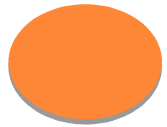
COMPILER DESIGN

UNIT 1

PART 2

COMPILER CONSTRUCTION TOOLS

- DEVELOPER MAY USE MODERN SOFTWARE DEVELOPMENT ENVIRONMENT CONTAINING TOOLS Ex: LANGUAGE EDITORS, VERSION MANAGERS, TEC.
- SOME SPECIAL TOOLS CAN ALSO BE USED. THESE TOOLS ARE THOSE WHICH HIDE THE DETAILS OF GENERATION ALGORITHM AND PRODUCE COMPONENTS THAT CAN BE EASILY INTEGRATED TO REMAINDER OF THE COMPILER.
  - SCANNER GENERATOR
  - PARSER GENERATOR
  - SYNTAX DIRECTED TRANSLATION ENGINE
  - CODE GENERATOR GENERATOR
  - DATA FLOW ANALYSIS ENGINES (FOR OPTIMIZATION)
  - COMPILER CONSTRUCTION TOOLKIT.



SINGLE PASS AND MULTI-PASS COMPILER

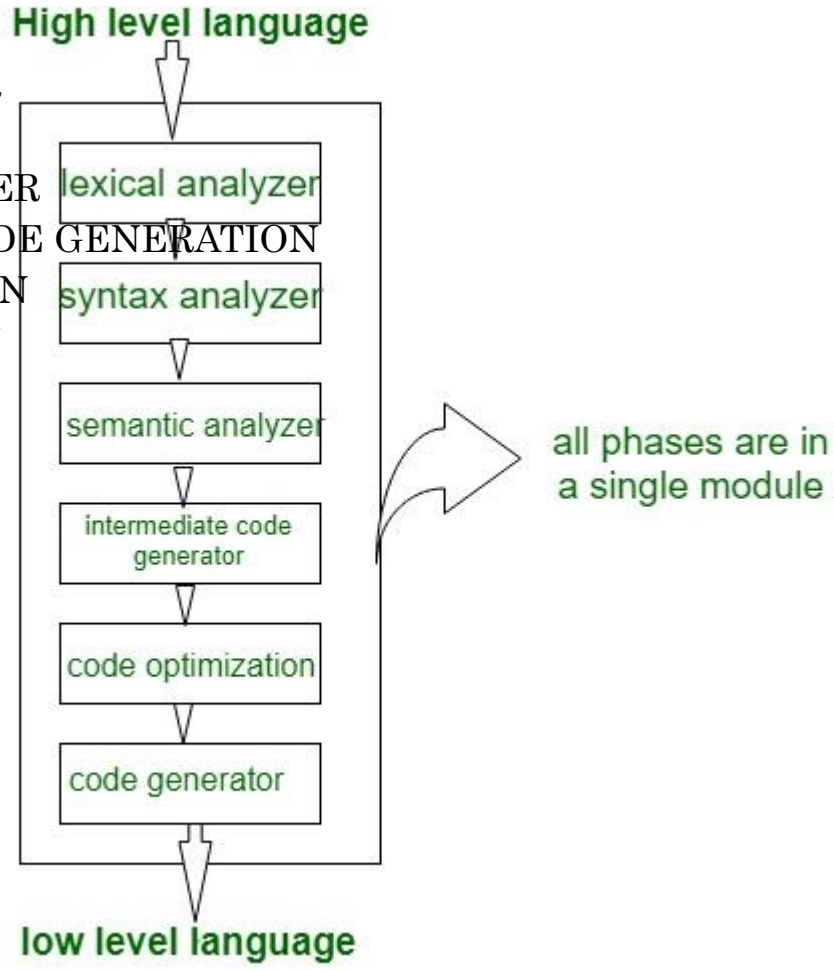
- A **Compiler** pass refers to the traversal of a compiler through the entire program.
- On behalf of passes compiler can be categorized as—
- Single paas compiler
- 2 paas or multi paas compiler

- If we combine or group all the phases of compiler design in a single module known as single pass compiler.
- A Two pass/multi-pass Compiler is a type of compiler that processes the source code or abstract syntax tree of a program multiple times. In multipass Compiler we divide phases in two pass

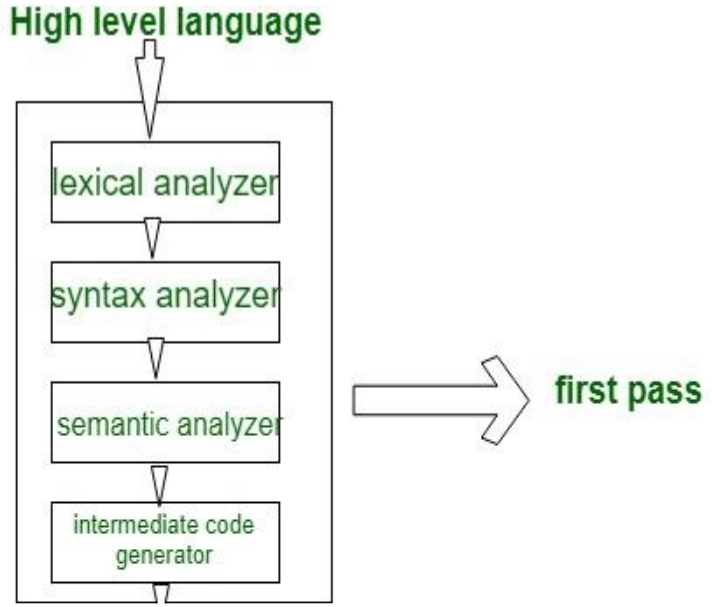
Pragya Gaur

SINGLE PAAS COMPILE COMPILER

1 LEXICAL ANALYZER
2 SYNTAX ANALYZER
3 SEMANTIC ANALYZER
4 INTERMEDIATE CODE GENERATION
5 CODE OPTIMIZATION
6 CODE GENERATION



MULTI PAAS



BOOTSTRAPPING

- THREE LANGUAGES ARE ASSOCIATED WITH A COMPILER

S- LANGUAGE IN WHICH COMPILER IS WRITTEN
A-SOURCE LANGUAGE

B- TARGET LANGUAGE

THE LANGUAGE OF COMPILER AND THE TARGET LANGUAGE ARE USUALLY THE SAME (ON WHICH COMPILER IS WORKING)

Pragya Gaur

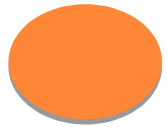
BOOTSTRAPPING

† IF THE COMPILER IS WRITTEN IN IT'S OWN LANGUAGE THEN THE PROBLEM ARIES "HOW TO COMPILE THE FIRST COMPILER".

† HERE COMES THE CONCEPT OF BOOTSTRAPPING.

"THE PROCESS OF MAKING A LANGUAGE AVAILABLE ON A MACHINE IS CALLED BOOTSTRAPPING"

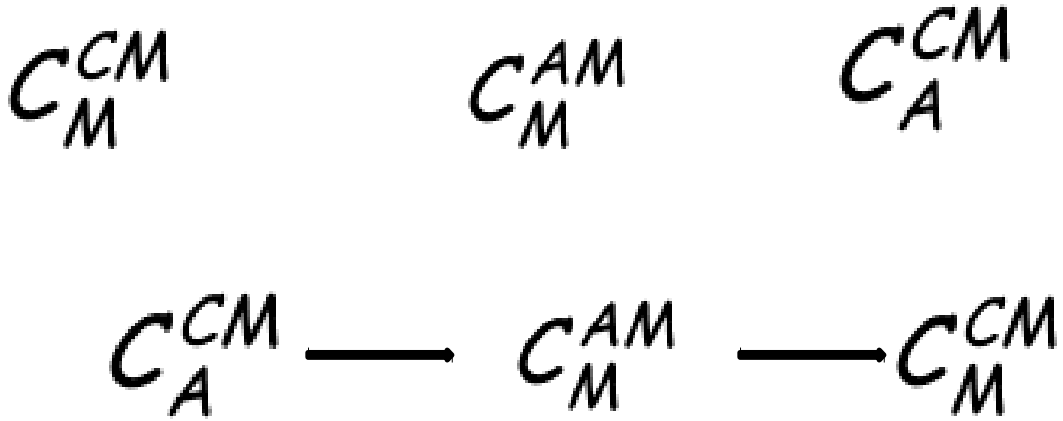
Pragya Gaur



BOOTSTRAPPING LANGUAGE S ON MACHINE A

EXAMPLE

TO MAKE LANGUAGE C AVAILABLE ON MACHINE M

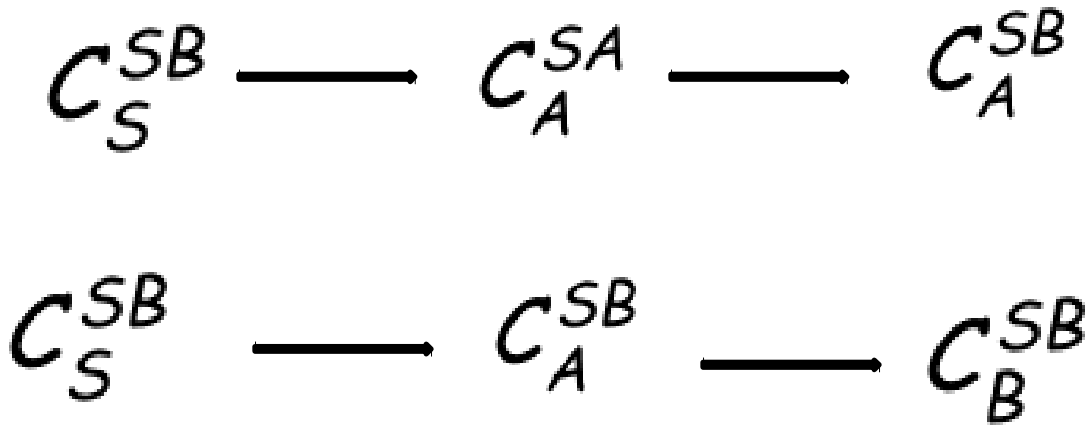


Pragya Gaur

BOOTSTRAPPING LANGUAGE S ON MACHINE A

R- Subset language
S- Source language
A- Language of machine A

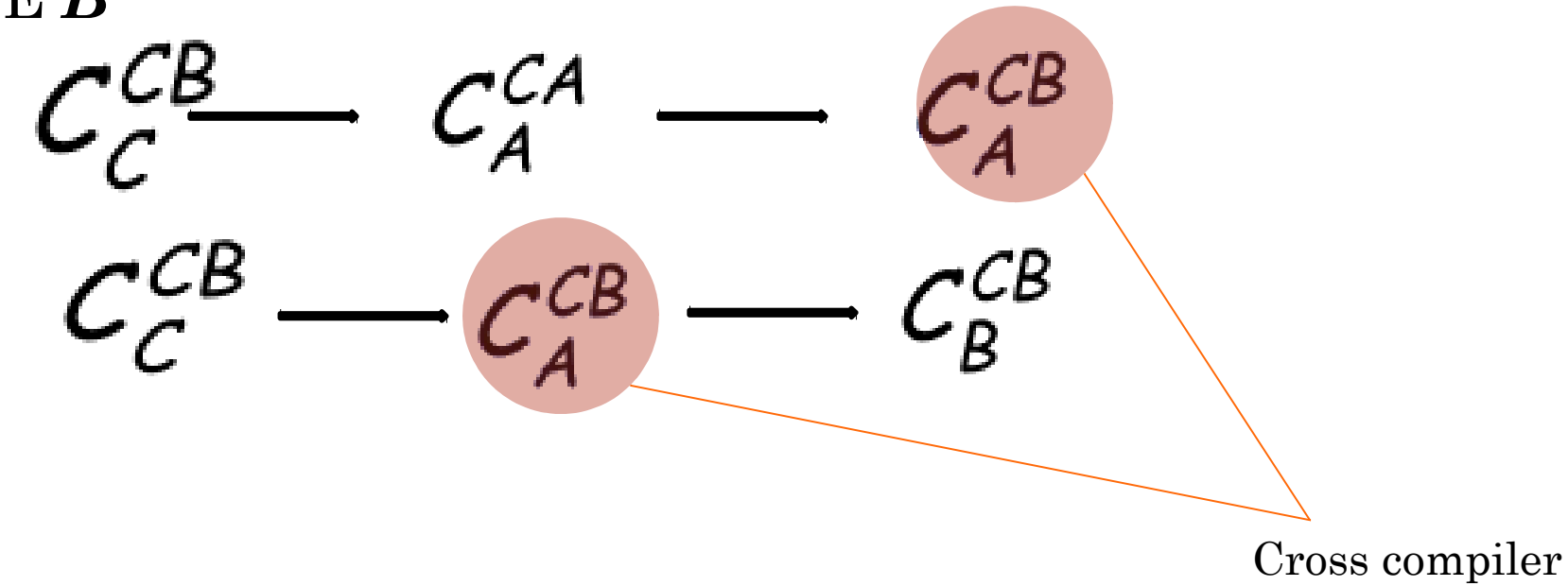
BOOTSTRAPPINGA COMPILERFOR A SECOND MACHINE B WITH THE HELP OF COMPILER FOR MACHINE A



Pragya Gaur

EXAMPLE

- SUPPOSE WE HAVE COMPILER OF LANGUAGE ***C*** FOR MACHINE ***A*** AND WE WANT TO BOOTSTRAPTHIS FOR MACHINE ***B***



Pragya Gaur

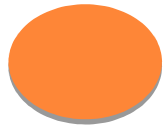
LEXICAL ANALYSIS

Lexical analysis is the first phase of a compiler.

