LEXICAL ANALYSIS

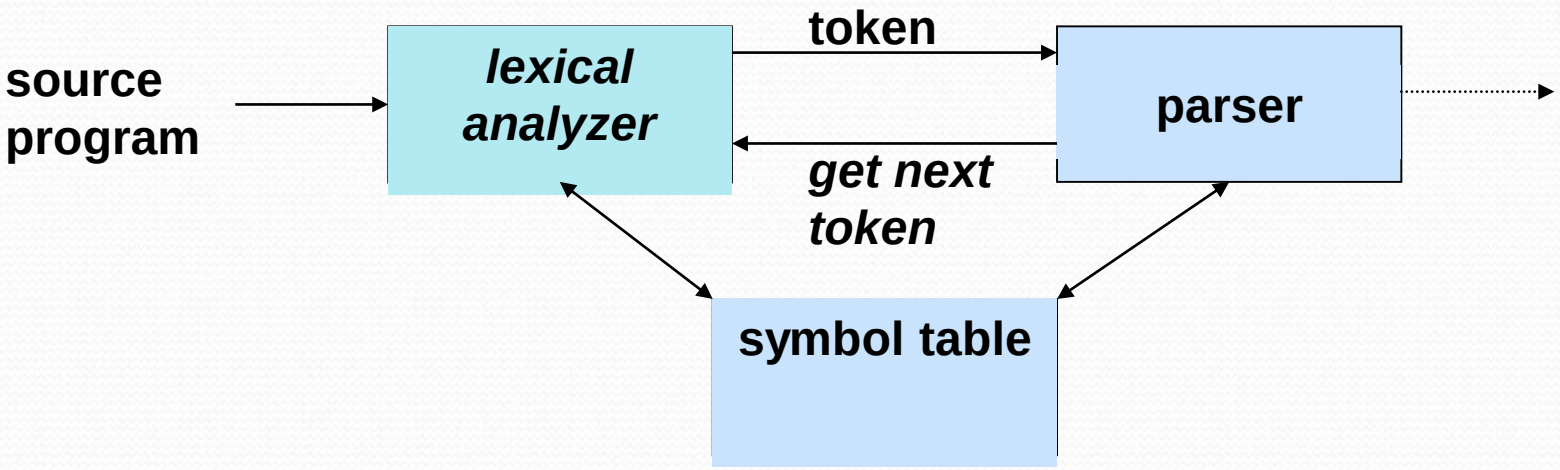
Shyamalendu Kandar
Assistant Professor, IT
IIEST, Shibpur

Introduction

- The word ‘Lexical’ in traditional sense means “Pertaining to words”.
- In terms of programming languages words are objects like variable name, number, keywords etc.
- Such words are traditionally called Tokens
- Tokens are sequence of characters having a collective meaning.
- Lexical Analyzer acts as an interface between the input source program to be compiled and the later stages of the compiler

Lexical Analyzer

- **Input program can be considered to be a sequence of characters.**
- **Lexical Analyzer** reads the source program character by character to produce tokens.
- Normally a lexical analyzer doesn't return a list of tokens at one shot, it returns a token when the parser asks a token from it.



- Lexical analyzer reads the source program from left to right and performs some secondary tasks like, removing source program's comments, white spaces in form of blank, tab & newline character.

The Role of Lexical Analyzer

- Lexical analyzer is the first phase of the compiler.
- Its main task is to **read the input characters and produce as output a sequence of tokens,** that the parser uses for the syntax analysis.
- Receiving a "**get next token**" command from the parser, the lexical analyzer reads input characters until it can identify the next token.
- It converts the **white spaces and comments** in the source program, in the form of **blank, tab and newline characters**.
- Participate in the creation and maintenance of **Symbol table**.
  - lexical only make entry of the symbols found from the source program in symbol table
  - Other fields like type, size etc are generally filled up by the parser.
- It correlates error messages from the compiler with the source program.(Lexical error.)

Issues in Lexical Analysis

There are several reasons for separating analysis phase of compiling into lexical analysis & parsing

A) Simpler Design: Lexical analysis phase removes comments, white spaces etc. and separates tokens.

B) Improves Efficiency: A large amount of time is spent in reading the source program and partitioning it into tokens. Specialized buffer technique for reading input characters and processing tokens can significantly speed up the performance of compiler.

C) Portability Enhanced: Some type of special symbols used in the program can be isolated in this phase. [Example // or /* */ in c is removed in Lexical analyzer]

Introducing Basic Terminology

TOKEN

- A classification for a common set of strings

  Examples Include <Identifier>, <number>, etc.
- ***A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages.***
- Example of tokens:

  Type token (id, num, real, . . .)

  Punctuation tokens (if, void, return, . . .)

  Alphabetic tokens (keywords)
- Example of non-tokens:

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

- PATTERN
  - The rules which characterize the set of strings for a token
  - Recall File and OS Wildcards ([A-Z]*.*)
  - There is a set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.
  - Regular expressions are an important notation for specifying patterns.
  - For example, the pattern for the Pascal identifier token, id, is: id → letter (letter | digit)*.
- LEXEME
  - Actual sequence of characters that matches pattern and is classified by a token
  - Identifiers: x, count, name, etc…
  - A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.
  - For example, the pattern for the RELOP token contains six lexemes (=, < >, <, < =, >, >=) so the lexical analyzer should return a RELOP token to parser whenever it sees any one of the six.

  const pi = 3.1416

  The substring pi is a lexeme for the token “identifier.”

Introducing Basic Terminology

Token	Informal description	Sample lexemes
if	Characters i, f	if
else	Characters e, l, s, e	else
Relation	< 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);

Id and number are

1. Stored in symbol table
2. Returned to parser

Token

- Token represents a ***set of strings described by a pattern***.
  - Identifier represents a ***set of strings which start with a letter continues with letters and digits***
  - The actual string is called as ***lexeme***.
  - Tokens: identifier, number, addop,…
- More than one patterns may match a lexeme.
  - Exp: Lexeme 12 may match with pattern of number or pattern of string.
- additional information should be held for that specific lexeme. This additional information is called as the *attribute* of the token.
- Token influence in parsing decision; the attributes influence the translation of token.
- For simplicity, a token may have a single attribute which holds the required information for that token.
- For identifiers, the attribute is the pointer to the symbol table, and the symbol table holds the actual attributes for that token.

  Another example: lexeme for an identifier and the line number on which it was first seen may be taken as attribute.
- Some attributes:

  – <id,attr> where attr is pointer to the symbol table

  – <assgop,_> no attribute is needed (if there is only one assignment operator)

  – <num,val> where val is the actual value of the number.
- Token type and its attribute uniquely identify a lexeme.

Token

- For simplicity, a token may have a single attribute which holds the required information for that token.
  - For identifiers, this attribute a pointer to the symbol table, and the symbol table holds the actual attributes for that token.

E.g. E = M * C ** 2

< id, pointer to the symbol-table entry for E >
<assign_op,>
< id, pointer to the symbol-table entry for M >
< mult_op,>
< id, pointer to the symbol-table entry for C >
<exp_op, >
<num, integer value 2> // integer valued attribute

Handling Lexical Errors

- Error Handling is very localized, with Respect to Input Source
- Very few errors are detected in Lexical phase.

For example: whil (x := o) do
generates **no** lexical errors in C , because 'whil' is a valid identifier. Lexical analyzer must return the token for an identifier and let some other phase of compiler handle any error.

A true lexical error:

int ıname;

int _count;

int a=b;

Does not matches with any pattern.

- Possible error recovery actions:

Check whether Prefix of remaining input can be transformed into a valid lexeme by just a single error transformation.

- Deleting or Inserting Input Characters
- Replacing or Transposing two adjacent Characters

- Or, skip over to next separator to "ignore" problem

Error Recovery

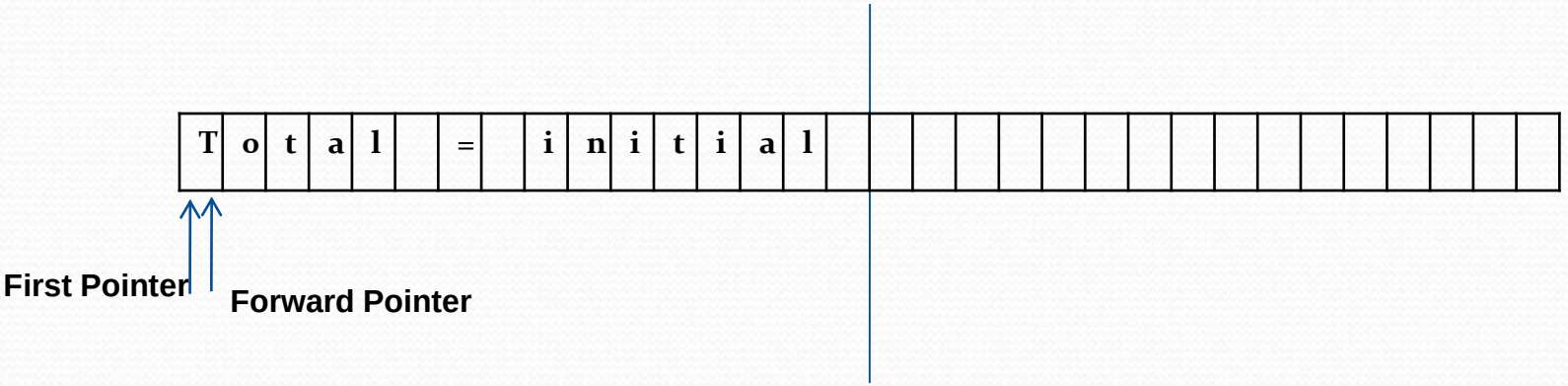
- Panic mode: successive characters are ignored until we reach to a well formed token
- Delete one character from the remaining input
- Insert a missing character into the remaining input
- Replace a character by another character
- Transpose two adjacent characters

Input Buffering

- Scanner performance is crucial:
  - This is the only part of the compiler that examines the entire input program one character at a time.
  - Disk input can be slow.
  - Considerable amount of time is spent (~25-30% of total compile time).
  - Speed of lexical analysis is a concern in compiler design.
- We need look ahead to determine when a match has been found.
- Scanners use *double-buffering* to minimize the overheads associated with this

Input Buffering

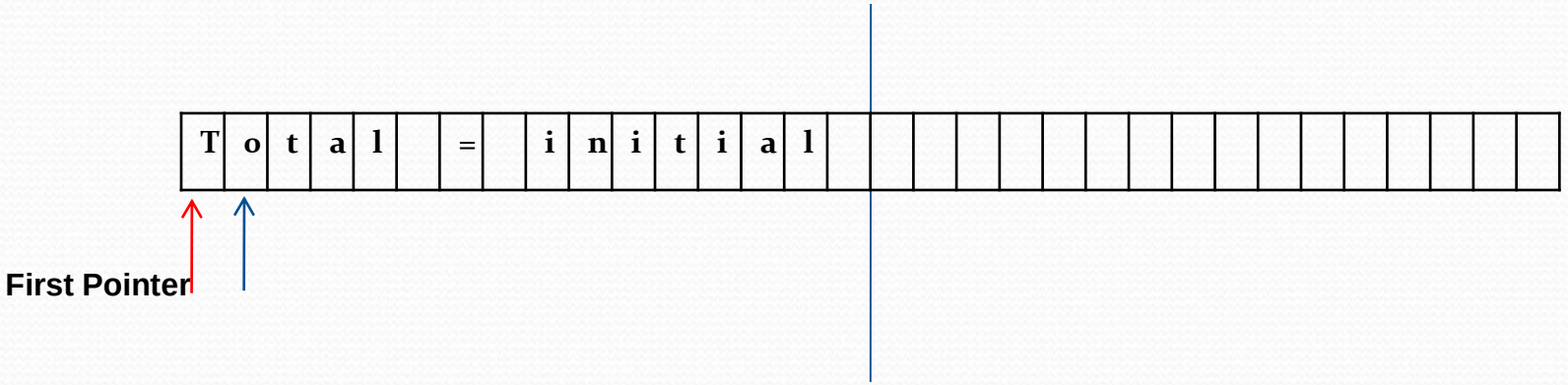
Total = initial + r_o_i *year ;



- We use a buffer divided into two N character halves. N is the number of characters on one disk block i.e. 1024 or 4096 etc.
- We read n input characters into each half of the buffer with one system read command, *rather than invoking a read command for each input character*.
- If fewer than input characters remain in the input, then a special character ‘eof’ is read into the buffer after the input character. ‘eof’ marks the end of the source file and is different from any input character.
- Initially both pointers point to the first character of the next lexeme to be found.
- One of them is called forward pointer. It scans ahead until a match for a pattern is found.
- Two pointers to the input buffer are maintained. The string of characters between the two pointers is the current lexeme.

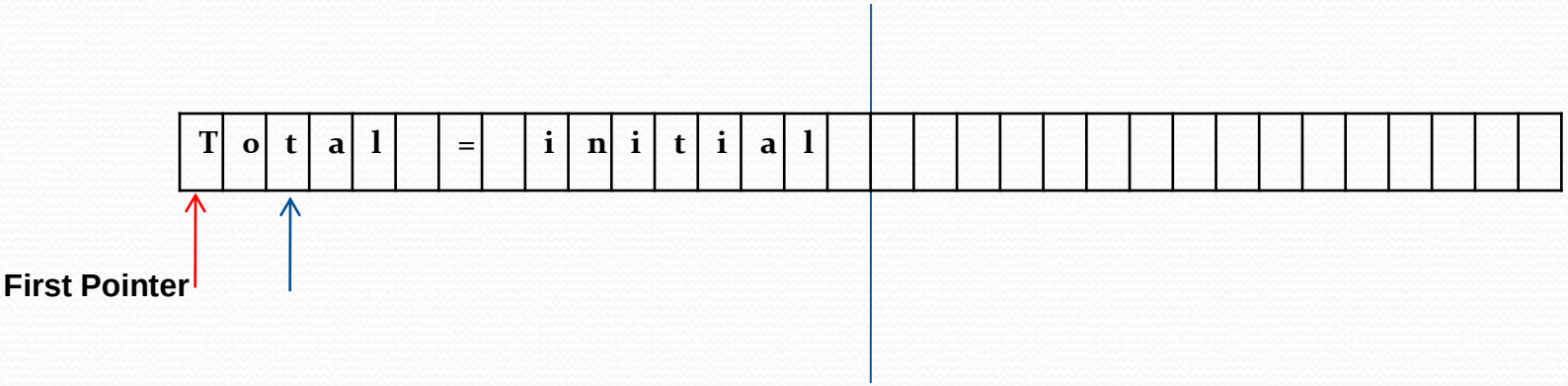
Input Buffering

Total = initial + r_o_i *year ;



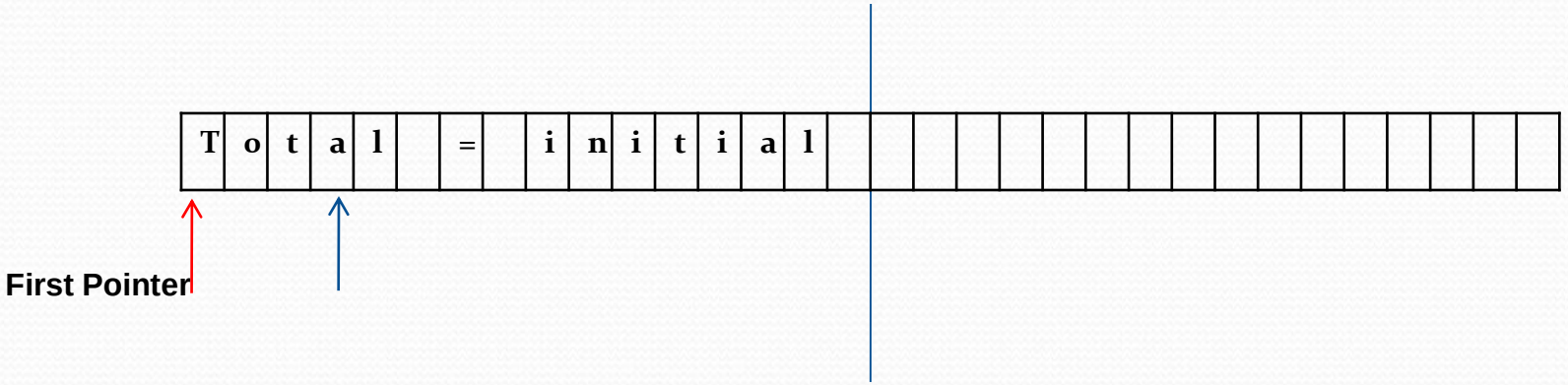
Input Buffering

Total = initial + r_o_i *year ;



