

Lexical Analysis

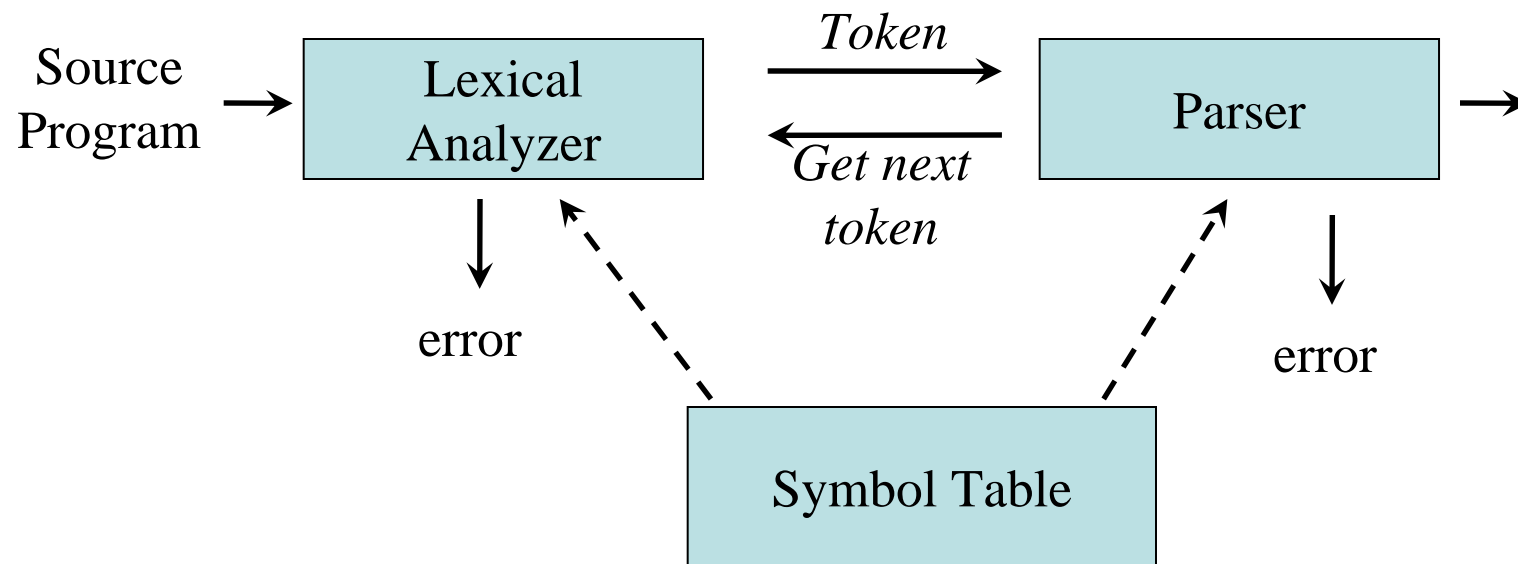
Reading: Chapter 3

Deals with techniques for specifying and
implementing lexical analyzer

Lexical Analyzer

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 may also perform other auxiliary operation like removing redundant white space, removing token separator (like semicolon) etc.

Tokens, Patterns, Lexemes

A *token* is a logical building block of language

For example: **id** and **num**

Lexemes are the specific character strings that make up a token

For example: **abc** and **123**

A token can represent more than one lexeme

Patterns are rules describing the set of lexemes belonging to a token

For example: “*letter followed by letters and digits*” and “*non-empty sequence of digits*”

Regular expressions are widely used to specify patterns

Attributes of Tokens

When a token represents more than one lexeme, lexical analyzer must provide additional information about the particular lexeme. This additional information is called as the *attribute* of the 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.

Some attributes:

- $\langle \text{id}, \text{attr} \rangle$ where attr is pointer to the symbol table
- $\langle \text{assgop}, _ \rangle$ no attribute is needed (if there is only one assignment operator)
- $\langle \text{num}, \text{val} \rangle$ where val is the actual value of the number.

Example: $\text{dest} = \text{source} + 5$

Tokens: $\langle \text{id}, \text{pt for dest} \rangle$, $\langle \text{assignop} \rangle$, $\langle \text{num}, 5 \rangle$

Token type and its attribute uniquely identifies a lexeme.

Lexical Analyzer implementation strategies

1. Use a lexical-analyzer generator like flex to produce lexical analyzer from a regular express based specification. The generator provides routines for reading and buffering the input.
2. Design a lexical analyzer in high level programming language like C, using the I/O facilities of the language to read and buffering the input.
3. Write the lexical analyzer in assembly language and explicitly manage the input and buffering

The above are in increasing order of complexity and efficiency

Lookahead and Buffering

Many times, a scanner has to look ahead several characters from the current character in order to recognize the token.

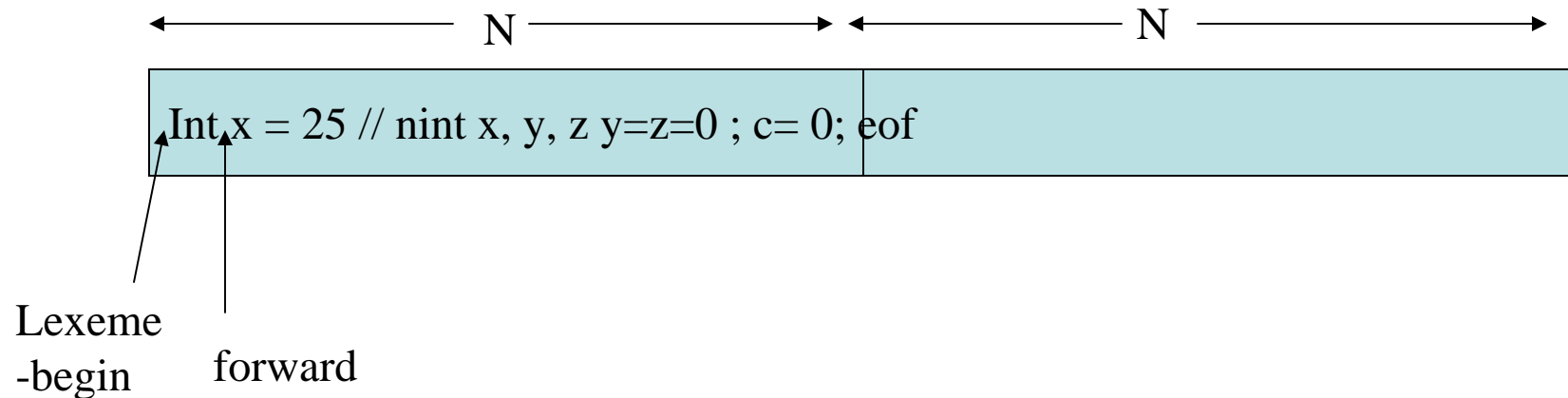
For example *int* is keyword in C, while the term *inp* may be a variable name. When the character '*i*' is encountered, the scanner can not decide whether it is a keyword or a variable name until it reads two more characters.

In order to efficiently move back and forth in the input stream, input buffering is used

Buffering

2N Buffering: Buffer divided into two N-character halves, where N is number of character on the disk block e.g. 1024 or 4096

Each read command reads a half buffer, if input fewer than N, then special character eof is read which indicate the end of source file.



Specification of Tokens

Stings & Languages

An *alphabet* Σ is a finite set of symbols (characters)

A *string* s is a finite sequence of symbols from Σ

- $|s|$ denotes the length of string s
- ε denotes the empty string, thus $|\varepsilon| = 0$

Operators on Strings:

- Concatenation: xy represents the concatenation of strings x and y . $s\varepsilon = s$
 $\varepsilon s = s$
- $s^n = s s s \dots s$ (n times) $s^0 = \varepsilon$

A *language* is a specific set of strings over some fixed alphabet Σ

- \emptyset the empty set is a language.
- $\{\varepsilon\}$ the set containing empty string is a language
- The set of well-formed C programs is a language
- The set of all possible identifiers is a language.

