RAJIV GANDHI INSTITUTE OF PETROLEUM TECHNOLOGY, JAIS, AMETHI

Department of Computer Science and Engineering

Compiler Design (CS312)

By

Dr. Kalka Dubey

Acknowledgements

- Most of the text in the slide is based on classic text **Compilers: Principles, Techniques, and Tools** by **Aho, Sethi, Ullman** and **Lam**

Organization of the course

- **Assignments/Quiz 20%**
- **Mid semester exam 30%**
- **End semester exam 50%**

Bit of History

- How are programming languages implemented? Two major strategies:
  - Interpreters (old and much less studied)
  - Compilers (very well understood with mathematical foundations)

Bit of History

- Some environments provide both interpreter and compiler. Lisp, scheme etc. provide
  - Interpreter for development
  - Compiler for deployment
  -

Bit of History

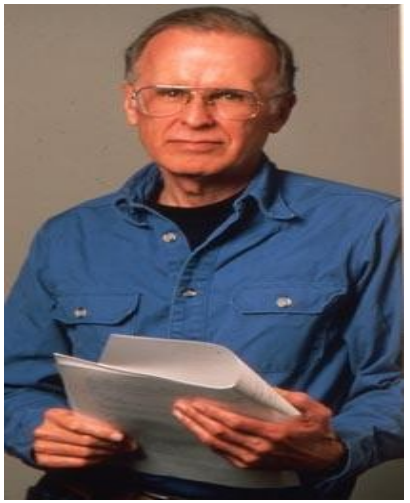
- Java
  - Java compiler: Java to interpretable bytecode
  - Java JIT: bytecode to executable image

Some early machines and implementations

- IBM developed 704 in 1954.
- All programming was done in assembly language.
- Cost of software development far exceeded cost of hardware.
- Low productivity.

Some early machines and implementations

- John Backus (in 1954): Proposed a program that translated high level expressions into native machine code. Skeptism all around.



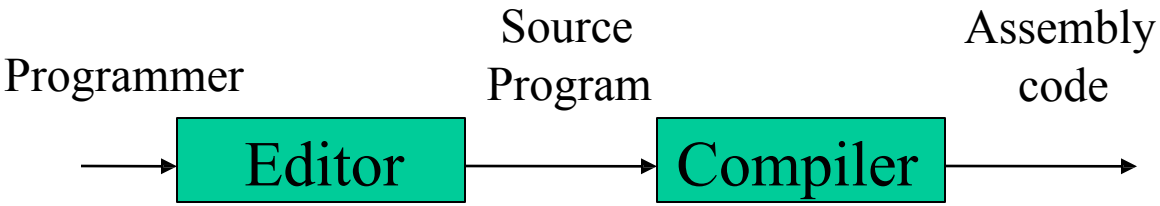
Fortran I

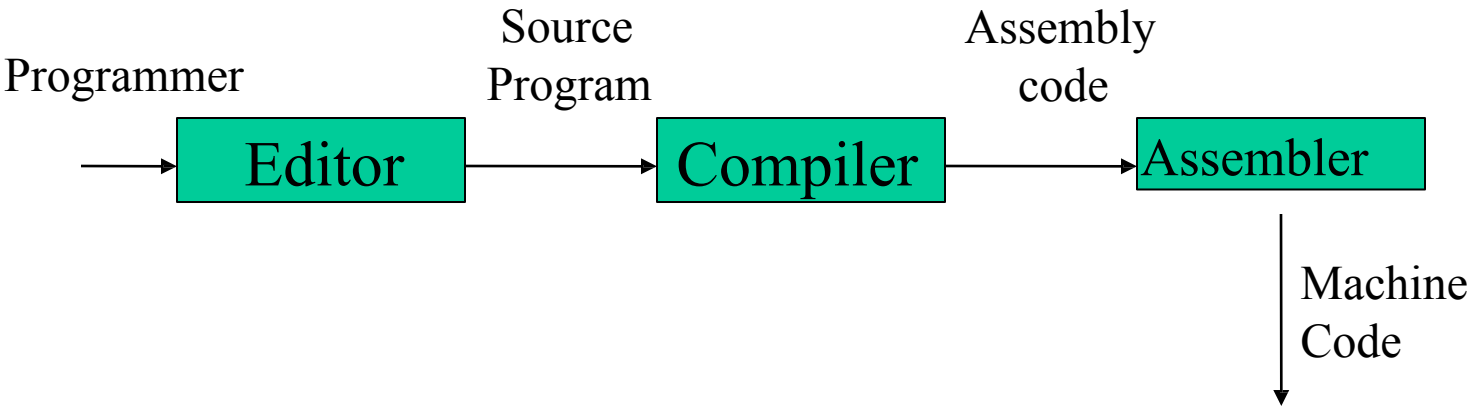
- The first compiler had a huge impact on the programming languages and computer science. The whole new field of compiler design was started
- More than half the programmers were using Fortran by 1958
- The development time was cut down to half
- Led to enormous amount of theoretical work (lexical analysis, parsing, optimization, structured programming, code generation, error recovery etc.)
- Modern compilers preserve the basic structure of the Fortran I compiler !!!

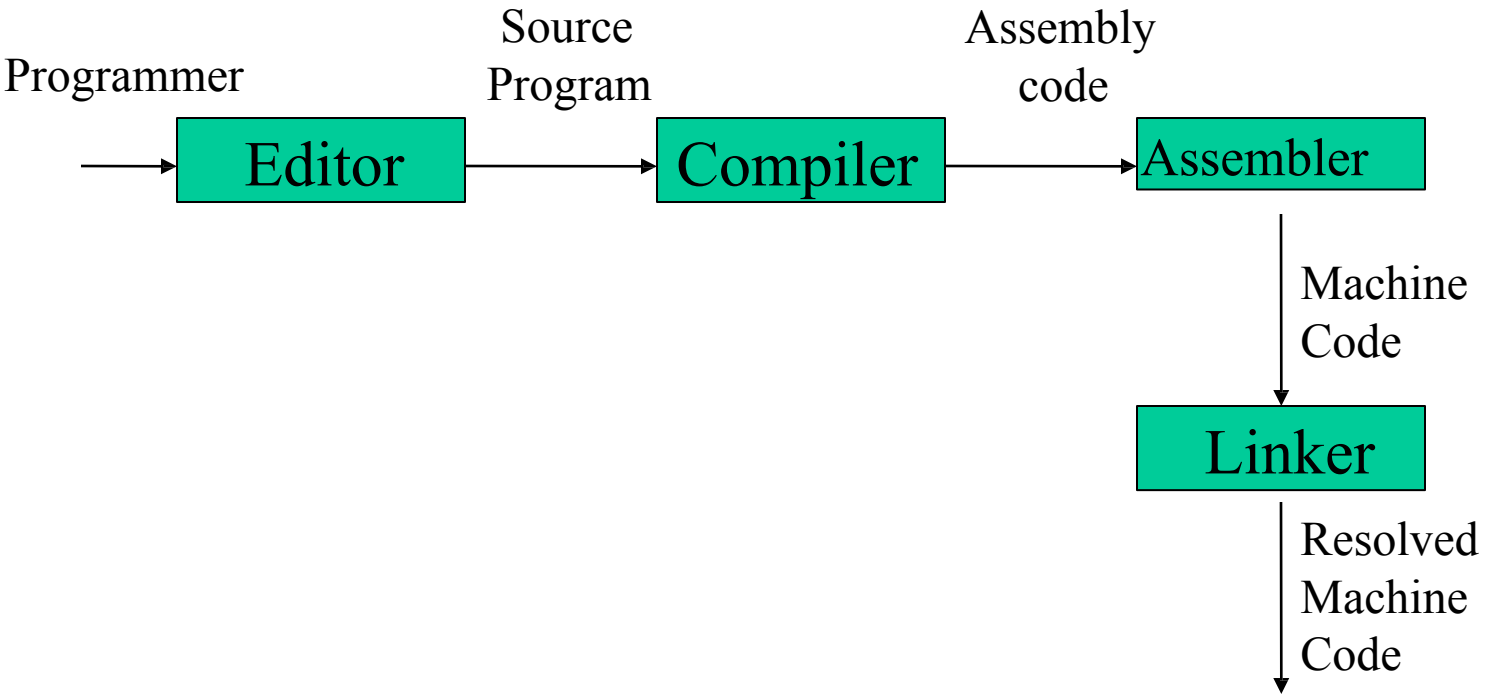
The big picture

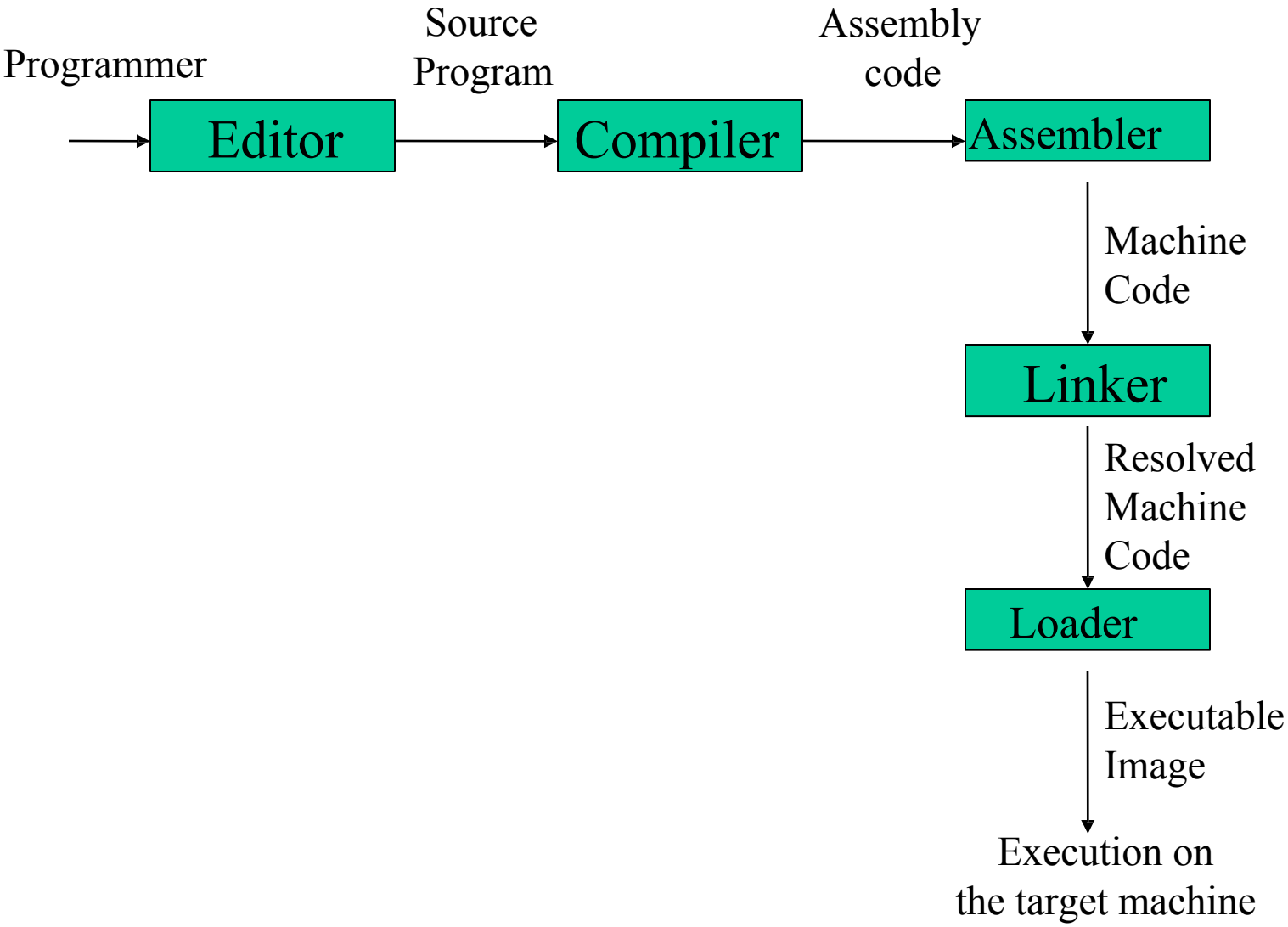
- Compiler is part of program development environment
- The other typical components of this environment are editor, assembler, linker, loader, debugger, profiler etc.
- The compiler (and all other tools) must support each other for easy program development

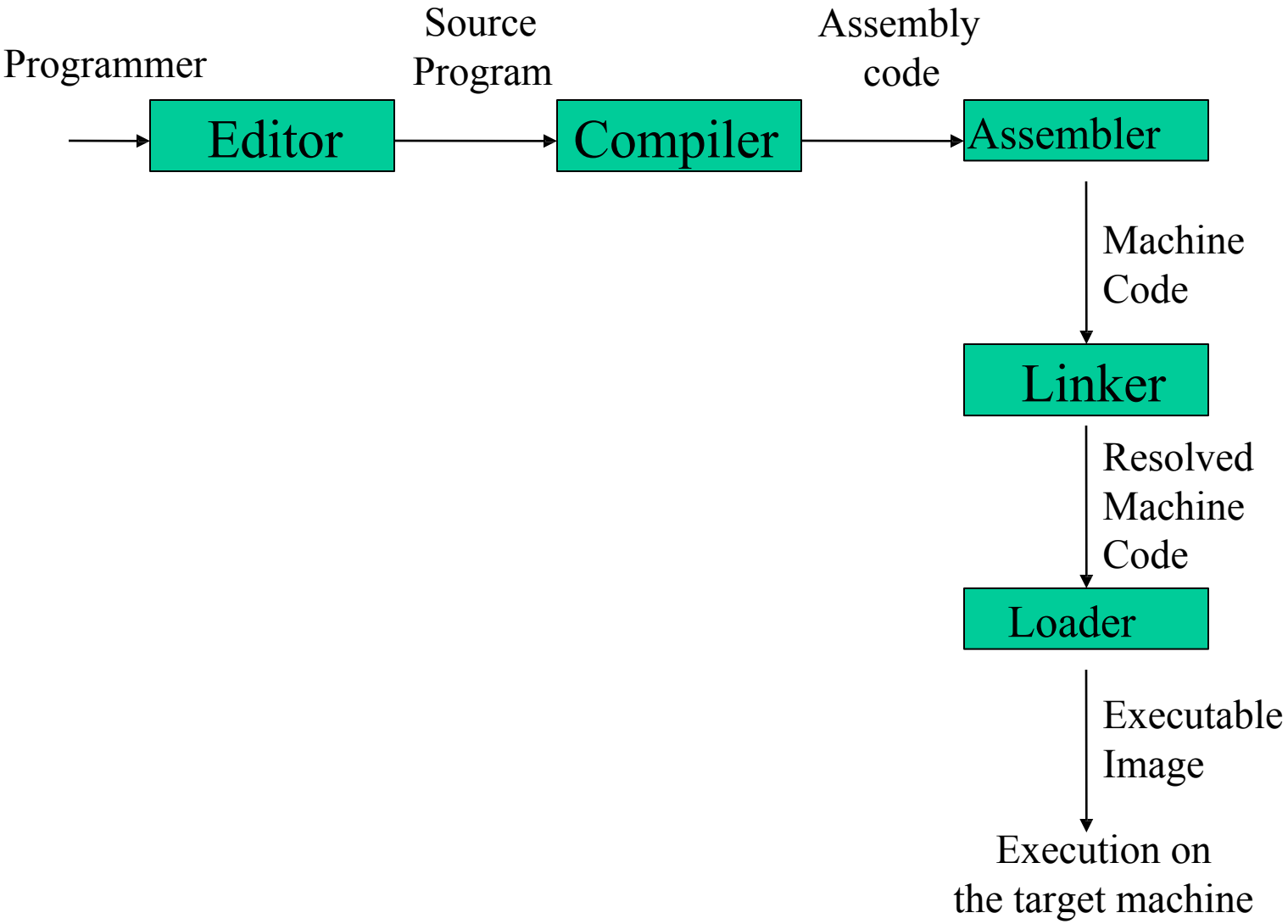

Programmer
Source
Program
Editor


Programmer
Source
Program
Assembly
code
Editor
Compiler

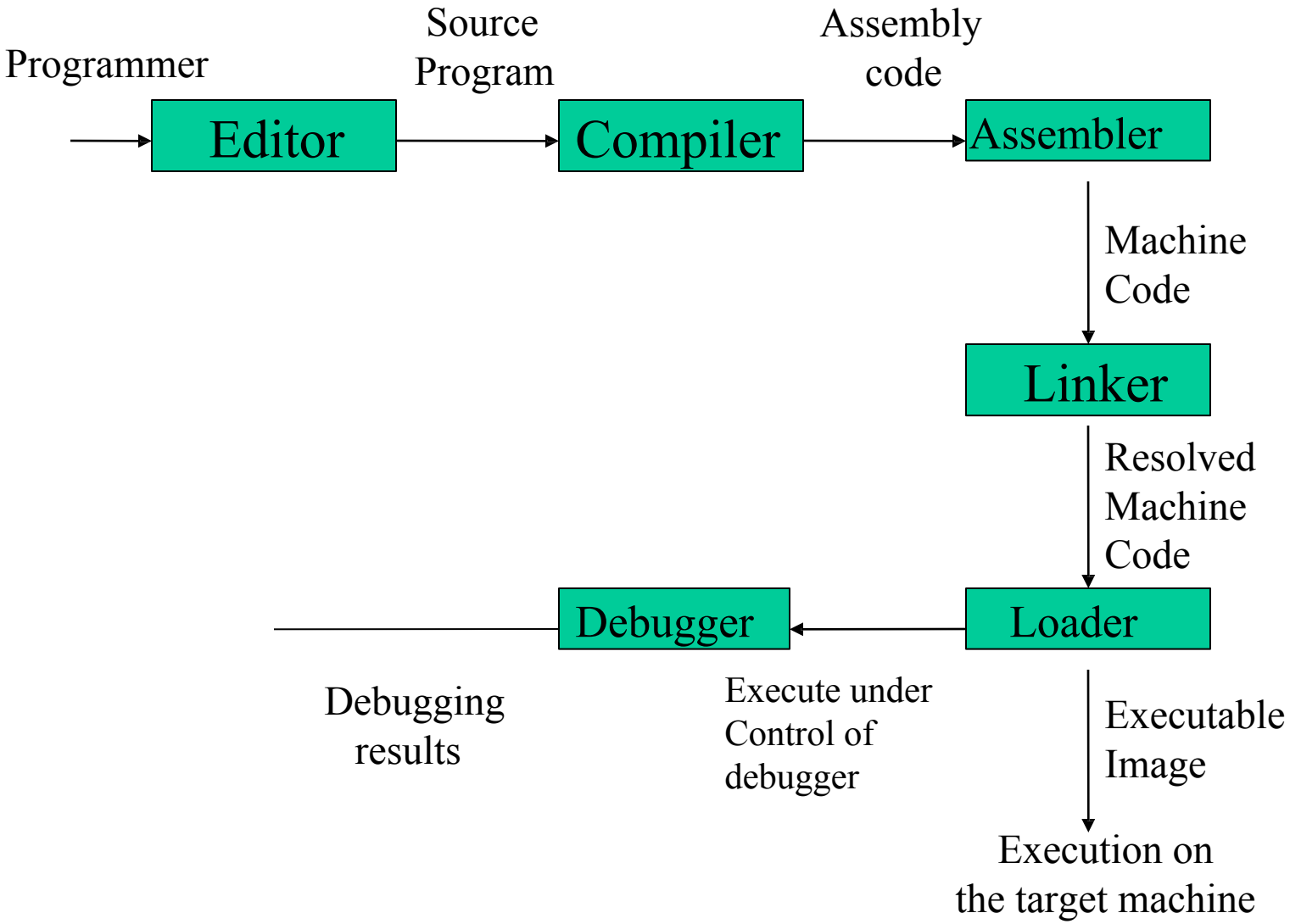

Programmer
Source
Program
Assembly
code
Editor
Compiler
Assembler
Machine
Code


Programmer
Source Program
Assembly code
Editor
Compiler
Assembler
Machine Code
Linker
Resolved Machine Code

