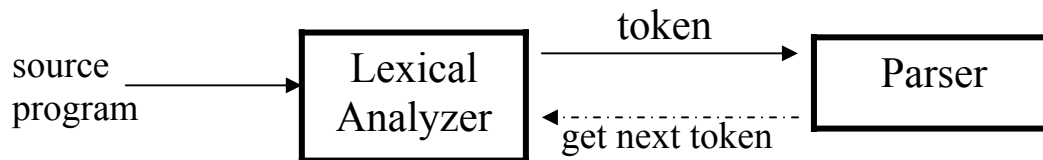


Lexical Analyzer

- **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.



Token

- Token represents a set of strings described by a pattern.
 - Identifier represents a set of strings which start with a letter continues with letters and digits
 - The actual string (newval) is called as *lexeme*.
 - Tokens: identifier, number, addop, delimiter, ...
- Since a token can represent more than one lexeme, additional information should be held for that specific 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:
 - <id,attr> where attr is pointer to the symbol table
 - <assgop,_> no attribute is needed (if there is only one assignment operator)
 - <num,val> where val is the actual value of the number.
- Token type and its attribute uniquely identify a lexeme.
- **Regular expressions** are widely used to specify patterns.

Terminology of Languages

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

$$s \varepsilon = s \quad \varepsilon s = s$$

$$- s^n = s s s \dots s \text{ (n times)} \quad s^0 = \varepsilon$$

Operations 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:
 - $L^0 = \{\varepsilon\} \quad L^1 = L \quad L^2 = LL$
- Kleene Closure
 - $L^* = \bigcup_{i=0}^{\infty} L^i$

Example

- $L_1 = \{a,b,c,d\} \quad L_2 = \{1,2\}$
- $L_1 L_2 = \{a1,a2,b1,b2,c1,c2,d1,d2\}$
- $L_1 \cup L_2 = \{a,b,c,d,1,2\}$
- $L_1^3 = \text{all strings with length three (using a,b,c,d)}$
- $L_1^* = \text{all strings using letters a,b,c,d and empty string}$

Regular Expressions

- We use regular expressions to describe tokens of a programming language.
- A regular expression is built up of simpler regular expressions (using defining rules)
- Each regular expression denotes a language.
- A language denoted by a regular expression is called as a **regular set**.

Regular Expressions (Rules)

Regular expressions over alphabet Σ

<u>Reg. Expr</u>	<u>Language it denotes</u>
ε	$\{\varepsilon\}$
$a \in \Sigma$	$\{a\}$
$(r_1) \mid (r_2)$	$L(r_1) \cup L(r_2)$
$(r_1) (r_2)$	$L(r_1) L(r_2)$
$(r)^*$	$(L(r))^*$
(r)	$L(r)$

- We may remove parentheses by using precedence rules.
 - * highest

- concatenation next
- | lowest
- $ab^*|c$ means $(a(b^*))|(c)$
- Ex:
 - $\Sigma = \{0,1\}$
 - $0|1 \Rightarrow \{0,1\}$
 - $(0|1)(0|1) \Rightarrow \{00,01,10,11\}$
 - $0^* \Rightarrow \{\epsilon, 0, 00, 000, 0000, \dots\}$
 - $(0|1)^* \Rightarrow$ all strings with 0 and 1, including the empty string

Regular Definitions

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

$$d_1 \rightarrow r_1$$

$$d_2 \rightarrow r_2$$

$$d_n \rightarrow r_n$$
 where d_i is a distinct name and r_i is a regular expression over symbols in $\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$
- Ex: Identifiers in C
 - letter $\rightarrow A | B | \dots | Z | a | b | \dots | z$
 - digit $\rightarrow 0 | 1 | \dots | 9$
 - id $\rightarrow \text{letter} (\text{letter} | \text{digit})^*$
 - If we try to write the regular expression representing identifiers without using regular definitions, that regular expression will be complex.

$$(A|\dots|Z|a|\dots|z) ((A|\dots|Z|a|\dots|z) | (0|\dots|9))^*$$
- Ex: Unsigned numbers in C
 - digit $\rightarrow 0 | 1 | \dots | 9$
 - digits $\rightarrow \text{digit} . \text{digit}^*$
 - opt-fraction $\rightarrow (. \text{digits}) | \epsilon$
 - opt-exponent $\rightarrow (E (+|-) | \epsilon \text{ digits}) | \epsilon$
 - unsigned-num $\rightarrow \text{digits} \text{opt-fraction} \text{opt-exponent}$

Finite Automata

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

- lexical analyzer.
- Both deterministic and non-deterministic finite automaton recognize regular sets.
 - 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.
 - 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)

- A non-deterministic finite automaton (NFA) is a mathematical model that consists of:
 - S - a set of states
 - Σ - a set of input symbols (alphabet)
 - move – a transition function move to map state-symbol pairs to sets of states.
 - s_0 - a start (initial) state
 - F – a set of accepting states (final states)
- ϵ - transitions are allowed in NFAs. In other words, we can move from one state to another one without consuming any symbol.
- 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 .

Deterministic Finite Automaton (DFA)

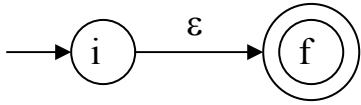
A Deterministic Finite Automaton (DFA) is a special form of a NFA.