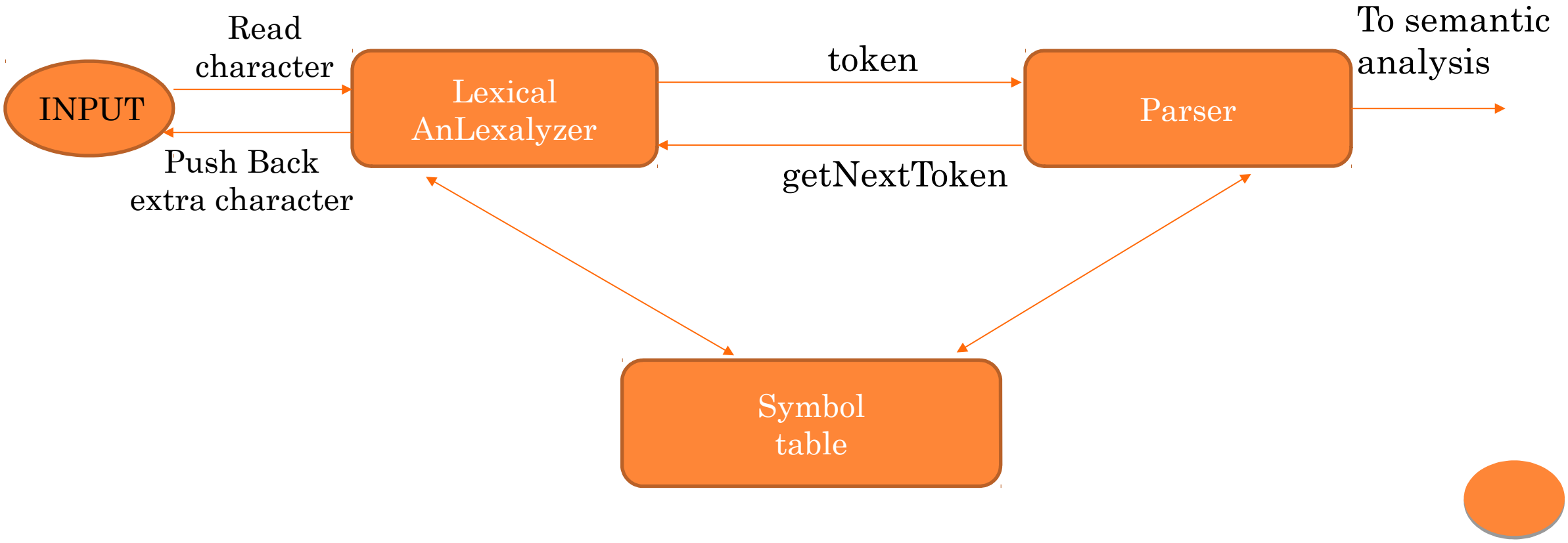
It takes the modified source code from language preprocessors that are written in the form of sentences.

The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

Pragya Gaur

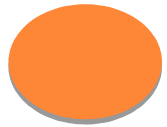


THE ROLE OF LEXICAL ANALYZER



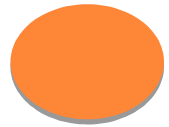
WHY TO SEPARATE LEXICAL ANALYSIS AND PARSING

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



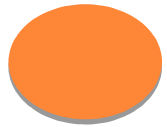
TOKENS, PATTERNS AND LEXEMES

- A token is a pair a token name and an optional token value
- A pattern is a description of the form that the lexemes of a token may take
- A lexeme is a sequence of characters in the source program that matches the pattern for a token

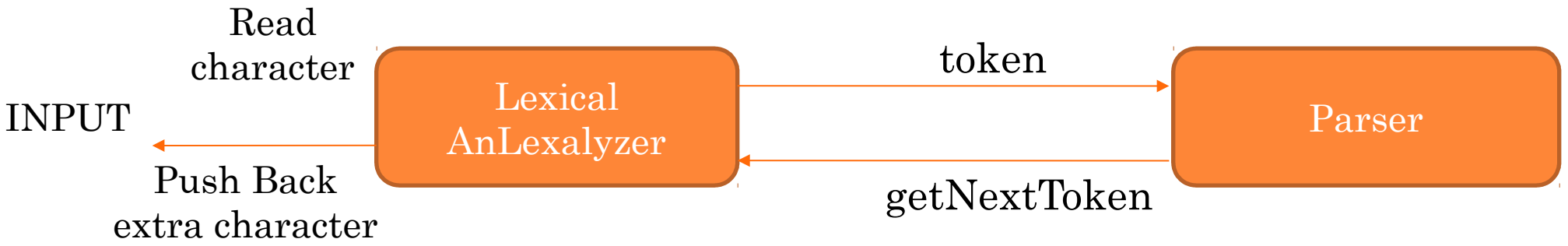


EXAMPLE

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"

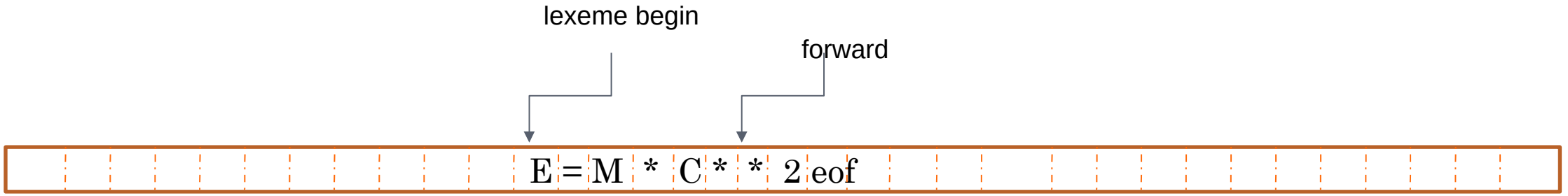


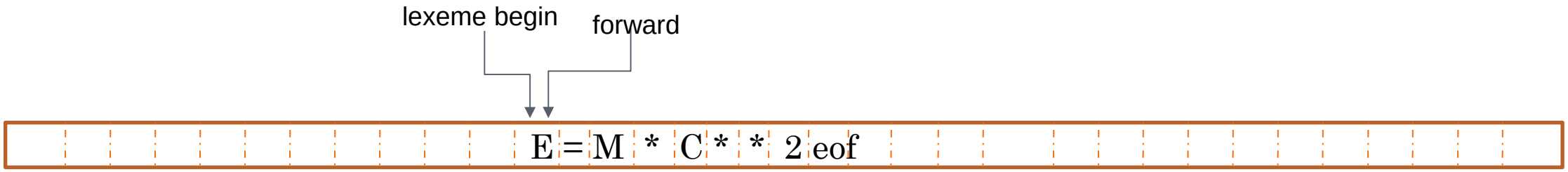
INPUT BUFFERING

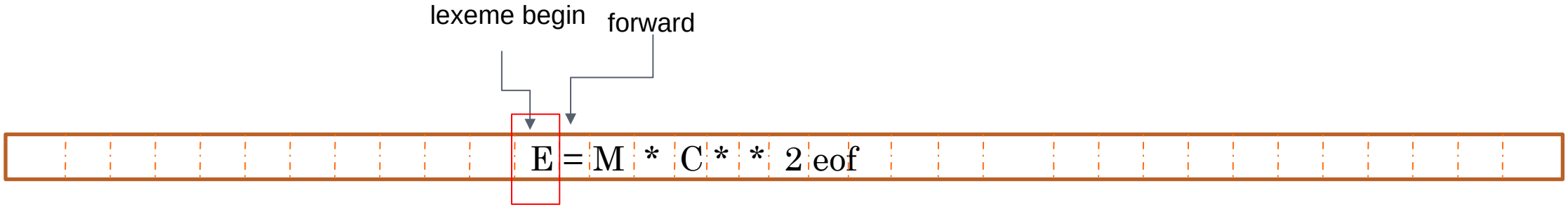


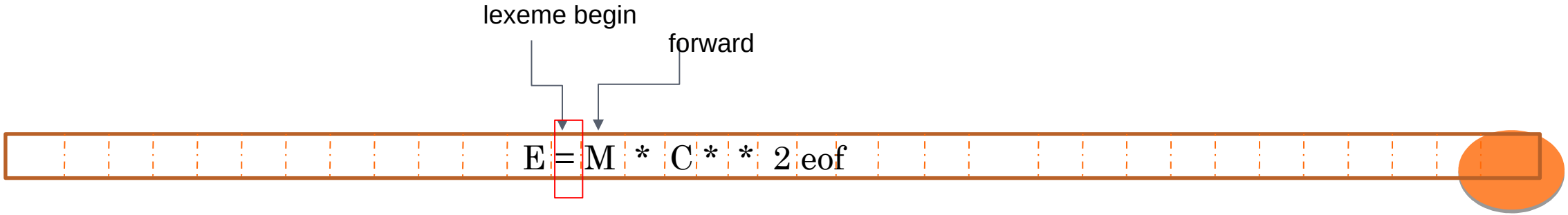
- Sometimes lexical analyzer needs to look ahead some symbols to decide about the token to return
- These extra characters has to be pushed back to the input becouse it can be begining of the next lexeme.
- The implementation of reading and pushing back char is usually done by setting up an input buffer.

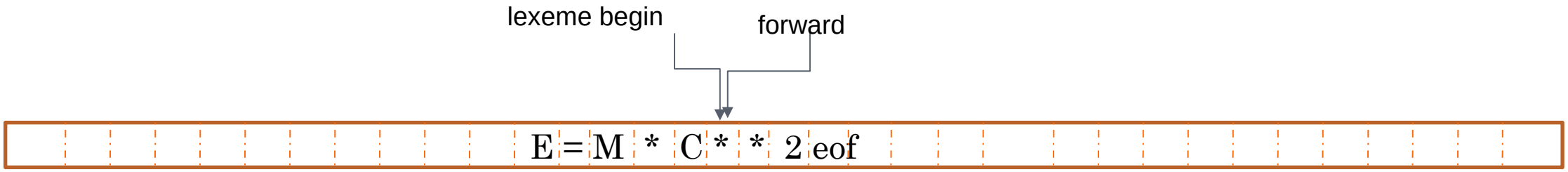
- Sometimes lexical analyzer needs to look ahead some symbols to decide about the token to return
- These extra characters has to be pushed back to the input becouse it can be beginning of the next lexeme.
- The implementation of reading and pushing back char is usually done by setting up an input buffer.
- For input buffering two pointers are maintained:
  - lexeme begin
  - forword

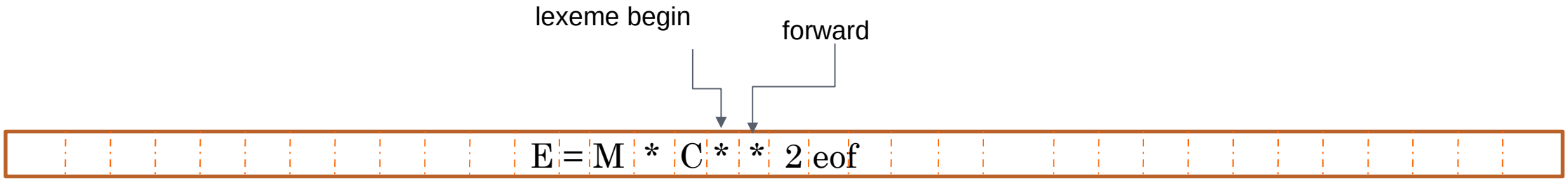


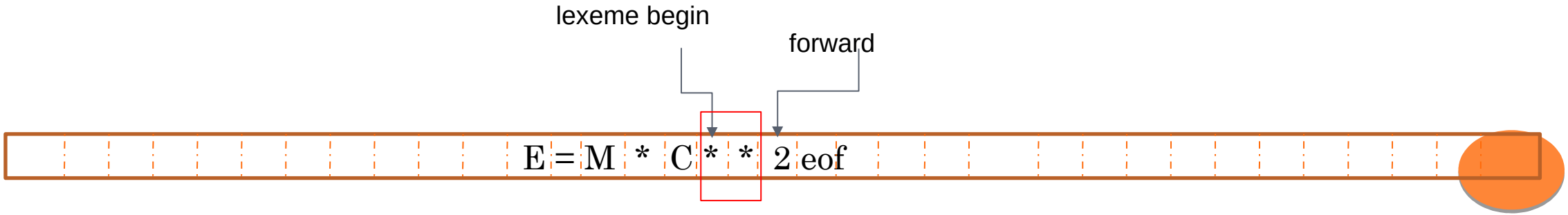

lexeme begin
forward
E = M * C * * 2 eof


lexeme begin
forward
E = M * C * * 2 eof


lexeme begin
forward
E = M * C * * 2 eof


lexeme begin
forward
E = M * C * * 2 eof


lexeme begin
forward
E = M * C * * 2 eof

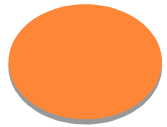

lexeme begin
forward
E = M * C * * 2 eof

2 BUFFER TECHNIQUE

- Because of the amount of time required to process char and the large no. of characters processed during the compilation of large source program specilized buffer techniques are used to reduce the overhead required to process a single input character.
- we use a buffer divided into 2 N-character halves.
- each buffer is of the same size N, (N is usually the size of a disk block.

