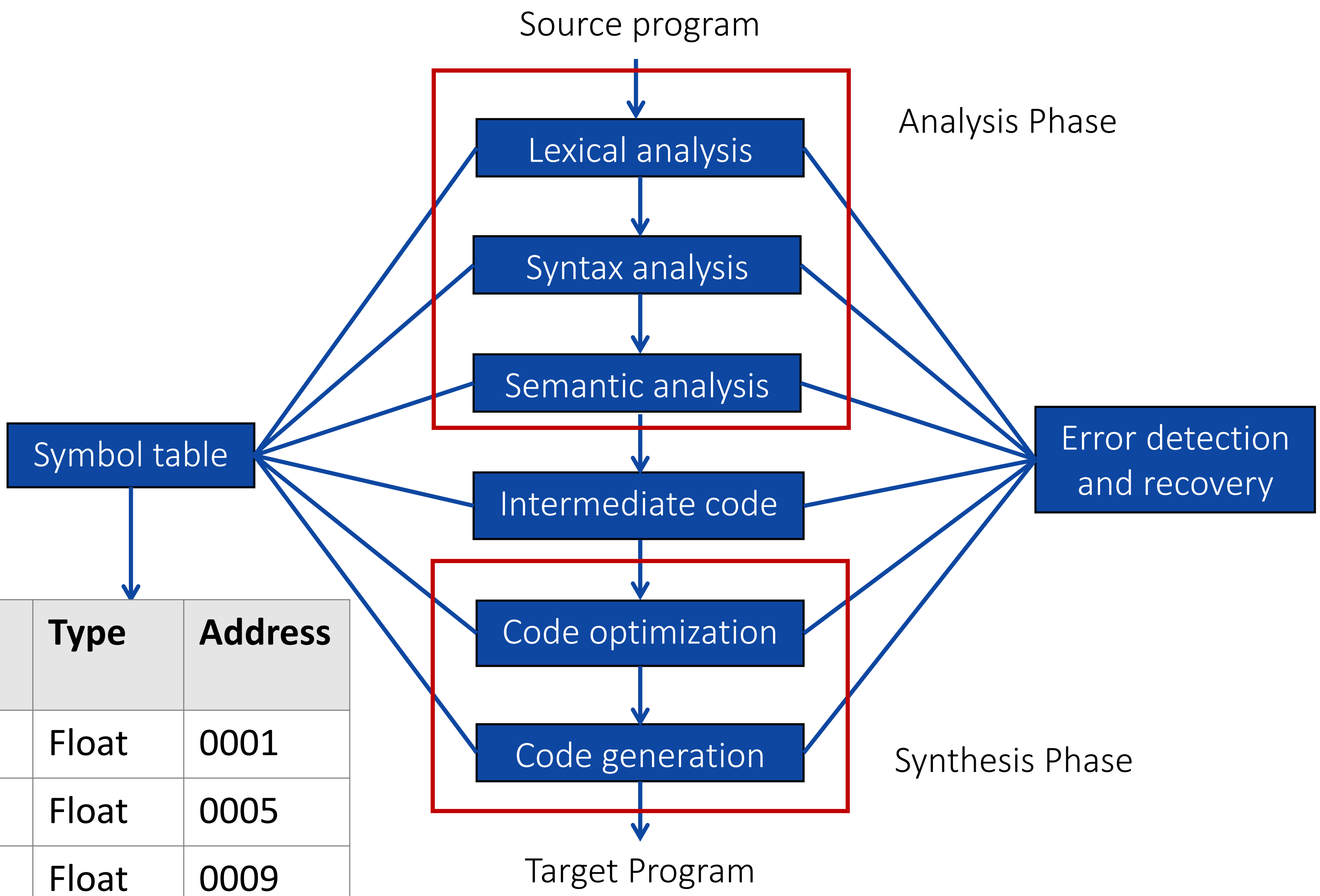
Symbol table

- Symbol table are data structures that are used by compilers to **hold information about source-program constructs.**
- It is used to store information about the occurrences of various entities such as, **objects, classes, variable names, functions, etc.,**
- It is used by both **analysis phase and synthesis phase**.
- Symbol table is used for the following purposes
    - It is used to store the name of all the entities in a structured form at one place
    - It is used to verify if a variable has been declared
    - It is used to determine the scope of a name
    - It is used to implement type checking by verifying assignments and expression in the source code are semantically correct.

Cont.,

- Symbol table can be a linear (Linked list) or hash table
- It maintains an entry for each name as,
  - <symbol name, type, attribute>
  - **Eg. <static, int, age>**

Phases of compiler



Variable Name	Type	Address
Position	Float	0001
Initial	Float	0005
Rate	Float	0009

Exercise 1

- Write output of all the phases of compiler for following statements:
    1. x = b-c*2
    2. I=p*n*r/100

Grouping of Phases

Front end & back end (Grouping of phases)

Front end

- Depends primarily on source language and largely independent of the target machine.
- It includes following phases:
    1. Lexical analysis
    2. Syntax analysis
    3. Semantic analysis
    4. Intermediate code generation
    5. Creation of symbol table

Back end

- Depends on target machine and do not depends on source program.
- It includes following phases:
    1. Code optimization
    2. Code generation phase
    3. Error handling and symbol table operation

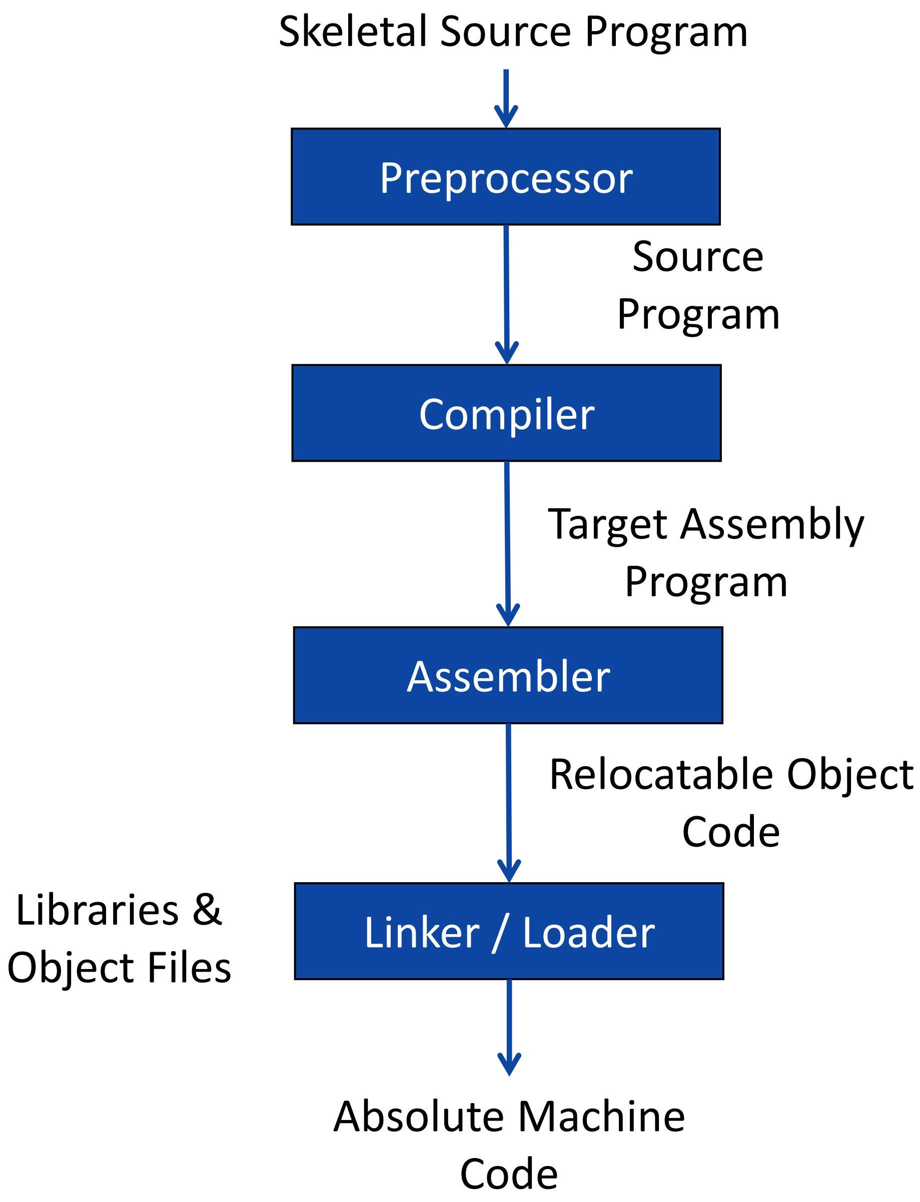
Difference between compiler & interpreter

Compiler	Interpreter
Scans the entire program and translates it as a whole into machine code.	It translates program's one statement at a time.
It generates intermediate code.	It does not generate intermediate code.
An error is displayed after entire program is checked.	An error is displayed for every instruction interpreted if any.
Memory requirement is more.	Memory requirement is less.
Example: C compiler	Example: Basic, Python, Ruby

Context of Compiler
(Cousins of compiler)

Context of compiler (Cousins of compiler)

- In addition to compiler, many other system programs are required to generate absolute machine code.
- These system programs are:
  - Preprocessor
  - Assembler
  - Linker
  - Loader

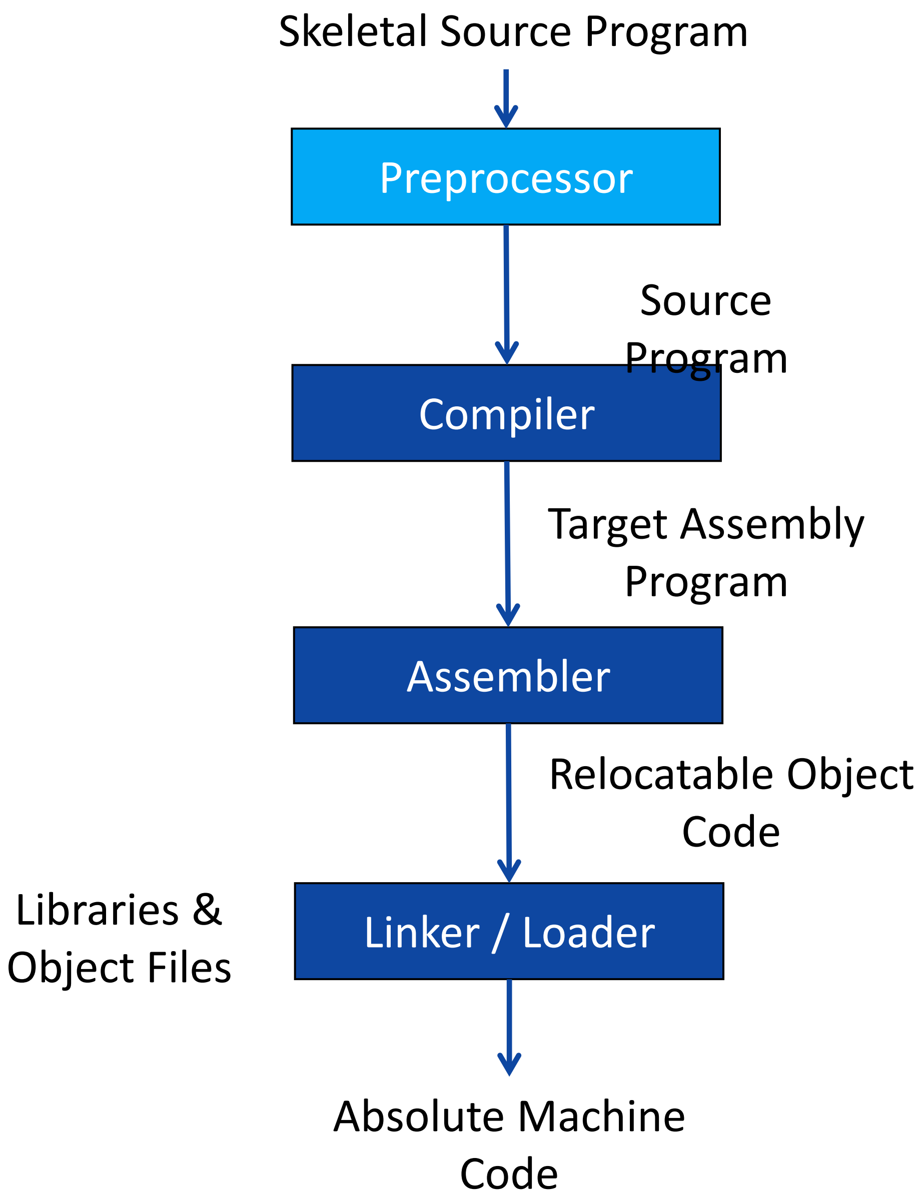


Context of compiler (Cousins of compiler)

Preprocessor

- **Some of the task performed by preprocessor:**

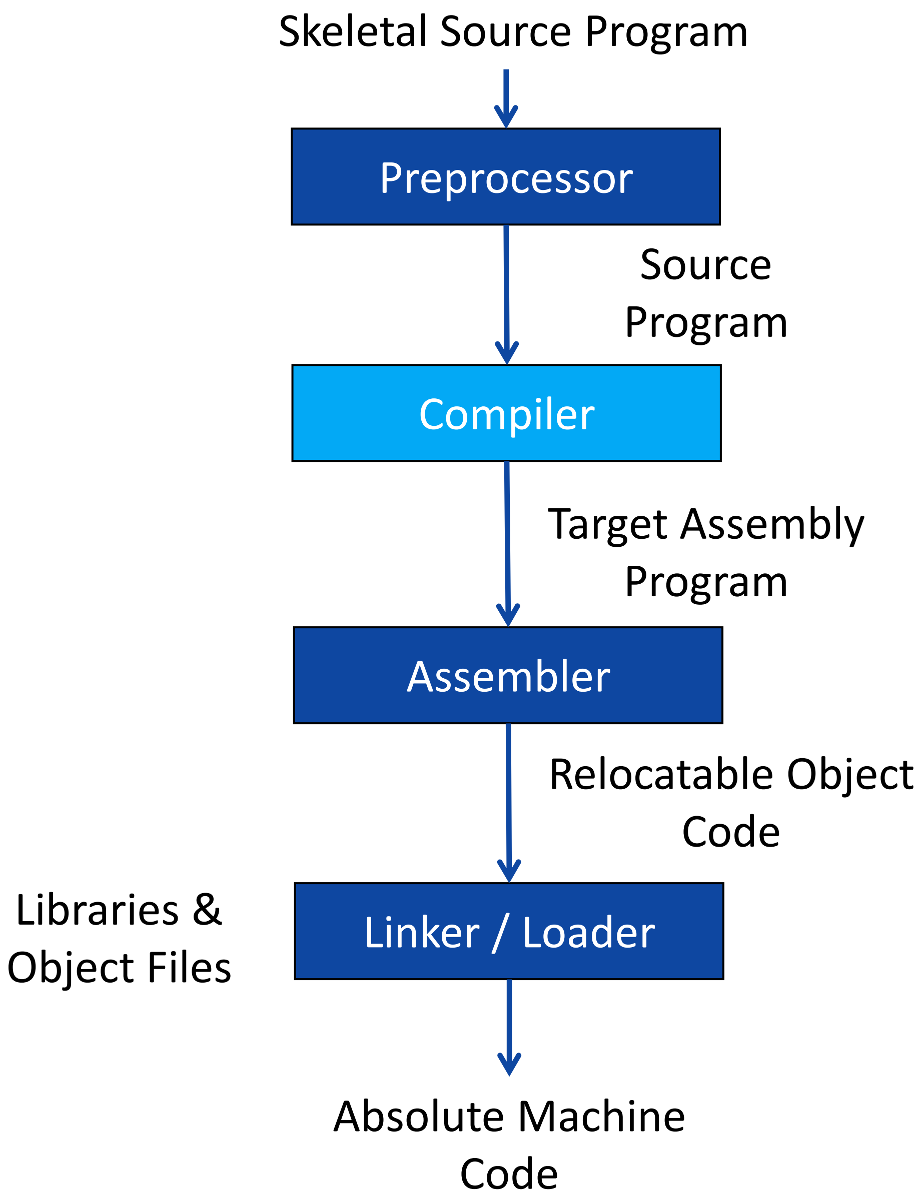
1. **Macro processing**: Allows user to define macros. Ex: #define PI 3.14159265358979323846
2. **File inclusion**: A preprocessor may include the header file into the program. Ex: #include<stdio.h>
3. **Rational preprocessor**: It provides built in macro for construct like while statement or if statement.
4. **Language extensions**: Add capabilities to the language by using built-in macros.
   - Ex: the language equal is a database query language embedded in C. Statement beginning with ## are taken by preprocessor to be database access statement unrelated to C and translated into procedure call on routines that perform the database access.