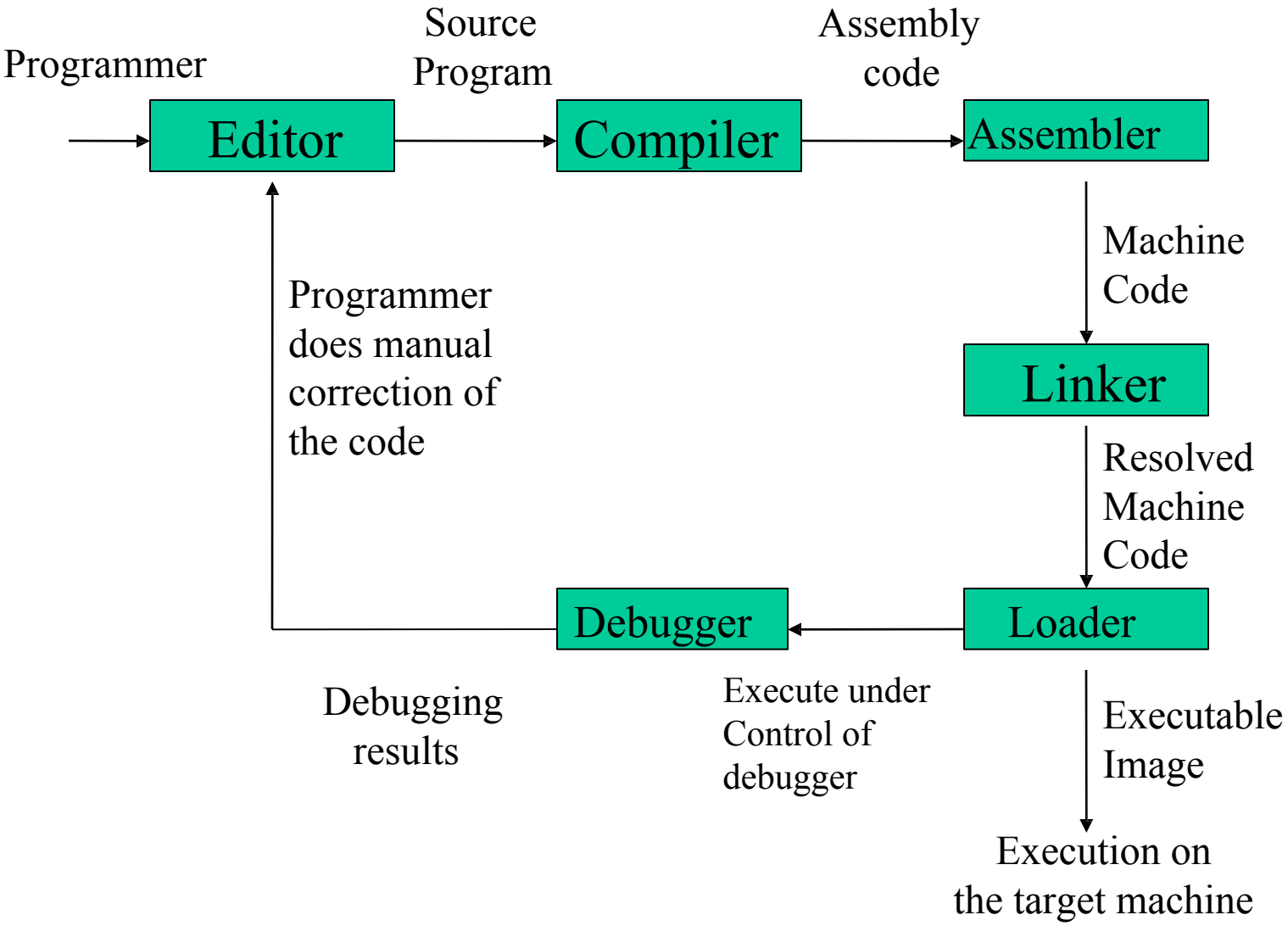

Programmer
Source Program
Assembly code
Editor
Compiler
Assembler
Machine Code
Linker
Resolved Machine Code
Loader
Executable Image
Execution on the target machine



Normally end up with error



Normally end up with error



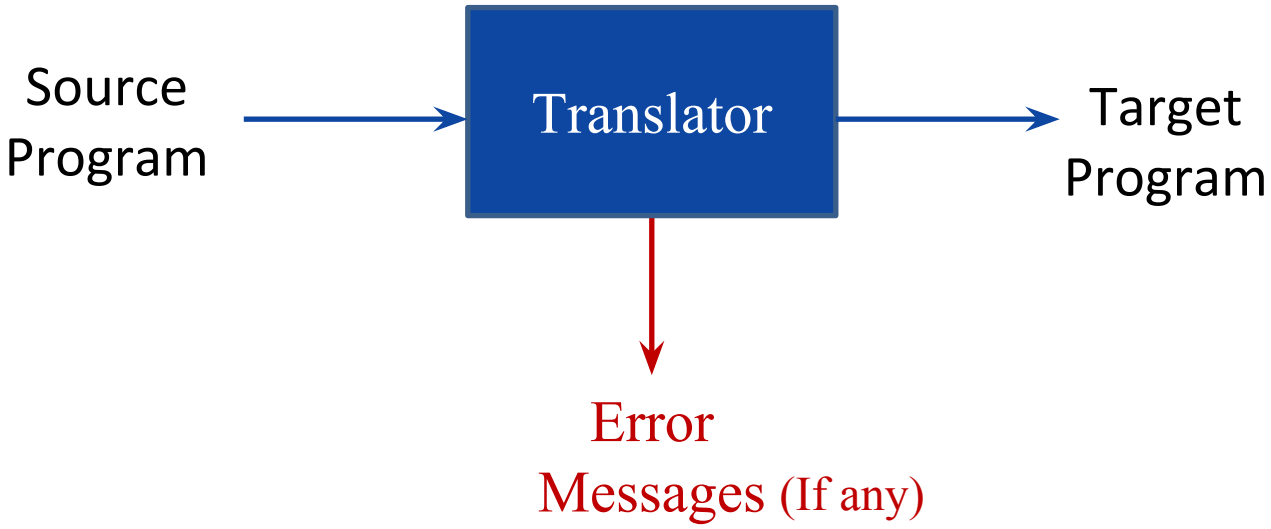
Normally end up with error

Translator

A translator is a program that **takes one form of program as input** and **converts it into another form.**

Types of translators are:

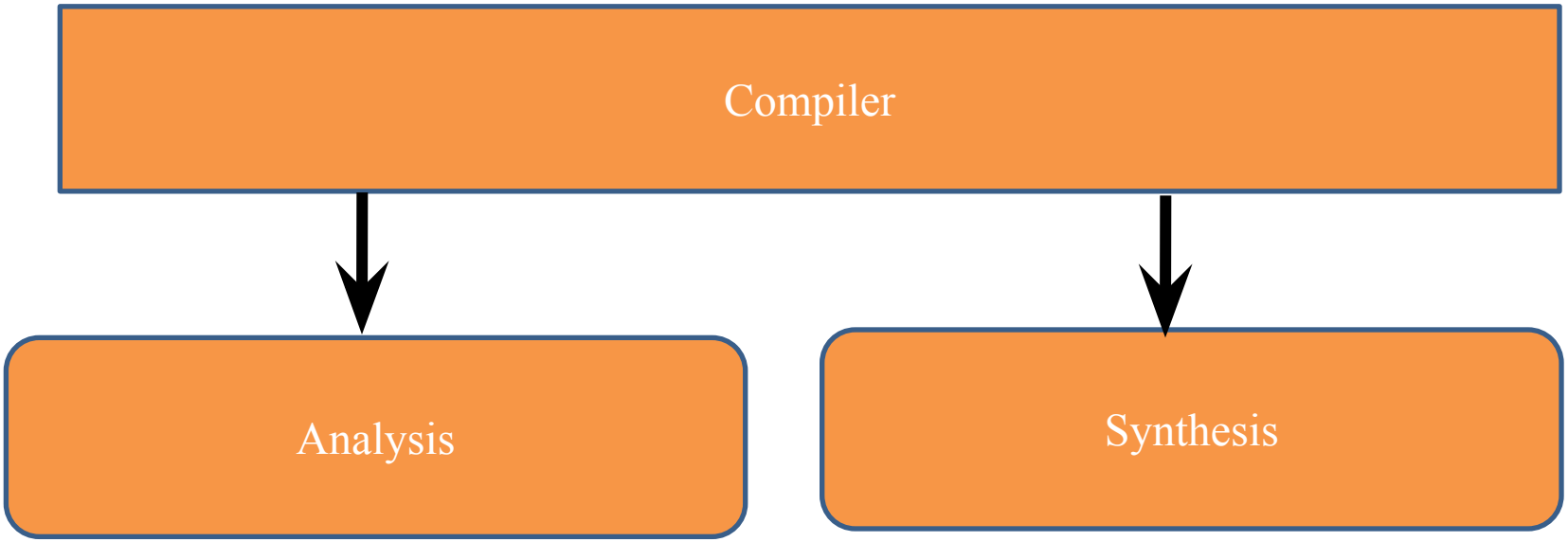
1. Compiler
2. Interpreter
3. Assembler



The Structure of a Compiler

- Any compiler must perform two major tasks

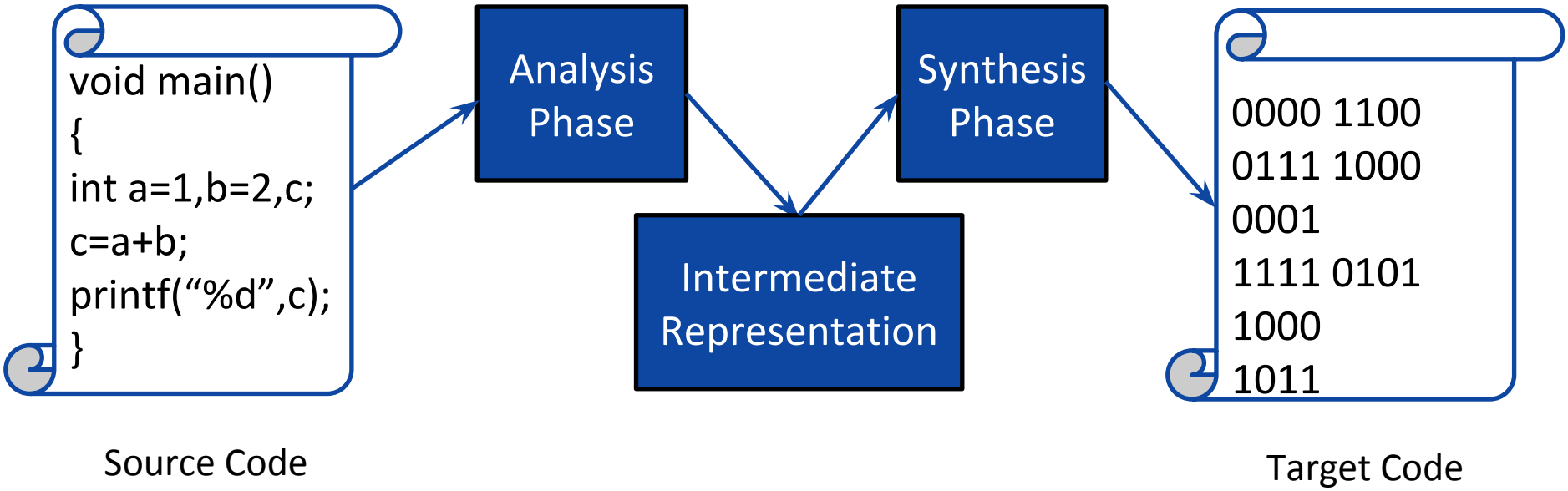
Analysis of the source program
Synthesis of a machine-language program



Analysis synthesis model of compilation

There are two parts of compilation.