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: regular expression for identifier
  - Letter_(letter_ | digit)*
- Each regular expression is a pattern specifying the form of strings



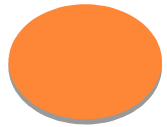
RECOGNITION OF TOKENS (CONT.)

- 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)+

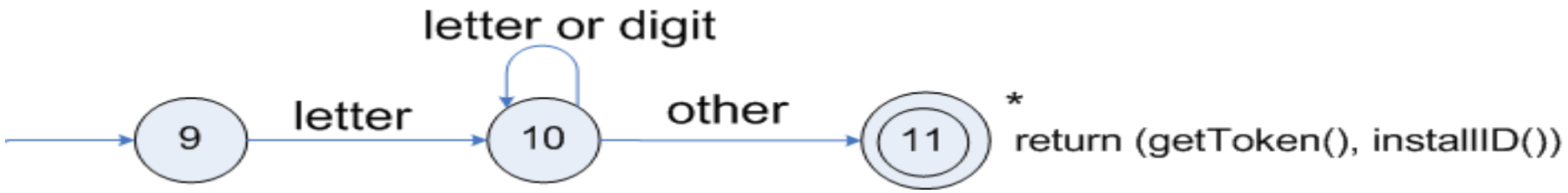


SIMPLE APPROCH TO DESIGN LEXICAL ANALYZER : TRANSITION DIAGRAMS

- We may use flowchart to describe the behaviour of any program.

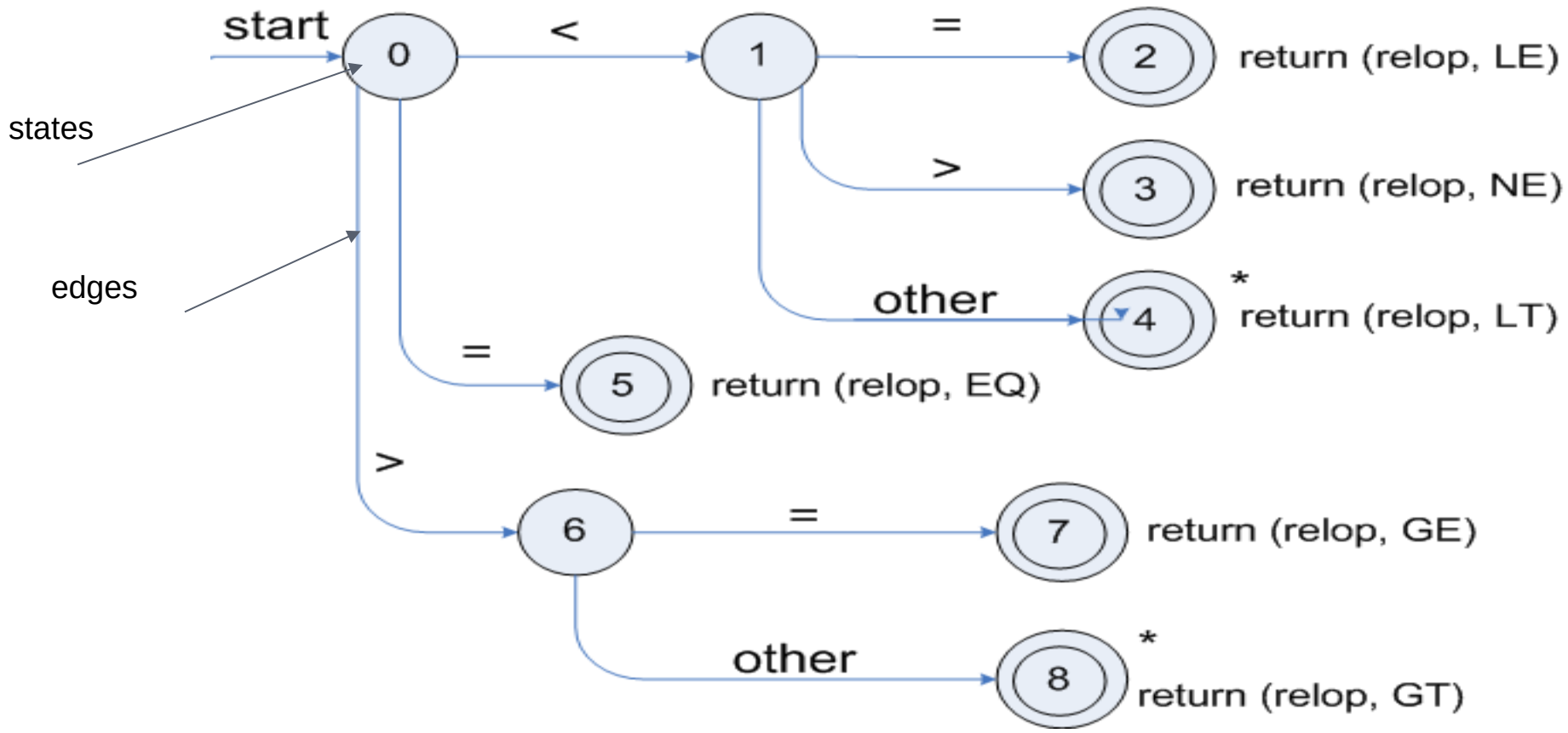
for this a specialized kind of flowchart “Transition Diagram” is used.

- Transition diagram for reserved words and identifiers



TRANSITION DIAGRAMS (CONT.)

- Transition diagram for relop

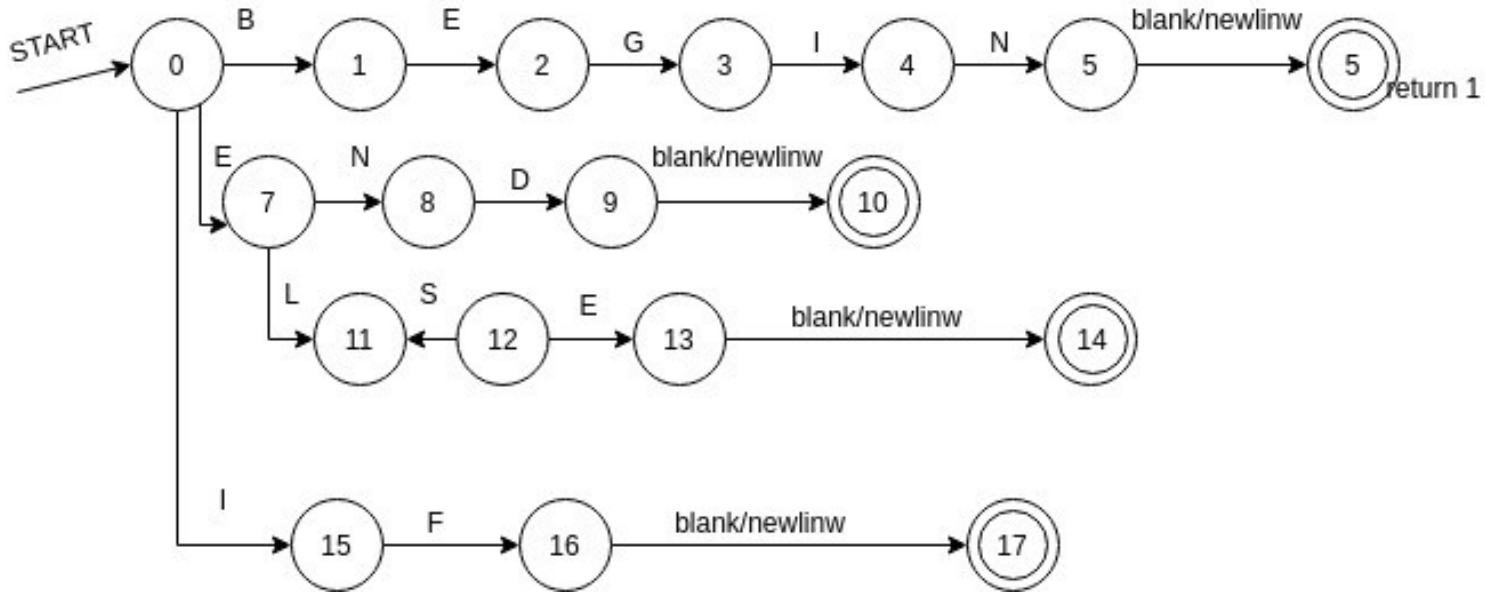


TRANSITION DIAGRAMS (CONT.)

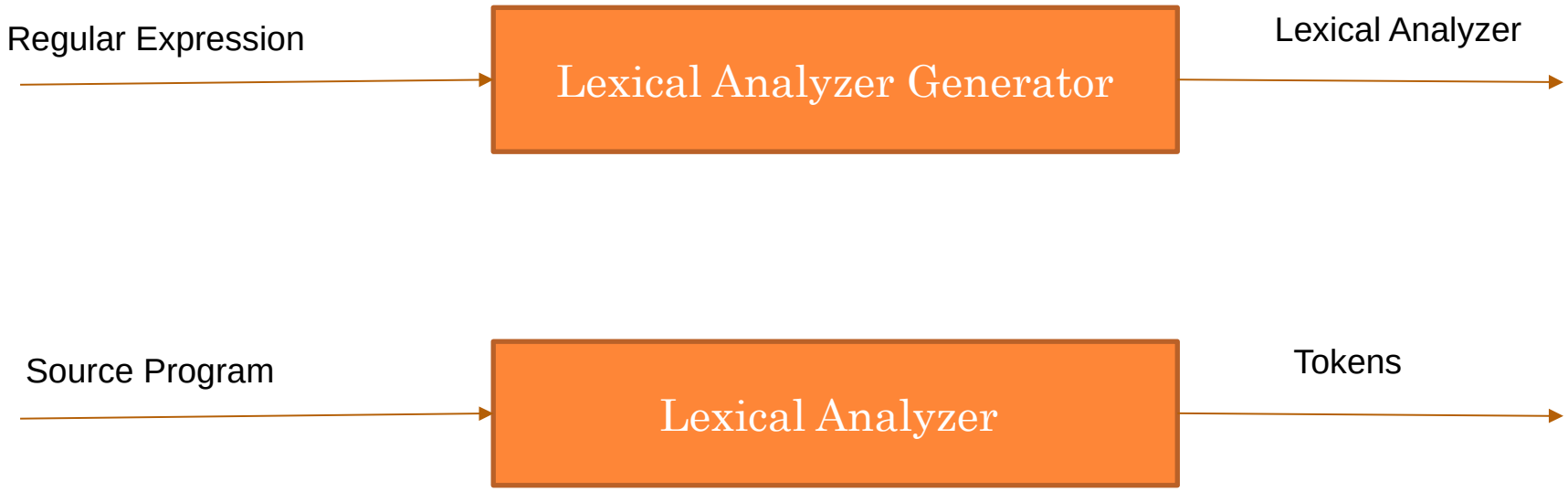
- Transition diagram for whitespace

TRANSITION DIAGRAMS (CONT.)

