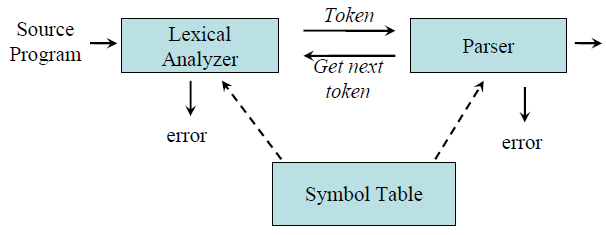
**Unit – 2 Lexical Analysis**

**Lexical Analyzer** reads the source program in character by character ways and returns the *tokens* of the source program. Normally a lexical analyzer doesn’t return a list of tokens, it returns a token only 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.



**Example:**

*newval := oldval + 12*

tokens: newval identifier

:= assignment operator

oldval identifier

+ add operator

12 a number

Put information about identifiers into the symbol table.

Regular expressions are used to describe tokens (lexical constructs).

A (Deterministic) Finite State Automaton (DFA) can be used in the implementation of a lexical analyzer.

**Tokens, Patterns, Lexemes**

1. A *token* is a logical building block of language. They are the sequence of characters having a collective meaning. Tokens are symbolic names for the entities that make up the text of the program; e.g. if for the keyword if, and id for any identifier. These make up the output of the lexical analyser.
2. A pattern is a rule that specifies when a sequence of characters from the input constitutes a token; Patterns are the rules for describing whether a given lexeme belonging to a token or not. Regular expressions are widely used to specify patterns. e.g the sequence i, f for the token if , and any sequence of alphanumerics starting with a letter for the token id.
3. A lexeme is a sequence of characters from the input that match a pattern (and hence constitute an instance of a token); for example if matches the pattern for if , and ktm123 matches the pattern for id. A token can represent more than one lexeme.

**Eg:** x = a +b \* 2

**lexemes:** { x, =, a, +, b, \*, 2 }

**tokens :** { <id,0>, <=>, <id,1>, <+>, <id,2>, <\*>, <num>}

**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.

Example: the tokens and the associated attribute for the following statement.

A=B\*C+2

<id, pointer to symbol table entry for A>

<assig operator>

<id, pointer to symbol table entry for B>

<mult\_op>

<id, pointer to symbol table entry for C>

<add\_op>

<num, integer value 2>

**Input 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 cannot 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.

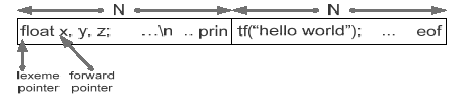


Fig: - An input buffer in two halves

Here, we divide the buffer into two halves with N-characters each.

Rather than reading character by character from file we read N input character at once. If there are fewer than N characters in input eof marker is placed.

There are two pointers (see in above fig.) the portion between lexeme pointer and forward pointer is current lexeme. Once the match for pattern is found, both the pointers points at the same place and forward pointer is moved.

The forward pointer performs tasks like below:

*If forward at end of first half then,*

*Reload second half*

*Forward++*

*end if*

*else if forward at end of second half then,*

*Reload first half*

*Forward=start of first half*

*end else if*

*else*

*forward++*

**Symbol Table**

Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc. Symbol table is used by both the analysis and the synthesis parts of a compiler.

A symbol table may serve the following purposes depending upon the language in hand:

* To store the names of all entities in a structured form at one place.
* To verify if a variable has been declared.
* To implement type checking, by verifying assignments and expressions in the source code are semantically correct.
* To determine the scope of a name (scope resolution).

A symbol table is simply a table which can be either linear or a hash table. It maintains an entry for each name in the following format:

<symbol name, type, attribute>

For example, if a symbol table has to store information about the following variable declaration:

static int interest;

then it should store the entry such as:

<interest, int, static>

A symbol table can be implemented in one of the following ways:

* Linear (sorted or unsorted) list
* Binary Search Tree
* Hash table

A symbol table should provide the following operations.

