3.1 Lexical Analysis

Lexical analysis is the first phase of a compiler. Lexical Analysis is also known as Scanner It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these sentences into a series of tokens, by removing any whitespace or comments in the source code. If the lexical analyzer finds a token as invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.

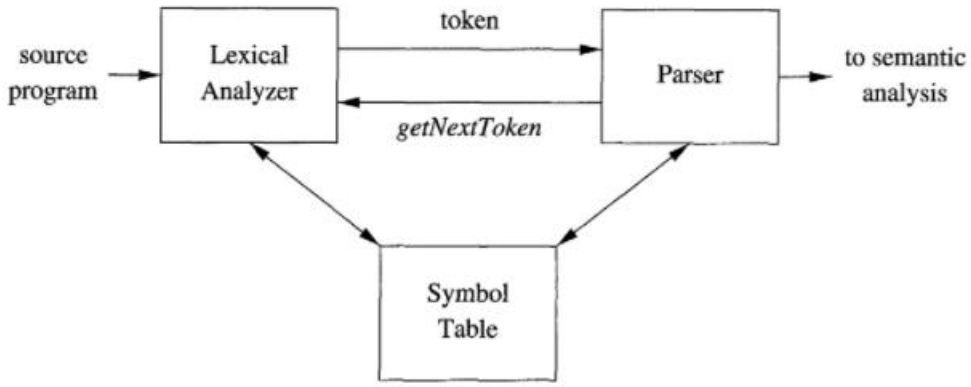

Figure 3.1: Interactions between the lexical analyzer and the parser

These interactions are suggested above in Fig. 3.1. Commonly, the interaction is implemented by having the parser (syntax phase) call the lexical analyzer. The call, suggested by the get Next Token command, causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it the next token, which it returns to the parser.

3.2 Tokens

Lexemes are said to be a sequence of characters (alphanumeric) which is also called as tokens. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions.

In programming language, keywords, constants, identifiers, strings, numbers, operators, and punctuations symbols can be considered as tokens.

Example: int value = 100;

Contains the tokens:

1) int (keyword)

2) value (identifier)

3) = (operator)

4) 100 (constant)

5); (symbol)

- A typical high-level language contains the following symbols :-

Symbols	Purpose
Addition(+), Subtraction(-), Modulo(%), Multiplication (*) and Division(/)	Arithmetic Operator
Comma(,), Semicolon(;), Dot(.), Arrow(->)	Punctuation
= , +=, /=, *=, -=	Assignment
==, !=, <, <=, >, >=	Comparison
#	Preprocessor
&	Location Specifier
&, &&, |, ||, !	Logical
>>, >>>, <<, <<<	Shift Operator

3.3String

A string over an alphabet is a finite sequence of symbols drawn from that alphabet. In language theory, the terms "sentence" and "word" are often used as synonyms for "string." The length of a string s, usually written |s|, is the number of occurrences of symbols in s. For example, banana is a string of length six. The empty string, denoted ε, is the string of length zero.

- If x and y are strings, then the concatenation of x and y, denoted xy, is the string formed by appending y to x. For example, if x = dog and y = house, then xy = doghouse. The empty string is the identity under concatenation; that is, for any string s, ε S = S ε = s.
- In lexical analysis, the most important operations on languages are union, concatenation, and closure, which are defined formally in Fig. 3.6. Union is the familiar operation on sets. The concatenation of languages is all strings formed by taking a string from the first language and a string from the second language, in all possible ways, and concatenating them. The (Kleene) closure

of a language L, denoted L*, is the set of strings you get by concatenating L zero or more times. Note that L^0, the "concatenation of L zero times," is defined to be {ε}, and inductively, $L^{i-1}L = L^i$. Finally, the positive closure, denoted L+, is the same as the Kleene closure, but without the term L^0. That is, ε will not be in L+ unless it is in L itself.