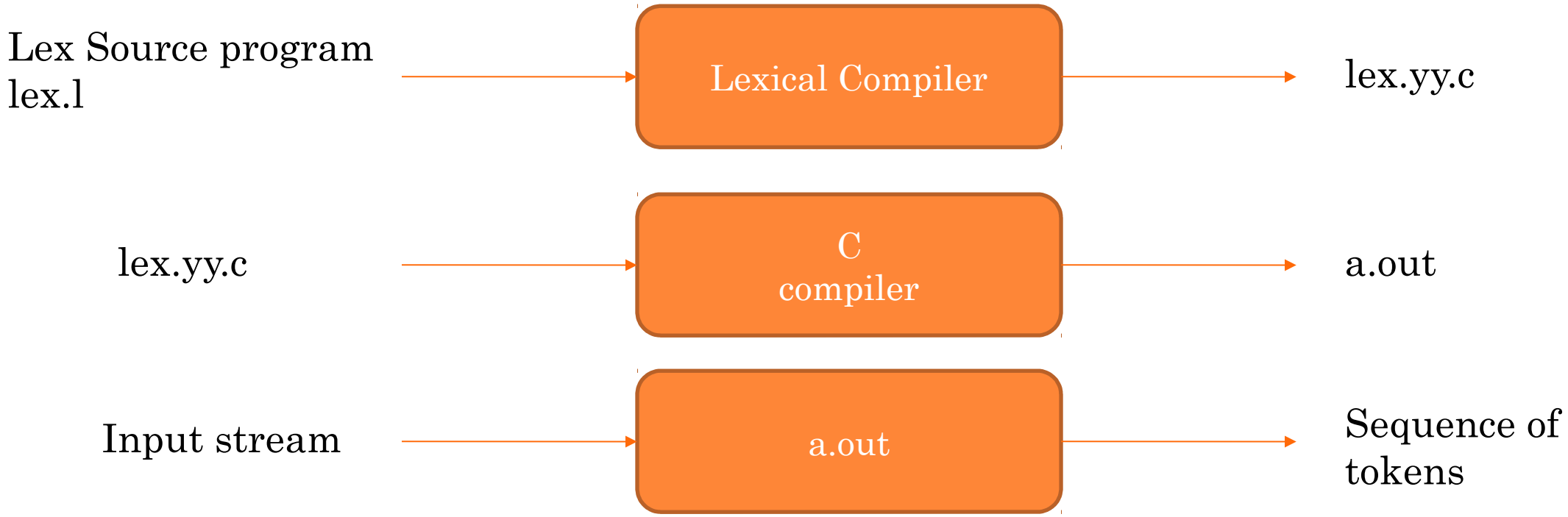
Transition Diagram for keyword



LEXICAL ANALYZER GENERATOR -



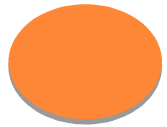
LEXICAL ANALYZER GENERATOR - LEX



STRUCTURE OF LEX PROGRAMS

declarations
%%
translation rules
%%
auxiliary functions

Pattern {Action}



GRAMMAR

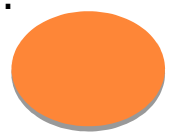
- the formal definition of the syntax of a programming language is usually called ‘Grammar’
- A Grammar consist of a set of rules to specify the sequence of characters that form allowable program in the language being defined.

- Formal Grammar : A Grammar specified using a strictly defined notations
- Equivalent Grammar : two grammar are equivalent if they produce the same language.

S -> bS
S -> Sb
S -> a

S -> bS
S -> aT
T -> bT
T -> ε

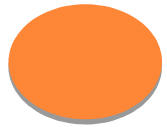
- Regular Grammar: Regular grammar are special cases of BNF grammar that turn out to be equivalent to the finite state automata.
- Regular Expression: a form of language definition that is equivalent to FSA and regular grammar.
  - a special text string for describing a search pattern.



FINITE AUTOMATA

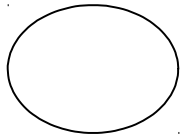
- Regular expressions = specification
- Finite automata = implementation

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

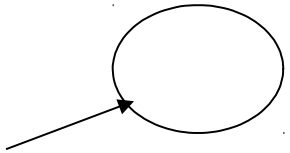


FINITE AUTOMATA STATE GRAPHS

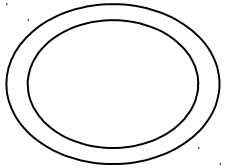
- A state



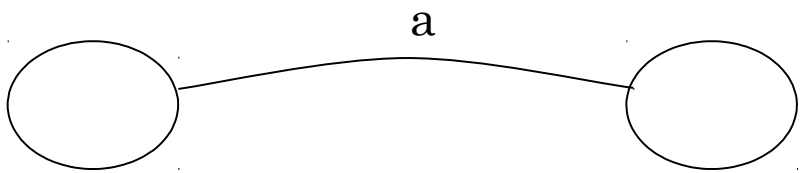
- The start state



- An accepting state

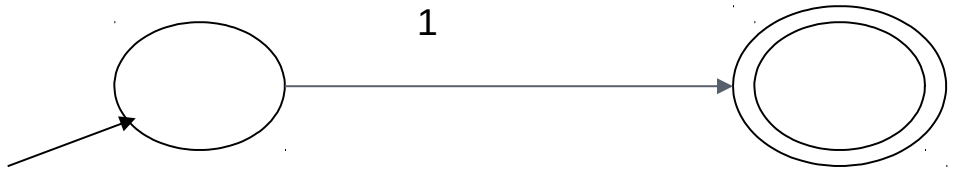


- A transition



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

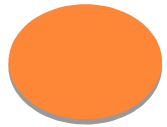
EPSILON MOVES

- Another kind of transition: ε-moves

- Machine can move from state A to state B without reading input

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



ARDEN'S THEOREM STATE THAT:

"If P and Q are two regular expressions over Σ, and if P does not contain ϵ , then the following equation in R

given by R = Q + RP

has an unique solution i.e.,

R = QP*."

Let's start by taking this equation as equation (i)

R = Q + RP .....(i)

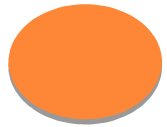
Now, replacing R by R = QP*, we get,

R = Q + QP*P

Taking Q as common,

R = Q(ϵ + P*P)

R = QP*



ARDEN'S THEOREM STATE THAT:

Initial and final state-q1

Step-01:

equations are

q1=q1 a + q3 a + ϵ (1

q2= q1 b + q2 b + q3 b (2

q3= q2 a (3

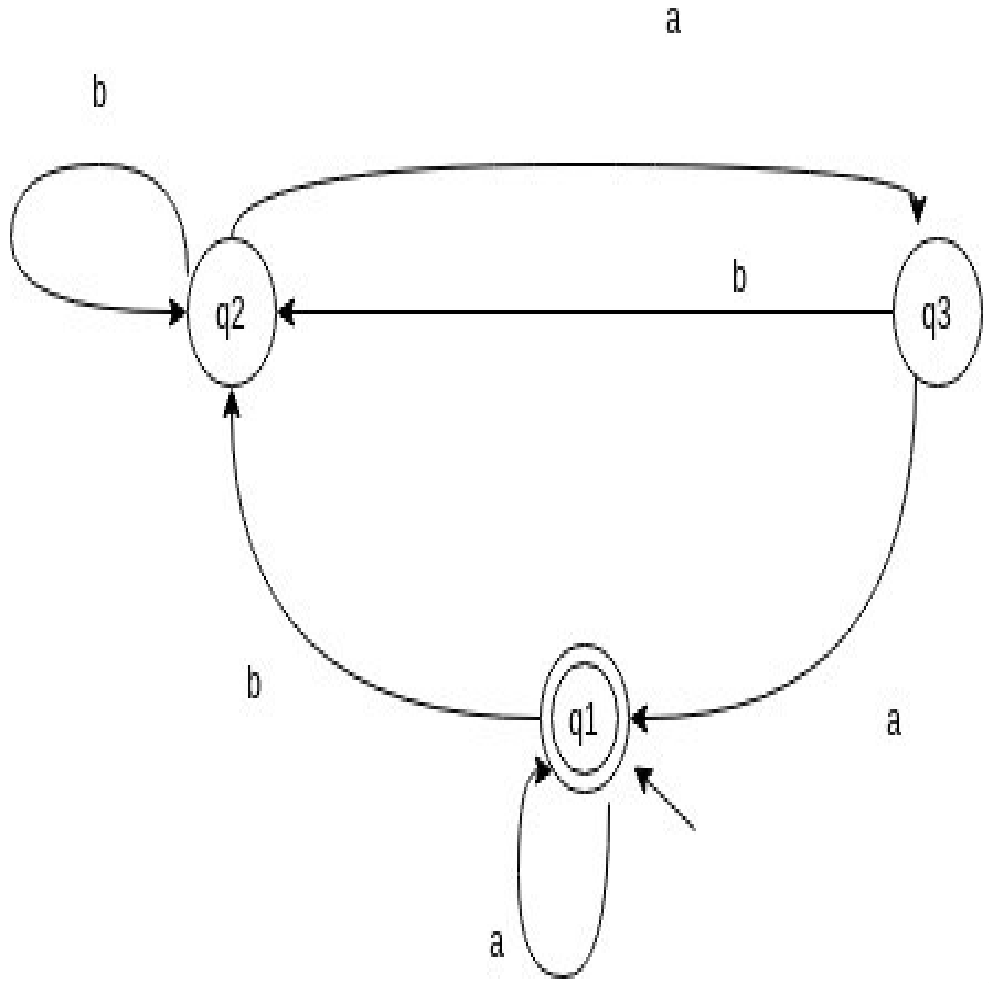
Bring state q2 in the form R = Q + RP.

q2= q1 b + q2 b + (q2 a) b

= q1 b + q2(b + a b)

by arden's theorem

q2=q1 b (b+ab)* (4



Using (3) in (1), we get-

$$q1 = q1\ a + q3\ a + \epsilon$$
$$= q1a + (q2\ a)a + \epsilon$$

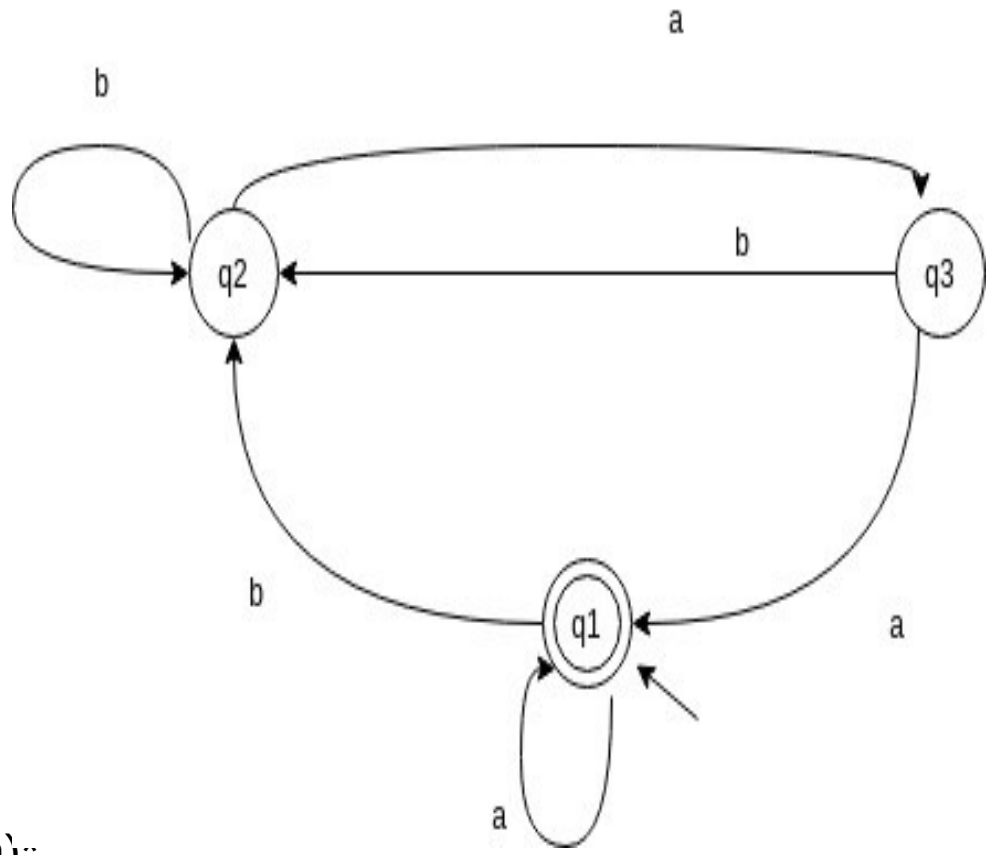
Using (4) in (1), we get-

$$= q1\ a + q1\ b\ (b+ab)^*\ a\ a + \epsilon$$
$$= q1\ (a + b\ (b+ab)^*\ a\ a) + \epsilon \qquad (5$$

Using Arden's Theorem in (5), we get-

$$q1 = \epsilon + q1\ (a + b\ (b+ab)^*\ a\ a)$$
$$q1 = (a+b(b+ab)^*aa)^*$$

Thus, Regular Expression for the given DFA = $(a+b(b+ab)^*aa)^*$

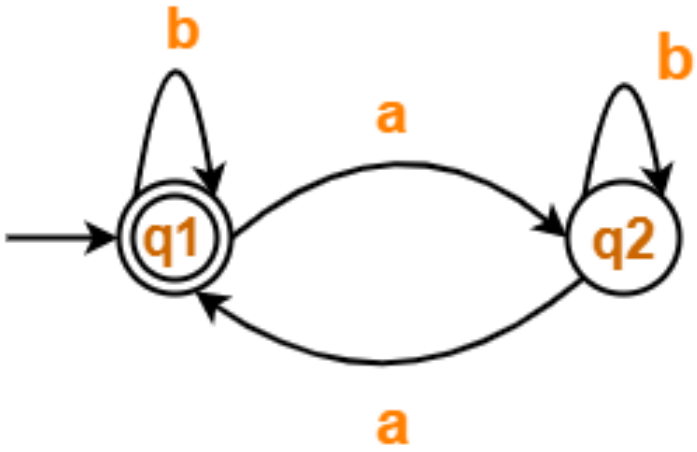


EXAMPLES

Step-01:

Form a equation for each state-

- q1 = ∈ + q1 b + q2 a (1)
- q2 = q1 a + q2 b (2)



Step-02:

Bring final state in the form R = Q + RP.