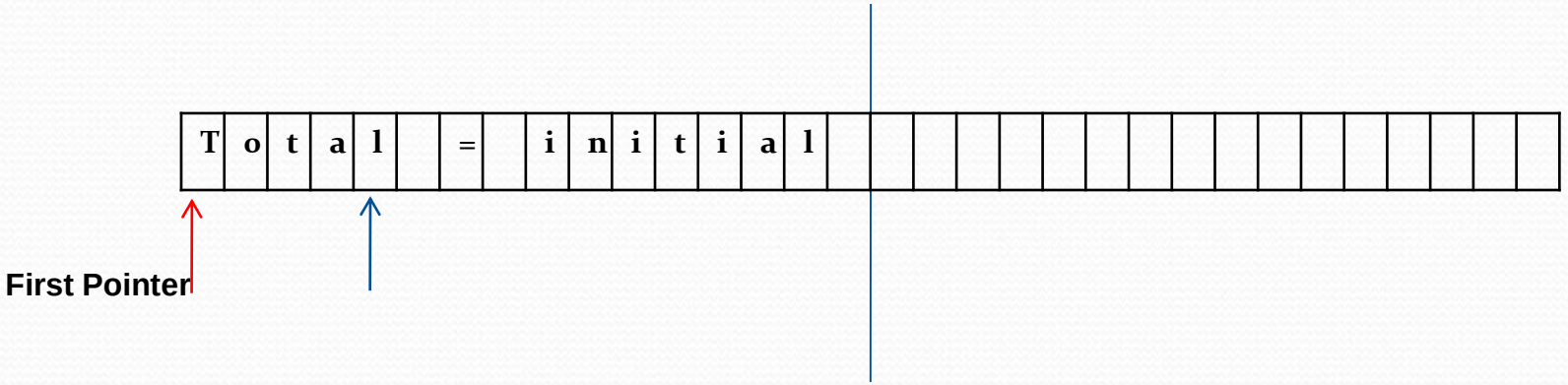
Input Buffering

Total = initial + r_o_i *year ;



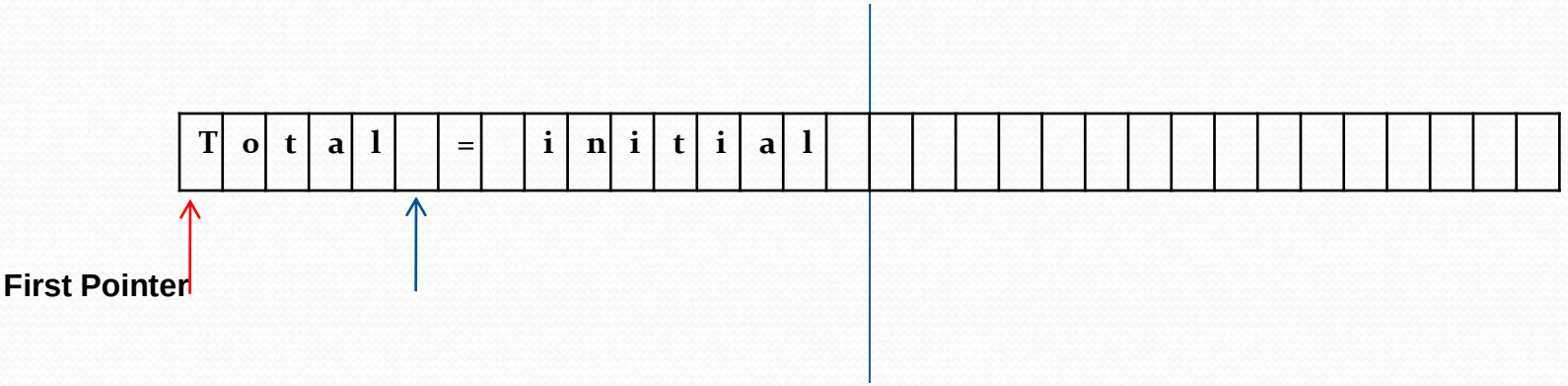
Input Buffering

Total = initial + r_o_i *year ;



Input Buffering

Total = initial + r_o_i *year ;



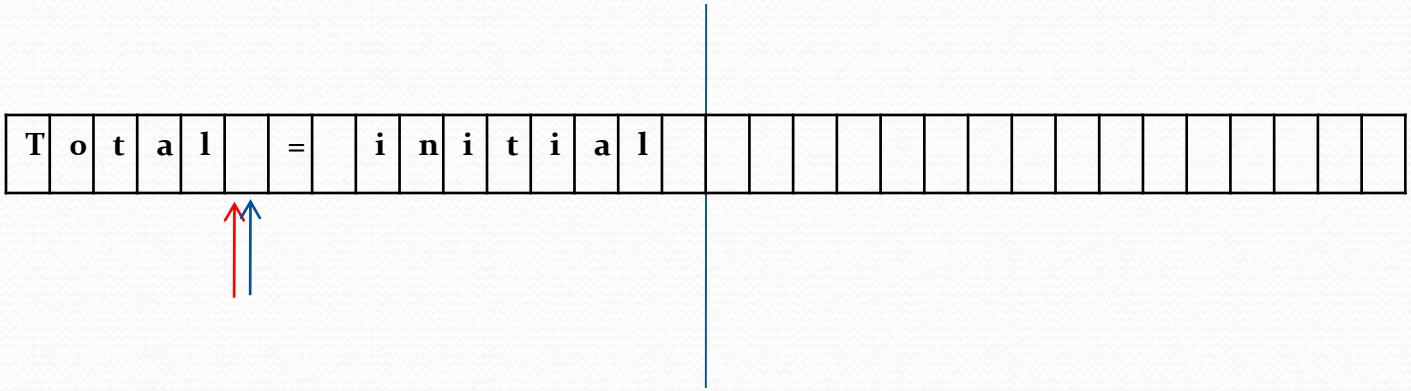
Matches with the pattern for id.

The string of characters between the two pointers is the current lexeme.

lexeme is Total.

Input Buffering

Total = initial + r_o_i *year ;

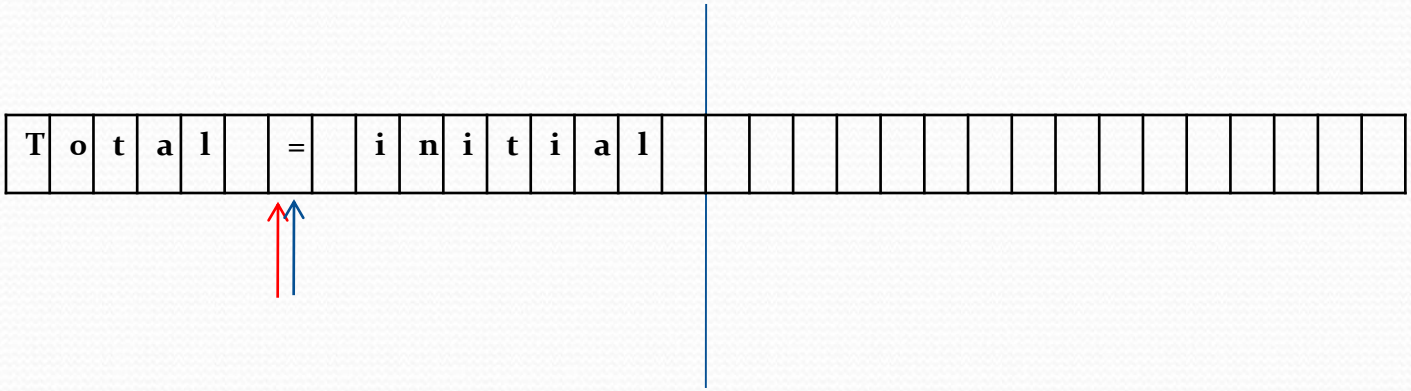


Once the next lexeme is determined, the forward pointer is set to the character at its right end.

After the lexeme is processed, both pointers are set to the character immediately past the lexeme.

Input Buffering

Total = initial + r_o_i *year ;

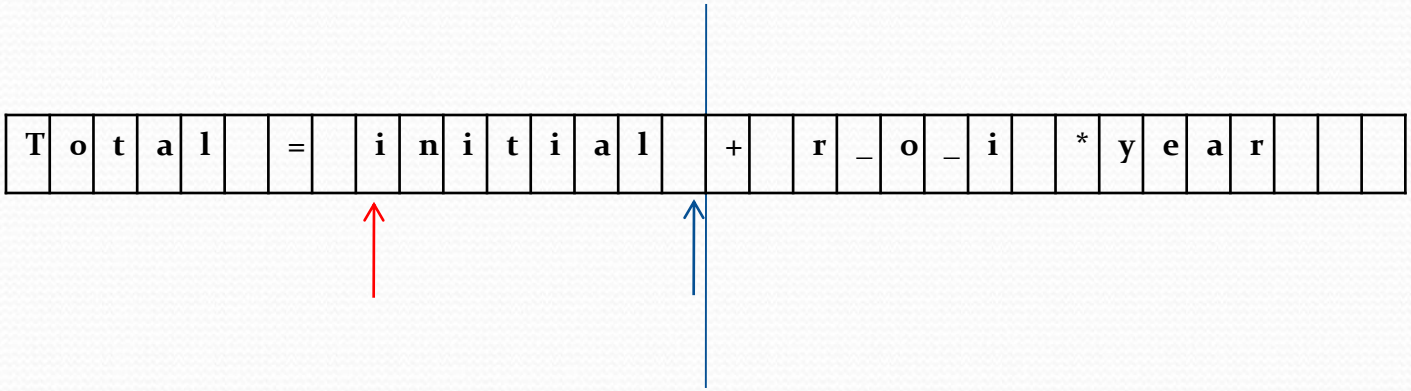


both pointers point to the first character of the next lexeme to be found.

Comment and white space can be treated as pattern that yield no token.

Input Buffering

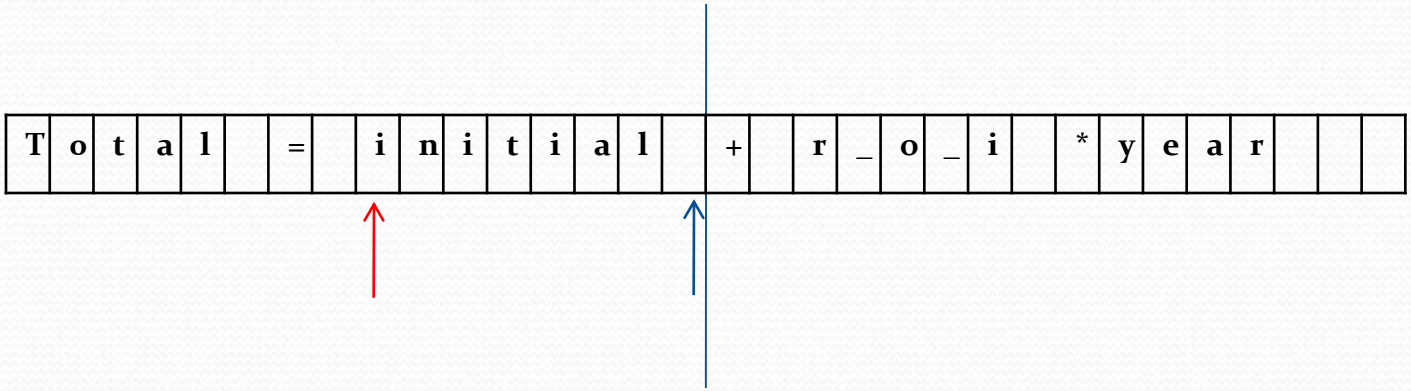
Total = initial + r_o_i *year ;



If the forward pointer is about to move half way mark, the right half is filled with n new input character.

Input Buffering

Total = initial + r_o_i *year ;

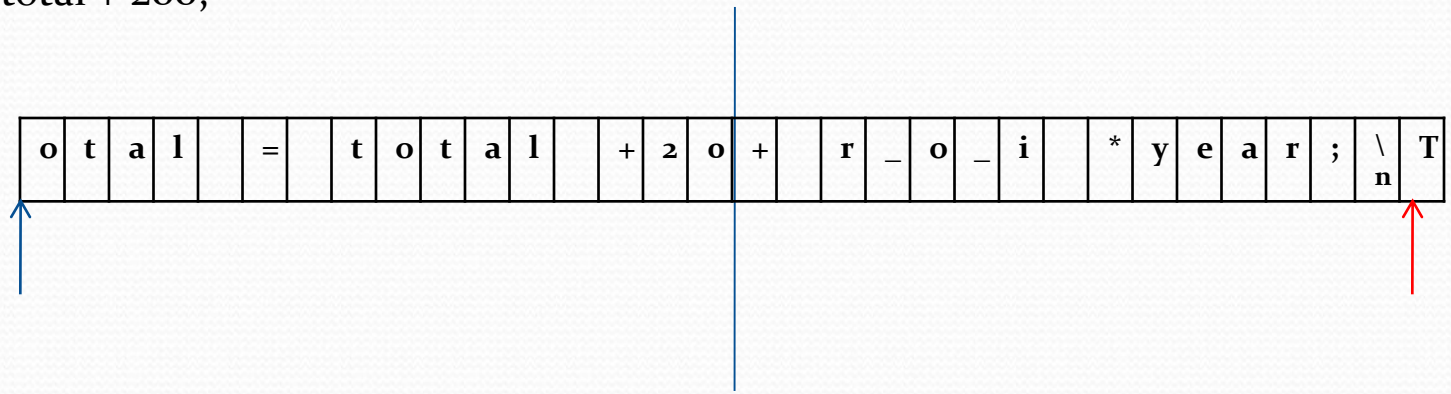


If the forward pointer is about to move half way mark, the right half is filled with n new input character.

Input Buffering

Total = initial + r_0_i *year ;

Total = total + 200;



If the forward pointer is about to move past the right end of the buffer, the left half is filled with N new characters and the forward pointer moves to the beginning of the buffer.

Input Buffering

Disadvantages:

The amount of lookahead is limited and this limited lookahead may make it impossible to recognize tokens in situations where the distance the forward pointer must travel is more than the length of the buffer.

Example: DECLARE(ARG1, ARG2, ...ARGn)

Here we can not determine whether DECLARE is a keyword or an array until we see the rightmost parenthesis. [Language PL/1]

Input Buffering--Algorithm

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

Specification of Token

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

Basic Terminology

- **Alphabet** : a finite set of symbols (ASCII characters)
- **String** :
  - Finite sequence of symbols on an alphabet
  - Sentence and word are also used in terms of string
  - ε is the empty string
  - |s| is the length of string s.
- **Language**: sets of strings over some fixed alphabet
  - ∅ the empty set is a language.
  - {ε} the set containing empty string is a language
  - The set of well-wormed C programs is a language
  - The set of all possible identifiers is a language.
- **Operators on Strings**:
  - *Concatenation*: xy represents the concatenation of strings x and y. $s\,\varepsilon = s$ $\varepsilon\, s = s$
  - $s^n = s\,s\,s\,..\,s$ (n times) $s^0 = \varepsilon$

Operations on Languages

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

Let L be the set of alphabet and D be the set of digits.

L∪D ----the set of letters and Digits.

LD ------the set of strings consisting of a letter followed by a digit.

L^* ------ Set of all strings of letters including ε.

$L(L \cup D)^*$ ------ set of all strings of letters and digits beginning with a letter.

