[Lexical Analysis]

Compiler Design and Construction (CSc 352)

Compiled By

Bikash Balami

Central Department of Computer Science and Information Technology (CDCSIT)

Tribhuvan University, Kirtipur

Kathmandu, Nepal

Lexical Analysis

This is the initial part of reading and analyzing the program text. The text is read and divided into tokens, each of which corresponds to a symbol in a programming language, e.g. a variable name, keyword or number etc. So a lexical analyzer or lexer, will as its input take a string of individual letters and divide this string into tokens. It will discard comments and white-space (i.e. blanks and newlines).

Overview of Lexical Analysis

A lexical analyzer, also called a scanner, typically has the following functionality and characteristics.

- Its primary function is to convert from a sequence of characters into a sequence of tokens. This means less work for subsequent phases of the compiler.
- The scanner must Identify and Categorize specific character sequences into tokens. It must know whether every two adjacent characters in the file belong together in the same token, or whether the second character must be in a different token.
- Most lexical analyzers discard comments & whitespace. In most languages these characters serve to separate tokens from each other.
- Handle lexical errors (illegal characters, malformed tokens) by reporting them intelligibly to the user.
- Efficiency is crucial; a scanner may perform elaborate input buffering
- Token categories can be (precisely, formally) specified using regular expressions, e.g.
- `IDENTIFIER=[a-zA-Z][a-zA-Z0-9]*`
- Lexical Analyzers can be written by hand, or implemented automatically using finite automata.

Role of Lexical Analyzer

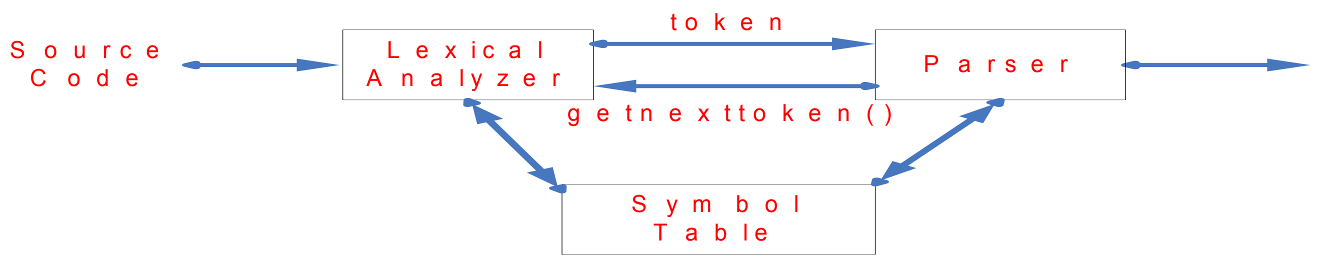


Figure:- Interaction of lexical analyzer with parser

The lexical analyzer works in lock step with the parser. The parser requests the lexical analyzer for the next token whenever it requires one using *getnexttoken()*. Lexical analyzer may also perform other auxiliary operations like removing redundant white spaces, removing token separators (like semicolon ;) etc. Some other operations performed by lexer, includes removal of comments, providing line number to the parser for error reporting. The Function of a lexical Analyzer is to read the input stream representing the Source program, one character at a time and to translate it into valid tokens.

Issues in Lexical Analysis

There are several reasons for separating the analysis phase of compiling into lexical analysis and parsing.

1) Simpler design is the most important consideration. The separation of lexical analysis from syntax analysis often allows us to simplify one or the other of these phases.

2) Compiler efficiency is improved.

3) Compiler portability is enhanced.

Tokens, Patterns, Lexemes

In compiler, a *token* is a single word of source code input. Tokens are the separately identifiable block with collective meaning. When a string representing a program is broken into sequence of substrings, such that each substring represents a constant, identifier, operator , keyword etc of the language, these substrings are called the tokens of the Language. They are the building block of the programming language. E.g. if, else, identifiers etc.

Lexemes are the actual string matched as token. They are the specific characters that make up of a token. For example ***abc*** and ***123***. A token can represent more than one lexeme. i.e. token ***intnum*** can represent lexemes ***123***, ***244***, ***4545*** etc.

Patterns are rules describing the set of lexemes belonging to a token. This is usually the regular expression. E.g. ***intnum*** token can be defined as [0-9][0-9]*.

Attribute of tokens

When a token represents more than one lexeme, lexical analyzer must provide additional information about the particular lexeme. i.e. In case of more than one lexeme for a token, we need to put extra information about the token. This additional information is called the *attribute* of the token. For e.g. token ***id*** matched ***var1, var2*** both, here in this case lexical analyzer must be able to represent ***var1***, and ***var2*** as different identifiers.

Example: take statement, ***area = 3.1416 * r * r***

1. getnexttoken() returns (id, attr) where *attr* is pointer to ***area*** to symbol table
2. getnexttoken() returns (assign) no attribute is needed, if there is only one assignment operator
3. getnexttoken() returns (floatnum, 3.1416) where 3.1416 is the actual value of ***floatnum***

…. So on.

Token type and its attribute uniquely identify a lexeme.

Lexical Errors

Though error at lexical analysis is normally not common, there is possibility of errors. When the error occurs the lexical analyzer must not halt the process. So it can print the error message and continue. Error in this phase is found when there are no matching string found as given by the pattern. Some error recovery techniques includes like deletion of extraneous character, inserting missing character, replacing incorrect character by correct one, transposition of adjacent characters etc. Lexical error recovery is normally expensive process. For e.g. finding the number of transformation that would make the correct tokens.

Implementing Lexical Analyzer (Approaches)

1. Use lexical analyzer generator like flex that produces lexical analyzer from the given specification as regular expression. We do not take care about reading and buffering the input.
2. Write a lexical analyzer in general programming language like C. We need to use the I/O facility of the language for reading and buffering the input.
3. Use the assembly language to write the lexical analyzer. Explicitly mange the reading if input.

The above strategies are in increasing order of difficulty, however efficiency may be achieved and as a matter of fact since we deal with characters in lexical analysis, it is better to take some time to get efficient result.