Using Arden's Theorem in (2), we get-

q2 = q1 a b* …….(3)

Using (3) in (1), we get-

$q1 = \in + q1\ b + q1\ a\ b^*\ a$

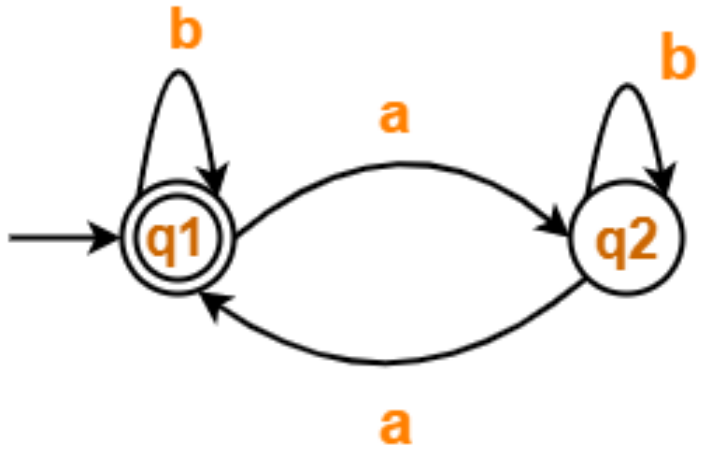
$q1 = \in + q1\ (b + a\ b^*\ a)$ .....(4)

Using Arden's Theorem in (4), we get-

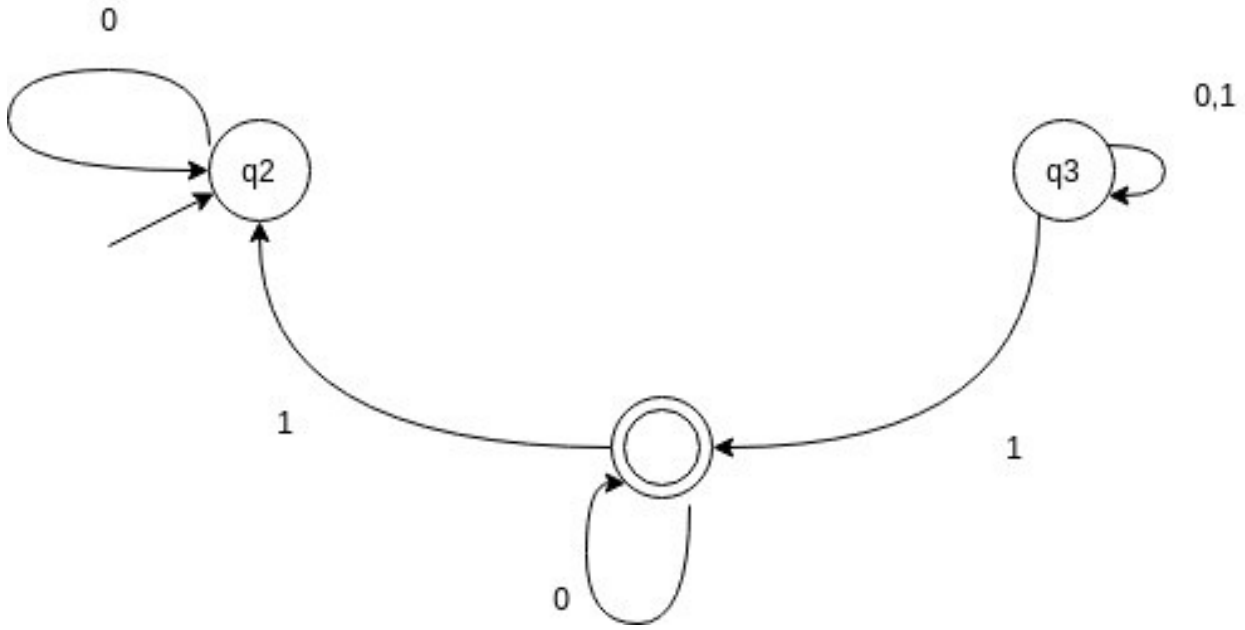
$q1 = \in (b + a\ b^*\ a)^*$

$q1 = (b + a\ b^*\ a)^*$

Thus, Regular Expression for the given DFA = $(b + a\ b^*\ a)^*$

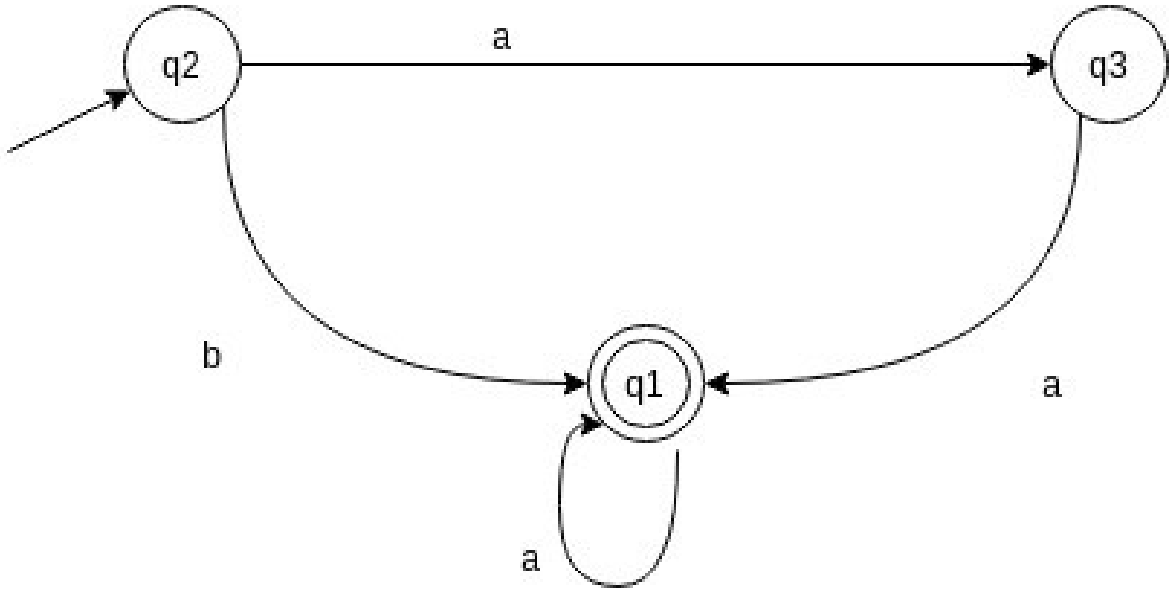


EXAMPLES



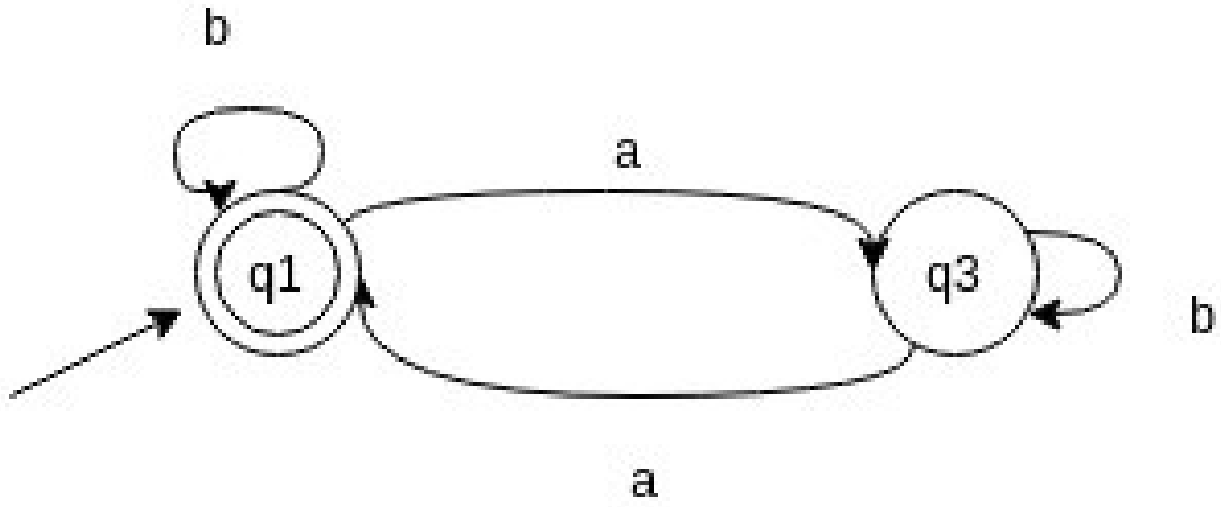
0*10*

EXAMPLES



(b+aa)a*

EXAMPLES

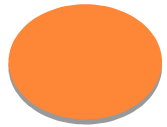


(b+ab*a)*

EXECUTION OF FINITE AUTOMATA

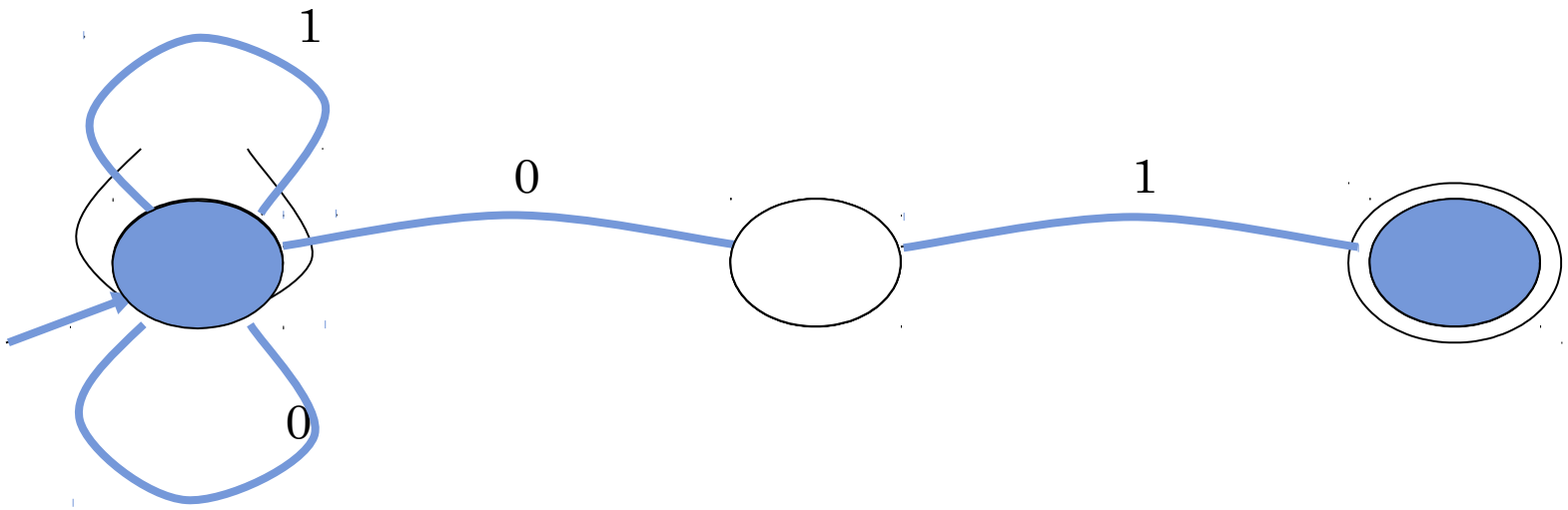
- A DFA can take only one path through the state graph
  - Completely determined by input

- NFAs can choose
  - Whether to make ε-moves
  - Which of multiple transitions for a single input to take



ACCEPTANCE OF NFAS

- An NFA can get into multiple states

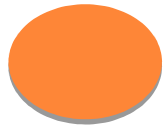


- Input: 1 0 1
- Rule: NFA accepts if it <u>can</u> get in a final state

NFA VS. DFA (1)

- NFAs and DFAs recognize the same set of languages (regular languages)

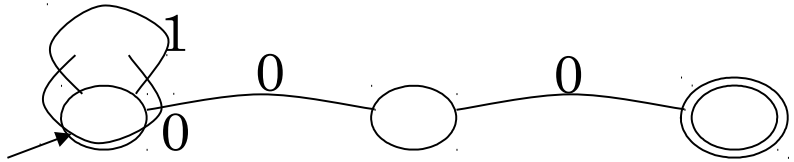
- DFAs are easier to implement
  - There are no choices to consider



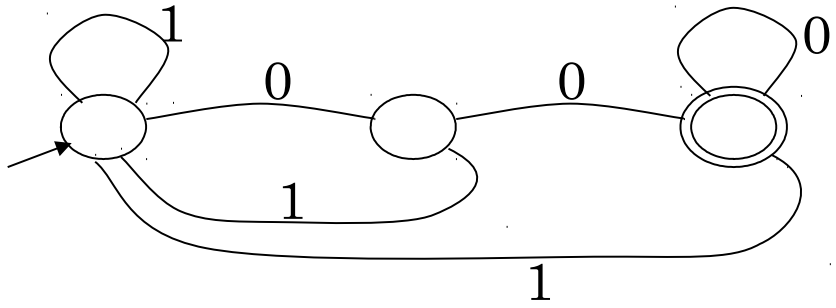
NFA VS. DFA (2)