Operations on Languages-Example

- $L_1 = \{a,b,c,d\}$ $L_2 = \{1,2\}$
- $L_1L_2 = \{a1,a2,b1,b2,c1,c2,d1,d2\}$ **{st | s is in** L_1 **and t is in** L_2**}**
- $L_1 \cup L_2 = \{a,b,c,d,1,2\}$ **{s | s is in** L_1 **or s is in** L_2 **}**
- L_1^3 = all strings with length three (using a,b,c,d)
- L_1^* = all strings using letters a,b,c,d and empty string
- L_1^+ = doesn't include the empty string

Regular Expression

- We use regular expressions to describe tokens of a programming language.
- A regular expression is built up of simpler regular expressions (using defining rules)
- Each regular expression denotes a language.
- A language denoted by a regular expression is called as a **regular set**.
- A **Regular Expression** is a Set of Rules / Techniques for Constructing Sequences of Symbols (Strings) From an Alphabet.
- Let Σ Be an Alphabet, r a Regular Expression Then L(r) is the Language That is Characterized by the Rules of r

Regular Expression

- ε is a regular expression, L(ε) = {ε}
- If a is a symbol in Σ then a is a regular expression, L(a) = {a}
- (r) | (s) is a regular expression denoting the language L(r) ∪ L(s)
- (r)(s) is a regular expression denoting the language L(r)L(s)
- (r)* is a regular expression denoting (L(r))*
- (r) is a regular expression denting L(r)
- **All are Left-Associative. Parentheses are dropped as allowed by precedence rules.**
- We may remove parentheses by using precedence rules.
  - * highest concatenation next | lowest

precedence

ab*|c means (a(b)*)|(c)

- Ex:Σ = {0,1}
  - 0|1 => {0,1}
  - (0|1)(0|1) => {00,01,10,11}
  - 0* => {ε ,0,00,000,0000,....}
  - (0|1)* => all strings with 0 and 1, including the empty string

Regular Expression-Example

- **All Strings that start with “tab” or end with “bat”:**
  tab{A,…,Z,a,…,z}*|{A,…,Z,a,….,z}*bat

- **All Strings in Which Digits 1,2,3 exist in ascending numerical order:**
  {A,…,Z}*1 {A,…,Z}*2 {A,…,Z}*3 {A,…,Z}*

Regular Definitions

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

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

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

$d_2 \rightarrow r_2$ r_i is a regular expression over symbols in

. $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$

$d_n \rightarrow r_n$

d_i is basic symbols r_i is previously defined names

Regular Definitions

- Ex: Identifiers in Pascal

  letter → A | B | … | Z | a | b | … | z

  digit → 0 | 1 | … | 9

  id → letter (letter | digit) * // regular definition

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

    (A|…|Z|a|…|z) ((A|…|Z|a|…|z) | (0|…|9)) * //More complex

- Ex: Unsigned numbers in Pascal

  digit → 0 | 1 | … | 9

  digits → digit +

  opt-fraction → . digits / ε

  opt-exponent → (E (+| - | ε) digits) | ε

  unsigned-num → digits opt-fraction opt-exponent

Try to match with 6.336E4 or 1.894E-4

Recognition of tokens

- **Shorthand Notation:**
- **"+" : one or more** $r^* = r^+ \mid \in$ **&** $r^+ = r\, r^*$
- **"?" : zero or one** $r^* = r \mid \in$
- **[range] : set range of characters (replaces "|")**
- **[A-Z] = A | B | C | ... | Z**

digit → 0 | 1 | ... | 9
digits → digit +
opt-fraction → (. digits)?
opt-exponent → (E (+| -) ? digits) ?
unsigned-num → digits opt-fraction opt-exponent

Recognition of tokens

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

  stmt -> if expr then stmt | if expr then stmt else stmt | ε

  expr -> term relop term | term

  term -> id | number

- 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)+ //If a match of **ws** is found lexical analyzer does not return a token.

Recognition of tokens

- Aim: Construct a lexical analyzer that will isolate the lexeme for the next token in the input buffer and produce an output a pair consisting the appropriate token and attribute value, using the following transitional table.

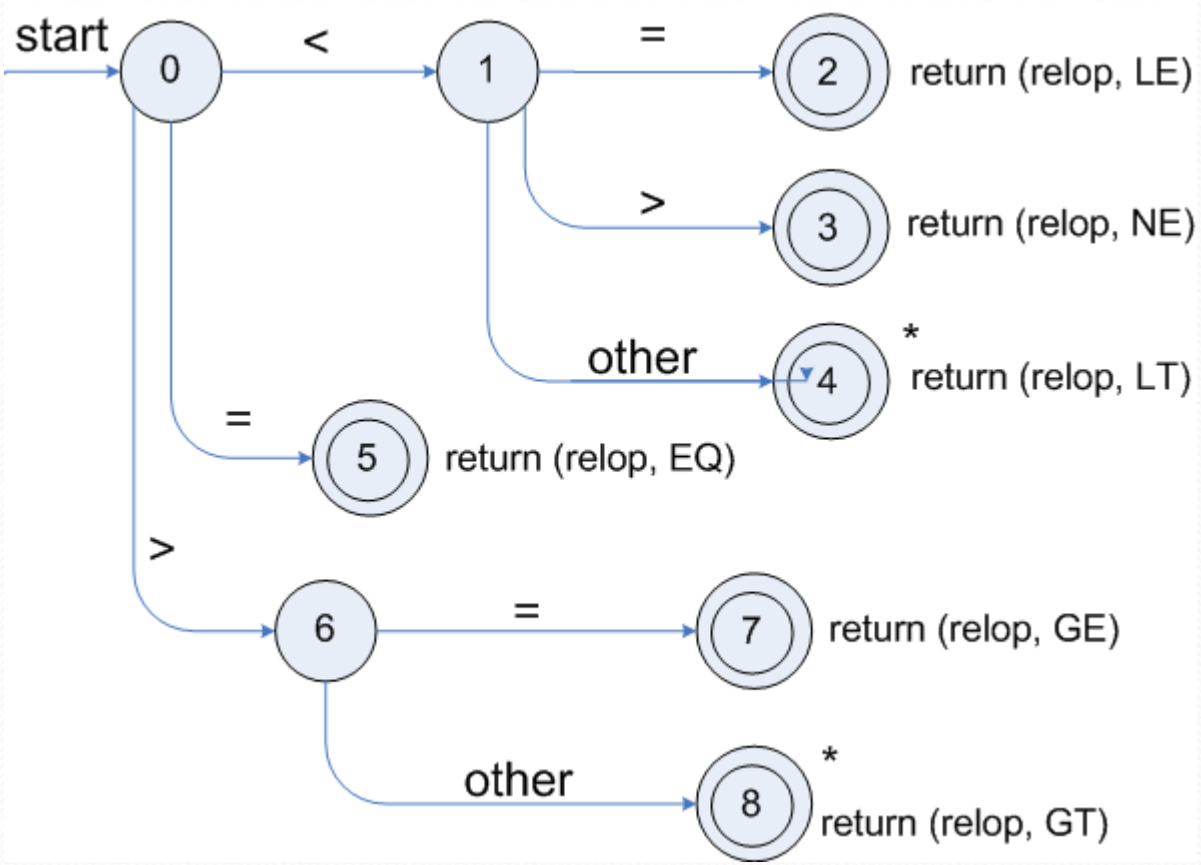
- Construct transitional diagram.

Transitional Diagram

- Transition Diagrams (TD) are used to represent the tokens
- As characters are read, the relevant TDs are used to attempt to match lexeme to a pattern
- Each TD has:
  - States (Q) : Represented by Circles
  - Inputs (∑):
  - Actions (δ): Represented by Arrows between states
  - Start State (qo): Beginning of a pattern (Arrowhead)
  - Final State(F) : End of pattern (Concentric Circles)
- Each TD is Deterministic - No need to choose between 2 different actions !

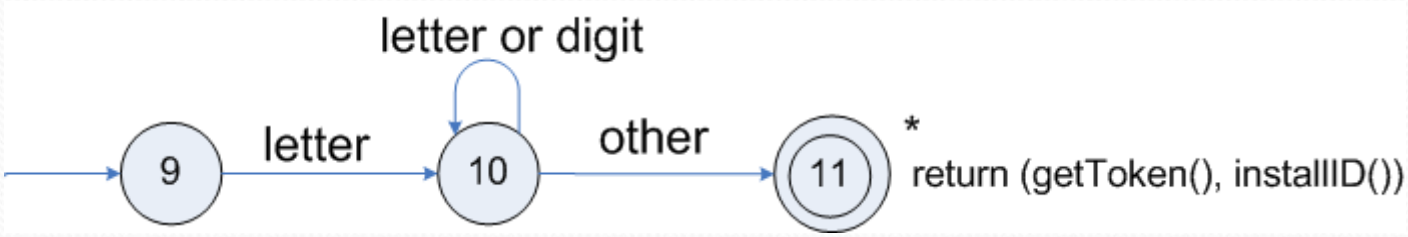
Transition diagrams

- Transition diagram for relop



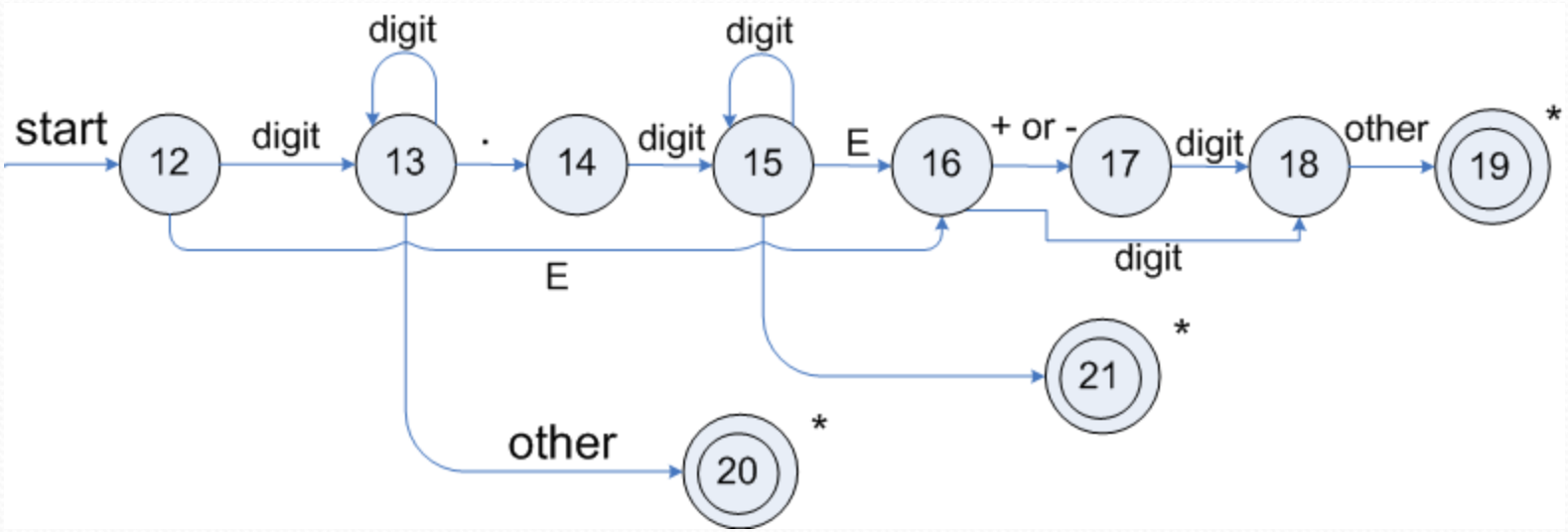
Transition diagrams (cont.)

- Transition diagram for reserved words and identifiers



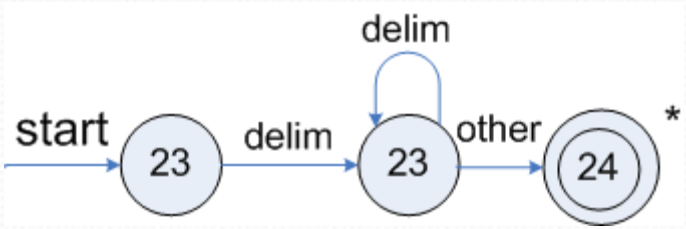
Transition diagrams (cont.)

- Transition diagram for unsigned numbers



Transition diagrams (cont.)

- Transition diagram for whitespace

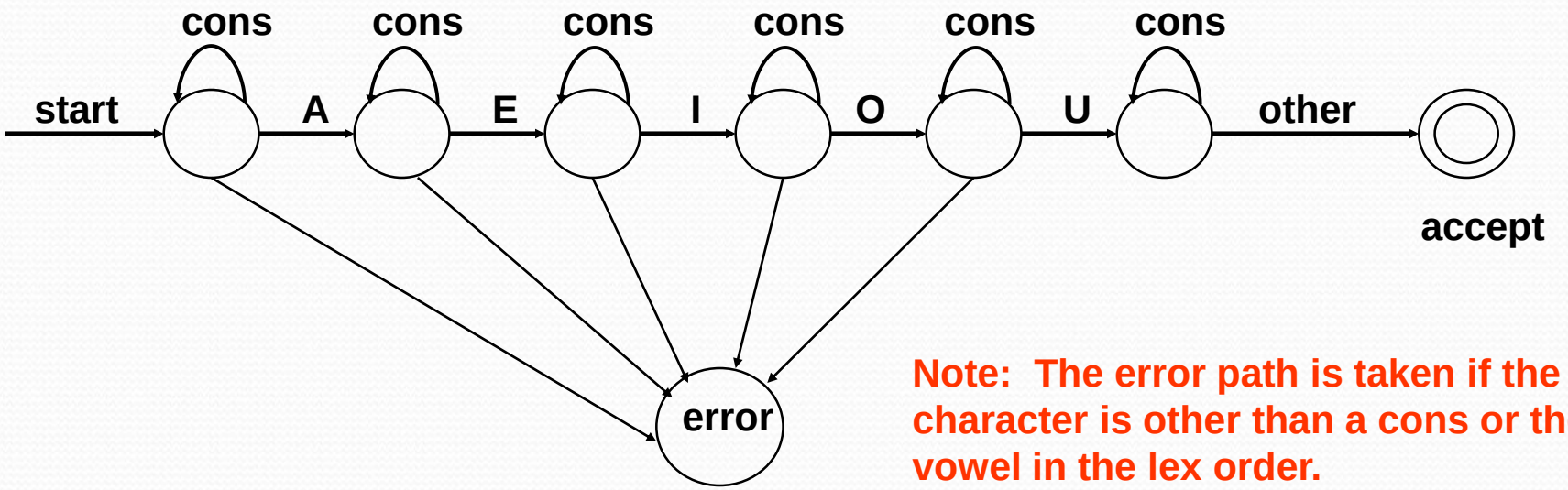


Transitional Diagram-Example

- **What would the transition diagram (TD) for strings containing each vowel, in their strict lexicographical order, look like ?**

cons → B | C | D | F | ... | Z

string → cons* A cons* E cons* I cons* O cons* U cons*



Note: The error path is taken if the character is other than a cons or the vowel in the lex order.

Architecture of a transition-diagram-based lexical analyzer

```
TOKEN getRelop()
{
    TOKEN retToken = new (RELOP)
    while (1) {                /* repeat character processing until a
                                          return or failure occurs  */
    switch(state) {
            case 0: c= nextchar();
                        if (c == '<') state = 1;
                        else if (c == '=') state = 5;
                        else if (c == '>') state = 6;
                        else fail();  /* lexeme is not a relop */
                        break;
            case 1: ...
            ...
            case 8: retract();
                        retToken.attribute = GT;
                        return(retToken);
    }
```

Finite Automata

- A ***recognizer*** for a language is a program that takes a string x, and answers "yes" if x is a sentence of that language, and "no" otherwise.
- We call the recognizer of the tokens as a ***finite automaton*****.**
- A finite automaton can be: ***deterministic(DFA)*** or ***non-deterministic*** *(NFA)*
- This means that we may use a deterministic or non-deterministic automaton as a lexical analyzer.
- Both deterministic and non-deterministic finite automaton recognize regular sets.

Non-Deterministic : Has more than one alternative action for the same input symbol.

Deterministic : Has at most one action for a given input symbol.

- Which one?
  - deterministic – faster recognizer, but it may take more space
  - non-deterministic – slower, but it may take less space
  - Deterministic automatons are widely used lexical analyzers.

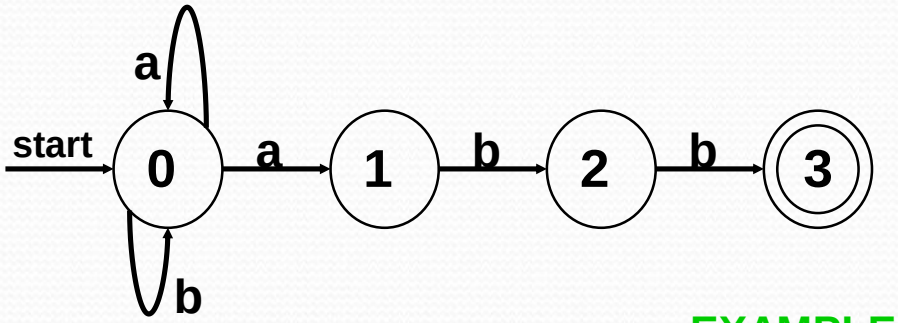
First, we define regular expressions for tokens; Then we convert them into a DFA to get a lexical analyzer for our tokens.

- Algorithm1: Regular Expression ➔ NFA ➔ DFA (two steps: first to NFA, then to DFA)
- Algorithm2: Regular Expression ➔ DFA (directly convert a regular expression into a DFA)

Non-Deterministic Finite Automaton (NFA)

- A non-deterministic finite automaton (NFA) is a mathematical model that consists of:
  - Q - a set of states
  - Σ - a set of input symbols (alphabet)
  - δ– a transition function move to map state-symbol pairs to sets of states. $Q \times \Sigma \rightarrow 2^Q$
  - q_o - a start (initial) state
  - F – a set of accepting states (final states)
- ε- transitions are allowed in NFAs. In other words, we can move from one state to another one without consuming any symbol.

How Does An NFA Work ?



- Given an input string, we trace moves
- If no more input & in final state, ACCEPT

EXAMPLE: **Input:**
ababb

***move*(0, a) = 1**
***move*(1, b) = 2**
***move*(2, a) = ? (undefined)**

REJECT !

