Stanley Abhadiomhen (2)
stanley.abhadiomhen@unn.edu.ng
Department of Computer Science
University Of Nigeria

# COS 331 2017/18 –Week 3 – Functional Relationship amongst Lexical, Syntax Analysis and Semantics Analysis

## 1. LEXiCAL ANALYSIS

### 1.1 OVERVIEW OF LEXICAL ANALYSIS

- To identify the tokens we need some method of describing the possible tokens that can appear in the input stream. For this purpose we introduce regular expression, a notation that can be used to describe essentially all the tokens of programming language.
- Secondly, having decided what the tokens are, we need some mechanism to recognize these in the input stream. This is done by the token recognizers, which are designed using transition diagrams and finite automata.

### 1.2 ROLE OF LEXICAL ANALYZER

The LA is the first phase of a compiler. It main task is to read the input character and produce as output a sequence of tokens that the parser uses for syntax analysis. In theory, the work that is done during lexical analysis can be made an integral part of syntax analysis, and in simple systems this is indeed often done. However, there are reasons for keeping the phases separate:

**Efficiency:** A lexer may do the simple parts of the work faster than the more general parser can.

**Modularity:** The syntactical description of the language need not be cluttered with small lexical details such as white-space and comments.

**Tradition:** Languages are often designed with separate lexical and syntactical phases in mind, and the standard documents of such languages typically separate lexical and syntactical elements of the languages.

**The goal** of lexical analysis is to make life easier for the subsequent syntax analysis phase.

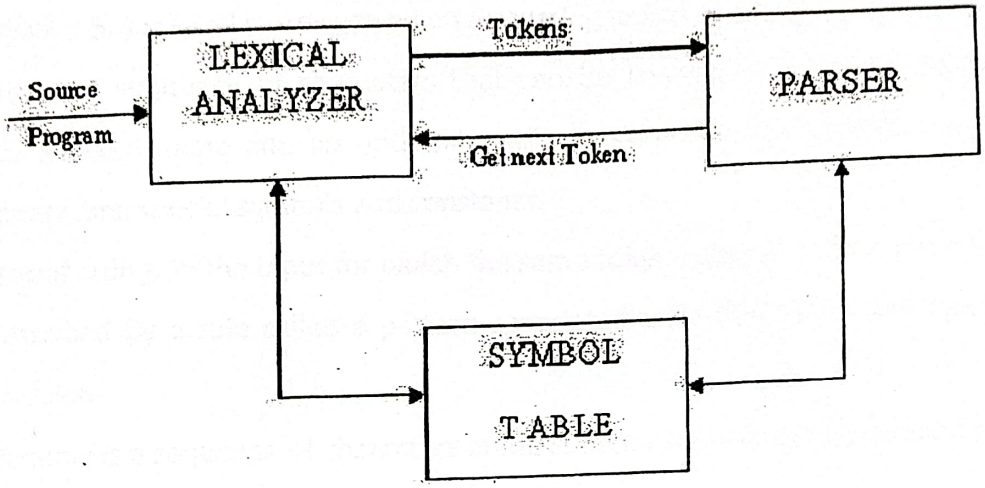



**Figure 1 Role of Lexical analyzer**

When the "get next token" command is received from the parser, the lexical analyzer reads the input character until the next token is identified. The LA return to the parser representation for the token it has found.

LA may carry out other tasks besides identification of lexemes. One such task is stripping out comments and whitespace such as blank, newline and tab. Another task is correlating error messages generated by the compiler with the source program.

**Removal of White Space and Comments**

If white space is eliminated by the lexical analyzer, the parser will never have to consider it. The alternative of modifying the grammar to incorporate white space into the syntax is not nearly as easy to implement.

```
for ( var peek = next input character ) {
if ( peek is a blank or a tab ) do nothing;
else if ( peek is a newline ) line = line+1;
else break;
}
```

(2)

## 1.3 TOKEN, LEXEME, PATTERN:

The three distinct but related words use when discussing lexical analysis:

**Token:** Token is a sequence of characters that can be treated as a single logical entity. It consisting of a token name and an optional attribute value. Typical tokens are, Identifiers, keywords, operators, special symbols and constants

**Pattern:** A set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token and specified using regular expression

**Lexeme:** A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

(2)

| Token | lexeme | pattern |
|---|---|---|
| const | const | const |
| if | if | If |
| relation | <,<=,=,<>,>=,> | < or <= or = or <> or >= or letter followed by letters & digit |
| i | pi | any numeric constant |
| num | 3.14 | any character b/w "and "except" |
| literal | "core" | pattern |