1. Analysis Phase
2. 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

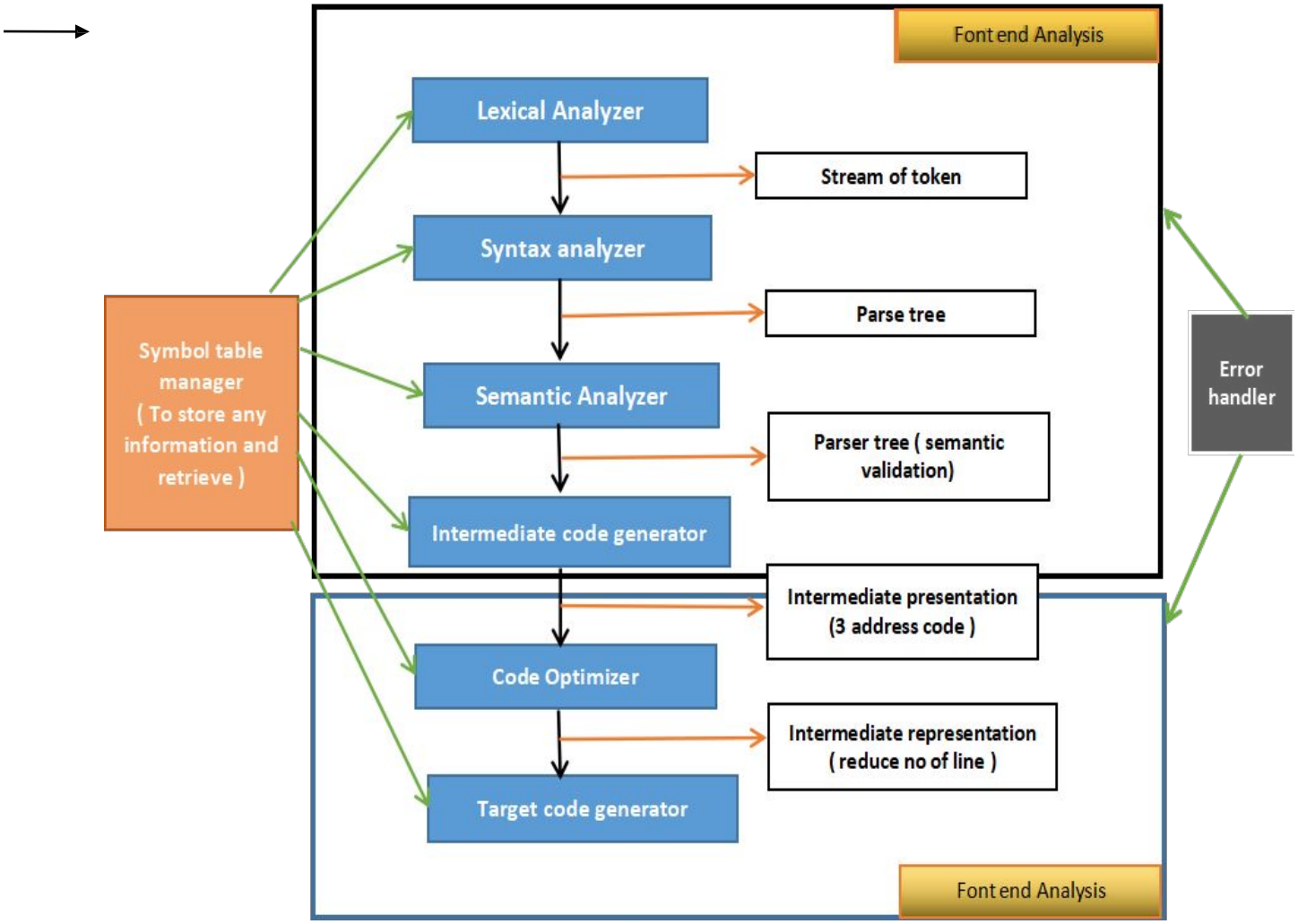
Synthesis Phase

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

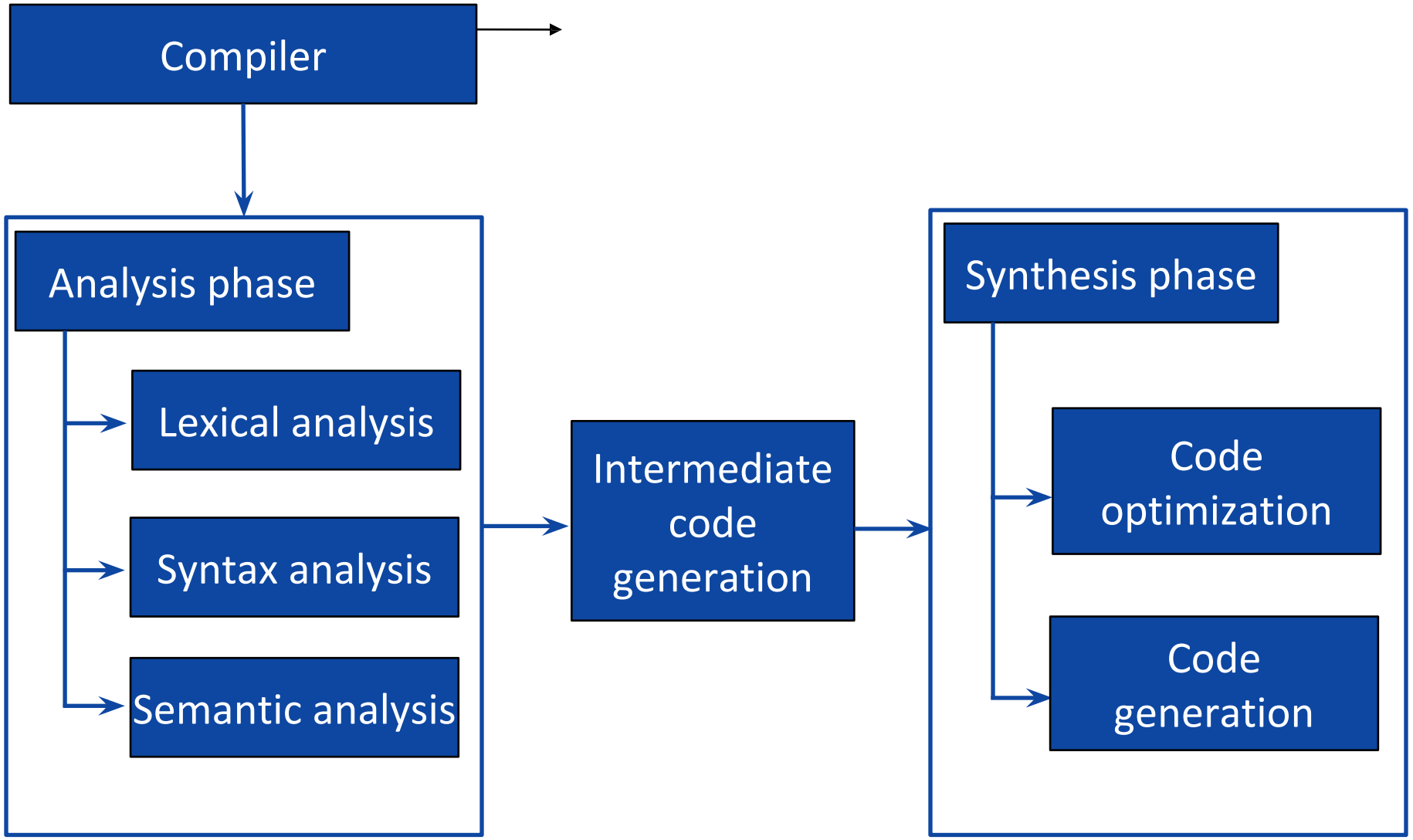
1. Code optimization

2. Code generation

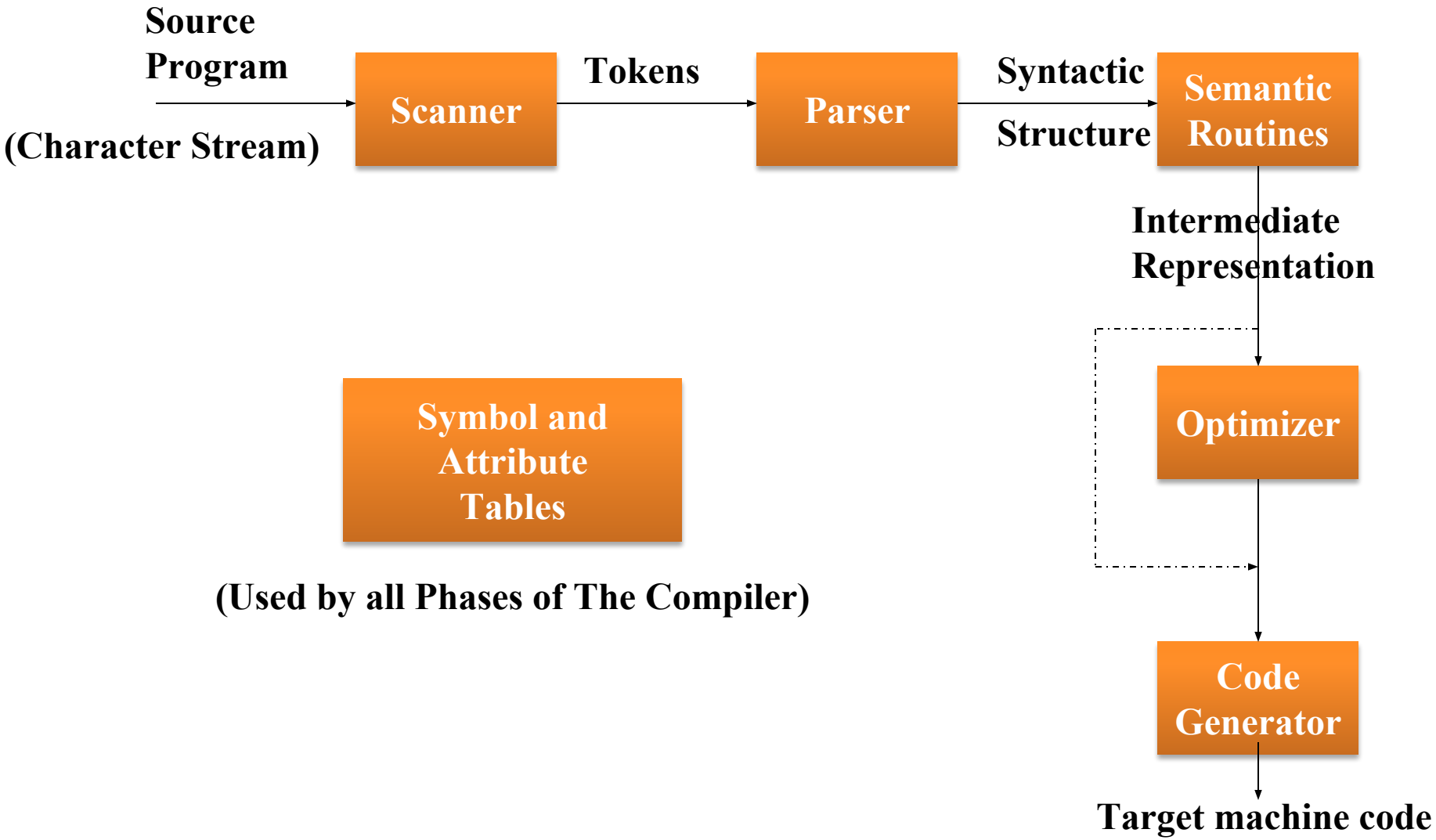
The Structure of a Compiler



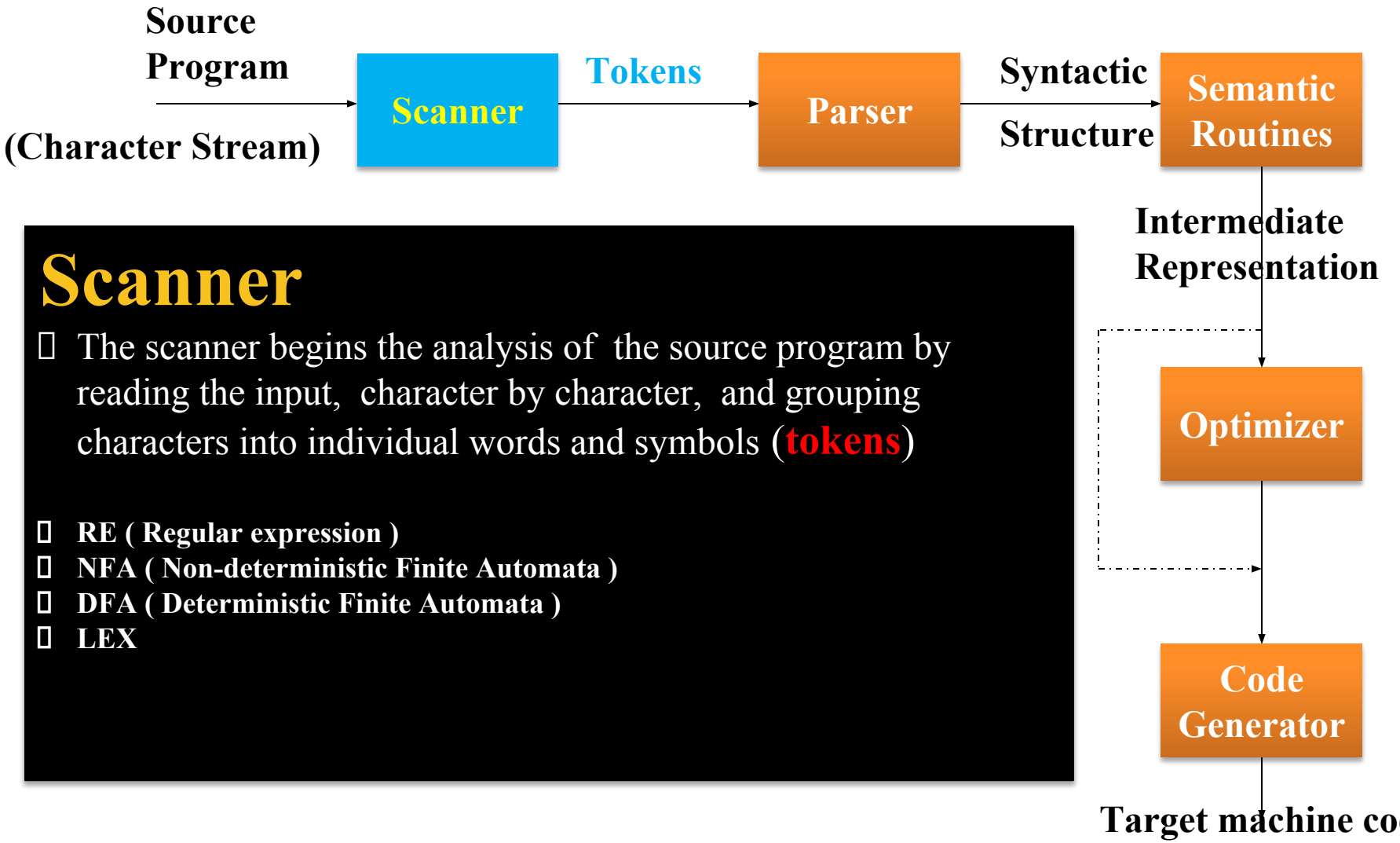
The Structure of a Compiler



The Structure of a Compiler



The Structure of a Compiler



Scanner

- The scanner begins the analysis of the source program by reading the input, character by character, and grouping characters into individual words and symbols (**tokens**)

- **RE (Regular expression)**
- **NFA (Non-deterministic Finite Automata)**
- **DFA (Deterministic Finite Automata)**
- **LEX**

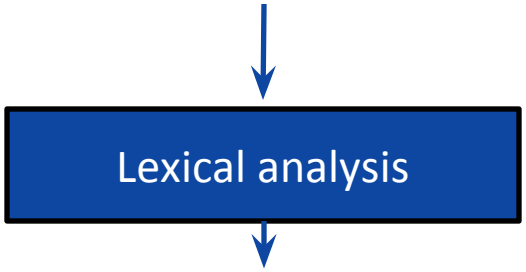
Lexical Analysis

- Lexical Analysis is also called **linear analysis** or **scanning**.

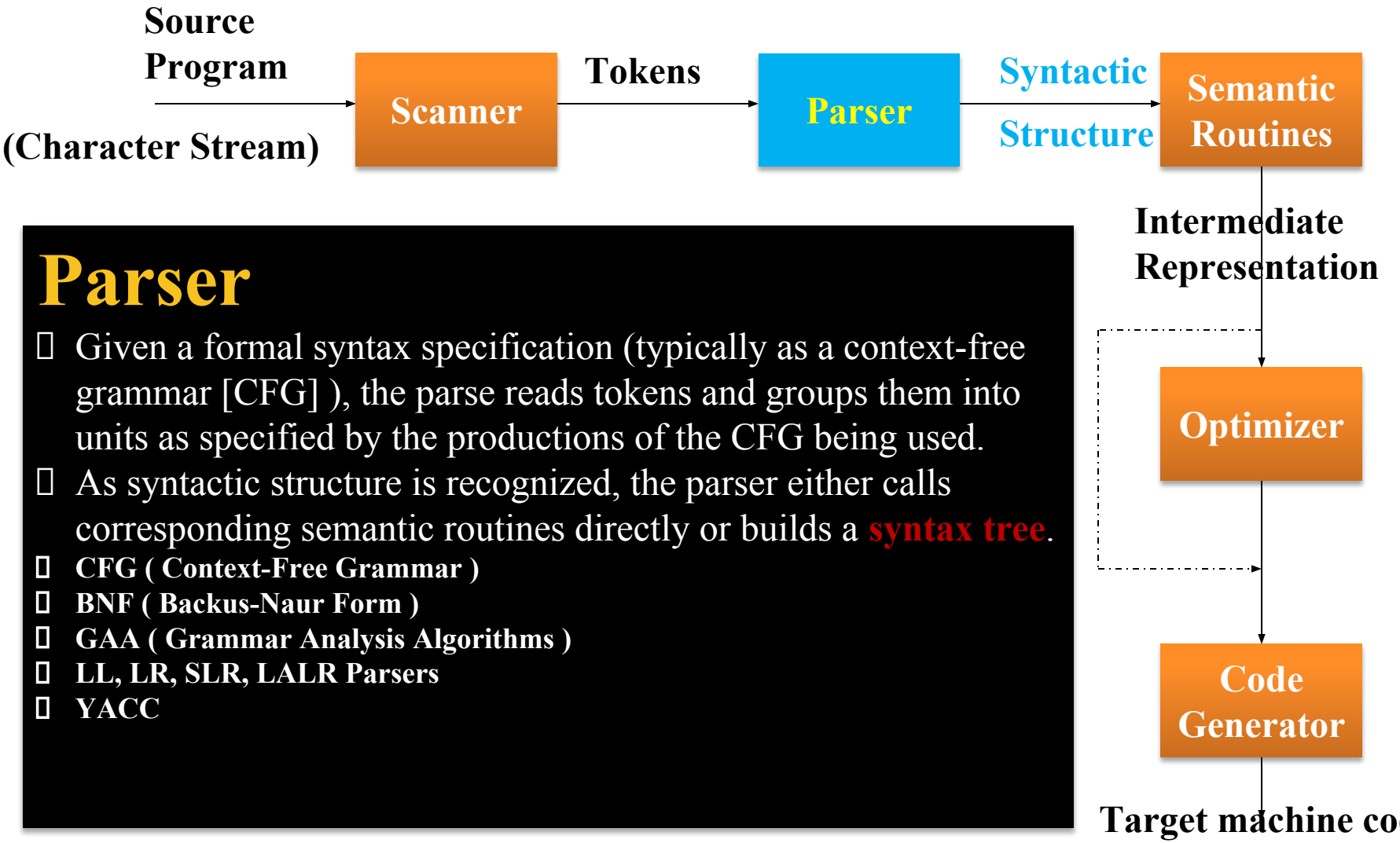
- Lexical Analyzer divides the given source statement into the **tokens**.
- Ex: **Position = initial + rate * 60** would be grouped into the following tokens:

Position (identifier)
= (Assignment symbol)
initial (identifier)
+ (Plus symbol)
rate (identifier)
* (Multiplication symbol)
60 (Number)

Lexical Analysis



The Structure of a Compiler



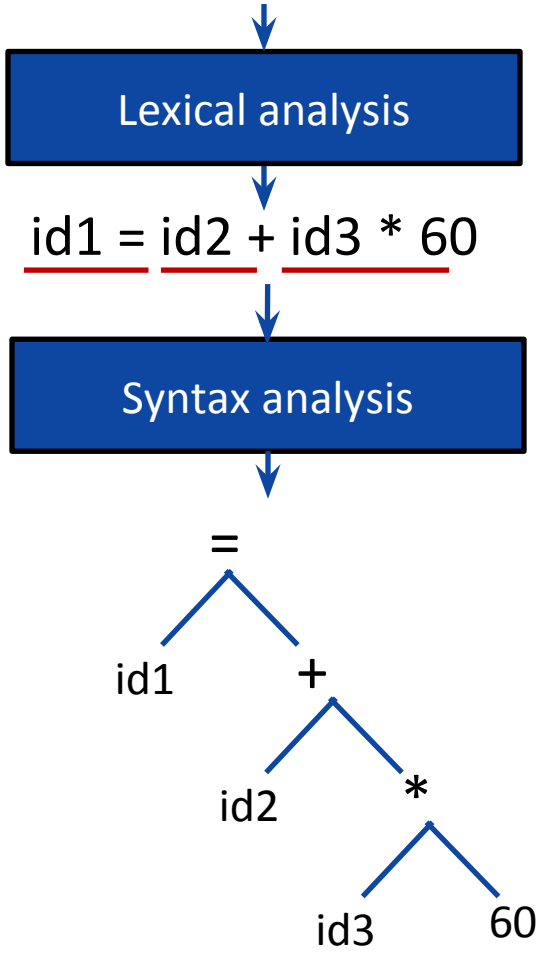
Parser

- Given a formal syntax specification (typically as a context-free grammar [CFG]), the parse reads tokens and groups them into units as specified by the productions of the CFG being used.
- As syntactic structure is recognized, the parser either calls corresponding semantic routines directly or builds a **syntax tree**.
- **CFG (Context-Free Grammar)**
- **BNF (Backus-Naur Form)**
- **GAA (Grammar Analysis Algorithms)**
- **LL, LR, SLR, LALR Parsers**
- **YACC**

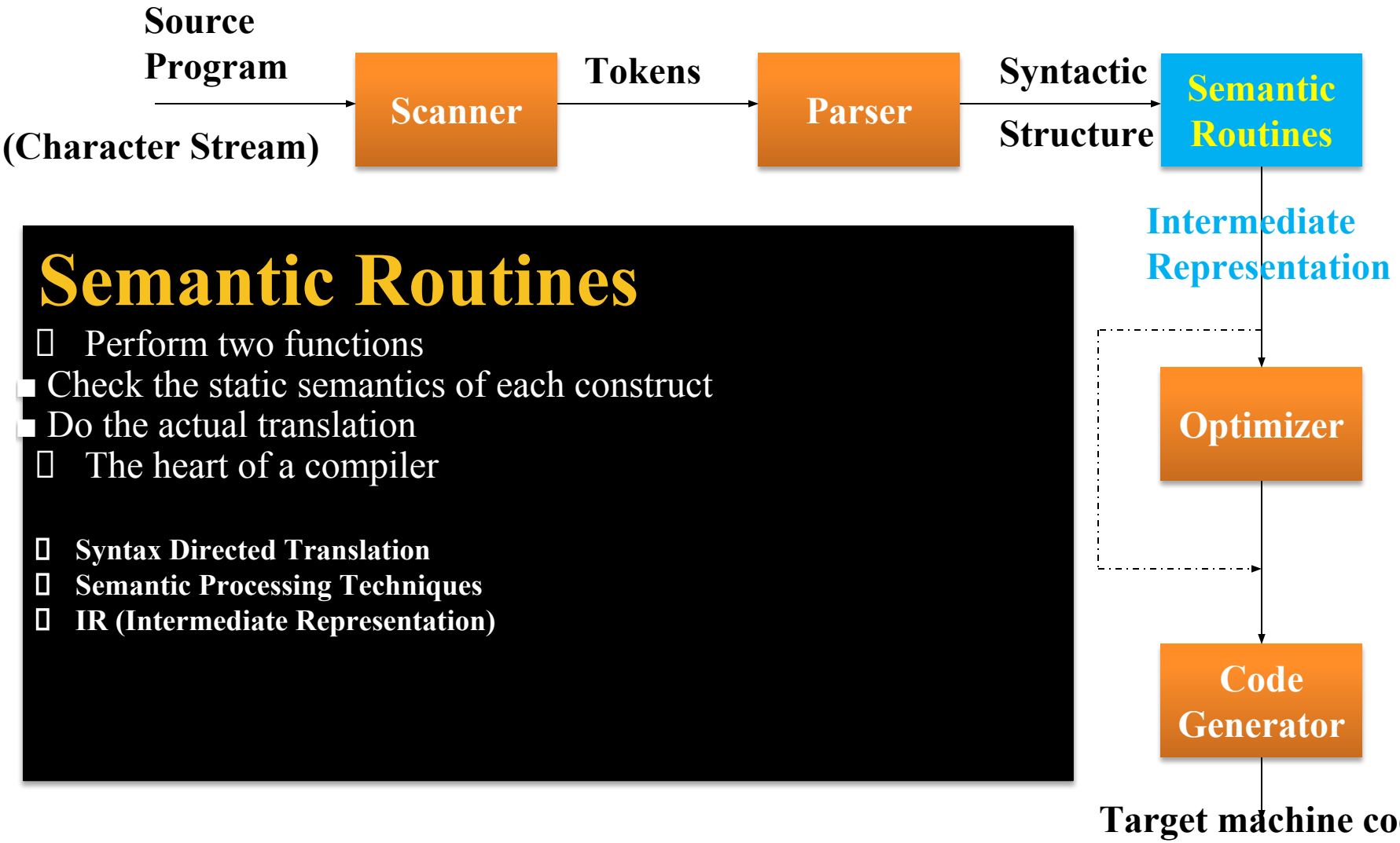
Syntax analysis

- Syntax Analysis is also called Parsing or Hierarchical Analysis.
- 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.