-OR-

***move*(0, a) = 0**
***move*(0, b) = 0**
***move*(0, a) = 1**
***move*(1, b) = 2**
***move*(2, b) = 3**
ACCEPT !

Problems with NFAs for Regular Expressions:

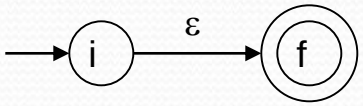
1. Valid input might not be accepted
2. NFA may behave differently on the same input

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

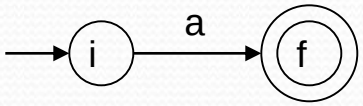
- This is one way to convert a regular expression into a NFA.
- Thomson's Construction is simple and systematic method.
- It guarantees that the resulting NFA will have exactly one final state, and one start state.
- Construction starts from simplest parts (alphabet symbols).
- To create a NFA for a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

Thomson's Construction (cont.)

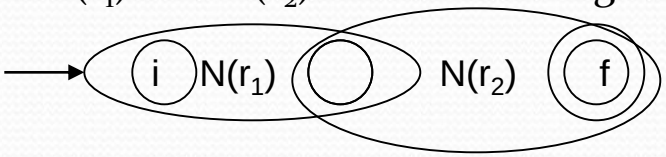
- To recognize an empty string ε



- To recognize a symbol a in the alphabet Σ

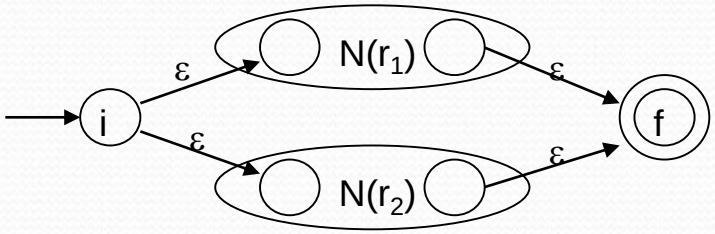


- If $N(r_1)$ and $N(r_2)$ are NFAs for regular expressions r_1 and r_2

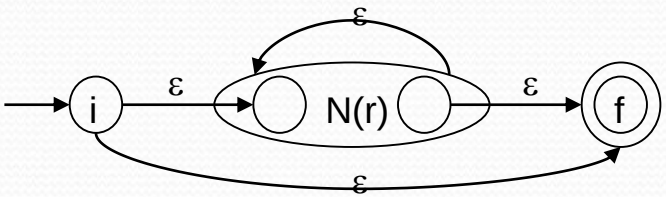


Final state of $N(r_2)$ become final state of $N(r_1r_2)$

- For regular expression $r_1 \mid r_2$



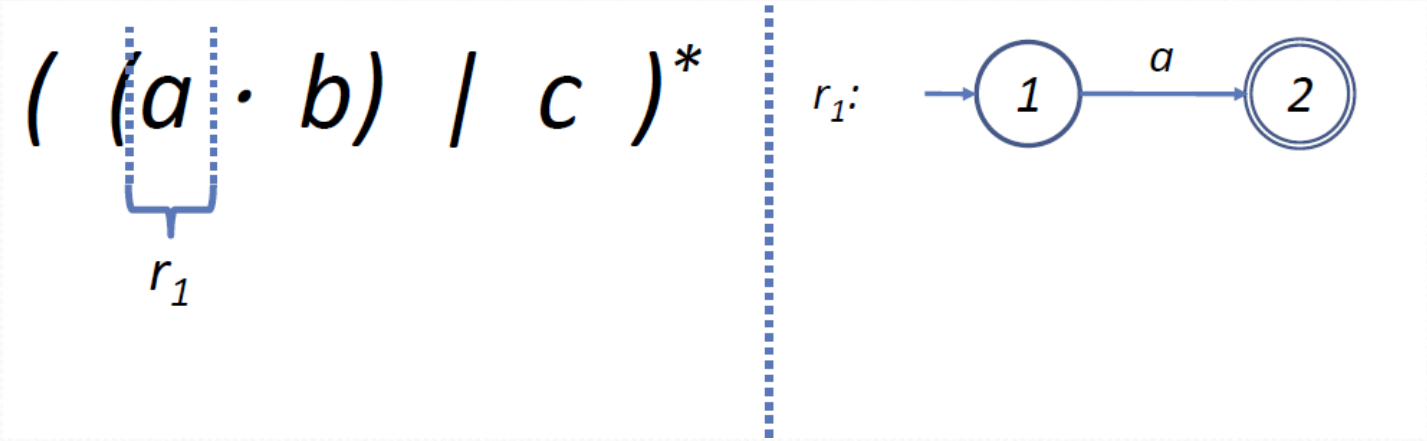
For regular expression r^*



Thomson's Construction-Example-1

- Construct the corresponding NFA with ε move for the given RE.

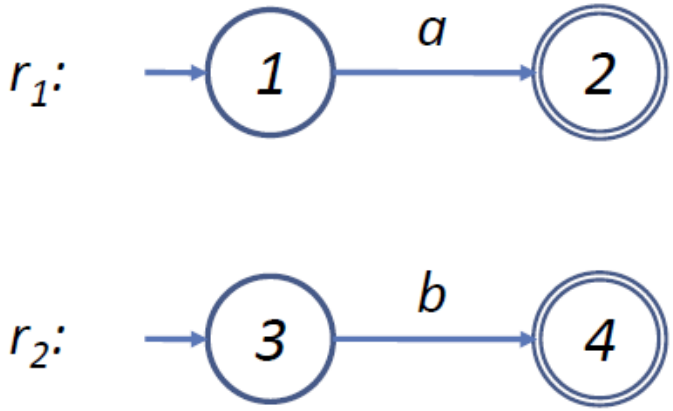
((a.b)|c)*



Thomson's Construction-Example-1

$((a \cdot b) \mid c)^*$

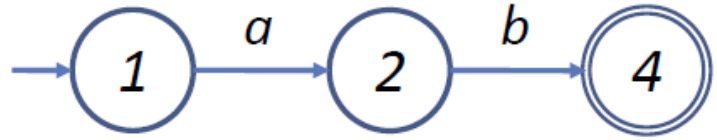
r_1 r_2



Thomson's Construction-Example-1

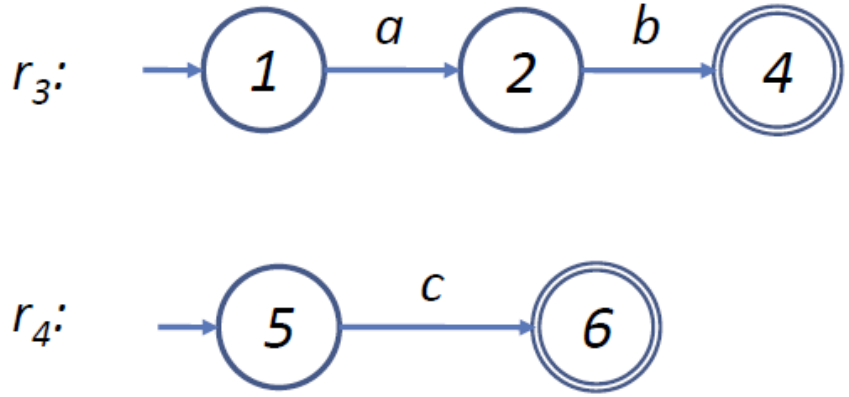
$$(\underbrace{(a \cdot b)}_{r_3} \mid c)^*$$

r_3:



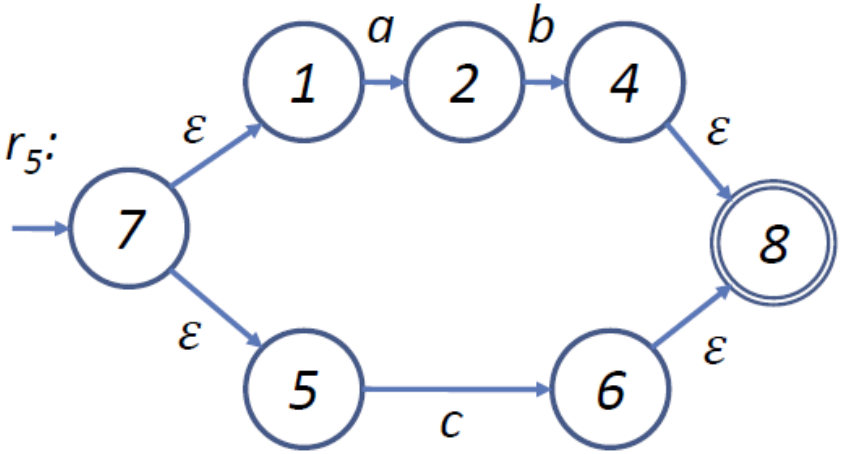
Thomson's Construction-Example-1

$$(\underbrace{(a \cdot b)}_{r_3} \mid \underbrace{c}_{r_4})^*$$

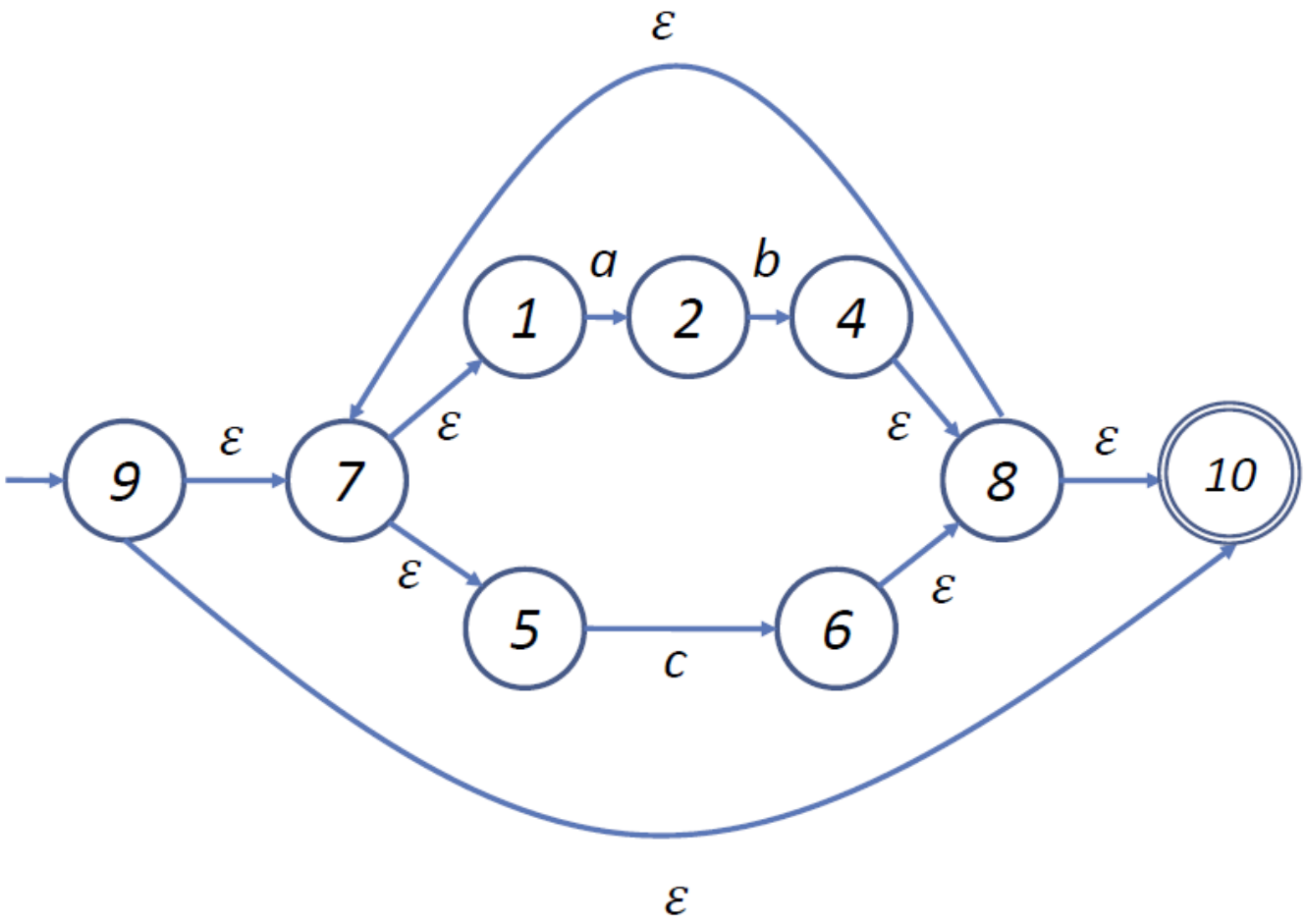


Thomson's Construction-Example-1

$$(\underbrace{(a \cdot b) \mid c}_{r_5})^*$$



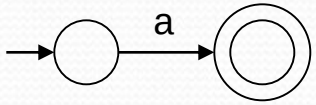
Thomson's Construction-Example-1



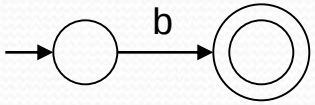
Thomson's Construction-Example-2

- Construct NFA for (a|b) * abb

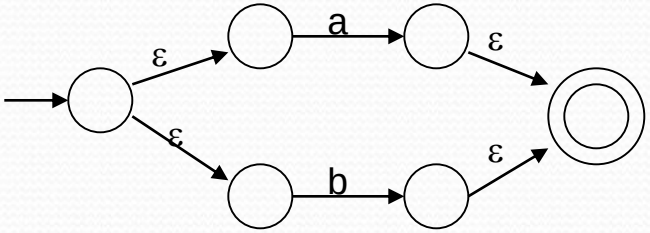
a:



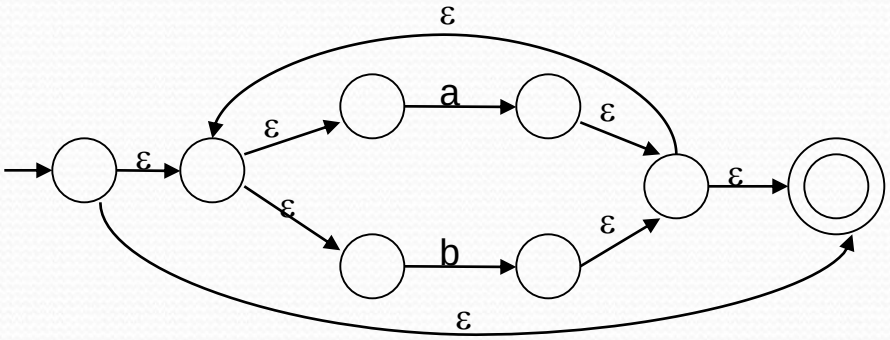
b:



(a | b)