Specification of Tokens

Operation on Languages

Concatenation: $L_1 L_2 = \{ s_1 s_2 \mid s_1 \in L_1 \text{ and } s_2 \in L_2 \}$

Union: $L_1 \cup L_2 = \{ s \mid s \in L_1 \text{ or } s \in L_2 \}$

Exponentiation: $L^0 = \{\epsilon\}$ $L^1 = L$ $L^2 = LL$

Kleene Closure: $L^* = \cup_{i=0, \dots, \infty} L^i$

Positive Closure: $L^+ = \cup_{i=1, \dots, \infty} L^i$

Specification of Tokens

Operation on Languages: Example

$$L_1 = \{a,b,c,d\} \quad L_2 = \{1,2\}$$

$$L_1 L_2 = \{a1,a2,b1,b2,c1,c2,d1,d2\}$$

$$L_1 \cup L_2 = \{a,b,c,d,1,2\}$$

$$L_1^3 = \text{all strings with length three (using } a,b,c,d \text{)}$$

$$L_1^* = \text{all strings using letters } a,b,c,d \text{ and empty string}$$

$$L_1^+ = \text{doesn't include the empty string}$$

Specification of Tokens

Regular Expressions

We use regular expressions to describe tokens of a programming language.

Basis symbols:

- ε is a regular expression denoting language $\{\varepsilon\}$
- $a \in \Sigma$ is a regular expression denoting $\{a\}$

If r and s are regular expressions denoting languages $L_1(r)$ and $L_2(s)$ respectively, then

- $r+s$ is a regular expression denoting $L_1(r) \cup L_2(s)$
- rs is a regular expression denoting $L_1(r) L_2(s)$
- r^* is a regular expression denoting $(L_1(r))^*$
- (r) is a regular expression denoting $L_1(r)$

A language defined by a regular expression is called a *regular set*

Specification of Tokens

Properties of Regular Expressions

For regular expression r , s & t

$$r + s = s + r \quad (\text{union is commutative})$$

$$r + (s + t) = (r + s) + t \quad (\text{union is associative})$$

$$(rs)t = r(st) \quad (\text{concatenation is associative})$$

$$r(s + t) = rs + rt \quad (\text{concatenation distributes over union})$$

$$\epsilon r = r, r\epsilon \quad (\epsilon \text{ is the identity element for concatenation})$$

$$r^{**} = r^* \quad (\text{closure is idempotent})$$

Specification of Tokens

Regular Expressions: Examples

Given the alphabet $A = \{0,1\}$

1. $1(1+0)^*0$ denotes the language of all string that begins with a 1 and ends with a 0
2. $(1+0)^*00$ denotes the language of all strings that ends with 00 (binary number multiple of 4)
3. $(01)^*+(10)^*$ denotes the set of all stings that describe alternating 1s and 0s

Regular Expressions: Exercise Review

Given the alphabet $A = \{0,1\}$ write the regular expression for the following

1. String that either have substring 001 or 100
2. Strings where no two 1s occurs consecutively
3. String which have an odd numbers of 0s
4. String which have an odd numbers of 0s and an even numbers 1s
5. String that have at most 2 0s
6. String that at least 3 1s
7. Strings that have at most two 0s and at least three 1s

Specification of Tokens

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:

$$\begin{array}{ll} d_1 \rightarrow r_1 & \text{where } d_i \text{ is a distinct name and} \\ d_2 \rightarrow r_2 & r_i \text{ is a regular expression over symbols in} \\ \cdot & \Sigma \cup \{d_1, d_2, \dots, d_{i-1}\} \\ d_n \rightarrow r_n & \end{array}$$

basic symbols previously defined names

Specification of Tokens

Regular Definitions: Examples

Identifiers in Pascal are defined as a string of letters and digits beginning with a letter

$$\text{letter} \rightarrow A \mid B \mid \dots \mid Z \mid a \mid b \mid \dots \mid z$$
$$\text{digit} \rightarrow 0 \mid 1 \mid \dots \mid 9$$
$$\text{id} \rightarrow \text{letter} (\text{letter} \mid \text{digit})^*$$

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

$$(A \mid \dots \mid Z \mid a \mid \dots \mid z) ((A \mid \dots \mid Z \mid a \mid \dots \mid z) \mid (0 \mid \dots \mid 9))^*$$

Regular definitions are not recursive:

digits \rightarrow **digit digits** | **digit** *wrong!*

Specification of Tokens

Notational Shorthand

The following shorthands are often used:

$$\begin{aligned}r^+ &= rr^* \\ r? &= r \mid \varepsilon \\ [\mathbf{a-z}] &= \mathbf{a} \mid \mathbf{b} \mid \mathbf{c} \mid \dots \mid \mathbf{z}\end{aligned}$$

Examples:

digit \rightarrow **[0-9]**

num \rightarrow **digit⁺ (. digit⁺)? (E (+ | -)? digit⁺)?**

Regular Definitions: Exercise Review

1. Write regular definition for specifying floating point number in a programming language like C
2. Write regular definitions for specifying an integer array declaration in language like C

Recognition of Tokens

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.

Recognition of token implies implementing a regular expression recognizer. That entails the implementation of Finite Automation

A finite automaton can be: deterministic (DFA) or non-deterministic (NFA)

This means that we may use a deterministic or non-deterministic automaton as a lexical analyzer.

Both deterministic and non-deterministic finite automaton recognize regular sets.

Which one?

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

Recognition of Tokens

Design of a Lexical Analyzer

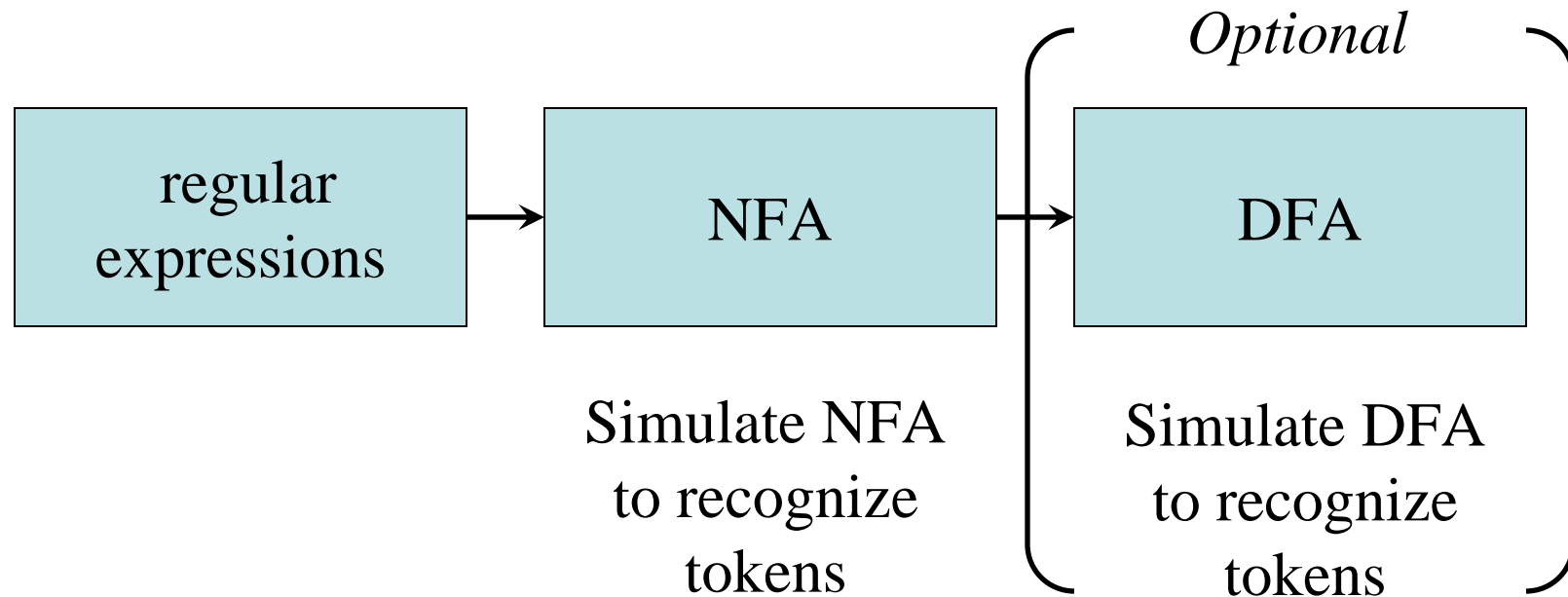
First, we define regular expressions for tokens; Then we convert them into a DFA to get a lexical analyzer for our tokens.