Look Ahead and Buffering

Most of the time recognizing tokens needs to look ahead several other characters after the matched lexeme before the token match is returned. For e.g. ***int*** is a keyword in C but ***integer*** is an identifier so when the scanner reads i, n, t then this time it has to look for other characters to see whether it is just ***int*** or some other word. In this case at next time we need move back to rescan the input again for the characters that are not used for the lexeme and this is time consuming . To reduce the overhead, and efficiently move back and forth ***input buffering*** technique is used.

Input Buffering

We will consider look ahead with 2N buffering and using the sentinels.

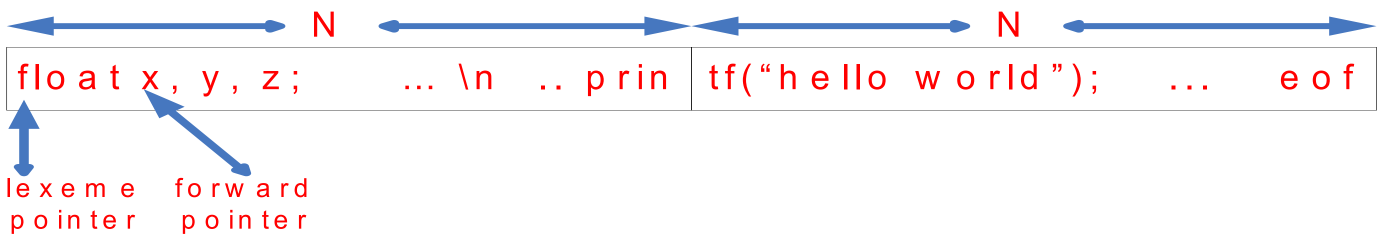


Figure:- An input buffer in two halves

We divide the buffer into two halves with N-characters each. normally N is the number if characters in one disk block like 1024 or 4096. Rather than reading character by character from file we read N input character at once. If there are fewer than N character in input eof marker is placed. There are two pointers (see fig in previous slide). The portion between lexeme pointer and forward pointer is current lexeme. Once the match for pattern is found both the pointers points at same place and forward pointer is moved. This method has limited look ahead so may not work all the time say multi line comment in C. In this approach we must check whether end of one half is reached (two test each time if forward pointer is not at end of halves) or not each time the forward pointer is moved. The number of such tests can be reduced if we place sentinel character at the end of each half.

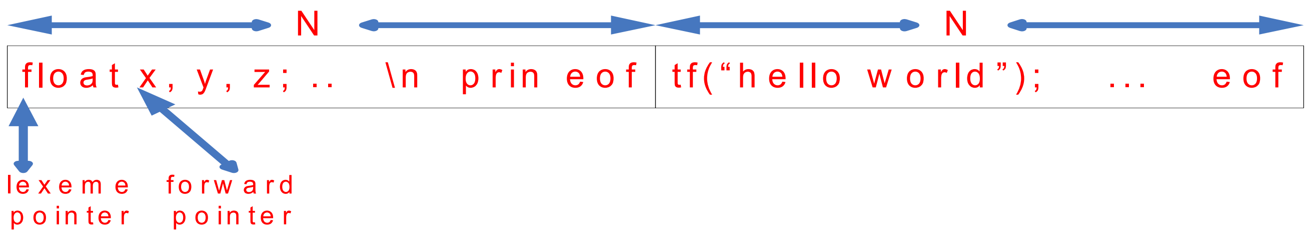


Figure:- Sentinels at end of each buffer half

```
if forward points end of first half
        reload second half
        forward++;
else if forward points end of second half
        reload first half
        forward = start of first half
else
        forward++
```

Figure:- Code to advance forward pointer(first scheme)

```
forward++
if forward points eof
        if forward points end of first half
            reload second half
            forward++;
        else if forward points end of second half
            reload first half
```

```
                    forward = start of first half
    else
            terminate lexical analysis
```

Figure:- Lookahead code with sentinels

Specifications of Tokens

Regular expressions are an important notation for specifying patterns. Each pattern matches a set of strings, so regular expressions will serve as names for sets of strings.

Some Definitions

- **Alphabets**
    - An alphabet A is a set of symbols that generate languages. For e.g. {0-9} is an alphabet that is used to produce all the non negative integer numbers. {0-1} is an alphabet that is used to produce all the binary strings.
- **String**
    - A string is a finite sequence of characters from the alphabet A Given the alphabet A, A^2 = A.A is set of strings of length 2, similarly A^n is set of strings of length n. The **length** of the string w is denoted by |w| i.e. numbet of characters (symbols) in w.We also have $A^0 = \{\varepsilon\}$, where ε is called empty string. |s| denotes the length of the string s.
- **Kleene Closure**
    - Kleene closure of an alphabet A denoted by A* is set of all strings of any length (0 also) possible from A. Mathematically $A^* = A^0 \cup A^1 \cup A^2 \cup \ldots \ldots\ldots$ For any string, w over alphabet A, $w \in A^*$.
- A **language** L over alphabet A is the set such that $L \subseteq A^*$.
- The string s is called **prefix** of w, if the string s is obtained by removing zero or more trailing characters from w. If s is a proper prefix, then $s \neq w$.
- The string s is called **suffix** of w, if the string s is obtained by deleting zero or more leading characters from w. we say s as proper suffix if $s \neq w$.
- The string s is called **substring** of w if we can obtain s by deleting zero or more leading or trailing characters from w. We say s as proper substring if $s \neq w$.
- Regular Operators

- The following operators are called **regular operators** and the language formed called **regular language.**
  - **.** → Concatenation operator, R.S = {rs | r ∈ R and s ∈ S }.
  - * → Kleene * operator, $A^* = \bigcup_{i=1}^{\infty} A^i$
  - +/∪/|→ Choice/union operator, R ∪ S = {t | t ∈ R or t ∈ S }.

Regular Expression (RE)

We use regular expression to describe the tokens of a programming language.