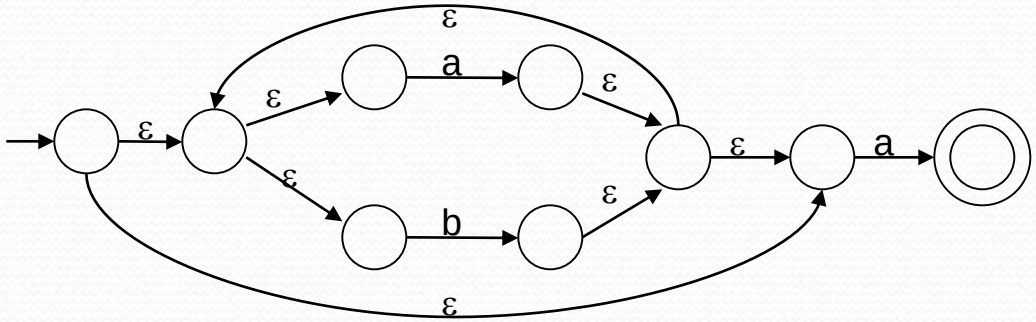


(a|b) *

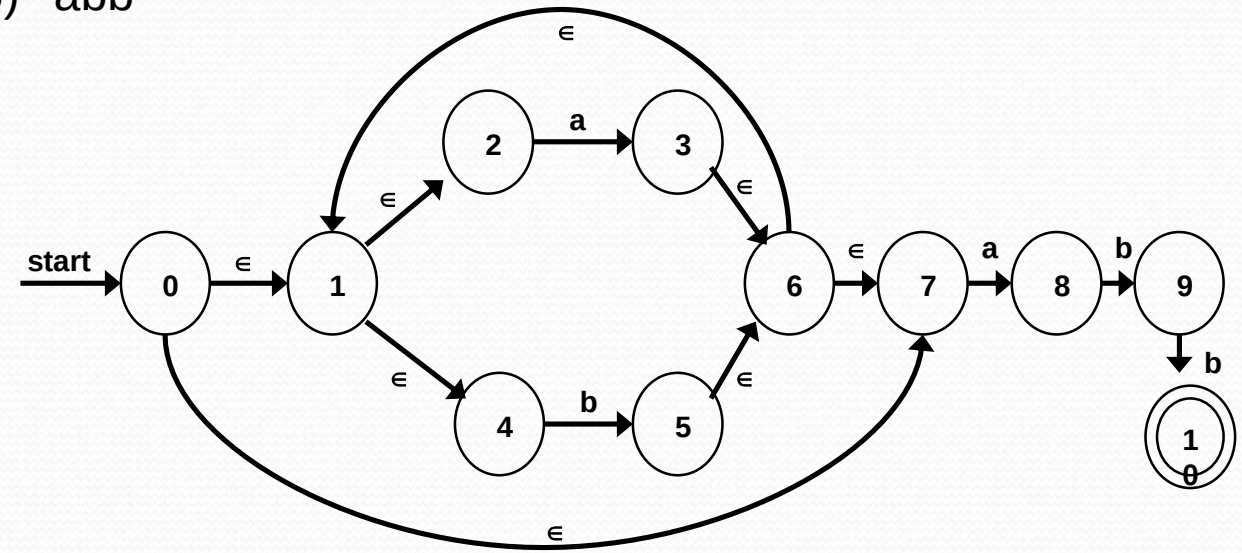


Thomson's Construction-Example-2

(a|b) * a

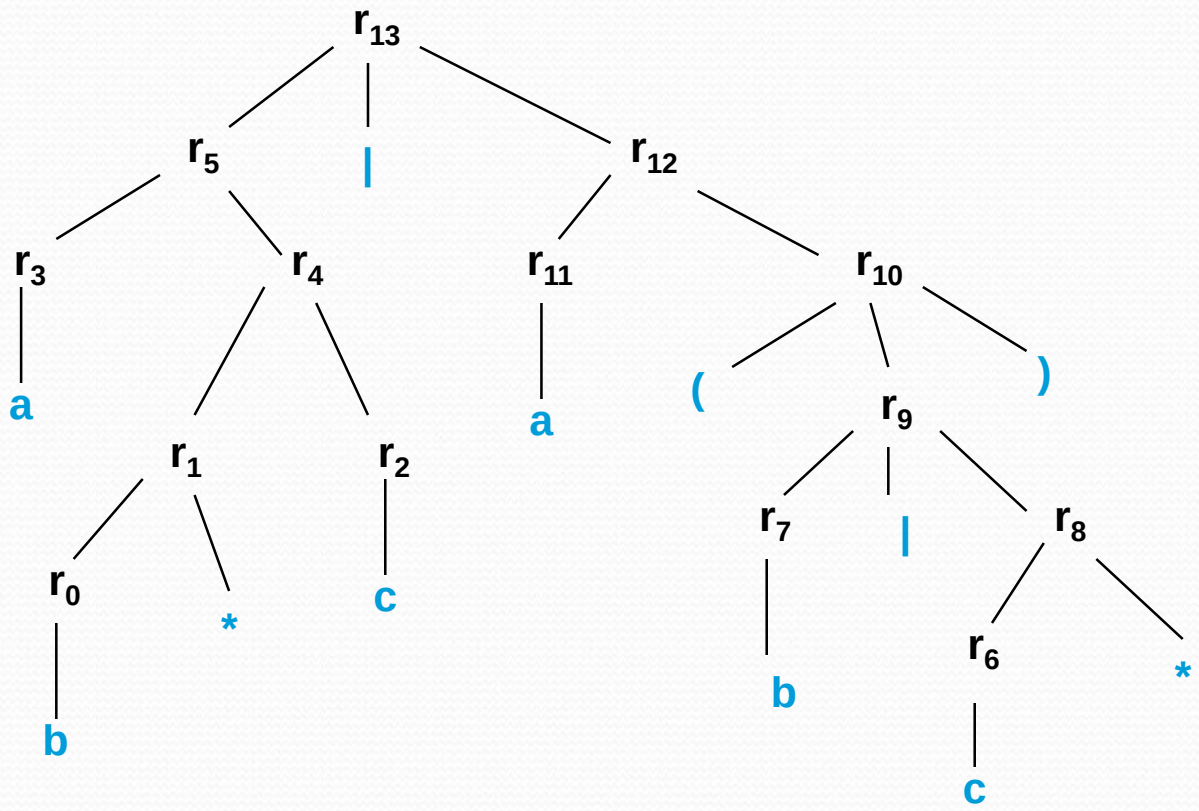


(a|b) * abb

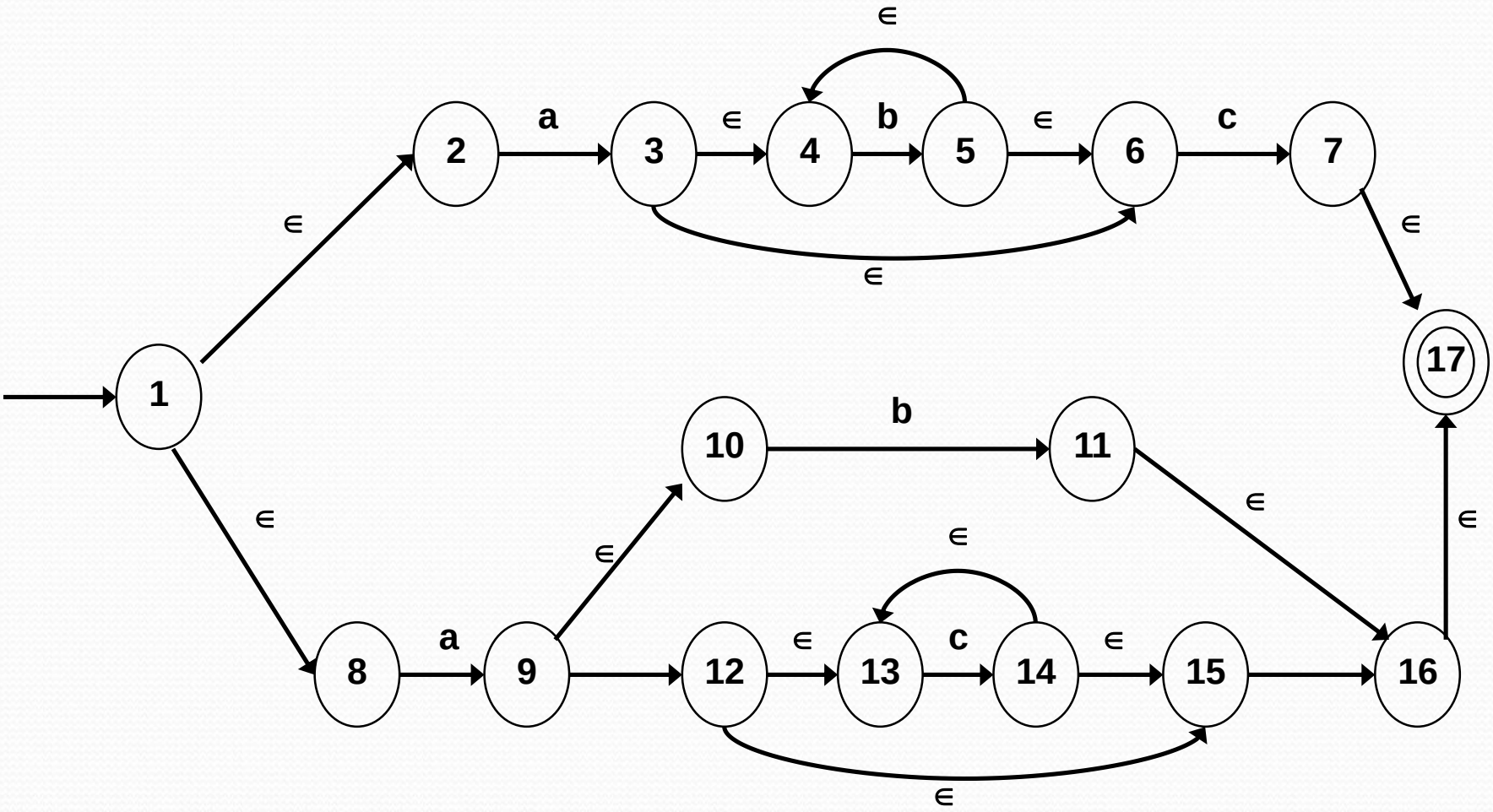


Exercise

- Construct RE for **(ab*c) | (a(b|c*))** using Thompson Construction.
- Parse tree for the Regular Expression



Exercise



Assignment

- Follow the website
- http://hackingoff.com/compilers/regular-expression-to-nfa-dfa

- Write a program [Using C graphics or Java Swing] to convert a RE to NFA

Deterministic Finite Automaton (DFA)

- A Deterministic Finite Automaton (DFA) is a special form of a NFA
- no state has ε- transition
- for each symbol a and state s, there is at most one labeled edge a leaving s.
- i.e. transition function is from pair of state-symbol to state (not set of states)

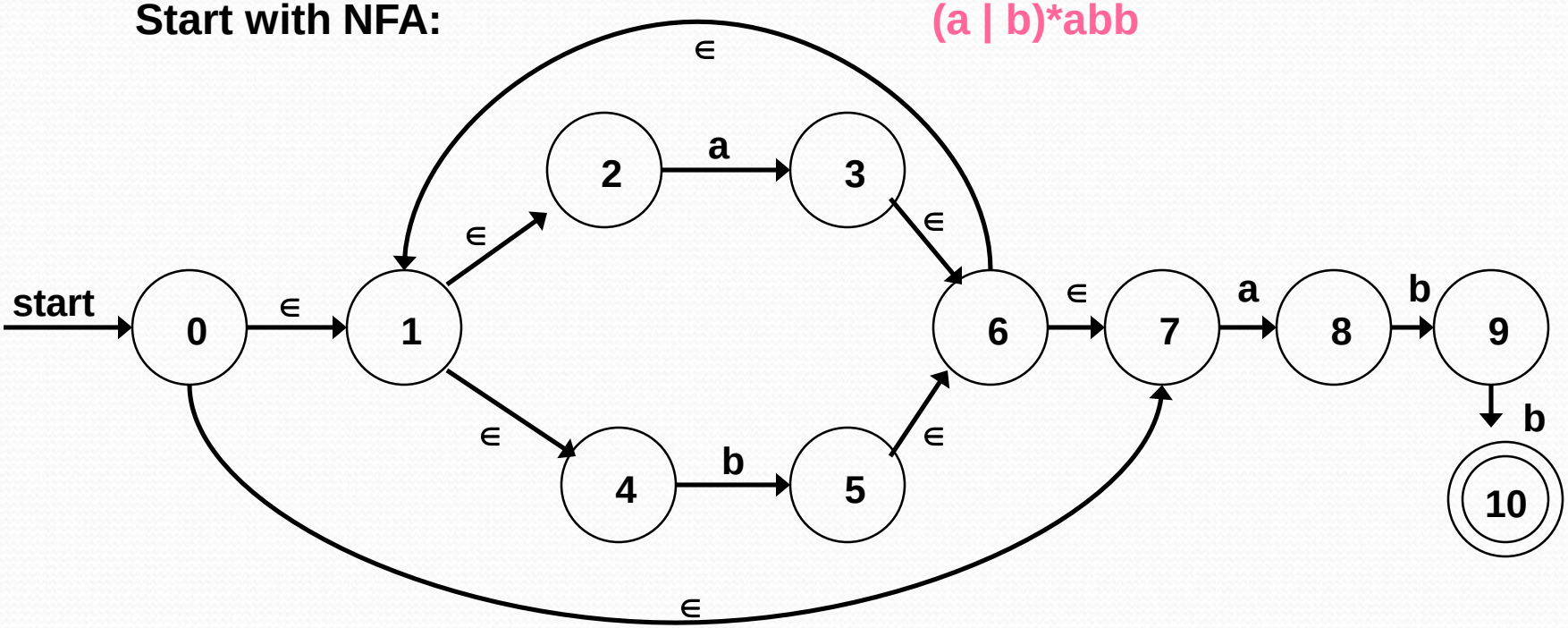
Converting a NFA into a DFA (subset construction)

put ε-closure($\{s_0\}$) as an unmarked state into the set of DFA (DS)
while (there is one unmarked S_1 in DS) do
    begin
        mark S_1
        for each input symbol a do
            begin
                $S_2 \leftarrow$ ε-closure(move(S_1,a))
                if (S_2 is not in DS) then
                        add S_2 into DS as an unmarked state
                transfunc[S_1,a] $\leftarrow S_2$
            end
        end

- a state S in DS is an accepting state of DFA if a state in S is an accepting state of NFA
- the start state of DFA is ε-closure($\{s_0\}$)

Illustrating Conversion – Example-1

Start with NFA: **(a | b)*abb**



First we calculate: ∈-closure(0) (i.e., state 0)

∈-closure(0) = {0, 1, 2, 4, 7} (all states reachable from 0 on ∈-moves)

Let A={0, 1, 2, 4, 7} be a state of new DFA, D.

Illustrating Conversion – Example-1

2nd , we calculate : a : ∈-closure(*move*(A,a)) and
b : ∈-closure(*move*(A,b))

a : ∈-closure(*move*(A,a)) = ∈-closure(*move*({0,1,2,4,7},a))}
adds {3,8} (since *move*(2,a)=3 and *move*(7,a)=8)

From this we have : ∈-closure({3,8}) = {1,2,3,4,6,7,8}
(since 3→6 →1 →4, 6 →7, and 1 →2 all by ∈-moves)

Let B={1,2,3,4,6,7,8} be a new state. Define Dtran[A,a] = B.

b : ∈-closure(*move*(A,b)) = ∈-closure(*move*({0,1,2,4,7},b))

adds {5} (since *move*(4,b)=5)

From this we have : ∈-closure({5}) = {1,2,4,5,6,7}
(since 5→6 →1 →4, 6 →7, and 1 →2 all by ∈-moves)

Let C={1,2,4,5,6,7} be a new state. Define Dtran[A,b] = C.

Illustrating Conversion – Example-1

3rd , we calculate for state B on {a,b}

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

Define Dtran[B,a] = B.

b : ∈-closure(*move*(B,b)) = ∈-closure(*move*({1,2,3,4,6,7,8},b))}
= {1,2,4,5,6,7,9} = D

Define Dtran[B,b] = D.

4th , we calculate for state C on {a,b}

a : ∈-closure(*move*(C,a)) = ∈-closure(*move*({1,2,4,5,6,7},a))}
= {1,2,3,4,6,7,8} = B

Define Dtran[C,a] = B.

b : ∈-closure(*move*(C,b)) = ∈-closure(*move*({1,2,4,5,6,7},b))}
= {1,2,4,5,6,7} = C

Define Dtran[C,b] = C.

Illustrating Conversion – Example-1

5th , we calculate for state D on {a,b}

a : ∈-closure(*move*(D,a)) = ∈-closure(*move*({1,2,4,5,6,7,9},a))}
= {1,2,3,4,6,7,8} = B

Define Dtran[D,a] = B.

b : ∈-closure(*move*(D,b)) = ∈-closure(*move*({1,2,4,5,6,7,9},b))}
= {1,2,4,5,6,7,10} = E

Define Dtran[D,b] = E.

Finally, we calculate for state E on {a,b}

a : ∈-closure(*move*(E,a)) = ∈-closure(*move*({1,2,4,5,6,7,10},a))}
= {1,2,3,4,6,7,8} = B

Define Dtran[E,a] = B.

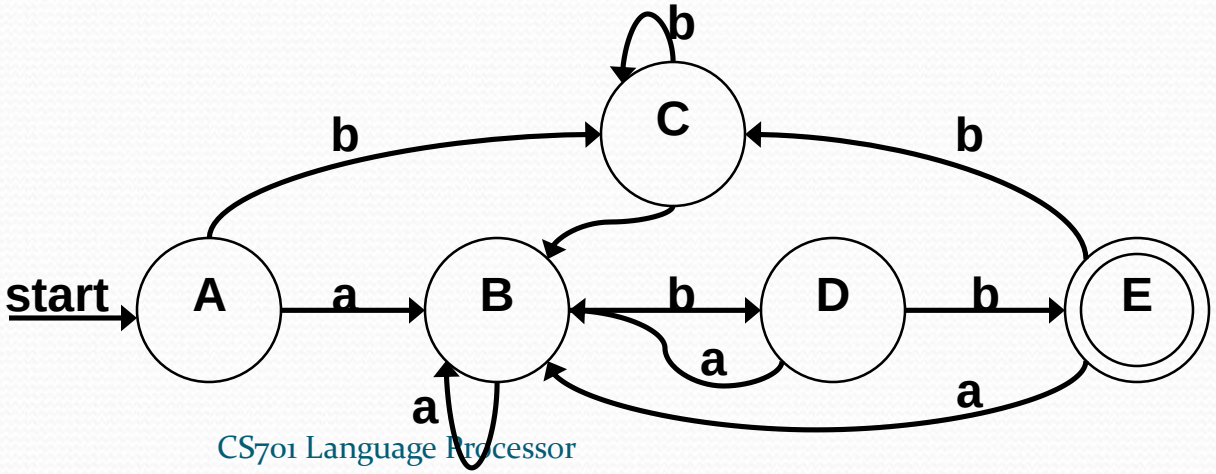
b : ∈-closure(*move*(*E*,b)) = ∈-closure(*move*({1,2,4,5,6,7,10},b))}
= {1,2,4,5,6,7} = C

Define Dtran[E,b] = C.