### insert(): This operation is more frequently used by analysis phase, i.e., the first half of the compiler where tokens are identified and names are stored in the table. This operation is used to add information in the symbol table about unique names occurring in the source code.

For example: int a; should be processed by the compiler as: insert(a, int);

* lookup(): This operation is used to search a name in the symbol table to determine if the symbol exists in the table. The format of lookup() function varies according to the programming language. The basic format should match the following:

lookup(symbol)

This method returns 0 (zero) if the symbol does not exist in the symbol table. If the symbol exists in the symbol table, it returns its attributes stored in the table.

**Error Handling**

A compiler should detect errors locate errors recover from errors. Error may be encountered in many phase of compiler. Objective of error handling is to go as far as possible in compilation whenever an error is encountered. Errors may occur in any of the three main analysis phases of compiling: lexical analysis, syntax analysis and semantic analysis.   
Examples:  
– Handling missing symbols during the lexical analysis by inserting symbol  
– Automatic type conversion during the semantic analysis

**Specification of Tokens**

**Stings & Languages**

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

– |*s* | denotes the length of string *s*

– ε denotes the empty string, thus |ε| = 0

**Operators on Strings:**

– Concatenation: xy represents the concatenation of strings x and y.

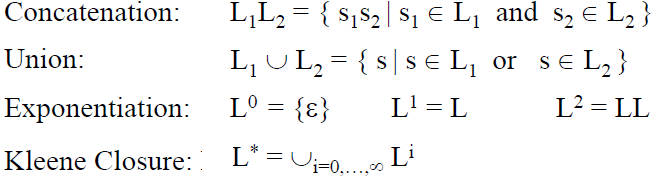
s ε = s

ε s = s

– Sn = s s s .. s ( n times) s0 = ε

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

**Operation on Languages**





**Operation on Languages: Example**

L1 = {a, b, c, d} L2 = {1, 2}

L1L2 = {a1, a2, b1, b2, c1, c2, d1, d2}

L1 ∪ L2 = {a, b, c, d, 1, 2}

L13 = all strings with length three (using a, b, c, d}

L1\* = all strings using letters a, b, c, d and empty string

L1+ = doesn’t include the empty string

**Regular Expressions**

Regular expressions are the algebraic expressions that are used to describe tokens of a programming language.

If *r* and *s* are regular expressions denoting languages *L1(r)* and *L2(s)* respectively, then

– *r+s* is a regular expression denoting *L1* (*r*) ∪ *L2(s)*

– *rs* is a regular expression denoting *L1* (*r*) *L2(s)*

– *r*\* is a regular expression denoting (*L1* (*r*))\*

– (*r*) is a regular expression denoting *L1* (*r*)

**Properties of Regular Expressions**

For regular expression r, s & t

r + s = s + r union is commutative)

r + (s + t) = (r + s) + t (union is associative)

(rs) t = r (st) (concatenation is associative)

r (s + t) = rs + rt (concatenation distributes over union)

ε r = r ε = r (ε is the identity element for concatenation)

r\*\* = r\* (closure is idempotent)

**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

4. (0\* 1 0\* 1 0\* 1 0\*) denotes the string having exactly three 1’s.

5. (1 1\* (0+ ε) 1 1\* (0+ ε) 1 1\*) denotes the string having at most two 0’s and at least three 1’s.

6. (A | B | C |………| Z | a | b | c |………| z | \_ |). ((A | B | C |………| Z | a | b | c |………| z | \_ |) (1 | 2 |…………….| 9))\* denotes the regular expression to specify the identifier like in C. [TU]

**Exercise:**

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

**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*.

The regular definition is a sequence of definitions of the form,

d1 → r1

d2 → r2

…………….

dn → rn

Where di is a distinct name and ri is a regular expression over symbols in Σ∪ {d1, d2... di-1}

Where, Σ = Basic symbol and

{d1, d2... di-1} = previously defined names.

**Regular Definitions: Examples**

Regular definition for specifying identifiers in a programming language like C

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

underscore →’\_’

digit →0 | 1 | 2 |…………….| 9

id → (letter | underscore).( letter | underscore | digit)\*

If we are trying to write the regular expression representing identifiers without using regular definition, it will be complex.