Algorithm1:

Regular Expression \rightarrow NFA \rightarrow DFA (two steps: first to NFA, then to DFA)

Algorithm2:

Regular Expression \rightarrow DFA (directly convert a regular expression into a DFA)



Non-Deterministic Finite Automaton (NFA)

An NFA is a 5-tuple $(S, \Sigma, \delta, s_0, F)$ where

S is a finite set of *states*

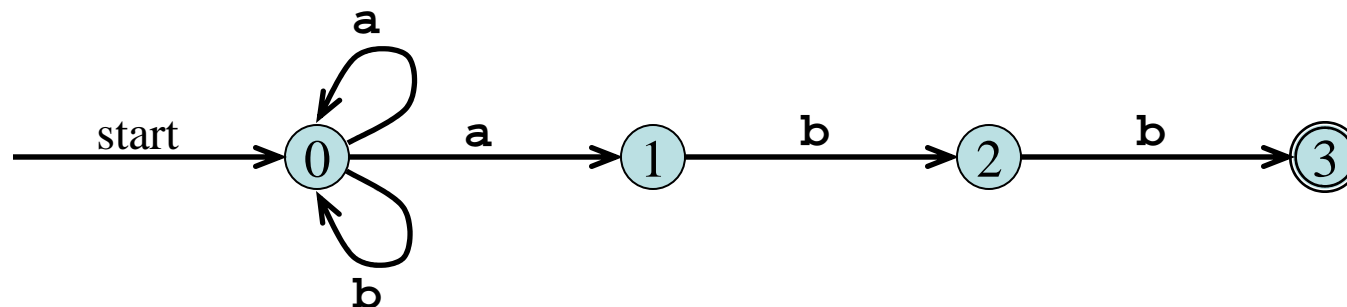
Σ is a finite set of symbols, the *alphabet*

δ is a *mapping* from $S \times \Sigma$ to a set of states

$s_0 \in S$ is the *start state*

$F \subseteq S$ is the set of *accepting (or final) states*

A NFA accepts a string x , if and only if there is a path from the starting state to one of accepting states such that edge labels along this path spell out x .



$S = \{0,1,2,3\}$

$\Sigma = \{\mathbf{a},\mathbf{b}\}$

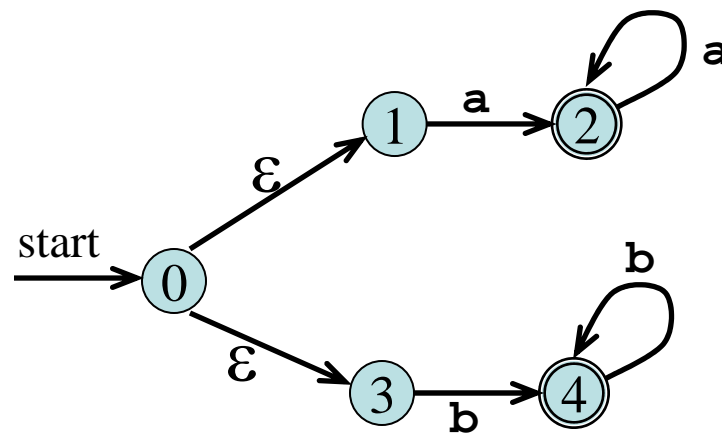
$s_0 = 0$

$F = \{3\}$

The above state machine is an NFA for regular expression $(a+b)^*abb$ having non deterministic transition at the state 0. On reading a it may either stay at state 0 or go to state 1.

ϵ - NFA

ϵ - transitions are allowed in NFAs. In other words, we can move from one state to another one without consuming any symbol.



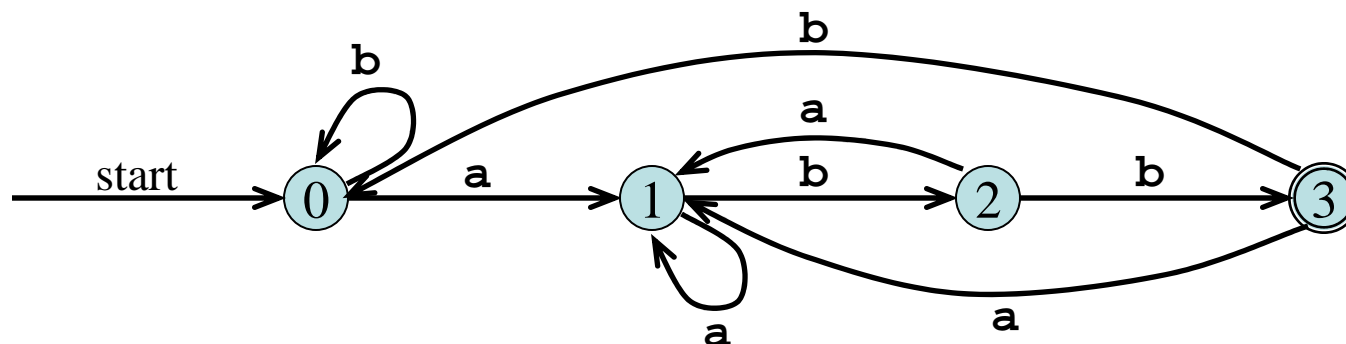
Regular expressions are easily convertible ϵ -NFA

The figure shows a state machine with ϵ moves that is equivalent to the regular expression $aa^* + bb^*$. The upper half of the state machine recognize aa^* and lower half recognize bb^* . The machine non-deterministically move to the upper half or lower half when started from state 0

Deterministic Finite Automaton (DFA)

A *deterministic finite automaton* is a special case of an NFA

- No state has an ϵ -transition
- For each state s and input symbol a there is at most one edge labeled a leaving s . i.e. *transition function is from pair of state-symbol to state (not set of states)*



The above figure shows DFA for the same regular expression **$(a+b)^*abb$**

Implementing a DFA

Input: An input string x , assume that the end of a string is marked with a special symbol (say eos).

Output: returns “yes” if the input string is accepted, else return “no”

| | |
|----------------------------------|--|
| $s \leftarrow s_0$ | { start from the initial state } |
| $c \leftarrow \text{nextchar}$ | { get the next character from the input string } |
| while ($c \neq \text{eos}$) do | { do until the end of the string } |
| begin | |
| $s \leftarrow \text{move}(s, c)$ | { transition function } |
| $c \leftarrow \text{nextchar}$ | |
| end | |
| if ($s \in F$) then | { if s is an accepting state } |
| return “yes” | |
| else | |
| return “no” | |

(an efficient implementation)

Implementing a NFA

```
S ←  $\epsilon$ -closure( $\{s_0\}$ )    { set all of states can be accessible from  $s_0$  by  $\epsilon$ -transitions }
c ← nextchar
while (c != eos) {
  begin
    s ←  $\epsilon$ -closure(move(S,c)) { set of all states can be accessible from a state in S
    c ← nextchar                by a transition on c }
  end
  if ( $S \cap F \neq \Phi$ ) then    { if S contains an accepting state }
    return "yes"
  else
    return "no"
```

Exercise: This algorithm is not efficient in compared to DFA. Prove the assertion!

Conversion: Regular Expression to NFA

Thomson's Construction

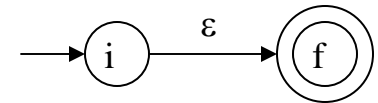
This is one way to convert a regular expression into a NFA.