OPERATION	DEFINITION AND NOTATION
Union of L and M	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation of L and M	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$

Figure 3.6: Definitions of operations on languages

Example 3.3: Let L be the set of letters $\{\mathtt{A}, \mathtt{B}, \ldots, \mathtt{Z}, \mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\}$ and let D be the set of digits $\{\mathtt{0}, \mathtt{1}, \ldots \mathtt{9}\}$. We may think of L and D in two, essentially equivalent, ways. One way is that L and D are, respectively, the alphabets of uppercase and lowercase letters and of digits. The second way is that L and D are languages, all of whose strings happen to be of length one. Here are some other languages that can be constructed from languages L and D, using the operators of Fig. 3.6:

1. $L \cup D$ is the set of letters and digits — strictly speaking the language with 62 strings of length one, each of which strings is either one letter or one digit.
2. LD is the set of 520 strings of length two, each consisting of one letter followed by one digit.
3. L^4 is the set of all 4-letter strings.
4. L^* is the set of all strings of letters, including ϵ, the empty string.
5. $L(L \cup D)^*$ is the set of all strings of letters and digits beginning with a letter.
6. D^+ is the set of all strings of one or more digits.

3.4 Regular Expression

The regular expressions are built recursively out of smaller regular expressions, using the rules described below. Each regular expression r denotes a language L(r), which is also defined recursively from the languages denoted by r's subexpressions. Here are the rules that define the regular

expressions over some alphabet Σ and the languages that those expressions denote.

There are two rules that form the basis:

1. ε is a regular expression, and L(ε) is { ε }, that is, the language whose sole member is the empty string.
2. If a is a symbol in **Σ,** then a is a regular expression, and L(a) = {a}, that is, the language with one string, of length one, with *a* in its one position.

INDUCTION: There are four parts to the induction whereby larger regular expressions are built from smaller ones. Suppose r and s are regular expressions denoting languages $L(r)$ and $L(s)$, respectively.

1. $(r)|(s)$ is a regular expression denoting the language $L(r) \cup L(s)$.
2. $(r)(s)$ is a regular expression denoting the language $L(r)L(s)$.
3. $(r)^*$ is a regular expression denoting $\big(L(r)\big)^*$.
4. (r) is a regular expression denoting $L(r)$. This last rule says that we can add additional pairs of parentheses around expressions without changing the language they denote.

Example 3.4: Let $\Sigma = \{a, b\}$.

1. The regular expression **a|b** denotes the language $\{a, b\}$.
2. **(a|b)(a|b)** denotes $\{aa, ab, ba, bb\}$, the language of all strings of length two over the alphabet Σ. Another regular expression for the same language is **aa|ab|ba|bb**.
3. **a*** denotes the language consisting of all strings of zero or more a's, that is, $\{\epsilon, a, aa, aaa, \ldots\}$.
4. **(a|b)*** denotes the set of all strings consisting of zero or more instances of a or b, that is, all strings of a's and b's: $\{\epsilon, a, b, aa, ab, ba, bb, aaa, \ldots\}$. Another regular expression for the same language is **(a*b*)***.
5. **a|a*b** denotes the language $\{a, b, ab, aab, aaab, \ldots\}$, that is, the string a and all strings consisting of zero or more a's and ending in b.

Example 3.5: C identifiers are strings of letters, digits, and underscores. Here is a regular definition for the language of C identifiers. We shall conventionally use italics for the symbols defined in regular definitions.

$$\begin{aligned} \textit{letter\_} &\rightarrow \texttt{A} \mid \texttt{B} \mid \cdots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \mid \texttt{\_} \\ \textit{digit} &\rightarrow \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\ \textit{id} &\rightarrow \textit{letter\_} \; (\; \textit{letter\_} \mid \textit{digit} \;)^* \end{aligned}$$

□

Example 3.6: Unsigned numbers (integer or floating point) are strings such as `5280`, `0.01234`, `6.336E4`, or `1.89E-4`. The regular definition

$$\begin{aligned} \textit{digit} &\rightarrow \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\ \textit{digits} &\rightarrow \textit{digit} \; \textit{digit}^* \\ \textit{optionalFraction} &\rightarrow \texttt{.} \; \textit{digits} \mid \epsilon \\ \textit{optionalExponent} &\rightarrow (\; \texttt{E} \; (\; \texttt{+} \mid \texttt{-} \mid \epsilon \;) \; \textit{digits} \;) \mid \epsilon \\ \textit{number} &\rightarrow \textit{digits} \; \textit{optionalFraction} \; \textit{optionalExponent} \end{aligned}$$

3.4 Finite Automata

Finite automata is a state machine that takes a string of symbols as input and changes its state accordingly. Finite automata is a recognized for regular expressions. When a regular expression string is fed into finite automata, it changes its state for each literal. If the input string is successfully processed and the automata reaches its final state, it is accepted, i.e., the string just fed was said to be a valid token of the language in hand. A recognizer/finite automaton for a language is a program that takes as input a string x and answers „yes“ if x is a sentence of the language L „no“ otherwise.