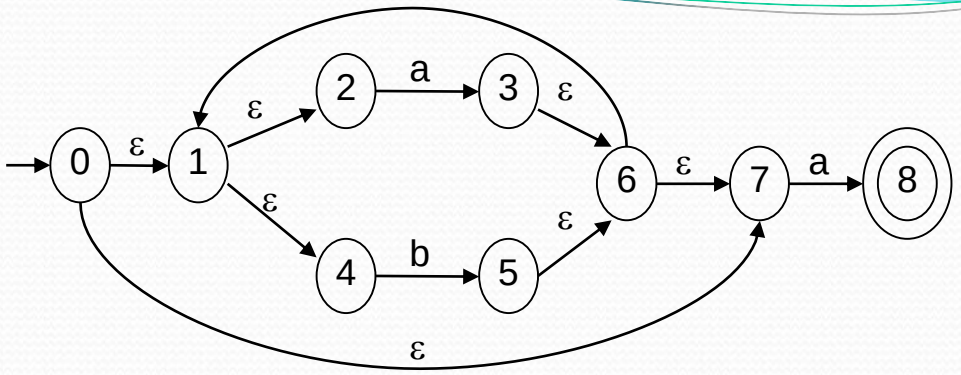
Illustrating Conversion – Example-1

This gives the transition table Dtran for the DFA of:

Dstates	Input Symbol	
	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C



Illustrating Conversion – Example-2



S_0 = ε-closure({0}) = {0,1,2,4,7} S_0 into DS as an unmarked state

⇓ mark S_0

ε-closure(move(S_0,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = S_1 S_1 into DS

ε-closure(move(S_0,b)) = ε-closure({5}) = {1,2,4,5,6,7} = S_2 S_2 into DS

transfunc[S_0,a] ← S_1 transfunc[S_0,b] ← S_2

⇓ mark S_1

ε-closure(move(S_1,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = S_1

ε-closure(move(S_1,b)) = ε-closure({5}) = {1,2,4,5,6,7} = S_2

transfunc[S_1,a] ← S_1 transfunc[S_1,b] ← S_2

⇓ mark S_2

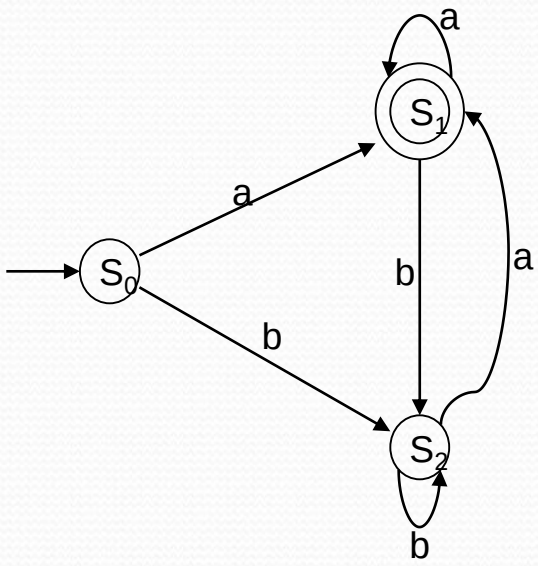
ε-closure(move(S_2,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = S_1

ε-closure(move(S_2,b)) = ε-closure({5}) = {1,2,4,5,6,7} = S_2

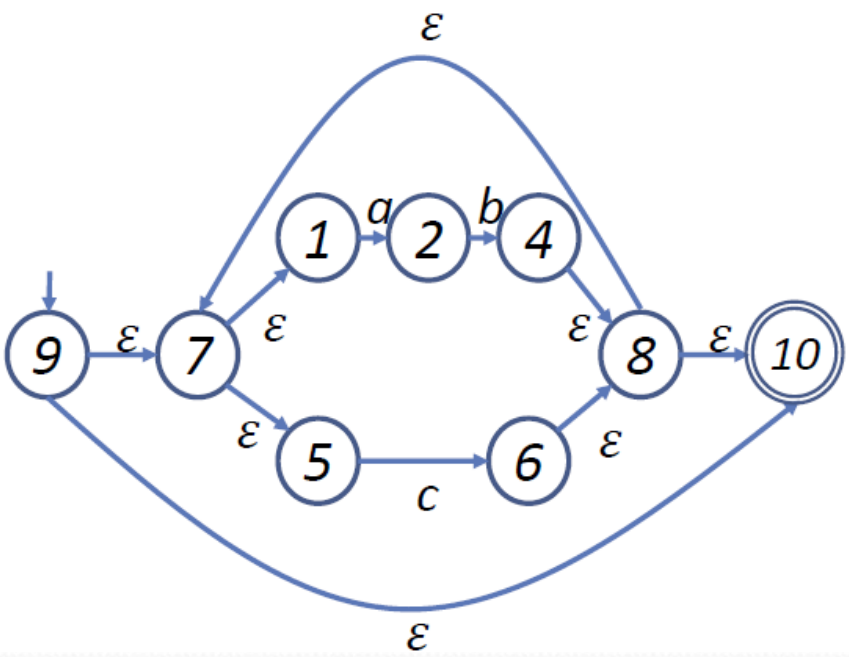
transfunc[S_2,a] ← S_1 transfunc[S_2,b] ← S_2

Illustrating Conversion – Example-2

S_0 is the start state of DFA since 0 is a member of S_0={0,1,2,4,7}
S_1 is an accepting state of DFA since 8 is a member of S_1 = {1,2,3,4,6,7,8}



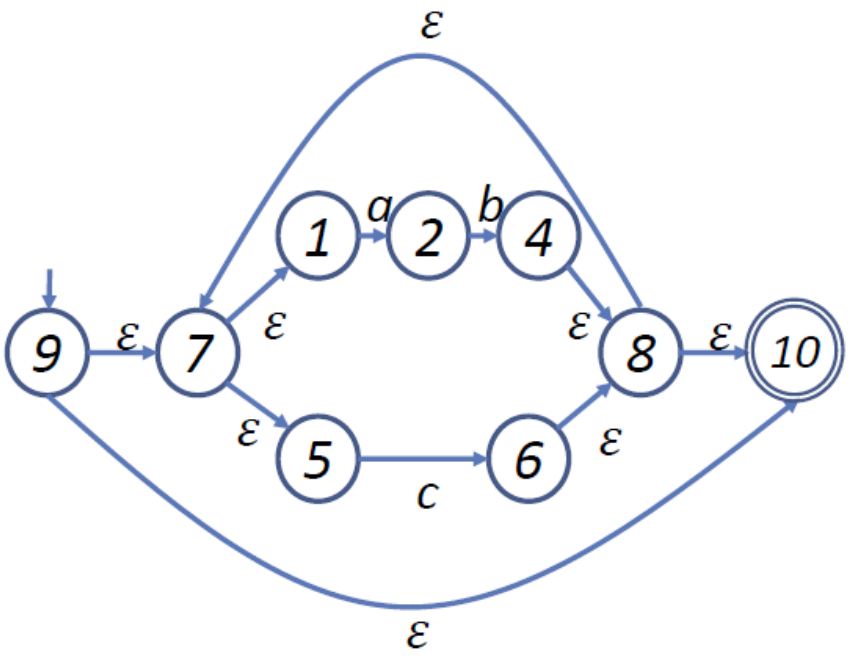
Illustrating Conversion – Example-3



D_{states}

NFA States	DFA State	Next State		
		a	*b*	*c*

Illustrating Conversion – Example-3



D_{states}

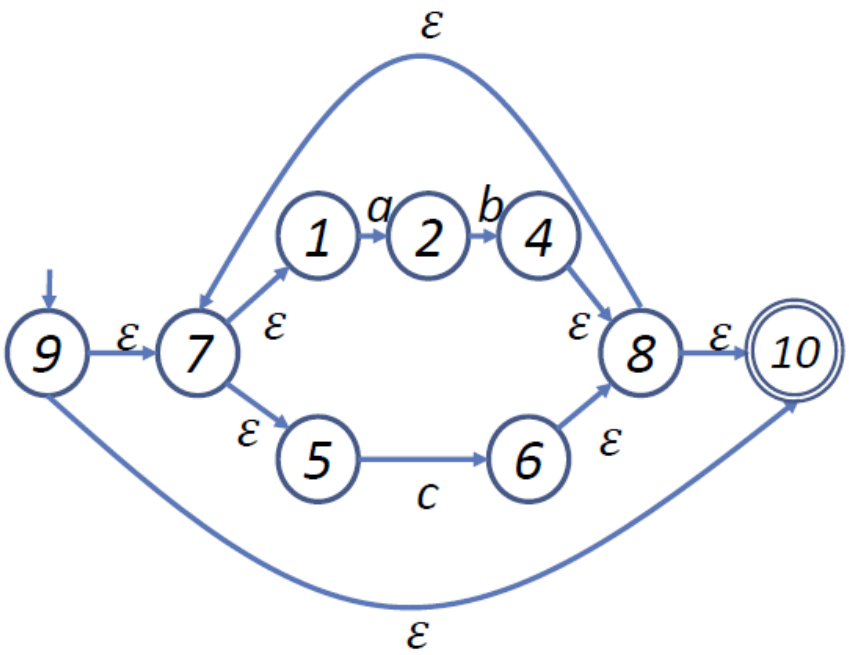
NFA States	DFA State	Next State		
		a	*b*	*c*
{9,7,1,5,10}	A			

ε-closure(move(A,a)) = ε-closure({2}) = {2} = B

ε-closure(move(A,b)) = {ϕ}

ε-closure(move(A,c)) = ε-closure({6}) = {6,8,10,7,1,5} =C

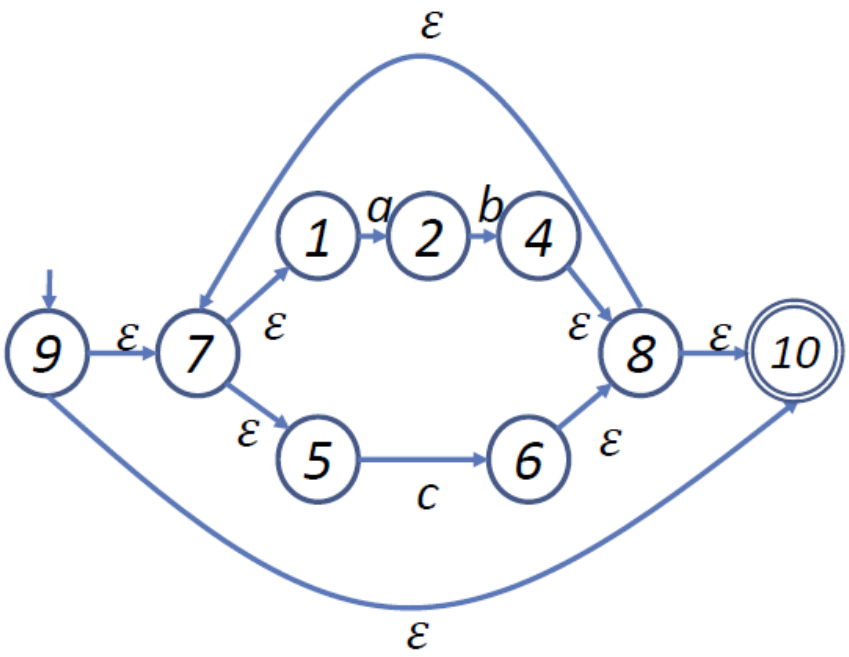
Illustrating Conversion – Example-3



D_{states}

NFA States	DFA State	Next State		
		a	*b*	*c*
{9,7,1,5,10}	A ✓	B		
{2}	B			

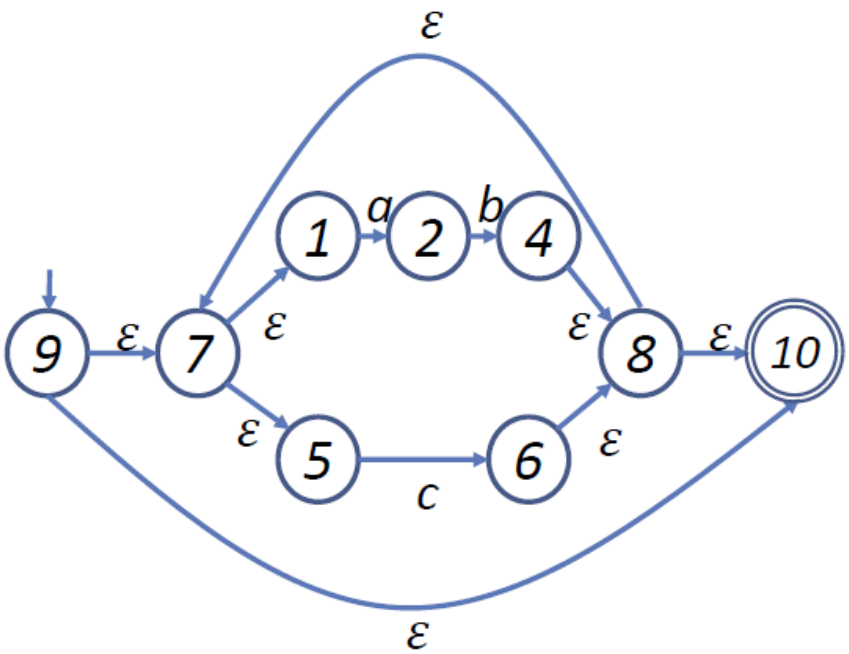
Illustrating Conversion – Example-3



D_{states}

NFA States	DFA State	Next State		
		a	*b*	*c*
{9,7,1,5,10}	A ✓	B	-	C
{2}	B			
{6,8,10,7,1,5}	C			

Illustrating Conversion – Example-3



D_{states}

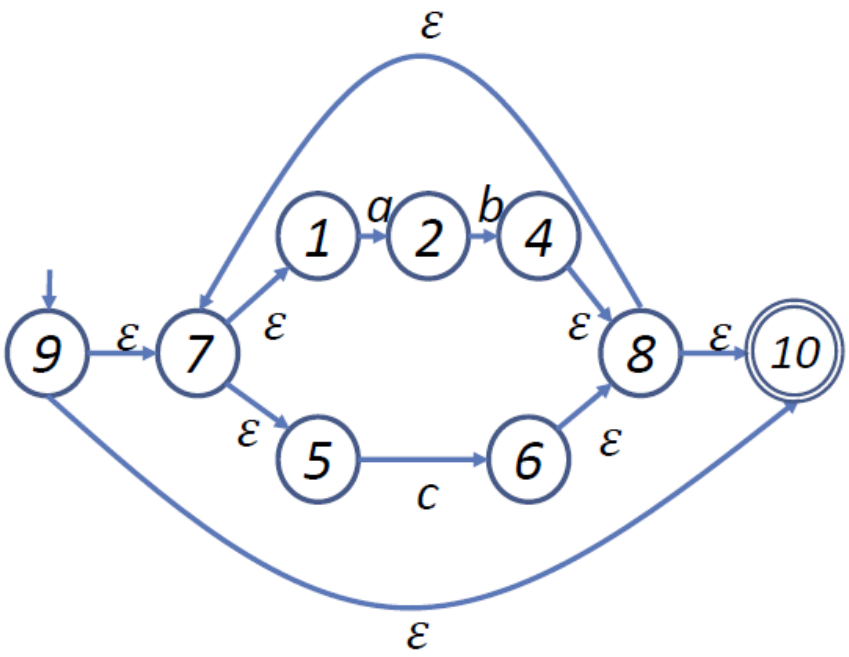
NFA States	DFA State	Next State		
		a	*b*	*c*
{9,7,1,5,10}	A ✓	B	-	C
{2}	B ✓	-	D	-
{6,8,10,7,1,5}	C			
{4,8,7,1,5,10}	D			