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

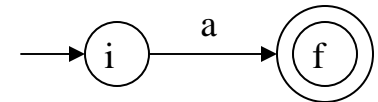
RE to NFA

Thomson's Construction

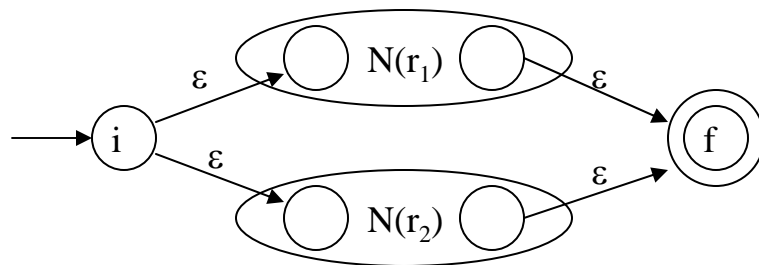
1. To recognize an empty string ϵ



2. To recognize a symbol a in the alphabet Σ



3. If $N(r_1)$ and $N(r_2)$ are NFAs for regular expressions r_1 and r_2
 a. For regular expression $r_1 + r_2$

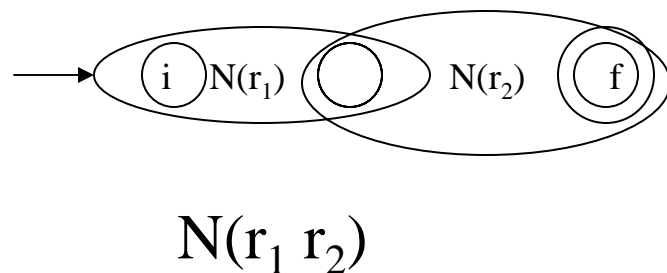


$N(r_1 + r_2)$

RE to NFA

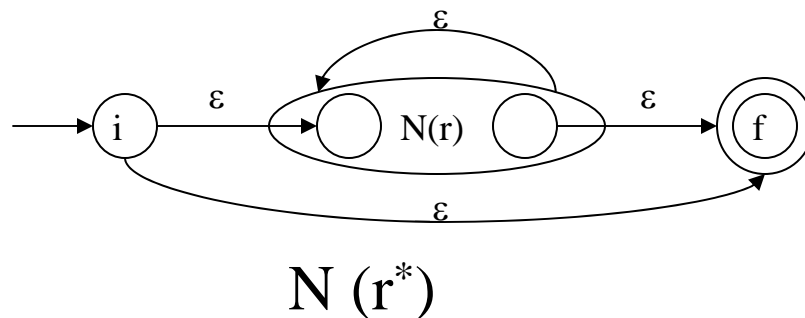
Thomson's Construction

b. For regular expression $r_1 r_2$



The start state of $N(r_1)$ becomes the start state of $N(r_1 r_2)$ and final state of $N(r_2)$ become final state of $N(r_1 r_2)$

c. For regular expression r^*

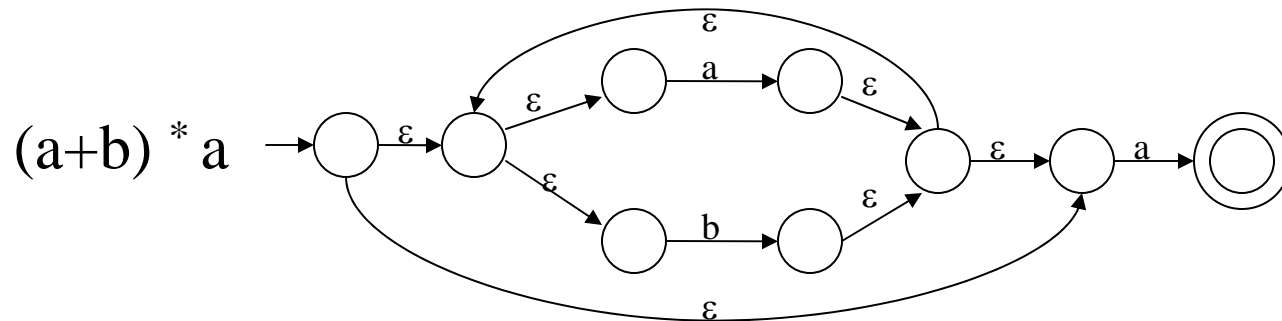
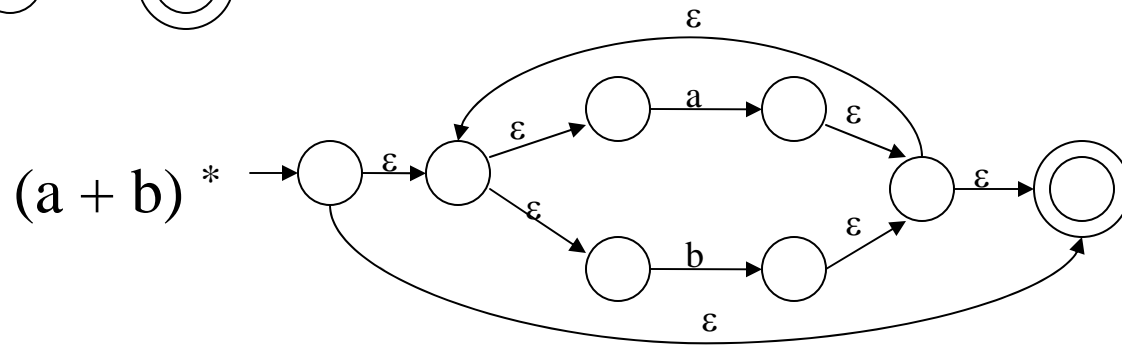
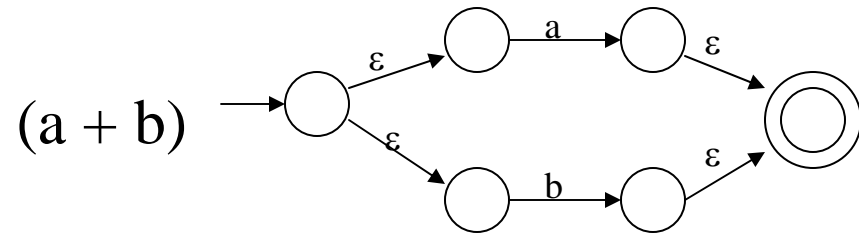
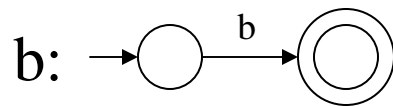
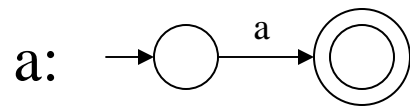


Using rule 1 and 2 we construct NFA's for each basic symbol in the expression, we combine these basic NFA using rule 3 to obtain the NFA for entire expression

RE to NFA

Thomson's Construction

Example:- NFA construction of RE $(a+b)^* a$



Conversion from NFA to DFA

Subset Construction

The *subset construction algorithm* converts an NFA into a DFA

We use the following operations to keep the track of sets of NFA

$\varepsilon\text{-closure}(s) \Rightarrow$ the set of NFA states reachable from state s on ε -transition i.e. $\{s\} \cup \{t \mid s \rightarrow_{\varepsilon} \dots \rightarrow_{\varepsilon} t\}$

$\varepsilon\text{-closure}(T) \Rightarrow$ the set of NFA states reachable from state s in T on ε -transition i.e. $\bigcup_{s \in T} \varepsilon\text{-closure}(s)$

$\text{move}(T, a) \Rightarrow$ the set of NFA states to which there is transition on input a from state s in T i.e. $\{t \mid s \rightarrow_a t \text{ and } s \in T\}$

The algorithm produces:

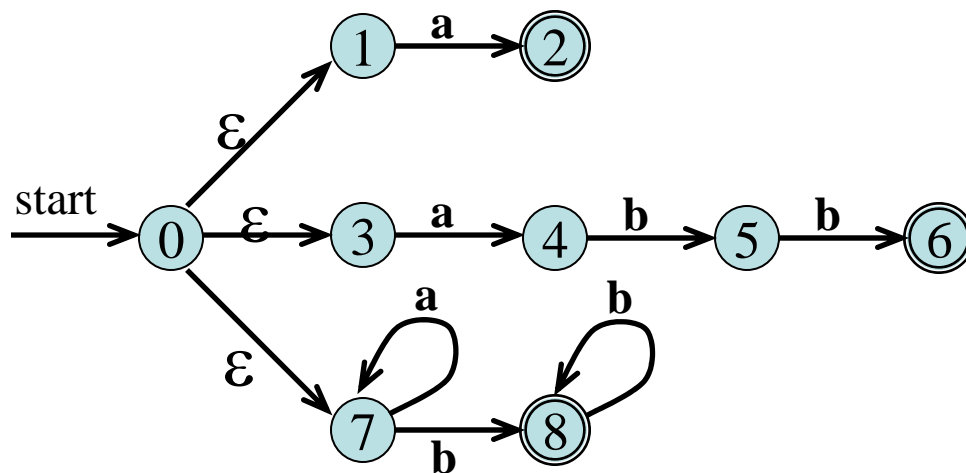
$Dstates$ is the set of states of the new DFA consisting of sets of states of the NFA