Context of compiler (Cousins of compiler)

Compiler

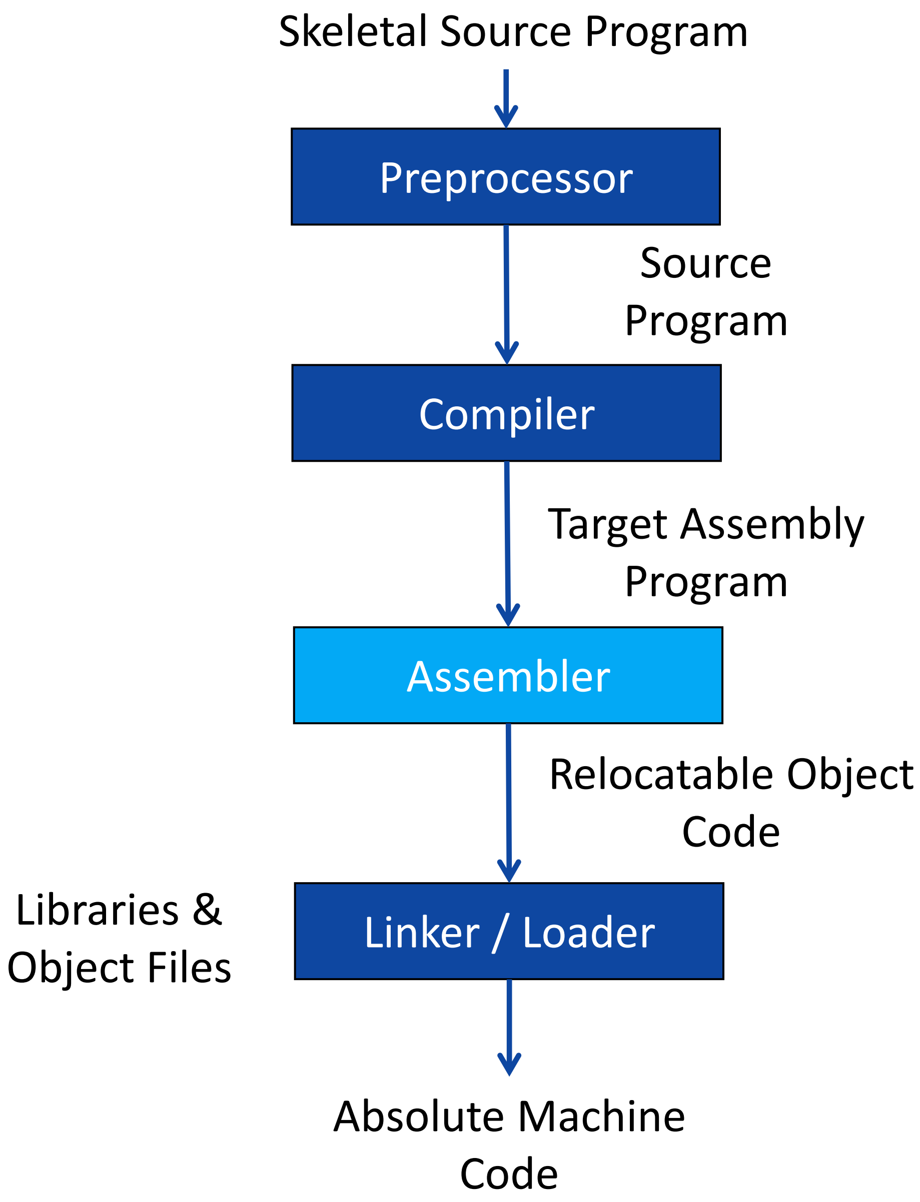
- A compiler is a program that reads a program written in source language and translates it into an equivalent program in target language.



Context of compiler (Cousins of compiler)

Assembler

- Assembler is a translator which takes the assembly program (mnemonic) as an input and generates the machine code as an output.



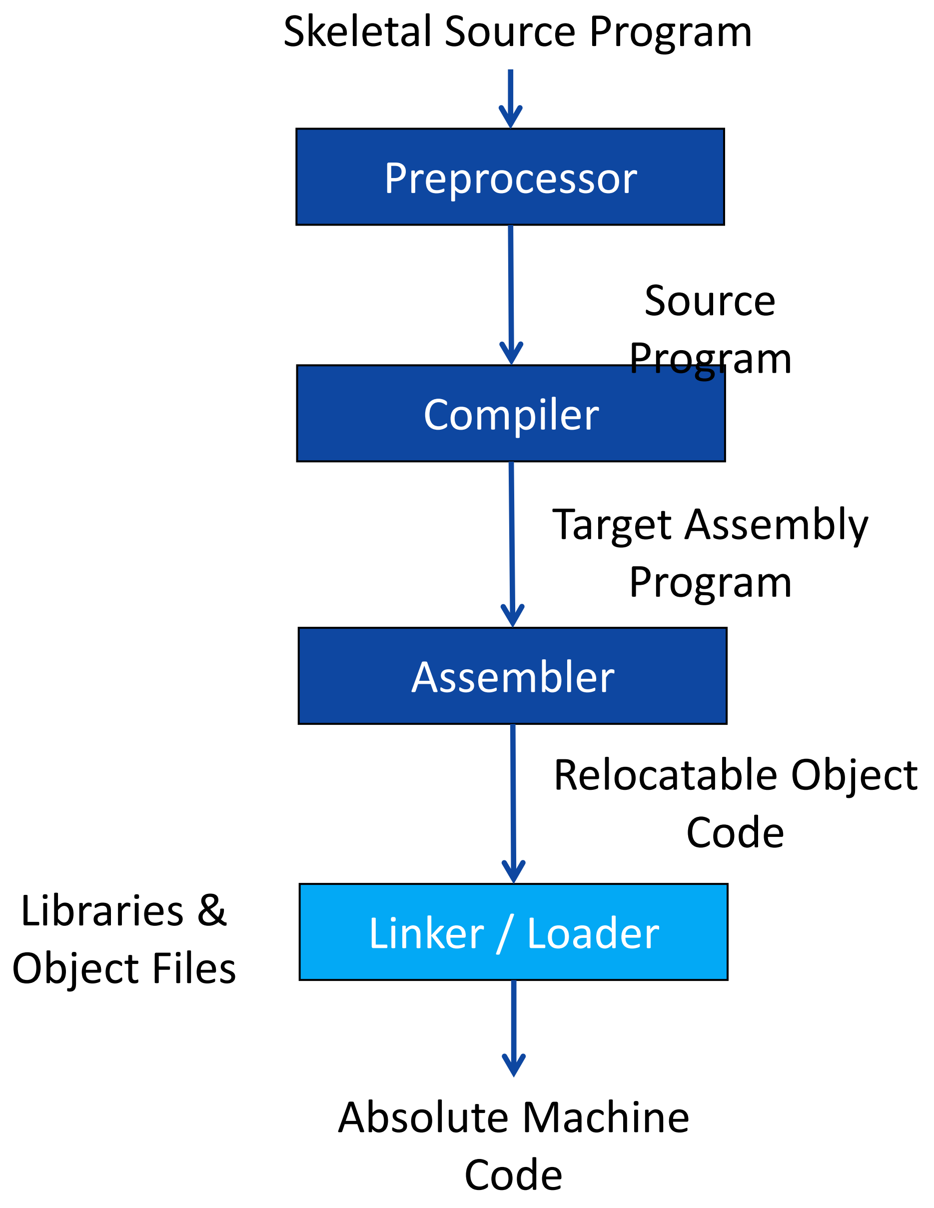
Context of compiler (Cousins of compiler)

Linker

- Linker makes a single program from a several files of relocatable machine code.
- These files may have been the result of several different compilation, and one or more library files.

Loader

- The process of loading consists of:
  - Taking relocatable machine code
  - Altering the relocatable address
  - Placing the altered instructions and data in memory at the proper location.



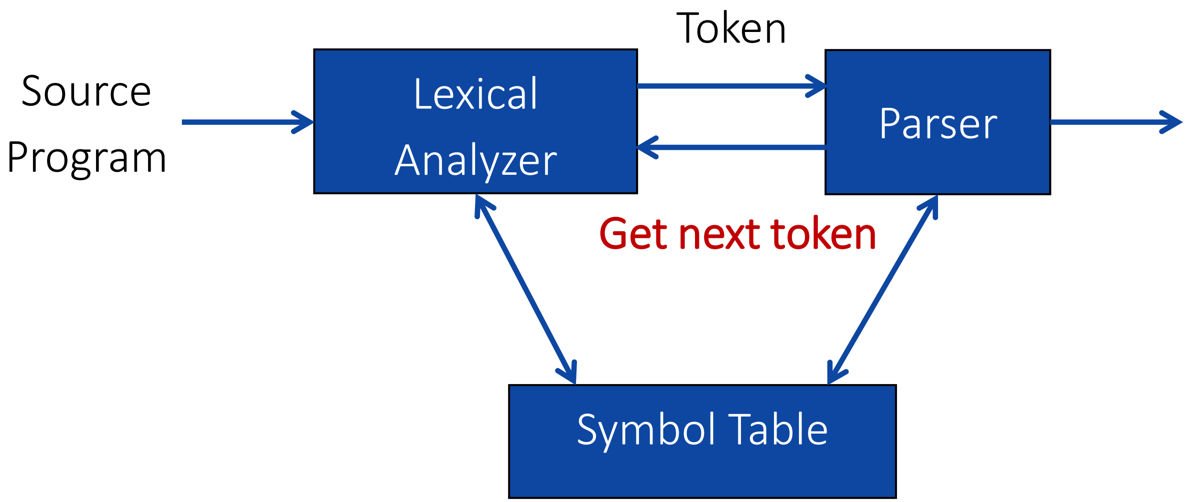
Types of compiler

Types of compiler

1. One pass compiler
    - It is a type of compiler that compiles whole process in one-pass.
2. Two pass compiler
    - It is a type of compiler that compiles whole process in two-pass.
    - It generates intermediate code.
3. Incremental compiler
    - The compiler which compiles only the changed line from the source code and update the object code.
4. Native code compiler
    - The compiler used to compile a source code for a same type of platform only.
5. Cross compiler
    - The compiler used to compile a source code for a different kinds platform.

Token, Pattern & Lexemes

Interaction of scanner & parser



- Upon receiving a **"Get next token"** command from parser, the lexical analyzer reads the input character until it can identify the next token.
- Lexical analyzer also stripping out comments and white space in the form of blanks, tabs, and newline characters from the source program.

Why to separate lexical analysis & parsing?

1. Simplicity in design.
2. Improves compiler efficiency.
3. Enhance compiler portability.

Token, Pattern & Lexemes

Token

Sequence of character having a collective meaning is known as **token**.

Categories of Tokens:

1. Identifier
2. Keyword
3. Operator
4. Special symbol
5. Constant

Pattern

The set of rules called **pattern** associated with a token.

Example: "*non-empty sequence of digits*", "*letter followed by letters and digits*"

Lexemes

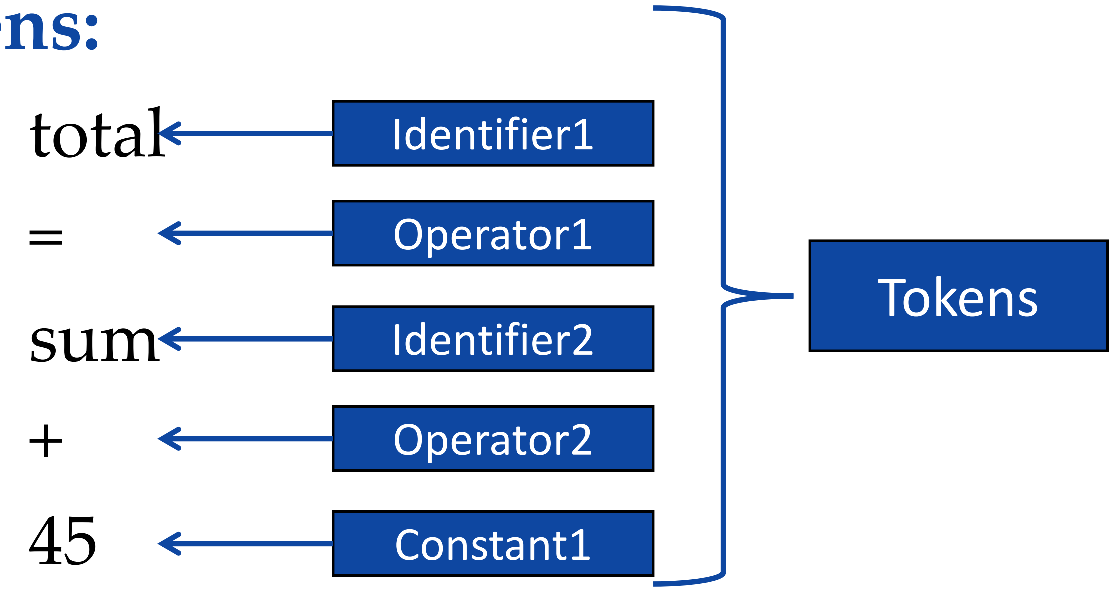
The sequence of character in a source program matched with a pattern for a token is called lexeme.

Example: Rate, DIET, count, Flag

Example: Token, Pattern & Lexemes

Example: total = sum + 45

Tokens:



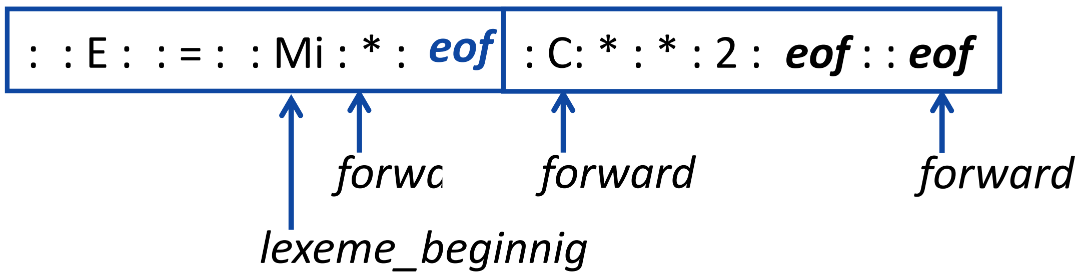
Lexemes

Lexemes of identifier: total, sum

Lexemes of operator: =, +

Lexemes of constant: 45

Sentinels




```
forward := forward + 1;
    if forward = eof then begin
        if forward at end of first half then begin
            reload second half;
            forward := forward + 1;
        end
        else if forward at the second half then begin
            reload first half;
            move forward to beginning of first half;
        end
        else terminate lexical analysis;
    end
```

Specification of tokens

Strings and languages

Term	Definition
Prefix of s	A string obtained by removing zero or more trailing symbol of string S. e.g., ban is prefix of banana.
Suffix of S	A string obtained by removing zero or more leading symbol of string S. e.g., nana is suffix of banana.
Sub string of S	A string obtained by **removing prefix and suffix** from S. e.g., nan is substring of banana
Proper prefix, suffix and substring of S	**Any nonempty string x that is respectively proper prefix, suffix or substring of S, such that s≠x.**
Subsequence of S	A string obtained by removing zero or more not necessarily contiguous symbol from S. e.g., baaa is subsequence of banana.

Exercise

- Write prefix, suffix, substring, proper prefix, proper suffix and subsequence of following string:

  String: Compiler

Operations on languages

Operation	**Definition**
Union of L and M Written L U 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^* *denotes "zero or more concatenation of" L.*
Positive closure of L ***Written L^+***	L^+ *denotes "one or more concatenation of" L.*

Regular Expression & Regular Definition

Regular expression

- A regular expression is a sequence of characters that define a pattern.

Notational shorthand's

1. One or more instances: +
2. Zero or more instances: *
3. Zero or one instances: ?
4. Alphabets: Σ

Rules to define regular expression

1. ∈ is a regular expression that denotes {∈}, the set containing empty string.
2. If a is a symbol in Σ then a is a regular expression, $L(a) = \{a\}$
3. Suppose $r$and s are regular expression denoting the languages $L(r)$ and $L(s)$. Then,
   a. $(\mathrm{r})|(\mathrm{s})$ is a regular expression denoting $\mathrm{L(r)\ U\ L(s)}$
   b. $(\mathrm{r})(\mathrm{s})$ is a regular expression denoting $\mathrm{L(r)L(s)}$
   c. $(\mathrm{r})^*$ is a regular expression denoting $\left(\mathrm{L(r)}\right)^*$
   d. (r) is a regular expression denoting $\mathrm{L((r))}$

The language denoted by regular expression is said to be a regular set.

Regular expression

- L = Zero or More Occurrences of a = a^*

*

$$\left.\begin{array}{l}\epsilon \\ a \\ aa \\ aaa \\ aaaa \\ aaaaa.....\end{array}\right\}$$ Infinite .....

Regular expression

a^+

- **L = One or More Occurrences of a =**

+

a
aa
aaa
aaaa
aaaaa.....

Infinite .....