**Figure 2: Example of Token, Lexeme and Pattern**

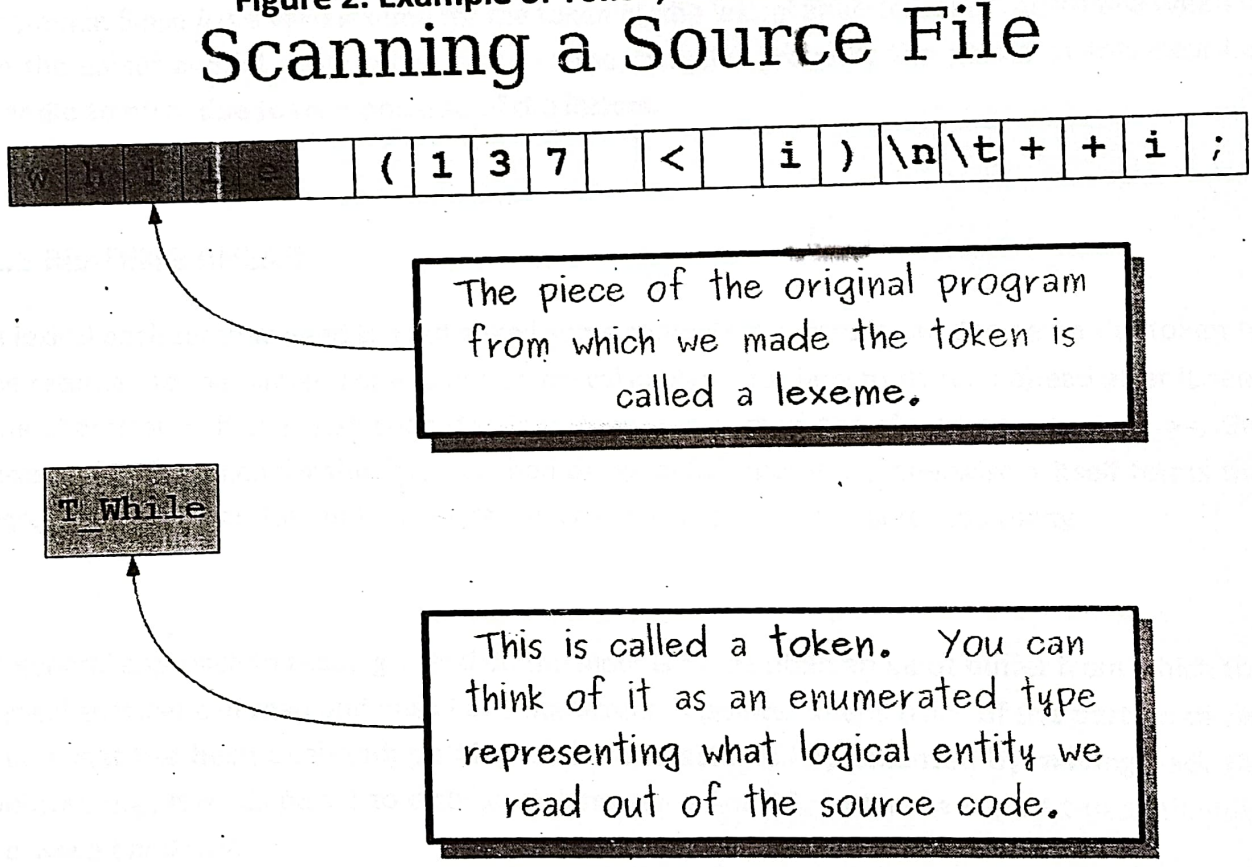



**Figure 3: Scanning Source File**

## 1.4 LEXICAL ERRORS

Lexical errors are the errors thrown by the lexer when unable to continue. Which means that there's no way to recognize a lexeme as a valid token for your lexer i.e. none of the patterns for

tokens matches any prefix of the remaining input. The simplest recovery strategy is "**panic mode" recovery**. We delete successive characters from the remaining input, until the lexical analyzer can find a well-formed token at the beginning of what input is left.

**Error-recovery actions are:**

- Delete one character from the remaining input.
- Insert a missing character in to the remaining input.
- Replace a character by another character.
- Transpose two adjacent characters.

For instance, if the string fi is encountered for the first time in a program e.g. fi(x>y).... a lexical analyzer cannot tell whether fi is a misspelling of the keyword if or an undeclared function identifier. Since fi is a valid lexeme for the token id, the lexical analyzer must return the token id to the parser and let some other phase of the compiler probably the parser in this case i.e. **handle an error due to transposition of the letters.**

## 1.5 READING AHEAD

A lexical analyzer may need to read ahead some characters before it can decide on the token to be returned to the parser. For example, a lexical analyzer for Java must read ahead after it sees the character >. If the next character is =, then > is part of the character sequence >=, the lexeme for the token for the "greater than or equal to" operator. Otherwise > itself forms the "greater than" operator, and the lexical analyzer has read one character too many.

A general approach to reading ahead on the input is to maintain an input buffer from which the lexical analyzer can read and push back characters. **A pointer keeps track of the portion of the input that has been analyzed; pushing back a character is implemented by moving back the pointer.** e.g., it reads past 1 to distinguish between 1 and 10, and it reads past t to distinguish between t and true.