Syntax Analysis



The Structure of a Compiler



Semantic Routines

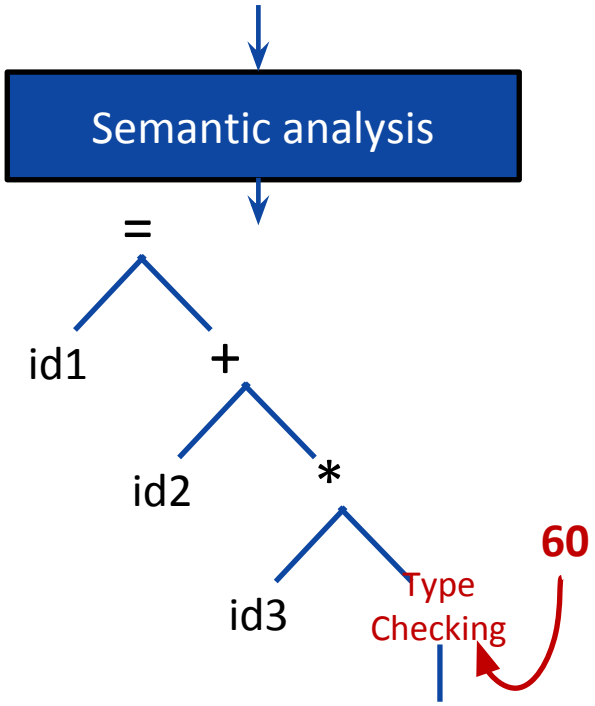
- Perform two functions
  - Check the static semantics of each construct
  - Do the actual translation
- The heart of a compiler

- **Syntax Directed Translation**
- **Semantic Processing Techniques**
- **IR (Intermediate Representation)**

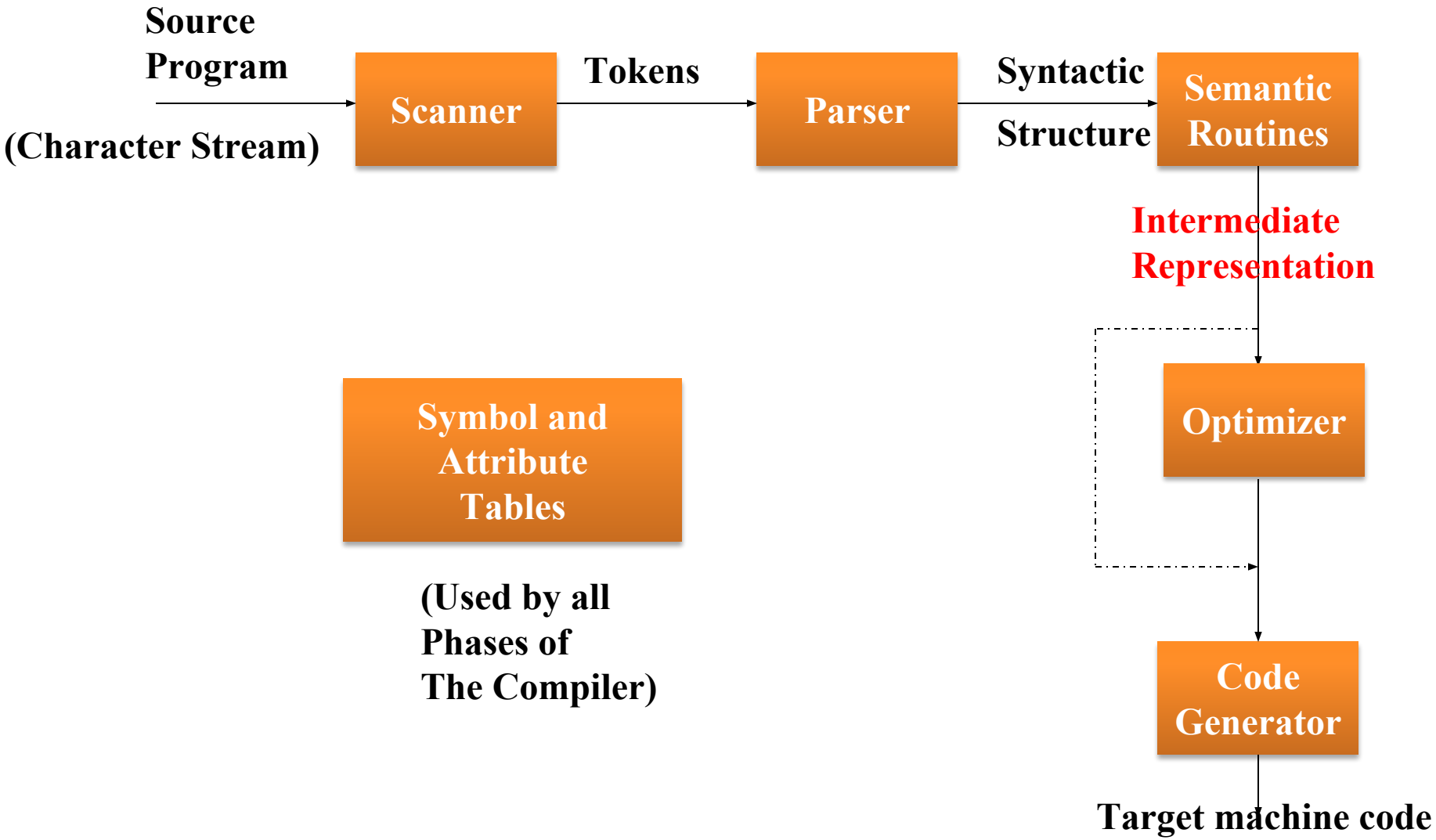
Semant analysis

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

Semant Analysis



The Structure of a Compiler



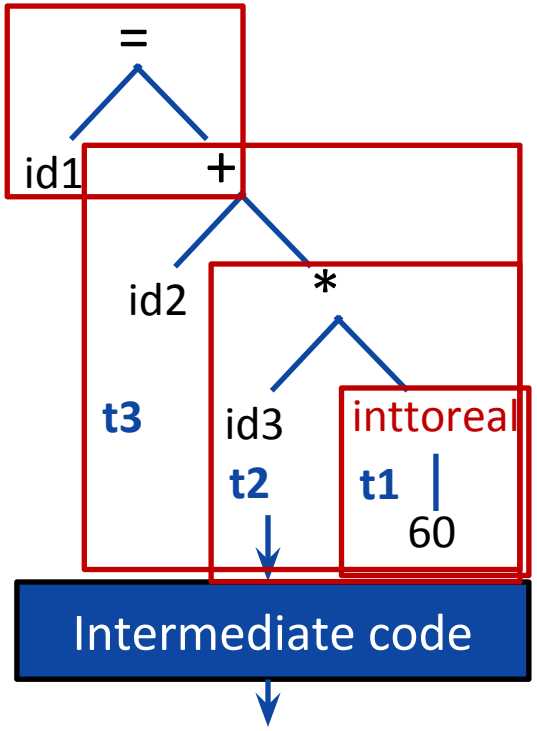
Intermediate code generator

- Two important properties of intermediate code :
- It should be easy to produce.
- 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.

Intermediate code generator



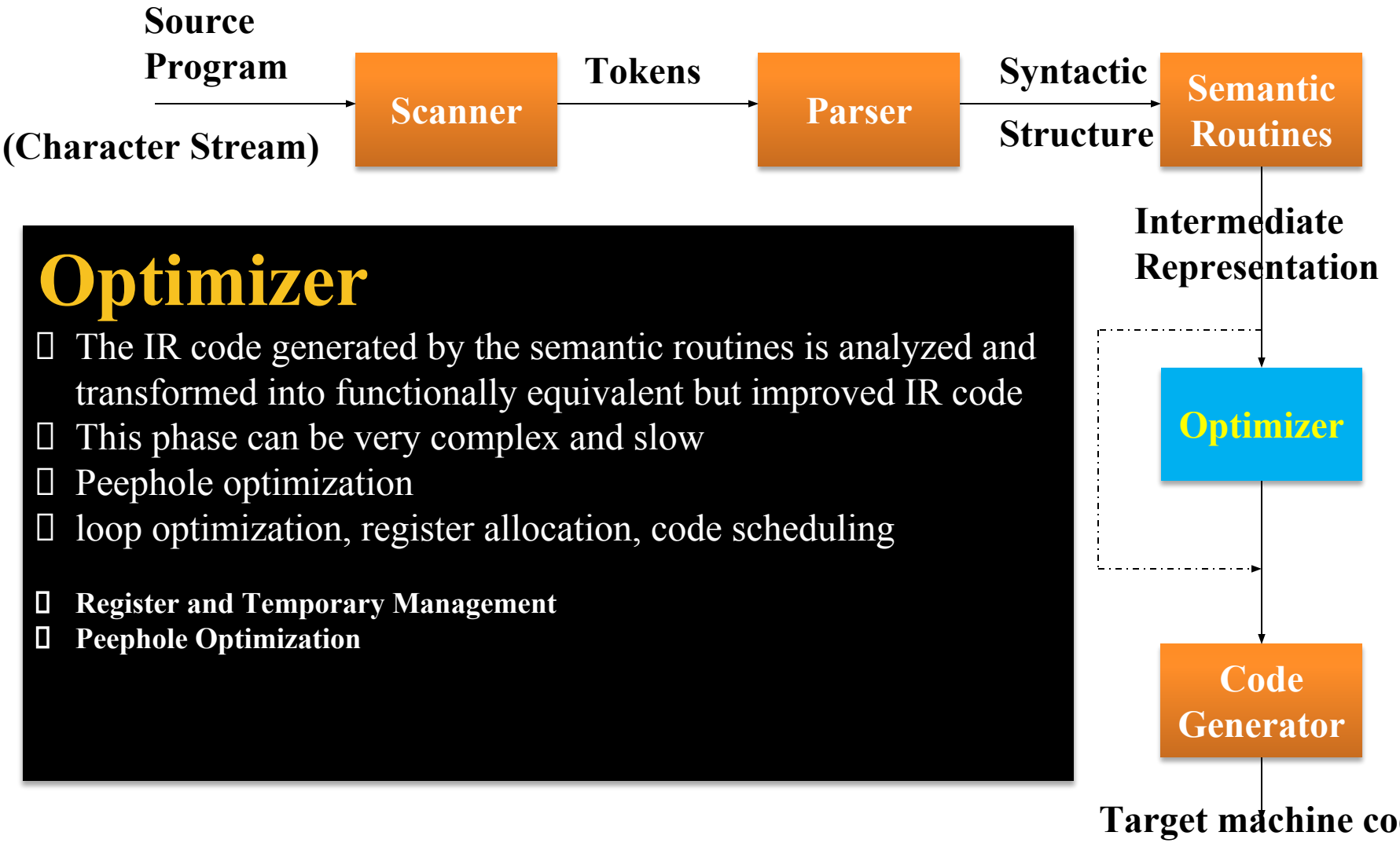
t1= int to real(60)

t2= id3 * t1

t3= t2 + id2

id1= t3

The Structure of a Compiler



Optimizer

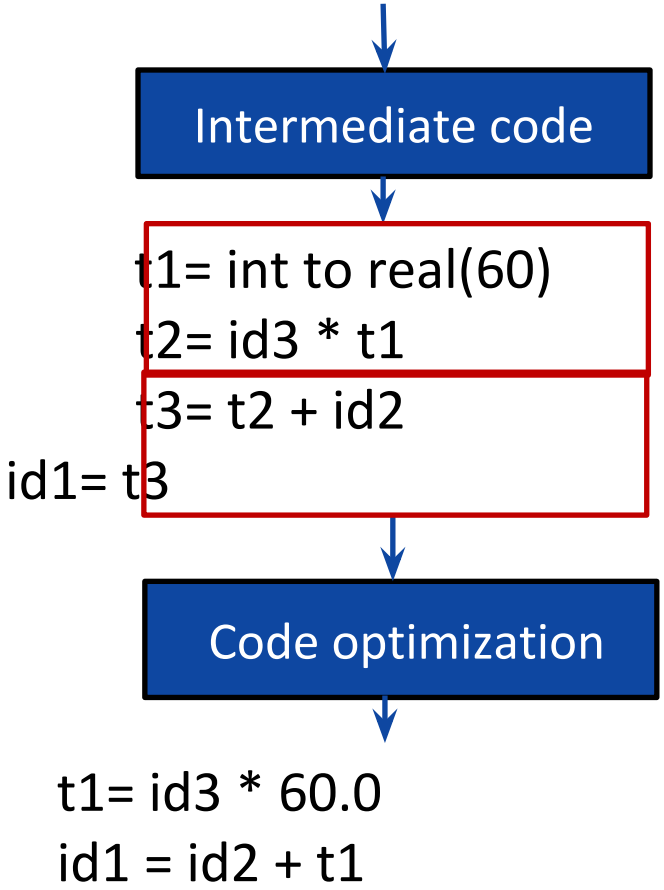
- The IR code generated by the semantic routines is analyzed and transformed into functionally equivalent but improved IR code
- This phase can be very complex and slow
- Peephole optimization
- loop optimization, register allocation, code scheduling

- **Register and Temporary Management**
- **Peephole Optimization**

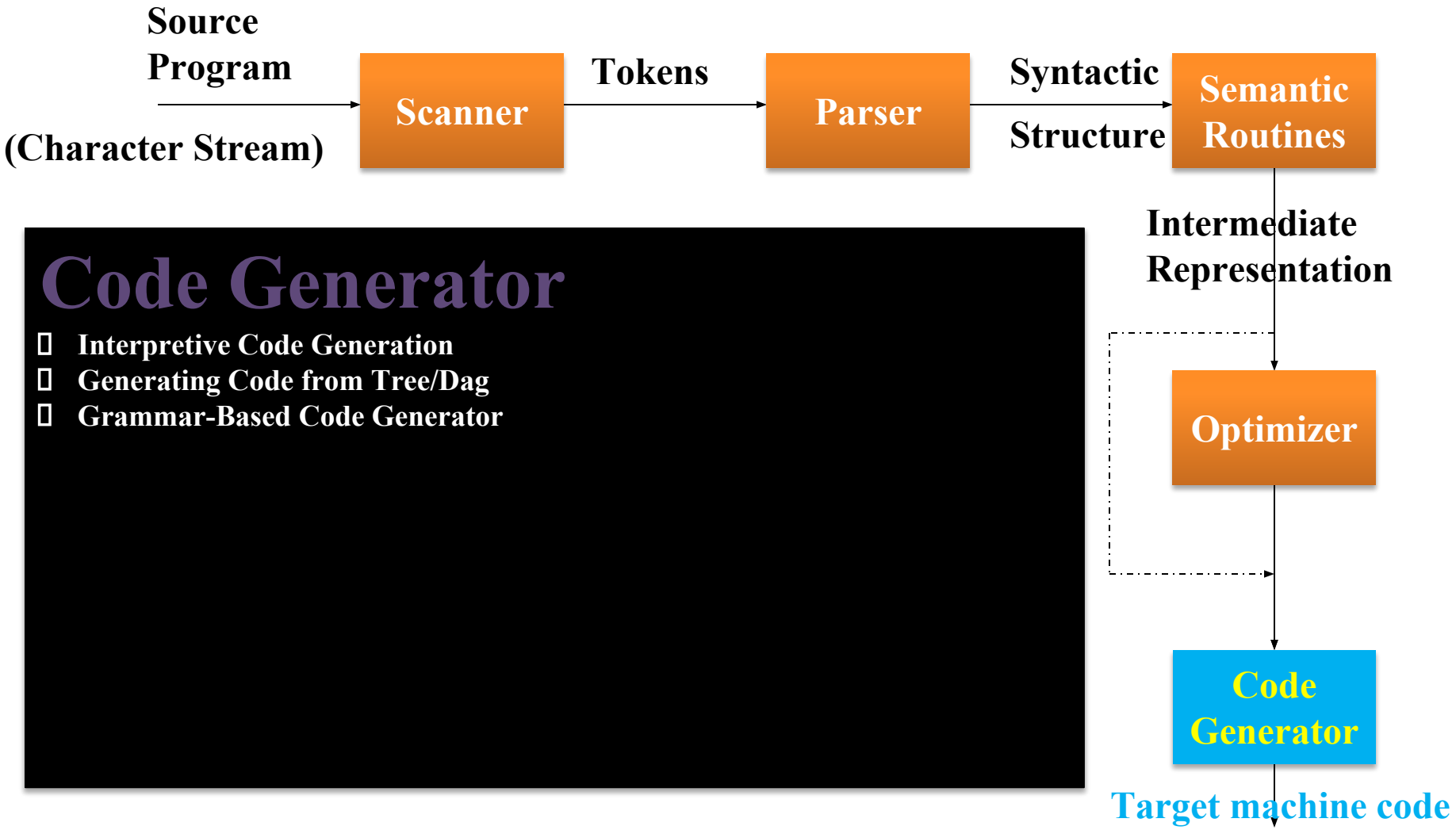
Code optimization

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

Code optimization



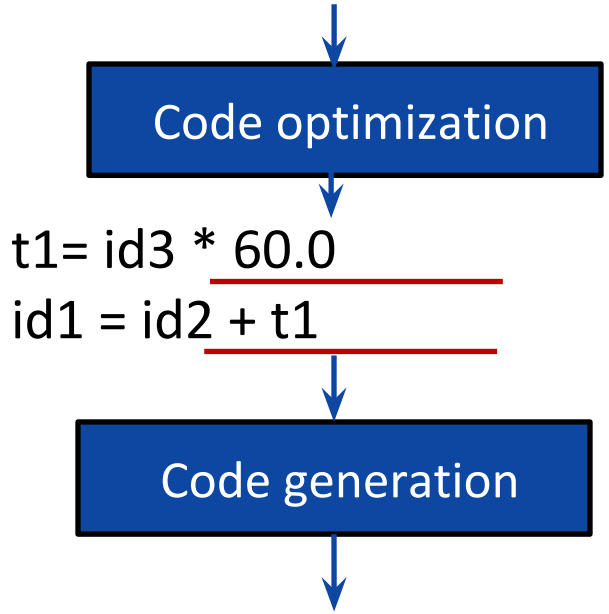
The Structure of a Compiler



Code generation

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

Code optimization



Id3 R2
Id2 R1

Summary

```
position := initial + rate * 60
```

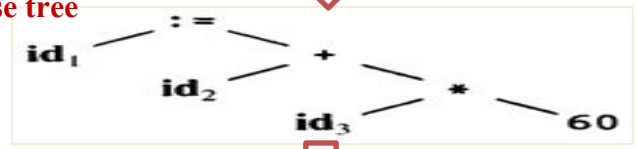
Scanner
[Lexical Analyzer]

Tokens

$id_1 := id_2 + id_3 * 60$

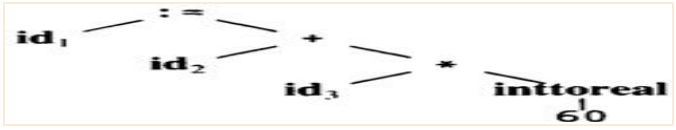
Parser
[Syntax Analyzer]

Parse tree



Semantic Process
[Semantic analyzer]

Abstract Syntax Tree w/ Attributes



SYMBOL TABLE

1	position	···
2	initial	···
3	rate	···
4		

Code Generator
[Intermediate Code Generator]

Non-optimized Intermediate Code

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

Code Optimizer

Optimized Intermediate Code

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

Code Optimizer

Target machine code

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

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.