Basis Symbol

- ε is a regular expression denoting language { ε }
- a ∈ A 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 $L_1(r)$

Examples

- (1+0)*0 is RE that gives the binary strings that are divisible by 2.
- 0*10*+0*110* is RE that gives binary strings having at most 2 1s.
- (1+0)*00 denotes the language of all strings that ends with 00 (binary number multiple of 4)
- (01)* + (10)* denotes the set of all strings that describes alternating 1s and 0s

Properties of RE

- r+s = s+r (+ is commutative)
- r+(s+t) = (r+s)+t ; r(st) = (rs)t (+ and . are associative)
- r(s+t) = (rs)+(rt); (r+s)t =(rt)+st) (. distributes over +)
- εr = rε (ε is identity element)
- r* = (r+ε)* (relation between * and ε)
- r** = r* (* is idempotent)

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. i.e defining RE giving a name and reusing it as basic symbol to produce another RE, gives the regular definition.
Example

- $d_1 \rightarrow r_1$, $d_2 \rightarrow r_2$, …, $d_n \rightarrow r_n$, Where each d_is are distinct name and r_is are REs over the alphabet $A \cup \{d_1, d_2, \ldots d_{i-1}\}$
- In C the RE for identifiers can be written using the regular definition as
    - letter → a + b + …. +z + A + B + …+ Z.
    - digit → 0 + 1 + …+9.
    - identifier → (letter + _)(letter + digit + _)*

[Note: Remember recursive regular definition may not produce RE, that means

digits → digits digits | digits *is wrong!!!*

Recognition of Tokens

A recognizer for a language is a program that takes a string *w*, and answers "**YES**" if *w* is a sentence of that language, otherwise "**NO**". The tokens that are specified using RE are recognized by using transition diagram or finite automata (FA). Starting from the start state we follow the transition defined. If the transition leads to the accepting state, then the token is matched and hence the lexeme is returned, otherwise other transition diagrams are tried out until we process all the transition diagram or the failure is detected. Recognizer of tokens takes the language L and the string s as input and try to verify whether $s \in L$ or not. There are two types of Finite Automata.

1. Deterministic Finite Automaton (DFA)
2. Non Deterministic Finite Automaton (NFA)

Deterministic Finite Automaton (DFA)

FA is deterministic, if there is exactly one transition for each (state, input) pair. It is faster recognizer but it make take more spaces. DFA is a five tuple (S, A, S_0, δ, F) where, S → finite set of states

A → finite set of input alphabets

S_0 → starting state

δ → transition function i.e. $\delta: S \times A \rightarrow S$

F → set of final states $F \subseteq S$

Implementing DFA

The following is the algorithm for simulating DFA for recognizing given string. For a given string w, in DFA D, with start state q_0, the output is "**YES**", if D accepts w, otherwise "**NO**".

```
DFASim(D, q0)    {
          q = q0;
          c = getchar();
          while (c != eof)
          {
                    q = move (q, c);  //this is δ function.
                    c = getchar();
          }
          if (s is in H)
                    return "yes'; // if D accepts s
          else
                    return "false";
}
```

Figure:- Simulating DFA

Non Deterministic Automata (NFA)

FA is non deterministic, if there is more than one transition for each (state, input) pair. It is slower recognizer but it make take less spaces. An NFA is a five tuple (S, A, S_0, δ, F) where,

$S \rightarrow$ finite set of states

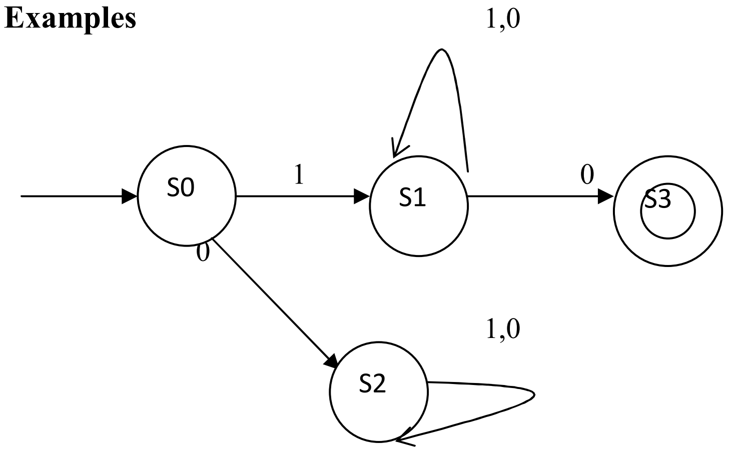
$A \rightarrow$ finite set of input alphabets

$S_0 \rightarrow$ starting state

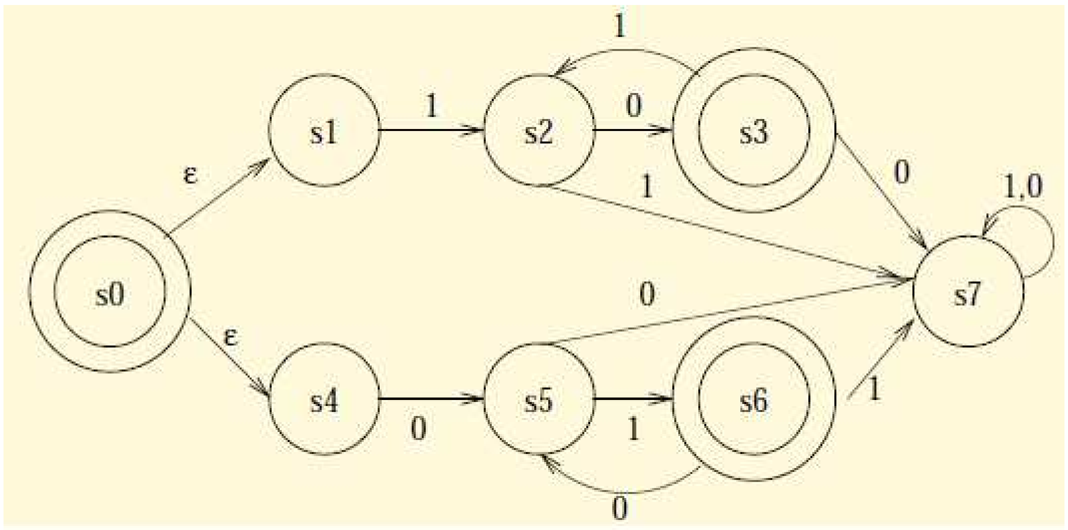
$\delta \rightarrow$ transition function i.e. $\delta: S \times A \rightarrow 2^{|S|}$

$F \rightarrow$ set of final states $F \subseteq S$

Examples



The above state machine is an NFA having a non – deterministic transition at state S1. On reading 0 it may either stay in S1 or goto S3 (accept). It's regular expression is $1(1 + 0)^*0$.