Precedence and associativity of operators

Operator	Precedence	Associative
Kleene *	1	left
Concatenation	2	left
Union |	3	left

Regular expression examples

1. 0 or 1

   Strings: 0, 1 $\quad$ $\mathbf{R.E. = 0 \mid 1}$

2. 0 or 11 or 111

   Strings: 0, 11, 111 $\quad$ $\mathbf{R.E. = 0 \mid 11 \mid 111}$

3. String having zero or more a.

   Strings: ϵ, a, aa, aaa, aaaa ….. . $\quad$ $\mathbf{R.E. = a^{*}}$

4. String having one or more a.

   Strings: a, aa, aaa, aaaa ….. . $\quad$ $\mathbf{R.E. = a^{+}}$

5. Regular expression over $\Sigma = \{a, b, c\}$ that represent all string of length 3.

   Strings: abc, bca, bbb, cab, aba …. . $\quad$ $\mathbf{R.E. = (a|b|c)\ (a|b|c)\ (a|b|c)}$

6. All binary string

   Strings: 0, 11, 101, 10101, 1111 … $\quad$ $\mathbf{R.E. = (0 \mid 1)^{+}}$

Regular expression examples

7. 0 or more occurrence of either a or b or both

   $Strings: \epsilon, a, aa, abab, bab \ldots$ $R.E. = (a \mid b) *$

8. 1 or more occurrence of either a or b or both

   $Strings: a, aa, abab, bab, bbbaaa \ldots$ $R.E. = (a \mid b)^{+}$

9. Binary no. ends with 0

   $Strings:\ 0, 10, 100, 1010, 11110 \ldots$ $R.E. = (0 \mid 1)^{*}\ 0$

10. Binary no. ends with 1

    $Strings: 1, 101, 1001, 10101, \ldots$ $R.E. = (0 \mid 1) * 1$

11. Binary no. starts and ends with 1

    $Strings:\ 11, 101, 1001, 10101, \ldots$ $R.E. = 1\ (0 \mid 1) * 1$

12. String starts and ends with same character

    $Strings: 00, 101, aba, baab \ldots$ $R.E. = 1\ (0 \mid 1) * 1$ or $0\ (0 \mid 1) * 0$

    $a\ (a \mid b)^{*}\ a$ or $b\ (a \mid b)^{*}\ b$

Regular expression examples

13. All string of a and b starting with a

$Strings: a, ab, aab, abb...$ $R.E. = a(a \mid b)^*$

14. String of 0 and 1 ends with 00

$Strings: 00, 100, 000, 1000, 1100...$ $R.E. = (0 \mid 1) * 00$

15. String ends with abb

$Strings: abb, babb, ababb...$ $R.E. = (a \mid b) * abb$

16. String starts with 1 and ends with 0

$Strings: 10, 100, 110, 1000, 1100...$ $R.E. = 1(0 \mid 1) * 0$

17. All binary string with at least 3 characters and 3rd character should be zero

$Strings: 000, 100, 1100, 1001...$ $R.E. = (0|1)(0|1)0(0 \mid 1) *$

18. Language which consist of exactly two b's over the set $\Sigma = \{a, b\}$

$Strings: bb, bab, aabb, abba...$ $R.E. = a^* \, b \, a^* \, b \, a^*$

Regular expression examples

24. The language with $\Sigma = \{a, b, c\}$ where a should be multiple of 3

    $\boldsymbol{Strings: aaa, baaa, bacaba, aaaaaa..}$ $\boldsymbol{R.E. = ((b|c)^*a(b|c)^*a(b|c)^*a(b|c)^*)^*}$

25. Even no. of 0

    $\boldsymbol{Strings: 00, 0101, 0000, 100100....}$ $\boldsymbol{R.E. = (1^*01^*01^*)^*}$

26. String should have odd length

    $\boldsymbol{Strings: 0, 010, 110, 000, 10010....}$ $\boldsymbol{R.E. = (0|1)\ ((0|1)(0|1))^*}$

27. String should have even length

    $\boldsymbol{Strings: 00, 0101, 0000, 100100....}$ $\boldsymbol{R.E. = ((0|1)(0|1))^*}$

28. String start with 0 and has odd length

    $\boldsymbol{Strings: 0, 010, 010, 000, 00010....}$ $\boldsymbol{R.E. = (0)\ ((0|1)(0|1))^*}$

30. String start with 1 and has even length

    $\boldsymbol{Strings: 10, 1100, 1000, 100100....}$ $\boldsymbol{R.E. = 1(0|1)((0|1)(0|1))^*}$

31. All string begins or ends with 00 or 11

    $\boldsymbol{Strings: 00101, 10100, 110, 01011 ...}$ $\boldsymbol{R.E. = (00|11)(0\ |\ 1) * |\ (0|1)^*\ (00|11)}$

Regular expression examples

31. Language of all string containing both 11 and 00 as substring

$\boldsymbol{Strings}$: $\mathbf{0011, 1100, 100110, 010011 \ldots}$

$\boldsymbol{R.E.} = ((\mathbf{0}|\mathbf{1})^*\mathbf{00}(\mathbf{0}|\mathbf{1})^*\mathbf{11}(\mathbf{0}|\mathbf{1})^*) \ | \ ((\mathbf{0}|\mathbf{1})^*\mathbf{11}(\mathbf{0}|\mathbf{1})^*\mathbf{00}(\mathbf{0}|\mathbf{1})^*)$

32. String ending with 1 and not contain 00

$\boldsymbol{Strings}$: $\mathbf{011, 1101, 1011 \ldots.}$

$\boldsymbol{R.E.} = (\mathbf{1}|\mathbf{01})^+$

33. Language of C identifier

$\boldsymbol{Strings}$: $\boldsymbol{area, i, redious, grade1 \ldots.}$

$\boldsymbol{R.E.} = (\_ + \boldsymbol{L})(\_ + \boldsymbol{L} + \boldsymbol{D})^*$

$\boldsymbol{where\ L\ is\ Letter}$ & $\mathbf{D\ is\ digit}$

Regular definition

- A regular definition gives names to certain regular expressions and uses those names in other regular expressions.
- Regular definition is a sequence of definitions of the form:

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$\ldots\ldots$$
$$d_n \rightarrow rn$$

Where d_i is a **distinct name** & r_i is a **regular expression**.

- Example: Regular definition for identifier

letter → A|B|C|………..|Z|a|b|………..|z

digit → 0|1|…….|9|

id→ letter (letter | digit)*

Regular definition example

- Example: Unsigned Pascal numbers

  3

  5280

  39.37

  6.336E4

  1.894E-4

  2.56E+7

Regular Definition

digit → 0|1|…..|9

digits → digit digit*

optional_fraction → .digits | ϵ

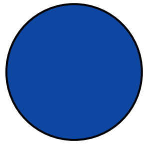
optional_exponent → (E(+|-|ϵ)digits)|ϵ

num → digits optional_fraction optional_exponent

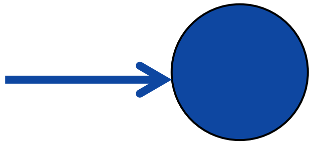
Transition Diagram

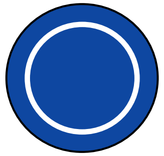
Transition Diagram

- A stylized flowchart is called transition diagram.

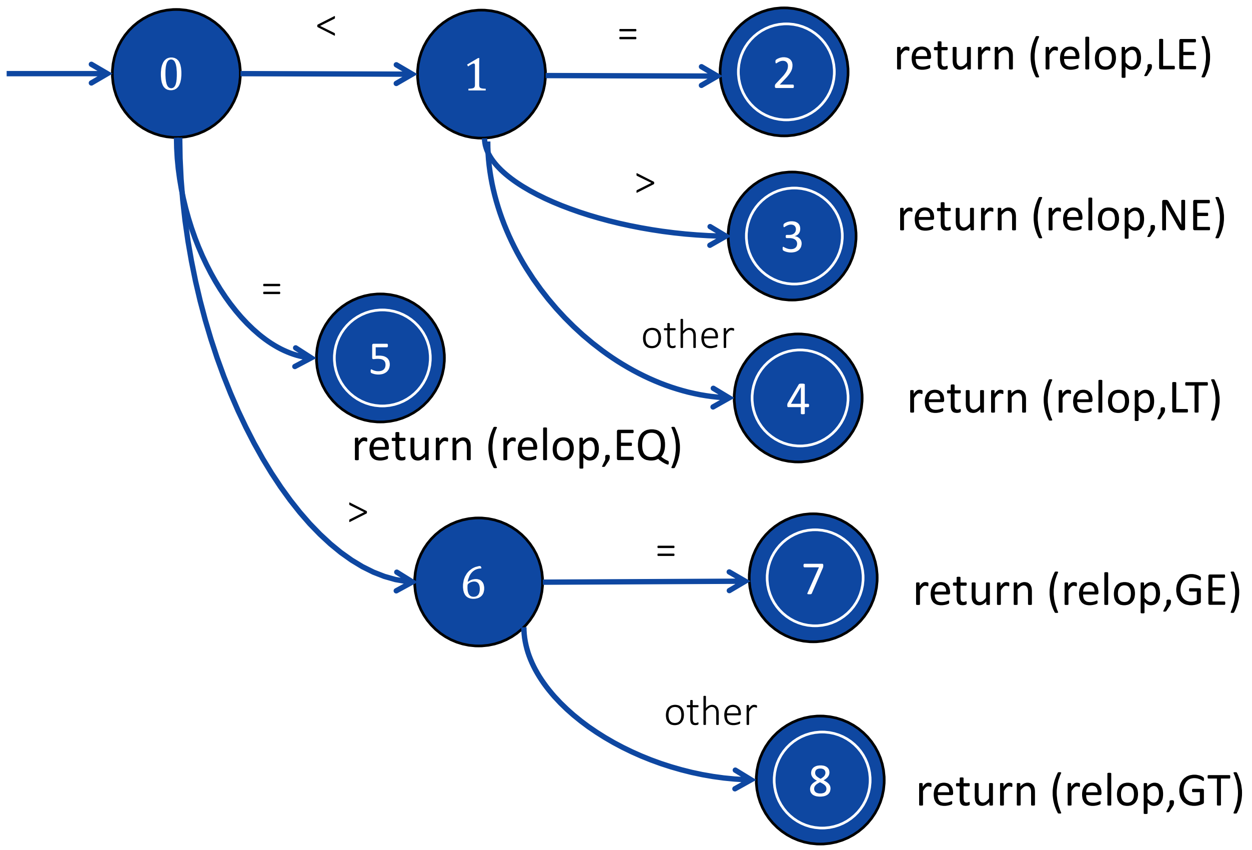

is a state


is a transition

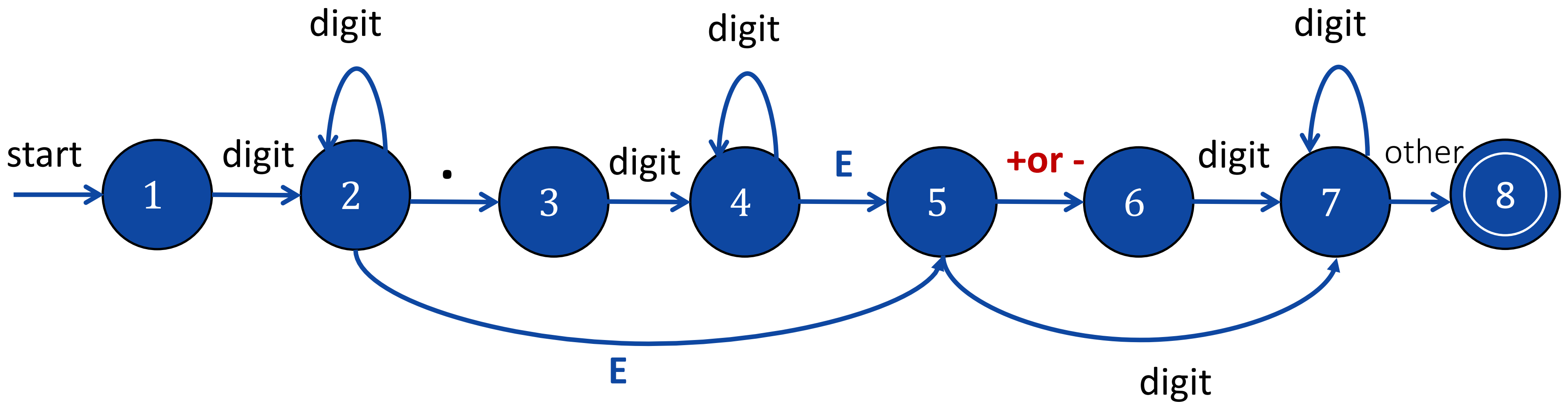

is a start state


is a final state

Transition Diagram : Relational operator



Transition diagram : Unsigned number



3

5280

39.37

1.894 E - 4

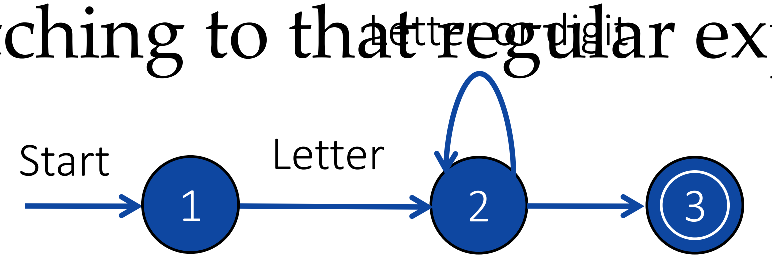
2.56 E + 7

45 E + 6

96 E 2

Hard coding and automatic generation lexical analyzers

- Lexical analysis is about identifying the pattern from the input.
- To recognize the pattern, transition diagram is constructed.
- It is known as hard coding lexical analyzer.
- Example: to represent identifier in ‘C’, the first character must be letter and other characters are either letter or digits.
- To recognize this pattern, hard coding lexical analyzer will work with a transition diagram.
- The automatic generation lexical analyzer takes special notation as input.
- For example, lex compiler tool will take regular expression as input and finds out the pattern matching to that regular expression.



Finite Automata

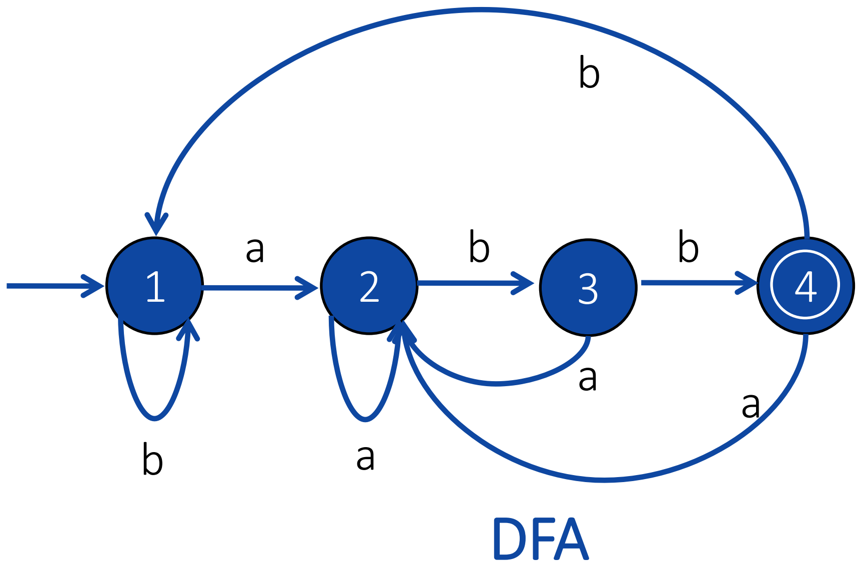
- Finite Automata are recognizers.
  - FA simply say "Yes" or "No" about each possible input string.
- Finite Automata is a mathematical model consist of:
  1. Set of states $\boldsymbol{S}$
  2. Set of input symbol $\boldsymbol{\Sigma}$
  3. A transition function ***move***
  4. Initial state $\boldsymbol{S_0}$
  5. Final states or accepting states **F**

Types of finite automata

- Types of finite automata are:

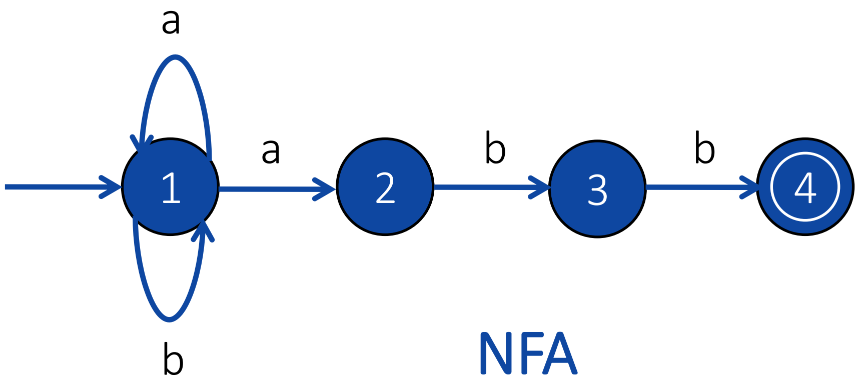
DFA

- Deterministic finite automata (DFA): have for each state exactly one edge leaving out for each symbol.



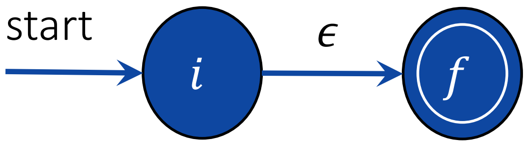
NFA

- Nondeterministic finite automata (NFA): There are no restrictions on the edges leaving a state. There can be several with the same symbol as label and some edges can be labeled with ϵ.