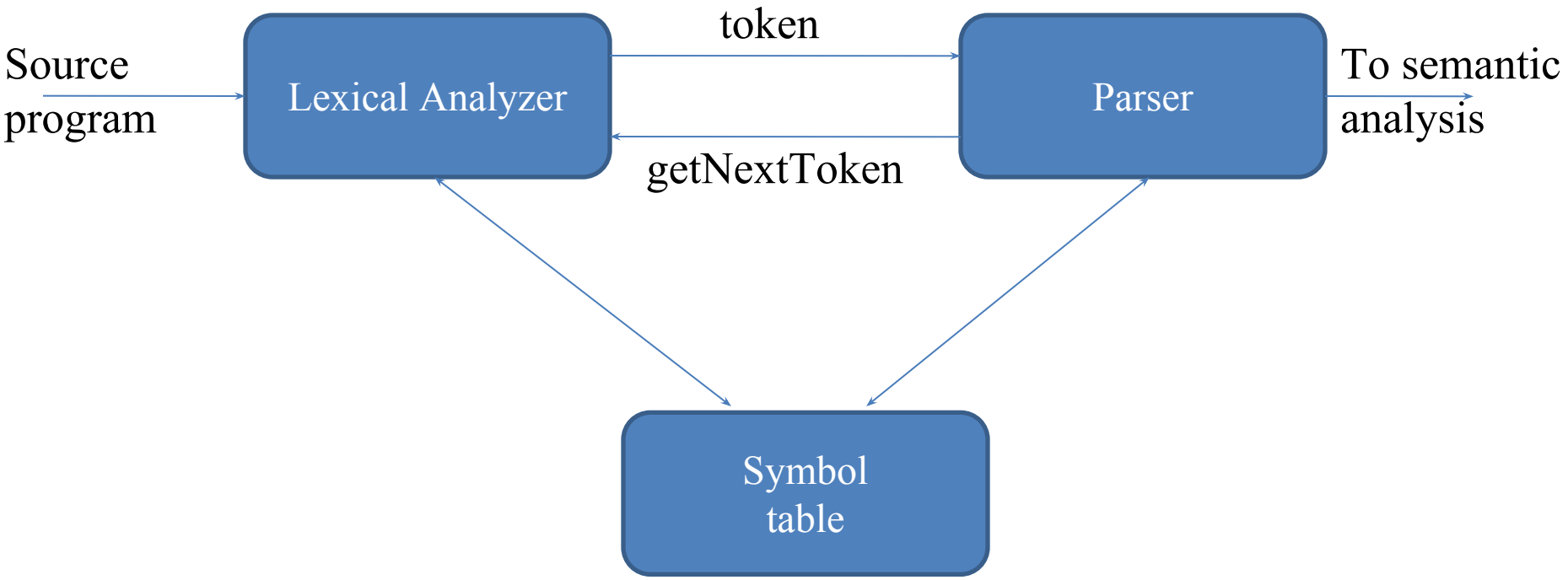
Lexical analyzer

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.

The role of lexical analyzer



Why to separate Lexical analysis and parsing

1. Simplicity of design
2. Improving compiler efficiency
3. Enhancing compiler portability

Tokens

- Sequence of character having a collective meaning is known as token.
- Categories of Tokens:

✔ Identifier
✔ Keyword
✔ Operator
✔ Special symbol
✔ 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

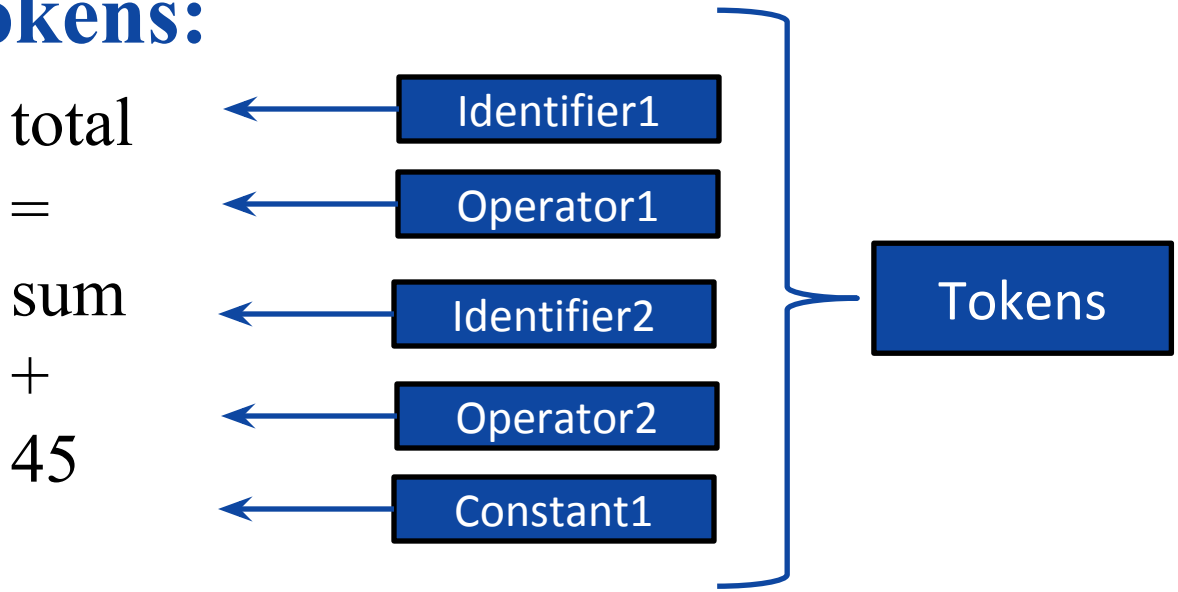
- The sequence of character in a source program matched with a pattern for a token is called lexeme.

- 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

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 tokens

E = M * C ** 2

<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>

Input buffering

- There are mainly two techniques for input buffering:

✔ Buffer pairs

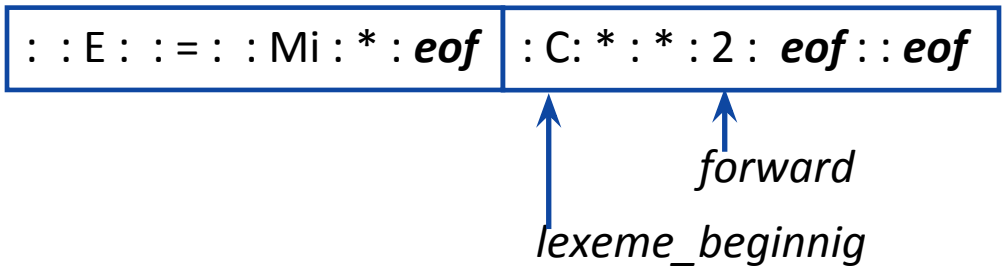
✔ Sentinels

Buffer Pair

- The lexical analysis scans the input string from left to right one character at a time.
- Buffer divided into two N-character halves, where N is the number of character on one disk block.

Sentinels

- In buffer pairs we must check, each time we move the forward pointer that we have not moved off one of the buffers.
- Thus, for each character read, we make two tests.
- We can combine the buffer-end test with the test for the current character.
- We can reduce the two tests to one 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

```
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
```

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 with another character.
- Transpose two adjacent characters

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 t that is respectively proper prefix, suffix or substring of S, such that **s≠t**.
Subsequence of S	A string obtained by removing **zero or more not necessarily contiguous symbol** from S. e.g., baaa is subsequence of banana.

Operations on languages

Operation	Definition
Union of L and M Written L U M	
Concatenation of L and M *Written LM*	
Kleene closure of L Written L^*	
Positive closure of L *Written* L^+	

Regular expression

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

Notational shorthand's

✔ One or more instances: +
✔ Zero or more instances: *
✔ Zero or one instances: ?
✔ Alphabets: Σ

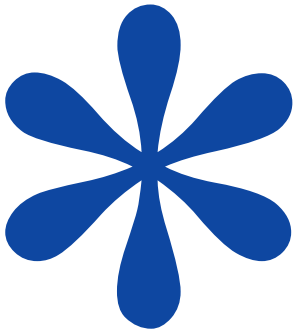
Regular expression

- $\in$ is a regular expression that denotes $\{\in\}$, the set containing empty string.
- If *a* is a symbol in Σ then *a* is a regular expression, $L(a)=\{a\}$
- Suppose *r* and *s* are regular expression denoting the languages *L* (*r*) and *L*(*s*). Then,
- (r)|(s) is a regular expression denoting L(r) U L(s)
- (r)(s) is a regular expression denoting L(r)L(s)
- (r)* is a regular expression denoting (L(r))*
- (r) is a regular expression denoting 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*

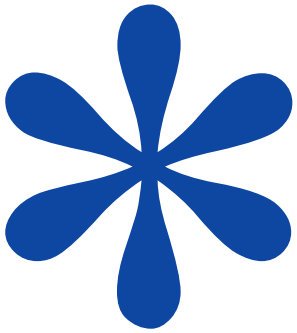


ϵ
a
aa
aaa
aaaa
aaaaa
.....

Infinite
.....

Regular expression

L = Zero or More Occurrences of a = a*



ϵ
a
aa
aaa
aaaa
aaaaa
.....

Infinite
.....

Regular expression

L = One or More Occurrences of a = 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

   R. E. = 0 | 1

2. 0 or 11 or 111

   Strings: 0, 11, 11

   R. E. = 0 | 11 | 111

3. String having zero or more *a*.

   Strings: ϵ, a, aa, a

   R. E. = a*

4. String having one or more *a*.

   Strings: a, aa, aaa

   R. E. = a+

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

   Strings: abc, bca,

   R. E. = (a|b|c) (a|b|c) (a|b|c)

6. All binary string

   Strings: 0, 11, 10

   R. E. = (0 | 1)+

Regular expression examples

7. 0 or more occurrence of either a or b or both
   $Strings: \epsilon, a, aa, a$
   $R.E. = (a \mid b)*$
8. 1 or more occurrence of either a or b or both
   $Strings: a, aa, ab$
   $R.E. = (a \mid b)^+$
9. Binary no. ends with 0
   $Strings: 0, 10, 10$
   $R.E. = (0 \mid 1)^*\, 0$
10. Binary no. ends with 1
    $Strings: 1, 101, 1$
    $R.E. = (0 \mid 1) * 1$
11. Binary no. starts and ends with 1
    $Strings: 11, 101, :$
    $R.E. = 1\,(0 \mid 1) * 1$
12. String starts and ends with same character
    $Strings: 00, 101, c$

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

Regular expression examples

13. All string of a and b starting with a

$R.E. = a(a \mid b)^*$

$Strings: a, ab, aab, abb\ldots$

14. String of 0 and 1 ends with 00

$R.E. = (0 \mid 1) * 00$

$Strings: 00, 100, 000, 1000, 1100\ldots$

15. String ends with abb

$R.E. = (a \mid b) * abb$

$Strings: abb, babb, ababb\ldots$

16. String starts with 1 and ends with 0

$R.E. = 1(0 \mid 1) * 0$

$Strings: 10, 100, 110, 1000, 1100\ldots$

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

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

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

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

Regular expression examples

19. The language with $\Sigma = \{a, b\}$ such that 3rd character from right end of the string is always a.

$\boldsymbol{Strings: aaa, aba, aaba, abb\ldots}$ $\qquad \boldsymbol{R.E. = (a \mid b) * a(a|b)(a|b)}$

20. Any no. of a followed by any no. of b followed by any no. of c

$\boldsymbol{Strings: \epsilon, abc, aabbcc, aabc, abb\ldots}$ $\qquad \boldsymbol{R.E. = a^{*}\, b^{*}\, c^{*}}$

21. String should contain at least three 1

$\boldsymbol{Strings: 111, 01101, 0101110\ldots.}$ $\boldsymbol{R.E. = (0|1)^{*}1\ (0|1)^{*}1\ (0|1)^{*}1\ (0|1)^{*}}$

22. String should contain exactly two 1

$\boldsymbol{Strings: 11, 0101, 1100, 010010, 100100\ldots.}$ $\qquad \boldsymbol{R.E. = 0^{*}10^{*}10^{*}}$

23. Length of string should be at least 1 and at most 3

$\boldsymbol{Strings: 0, 1, 11, 01, 111, 010, 100\ldots.}$

$$\boldsymbol{R.E. = (0|1) \mid (0|1)(0|1) \mid (0|1)(0|1)(0|1)}$$

24. No. of zero should be multiple of 3

$\boldsymbol{Strings: 000, 010101, 110100, 000000, 100010010\ldots.}$

$$\boldsymbol{R.E. = (1^{*}01^{*}01^{*}01^{*})^{*}}$$

Regular expression examples

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

Strings*: *aaa, baaa

$$R.E. = ((b|c)^* a (b|c)^* a (b|c)^* a (b|c)^*)^*$$

26. Even no. of 0

***Strings*: 00, 0101, 0000, 100100….**

$$R.E. = (1^*01^*01^*)^*$$

27. String should have odd length

***Strings*: 0, 010, 110, 000, 10010….**

$$R.E. = (0|1)\ ((0|1)(0|1))^*$$

28. String should have even length

***Strings*: 00, 0101, 0000, 100100….**

$$R.E. = ((0|1)(0|1))^*$$

29. String start with 0 and has odd length

***Strings*: 0, 010, 010, 000, 00010….**

$$R.E. = (0)\ ((0|1)(0|1))^*$$

30. String start with 1 and has even length

***Strings*: 10, 1100, 1000, 100100….**

$$R.E. = 1(0|1)((0|1)(0|1))^*$$

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:

$$d1 \rightarrow r1$$
$$d2 \rightarrow r2$$
$$\ldots\ldots$$
$$dn \rightarrow rn$$

Where *di* is a distinct name & *ri* is a regular expression.

- Example: Regular definition for identifier

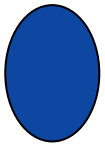
letter → A|B|C|……..|Z|a|b|……..|z

digit → 0|1|…….|9|

id → letter (letter | digit)*

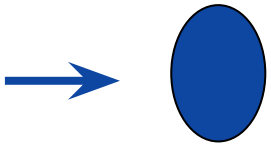
Transition diagrams

A stylized flowchart is called transition diagram.