The above figure shows a state machine with ε moves that is equivalent to the regular expression (10)* + (01)*. The upper half of the a machine recognizes (10)* and the lower half recognizes (01)*. The machine non – deterministically moves to the upper half of the lower half when started from state s0.

Algorithm

```
S = ε-closure({$S_0$}) // set of all states that can be accessed from $S_0$ by ε-transitions
c = getchar()
while(c            != eof)
{
        S = ε-closure(move(S,c)) //set of all states that can be accessible from a state in S by a transition on c
        c = getchar()
}
if ( $S \cap F \neq \phi$) then
        return "YES"
else
        return "NO"
```

Figure :- Simulating NFA

Reducibility

1. NFA to DFA
2. RE to NFA
3. RE to DFA

NFA to DFA (sub set construction)

This sub set construction is an approach for an algorithm that constructs DFA from NFA, that recognizes the same language. Here there may be several accepting states in a given subset of nondeterministic states. The accepting state corresponding to the pattern listed first in the lexical analyser generator specification has priority. Here also state transitions are made until a state is reached which has no next state for the current input symbol. The last input position at which the DFA entered an accepting state gives the lexeme. We need the following operations.

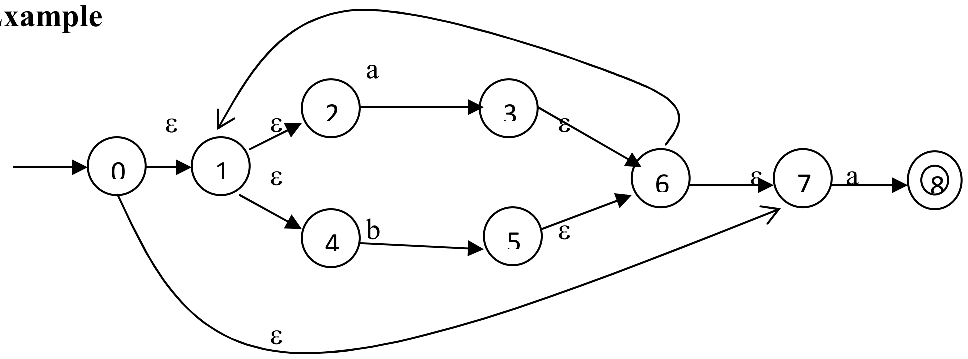
- ε-closure(S) → the set of NFA states reachable from NFA state *S* on ε-transition
- ε-closure(T) → the set of NFA states reachable from some NFA states *S* in *T* on ε-transition
- Move(T,a) → the set of NFA states to which there is a transition on input symbol *a* from NFA state *S* in *T*.

Subset Construction Algorithm

```
Put ε-closure(S_0) as an unmarked state in DStates
While there is an unmarked state T in DStates do
    Mark T
    For each input symbol a ∈ A 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
```

- *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
- A set 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)

Example



S_0 = ε-closure({0}) = {0, 1, 2, 7} → S_0 into *DStates* 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 *DStates* as an unmarked state

ε-closure(move(S_0,b)) = ε-closure({5}) = {1, 2, 4, 5, 6, 7} = S_2 → S_2 into *DStates* as an unmarked state

DTran[S_0, a] ← S_1

DTran[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

DTran[S_1, a] ← S_1

DTran[S_1, b] ← S_2

Mark S2

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

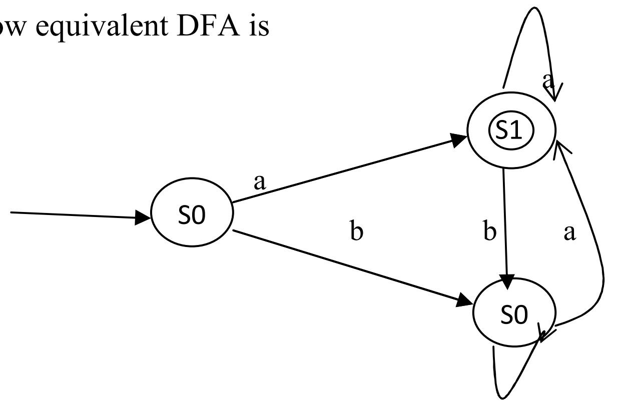
DTran[S_2, a] ← S_1

DTran[S_2, b] ← S_2

S_0 is the start 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 $S_1 = \{1, 2, 3, 4, 6, 7, 8\}$

Now equivalent DFA is



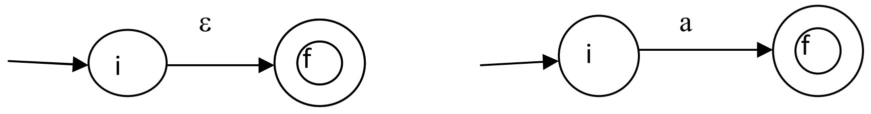
RE to NFA (Thomson's Construction)

Input → RE, r, over alphabet A

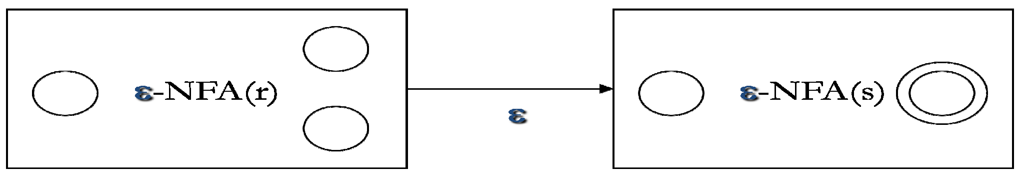
Output → ε-NFA accepting L(r)

Procedure → process in bottom up manner by creating ε-NFA for each symbol in A including ε. Then recursively create for other operations as shown below