(A | B | C |………| Z | a | b | c |………| z | \_ |). ((A | B | C |………| Z | a | b | c |………| z | \_ |) (1 | 2 |…………….| 9))\*

**Exercise**:

1 Write regular definition for specifying floating point number in a programming language like C

**Soln:** digit →0 | 1 | 2 |…………….| 9

num→digit\* (**.**digit+)

2. Write regular definitions for specifying an integer array declaration in language like C

**Soln:** letter → A | B | C |………| Z | a | b | c |………| z

underscore →’\_’

digit → 1 | 2 |…………….| 9

array → (letter | underscore).( letter | underscore | digit)\* ([digit+.0\*])+

**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.

• 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.

**Design of a Lexical Analyzer**

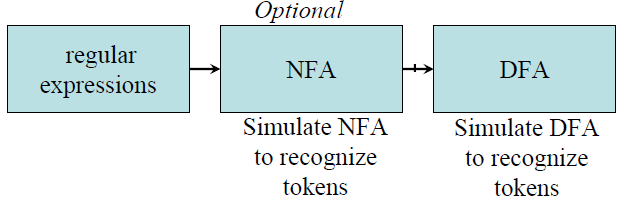
First, we define regular expressions for tokens; then we convert them into a DFA to get a lexical analyzer for our tokens.

**Algorithm1**:

Regular Expression → 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)**

An NFA is a 5-tuple (*S*, Σ, δ, *s*0, *F*) where

*S* is a finite set of *states*

Σ is a finite set of symbols

δ is a *transition function*

*s*0 ∈ *S* is the *start state*

*F* ⊆ *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.

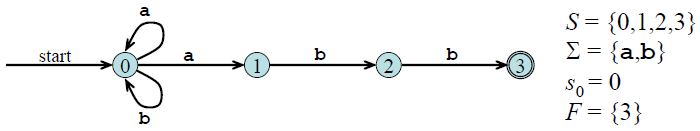


Fig: - NFA for regular expression (a + b)\*a b b

**ε- NFA**

In NFA if a transition made without any input symbol is called ε-NFA.

Here we need ε-NFA because the regular expressions are easily convertible to ε-NFA.

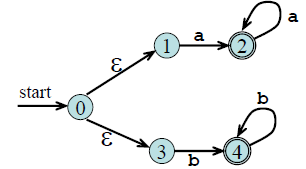


Fig: - ε-NFA for regular expression aa\* +bb\*

**Deterministic Finite Automaton (DFA)**

DFA is a special case of NFA. There is only difference between NFA and DFA is in the transition function.

In NFA transition from one state to multiple states take place while in DFA transition from one state to only one possible next state take place.

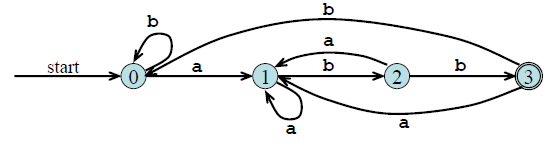


Fig:-DFA for regular expression (**a**+**b**)\***abb**

**Conversion: Regular Expression to NFA**

***Thomson’s Construction***

Thomson’s Construction is simple and systematic method.

It guarantees that the resulting NFA will have exactly one final state, and one start state.

**Method:**

-First parse the regular expression into sub-expressions

-construct NFA’s for each of the basic symbols in regular expression (r)

-Finally combine all NFA’s of sub-expressions and we get required NFA of given regular expression.