### 1.5.1 INPUT BUFFERING

The LA scans the characters of the source program one at a time to discover tokens. Because of large amount of time can be consumed scanning characters, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input

character.

Buffering techniques:

- Buffer pairs
- Sentinels

Lexical analyzers, whether automatically generated or handwritten, must manage their input efficiently. Of course, input is buffered, so that a large batch of characters is obtained at once; then the lexer can process one character at a time in the buffer. The lexer must check, for each character, whether the end of the buffer is reached. By putting a sentinel (eof)

The lexical analyzer scans the characters of the source program one at a time to discover tokens. Often, however, many characters beyond the next token many have to be examined before the next token itself can be determined. For this and other reasons, it is desirable for the lexical analyzer to read its input from an input buffer. Figure 4 shows a buffer divided into two halves of, say 100 characters each. One pointer marks the beginning of the token being discovered. A **look ahead** pointer scans ahead of the beginning point, until the token is discovered .we view the position of each pointer as being between the character last read and the character next to be read.

In practice each buffering scheme adopts one convention either a pointer is at the symbol last read or the symbol it is ready to read.

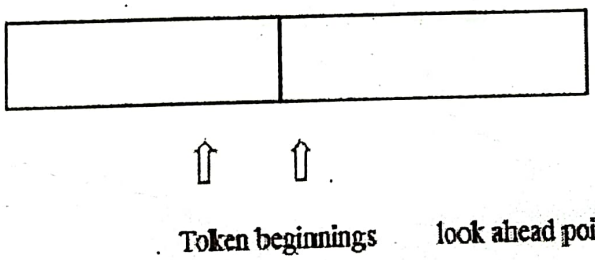



**Figure 4: Equal Half Buffers**

Token beginnings look ahead pointer and the distance which the lookahead pointer may have to travel past the actual token may be large. For example, in a PL/I program we may see: DECALRE (ARG1, ARG2... ARG n) without knowing whether DECLARE is a keyword or an array name until we see the character that follows the right parenthesis. In either case, the token itself ends at the second E. If the lookahead pointer travels beyond the buffer half in which it began, the other half must be loaded with the next characters from the source file. Since the

buffer shown in above figure is of limited size there is an implied constraint on how much lookahead can be used before the next token is discovered. In the above example, if the lookahead traveled to the left half and all the way through the left half to the middle, we could not reload the right half, because we would lose characters that had not yet been grouped into tokens. While we can make the buffer larger if we chose or use another buffering scheme, we cannot ignore the fact that overhead is limited.

The sentinel is a special character that cannot be part of the source program, and a natural choice is the character eof. Any eof that appears other than at the end of a buffer means that the input is at an end.

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

Lookahead code with sentinels

## 1.6 REGULAR EXPRESSIONS

(7)

- Is a regular expression denoting {€}, that is, the language containing only the empty string.
- For each 'a' in Σ, is a regular expression denoting {a}, the language with only one string consisting of the single symbol 'a'.
- If R and S are regular expressions, then

| |
|---|
| (R) \| (S) means L(r) U L(s) |
| R.S means L(r).L(s) |
| R* denotes L(r*) |

## 1.7 REGULAR DEFINITIONS

For notational convenience, we may wish to give names to regular expressions and to define regular expressions using these names as if they were symbols.

Identifiers are the set or string of letters and digits beginning with a letter. The following regular definition provides a precise specification for this class of string.

Using Regular expression operator precedence

| |
|---|
| (R) |
| R* |
| R1R2 |
| R1 \| R2 |

| Token Name | Pattern |
|---|---|
| If | If |
| Id | [a-z][a-z0-9]* |
| ws no token, just white space | ("--"[a-z]*"\n")\|(" "\|"\n"\|"\t")+ |
| Else | Else |
| Relop | < \| > \|<= \| >= \| = = \| < > |
| Lparen | ( |

| | |
|---|---|
| Rparen | ) |

**Regular expressions for some tokens**

**Recognition of tokens:**

We learn how to express pattern using regular expressions. Now, we must study how to take the patterns for all the needed tokens and build a piece of code that examines the input string and finds a prefix that is a lexeme matching one of the patterns.

```
Stmt →if (expr) stmt
     | If (expr) stmt else stmt
     | ϵ
Expr →term relop term
     | term
Term →id
     |number
```

A grammar for branching statements

For relop, we use the comparison operations of languages like Pascal or SQL where = is "equals" and < > is "not equals" because it presents an interesting structure of lexemes. The terminal of grammar, which are if, lparen, rparen, else, relop, id and numbers are the names of tokens as far as the lexical analyzer is concerned, the patterns for the tokens are described using regular definitions.