- For a given language the NFA can be simpler than the DFA

NFA



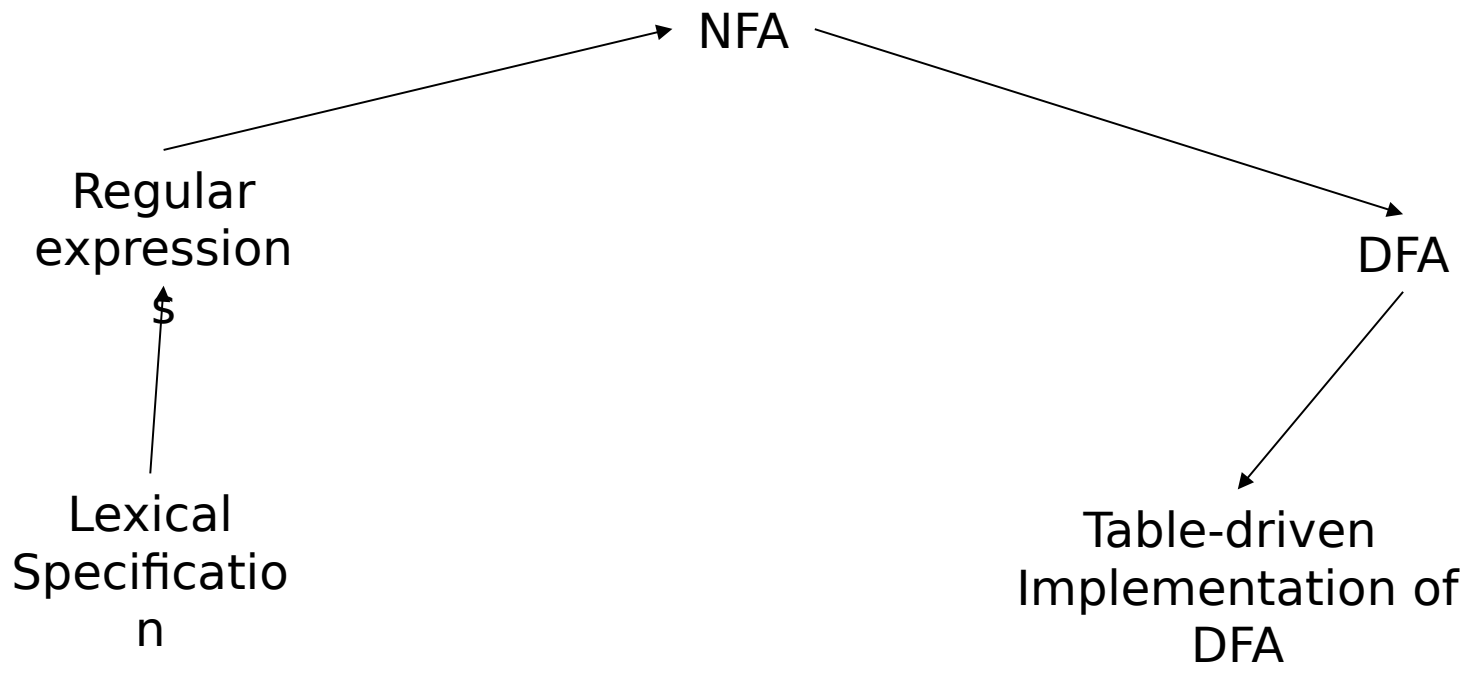
DFA



- DFA can be exponentially larger than NFA

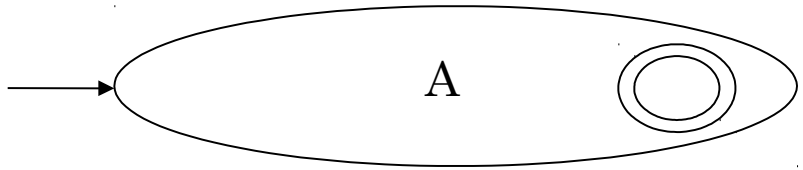
REGULAR EXPRESSIONS TO FINITE AUTOMATA

- High-level sketch

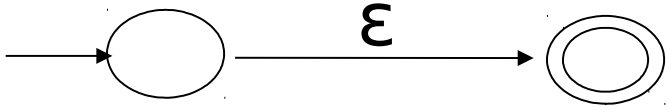


REGULAR EXPRESSIONS TO NFA (1)

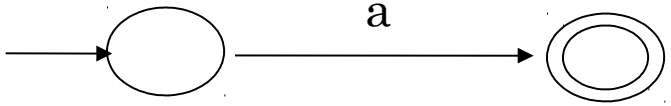
- For each kind of rexp, define an NFA
  - Notation: NFA for rexp A



- For ε

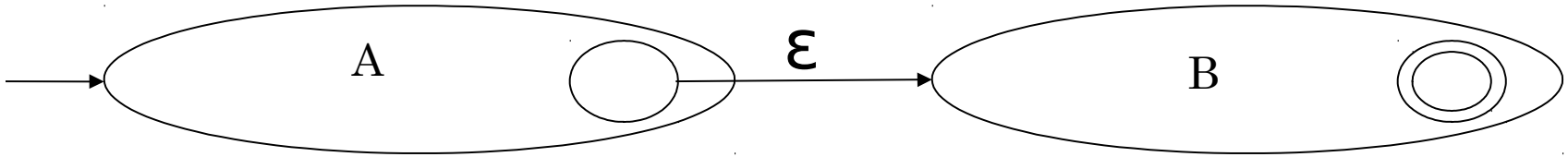


- For input a

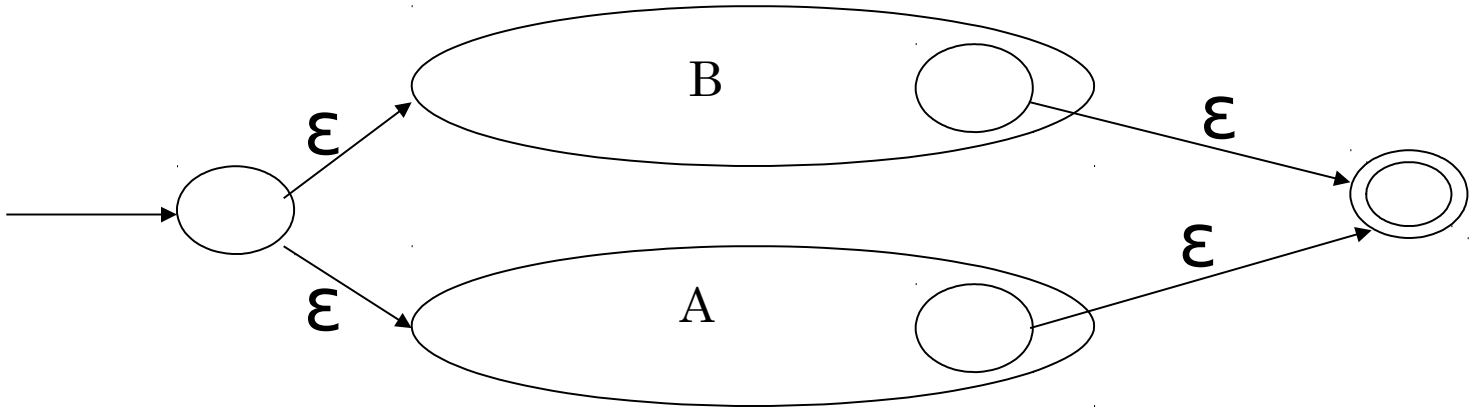


REGULAR EXPRESSIONS TO NFA (2)

- For AB

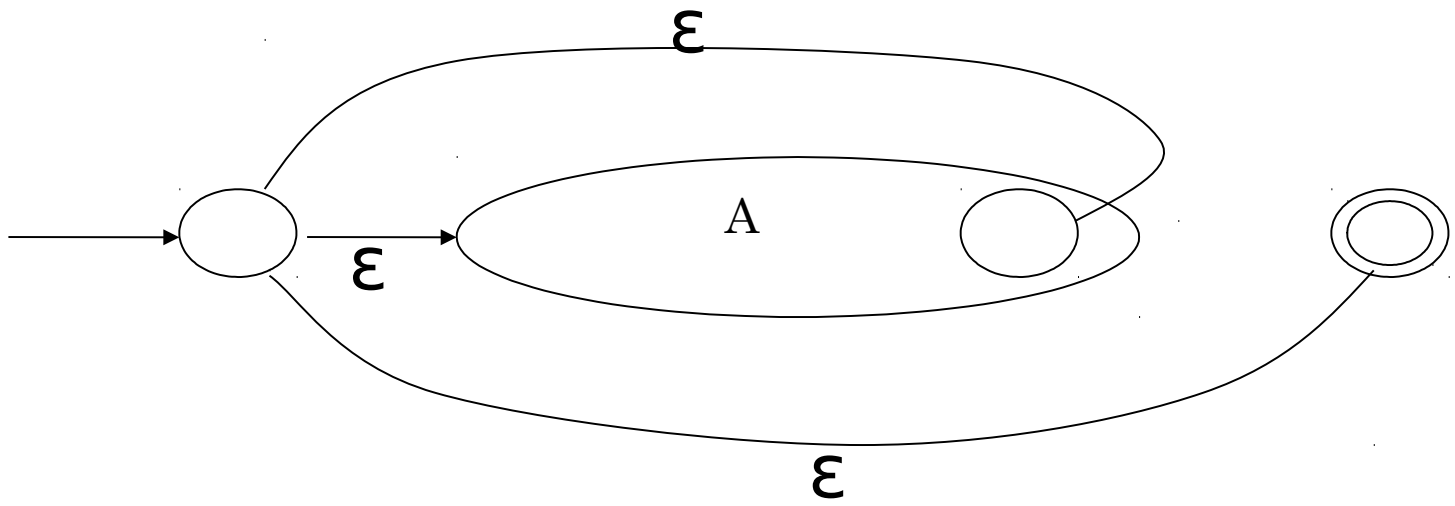


- For A | B



REGULAR EXPRESSIONS TO NFA (3)