1. For a in A and ε, both are RE and constructed as

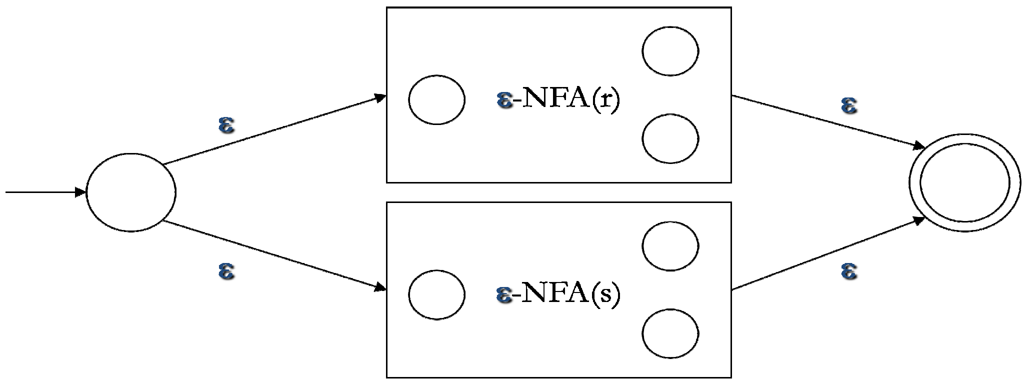


2. For REs r and s, r.s is RE too and it is constructed as

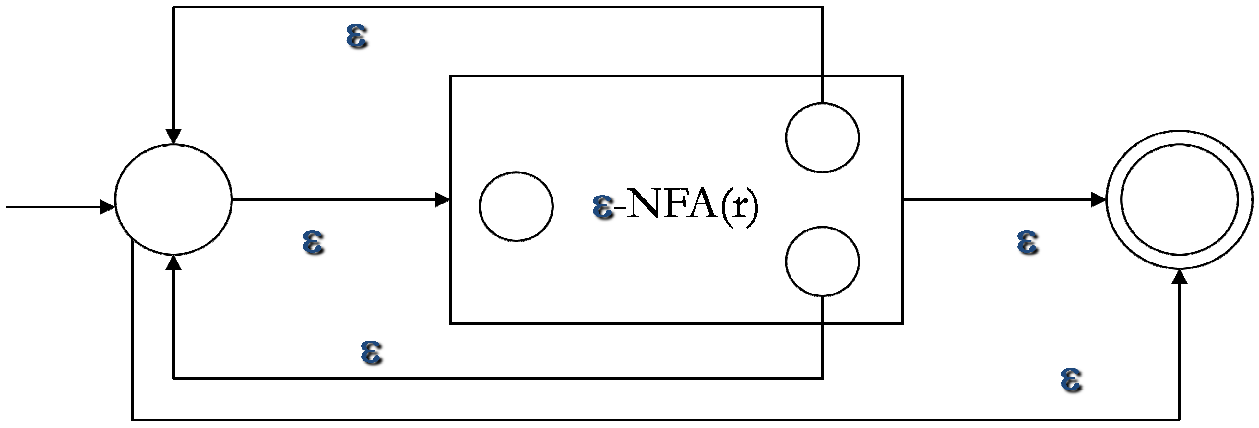


   i.e. The start of r, becomes the start of r.s and final state of s becomes the final state of r.s

3. For REs r and s, r+s is RE too and it is constructed as



4. For REs r, r* is RE too and it is constructed as

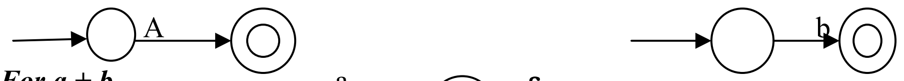


- For REs (r), ε-NFA((r)) is ε-NFA(r)
- ε-NFA(r) has at most twice as many state as number of symbols and operators in r since each construction takes a tmost two new states.
- ε-NFA(r) has exactly one start and one accepting state.
- Each state of ε-NFA(r) has either one outgoing transition on a symbol in A or at most two outgoing ε-transitions