In addition, we assign the lexical analyzer the job stripping out white space, by recognizing the "token" we defined by: WS → ("--"[a-z]*"\n")|(" "|"\n"|"\t")+ for blank|tab|newline

Here, blank, tab and newline are abstract symbols that we use to express the ASCII characters of the same names. Token **ws** is different from the other tokens in that ,when we recognize it,

we do not return it to parser ,but rather restart the lexical analysis from the character that follows the white space . It is the following token that gets returned to the parser.

| Lexeme | Token Name | Attribute Value |
|---|---|---|
| Any WS | - | - |
| if | If | - |
| else | Else | - |
| Any id | Id | Pointer to table entry |
| Any number | number | Pointer to table entry |
| < | relop | LT |
| <= | relop | LE |
| == | relop | EQ |
| <> | relop | NE |
| ( | lparen | |
| ) | rparen | |

## 1.8 TRANSITION DIAGRAM

As an intermediate step in the construction of a lexical analyzer, we first convert regular-expression patterns into stylized flowcharts, called **Transition diagrams.**

Transition Diagram has a collection of nodes or circles, called states. Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns. Edges are directed from one state of the transition diagram to another. Each edge is labeled by a symbol or set of symbols.

If we are in one state s, and the next input symbol is a, we look for an edge out of state s labeled by a. if we find such an edge ,we advance the forward pointer and enter the state of the transition diagram to which that edge leads.

**Some important conventions about transition diagrams are**

- Certain states are said to be accepting or final .These states indicates that a lexeme has been found, although the actual lexeme may not consist of all positions b/w the lexeme Begin and forward pointers we always indicate an accepting state by a double circle.
- In addition, if it is necessary to return the forward pointer one position, then we shall additionally place a * near that accepting state.
- One state is designed the start state ,or initial state ., it is indicated by an edge labeled "start" entering from nowhere .the transition diagram always begins in the state before any input symbols have been used.

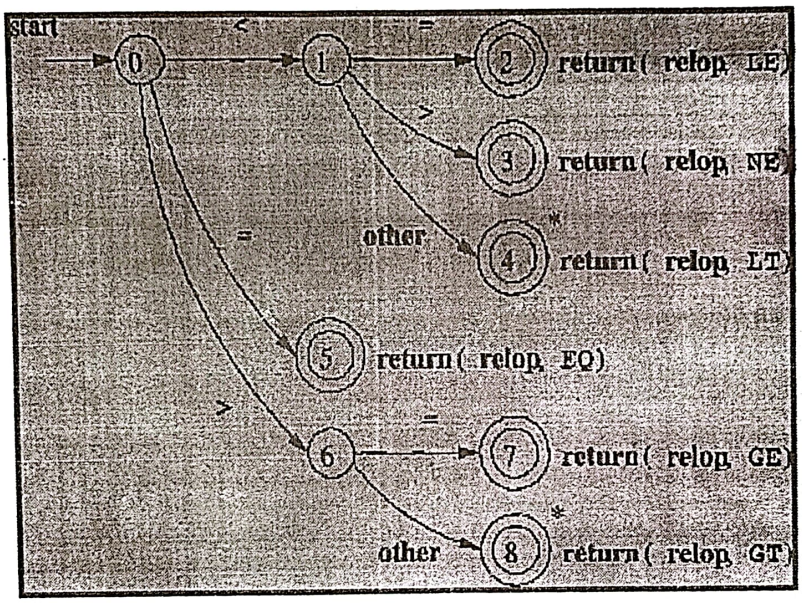



**Figure 5: Transition diagram of Relational operators**

Example: Figure 5 is a transition diagram that recognizes the lexemes matching the token relop. We begin in **state 0,** the start state. If we see < as the first input symbol, then among the lexemes that match the pattern for relop we can only be looking at <, <>, or <=. We therefore go to **state 1,** and look at the next character. If it is =, then we recognize lexeme <=, enter **state 2,** and return the token relop with attribute LE, the symbolic constant representing this particular comparison operator. If in **state 1** the next character is >, then instead we have lexeme <>, and enter **state 3** to return an indication that the not-equals operator has been

found. On any other character, the lexeme is <, and we enter state 4 to return that information. Note, however, that state 4 has a * to indicate that we must retract the input one position. On the other hand, if in state 0 the first character we see is =, then this one character must be the lexeme. We immediately return that fact from state 5.

The remaining possibility is that the first character is >. Then, we must enter state 6 and decide, on the basis of the next character, whether the lexeme is >= (if we next see the = sign), or just > (on any other character). Note that if, in state 0, we see any character besides <, =, or >, we cannot possibly be seeing a relop lexeme, so this transition diagram will not be used.

Position in a transition diagram, are drawn as circles and are called as states.