1. To recognize an empty string ε

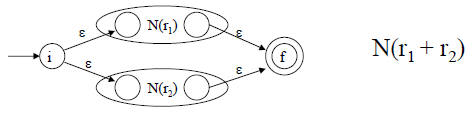


2. To recognize a symbol a in the alphabet Σ

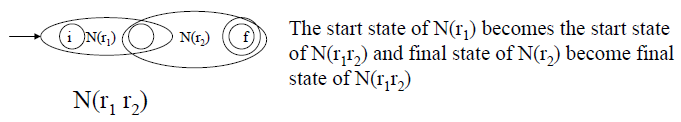


3. If N (r1) and N (r2) are NFAs for regular expressions r1 and r2

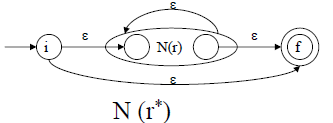
a. For regular expression r1 + r2



b. For regular expression r1 r2

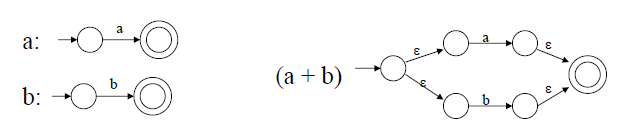


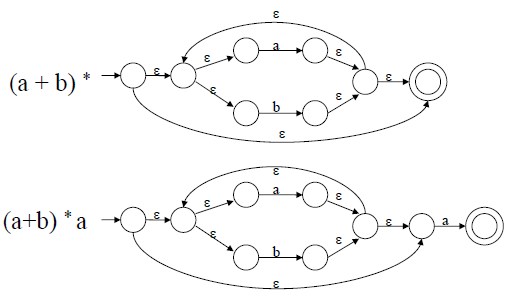
c. For regular expression r\*



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.

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





**Conversion from NFA to DFA**

**Subset Construction Algorithm**

put ε*-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* ∈ Σ **do**

*U* = ε*-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**

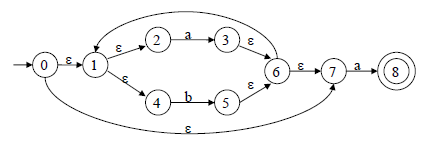
**end do**

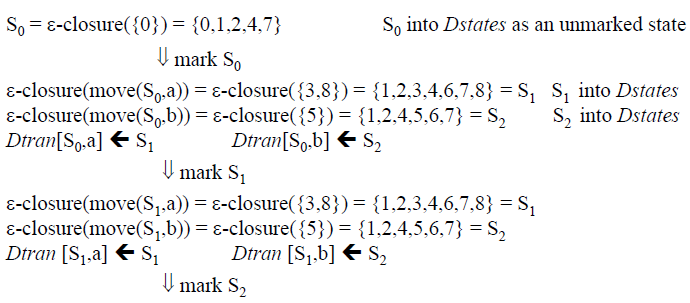
***The algorithm produces:***

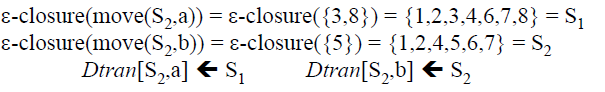
*Dstates: Dstates* is the set of states of the new DFA consisting of sets of states of the NFA

*Dtran: Dtran* is the transition table of the new DFA

**Subset Construction Example (NFA to DFA)**

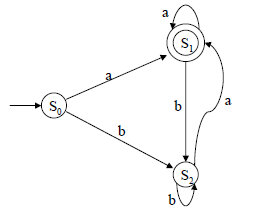






S0 is the start state of DFA since 0 is a member of S0= {0, 1, 2, 4, 7}

S1 is an accepting state of DFA since 8 is a member of S1 = {1, 2, 3, 4, 6, 7, 8}

this is final DFA

**Exercise:**

Convert the following regular expression first into NFA and then into DFA

1. 0+ (1+0)\*00
2. zero -> 0; one ->1; bit -> zero + one; bits -> bit\*

**Conversion from RE to DFA Directly**

Important States:

A state S of an NFA without ε- transition is called the important state if,



In an optimal state machine all states are important states

Augmented Regular Expression:

When we construct an NFA from the regular expression then the final state of resulting NFA is not an important state because it has no transition. Thus to make important state of the accepting state of NFA we introduce an ‘augmented’ character (#) to a regular expression r.

This resulting regular expression is called the augmented regular expression of original expression r.