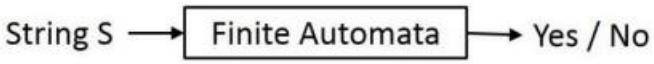

Figure 2 Finite Automaton

Types:

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

(a) *Nondeterministic finite automata* (NFA) have no restrictions on the labels of their edges. A symbol can label several edges out of the same state, and ϵ, the empty string, is a possible label.

(b) *Deterministic finite automata* (DFA) have, for each state, and for each symbol of its input alphabet exactly one edge with that symbol leaving that state.

3.4.1 Nondeterministic Finite Automata

A *nondeterministic finite automaton* (NFA) consists of:

1. A finite set of states S.
2. A set of input symbols Σ, the *input alphabet*. We assume that ϵ, which stands for the empty string, is never a member of Σ.
3. A *transition function* that gives, for each state, and for each symbol in $\Sigma \cup \{\epsilon\}$ a set of *next states*.
4. A state s_0 from S that is distinguished as the *start state* (or *initial state*).
5. A set of states F, a subset of S, that is distinguished as the *accepting states* (or *final states*).

Example 3.14: The transition graph for an NFA recognizing the language of regular expression **(a|b)*abb** is shown in Fig. 3.24. This abstract example, describing all strings of *a*'s and *b*'s ending in the particular string *abb*, will be used throughout this section. It is similar to regular expressions that describe languages of real interest, however. For instance, an expression describing all files whose name ends in `.o` is **any*** `.o`, where **any** stands for any printable character.

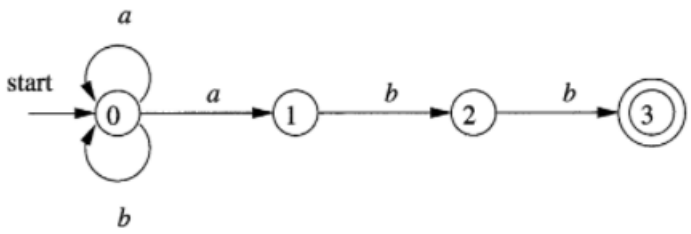

Figure 3.24: A nondeterministic finite automaton

Transition Tables

We can also represent an NFA by a transition table, whose rows correspond to states, and whose columns correspond to the input symbols and ε. The entry for a given state and input is the value of the transition function applied to those arguments. If the transition function has no information about that state-input pair, we put θ in the table for the pair.

Example:

The transition table for the NFA of Fig. 3.24 is shown in Fig. 3.25.

STATE	a	b	ϵ
0	$\{0,1\}$	$\{0\}$	$\emptyset$
1	$\emptyset$	$\{2\}$	$\emptyset$
2	$\emptyset$	$\{3\}$	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$

Figure 3.25: Transition table for the NFA of Fig. 3.24

3.4.2 Deterministic Finite Automata

A deterministic finite automaton (DFA) is a special case of an NFA where:

1. There are no moves on input ε, and

2. For each state s and input symbol a, there is exactly one edge out of s

labeled a.

If we are using a transition table to represent a DFA, then each entry is a single state. we may therefore represent this state without the curly braces that we use to form sets.

Example 3.19: In Fig. 3.28 we see the transition graph of a DFA accepting the language $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}$, the same as that accepted by the NFA of Fig. 3.24. Given the input string *ababb*, this DFA enters the sequence of states $0, 1, 2, 1, 2, 3$ and returns "yes." □

```
s = s0;
c = nextChar();
while ( c != eof ) {
        s = move(s, c);
        c = nextChar();
}
if ( s is in F ) return "yes";
else return "no";
```

Figure 3.27: Simulating a DFA

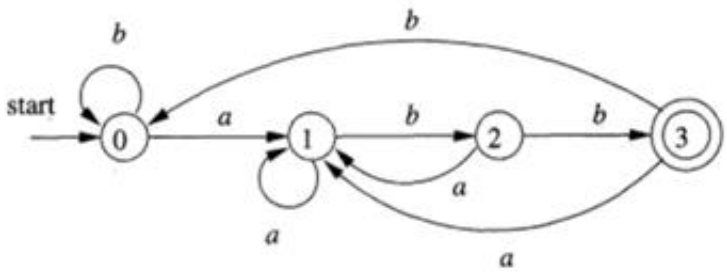

Figure 3.28: DFA accepting $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}$