There are several ways that a collection of transition diagrams can be used to build a lexical analyzer. Regardless of the overall strategy, each state is represented by a piece of code. Example using Switch statement

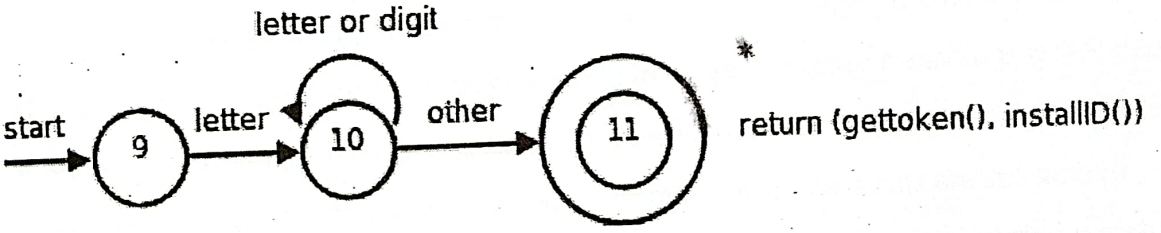



**Figure 6: Transition diagram of Identifier**

The above TD for an identifier, defined to be a letter followed by any number of letters or digits. A sequence of transition diagram can be converted into program to look for the tokens specified by the diagrams. Each state gets a segment of code.

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

Sketch of implementation of `relop` transition diagram

For Example, If the next input character is not one that can begin a comparison operator, then a function f a i l () is called to reset the forward pointer to lexemeBegin, in order to allow another transition diagram to be applied the true beginning of the unprocessed input.

Alternatively, if there is no other transition diagram that remains unused, fail() could initiate an error-correction phase that will try to repair the input and find a lexeme, as discussed in Section 1.4.

We could fit each transition diagrams into entire lexical analyzer.

- We could arrange for the transition diagrams for each token to be tried sequentially. If function f a i l () is called it should reset the forward pointer to lexemeBegin, in order to allow another transition diagram to be applied the true beginning of the unprocessed input.
- We could run the various transition diagrams "in parallel," feeding the next input character to all of them and allowing each one to make whatever transitions it required. If we use this strategy, we must be careful to resolve the case where one diagram finds a lexeme that matches its pattern, while one or more other diagrams are still able to process input. The normal strategy is to take the longest prefix of the input that matches any pattern. That rule allows us to prefer identifier thenext to keyword then, or the operator -> to -, for example.
- The preferred approach is to combine all the transition diagrams into one. Thus, we could simply combine all the start state for each transition diagram into one start state, leaving other transitions intact. However, in general, the problem of combining transition diagrams for several tokens is more complex.

## Recognition of Reserved Words and Identifiers

Recognizing keywords and identifiers is difficult. Usually, keywords like if or else are reserved so, they are not identifiers even though they look like identifiers. Thus, although we typically use a transition diagram like that of Fig. 6 to search for identifier lexemes, this diagram will also recognize the keywords if and else.

There are two ways that we can handle reserved words that look like identifiers:

- Install the reserved words in the symbol table initially. A field of the symbol-table entry indicates that these strings are never ordinary identifiers, and tells which token they represent. We have supposed that this method is in use in Fig 6. When we find an identifier, a call to installID places it in the symbol table if it is not already there and returns a pointer to the symbol-table entry for the lexeme found. Of course, any identifier not in the symbol table during lexical analysis cannot be a reserved word, so its token is id. The function getToken examines the symbol table entry for the lexeme found, and returns whatever token name the symbol table says this lexeme represents — either id or one of the keyword tokens that was initially installed in the table.
- Create separate transition diagrams for each keyword; an example for the keyword if is shown in Fig 6. Note that such a transition diagram consists of states representing the situation after each successive letter of the keyword is seen, followed by a test for a "nonletter-or-digit," i.e., any character that cannot be the continuation of an identifier. It is necessary to check that the identifier has ended, or else we would return token t h e n in situations where the correct token was id, with a lexeme like ifnextvalue that has then as a proper prefix. If we adopt this approach, then we must prioritize the tokens so that the reserved-word tokens are recognized in preference to id, when the lexeme

matches both patterns. There are two important disambiguation rules used by Le and other similar lexical-analyzer generators:

**Longest match:** The longest initial substring of the input that can match any regular expression is taken as the next token.

**Rule priority:** For a particular longest initial substring, the first regular expression that can match determines its token-type. This means that the order of

## 1.9. FINITE AUTOMATON

- 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 recognizes regular sets.
- Which one?
  - Deterministic – faster recognizer, but it may take more space
  - Non-deterministic – slower, but it may take less space
  - Deterministic automatons are widely used lexical analyzers.
- First, we define regular expressions for tokens; then we convert them into a DFA to get a lexical analyzer for our tokens.