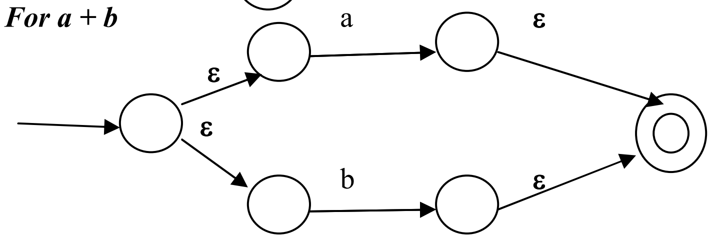
Example

(a + b)*a To NFA

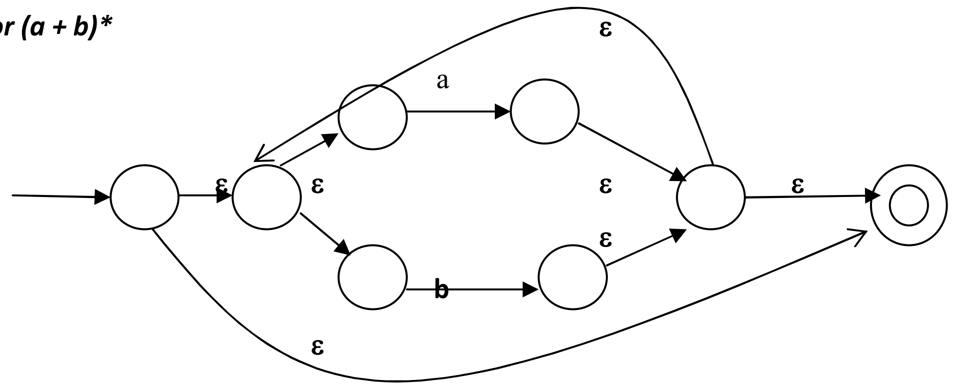
For a ***For b***



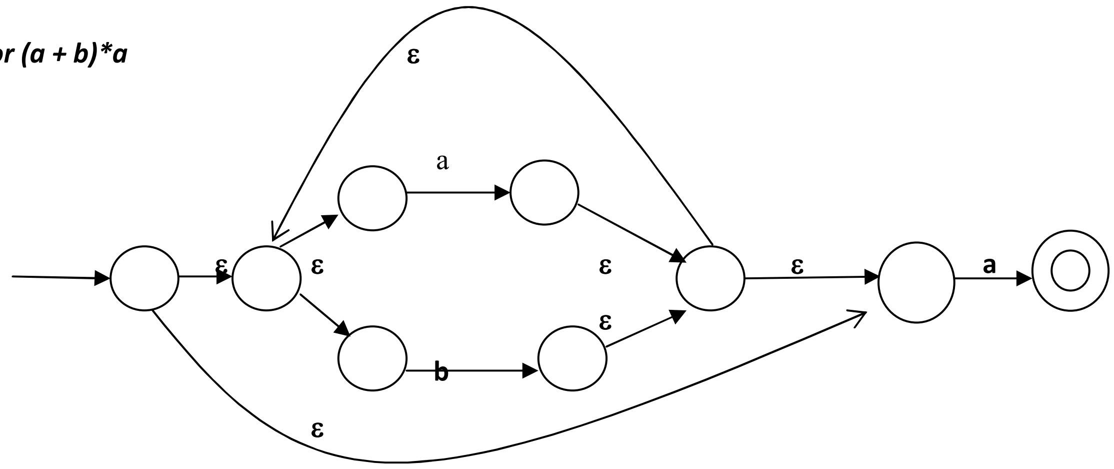
For a + b



For (a + b)*



For (a + b)*a



RE to DFA

Important States

The state s in ε -NFA is an important state if it has no null transition. In optimal state machine all states are important states

Augmented Regular Expression

ε -NFA created from RE has exactly one accepting state and accepting state is not important state since there is no transition so by adding special symbol # on the RE at the rightmost position, we can make the accepting state as an important state that has transition on #. Now the accepting state is there in optimal state machine. The RE (r)# is called augmented RE.

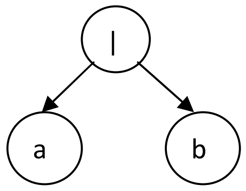
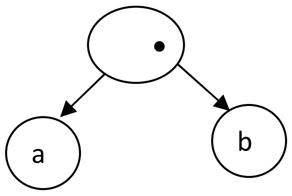
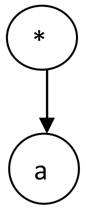
Procedure

1. Convert the given expression to augmented expression by concatenating it with "#" i.e (r) → (r)#.
2. Construct the syntax tree of this augmented regular expression. In this tree, all the operators will be inner nodes and all the alphabet, symbols including "#" will be leaves.
3. Numbered each leaves.

4. Traverse tree to construct *nullable*, *firstpos*, *lastpos* and *followpos*.
5. Construct the DFA from so obtained *followpos*.

- **Some Definitions**
  - **nullable(n):** If the subtree rooted at ***n*** can have valid string as ε, then nullable(n) is true, false otherwise.
  - **firstpos(n):** The set of all the positions that can be the first symbol of the substring rooted at ***n***.
  - **lastpos(n):** The set of all the positions that can be the last symbol of the substring rooted at ***n***.
  - **followpos(i):** The set of all positions that can follow ***i*** for valid string of regular expression. We calculate follwopos, we need above three functions.

Rule to evaluate *nullable*, *firstpos* and *lastpos*

Node- n	nullable(n)	firstpos(n)	lastpos(n)
leaf labeled ε	True	{}	{}
non null leaf position i	False	{i}	{i}
	nullable(a) or nullable(b)	firstpos(a) ∪ firstpos(b)	lastpos(a) ∪ lastpos(b)
	nullable(a) and nulllable(b)	if nullable(a) is true then firstpos(a) ∪ firstpos(b) else firstpos(a)	if nullable(b) is true then lastpos(a) ∪ lastpos(b) else lastpos(b)
	True	firstpos(a)	lastpos(a)

Computation of *followpos*

Algorithm

```
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 then
                for each i in lastpos(n) do
                        followpos(i) = followpos(i) $\cup$ firstpos(n)
                end do
        end if
end do
```

Algorithm to create DFA from RE

1. Create a syntax tree of (r)#
2. Evalauate the functions nullable, firstpos, lastpos ad followpos
3. Start state of DFA = S = S_0 = firspos(r), where *r* is the root of the tree, as an unmarked state
4. While there is unmarked state T in the state of DFA

```
Mark T
for each input symbol a in A do
        Let $s_1$, $s_2$, ……, $s_n$ are positions in S and symbols in those positions is a
        S' $\leftarrow$ followpos($s_1$) $\cup$ …… $\cup$ followpos($s_n$)
        Move(S,a) $\leftarrow$ S'
        if (S' is not empty and not in the states of DFA)
                Put S' into states of DFA and unmarked it
        end if
end do
end do
```

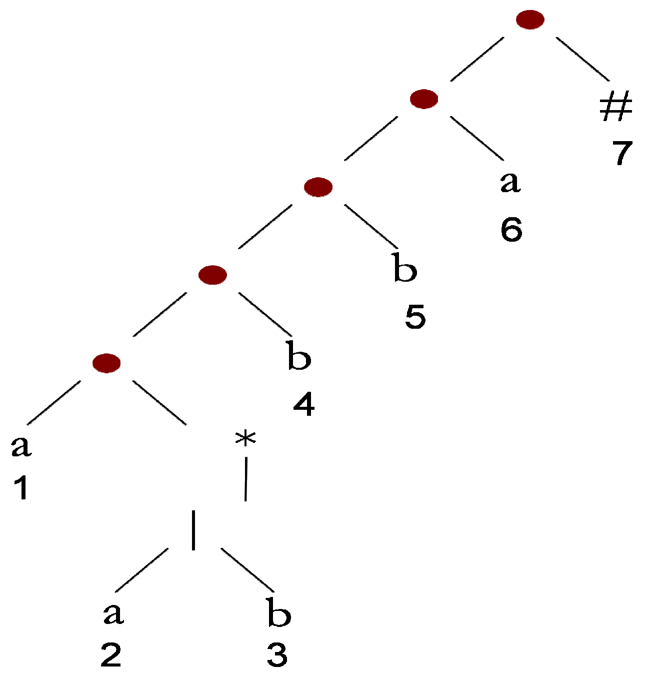
Note:

The start state of resulting DFA is *firstpos*(root)

The final states of DFA are all states containing the position of #.