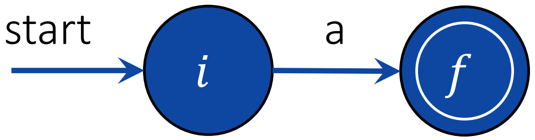


Regular expression to NFA using Thompson's rule

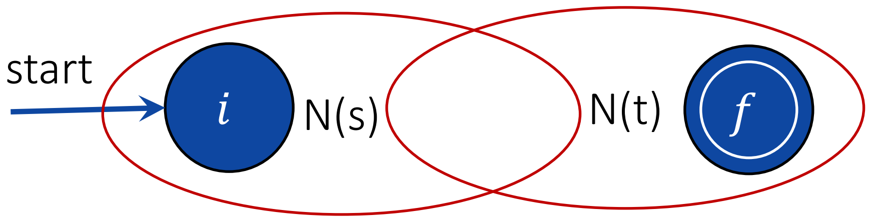
1. For $\in$, construct the NFA



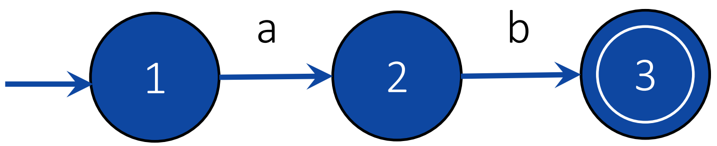
2. For a in Σ, construct the NFA



3. For regular expression st

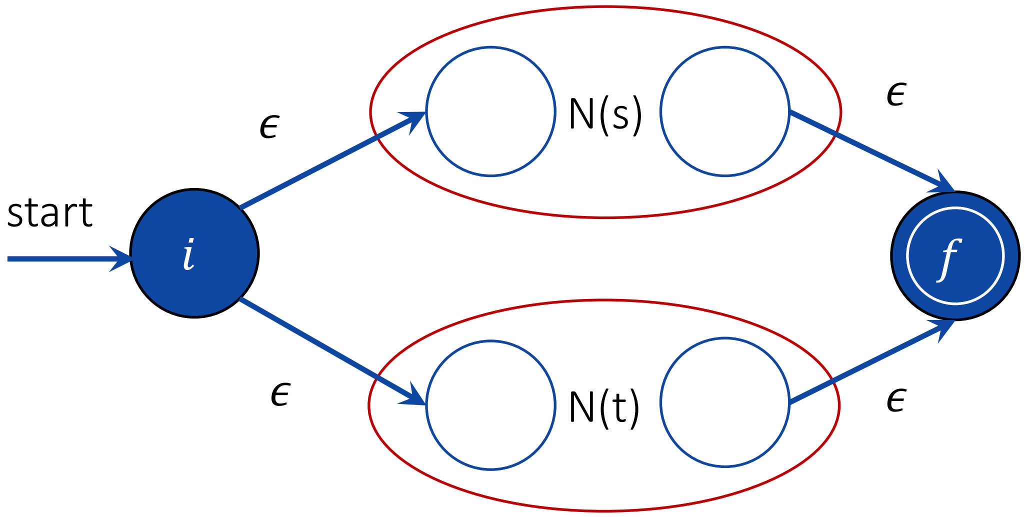


Ex: ab

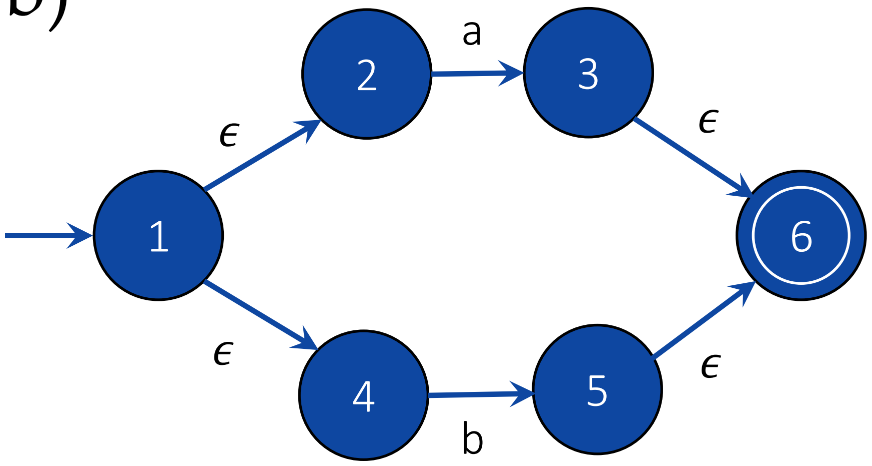


Regular expression to NFA using Thompson's rule

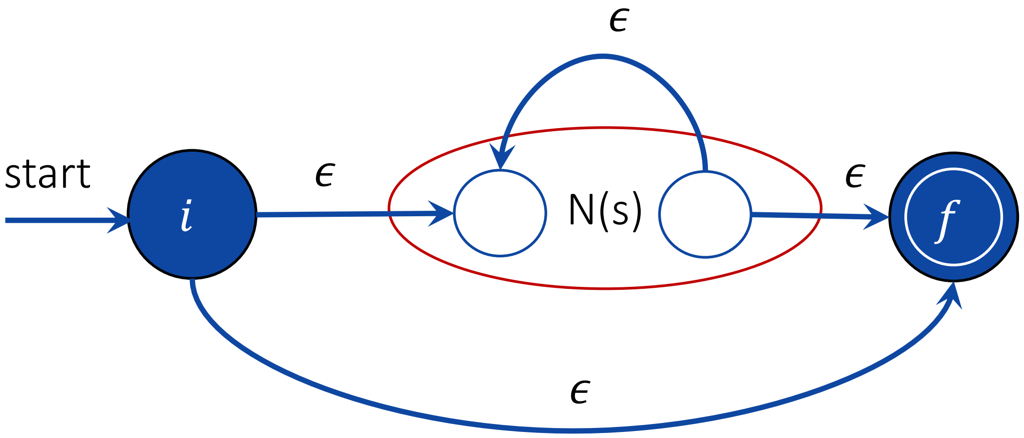
4. For regular expression $s|t$



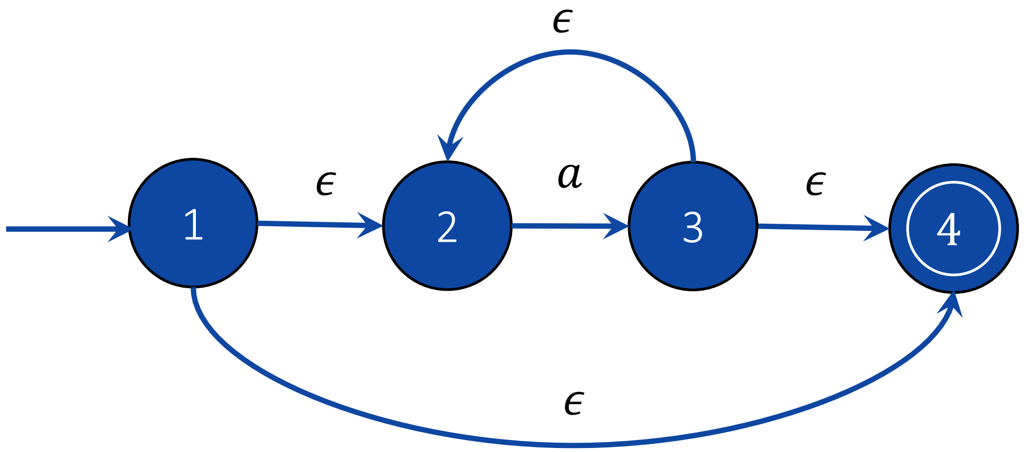
Ex: (a|b)



5. For regular expression $s*$

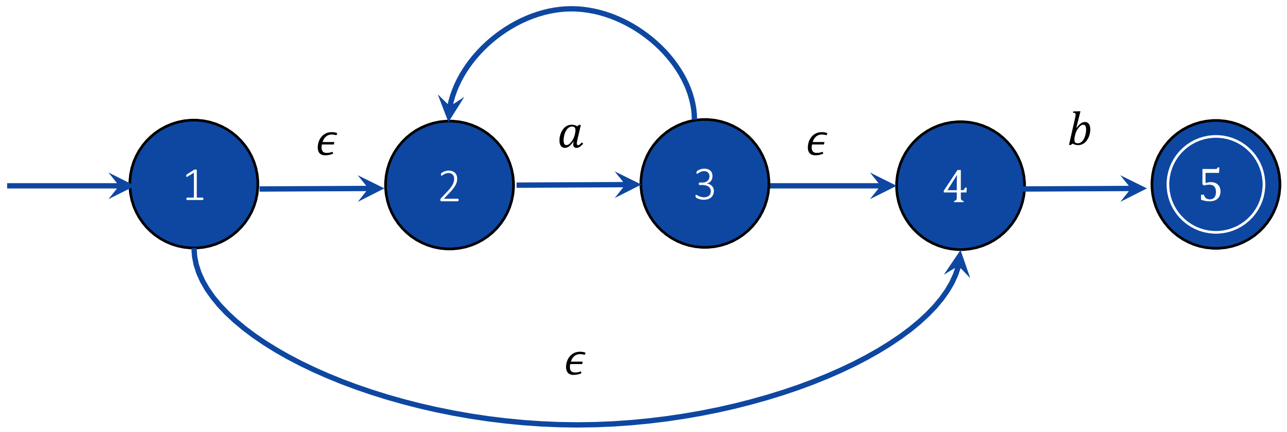


Ex: a*

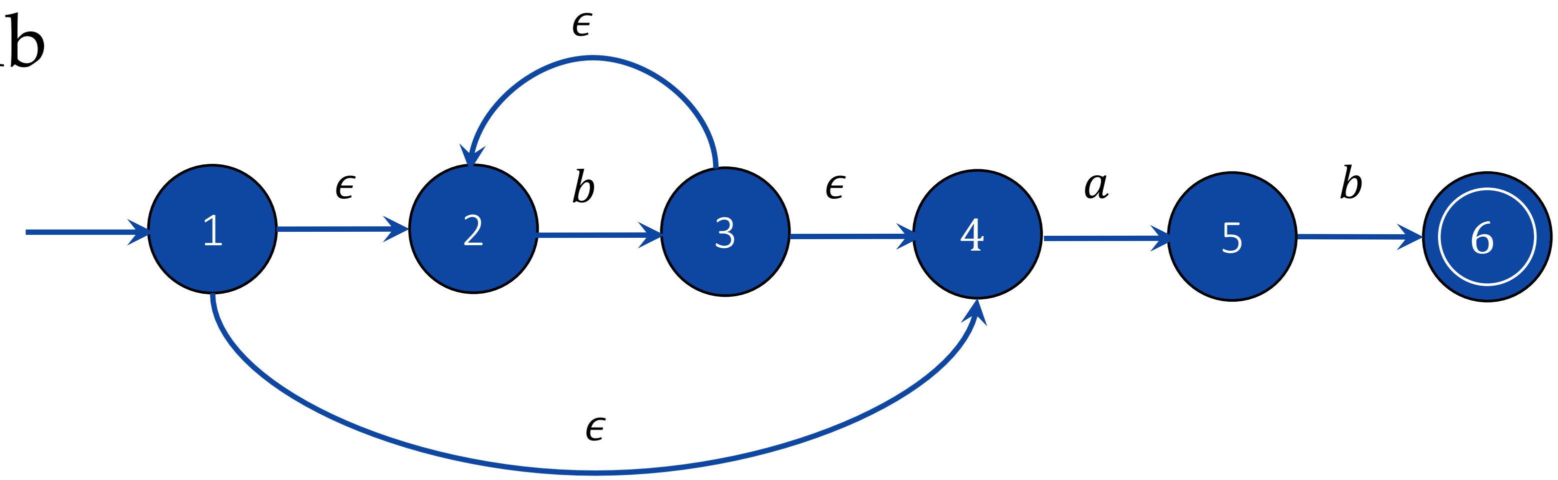


Regular expression to NFA using Thompson's rule

- a*b



- b*ab



Exercise

Convert following regular expression to NFA:

1. abba
2. bb(a)*
3. (a|b)*
4. a* | b*
5. a(a)*ab
6. aa*+ bb*
7. (a+b)*abb
8. 10(0+1)*1
9. (a+b)*a(a+b)
10. (0+1)*010(0+1)*
11. (010+00)*(10)*
12. 100(1)*00(0+1)*

Conversion from NFA to DFA using subset construction method

Subset construction algorithm

Input: An NFA N.

Output: A DFA D accepting the same language.

Method: Algorithm construct a transition table $Dtran$ for D. We use the following operation:

OPERATION	DESCRIPTION
$\epsilon - closure(s)$	Set of NFA states reachable from NFA state s on $\epsilon-$ transition alone.
$\epsilon - closure(T)$	Set of NFA states reachable from some NFA state s in T on $\epsilon-$ transition alone.
$\boldsymbol{Move}\ (T, a)$	Set of NFA states to which there is a transition on input symbol a from some NFA state s in T.

Subset construction algorithm

initially $\in -closure(s_0)$ be the only state in $Dstates$ and it is unmarked;

while there is unmarked states T in $Dstates$ ***do begin***

mark T;

for each input symbol a ***do begin***

$U = \epsilon - closure\big(move(T, a)\big)$;

if U is not in $Dstates$ ***then***

add U as unmarked state to $Dstates$;