ε-closure(move(B,a)) = {ϕ}

ε-closure(move(B,b)) = ε-closure({4}) = {1,4,5,7,8,10} =D

ε-closure(move(A,c)) = {ϕ}

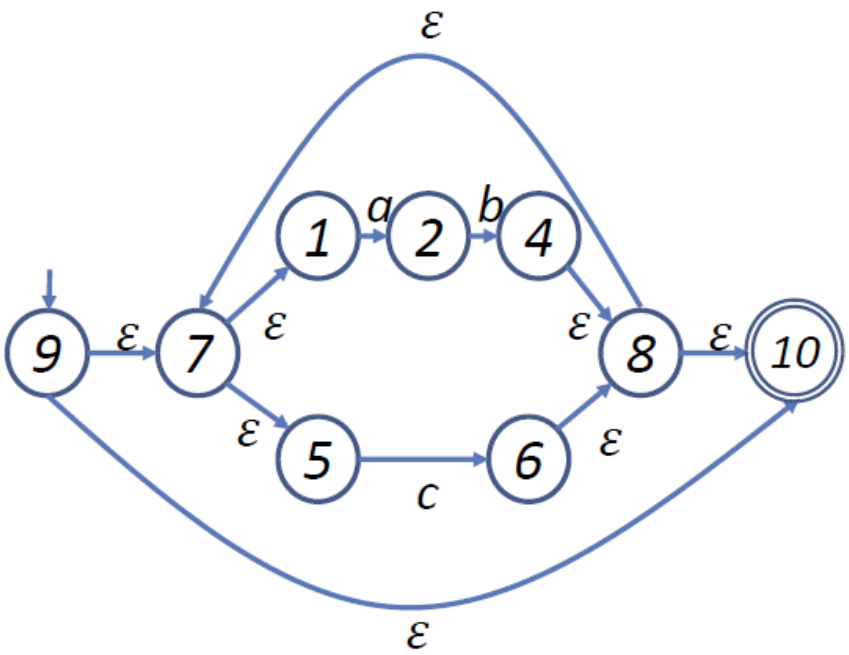
Illustrating Conversion – Example-3



D_{states}

NFA States	DFA State	Next State		
		a	*b*	*c*
{9,7,1,5,10}	A ✓	B	-	C
{2}	B ✓	-	D	-
{6,8,10,7,1,5}	C ✓	B	-	C
{4,8,7,1,5,10}	D			

Illustrating Conversion – Example-3

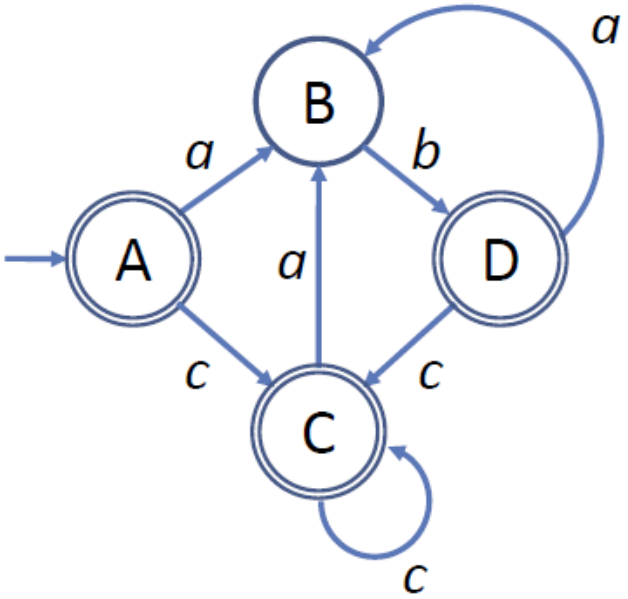


D_{states}

NFA States	DFA State	Next State		
		a	*b*	*c*
{9,7,1,5,10}	A ✓	B	-	C
{2}	B ✓	-	D	-
{6,8,10,7,1,5}	C ✓	B	-	C
{4,8,7,1,5,10}	D ✓	B	-	C

Illustrating Conversion – Example-3

NFA States	DFA State	Next State		
		a	*b*	*c*
{9,7,1,5,10}	A ✓	B	-	C
{2}	B ✓	-	D	-
{6,8,10,7,1,5}	C ✓	B	-	C
{4,8,7,1,5,10}	D ✓	B	-	C



Assignment

1. Find out the space and time complexity separately for DFA and NFA for checking whether a string is accepted by it or not.

Ans.

FA	Space	Time
NFA	$O(\|r\|)$	$O(\|r\| \times \|x\|)$
DFA	$O(2^{\|r\|})$	$O(\|x\|)$

Where $|r|$ is the length of the or the Regular expression constructed from the string x.

2. Find the time complexity of the algorithm to convert an NFA to DFA.

Assignment

- Follow the website
- http://hackingoff.com/compilers/nfa-to-dfa-conversion

Write a program [Using C graphics or Java Swing] to convert a RE to NFA

Converting Regular Expressions Directly to DFAs

Syntax Tree Construction:

- We may convert a regular expression into a DFA (without creating a NFA first).
- First we augment the given regular expression by concatenating it with a special symbol #.

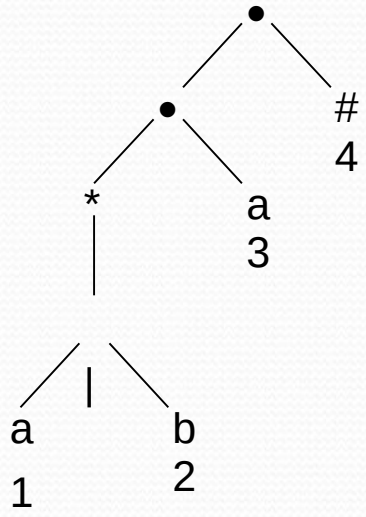
  r ➔ (r)# augmented regular expression
- Then, we create a syntax tree for this augmented regular expression.
- In this syntax tree, all alphabet symbols (plus # and the empty string) in the augmented regular expression will be on the leaves, and all inner nodes will be the operators in that augmented regular expression.
- Then each alphabet symbol (plus #) will be numbered (position numbers).

Constructing DFA

- Compute four functions: nullable, firstpos, lastpos and followpos by traversing T.
- Finally DFA is constructed from followpos.

Converting Regular Expressions Directly to DFA

$(a|b)^* a$ ➔ $(a|b)^* a \#$ augmented regular expression

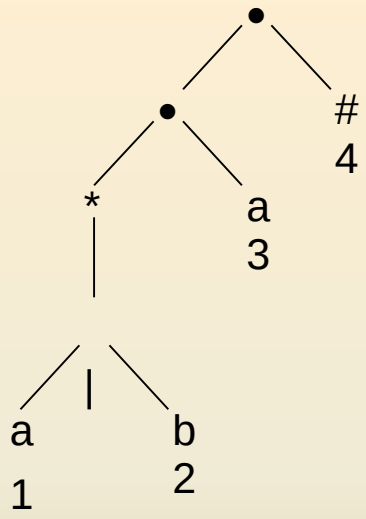
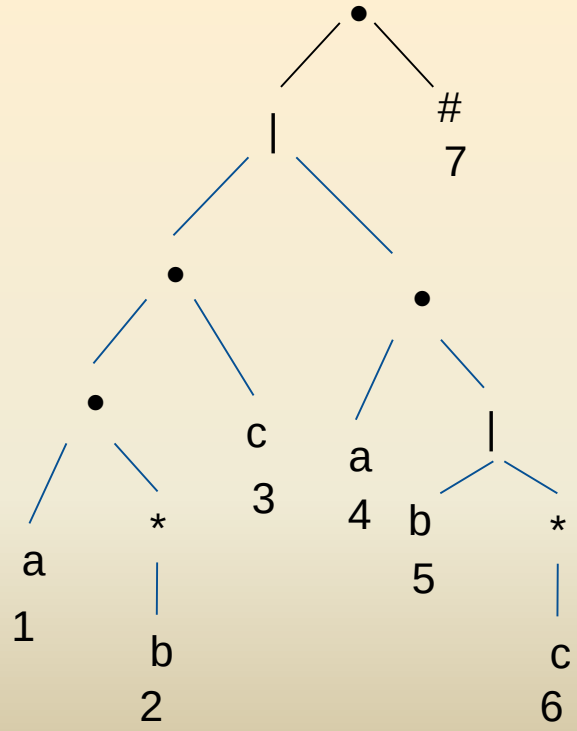


- each symbol is numbered (positions)
- each symbol is at a leave

- inner nodes are operators

Syntax tree of $(a|b)^* a \#$

Converting Regular Expressions Directly to DFA

(a|b)* a ➔ (a|b)* a #	augmented regular expression	**(ab*c) | (a(b|c*)) ➔ (ab*c) | (a(b|c*))#**
	• each symbol is numbered (positions) • each symbol is at a leave • inner nodes are operators	

Converting Regular Expressions Directly to DFA

followpos()

The notion of a position matching an input symbol is defined in terms of the function *followpos* on position of the syntax tree.

If i is a position then *followpos(i) is the set of position j such that there is some input string ……cd….. Such that i corresponds to this occurrence of c and j to this occurrence of d*

Then we define the function **followpos** for the positions (positions assigned to leaves).

followpos(i) -- is the set of positions which can follow the position i in the strings generated by the augmented regular expression.

For example, (a | b) * a #
1 2 3 4

followpos is just defined for leaves, it is not defined for inner nodes.

followpos(1) = {1,2,3}
followpos(2) = {1,2,3}
followpos(3) = {4}
followpos(4) = {}

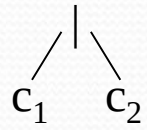
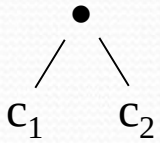
Converting Regular Expressions Directly to DFA

firstpos, lastpos, nullable

- To evaluate followpos, we need three more functions to be defined for the nodes (not just for leaves) of the syntax tree.
- **firstpos(n)** -- the set of the positions of the **first** symbols of strings generated by the sub-expression rooted by n.
- **lastpos(n)** -- the set of the positions of the **last** symbols of strings generated by the sub-expression rooted by n.
- **nullable(n)** -- *true* if the empty string is a member of strings generated by the sub-expression rooted by n

  false otherwise

How to evaluate firstpos, lastpos, nullable

n	nullable(n)	firstpos(n)	lastpos(n)
leaf labeled ε	true	Φ	Φ
leaf labeled with position i	false	{i}	{i}
	nullable(c_1) or nullable(c_2)	firstpos(c_1) ∪ firstpos(c_2)	lastpos(c_1) ∪ lastpos(c_2)
	nullable(c_1) and nullable(c_2)	if (nullable(c_1)) firstpos(c_1) ∪ firstpos(c_2) else firstpos(c_1)	if (nullable(c_2)) lastpos(c_1) ∪ lastpos(c_2) else lastpos(c_2)
	true	firstpos(c_1)	lastpos(c_1)

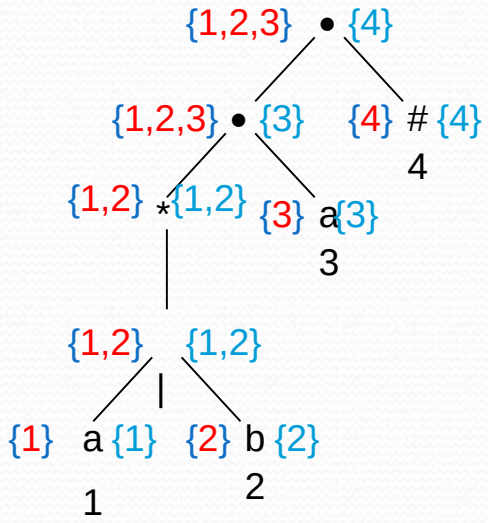
How to evaluate followpos

- Two-rules define the function followpos:

1. If **n** is concatenation-node with left child c_1 and right child c_2, and **i** is a position in **lastpos(c_1)**, then all positions in **firstpos(c_2)** are in **followpos(i)**.
2. If **n** is a star-node, and **i** is a position in **lastpos(n),** then all positions in **firstpos(n)** are in **followpos(i)**.

If firstpos and lastpos have been computed for each node, followpos of each position can be computed by making one depth-first traversal of the syntax tree.

Example -- (a | b) * a



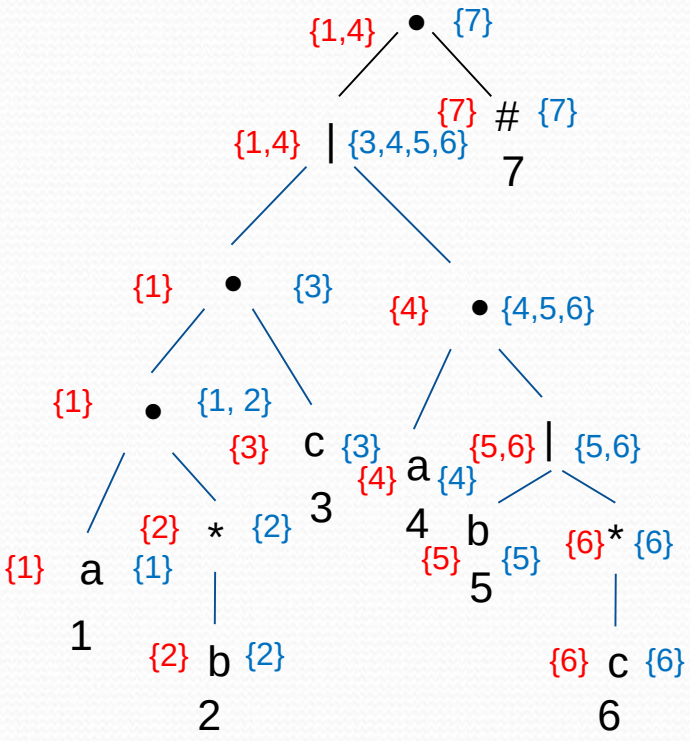
red – firstpos
blue – lastpos

Then we can calculate followpos

followpos(1) = {1,2,3}
followpos(2) = {1,2,3}
followpos(3) = {4}
followpos(4) = {}

- After we calculate follow positions, we are ready to create DFA for the regular expression.

Example (ab*c) | (a(b|c*)) → (ab*c) | (a(b|c*))#



followpos(1)= { 2,3 }

followpos(2)={ 2,3 }

followpos(3)={ 7}

followpos(4)={ 7,5,6 }

followpos(5)={ 7, }

followpos(6)={7,6 }

followpos(7)={ }

Assignment

- Write a C program to convert a RE directly to a DFA.

  (Help: Compiler Design by K. Muneeswaran)

- If firstpos and lastpos have been computed for each node, *followpos* of each position can be computed by making one depth-first traversal of the syntax tree.

Algo(RE→ DFA)

- //n→ a node in the syntax tree //i→ position of the node n

//N1, N2 → children sub expression of node n

//Returns true if ε is one of the strings generated by the sub expression whose root is at n.

```
nullable(n)
{
 switch(n)
 {
  case ε:
    return TRUE
  case a:
    return FALSE
 case N1|N2:
    return nullable(N1) OR nullable(N2)
case N1.N2:
   return nullable(N1) AND nullable(N2)
case N1*:
    return TRUE
```

Algorithm (RE → DFA)

- Create the syntax tree of (r) #
- Calculate the functions: followpos, firstpos, lastpos, nullable
- Put firstpos(root) into the states of DFA as an unmarked state.
- *while* (there is an unmarked state S in the states of DFA) *do*
  - mark **S**
  - *for each* input symbol **a** *do*
    - let $s_1,\ldots,s_n$ are positions in **S** and symbols in those positions are **a**
    - **S'** ← followpos(s_1) ∪ … ∪ followpos(s_n)
    - **move(S,a)** ← **S'**
    - if (**S'** is not empty and not in the states of DFA)
      - put **S'** into the states of DFA as an unmarked state.

- *the start state of DFA is firstpos(root)*
- *the accepting states of DFA are all states containing the position of #*

Example –1

$$(a \mid b)^* a \#$$

1 2 3 4

followpos(1)={1,2,3} followpos(2)={1,2,3} followpos(3)={4} followpos(4)={}

S_1=firstpos(root)={1,2,3}

⇓ mark S_1

a: followpos(1) ∪ followpos(3)={1,2,3,4}=S_2 move(S_1,a)=S_2

b: followpos(2)={1,2,3}=S_1 move(S_1,b)=S_1

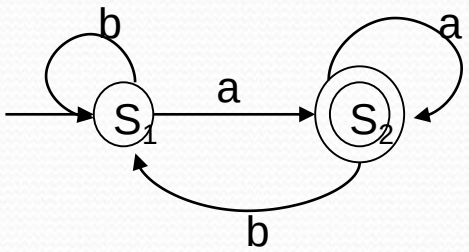
⇓ mark S_2

a: followpos(1) ∪ followpos(3)={1,2,3,4}=S_2 move(S_2,a)=S_2

b: followpos(2)={1,2,3}=S_1 move(S_2,b)=S_1

start state: S_1

accepting states: {S_2}



Example --2

$(\ a\ |\ \varepsilon)\ b\ c^*\ \#$
$\quad 1 \quad\quad 2\ \ 3\ \ 4$

Draw the parse tree and compute

followpos(1)={2} followpos(2)={3,4} followpos(3)={3,4} followpos(4)={}

S_1=firstpos(root)={1,2}

⇓ mark S_1

a: followpos(1)={2}=S_2 move(S_1,a)=S_2

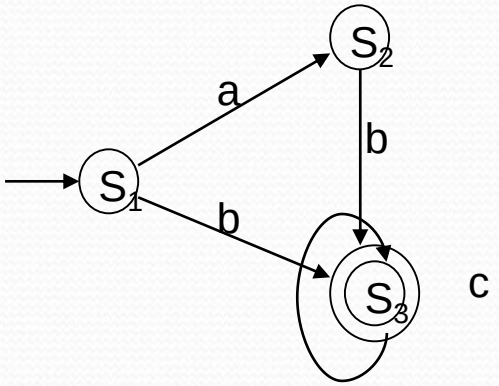
b: followpos(2)={3,4}=S_3 move(S_1,b)=S_3

⇓ mark S_2

b: followpos(2)={3,4}=S_3 move(S_2,b)=S_3

⇓ mark S_3

c: followpos(3)={3,4}=S_3 move(S_3,c)=S_3



start state: S_1

accepting states: {S_3}

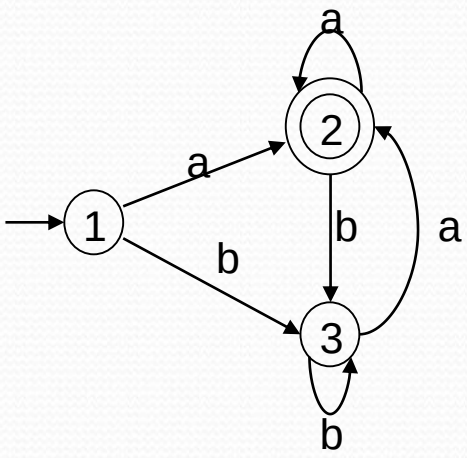
Minimizing Number of States of a DFA

- partition the set of states into two groups:
  - G_1 : set of accepting states
  - G_2 : set of non-accepting states

- For each new group G
  - partition G into subgroups such that states s_1 and s_2 are in the same group iff for all input symbols a, states s_1 and s_2 have transitions to states in the same group.