$Dtran$ is the transition table of the new DFA

Conversion from NFA to DFA

Subset Construction

ϵ -closure and *move* Examples



$$\epsilon\text{-closure}(\{0\}) = \{0,1,3,7\}$$

$$\text{move}(\{0,1,3,7\}, \mathbf{a}) = \{2,4,7\}$$

$$\epsilon\text{-closure}(\{2,4,7\}) = \{2,4,7\}$$

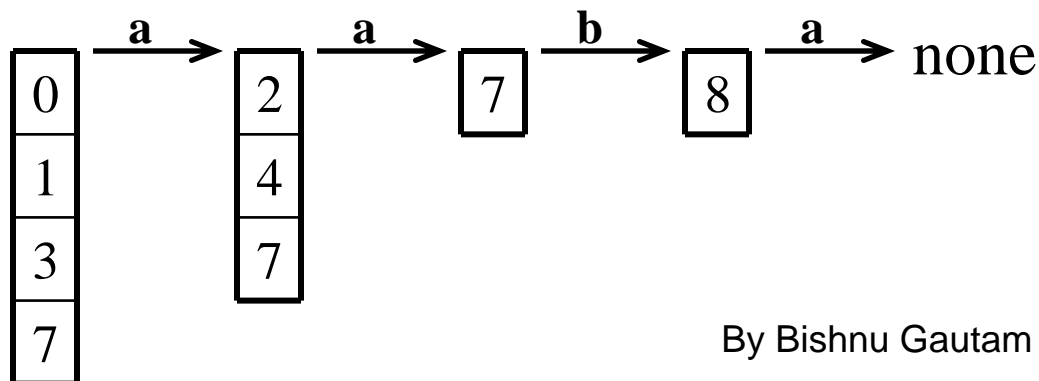
$$\text{move}(\{2,4,7\}, \mathbf{a}) = \{7\}$$

$$\epsilon\text{-closure}(\{7\}) = \{7\}$$

$$\text{move}(\{7\}, \mathbf{b}) = \{8\}$$

$$\epsilon\text{-closure}(\{8\}) = \{8\}$$

$$\text{move}(\{8\}, \mathbf{a}) = \emptyset$$



Conversion from NFA to DFA

Subset Construction Algorithm

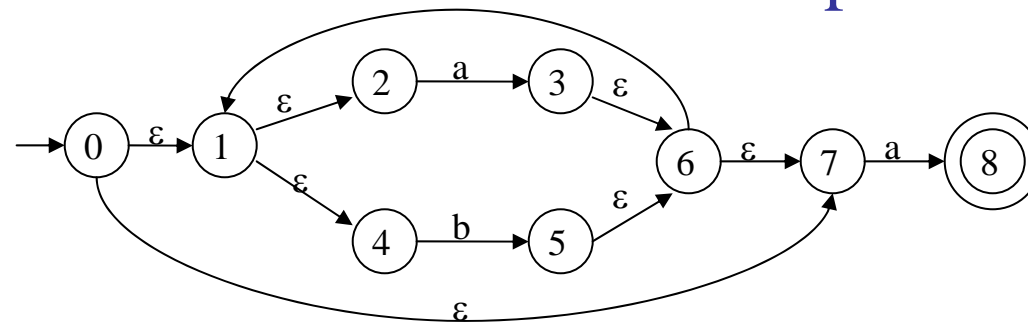
```
put  $\epsilon$ -closure( $s_0$ ) as an unmarked state in to  $Dstates$ 
while there is an unmarked state  $T$  in  $Dstates$  do
    mark  $T$ 
    for each input symbol  $a \in \Sigma$  do
         $U = \epsilon$ -closure(move( $T, a$ ))
        if  $U$  is not in  $Dstates$  then
            add  $U$  as an unmarked state to  $Dstates$ 
        end if
         $Dtran[T, a] = U$ 
    end do
end do
```

A state of $Dstates$ is an accepting state of DFA if it is a set of NFA states containing at least one accepting state of NFA

The start state of DFA is ϵ -closure(s_0)

Conversion from NFA to DFA

Subset Construction Example



$$S_0 = \varepsilon\text{-closure}(\{0\}) = \{0,1,2,4,7\}$$

S_0 into $Dstates$ as an unmarked state

\Downarrow mark S_0

$$\varepsilon\text{-closure}(\text{move}(S_0, a)) = \varepsilon\text{-closure}(\{3,8\}) = \{1,2,3,4,6,7,8\} = S_1 \quad S_1 \text{ into } Dstates$$

$$\varepsilon\text{-closure}(\text{move}(S_0, b)) = \varepsilon\text{-closure}(\{5\}) = \{1,2,4,5,6,7\} = S_2 \quad S_2 \text{ into } Dstates$$

$$Dtran[S_0, a] \leftarrow S_1 \quad Dtran[S_0, b] \leftarrow S_2$$

\Downarrow mark S_1

$$\varepsilon\text{-closure}(\text{move}(S_1, a)) = \varepsilon\text{-closure}(\{3,8\}) = \{1,2,3,4,6,7,8\} = S_1$$

$$\varepsilon\text{-closure}(\text{move}(S_1, b)) = \varepsilon\text{-closure}(\{5\}) = \{1,2,4,5,6,7\} = S_2$$

$$Dtran[S_1, a] \leftarrow S_1 \quad Dtran[S_1, b] \leftarrow S_2$$

\Downarrow mark S_2

By Bishnu Gautam

Conversion from NFA to DFA

Subset Construction Example

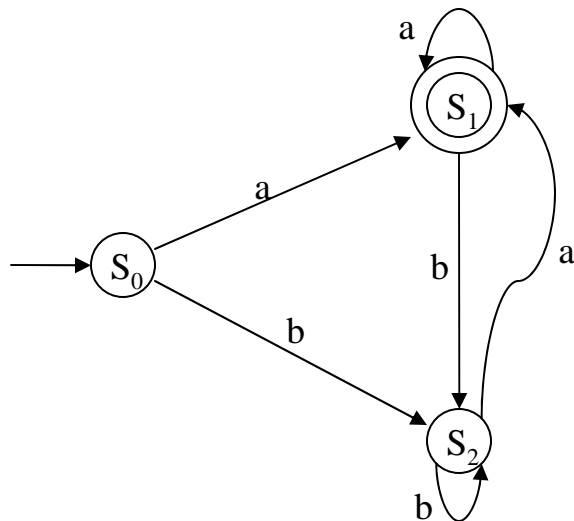
$$\varepsilon\text{-closure}(\text{move}(S_2, a)) = \varepsilon\text{-closure}(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\} = S_1$$

$$\varepsilon\text{-closure}(\text{move}(S_2, b)) = \varepsilon\text{-closure}(\{5\}) = \{1, 2, 4, 5, 6, 7\} = S_2$$

$$Dtran[S_2, a] \leftarrow S_1 \quad Dtran[S_2, b] \leftarrow S_2$$

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 $S_1 = \{1, 2, 3, 4, 6, 7, 8\}$



Exercise Review

1. Convert the following regular expression first into NFA and then into DFA
 1. $0+(1+0)^*00$
 2. zero \rightarrow 0; one \rightarrow 1; bit \rightarrow zero + one; bits \rightarrow bit*
2. Write an algorithm for computing ε -closure(s) of any state s in NFA
3. Calculate and compare the time and space complexity to recognize the same string by NFA and by DFA

Conversion from RE to DFA Directly

Some Definitions

Important States: The “*important states*” of an NFA are those without an ε -transition, that is if $move(\{s\}, a) \neq \emptyset$ for some a then s is an important state

In an optimal state machine all states are important states

The subset construction algorithm uses only the important states when it determines ε -closure($move(T, a)$)

Augmented Regular Expression: Since the accepting state does not have any transition i. e. it is not important state. We introduced an “augmented” character # from the accepting state to make accepting state important state. The regular expression $(r)\#$ is called the augmented regular expression of the original expression r .