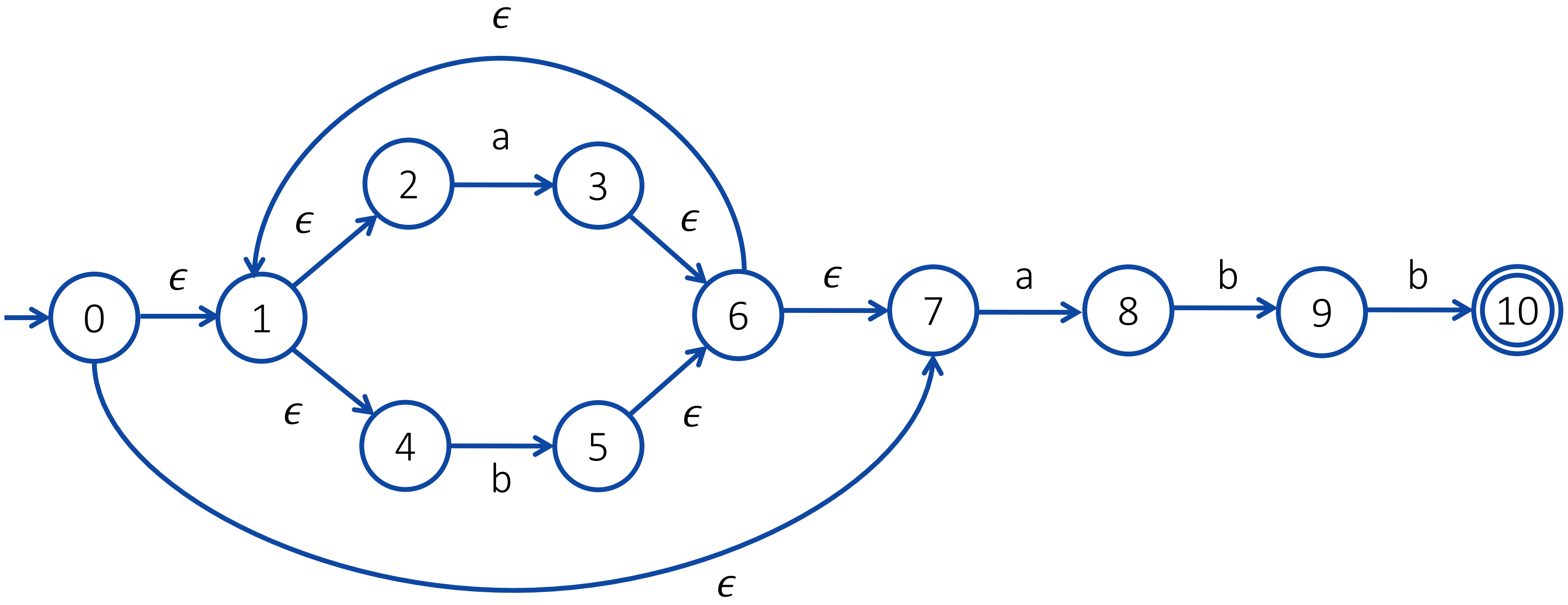
$Dtran[T, a] = U$

end

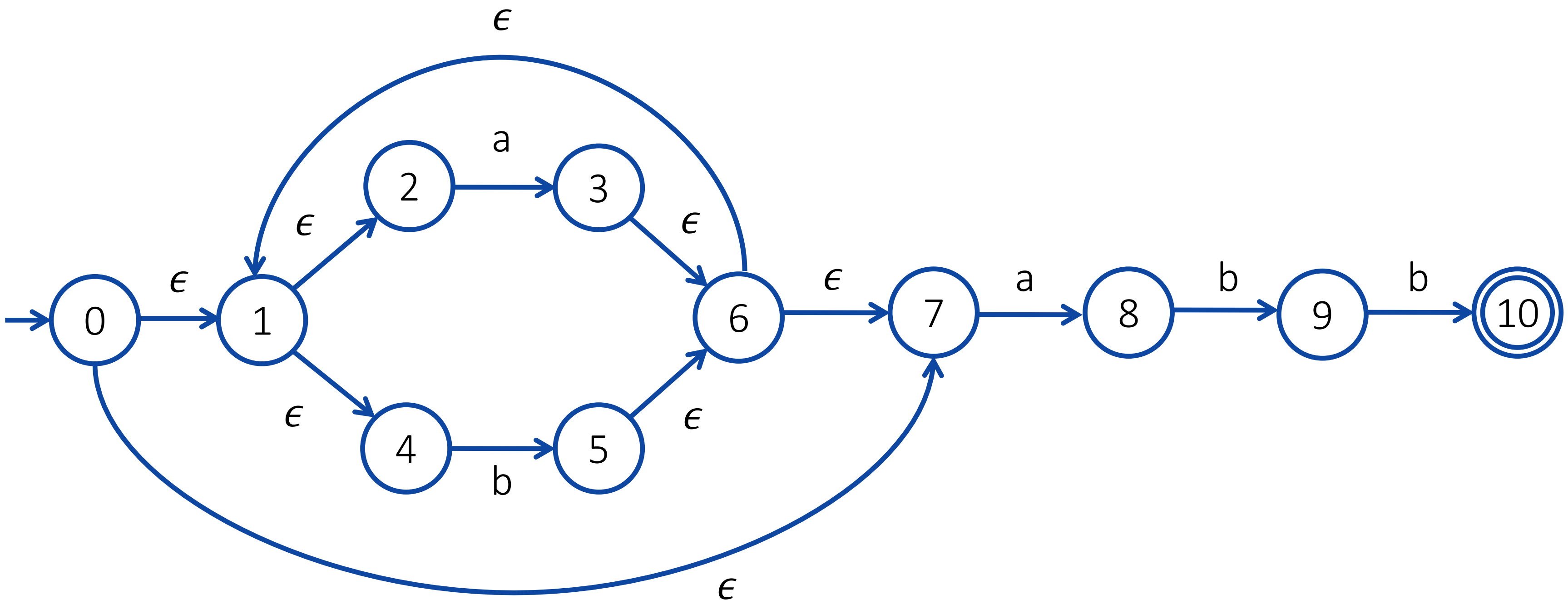
end

Conversion from NFA to DFA

$(a|b)^*abb$



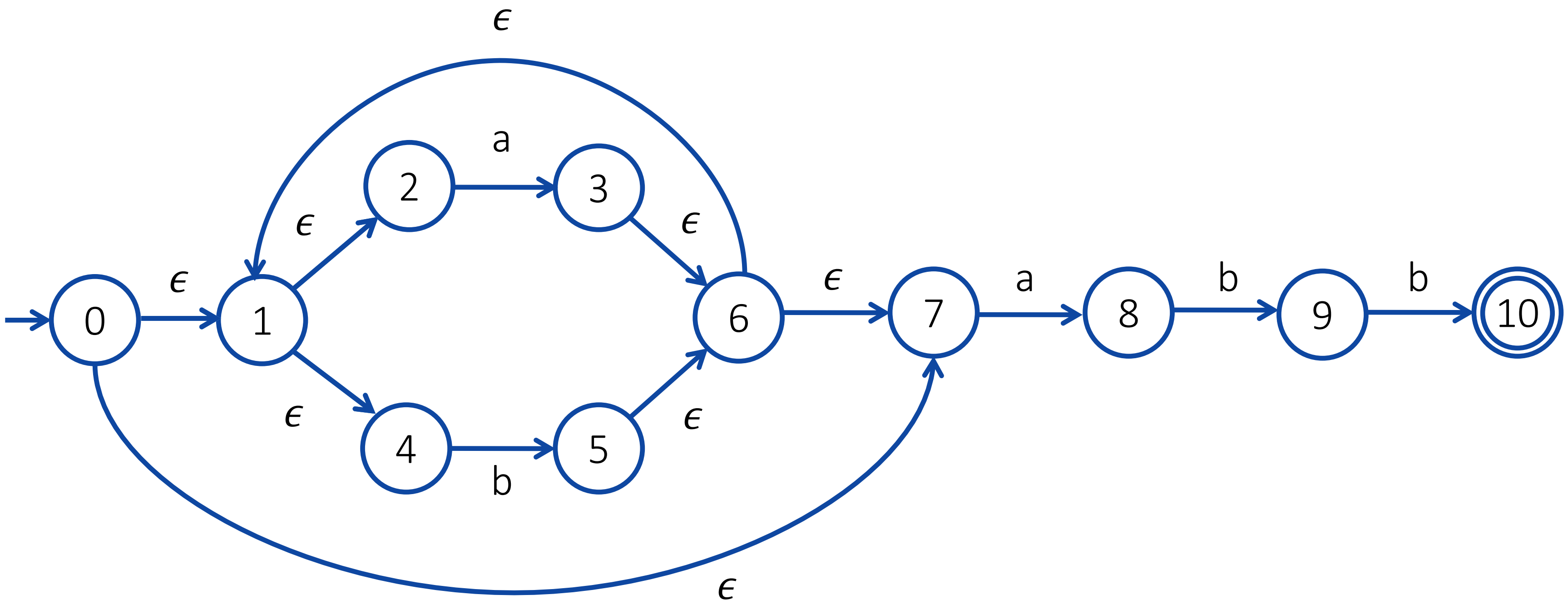
Conversion from NFA to DFA



ϵ- Closure(0)=

= {0,1,2,4,7} **---- A**

Conversion from NFA to DFA



A= {0, 1, 2, 4, 7}

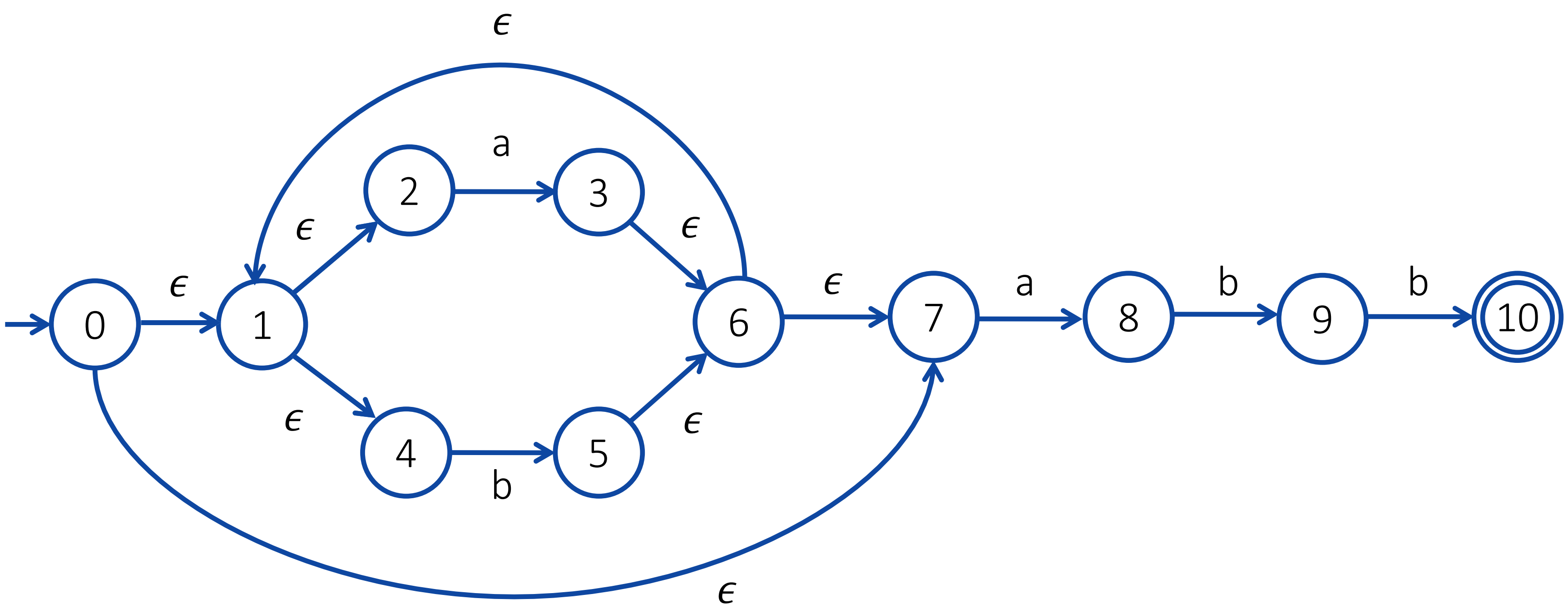
Move(A,a) = {3,8}

ϵ- Closure(Move(A,a))

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

States	a	b
A = {0,1,2,4,7}		
B = {1,2,3,4,6,7,8}		

Conversion from NFA to DFA



States	a	b
A = {0,1,2,4,7}	B	
B = {1,2,3,4,6,7,8}		
C = {1,2,4,5,6,7}		

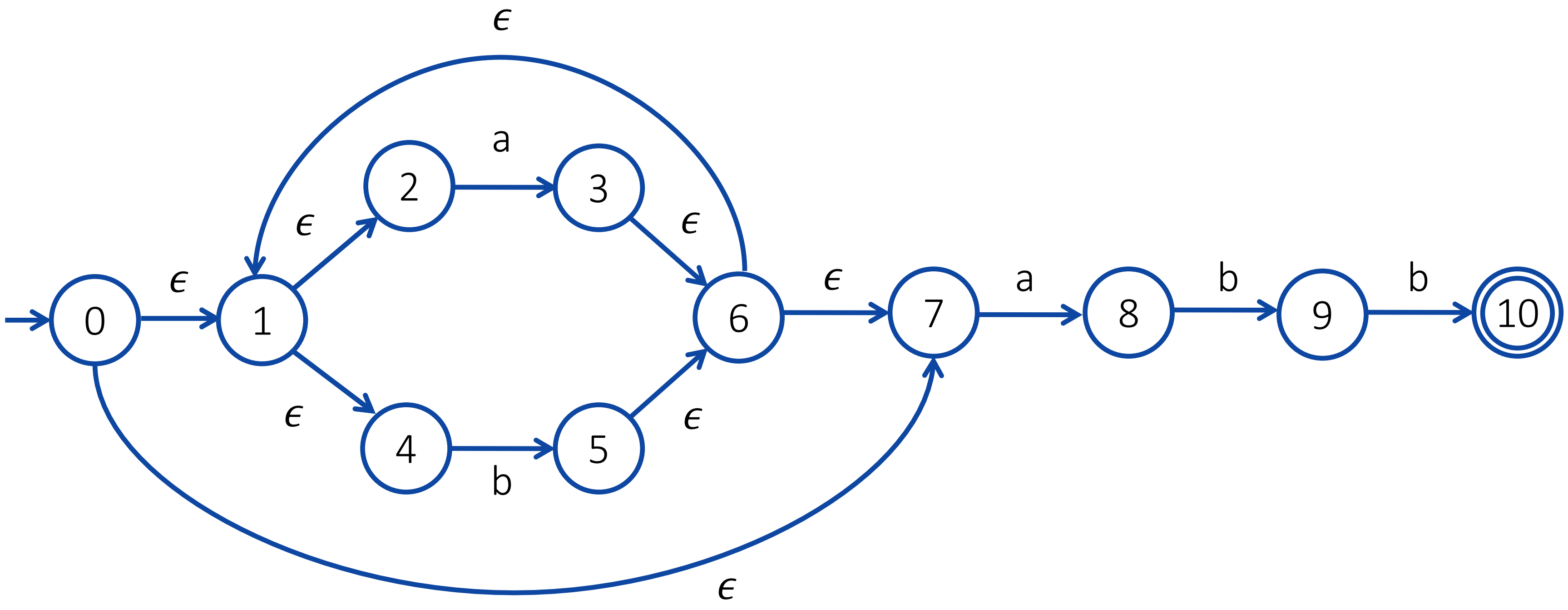
A= {0, 1, 2, 4, 7}

Move(A,b) =

ϵ- Closure(Move(A,b)) =

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

Conversion from NFA to DFA



States	a	b
A = {0,1,2,4,7}	**B**	**C**
B = {1,2,3,4,6,7,8}		
C = {1,2,4,5,6,7}		

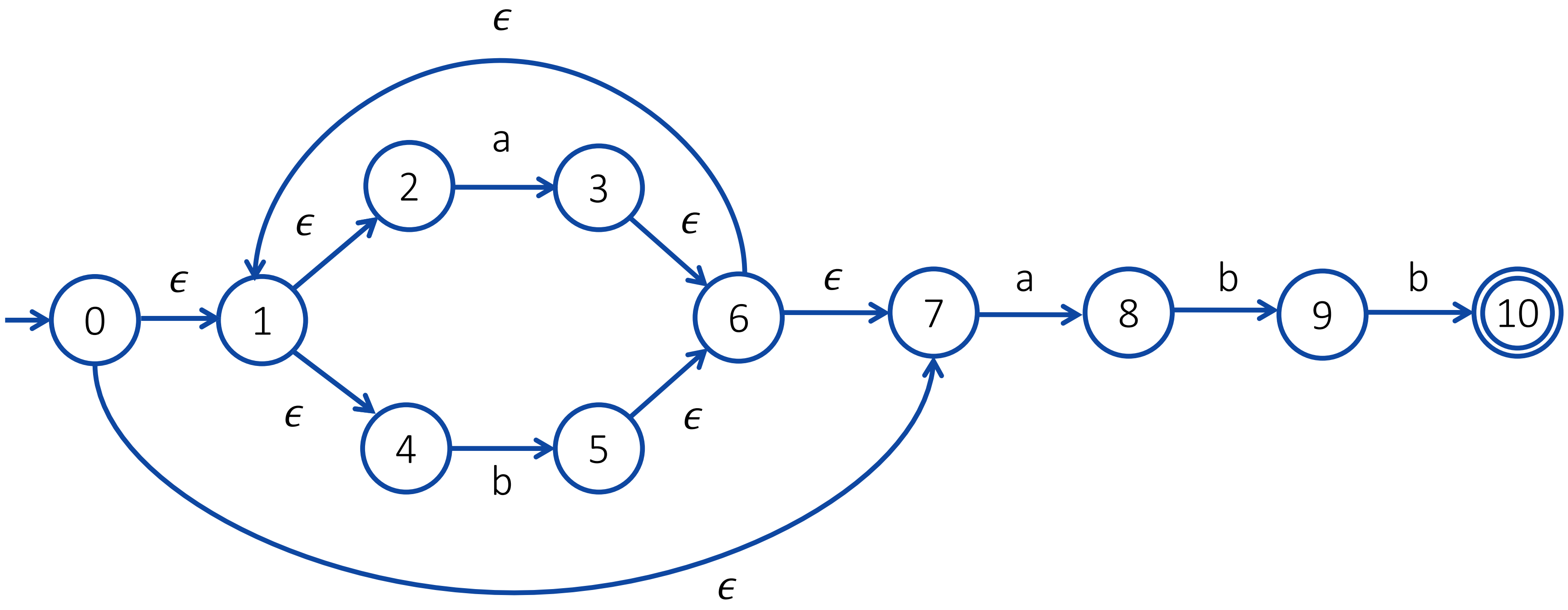
B = {1, 2, 3, 4, 6, 7, 8}

Move(B,a) = {3,8}

ϵ- Closure(Move(B,a))

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

Conversion from NFA to DFA



States	a	b
A = {0,1,2,4,7}	B	C
B = {1,2,3,4,6,7,8}	B	
C = {1,2,4,5,6,7}		
D = {1,2,4,5,6,7,9}		

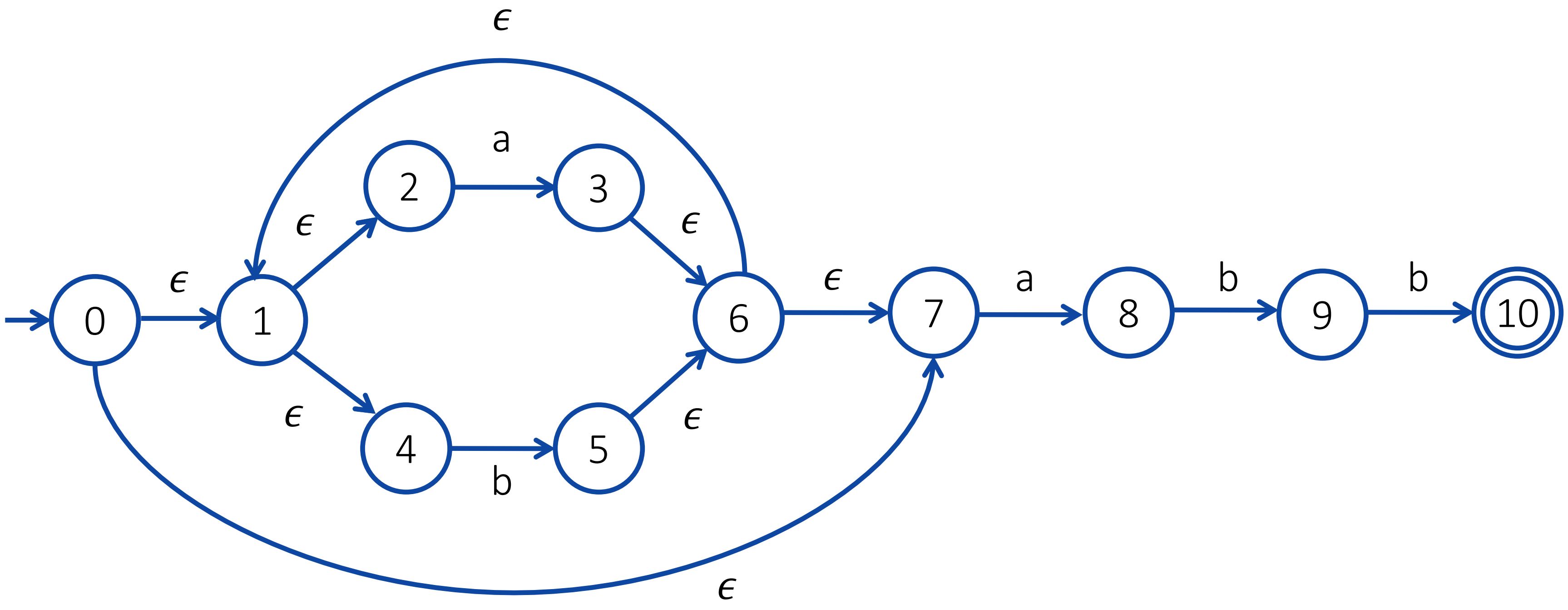
B= {1, 2, 3, 4, 6, 7, 8}

Move(B,b) = {5,9}

ϵ- Closure(Move(B,b))

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

Conversion from NFA to DFA



States	a	b
A = {0,1,2,4,7}	**B**	**C**
B = {1,2,3,4,6,7,8}	**B**	**D**
C = {1,2,4,5,6,7}		
D = {1,2,4,5,6,7,9}		

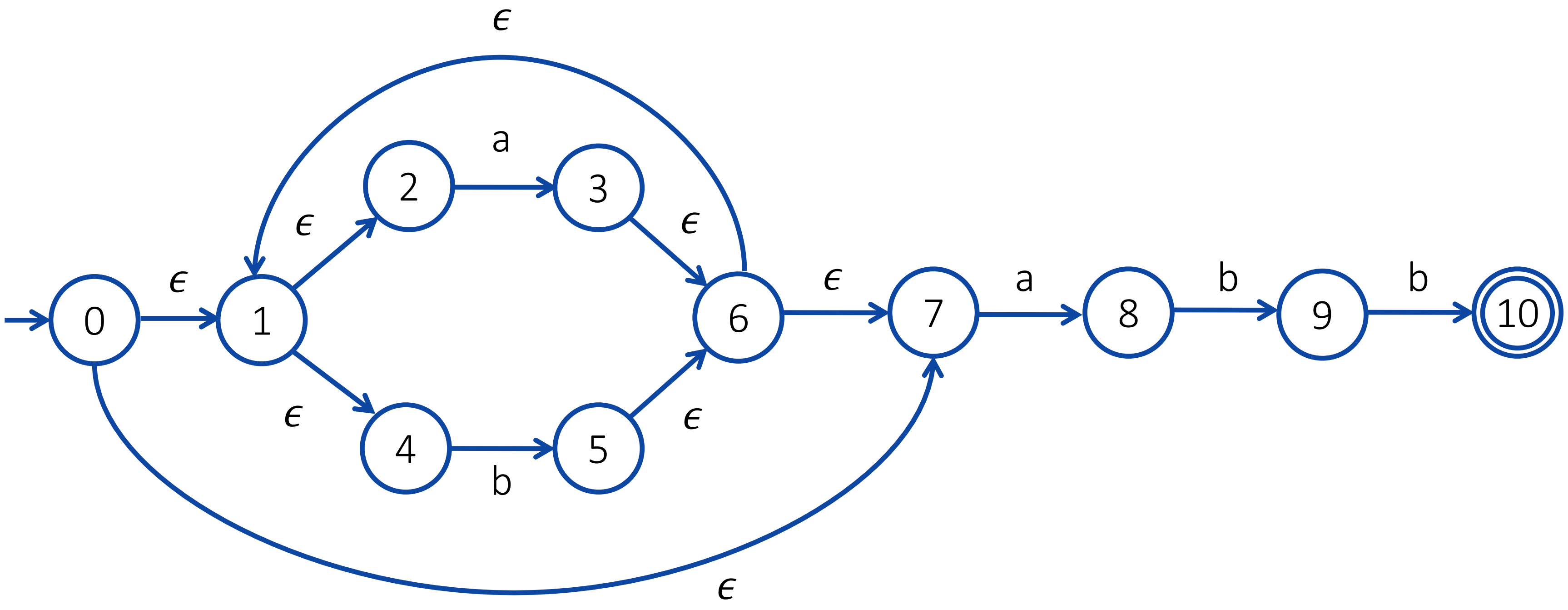
C= {1, 2, 4, 5, 6 ,7}

Move(C,a) = {3,8}

ϵ- Closure(Move(C,a))