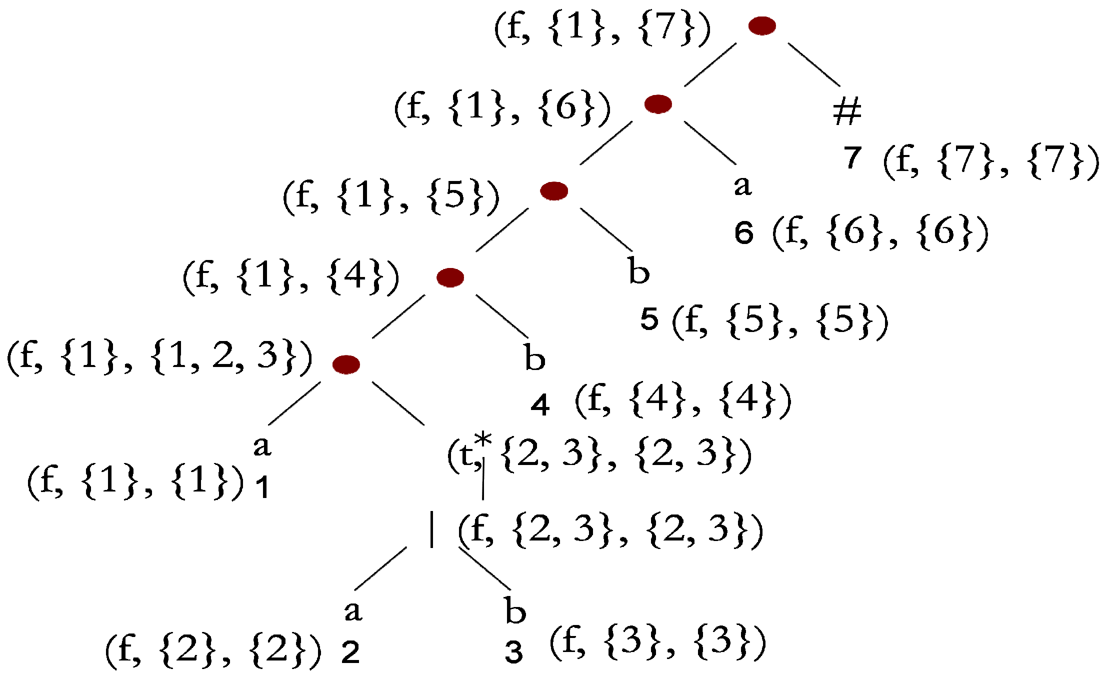
Example

Construct DFA from RE a(a + b)*bba#



Create a syntax tree

Calculate *nullable*, *firstpos* and *lastpos*



Calculate followpos

Using rules we get

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

followpos(2): {2, 3, 4}

followpos(3): {2, 3, 4}

followpos(4): {5}

followpos(5): {6}

followpos(6): {7}

followpos(7): -

Now,

Starting state = S1 = firstpos(root) = {1}

Mark S1

For a : followpos(1) = {2, 3, 4} = S2

For b : ϕ

Mark S2

For a : followpos(2) = {2, 3, 4} = S2 (because among {2, 3, 4}, position of a is 2)

For b : followpos(3) ∪ followpos(4) = {2, 3, 4, 5} = S3 (because among {2, 3, 4}, position of b is

3 and 4)

Mark S3

For a : followpos(2) = {2, 3, 4} = S2

For b : followpos(3) ∪ followpos(4) ∪ followpos(5) = {2, 3, 4, 5, 6} = S4

Mark S4

For a : followpos(2) ∪ followpos(6) = {2, 3, 4, 7} = S5 (since it contains 7 i.e. #, so it is accepting state)

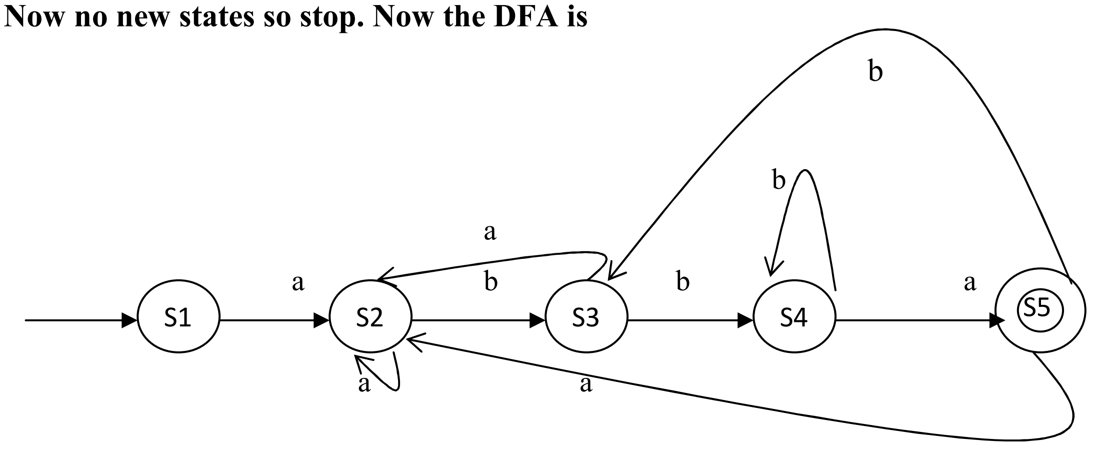
For b : followpos(3) ∪ followpos(4) ∪ followpos(5) = {2, 3, 4, 5, 6} = S4