Conversion from RE to DFA Directly

Step by step process

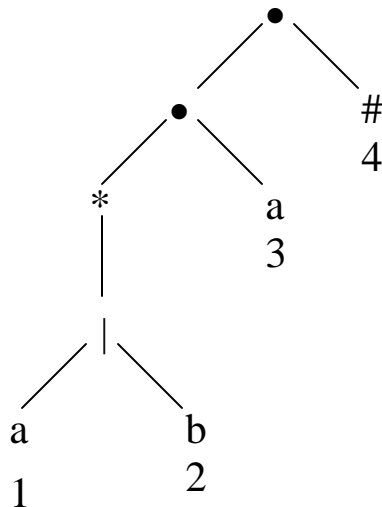
1. Augment the given regular expression by concatenating it with special symbol # i.e $r \rightarrow (r)\#$
2. Create the syntax tree for this augmented regular expression

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.
3. Then each alphabet symbol (plus #) will be numbered (position numbers)
4. Traverse the tree to construct functions *nullable*, *firstpos*, *lastpos*, and *followpos*
5. Finally construct the DFA from the *followpos*

Conversion from RE to DFA

Syntax Tree Construction

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



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

- each symbol is numbered (positions)
- each symbol is at a leaf
- inner nodes are operators

Conversion from RE to DFA

followpos

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, $(a \mid b)^* a \#$
 1 2 3 4

$\text{followpos}(1) = \{1, 2, 3\}$

$\text{followpos}(2) = \{1, 2, 3\}$

$\text{followpos}(3) = \{4\}$

$\text{followpos}(4) = \{\}$

Conversion from RE to DFA

firstpos, lastpos, nullable

To evaluate *followpos*, we need three functions to defined the nodes (not just for leaves) of the syntax tree.

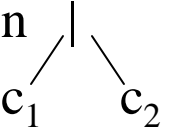
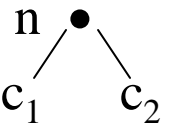
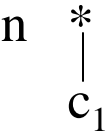
firstpos(n) -- the set of the positions of the **first** symbols of strings generated by the sub-expression rooted by n.

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

Conversion from RE to DFA

Rules for calculating nullable, firstpos & lastpos

| node <u>n</u> | <u>nullable(n)</u> | <u>firstpos(n)</u> | <u>lastpos(n)</u> |
|---|---|--|---|
| is leaf labeled ϵ | true | Φ | Φ |
| is leaf labeled with position i | false | {i} (position of leaf node) | {i} |
|  | nullable(c_1) or nullable(c_2) | $\text{firstpos}(c_1) \cup \text{firstpos}(c_2)$ | $\text{lastpos}(c_1) \cup \text{lastpos}(c_2)$ |
|  | nullable(c_1) and nullable(c_2) | if (nullable(c_1)) then $\text{firstpos}(c_1) \cup \text{firstpos}(c_2)$ else firstpos(c_1) | if (nullable(c_2)) then $\text{lastpos}(c_1) \cup \text{lastpos}(c_2)$ else lastpos(c_2) |
|  | true | firstpos(c_1) | lastpos(c_1) |