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

Conversion from NFA to DFA



States	a	b
A = {0,1,2,4,7}	B	C
B = {1,2,3,4,6,7,8}	B	D
C = {1,2,4,5,6,7}	B	
D = {1,2,4,5,6,7,9}		

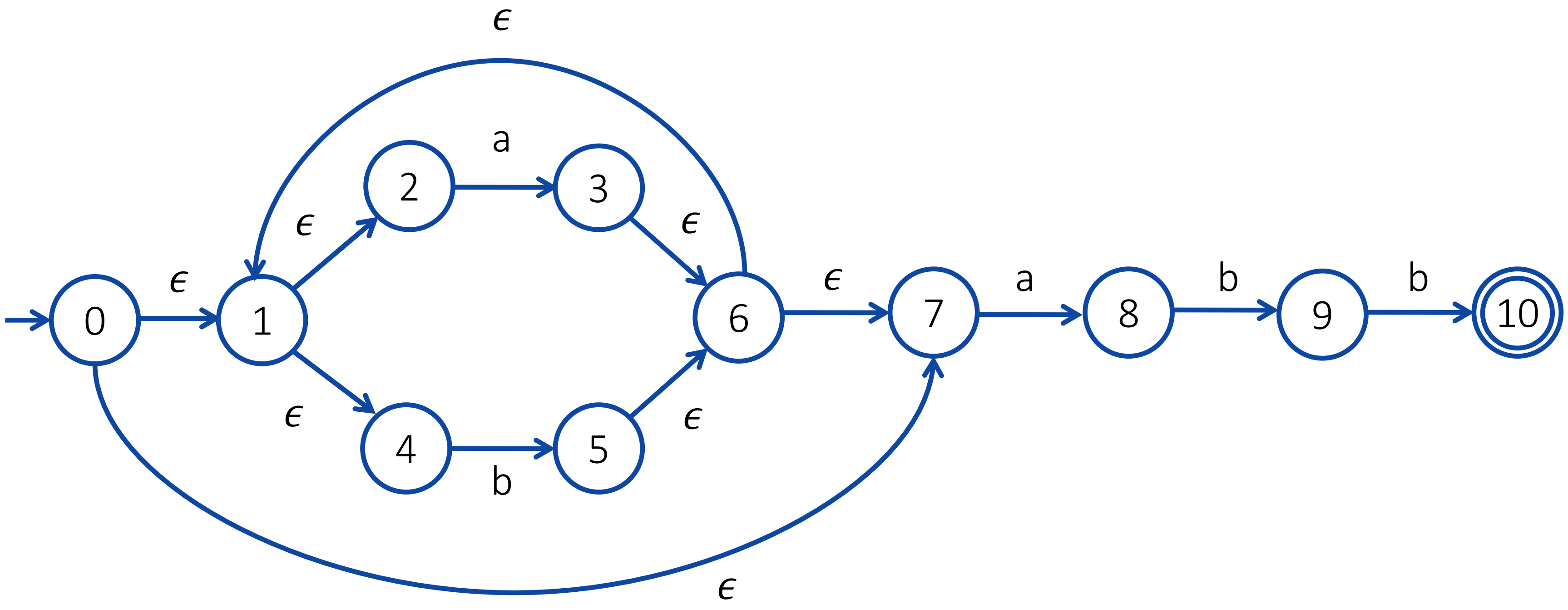
C= {1, 2, 4, 5, 6, 7}

Move(C,b) =

ϵ- Closure(Move(C,b))=

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

Conversion from NFA to DFA



States	a	b
A = {0,1,2,4,7}	B	C
B = {1,2,3,4,6,7,8}	B	D
C = {1,2,4,5,6,7}	B	C
D = {1,2,4,5,6,7,9}		

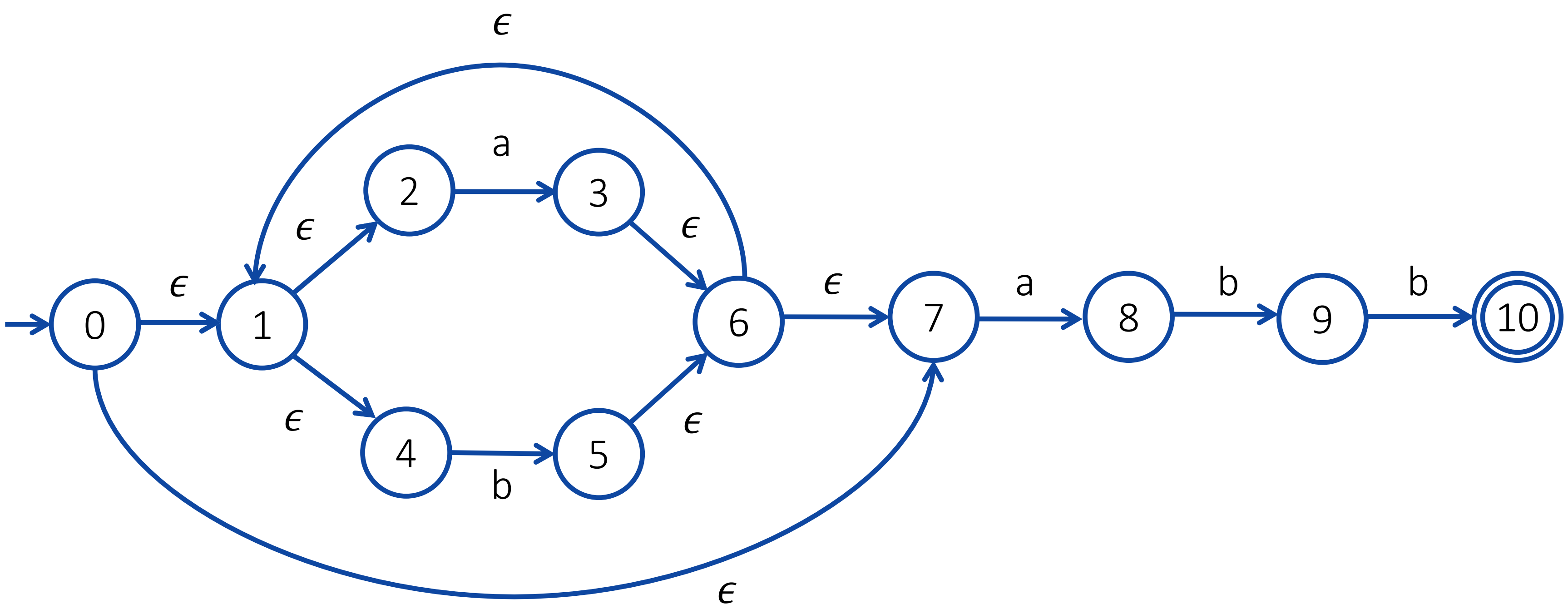
D= {1, 2, 4, 5, 6, 7, 9}

Move(D,a) = {3,8}

ϵ- Closure(Move(D,a))

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

Conversion from NFA to DFA



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

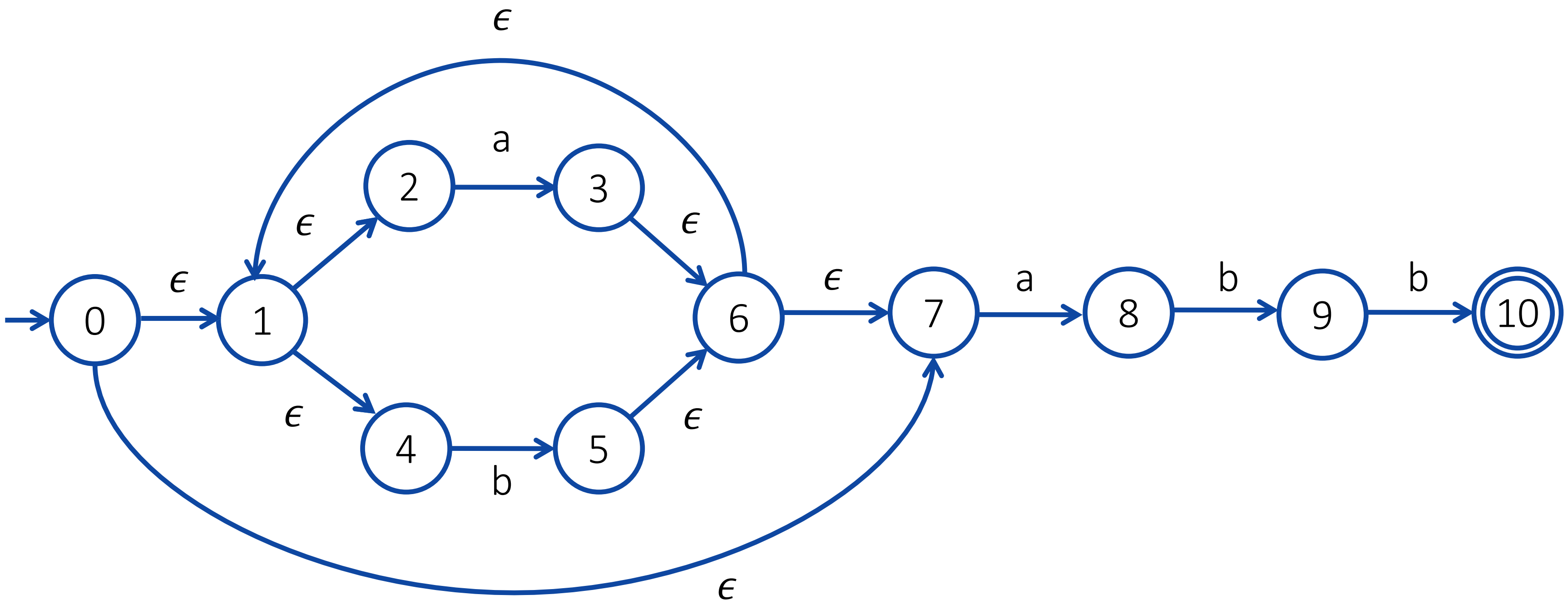
Move(D,b)= {5,10}

ϵ- Closure(Move(D,b))

= {1,2,4,5,6,7,10} ---- E

States	a	b
A = {0,1,2,4,7}	B	C
B = {1,2,3,4,6,7,8}	B	D
C = {1,2,4,5,6,7}	B	C
D = {1,2,4,5,6,7,9}	B	
E = {1,2,4,5,6,7,10}		

Conversion from NFA to DFA



States	a	b
A = {0,1,2,4,7}	B	C
B = {1,2,3,4,6,7,8}	B	D
C = {1,2,4,5,6,7}	B	C
D = {1,2,4,5,6,7,9}	B	E
E = {1,2,4,5,6,7,10}		

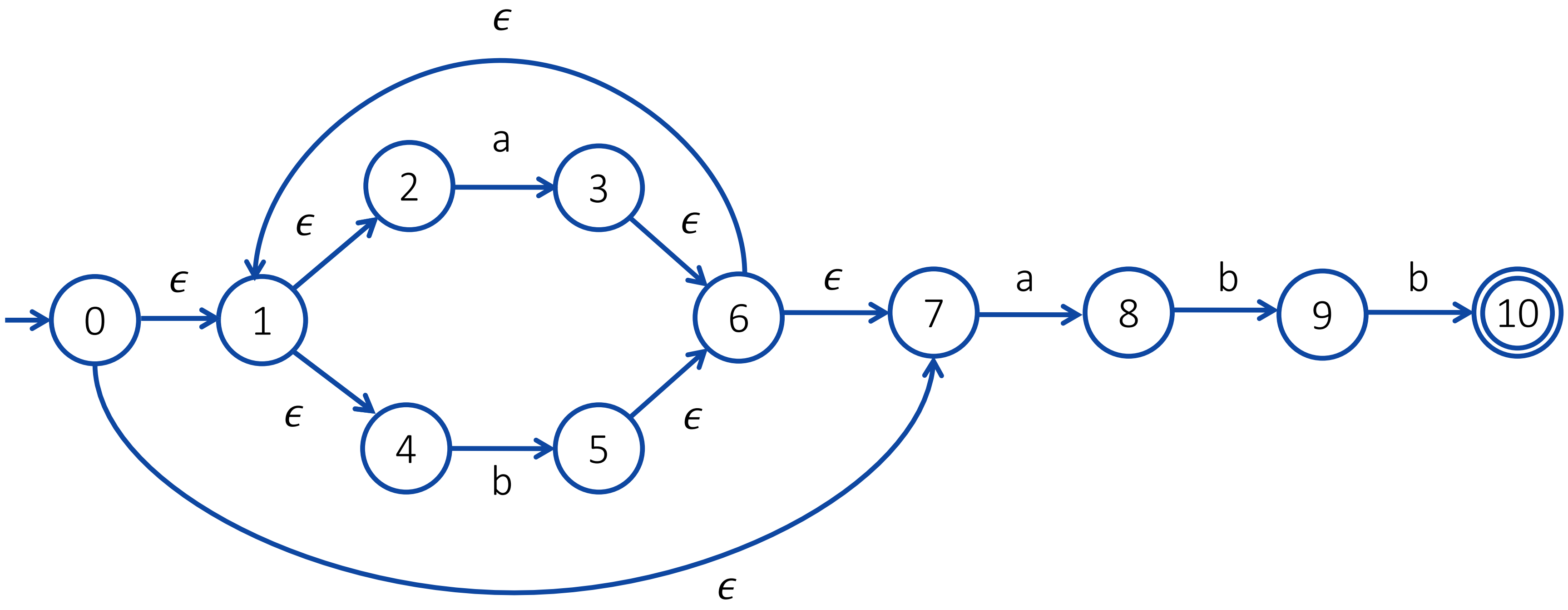
E= {1, 2, 4, 5, 6, 7, 10}

Move(E,a) = {3,8}

ϵ- Closure(Move(E,a))

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

Conversion from NFA to DFA



States	a	b
A = {0,1,2,4,7}	B	C
B = {1,2,3,4,6,7,8}	B	D
C = {1,2,4,5,6,7}	B	C
D = {1,2,4,5,6,7,9}	B	E
E = {1,2,4,5,6,7,10}	B	

E= {1, 2, 4, 5, 6, 7, 10}

Move(E,b)=

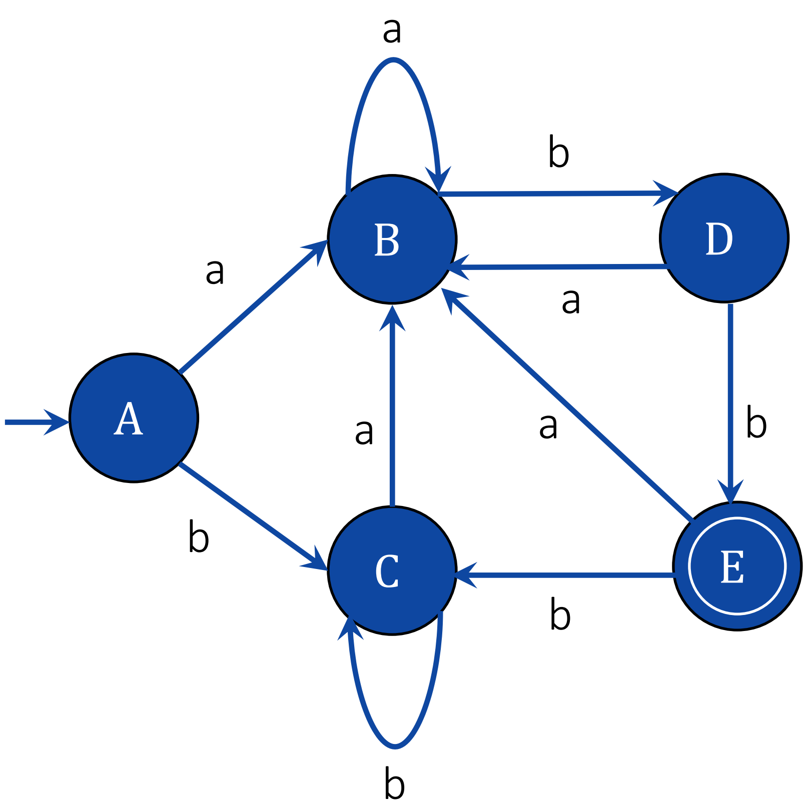
ϵ- Closure(Move(E,b))=

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

Conversion from NFA to DFA

States	a	b
A = {0,1,2,4,7}	B	C
B = {1,2,3,4,6,7,8}	B	D
C = {1,2,4,5,6,7}	B	C
D = {1,2,4,5,6,7,9}	B	E
E = {1,2,4,5,6,7,10}	B	C

Transition Table



DFA

Note:

- Accepting state in NFA is 10
- 10 is element of E
- So, E is acceptance state in DFA

Exercise

- Convert following regular expression to DFA using subset construction method:
  1. $(a+b)^*a(a+b)$
  2. $(a+b)^*ab^*a$

DFA optimization

1. Construct an initial partition Π of the set of states with two groups: the accepting states F and the non-accepting states $S - F$.
2. Apply the repartition procedure to Π to construct a new partition Πnew.
3. If $\Pi new = \Pi$, let $\Pi final = \Pi$ and continue with step (4). Otherwise, repeat step (2) with $\Pi = \Pi new$.

for each group G of Π **do begin**

partition G into subgroups such that two states s and t
of G are in the same subgroup if and only if for all
input symbols a, states s and t have transitions on a
to states in the same group of Π.

replace G in Πnew by the set of all subgroups formed.