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

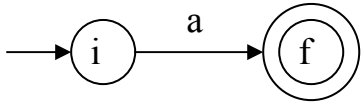
Converting a Regular Expression into A NFA (Thomson's Construction)

- 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.

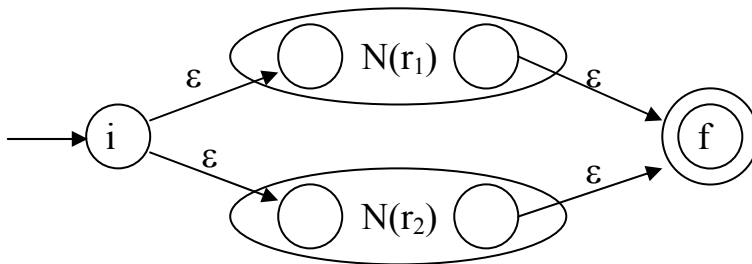
→ To recognize an empty string ϵ



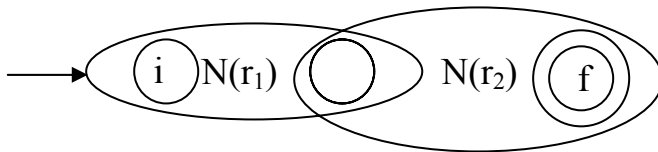
→ To recognize a symbol a in the alphabet Σ



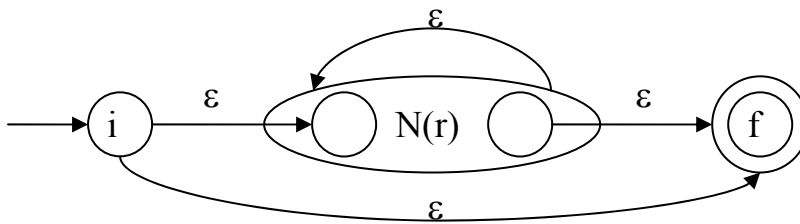
→ If $N(r_1)$ and $N(r_2)$ are NFAs for regular expressions r_1 and r_2



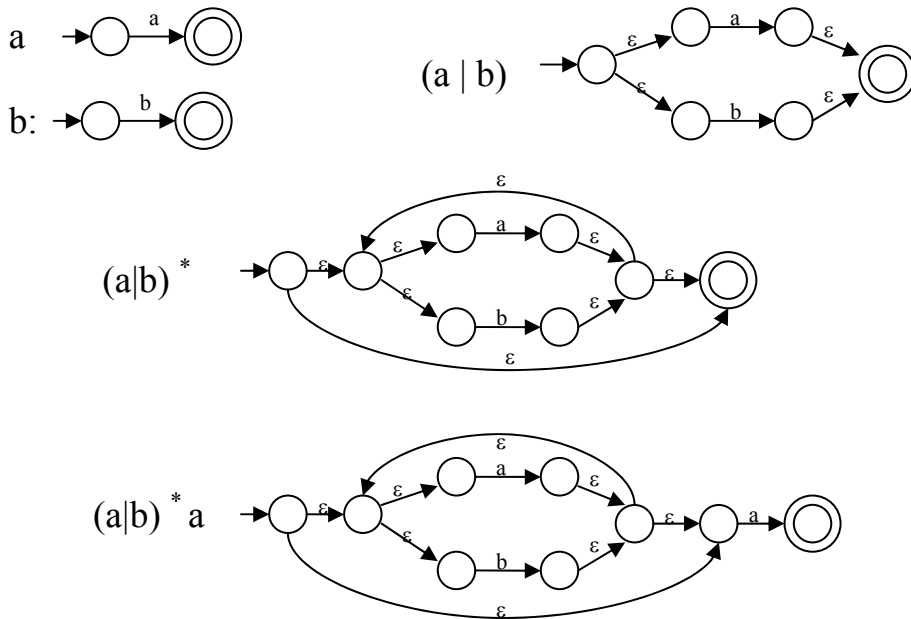
→ For regular expression $r_1 r_2$



→ For regular expression r^*



Thomson's Construction (Example - $(a|b)^* a$)



Converting a NFA into a DFA (subset construction)

```

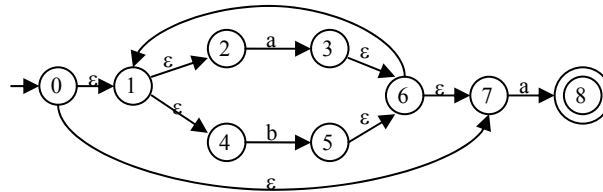
put  $\epsilon$ -closure( $\{s_0\}$ ) as an unmarked state into the set of DFA (DS)
while (there is one unmarked  $S_1$  in DS) do
  begin
    mark  $S_1$ 
    for each input symbol a do
      begin
         $S_2 \leftarrow \epsilon$ -closure(move( $S_1, a$ ))
        if ( $S_2$  is not in DS) then
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

Note:

ϵ -closure($\{s_0\}$) : is the set of all states can be accessible from s_0 by ϵ -transition.
 move(s, a) : set of states to which there is a transition on **a** from a state **s** in S_1

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

Converting a NFA into a DFA (Example)



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

↓ mark S_0

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

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

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

↓ mark S_1

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

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

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

↓ mark S_2

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

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

transfunc[S_2,a] ← S_1 transfunc[S_2,b] ← 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 $S_1 = \{1,2,3,4,6,7,8\}$