Conversion from RE to DFA

How to evaluate followpos

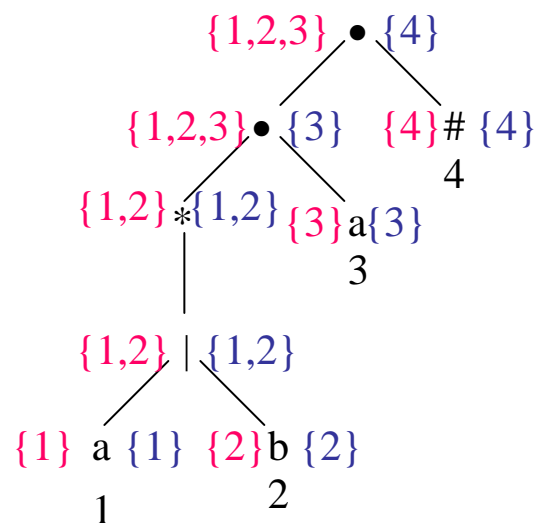
```
for each node  $n$  in the tree do  
    if  $n$  is a cat-node with left child  $c_1$  and right child  $c_2$  then  
        for each  $i$  in  $lastpos(c_1)$  do  
             $followpos(i) := followpos(i) \cup firstpos(c_2)$   
        end do  
    else if  $n$  is a star-node  
        for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

Compute the followpos bottom-up for each node of syntax tree

Conversion from RE to DFA

How to evaluate followpos: Example

For regular expression: $(a \mid b)^* a \#$



red – firstpos

blue – lastpos

Then we can calculate followpos

$\text{followpos}(1) = \{1,2,3\}$

$\text{followpos}(2) = \{1,2,3\}$

$\text{followpos}(3) = \{4\}$

$\text{followpos}(4) = \{\}$

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

Conversion from RE to DFA

Algorithm

Create the syntax tree of $(r) \#$

Calculate the functions: nullable, firstpos, lastpos & followpos

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, \dots, s_n are positions in S and symbols in those positions is a
 - $S' \leftarrow \text{followpos}(s_1) \cup \dots \cup \text{followpos}(s_n)$
 - $\text{move}(S, a) \leftarrow S'$
 - 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 $\#$

Conversion from RE to DFA

Example1

For the RE --- $(a \mid b)^* a \#$
 $\begin{matrix} 1 & 2 & 3 & 4 \end{matrix}$

$\text{followpos}(1)=\{1,2,3\}, \text{followpos}(2)=\{1,2,3\}, \text{followpos}(3)=\{4\}, \text{followpos}(4)=\{\}$

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

mark S_1

for a: $\text{followpos}(1) \cup \text{followpos}(3) = \{1,2,3,4\} = S_2$ $\text{move}(S_1, a) = S_2$

for b: $\text{followpos}(2) = \{1,2,3\} = S_1$ $\text{move}(S_1, b) = S_1$

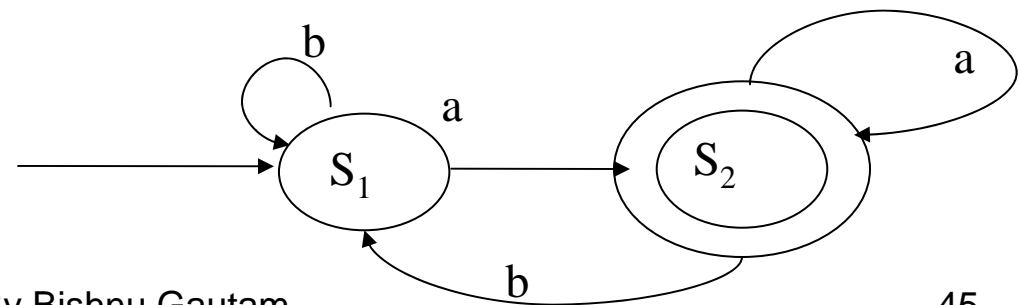
mark S_2

for a: $\text{followpos}(1) \cup \text{followpos}(3) = \{1,2,3,4\} = S_2$ $\text{move}(S_2, a) = S_2$

for b: $\text{followpos}(2) = \{1,2,3\} = S_1$ $\text{move}(S_2, b) = S_1$

start state: S_1

accepting states: $\{S_2\}$



By Bishnu Gautam

Conversion from RE to DFA

Example2

For RE---- (a | ϵ) b c* #

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

$S_1 = \text{firstpos}(\text{root}) = \{1, 2\}$

mark S_1

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

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

mark S_2

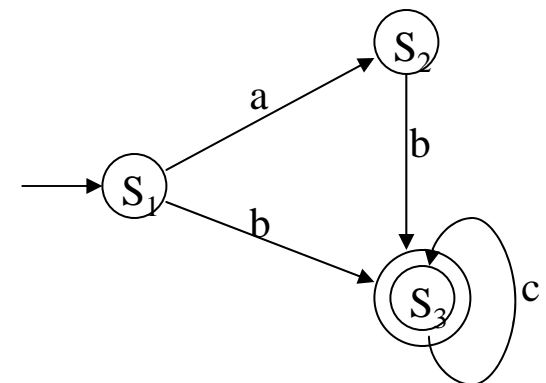
for b: followpos(2)={3,4}= S_3 move(S_2 , b) = S_3

mark S_3

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

start state: S_1

accepting states: { S_3 }



State Minimization in 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

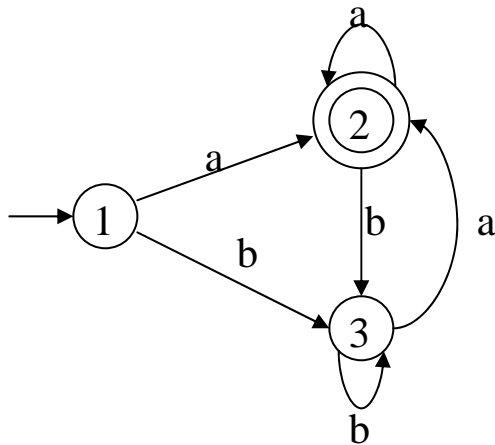
- partition G into subgroups such that states s_1 and s_2 are in the same group iff for all input symbols a , states s_1 and s_2 have transitions to states in the same group.

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.

State Minimization in DFA

Example 1



$$G_1 = \{2\}$$

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

G_2 cannot be partitioned because

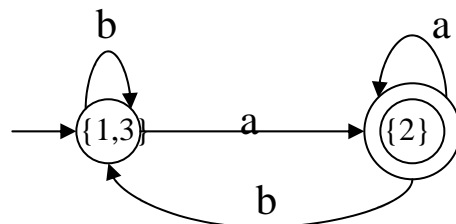
$$\text{move}(1,a)=2$$

$$\text{move}(1,b)=3$$

$$\text{move}(3,a)=2$$

$$\text{move}(2,b)=3$$

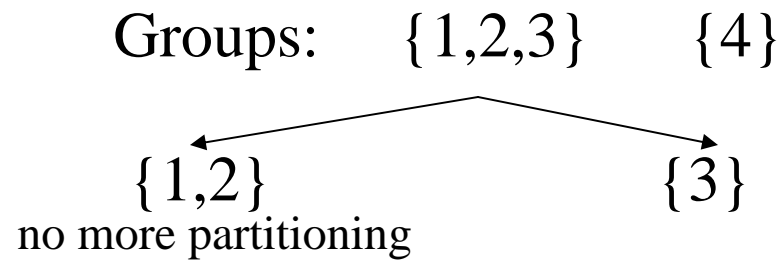
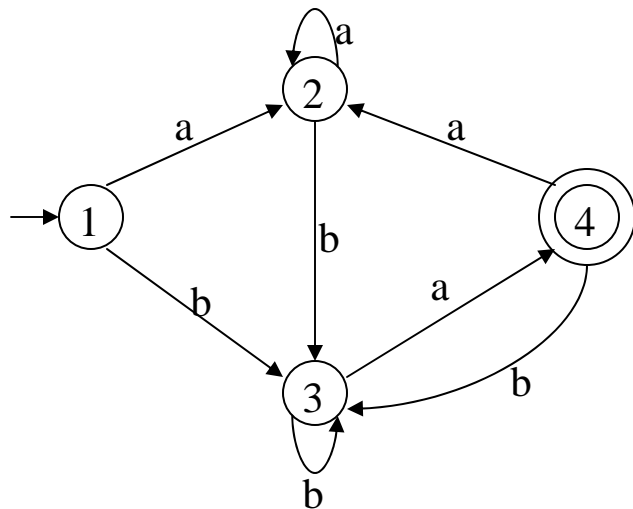
So, the minimized DFA (with minimum states)



By Bishnu Gautam

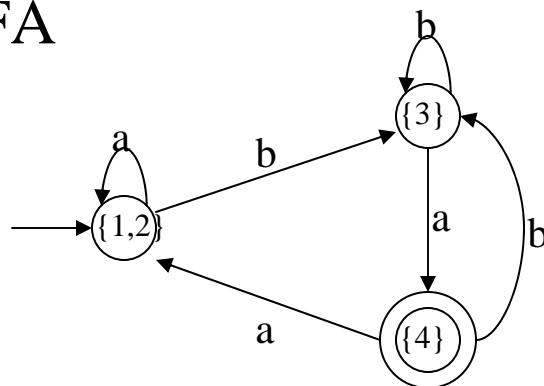
State Minimization in DFA

Example2



| <u>a</u> | <u>b</u> |
|----------|----------|
| 1->2 | 1->3 |
| 2->2 | 2->3 |
| 3->4 | 3->3 |

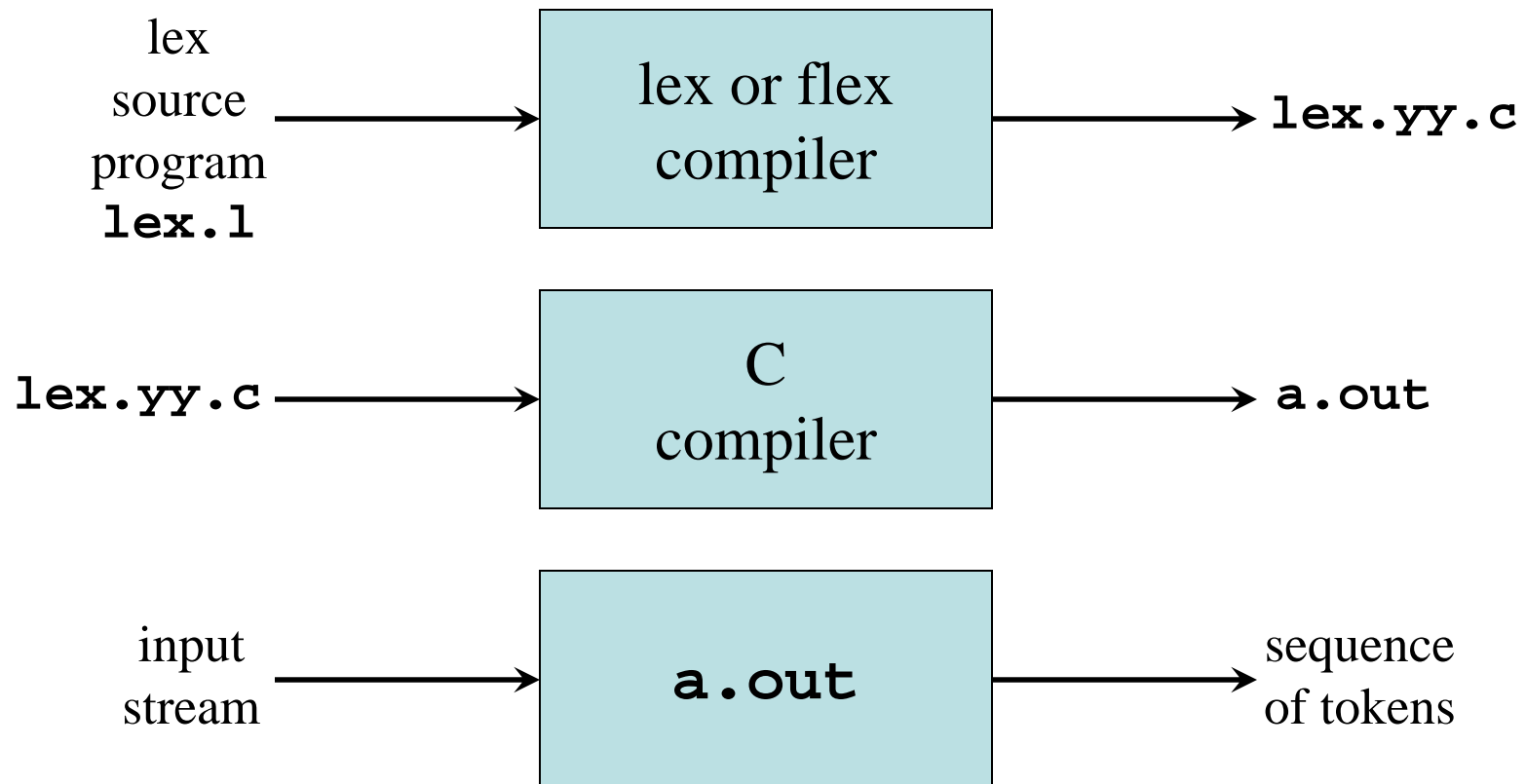
So, the minimized DFA



By Bishnu Gautam

Flex: Language for Lexical Analyzer

Systematically translate regular definitions into C source code for efficient scanning
Generated code is easy to integrate in C applications



Flex: An Introduction

Flex is a latter version of lex.

Flex was written by Jef Poskanzer. It was considerably improved by Vern Paxson and Van Jacobson.

It takes as its input a text file containing *regular expressions*, together with the action to be taken when each expression is matched. It produces an output file that contains *C source code* defining a function *yylex* that is a table-driven implementation of a DFA corresponding to the regular expressions of the input file. The Flex output file is then compiled with a C compiler to get an executable

Flex Specification

A *flex specification* consists of three parts:

regular definitions, C declarations in % { % }

% %

translation rules

% %

user-defined auxiliary procedures

The *translation rules* are of the form:

p_1 { *action*₁ }

p_2 { *action*₂ }

...

p_n { *action*_n }

In all parts of the specification comments of the form **/* comment text */** are permitted

Flex Specification

Definitions:

It consist two things:

- Any C code that is external to any function should be in `% { % }`
- Declaration of simple name definitions i.e specifying regular expression e.g

`DIGIT` `[0-9]`

`ID` `[a-z] [a-z0-9] *`

The subsequent reference is as `{DIGIT}`, `{DIGIT}+` or `{DIGIT}*`

Rules:

Contains a set of regular expressions and actions (C code) that are executed when the scanner matches the associated regular expression e.g

`{ID} printf("%s" , getlogin()) ;`

Any code that follows a regular expression will be inserted at the appropriate place in the recognition procedure `yylex()`

Finally the user code section is simply copied to `lex.yy.c`

Flex operators and Meaning

| | |
|------------------------------------|--|
| x | match the character x |
| \. | match the character . |
| “string” | match contents of string of characters |
| . | match any character except newline |
| ^ | match beginning of a line |
| \$ | match the end of a line |
| [xyz] | match one character x , y , or z (use \ to escape -) |
| [^xyz] | match any character except x , y , and z |
| [a-z] | match one of a to z |
| r* | closure (match zero or more occurrences) |
| r+ | positive closure (match one or more occurrences) |
| r? | optional (match zero or one occurrence) |
| r₁r₂ | match r₁ then r₂ (concatenation) |
| r₁ r₂ | match r₁ or r₂ (union) |
| (r) | grouping |
| r₁\r₂ | match r₁ when followed by r₂ |
| {d} | match the regular expression defined by d |

Flex Global Function, Variables & Directives

yylex() is the scanner function that can be invoked by the parser

yytext extern char *yytext; is a global char pointer holding the currently matched lexeme.

yylen extern int yylen; is a global int that contains the length of the currently matched lexeme.

ECHO copies yytext to the scanner's output

REJECT directs the scanner to proceed on to the "second best" rule which matched the input

yyomore() tells the scanner that the next time it matches a rule, the corresponding token should be appended onto the current value of *yytext* rather than replacing it

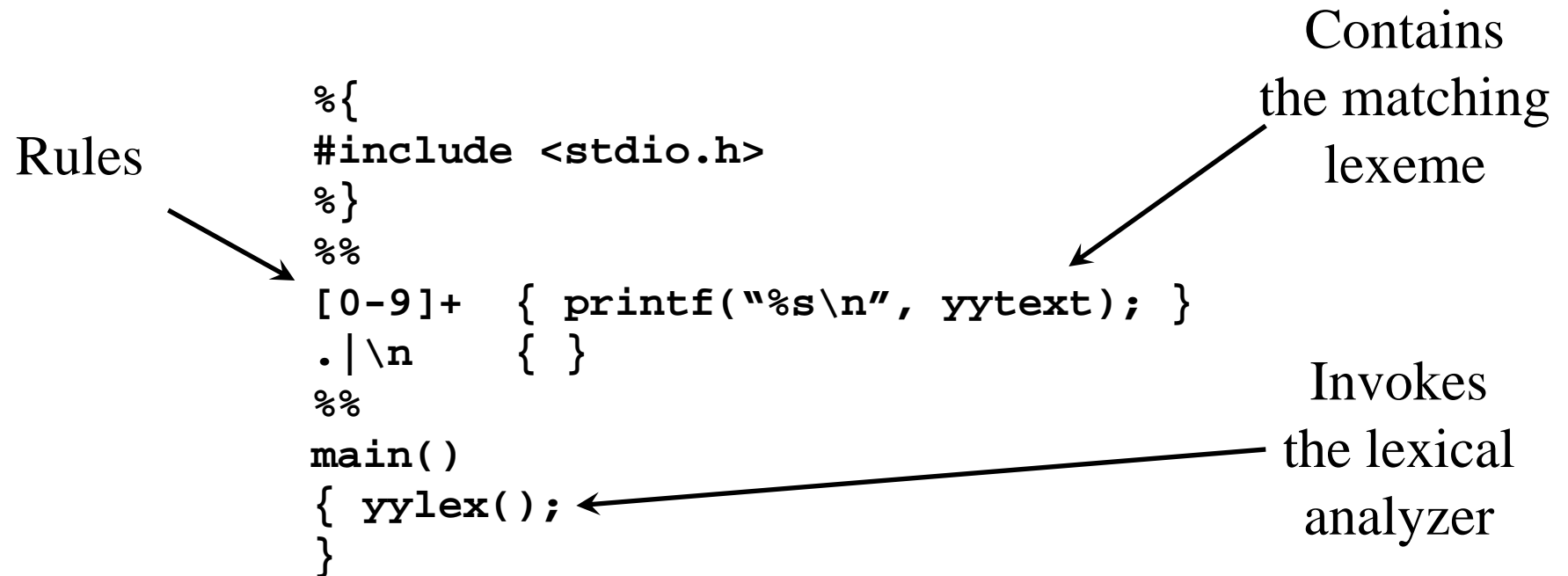
yyless(n) returns all but the first n characters of the current token back to the input stream, where they will be rescanned when the scanner looks for the next match

unput(c) puts the character c back onto the input stream. It will be the next character scanned

input() reads the next character from the input stream

YY_FLUSH_BUFFER flushes the scanner's internal buffer so that the next time the scanner attempts to match a token, it will first refill the buffer

Flex Example1



Flex Example2

Translation
rules