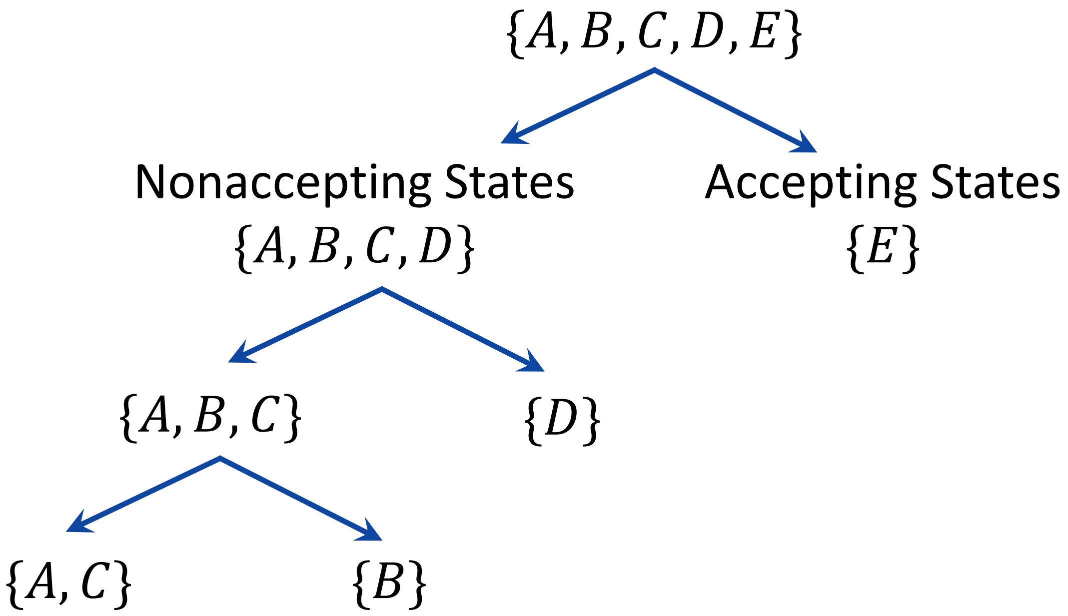
end

DFA optimization

4. Choose one state in each group of the partition $\Pi final$ as the representative for that group. The representatives will be the states of M'. Let s be a representative state, and suppose on input a there is a transition of M from s to t. Let r be the representative of t's group. Then M' has a transition from s to r on a. Let the start state of M' be the representative of the group containing start state s_0 of M, and let the accepting states of M' be the representatives that are in F.

5. If M' has a dead state d, then remove d from M'. Also remove any state not reachable from the start state.

DFA optimization



- Now no more splitting is possible.
- If we chose A as the representative for group (AC), then we obtain reduced transition table

States	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

States	a	b
A	B	A
B	B	D
D	B	E
E	B	A

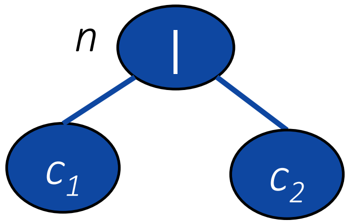
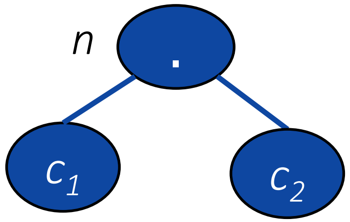
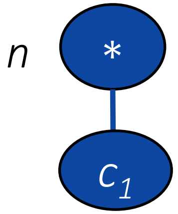
Optimized Transition Table

Conversion from regular expression to DFA

Rules to compute nullable, firstpos, lastpos

- nullable(n)
  - The subtree at node *n* generates languages including the empty string.
- firstpos(n)
  - The set of positions that can match the first symbol of a string generated by the subtree at node *n*.
- lastpos(n)
  - The set of positions that can match the last symbol of a string generated be the subtree at node *n*.
- followpos(i)
  - The set of positions that can follow position *i* in the tree.

Rules to compute nullable, firstpos, lastpos

<table>
<tr><th>Node n</th><th>nullable(n)</th><th>firstpos(n)</th><th>lastpos(n)</th></tr>
<tr><td>A leaf labeled by ε</td><td>true</td><td>∅</td><td>∅</td></tr>
<tr><td>A leaf with position i</td><td>false</td><td>{i}</td><td>{i}</td></tr>
<tr><td></td><td>nullable(c_1)
or
nullable(c_2)</td><td>firstpos(c_1)
∪
firstpos(c_2)</td><td>lastpos(c_1)
∪
lastpos(c_2)</td></tr>
<tr><td></td><td>nullable(c_1)
and
nullable(c_2)</td><td>if (nullable(c_1))
then firstpos(c_1) ∪
firstpos(c_2)
else firstpos(c_1)</td><td>if (nullable(c_2))
then lastpos(c_1) ∪
lastpos(c_2)
else lastpos(c_2)</td></tr>
<tr><td></td><td>true</td><td>firstpos(c_1)</td><td>lastpos(c_1)</td></tr>
</table>

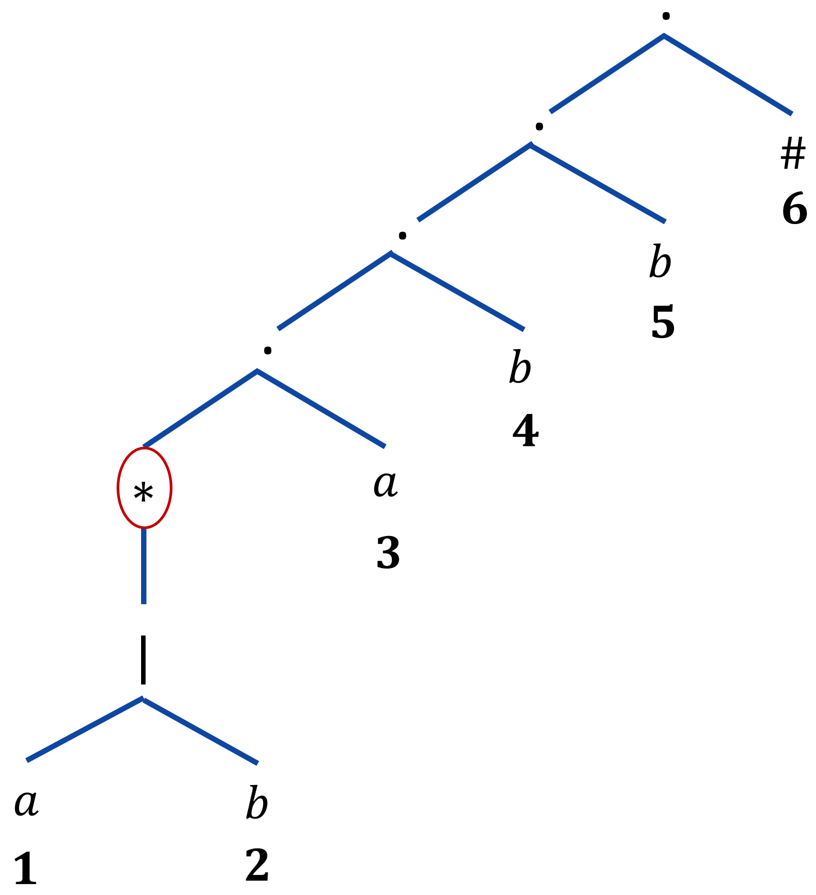
Rules to compute followpos

1. If n is **concatenation** node with left child c1 and right child c2 and ***i*** is a position in lastpos(c1), then all position in firstpos(c2) are in followpos(i)

2. If n is * node and ***i*** is position in lastpos(n), then all position in firstpos(n) are in followpos(i)

Conversion from regular expression to DFA

(a|b)* abb #



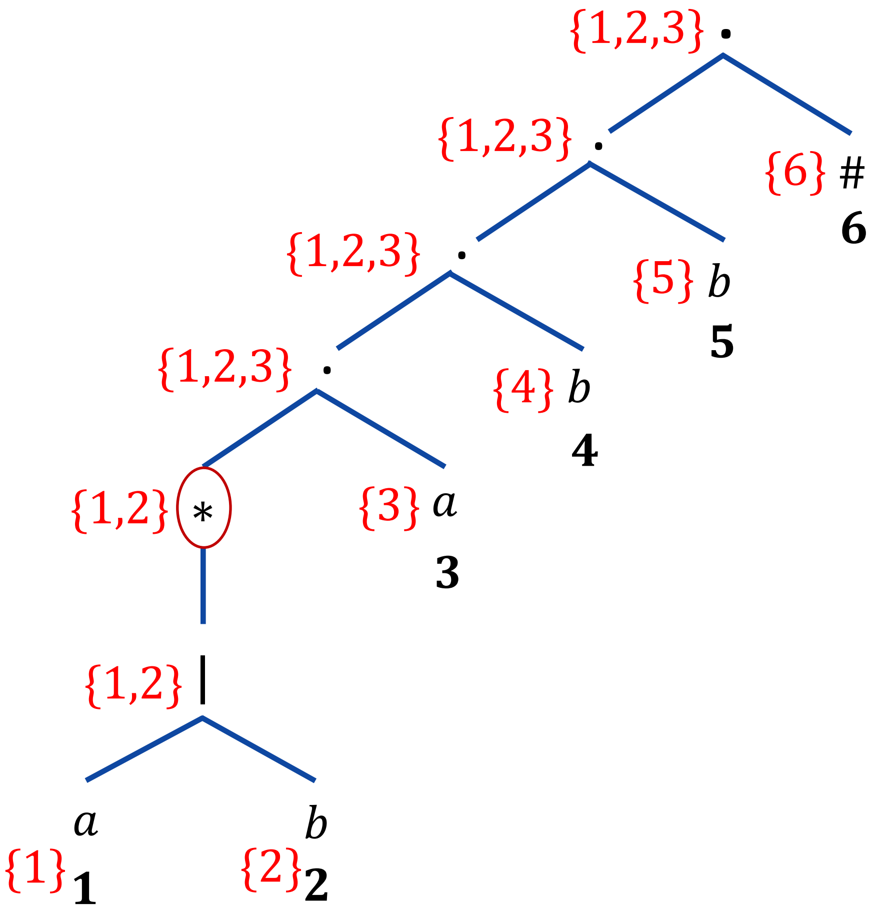
Step 1: Construct Syntax Tree

Step 2: Nullable node

Here, * is only nullable node

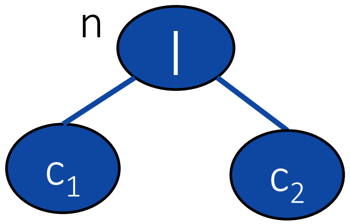
Conversion from regular expression to DFA

Step 3: Calculate firstpos

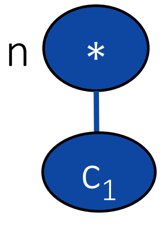


Firstpos —

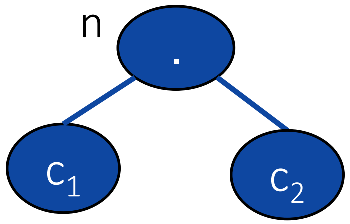
A leaf with position $\boldsymbol{i} = \{\boldsymbol{i}\}$



$firstpos(c_1) \cup firstpos(c_2)$



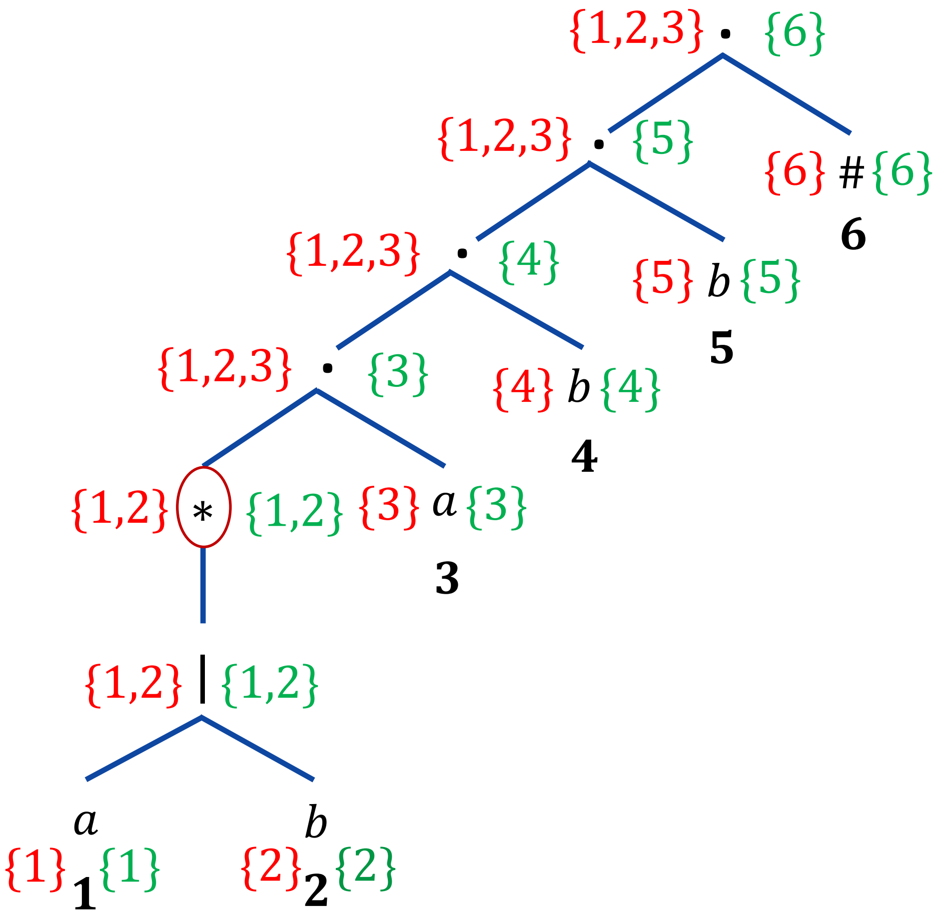
$firstpos(c_1)$



if $(nullable(c_1))$
then $firstpos(c_1) \cup firstpos(c_2)$
else $firstpos(c_1)$

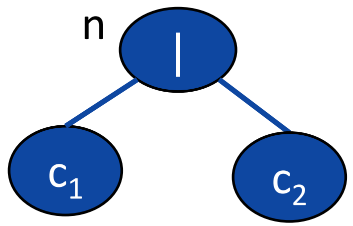
Conversion from regular expression to DFA

Step 3: Calculate lastpos

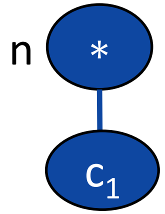


Lastpos —

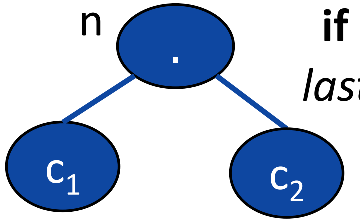
A leaf with position $i = \{i\}$



$lastpos(c_1) \cup lastpos(c_2)$



$lastpos(c_1)$

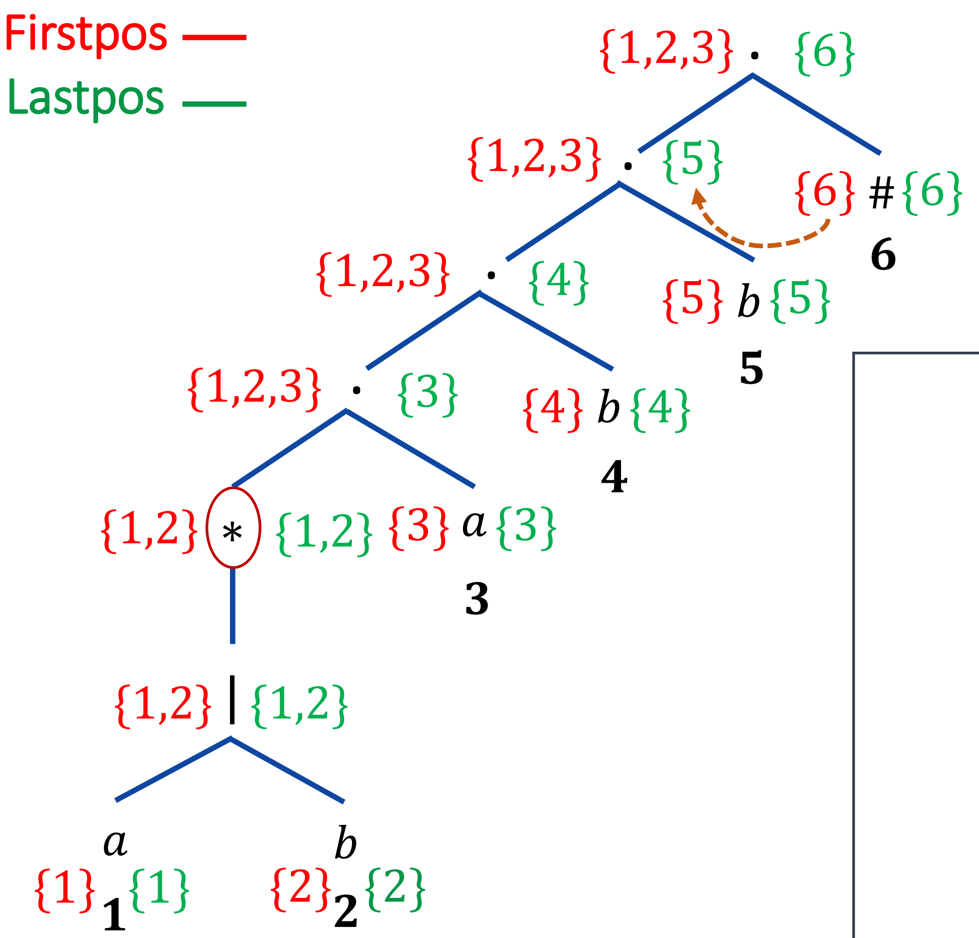


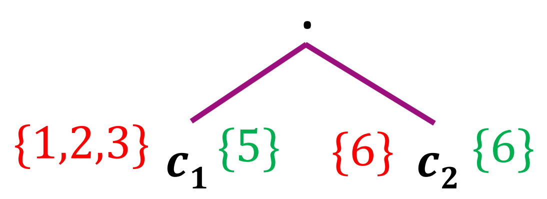
if $(nullable(c_2))$ **then**
$lastpos(c_1) \cup lastpos(c_2)$
else $lastpos(c_2)$