- For A*

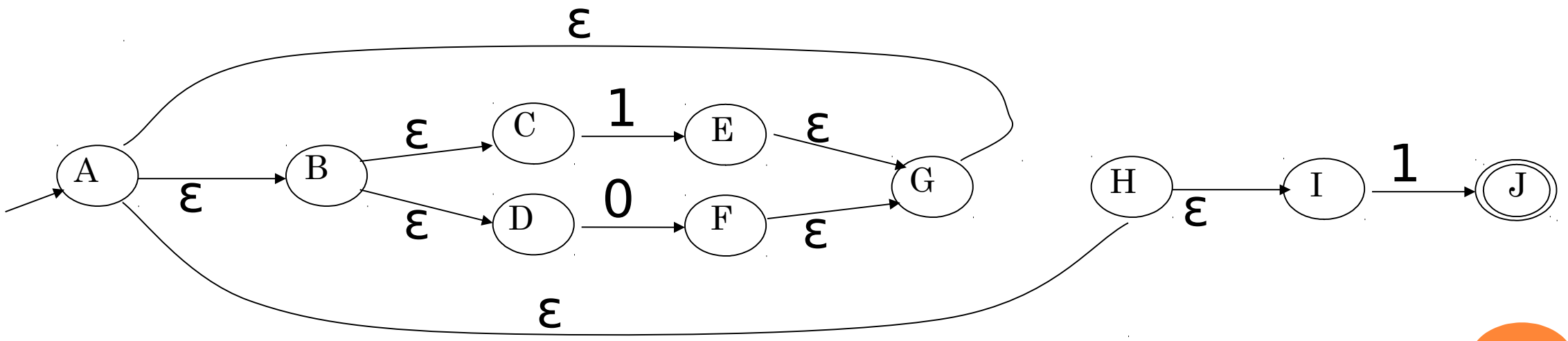


EXAMPLE OF REGEXP -> NFA CONVERSION

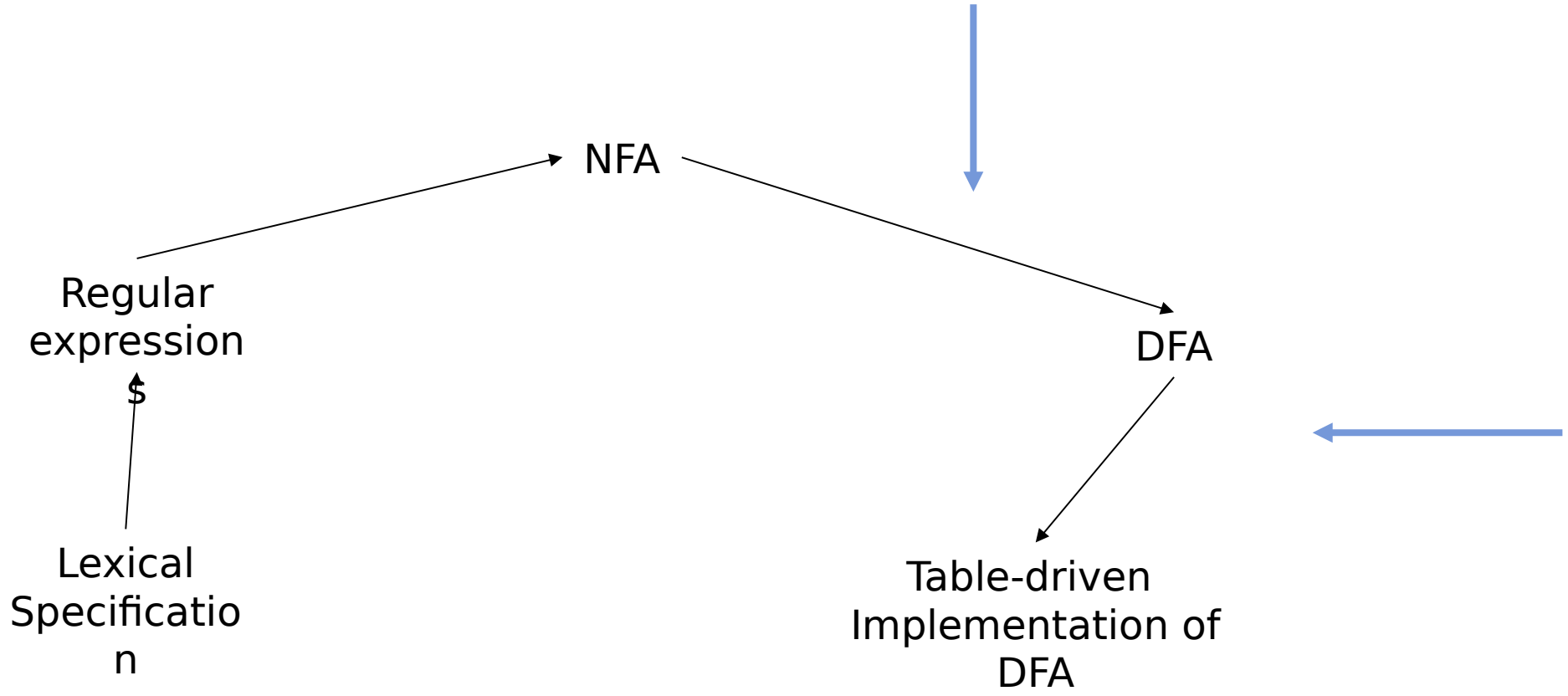
- Consider the regular expression

$$(1 \mid 0)^*1$$

- The NFA is



NEXT

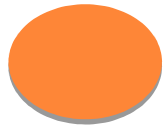

NFA
Regular expression s
DFA
Lexical Specification
Table-driven Implementation of DFA

NFA TO DFA. THE TRICK

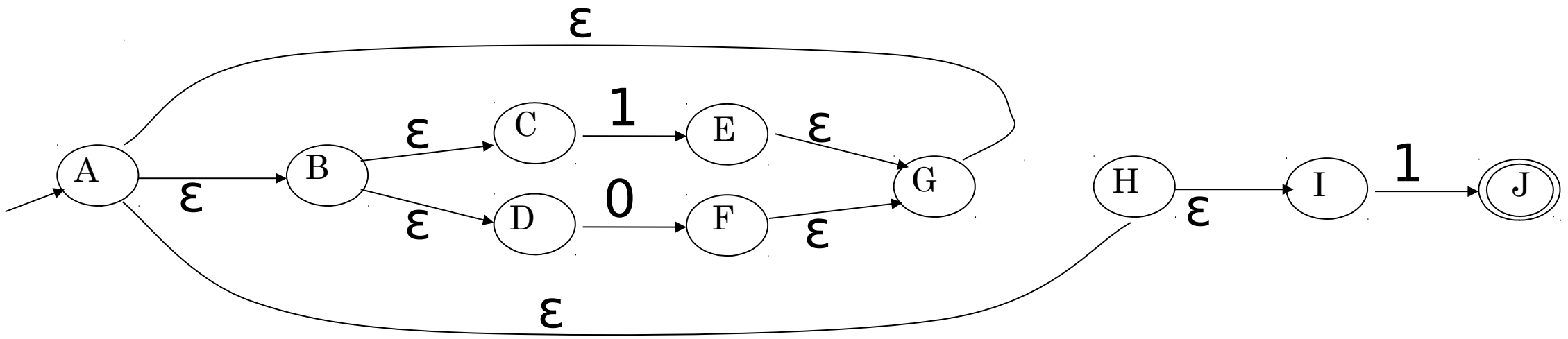
- Simulate the NFA
- Each state of resulting DFA

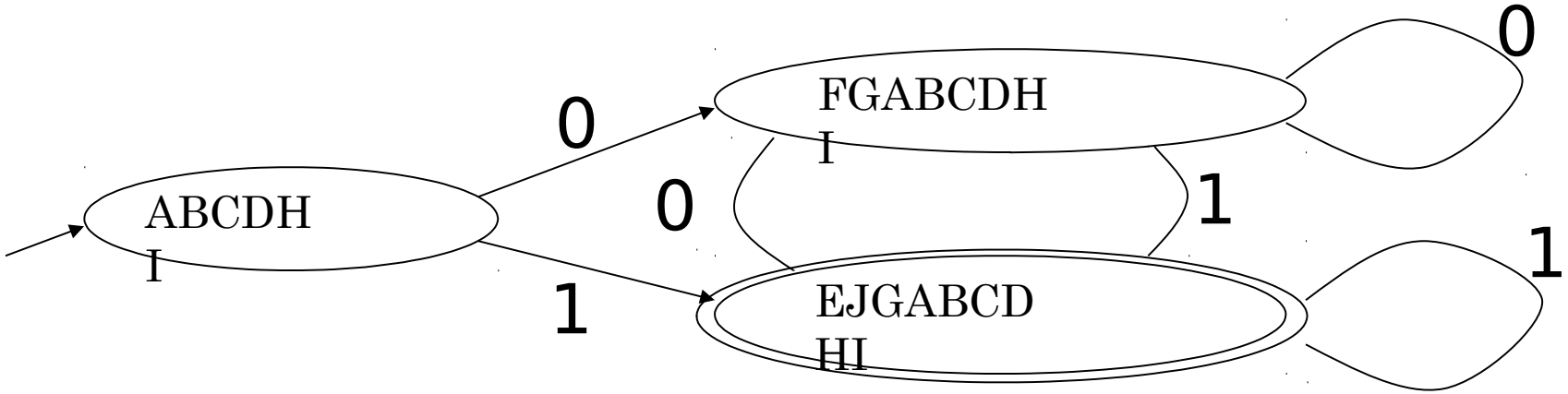
  = a non-empty subset of states of the NFA
- Start state

  = the set of NFA states reachable through ε-moves from NFA start state
- Add a transition $S \rightarrow^{a} S'$ to DFA iff
  - S' is the set of NFA states reachable from the states in S after seeing the input a
    - considering ε-moves as well



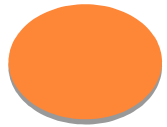
NFA -> DFA EXAMPLE





NFA TO DFA. REMARK

- An NFA may be in many states at any time
- How many different states ?
- If there are N states, the NFA must be in some subset of those N states
- How many non-empty subsets are there?
  - $2^N - 1$ = finitely many, but exponentially many



IMPLEMENTATION

- A DFA can be implemented by a 2D table T
  - One dimension is “states”
  - Other dimension is “input symbols”
  - For every transition $S_i \rightarrow^a S_k$ define T[i,a] = k
- DFA “execution”
  - If in state S_i and input a, read T[i,a] = k and skip to state S_k
  - Very efficient

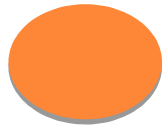
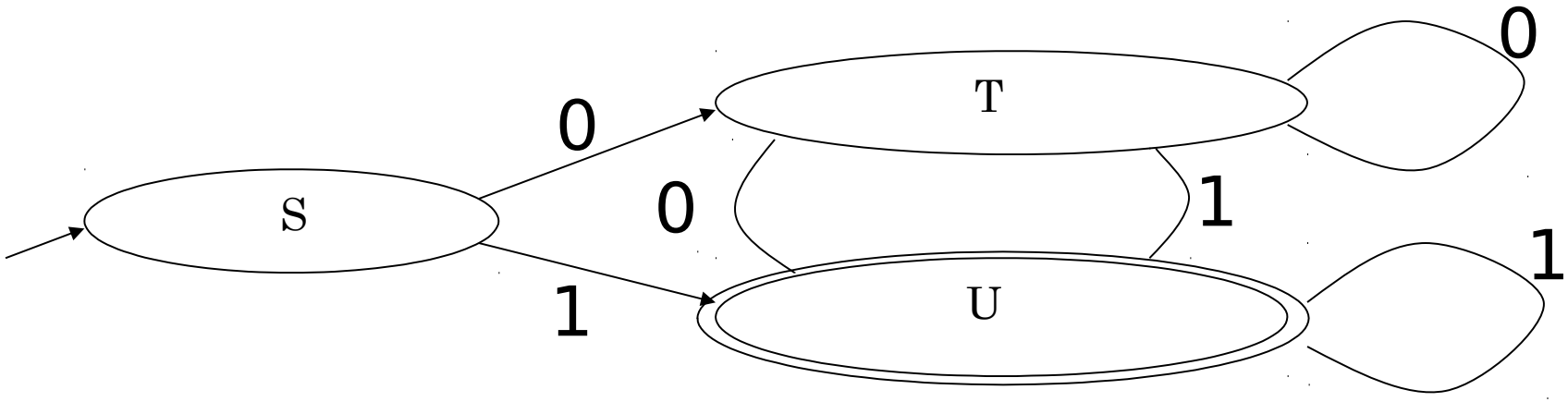


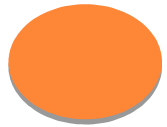
TABLE IMPLEMENTATION OF A DFA



	0	1
S	T	U
T	T	U
U	T	U

IMPLEMENTATION (CONT.)

- NFA -> DFA conversion is at the heart of tools such as flex or jflex
- But, DFAs can be huge
- In practice, flex-like tools trade off speed for space in the choice of NFA and DFA representations



WHAT ARE CONTEXT FREE GRAMMARS?

- In Formal Language Theory , a Context free Grammar(CFG) is a formal grammar in which every production rule is of the form

V w ⟶

Where V is a single nonterminal symbol and w is a string of terminals and/or nonterminals (w can be empty)

- The languages generated by context free grammars are knows as the context free languages

Pragya Gaur

FORMAL DEFINITION OF CFG

A context-free grammar G is a 4-tuple (V, ∑, R, S), where:

- V is a finite set; each element v ∈ V is called *a non-terminal character* or a *variable.*
- ∑ is a finite set of *terminals*, disjoint from , which make up the actual content of the sentence.
- R is a finite relation from V to (V U ∑)* , where the asterisk represents the Kleene star operation.

If (α,β) ∈ R, we write production α ◊ β

β is called a **sentential form**

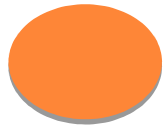
- S, the **start symbol**, used to represent the whole sentence (or program). It must be an element of V.

CONTEXT-FREE LANGUAGES

•Given a context-free grammar
G = (V,∑,R, S), the **language generated** or derived from G is the set
L(G) = {w :S ⇒* w}

A language L is context-free if there is a context-free grammar G = (V,∑, R, S), such that L is generated from G.

Pragya Gaur

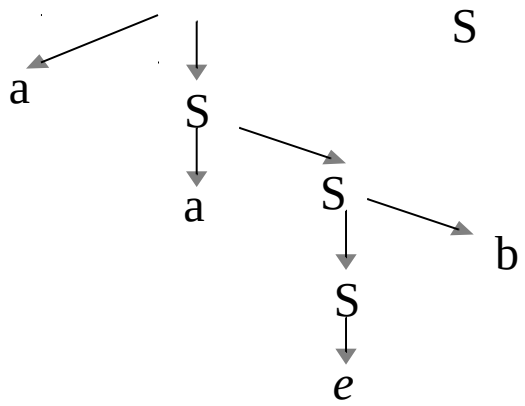


EXAMPLE :WELL-FORMED PARENTHESES

- The canonical example of a context free grammar is parenthesis matching, which is representative of the general case. There are two terminal symbols "(" and ")" and one nonterminal symbol S. The production rules are
- S → S
- SS → (S)
- S → ()
- The first rule allows Ss to multiply; the second rule allows Ss to become enclosed by matching parentheses; and the third rule terminates the recursion.