is a state

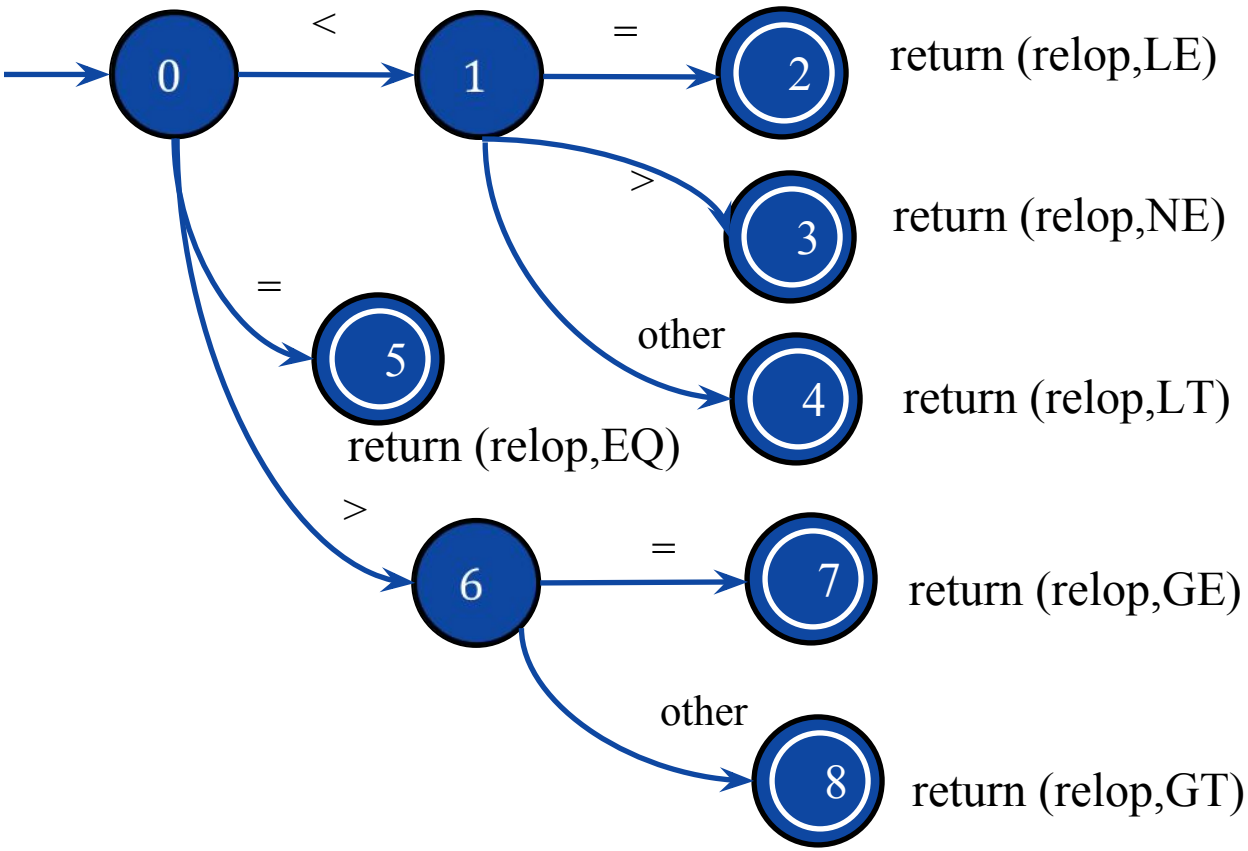
is a transition



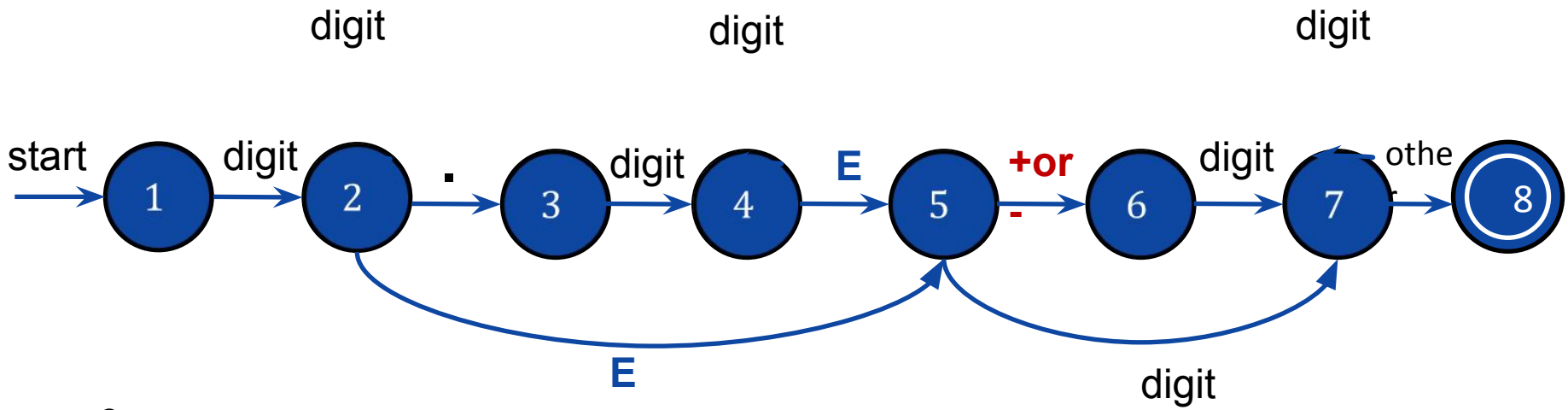
is a start state

is a final state

Transition Diagram : Relational operator



Transition diagram : Unsigned number



3

5280

39.37

1.894 **E** - 4

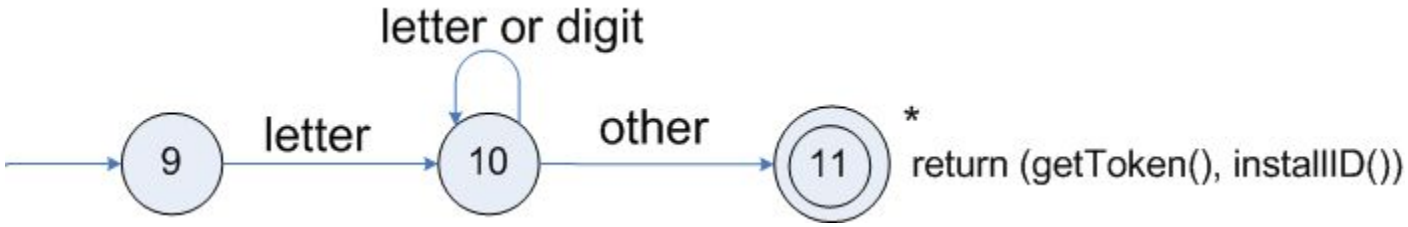
2.56 **E** + 7

45 **E** + 6

96 **E** 2

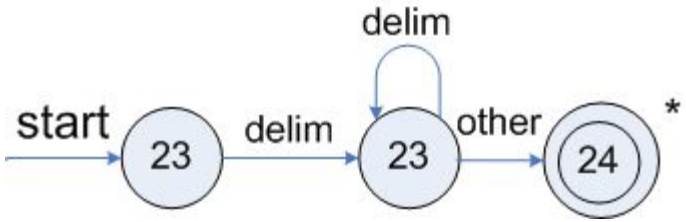
Transition diagrams (cont.)

Transition diagram for reserved words and identifiers

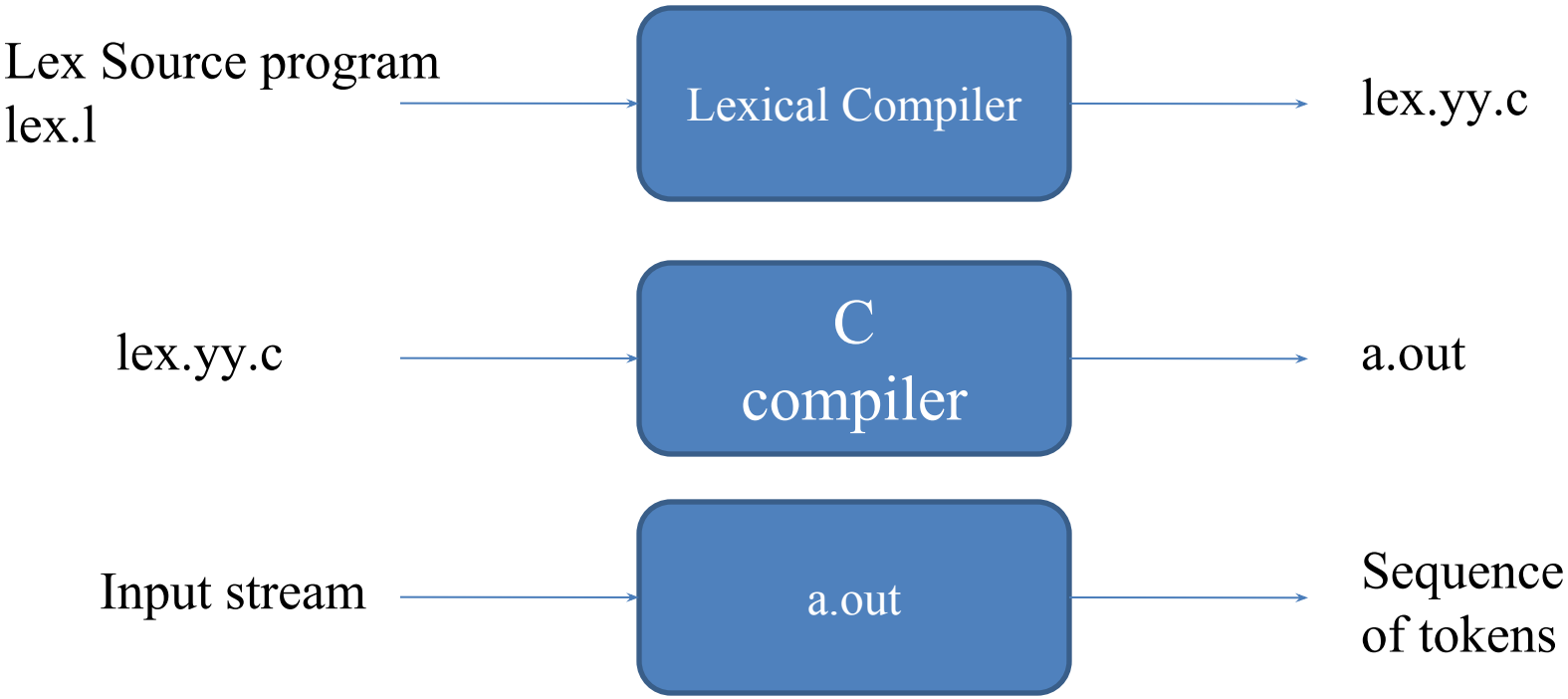


Transition diagrams (cont.)

Transition diagram for whitespace



Lexical Analyzer Generator - Lex



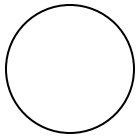
Finite Automata

- Finite Automata are recognizers.
  - ✓ FA simply say "Yes" or "No" about each possible input string.

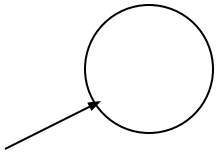
- Finite Automata is a mathematical model consist of:
  1. Set of states ***S***
  2. Set of input symbol ***Σ***
  3. A transition function ***move***
  4. Initial state S_0
  5. Final states or accepting states **F**

Finite Automata State Graphs

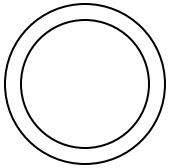
A state



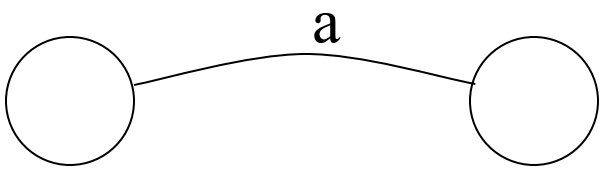
- The start state



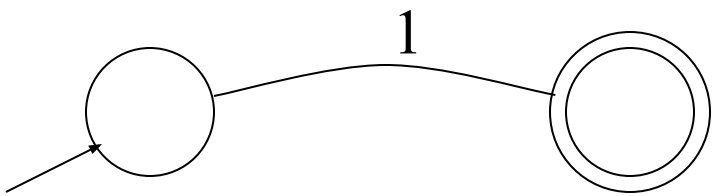
- An accepting state



- A transition



A Simple Example

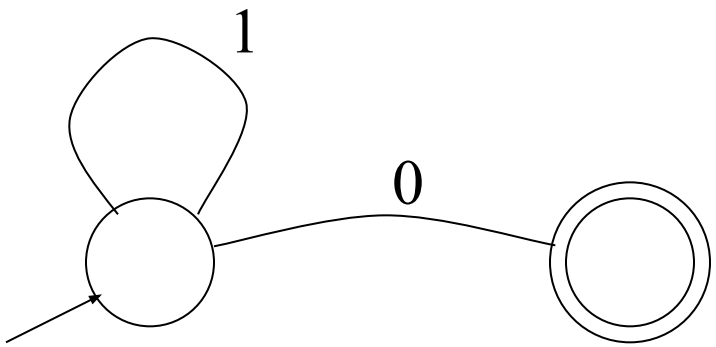


- A finite automaton that accepts only “1”
- A finite automaton accepts a string if we can follow transitions labeled with the characters in the string from the start to some accepting state

Another Simple Example

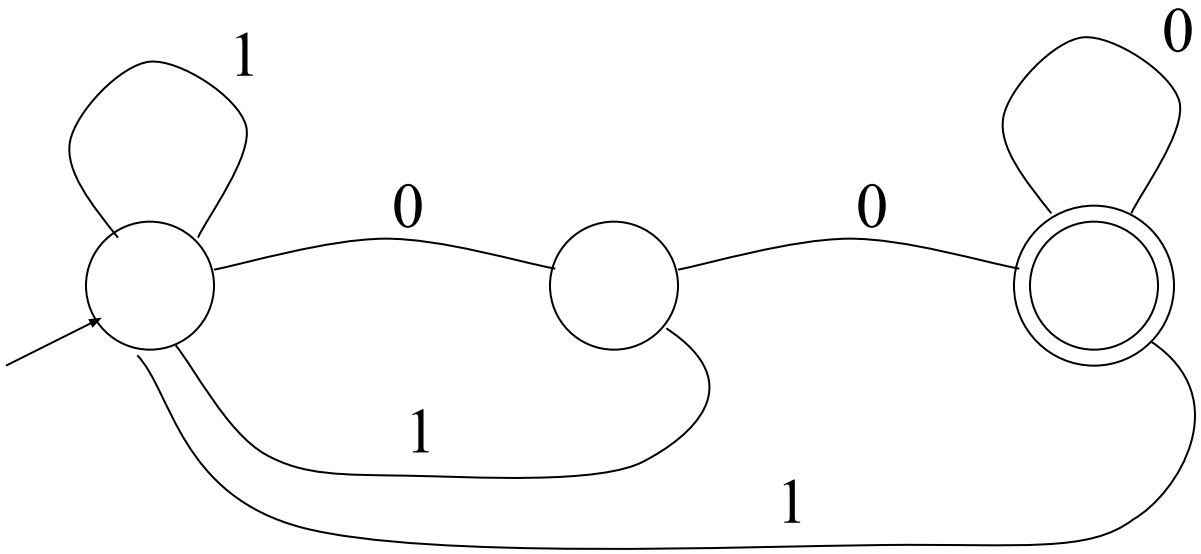
A finite automaton accepting any number of 1's followed by a single 0

Alphabet: {0,1}



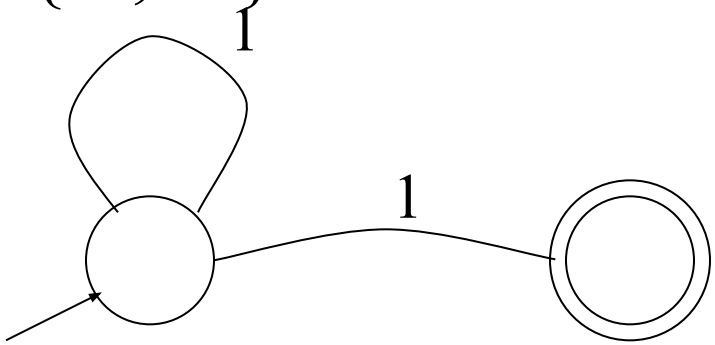
Another Example

Alphabet {0,1}
What language does this recognize?



Another Example

Alphabet still { 0, 1 }



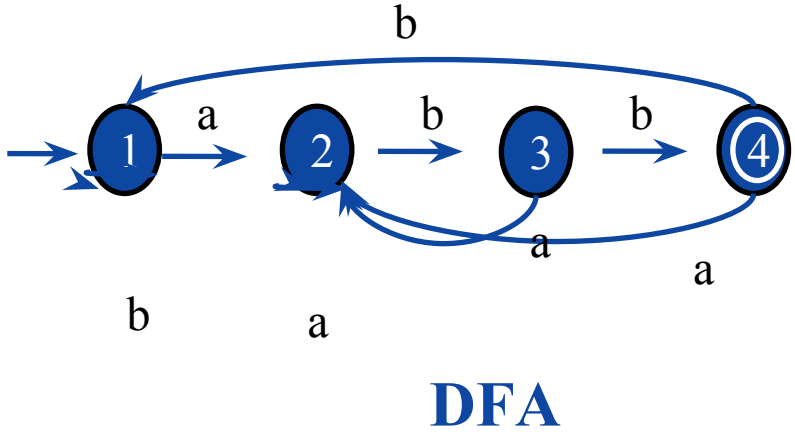
The operation of the automaton is not completely defined by the input

On input “11” the automaton could be in either state

Types of finite automata

DFA

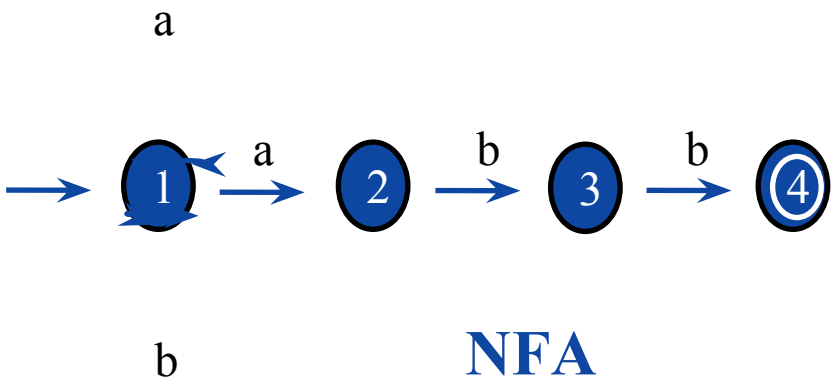
- Deterministic finite automata (DFA): have for each state exactly one edge leaving out for each symbol.



DFA

NFA

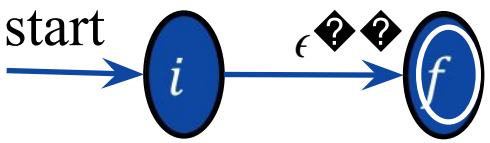
- Nondeterministic finite automata (NFA): There are no restrictions on the edges leaving a state. There can be several with the same symbol as label and some edges can be labeled with ϵ.