Mark S5

For a : followpos(2) = {2, 3, 4} = S2

For b : followpos(3) ∪ followpos(4) = {2, 3, 4, 5} = S3

Now no new states so stop. Now the DFA is



State Minimization in DFA

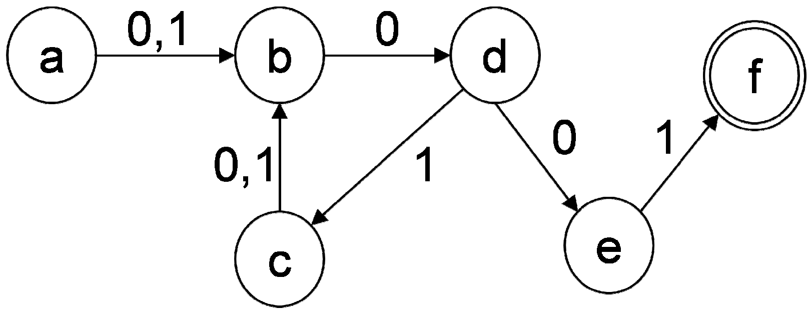
DFA minimization refers to the task of transforming a given deterministic finite automaton (DFA) into an equivalent DFA which has minimum number of states. Two states p and q are called **equivalent** if for all input string s, δ(p, w) is an accepting state iff δ(q, w) is an accepting state., otherwise **distinguishable** states. We say that, string w distinguishes state s from state t if, by starting with DFA M in state s and feeding it input w, we end up in an accepting state, but starting in state t and feeding it with same input w, we end up in a non accepting state, or vice – versa. It finds the states that can be distinguished by some input string. Each group of states that cannot be distinguished is then merged into a single state.

Procedure

1. So partition the set of states into two partition a) set of accepting states and b) set of nonaccepting states.
2. Split the partition on the basis of distinguishable states and put equivalent states in a group
3. To split we process the transition from the states in a group with all input symbols. If the transition on any input from the states in a group is on different group of states then they are distinguishable so remove those states from the current partition and create groups.
4. Process until all the partition contains equivalent states only or have single state.

Example

Optimize the DFA

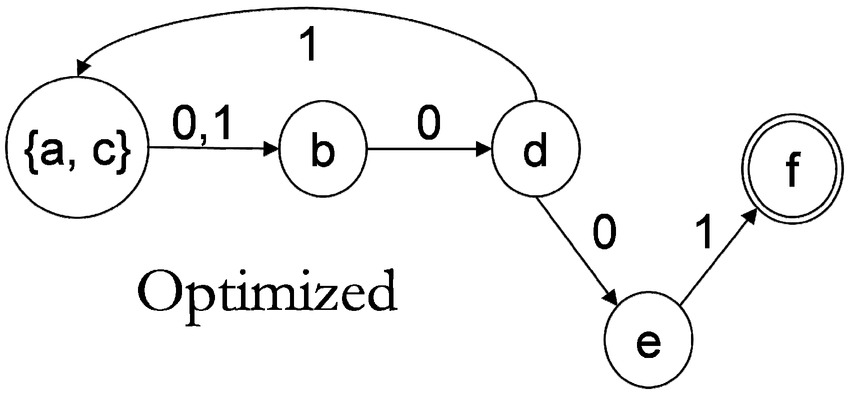


Partition 1: {{a, b, c, d, e}, {f}} with input 0; a → b and b → d, c → b, d → e all transition in same group. with input 1; e → f (different group) so e is distinguishable from others.

Partition 2: {{a, b, c, d}, {e}, {f}} with input 0; d → e (different group).

Partition 3: {{a, b, c}, {d}, {e}, {f}} with input 0; b → d (Watch!!!)

Partition 4: {{a, c}, {b}, {d}, {e}, {f}} with both 0 and 1 a, c → b so no split is possible here, a and c are equivalent.



Space Time Tradeoffs: NFA Vs DFA

- Given the RE r and the input string s to determine whether s is in L(r) we can either construct NFA and test or we can construct DFA and test for s after NFA is constructed from r.
- ε- NFA (for NFA only constant time differs)
  - Space complexity: O(|r|) (at most twice the number of symbols and operators in r).

        - Time complexity: O(|r|*|s|) (test s, there may be O(|s|) test for each possible transition).
- DFA
    - Space complexity: $O(2^{|r|})$ (ε- NFA construction and then subset construction).
    - Time complexity: O(|s|) (single transition so linear test for s).

If we can create DFA from RE by avoiding transition table, then we can improve the performance

Exercises

1. Given alphabet A = {0, 1}, write the regular expression for the following
   a. Strings that can either have sub strings 001 or 100
   b. String where no two 1s occurs consecutively
   c. String which have an odd numbers of 0s
   d. String which have an odd number of 0s and even number of 1s
   e. String that have at most 2 0s
   f. String that have at least 3 1s
   g. String that have at most two 0s and at least three 1s
2. Write regular definition for specifying integer number, floating number, integer array declaration in C.
3. Convert the following regular expression first into NFA and DFA.
   a. 0 + (1 + 0)*00
   b. zero → 0
      one → 1
      bit → zero + one
      bits → bit*
4. Write an algorithm for computing *ε-closure(s)* of any state *s* in NFA.
5. Converse the following RE to DFA
   a. (a + b)*a
   b. (a + ε) b c *
6. Describe the languages denoted by the following RE
   a. 0 (0 + 1) * 0
   b. ((ε + 0) 1 *) *
   c. (0 + 1)*0(0 + 1)(0 + 1)
   d. 0*10*10*10*
   e. (00 + 11)* ((01 + 10) (00 + 11)* (01 + 10) (00 + 11)*)*
7. Construct NFA from following RE and trace the algorithm for ababbab
   a. (a + b)*
   b. (a* + b*)*
   c. ((ε + a) b*)*)

    d. (a + b)* abb(a + b)*

8. Show that following RE are equivalent by showing their minimum state DFA
    a. (a + b)*
    b. (a* + b*)*
    c. ((ε + a)b*)*

Bibliography

- Aho, R. Sethi, J. Ullman, Compilers: Principles, Techniques, and Tools. 2008
- Notes from CDCSIT, TU of Samujjwal Bhandari and Bishnu Gautam
- Manuals of Srinath Srinivasa, Indian Institute of Information Technology, Bangalore