- Start state of the minimized DFA is the group containing the start state of the original DFA.
- Accepting states of the minimized DFA are the groups containing the accepting states of the original DFA.

Minimizing DFA - Example

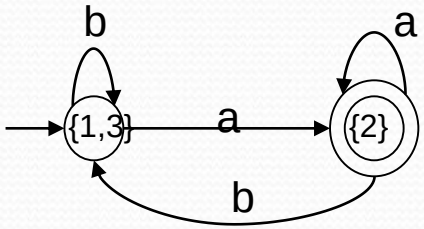


$G_1 = \{2\}$
$G_2 = \{1,3\}$

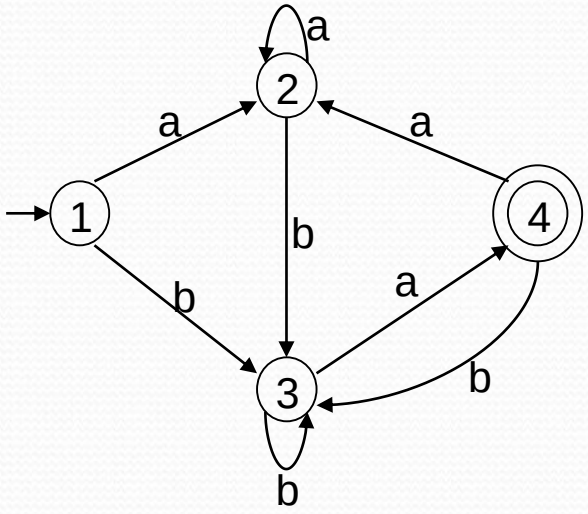
G_2 cannot be partitioned because

move(1,a)=2 move(1,b)=3
move(3,a)=2 move(2,b)=3

So, the minimized DFA (with minimum states)



Minimizing DFA – Another Example

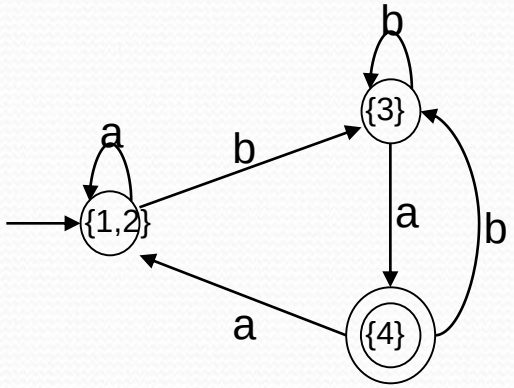


Groups: {1,2,3}{4}

no more partitioning

a	b
1->2	1->3
2->2	2->3
3->4	3->3

So, the minimized DFA



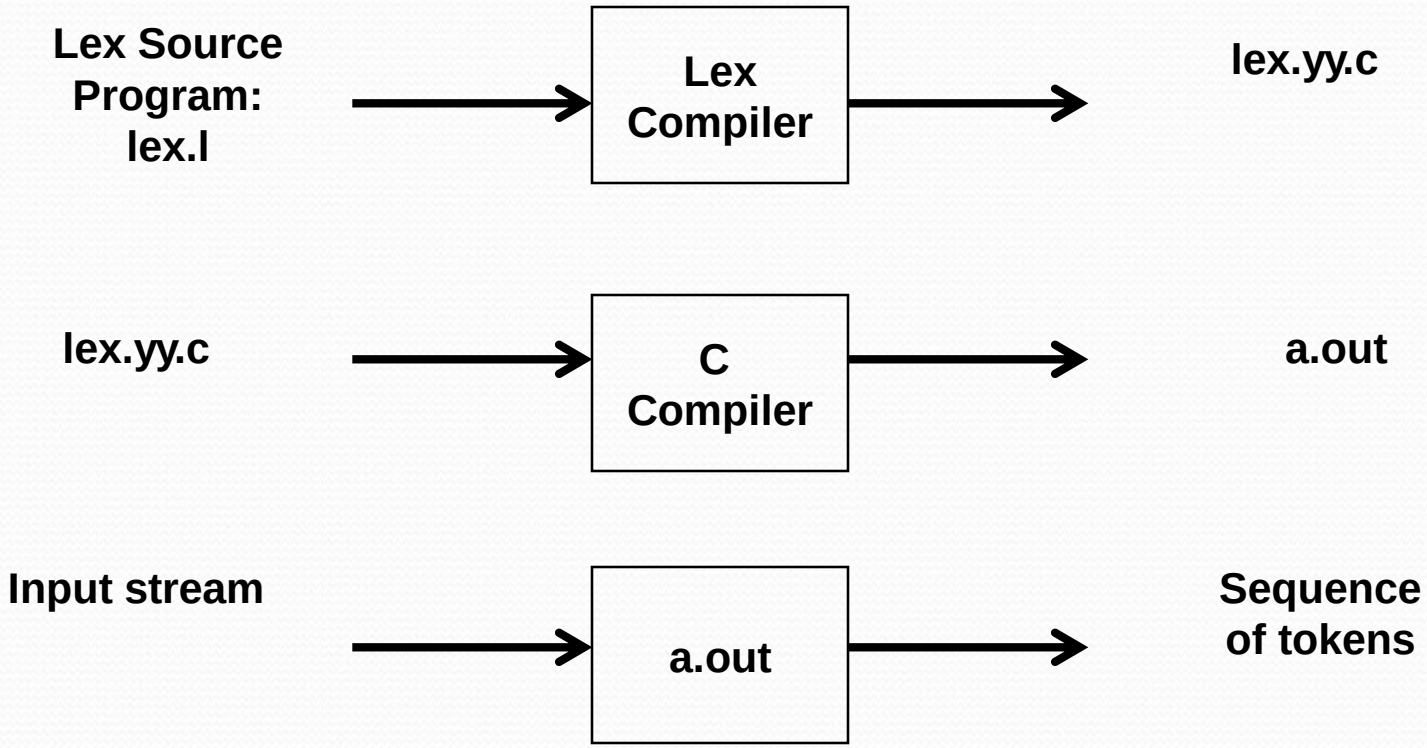
Assignment

What is the time complexity of the algorithm to minimize the number of states of a DFA?

Ans. O(n log n).

LEX

- LEX – A Lexical Analyzer Generator.
- A Compiler that Takes as Source a Specification for:
  - Tokens/Patterns of a Language
  - Generates a "C" Lexical Analyzer Program



Format of a Lexical Specification – 3 Parts

- Declarations:
  - Defs, Constants, Types, #includes, etc. that can Occur in a C Program
  - Regular Definitions (expressions)
- Translation Rules:
  - Pairs of (Regular Expression, Action)
  - Informs Lexical Analyzer of Action when Pattern is Recognized
- Auxiliary Procedures:
  - Designer Defined C Code
  - Can Replace System Calls

Lex.l File Format:
DECLARATIONS
%%
TRANSLATION RULES
%%
AUXILIARY PROCEDURES

Example lex.l File

```
%{
#define  T_IDENTIFIER  300
#define  T_INTEGER     301
#define  T_REAL        302
#define  T_STRING      303
#define  T_ASSIGN      304
#define  T_ELSE        305
#define  T_IF          306
#define  T_THEN        307
#define  T_EQ          308
#define  T_LT          309
#define  T_NE          310
#define  T_GE          311
#define  T_GT          312
%}

letter                  [a-zA-Z]
digit                   [0-9]
ws                      [ \t\n]+
id                      [A-Za-z][A-Za-z0-9]*
comment                 "(*"([^*]|\n|"*"+[^)])*"*"+")"
integer                 [0-9]+/([^0-9]|"..")
real                    [0-9]+"."[0-9]*([0-9]|"E"[+-]?[0-9]+)
string                  \'([^']|\'\')*\'
%%

":="                    {printf(" %s ", yytext);return(T_ASSIGN);}
"else"                  {printf(" %s ", yytext);return(T_ELSE);}
```

User Defined Values to Each Token (else lex will assign)

Regular Expression Rules for later token definitions

Token Definitions

Example lex.l File

```
"then"                  {
#ifdef PRNTFLG
printf(" %s ", yytext);
#endif
            return(T_THEN);
            }
"<="                    {printf(" %s ", yytext);return(T_EQ);}
"<"                     {printf(" %s ", yytext);return(T_LT);}
"<>"                    {printf(" %s ", yytext);return(T_NE);}
">="                    {printf(" %s ", yytext);return(T_GE);}
">"                     {printf(" %s ", yytext);return(T_GT);}

{id}                    {printf(" %s ", yytext);return(T_IDENTIFIER);}
{integer}   {printf(" %s ", yytext);return(T_INTEGER);}
{real}                  {printf(" %s ", yytext);return(T_REAL);}
{string}    {printf(" %s ", yytext);return(T_STRING);}
{comment}               {/* T_COMMENT */}
{ws}                    {/* spaces, tabs, newlines */}
%%
yywrap(){return 1;}

main()
{
int i;
do {
  i = yylex();
} while (i!=0);
}
```

Conditional compilation action

Token Definitions

Discard

EOF for input

Three Variables:

yytext = “currenttoken”

yylen = 12

yylval = 300

- Add 3 to every positive number divisible by 7.

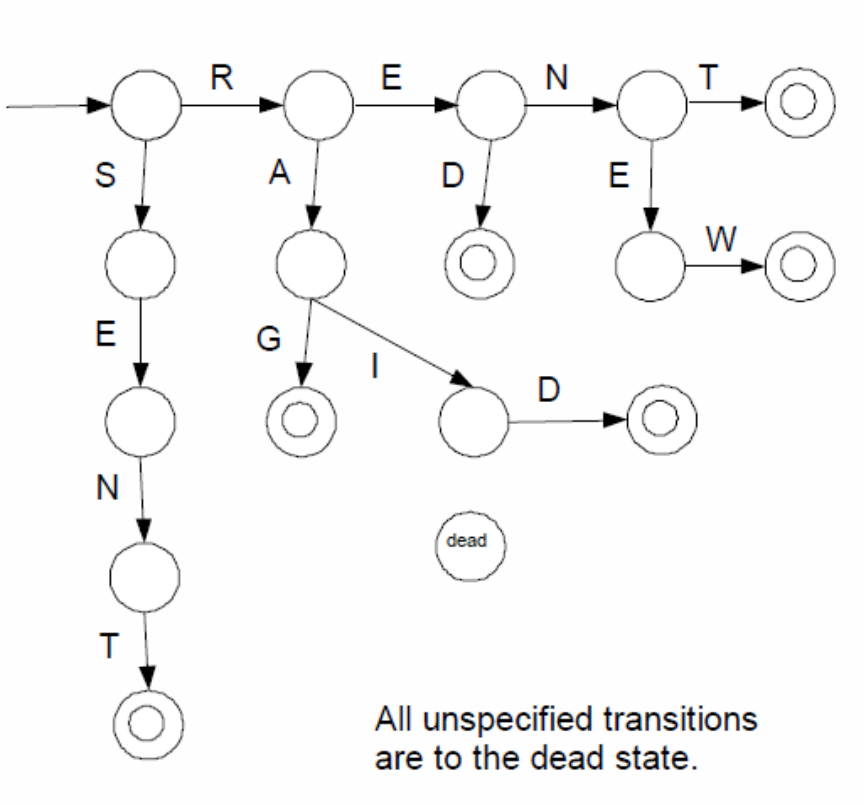
- %%

```
int k;
[0-9]+   {
            k = atoi(yytext);
            if (k%7 == 0)
               printf("%d", k+3);
              else
                 printf("%d",k);
          }
```

The rule [0-9]+ recognizes strings of digits; atoi converts the digits to binary and stores the result in k.

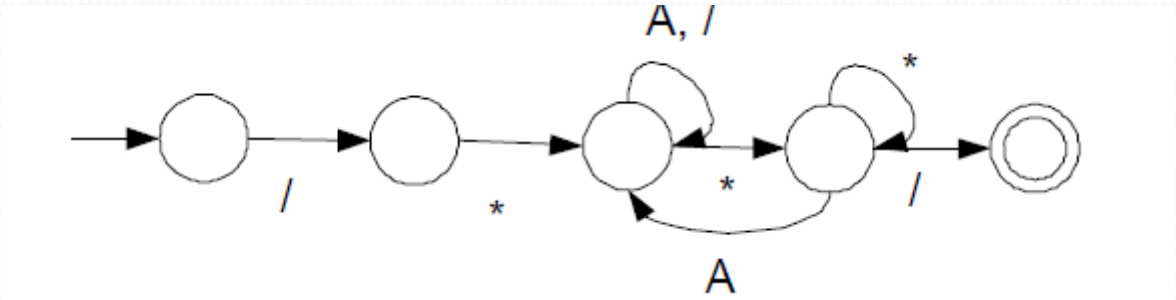
Exercise

- Show a *finite state machine which will recognize the words RENT, RENEW, RED,* RAID, RAG, and SENT. Use a different accepting state for each of these words.



Exercise

- Show a *finite state machine that will accept C-style comments /* as shown* here */. Use the symbol A to represent any character other than * or /; thus the input alphabet will be {/,*,A}.



Exercise

- a number-string is a non-empty sequence of decimal digits, i.e., something in the language defined by the regular expression [0-9]+. The value of a number-string is the usual interpretation of a number-string as an integer number. Note that leading zeroes are allowed.

Make for each of the following languages a regular expression that describes that language.

- a) All number-strings that have the value 42.
- b) All number-strings that do not have the value 42.
- c) All number-strings that have a value that is strictly greater than 42.

Solve:

a) 0*42

b) The number must either be a one-digit number, a two-digit number different from 42 or have at least three significant digits:

0([0−9] | [1−3][0−9] | 4[0−1] | 4[3−9] | [5−9][0−9] | [1−9][0−9][0−9]+)

c) The number must either be a two-digit number greater than 42 or have at least three significant digits:

0(4[3−9] | [5−9][0−9] | [1−9][0−9][0−9]+)

Exercise

- Given the regular expression

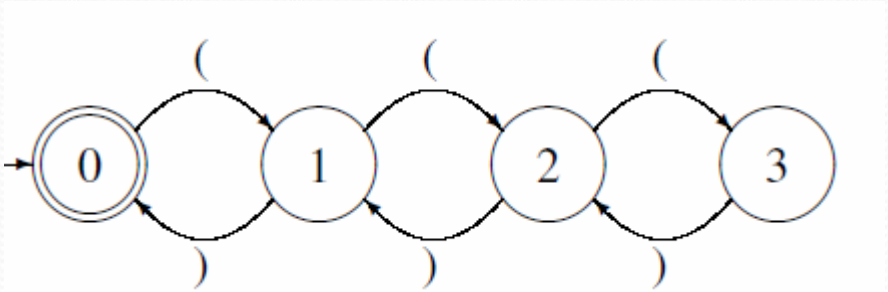
a)((a|b)(a|bb))*

b) a*(a|b)aa

Construct equivalent DFA.

Exercise

- Construct a DFA that recognizes balanced sequences of parenthesis with a maximal nesting depth of 3, e.g., e, ()(), (()(())) or (()())()() but not (((()))) or (()(()(()))).



References

- Alfred V. *Aho*, Ravi. *Sethi*, Jeffrey D. *Ullman*,**Compilers: Principles, Techniques, and Tools**
- **ANDREW W. APPEL, JENS PALSBERG, Modern Compiler Implementation in Java, Second Edition**
- Torben Ægidius Mogensen, Basics of Compiler Design