PARSE TREE

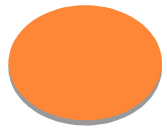
- A parse tree of a derivation is a tree in which:
  - Each internal node is labeled with a nonterminal
  - If a rule $A \lozenge A_1A_2\ldots A_n$ occurs in the derivation then A is a parent node of nodes labeled $A_1, A_2, \ldots, A_n$



LEFTMOST, RIGHTMOST DERIVATIONS

- A **left-most derivation** of a sentential form is one in which rules transforming the left-most nonterminal are always applied

- A **right-most derivation** of a sentential form is one in which rules transforming the right-most nonterminal are always applied



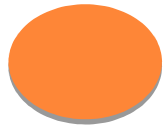
AMBIGUOUS GRAMMAR

- . A grammar G is ambiguous if there is a word w ∈ L(G) having are least two different parse trees

$$S \Diamond A \quad S \Diamond B \quad S \Diamond AB$$
$$A \Diamond aA \quad B \Diamond bB \quad A \Diamond e \quad B \Diamond e$$

- Notice that a has at least two left-most derivations

Pragya Gaur



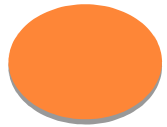
AMBIGUITY & DISAMBIGUATION

- Given an ambiguous grammar, would like an equivalent unambiguous grammar.
  - Allows you to know more about structure of a given derivation.
  - Simplifies inductive proofs on derivations.
  - Can lead to more efficient parsing algorithms.
  - In programming languages, want to impose a canonical structure on derivations. E.g., for **1+2×3**.
- Strategy: Force an ordering on all derivations.

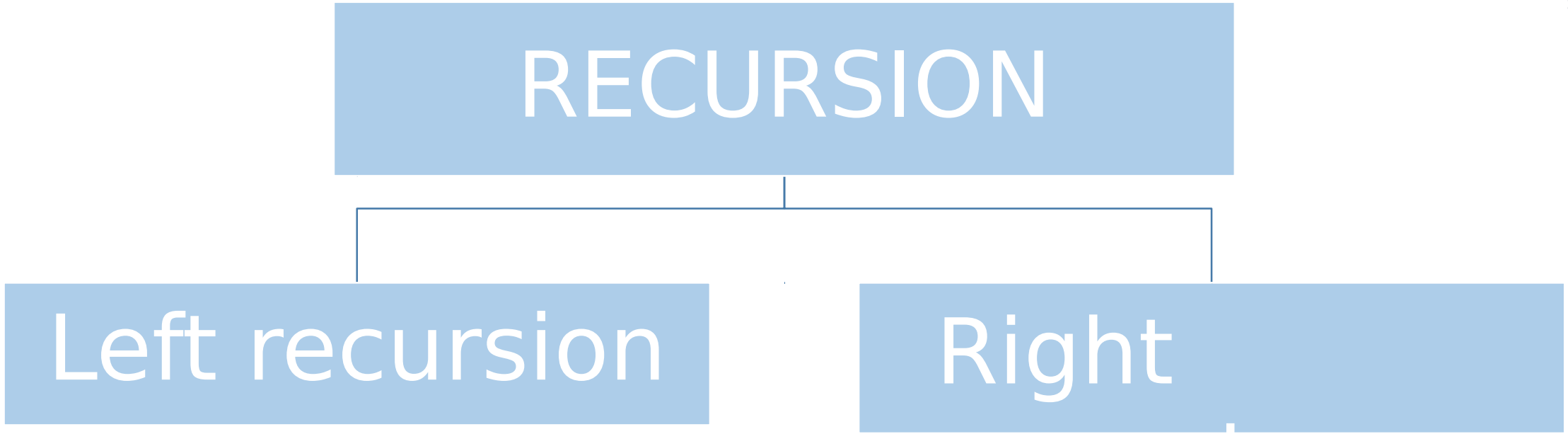
DEFINITION

- Recursion is the process a procedure goes through when one of the steps of the procedure involves invoking the procedure itself

Pragya Gaur



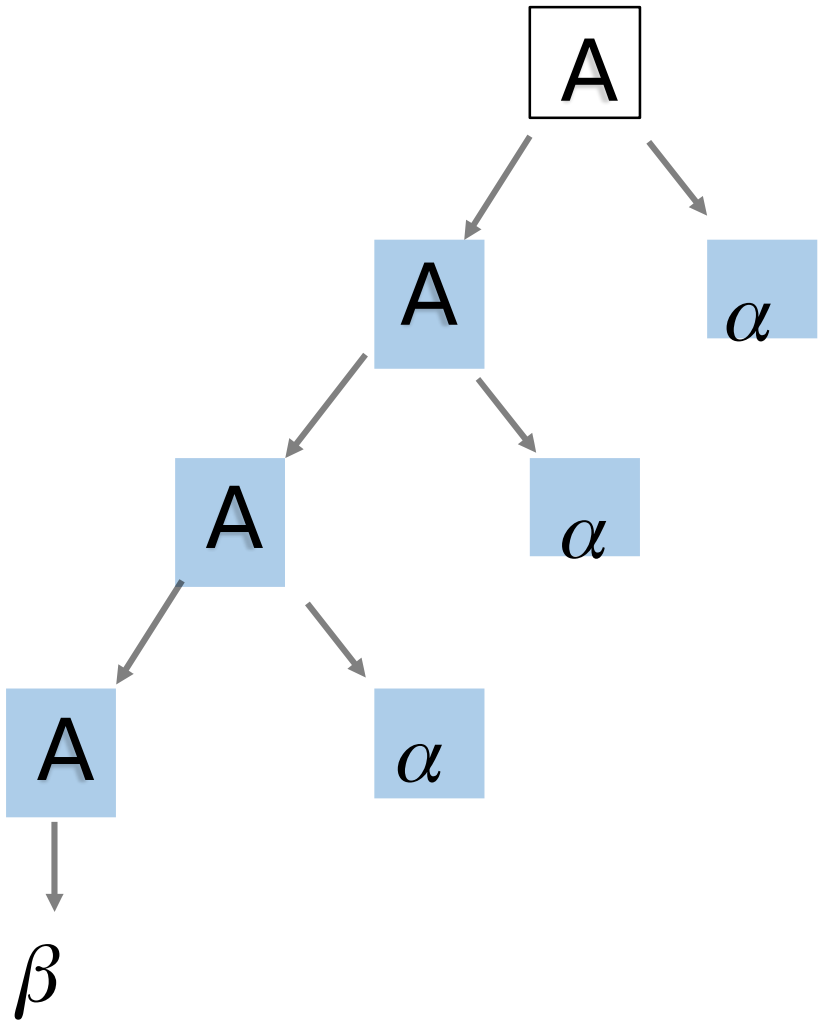
Classification

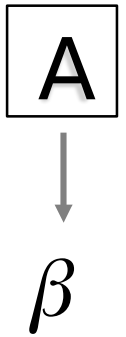


Pragya Gaur

LEFT RECURSION

$A \rightarrow A\alpha/\beta$



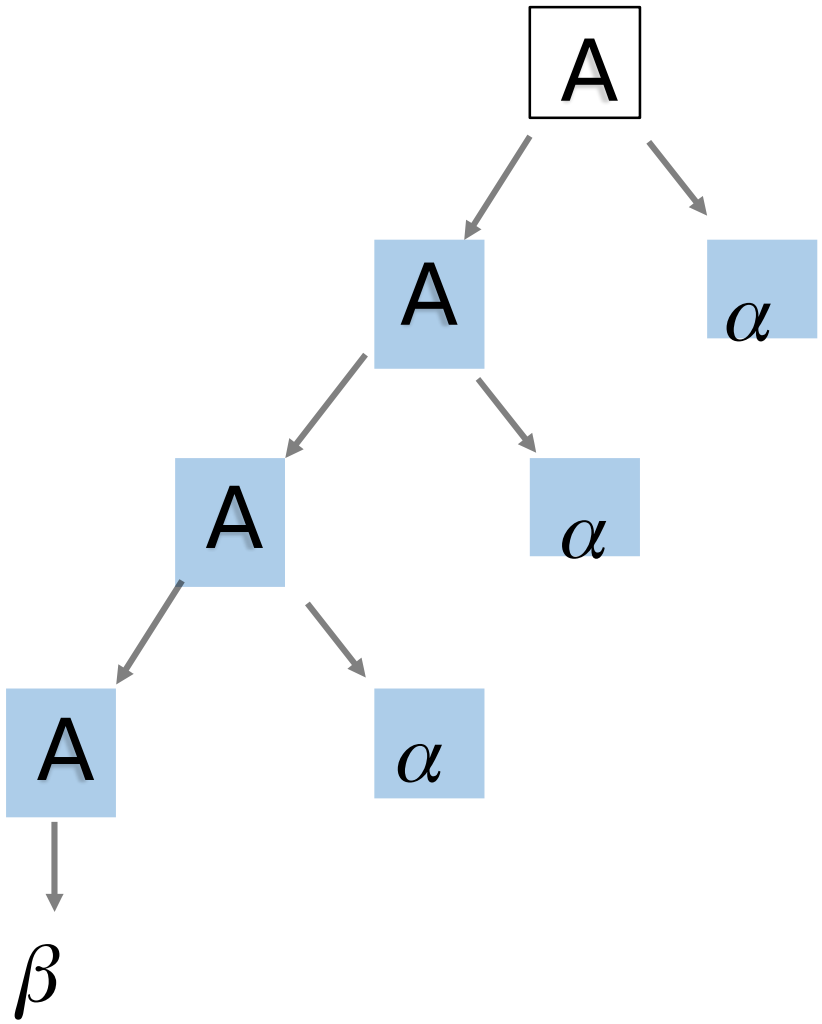


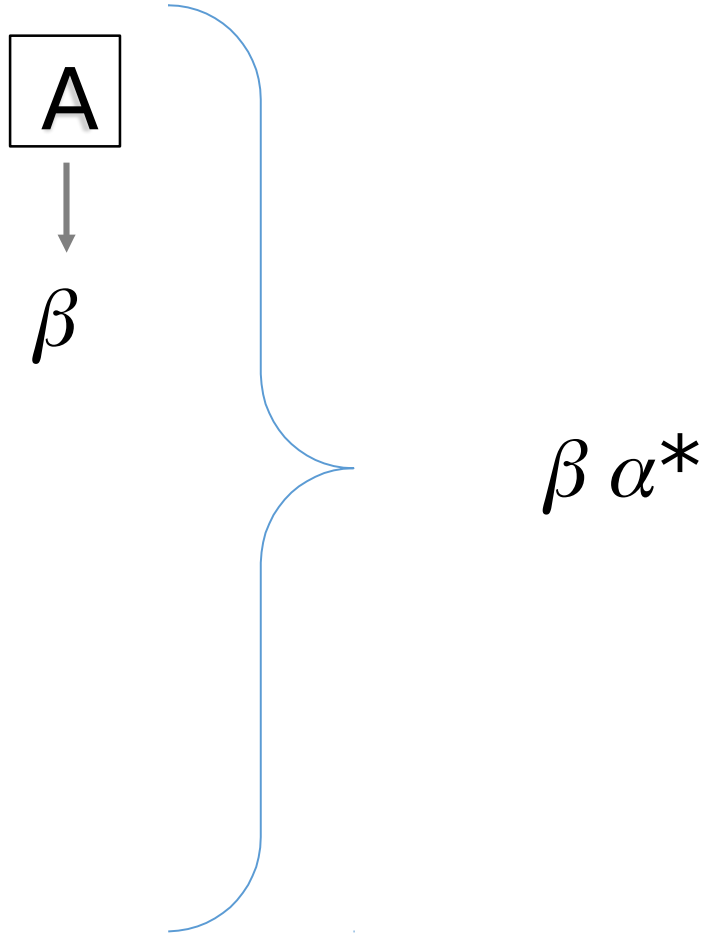
Top down parser doesn't allow in Left Recursion bcz of it's infinite problem.
So… we need to eliminate this problem.

Infinite problem

ELIMINATION OF LEFT RECURSION

$A \rightarrow A\alpha/\beta$





Pragya Gaur

ELIMINATION OF LEFT RECURSION

$$A \rightarrow \beta\, \alpha^*$$

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A'/\in$$

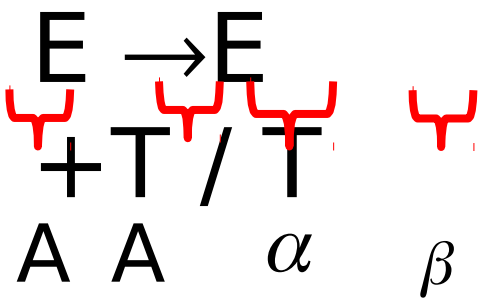
$$A \rightarrow A\alpha/\beta$$

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A'/\in$$

Pragya Gaur

EXAMPLE



$$E \rightarrow TE'$$

$$E' \rightarrow +T\,E'/\in$$

$$A \rightarrow A\alpha/\beta$$

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A'/\in$$

LEFT FACTORING

- In left factoring it is not clear which of two alternative productions to use to expand a nonterminal A.

  i.e. if A → $\alpha\beta_1 | \alpha\beta_2$

- W e don't know whether to expand A to $\alpha\beta_1$ or to $\alpha\beta_2$
- To remove left factoring for this grammar replace all A productions containing α as prefix by A → $\alpha A'$ then A′ → $\beta_1 | \beta_2$