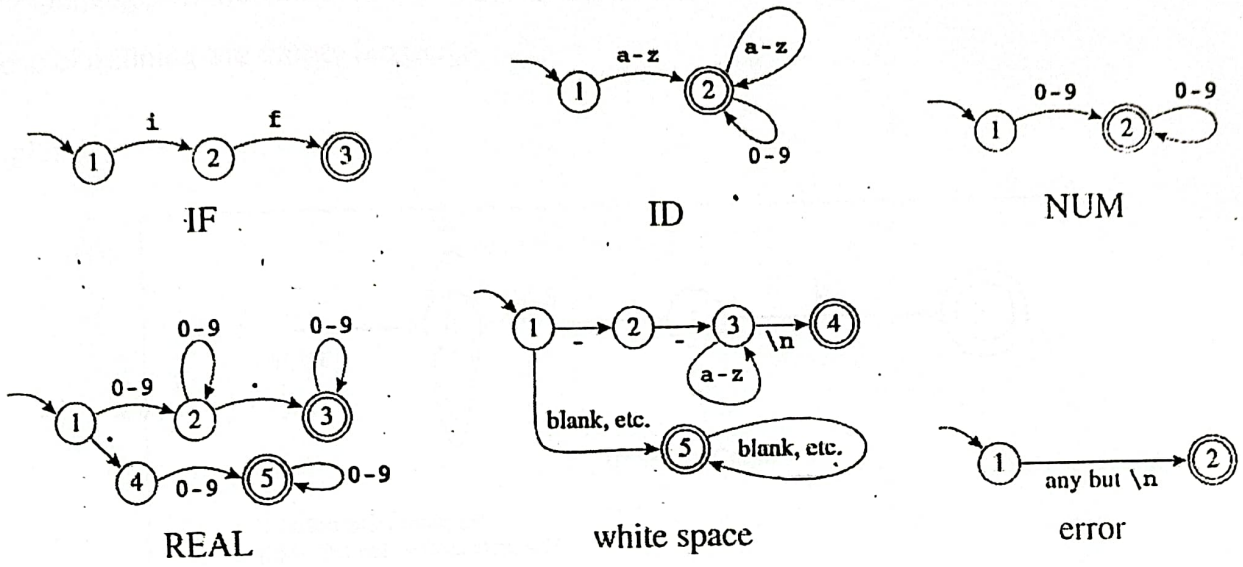



Figure 7: Finite automata for lexical tokens.

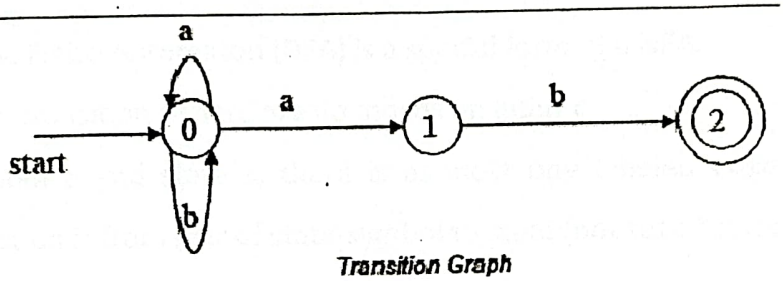
### 1.9.1 Non-Deterministic Finite Automaton (NFA)

- 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.
  - s0 - 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.

We can represent either an NFA or DFA by a transition graph, where the nodes are states and the labeled edges represent the transition function. There is an edge labeled **a** from state **s** to state **t** if and only if **t** is one of the next states for state **s** and input **a**. This graph is very much like a transition diagram, except regular expressions cannot describe the empty language, since we would never want to use this pattern in practice. However, finite automata can define the

empty language. In the theory, 0 is treated as an additional regular expression for the sole purpose of defining the empty language.

Example:




*Transition Graph*

0 is the start state s0
{2} is the set of final states F
$\Sigma$ = {a,b}
S = {0,1,2}

Transition Function:

| | a | b |
|---|---|---|
| 0 | {0,1} | {0} |
| 1 | {} | {2} |
| 2 | {} | {} |

The language recognized by this NFA is (a|b)*ab

On the first transition, this machine must choose which way to go. It is required to accept the string if there is any choice of paths that will lead to acceptance. Thus, it must "guess," and must always guess correctly.

We can also represent an NFA by a transition table, whose rows correspond to states, and whose columns correspond to the input symbols

Following our convention for transition diagrams, the double circle around state 2 indicates that this state is accepting. Notice that the only ways to get from the start state 0 to the accepting state is to follow some path that stays in state 0 for a while and goes to states 1 and 2

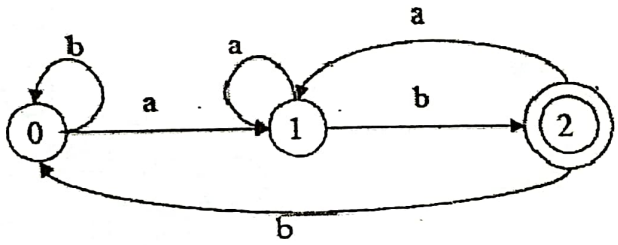
by reading **ab** from the input. Thus, the only strings getting to the accepting state are those that end in **ab**.

## 1.9.2 Deterministic Finite Automaton (DFA)

- A Deterministic Finite Automaton (DFA) is a special form of a NFA.
- No state has ε- transition : There are no moves on input ε
- 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)

Example:

The DFA to recognize the language (a|b)* ab is as follows.