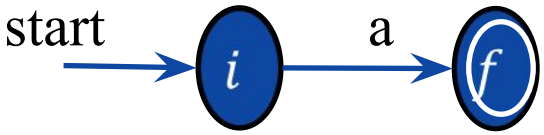
NFA

Regular expression to NFA using Thompson's rule

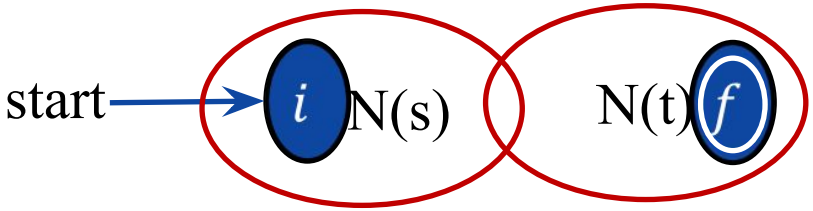
1. For ∈ , construct the NFA



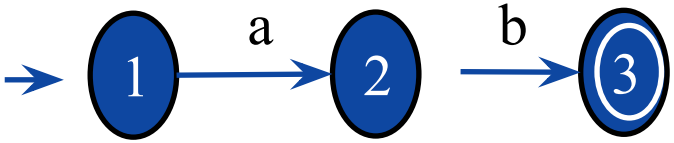
2. For *a* in Σ, construct the NFA



3. For regular expression *st*

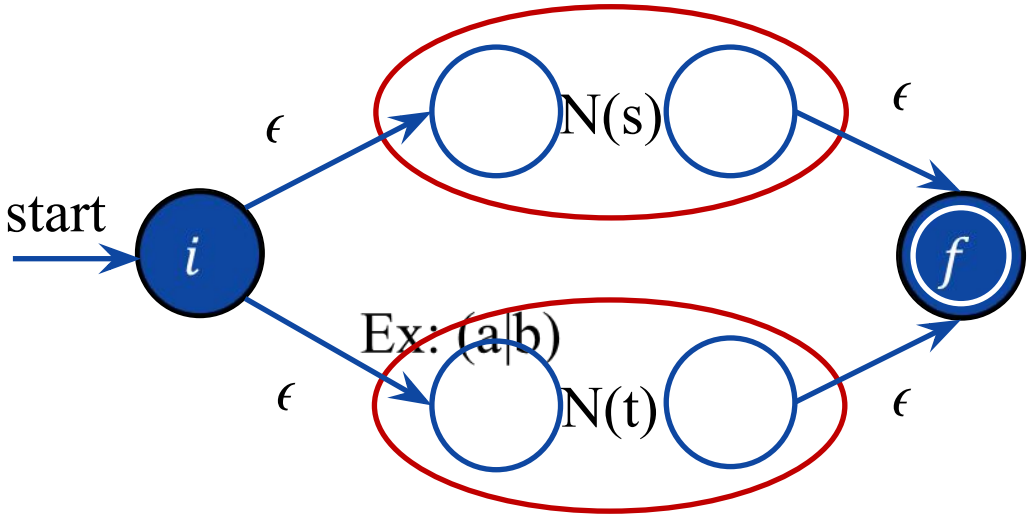


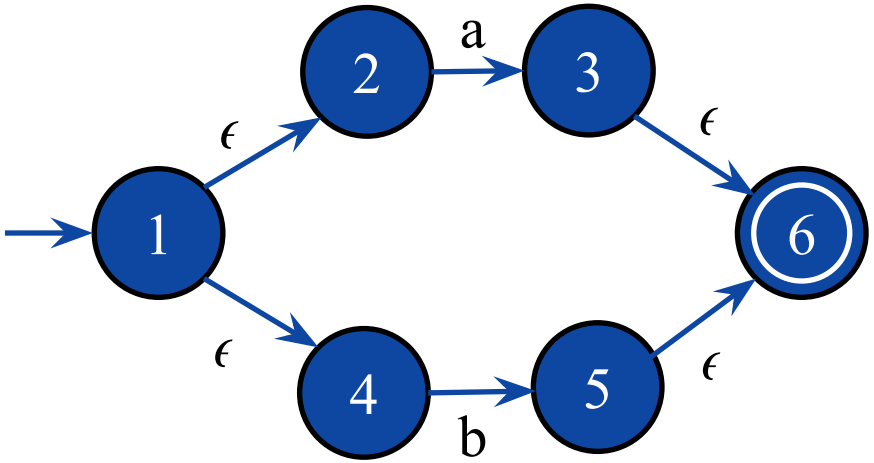
Ex: ab



Regular expression to NFA using Thompson's rule

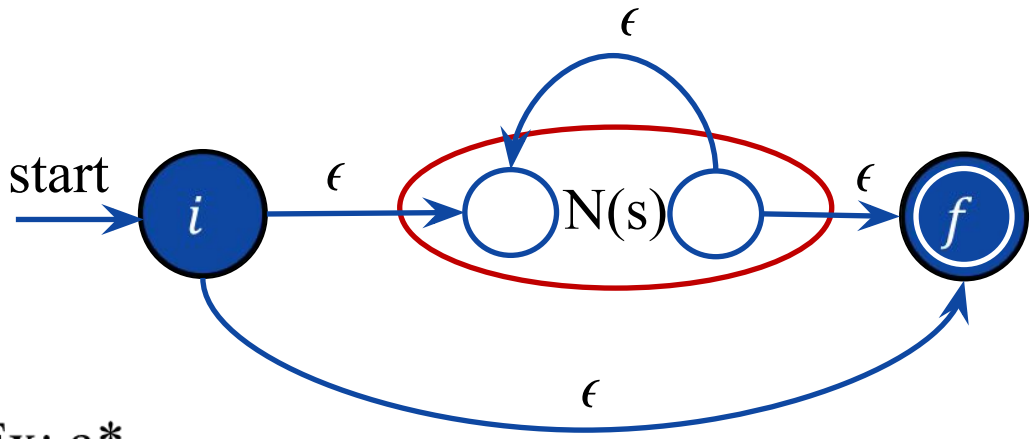
4. For regular expression $s|t$



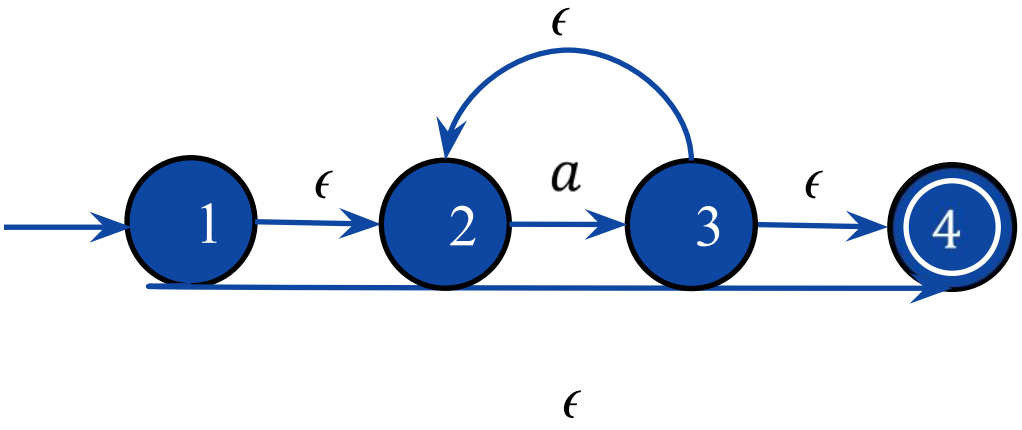


Regular expression to NFA using Thompson's rule

5. For regular expression s^*

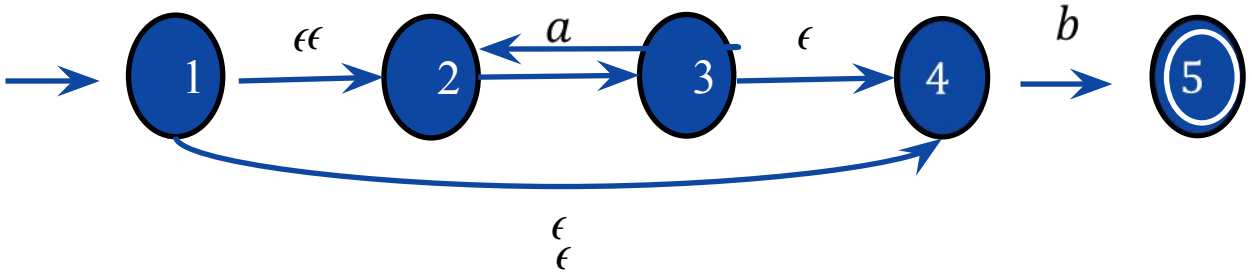


Ex: a*

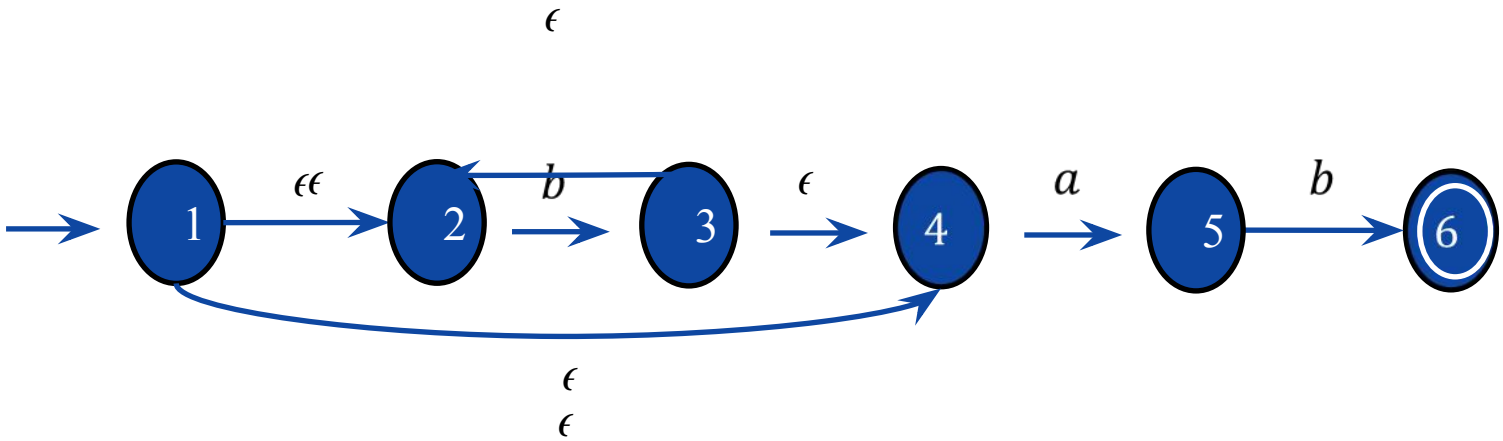


Regular expression to NFA using Thompson's rule

a*b



b*ab



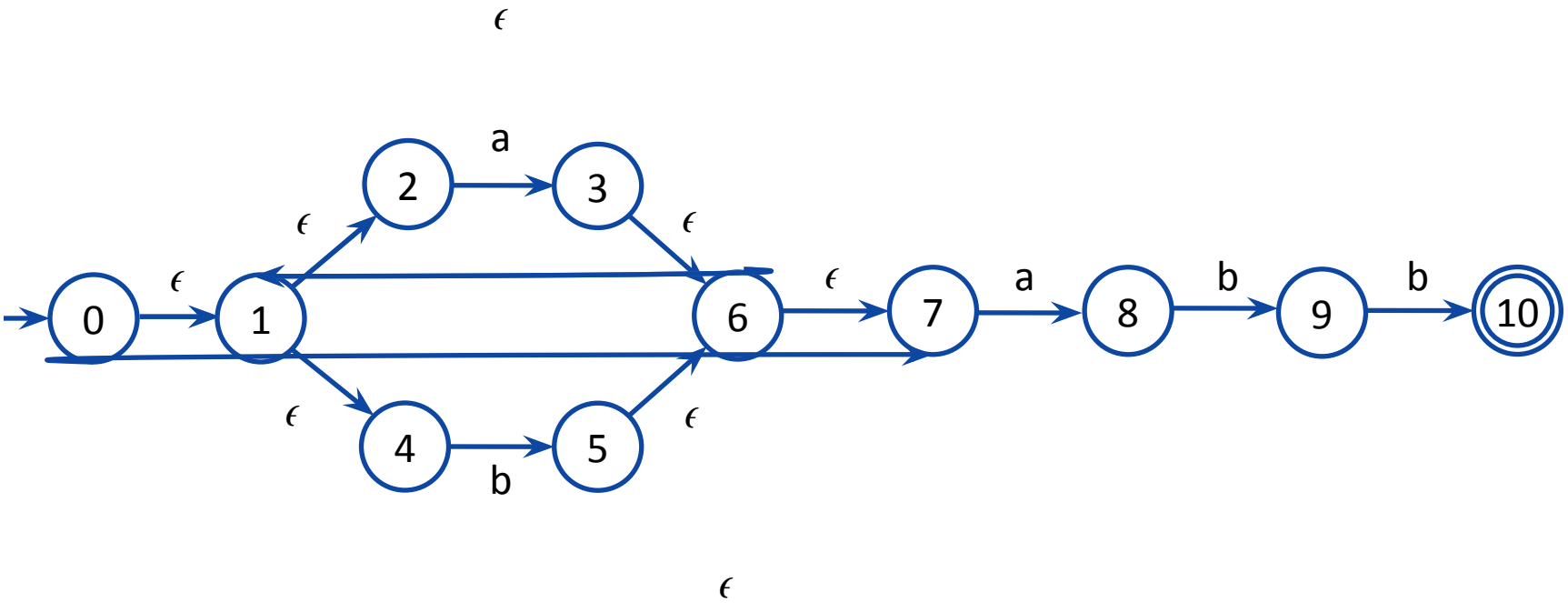
Regular expression to NFA using Thompson's rule

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)*

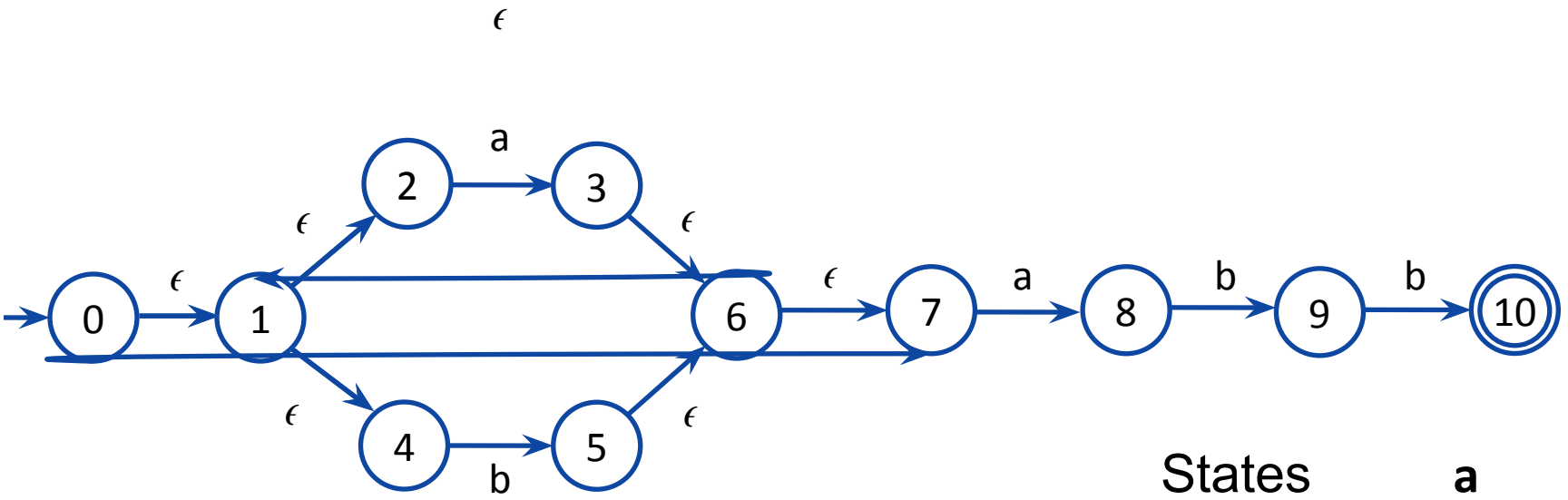
Conversion from NFA to DFA



ϵ- Closure(0)=

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

Conversion from NFA to DFA



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

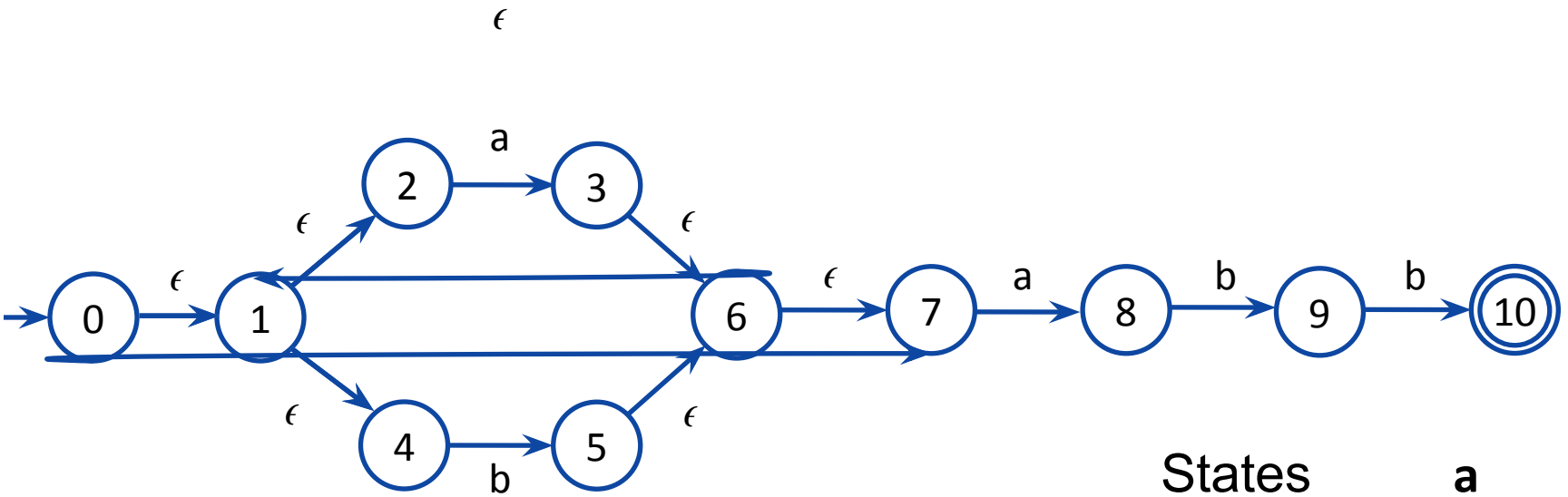
A= {0, 1, 2, 4, 7}

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

ϵ- Closure(Move(A,a))