Conversion from regular expression to DFA

Step 4: Calculate followpos

Position	followpos
5	6



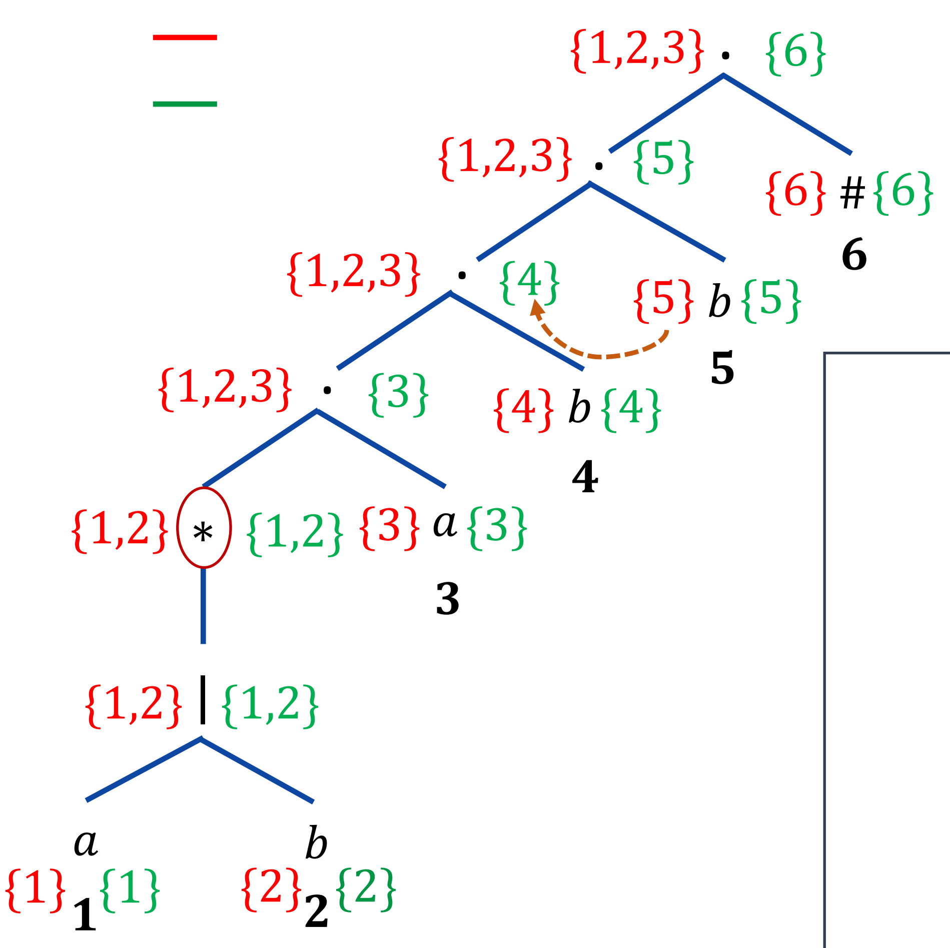


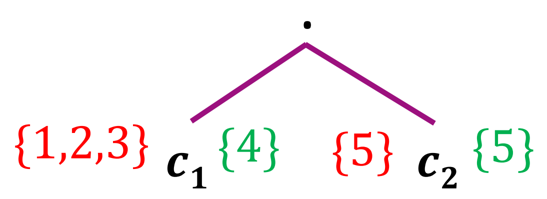
$$i = lastpos(c_1) = \{5\}$$
$$firstpos(c_2) = \{6\}$$
$$followpos(5) = \{6\}$$

Conversion from regular expression to DFA

Step 4: Calculate followpos

Position	followpos
5	6
4	5



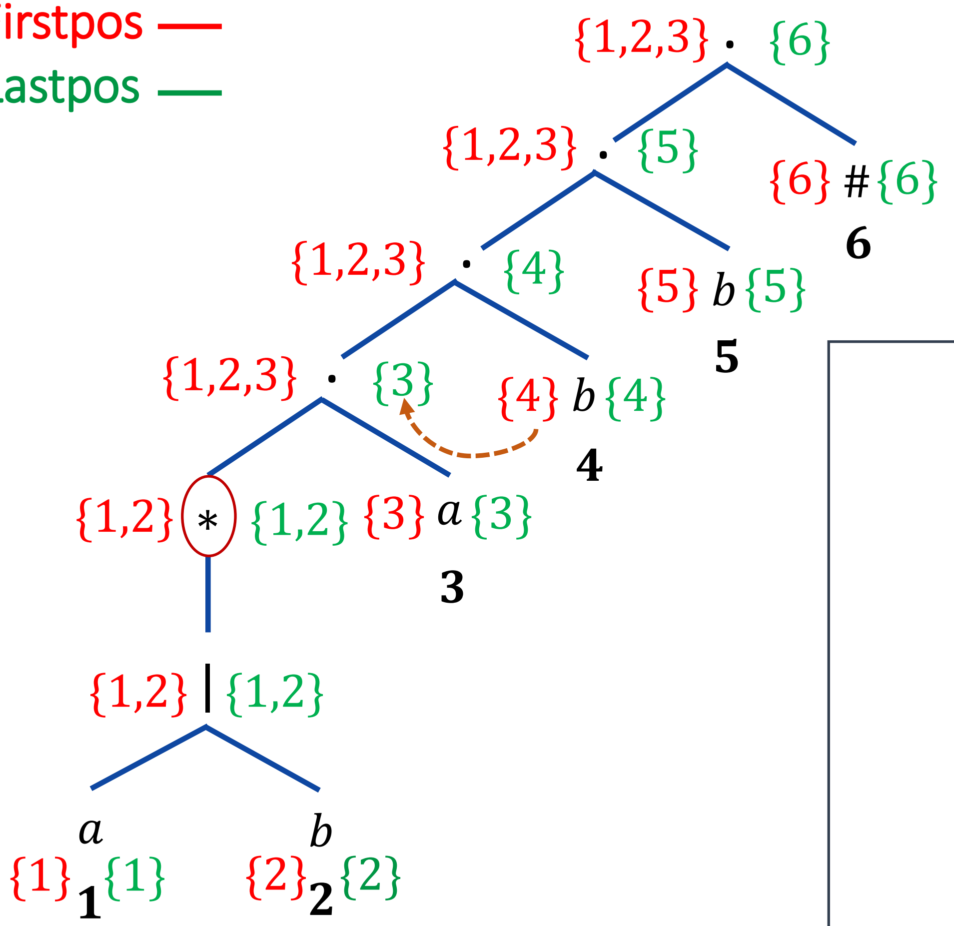


$$i = lastpos(c_1) = \{4\}$$
$$firstpos(c_2) = \{5\}$$
$$followpos(4) = \{5\}$$

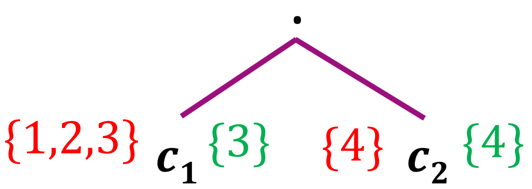
Conversion from regular expression to DFA

Step 4: Calculate followpos

Firstpos —
Lastpos —



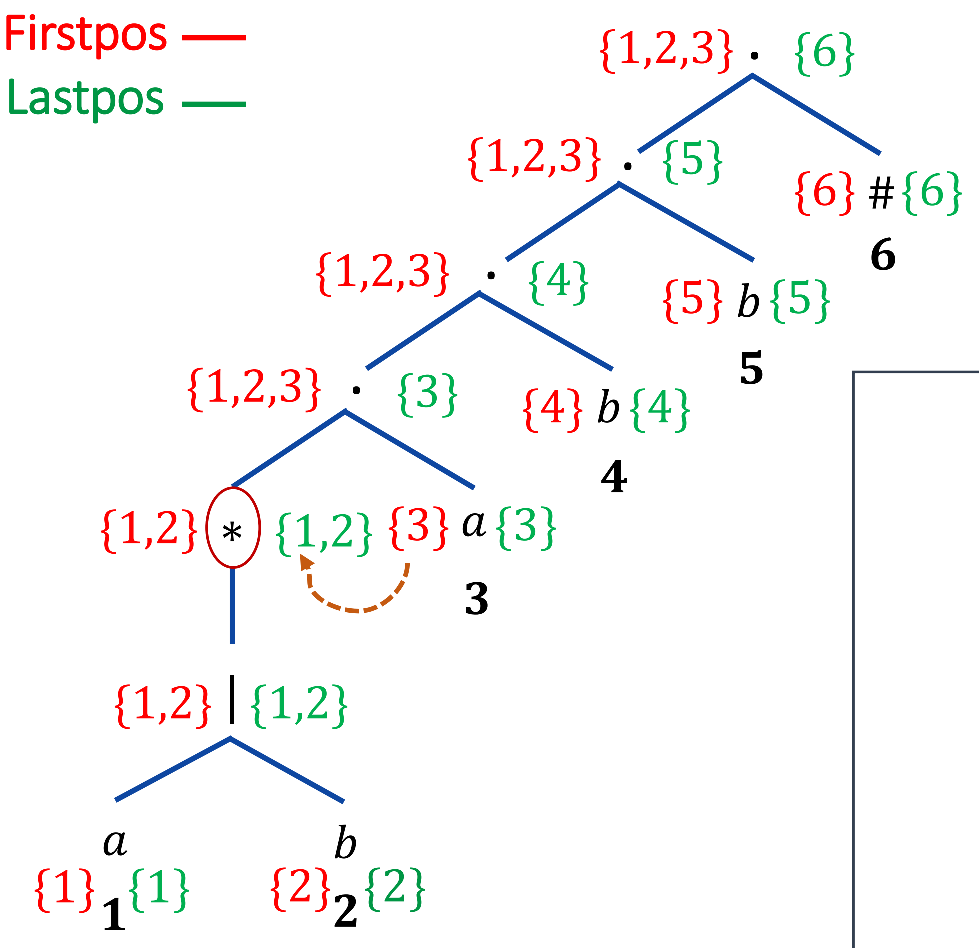
Position	followpos
5	6
4	5
3	4



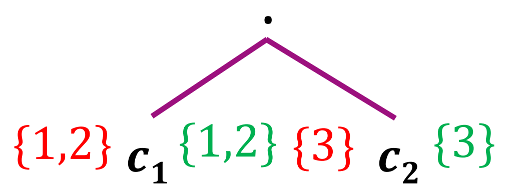
$$i = lastpos(c_1) = \{3\}$$
$$firstpos(c_2) = \{4\}$$
$$followpos(3) = \{4\}$$

Conversion from regular expression to DFA

Step 4: Calculate followpos



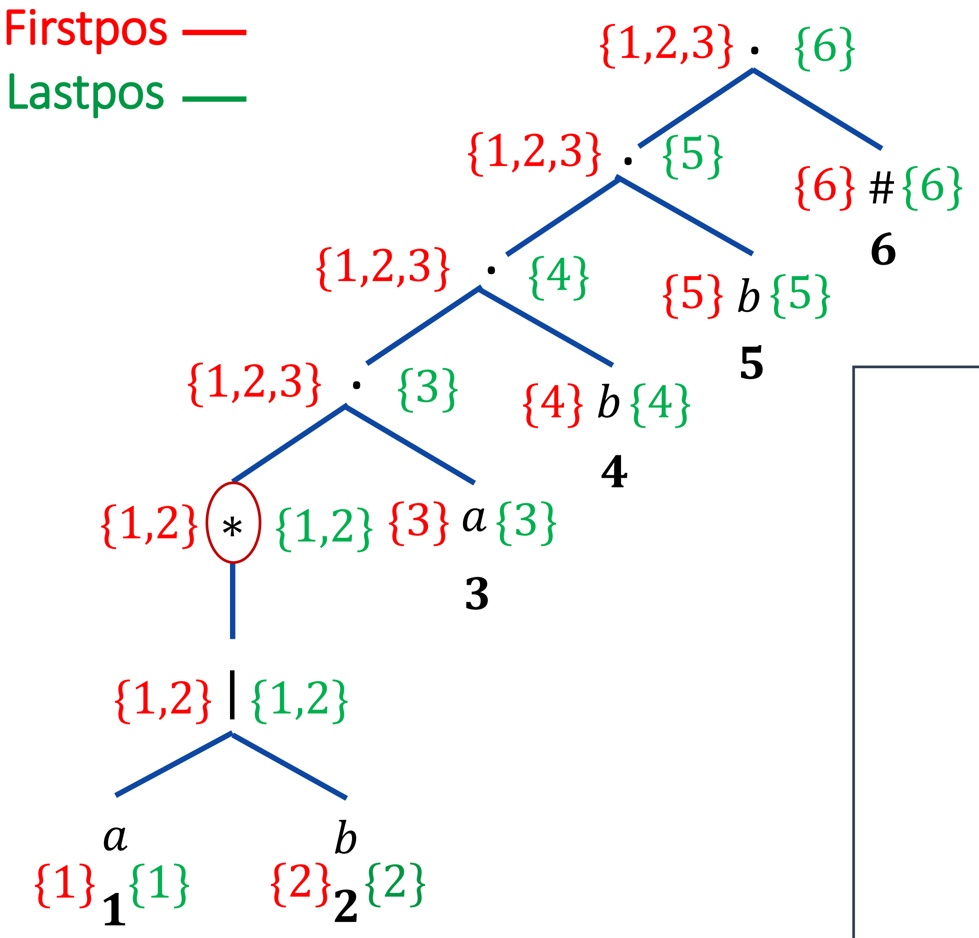
Position	followpos
5	6
4	5
3	4
2	3
1	3



$$i = lastpos(c_1) = \{1,2\}$$
$$firstpos(c_2) = \{3\}$$
$$followpos(1) = \{3\}$$
$$followpos(2) = \{3\}$$

Conversion from regular expression to DFA

Step 4: Calculate followpos



Position	followpos
5	6
4	5
3	4
2	1,2,3
1	1,2,3

$\{1,2\}$ $*$ $\{1,2\}$

$\boldsymbol{n}$

$$i = lastpos(n) = \{1,2\}$$
$$firstpos(n) = \{1,2\}$$
$$followpos(1) = \{1,2\}$$
$$followpos(2) = \{1,2\}$$

Conversion from regular expression to DFA

Initial state = $firstpos$ of root = {1,2,3} ----- A

<u>State A</u>

δ((1,2,3),a) = followpos(1) U followpos(3)

=(1,2,3) U (4) = {1,2,3,4} ----- B

δ((1,2,3),b) = followpos(2)

=(1,2,3) ----- A

Position	followpos
5	6
4	5
3	4
2	1,2,3
1	1,2,3

States	a	b
A={1,2,3}		
B={1,2,3,4}		

Conversion from regular expression to DFA

State B

$\delta((1,2,3,4),a)$ = followpos(1) U followpos(3)

=(1,2,3) U (4) = {1,2,3,4} ----- B

$\delta((1,2,3,4),b)$ = followpos(2) U followpos(4)

=(1,2,3) U (5) = {1,2,3,5} ----- C

State C

$\delta((1,2,3,5),a)$ = followpos(1) U followpos(3)

=(1,2,3) U (4) = {1,2,3,4} ----- B

$\delta((1,2,3,5),b)$ = followpos(2) U followpos(5)

=(1,2,3) U (6) = {1,2,3,6} ----- D

Position	followpos
5	6
4	5
3	4
2	1,2,3
1	1,2,3

States	a	b
A={1,2,3}	B	A
B={1,2,3,4}		
C={1,2,3,5}		
D={1,2,3,6}		

Conversion from regular expression to DFA

State D

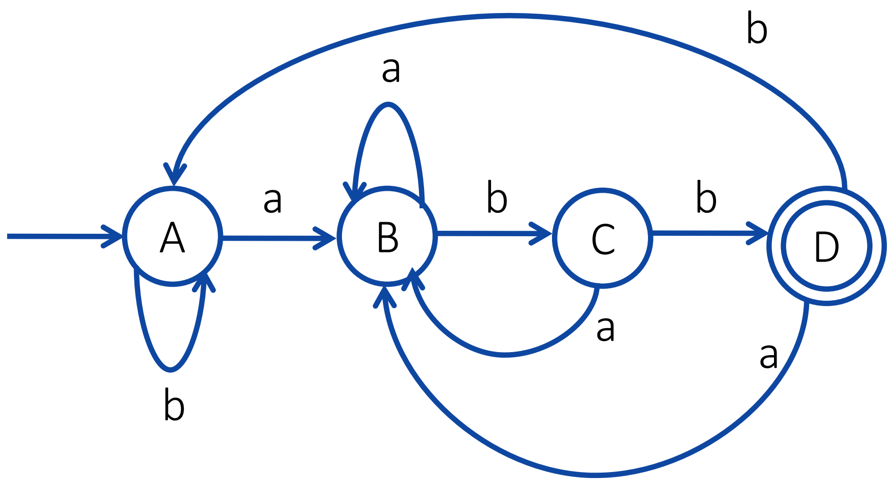
δ((1,2,3,6),a) = followpos(1) U followpos(3)

=(1,2,3) U (4) = {1,2,3,4} ----- B

δ((1,2,3,6),b) = followpos(2)

=(1,2,3) ----- A

Position	followpos
5	6
4	5
3	4
2	1,2,3
1	1,2,3



DFA

States	a	b
A={1,2,3}	B	A
B={1,2,3,4}	B	C
C={1,2,3,5}	B	D
D={1,2,3,6}		

Conversion from regular expression to DFA

Construct DFA for following regular expression:

1. (c | d)*c
2. (a+b)*+(a.c)*