**Conversion steps:**

1. Augment the given regular expression by concatenating it with special symbol # i.e. r ->(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.

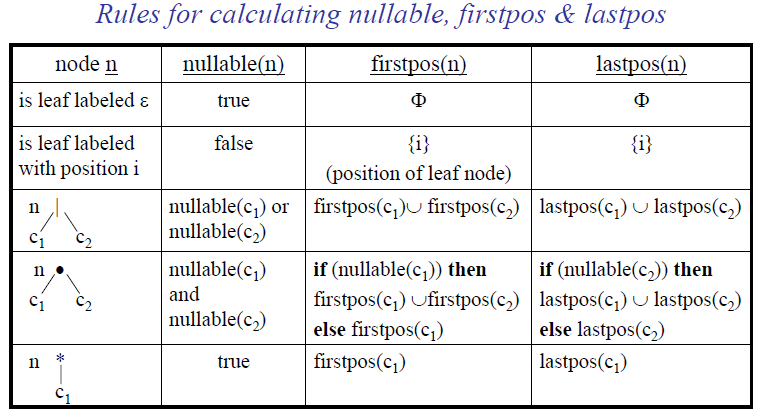
3. Then each alphabet symbol (plus #) will be numbered (position numbers)

4. Traverse the tree to construct functions *nullable*, *firstpos*, *lastpos*, and *followpos*

5. Finally construct the DFA from the *followpos*

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

syntax tree.



*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*) ∪ *firstpos*(*c*2)

**end do**

**else if** *n* is a star-node

**for** each *i* in *lastpos*(*n*) **do**

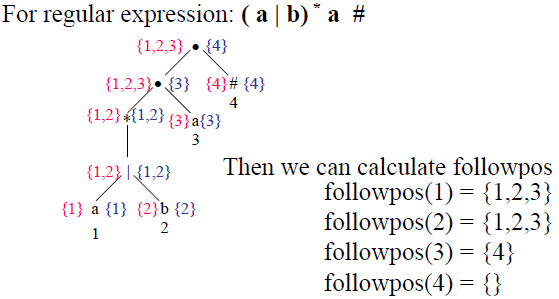
*followpos*(*i*) := *followpos*(*i*) ∪ *firstpos*(*n*)

**end do**

**end if**

**end do**

*How to evaluate followpos: Example*



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

**Conversion from RE to DFA Example1**

*Note: - the start state of DFA is firstpos(root)*

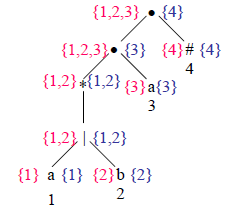
*the accepting states of DFA are all states containing the position of #*

For the RE --- ( a | b) \* a

Its augmented regular expression is;



The syntax tree is:



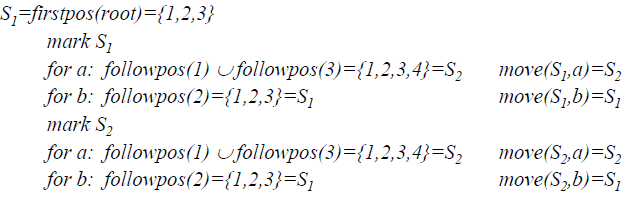
Now we calculate followpos,

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

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

followpos(3)={4}

followpos(4)={}



Now



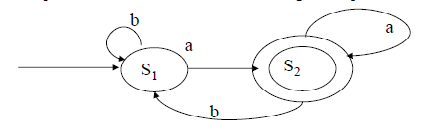
Note:- Accepting states=states containing position of # ie 4. 