*Transition Graph*

0 is the start state s0
{2} is the set of final states F
Σ = {a,b}
S = {0,1,2}

Transition Function:

| | a | b |
|---|---|---|
| 0 | 1 | 0 |
| 1 | 1 | 2 |
| 2 | 1 | 0 |

***Note that the entries in this function are single value and not set of values (unlike NFA).***

A DFA accepts or rejects a string as follows. Starting in the start state, for each character in the input string the automaton follows exactly one edge to get to the next state. The edge must be labeled with the input character.

It is fortunate indeed that every regular expression and every NFA can be converted to a DFA accepting the same language, because it is the DFA that we really implement or simulate when building lexical analyzers.

### 1.9.3 Converting RE to NFA

Nondeterministic automata are a useful notion because it is easy to convert a (static, declarative) regular expression to a (simulatable, quasi-executable) NFA. The conversion algorithm turns each regular expression into an NFA with a tail (start edge) and a head (ending state). For example, the single-symbol regular expression a converts to the NFA

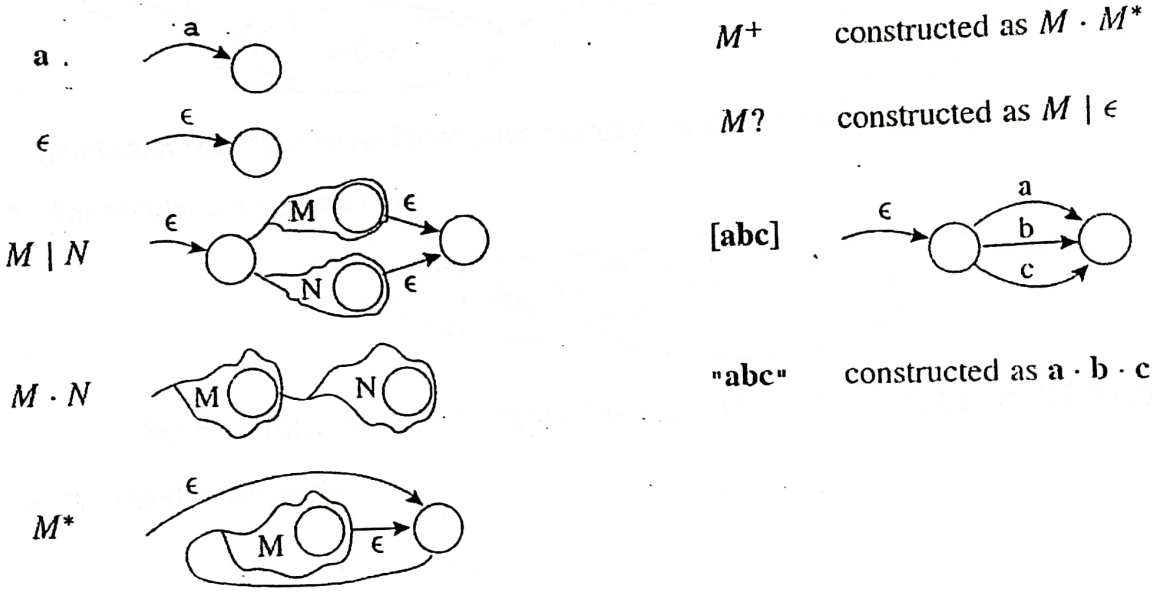
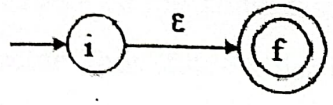


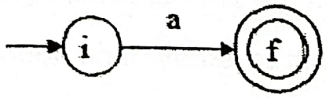

**Figure 8 Rules for translating regular expressions to nondeterministic**

- 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.

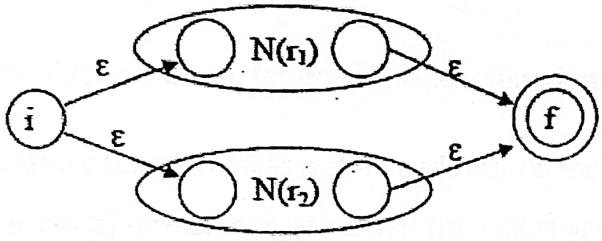
- To recognize an empty string ε:




- To recognize a symbol a in the alphabet Σ:

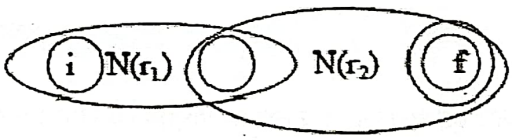



- For regular expression r1 | r2:




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

- For regular expression r1 r2




Here, final state of N(r1) becomes the final state of N(r1r2).