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

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}

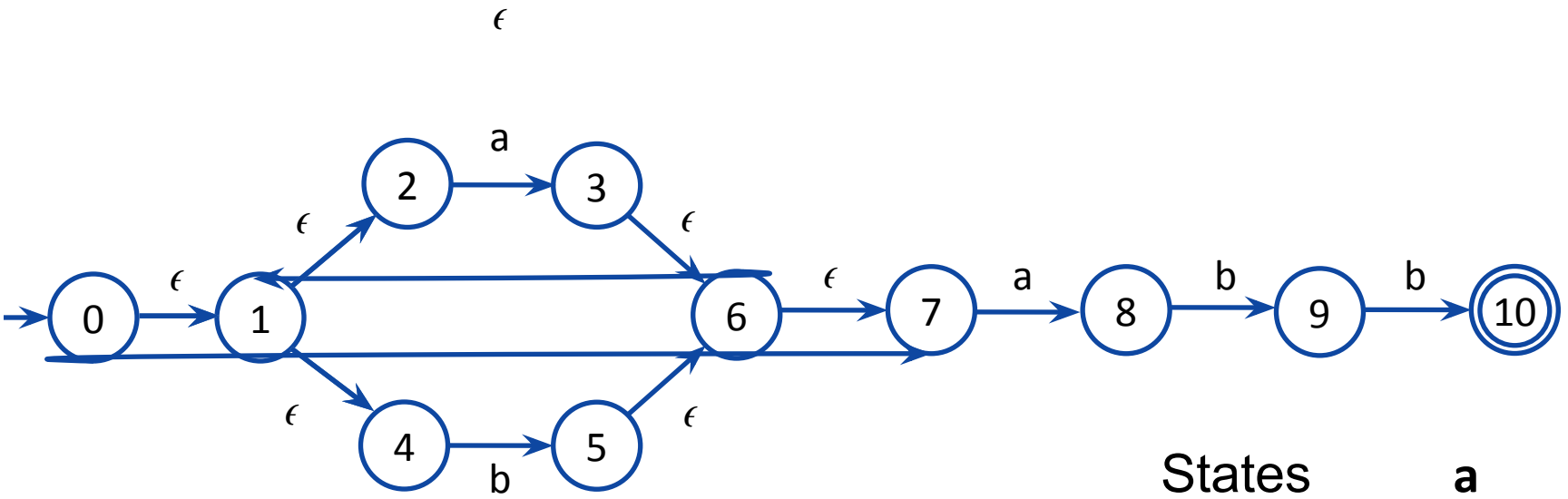
A= {0, 1, 2, 4, 7}

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

ϵ

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

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

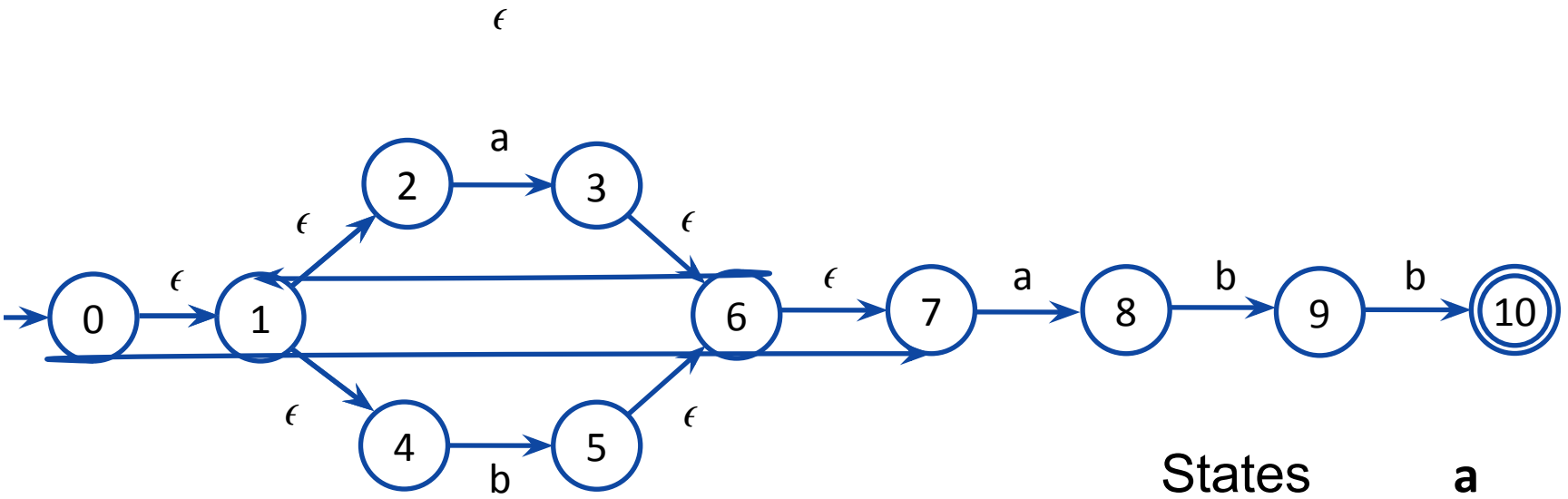
B = {1, 2, 3, 4, 6, 7, 8}

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}**		

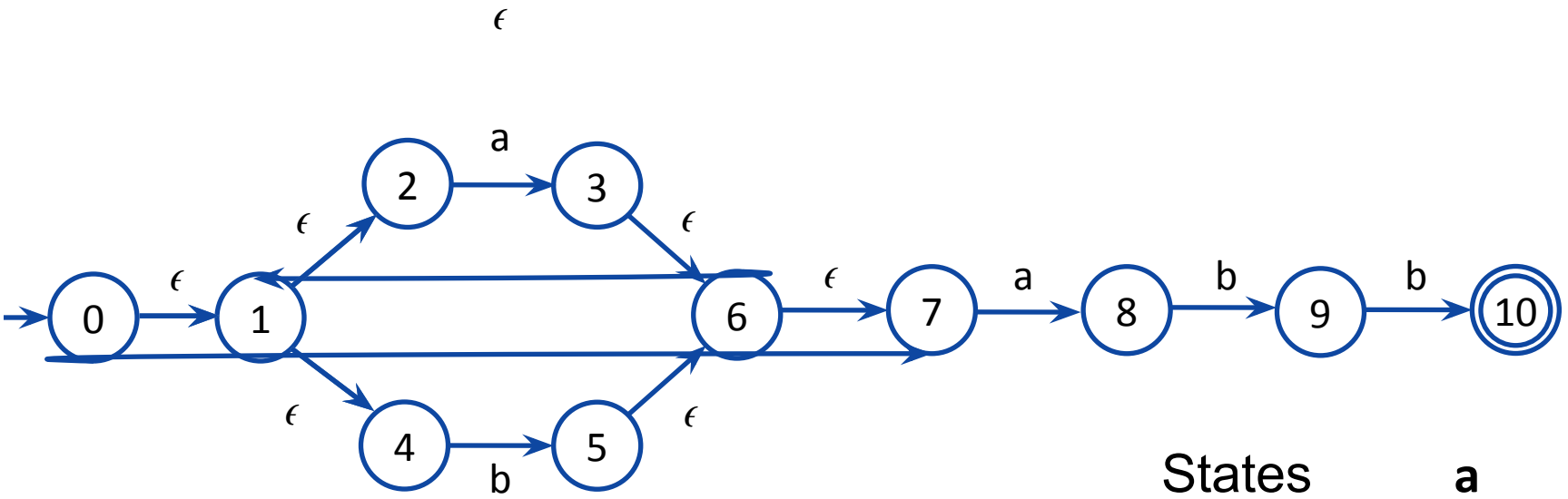
B = {1, 2, 3, 4, 6, 7, 8}

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}**		

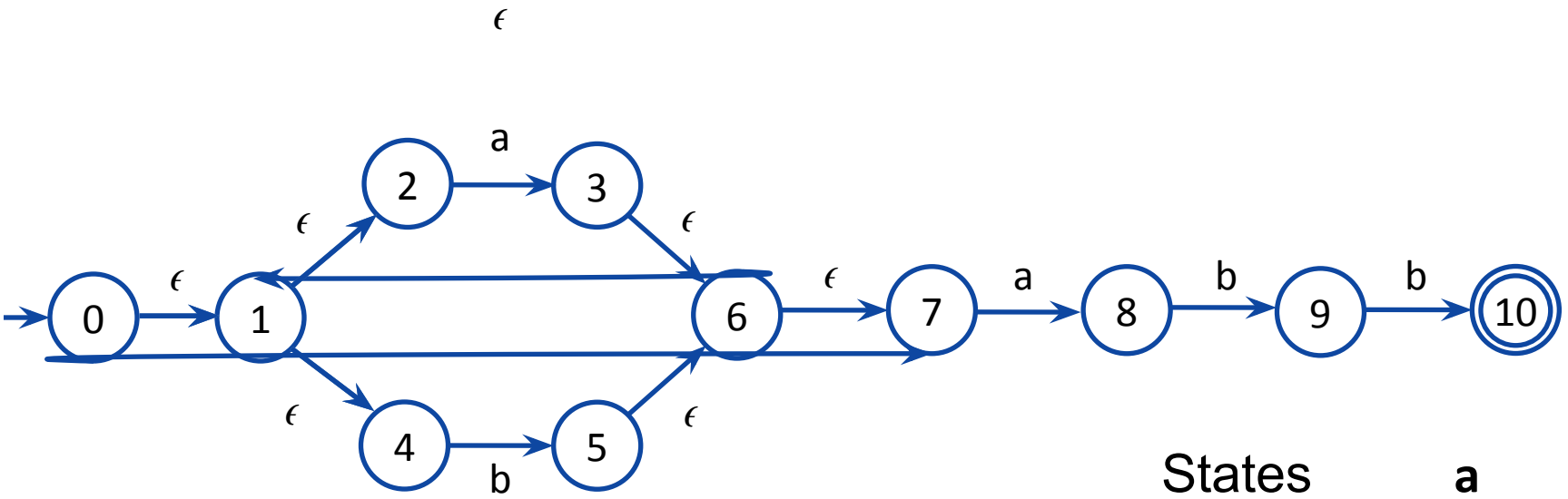
C= {1, 2, 4, 5, 6 ,7}

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}**		

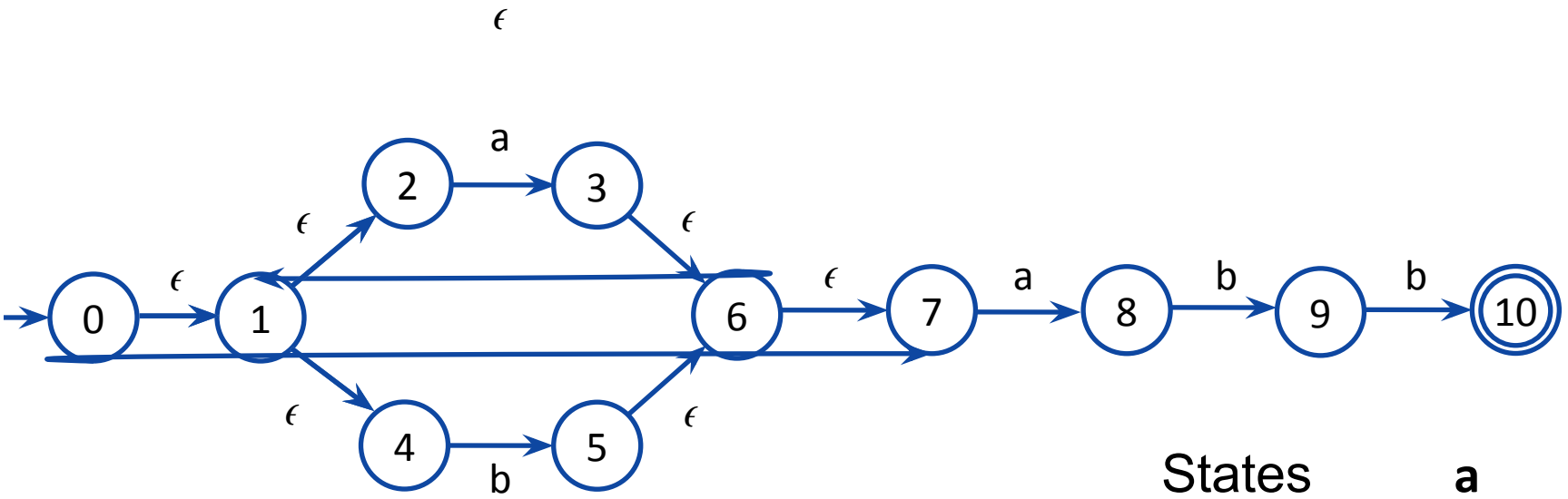
C= {1, 2, 4, 5, 6 ,7}

Move(C,a) =

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}

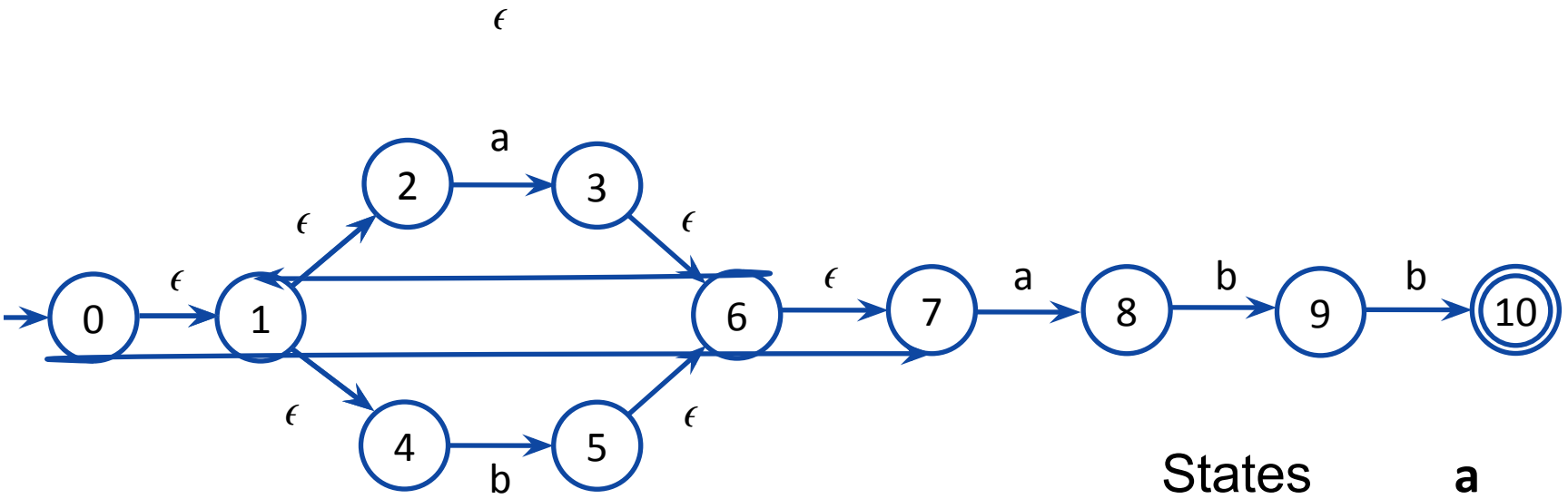
Move(D,a) = {3,8}

ϵ- Closure(Move(D,a))

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

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

Conversion from NFA to DFA



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

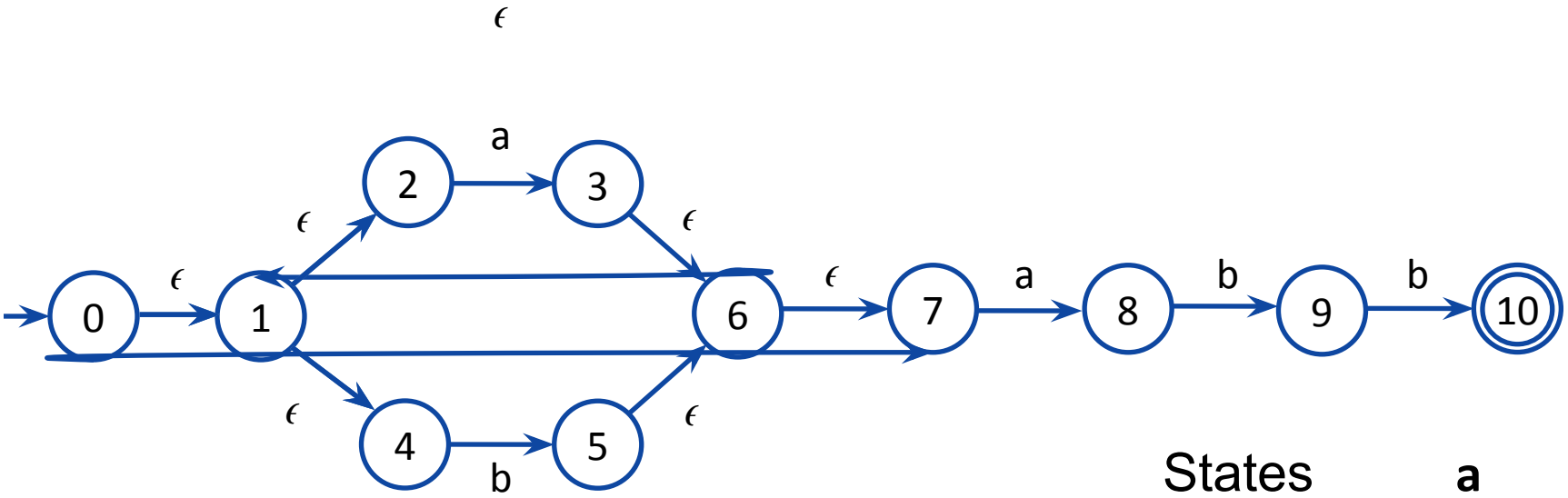
Move(D,b) = {5,10}

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



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

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

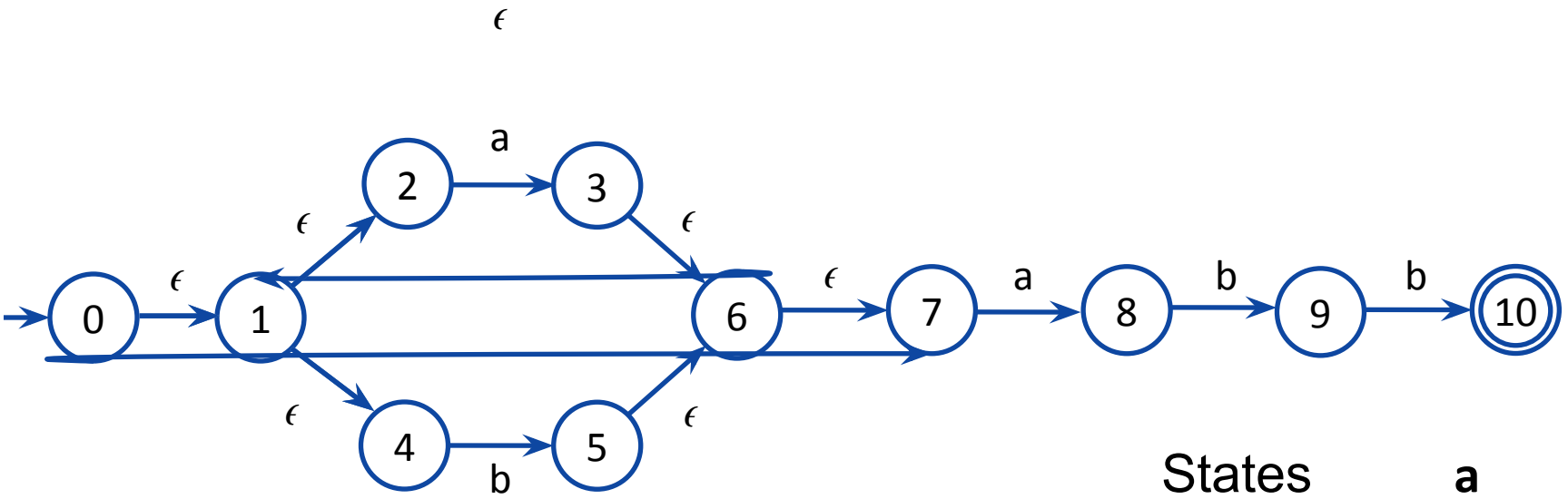
ϵ- Closure(Move(E,a))

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

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}

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) =

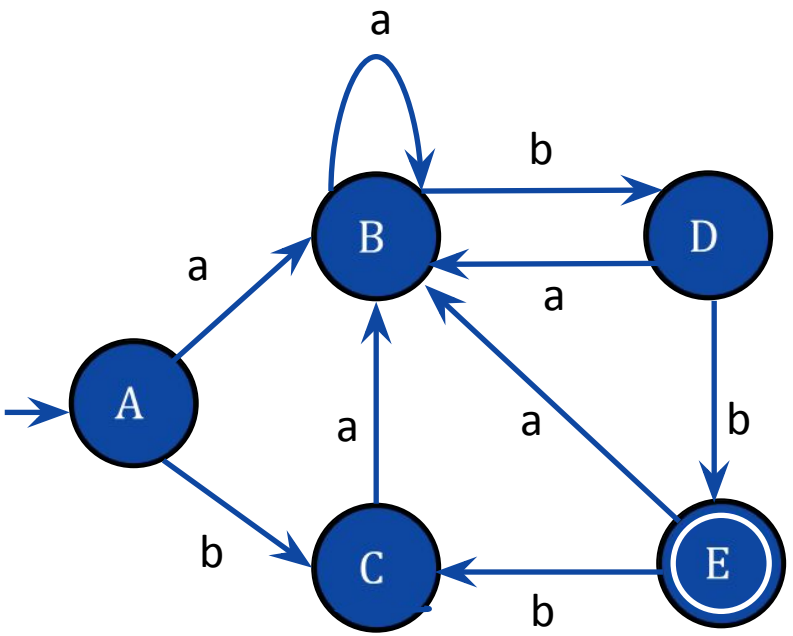
ϵ- Closure(Move(E,b))

= {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 the following regular expression to DFA using subset construction method:

1. (a+b)*a(a+b)

2. (a+b)*ab*a