Fig: Resulting DFA of given regular expression

*Conversion from RE to DFA*

Example2

For RE---- ( a | ε) b c\* #

1 2 3 4

followpos(1)={2}

followpos(2)={3,4}

followpos(3)={3,4}

followpos(4)={}

*S1=firstpos(root)={1,2}*

*mark S1*

*for a: followpos(1)={2}=S2 move(S1,a)=S2*

*for b: followpos(2)={3,4}=S3 move(S1,b)=S3*

*mark S2*

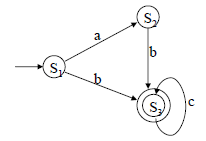
*for b: followpos(2)={3,4}=S3 move(S2,b)=S3*

*mark S3*

*for c: followpos(3)={3,4}=S3 move(S3,c)=S3*

start state: S1

accepting states: {S3}

a, c

c a, b

fig:- DFA for above RE

**State minimization in DFA:**

Partition the set of states into two groups:

– G1: set of accepting states

– G2: set of non-accepting states

For each new group G:

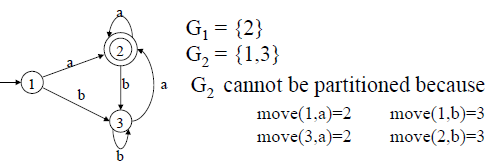
– partition G into subgroups such that states s1 and s2 are in the same group if for all input symbols a, states s1 and s2 have transitions to states in the same group.

Start state of the minimized DFA is the group containing the start state of the original DFA.

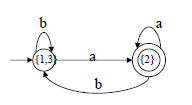
Accepting states of the minimized DFA are the groups containing the accepting states of the original DFA.

State Minimization in DFA

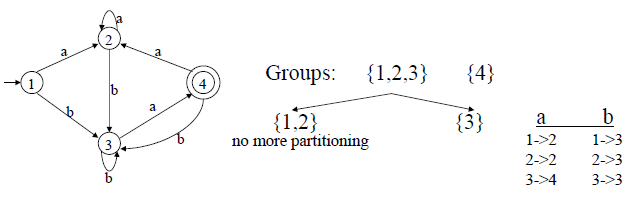
Example1:



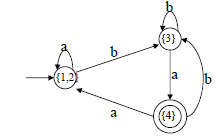
So, the minimized DFA (with minimum states)

****

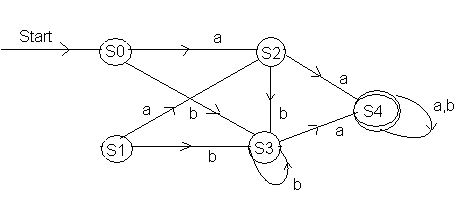
Example 2:



So minimized DFA is:



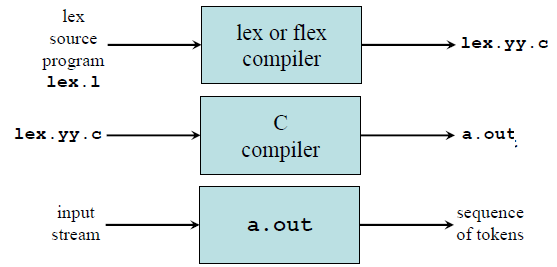
Exercise: Minimize the DFA.



**Flex: Language for Lexical Analyzer**

**Flex: An introduction**

FLEX (Fast LEXical analyzer generator) is a tool for generating scanners. A scanner is a program which recognizes lexical patterns in text. In stead of writing a scanner from scratch, you only need to identify the *vocabulary* of a certain language (e.g. Simple), write a specification of patterns using regular expressions (e.g. DIGIT [0-9]), and FLEX will construct a scanner for you. FLEX is generally used in the manner depicted here:



First, FLEX reads a specification of a scanner either from an input file \*.lex, or from standard input, and it generates as output a C source file *lex.yy.c*. Then, *lex.yy.c* is compiled and linked with the "-lfl" library to produce an executable *a.out*. Finally, *a.out* analyzes its input stream and transforms it into a sequence of tokens.

* **\*.lex** is in the form of pairs of regular expressions and C code. ([sample1.lex](http://alumni.cs.ucr.edu/~lgao/teaching/flex/sample1.lex), [sample2.lex](http://alumni.cs.ucr.edu/~lgao/teaching/flex/sample2.lex))
* **lex.yy.c** defines a routine *yylex()* that uses the specification to recognize tokens.
* **a.out** is actually the scanner!