- For regular expression r*

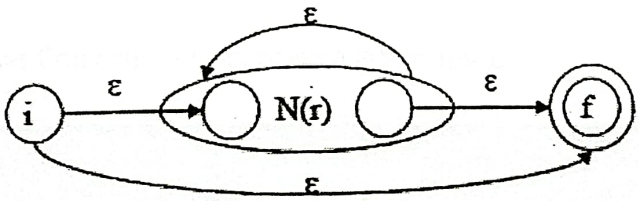



Figure 8 shows the rules for translating regular expressions to nondeterministic automata. We illustrate the algorithm on some of the expressions in Figure 7 – for the tokens IF, ID, NUM, and error. Each expression is translated to an NFA, the "head" state of each NFA is marked final with a different token type, and the tails of all the expressions are joined to a new start node. The result – after some merging of equivalent NFA states – is shown in Figure 9.

- From the point of view of the input, any two states that are connected by an –transition may as well be the same, since we can move from one to the other without consuming any character. Thus states which are connected by an -transition will be represented by the same states in the DFA.
- If it is possible to have multiple transitions based on the same symbol, then we can regard a transition on a symbol as moving from a state to a set of states (ie. the union of all those states reachable by a transition on the current symbol). Thus these states will be combined into a single DFA state.

**To perform this operation, let us define two functions:**

- The -closure function takes a state and returns the set of states reachable from it based on (one or more) -transitions. Note that this will always include the state itself. We should be able to get from a state to any state in its -closure without consuming any input.
- The function move takes a state and a character, and returns the set of states reachable by one transition on this character.

We can generalize both these functions to apply to sets of states by taking the union of the application to individual states.

For Example, if A, B and C are states, move ({A,B,C},`a') = move(A,`a') move(B,`a') move(C,`a').

The Subset Construction Algorithm is a follows:

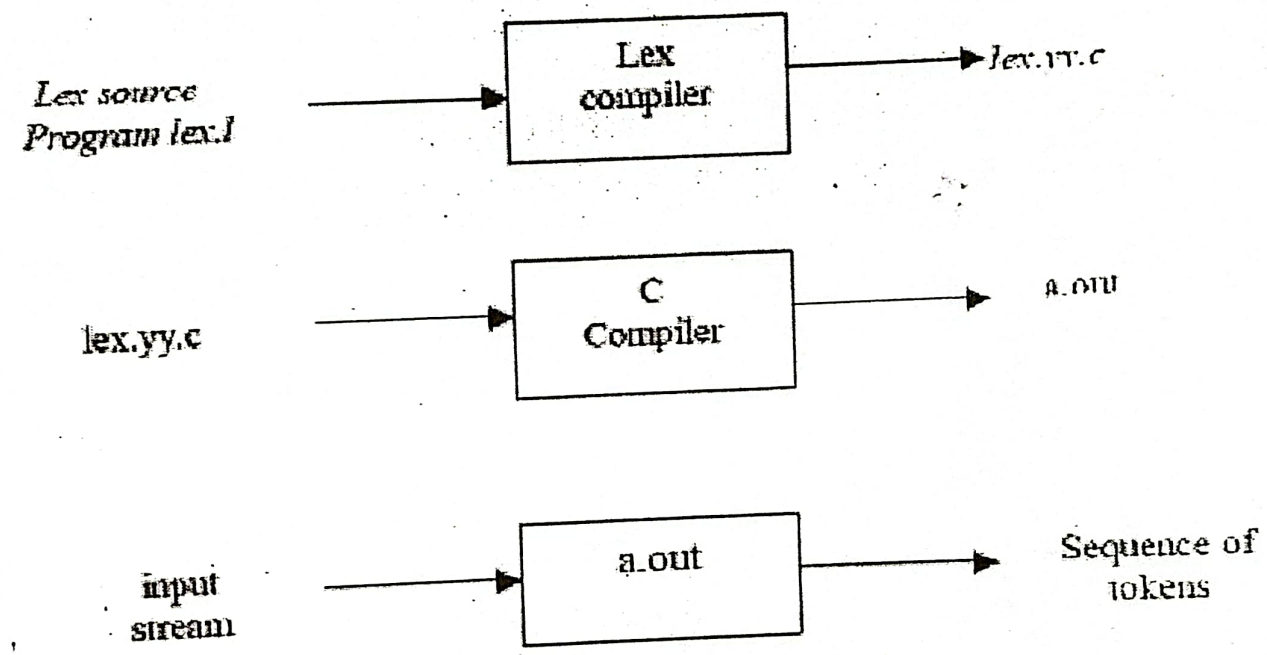
```
put ε-closure({s0}) as an unmarked state into the set of DFA (DS)
while (there is one unmarked S1 in DS) do
    begin
        mark S1
        for each input symbol a do
            begin
                S2 ← ε-closure(move(S1,a))
                if (S2 is not in DS) then
```

```
            add S2 into DS as an unmarked state
            transfunc[S1,a] ← S2
        end
    end
```

- a state S in DS is an accepting state of DFA if a state in S is an accepting state of NFA
- the start state of DFA is ε-closure({s0})

## 2.0 Lexical Analyzer Generator

### 2.1 Lex specifications:




A Lex program (the .l file) consists of three parts:

Declarations

%%

Translation rules

%%

Auxiliary procedures