```
%{  
#include <stdio.h>  
int ch = 0, wd = 0, nl = 0;  
%}  
delim      [ \t]+  
%%  
\n         { ch++; wd++; nl++; }  
^{delim}   { ch+=yyleng; }  
{delim}    { ch+=yyleng; wd++; }  
.  
%%  
main()  
{ yylex();  
  printf("%8d%8d%8d\n", nl, wd, ch);  
}
```

Regular
definition



Flex Example3

Translation rules

```
%{
#include <stdio.h>
%}
digit      [0-9]
letter     [A-Za-z]
id         {letter}({letter}|{digit})*
%%
{digit}+   { printf("number: %s\n", yytext); }
{id}       { printf("ident: %s\n", yytext); }
.          { printf("other: %s\n", yytext); }
%%
main()
{ yylex();
}
```

Regular definitions

The diagram illustrates the components of a Flex lexer. The 'Translation rules' label points to the section where rules are defined (e.g., `digit`, `letter`, `id`), and the 'Regular definitions' label points to the regular expressions used in these rules (e.g., `[0-9]`, `[A-Za-z]`).

Flex File Execution

1. Save the Flex specification in a text file, say `wordcount.flex`.
2. Convert the Flex program file to a C program with the command `flex wordcount.flex`. If there is no error, a file named `lex.yy.c` will exist.
3. Compile this C: `gcc lex.yy.c -lfl -o wordcount`. The `lfl` option instructs the linker to use the routines in the Flex library if needed. This option is most of the time necessary in order for the linker to resolve all external references.
4. Run the word-counter with the command `./wordcount filename`. To verify the output, use the UNIX program `wc`

Lab Work

1. Write a flex program to scan an input C source file and replace all “float” by “double”
2. Write a flex program that scans an input C source file and produces a table of all macros (#defines) and their values.
3. Write a flex program that reads an input file that contains numbers (either integers or floating-point numbers) separated by one or more white spaces and adds all the numbers.
4. Write a flex program that reads an input file that contains numbers (either integers or floating-point numbers) separated by one or more white spaces and prints the minimum and maximum of the numbers.
5. Write a flex program that scans a text file and puts a HTML mailto tag around every email address present in the file.
6. Write a flex program that scans an input C source file and recognizes identifiers and keywords. The scanner should output a list of pairs of the form (token; lexeme), where token is either “identifier” or “keyword” and lexeme is the actual string matched.

Home Work

Question no 3.6, 3.15, 3.16 & 3.22 from the text book