The flex program reads the given input files, or its standard input if no file names are given, for a description of a scanner to generate. The description is in the form of pairs of regular expressions and C code, called rules. flex generates as output a C source file, ‘lex.yy.c’ by default, which defines a routine yylex(). This file can be compiled and linked with the flex runtime library to produce an executable. When the executable is run, it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code.

**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 {action1}

*p*2 {action2}

…………………..

*pn* { *actionn* }

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

***Regular 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}\*

***Translation 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*

**Practice**

• Get familiar with FLEX

1. Try sample\*.lex

2. Command Sequence:

flex sample\*.lex

gcc lex.yy.c -lfl

./a.out

**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)

*r*1*r*2 match *r*1 then *r*2 (concatenation)

*r*1**|***r*2 match *r*1 or *r*2 (union)

**(** *r* **)** grouping

*r*1**\***r*2 match *r*1 when followed by *r*2

**{***d***}** match the regular expression defined by *d*

‘r{2,5}’ anywhere from two to five r’s

‘r{2,}’ two or more r’s

‘r{4}’ exactly 4 r’s

**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.

***yyleng*** extern int yyleng; 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

***yymore()*** 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.

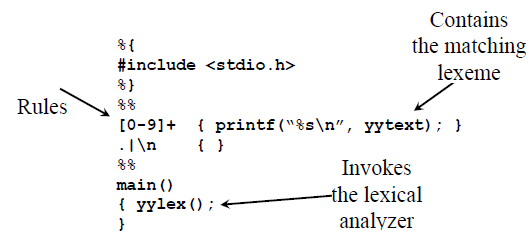
***yyless(n)*** returns all but the first n characters of the current token back to the input stream, where they will be rescanned when the scanner looks for the next match

***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***



***Example 2 Sample2.lex***

/\*

\* Description: Count the number of characters and the number of lines

\* from standard input

\* Usage:

(1) $ flex sample2.lex

\* (2) $ gcc lex.yy.c -lfl

\* (3) $ ./a.out

\* stdin> whatever you like

\* stdin> Ctrl-D

\* Questions: Is it ok if we do not indent the first line?

\* What will happen if we remove the second rule?

\*/

int num\_lines = 0, num\_chars = 0;

%%

\n ++num\_lines; ++num\_chars;

. ++num\_chars;

%%

main()

{

yylex();

printf("# of lines = %d, # of chars = %d\n", num\_lines, num\_chars);

}

**Example 3: Sample1.lex**

/\*

\* Sample Scanner1:

\* Description: Replace the string "username" from standard input

\* with the user's login name (e.g. lgao)

\* Usage: (1) $ flex sample1.lex

\* (2) $ gcc lex.yy.c -lfl

\* (3) $ ./a.out

\* stdin> username

\* stdin> Ctrl-D

\* Question: What is the purpose of '%{' and '%}'?

\* What else could be included in this section?

\*/

%{

/\* need this for the call to getlogin() below \*/

#include <unistd.h>

%}

%%

username printf("%s\n", getlogin());

%%

main()

{

yylex();

}

***Example 4***

/\* Sample Scanner

\* Description: Recognize the 32-bit hexadecimal integer from stdin

\* Pattern: 0[xX]([0-9a-fA-F]{1,8})

\*/

digit [0-9]

alpha [a-fA-F]

hextail ({digit}|{alpha}){1,8}

hex 0[xX]{hextail}

%%

{hex} printf("Found a HEX number %s !", yytext);

. printf("");

%%

main()

{

printf("Give me your input:\n");

yylex();

}

**Practice**

|  |
| --- |
| * Get familiar with FLEX   1. Try sample\*.lex   2. Command Sequence:     flex sample\*.lex     gcc lex.yy.c -lfl     ./a.out * Understand the input file   1. Format:     definitions     %%     rules     %%     user code   2. The *definitions* section: "name definition" The *rules* section: "pattern action" The *user code* section: "yylex() routine"   3. Try to answer questions listed in the sample files |