Computer ScienceCompiler Design
Module 3

Module 3 – Lexical Phase – Introduction

The objective of this module is to get a clear understanding of the lexical phase and to learn about defining patterns using regular expression.

This module discusses the functionalities of the lexical phase of the compiler. This will brief why we need lexical phase, what it does and how to do it efficiently.

3. 1 Functions of the Lexical Analyser

The main job of the lexical phase is scanning and lexical analysis. Scanning is the first part which helps in removal of comments and compaction of consecutive white space characters into single white space. The second part is the lexical analysis which is more complex. This part of the lexical phase helps in producing tokens. The input to this phase is the high level language text file and output will be a sequence of tokens. After this phase, the input text file no longer exists and is split into tokens to be passed on to the next phase of the compiler. In the process of splitting the input into tokens, the lexical analyser does the following functions:

- Identifies language keywords and standard identifiers
- Handles include files and macros
- Counts line numbers
- Removes whitespaces
- Reports illegal symbols
- Creates symbol table

Lexical analyzer does not have to be an individual phase. But having a separate phase simplifies the design and improves efficiency and portability. With these things in mind and in order to efficiently do these functions, the lexical analyser is not assigned a single pass of its own, rather

the lexical and the syntax analyser together functions as one pass and is specified in Figure 3.1.

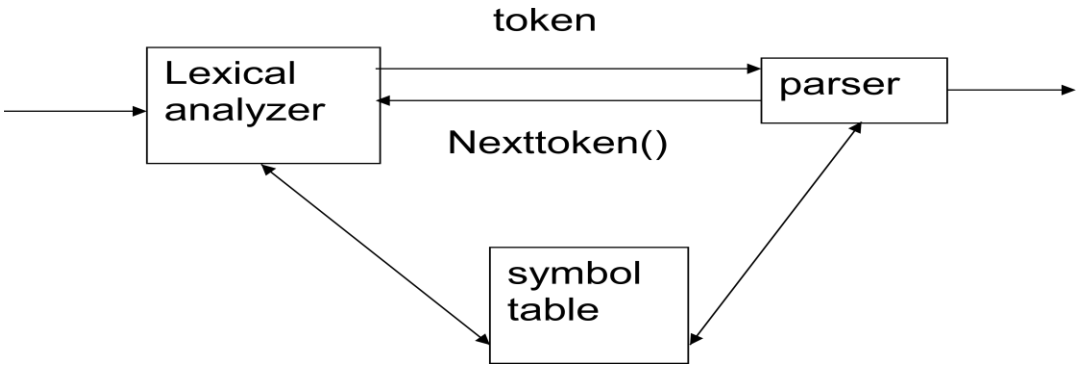

Figure 3.1 Role of the lexical analyser and parser

From figure 3.1, it could be understood that, the lexer and the parser are combined together into one pass, where the parser issues a "Nexttoken()" command and the lexer issues or gives a token to the parser. The grouping of the phases helps in achieving the following:

- Simplifies the syntax analysis in terms of language definition
- Enhances modularity and supports portability. It helps in using the same analyser-parser combination for various high-level languages across platforms as the analysis phase is target independent.
- Reusability
- Efficiency

Definitions

Let us look at some basic definitions pertaining to the lexical phase of the compiler. Token is a group of characters having a collective meaning. Lexeme is a particular instant of a token. For example, consider the variable "pi". So, token could be thought of as a name given to a lexeme. For this variable the token is "identifier" and the corresponding lexeme is "pi". Similarly for the string "if", the lexeme is "if" and the token is "keyword". The lexemes are typically described by defining a pattern for it. The set of rules describing how a token can be formed is defined as its pattern. So, defining patterns for every possible lexeme is one of the main jobs of this phase of the compiler. For example, for the token identifier the pattern is [a-z]|[A-Z]) ([a-z]|[A-Z]|[0-9])*. Here, this corresponds to a regular expression. The operator „[„ and „]" indicates that any one character in this range could be used. The operator „|" is the union operator. It indicates either this or that. The * is a Kleene closure operator which indicates zero or more combination of the symbols that has this operator. The implicit operator here is the concatenation operator "." which indicates the sequence of symbols that occur. We will discuss more on defining regular expressions in the subsequent sections.

Issues

Tokenizing is the primary job of this phase of the compiler. Hence, the primary issue is to find out how to identify tokens? Typically tokens are identified by defining patterns either as a regular expression or as an automata. The next issue is how the lexical phase will recognize the tokens using a token specification. This indirectly poses ways of implementing the nexttoken() routine. In order to solve these issues, the first two phases of the compiler are integrated.

Lexical Analysis Problem

Formally stating, given a set of token descriptions in terms of token name and regular expression defining the pattern for a lexeme, the lexical phase accepts an input string and partitions the strings into tokens (class, value). In choosing the lexeme, there could be two matching prefix that would be a part of two different tokens. For example, consider the operator "**" for exponentiation. Will the compiler, consider this is as a multiplication operator on seeing the first "*" and will it log the second "*" as an error? It considers this as an exponentiation operator because it tries to match the longest matching prefix. Thus ambiguity encountered in this phase is typically resolved by choosing the longest matching token and between two equal length tokens, the first one is selected. Table 3.1 lists some examples of tokens and their corresponding example lexemes.

Table 3.1 Few Examples of Tokens and their description.

TOKEN	Description	Sample lexeme
if	Character i, f	If
else	Characters e, l, s, e	else
Comparison	< or > or < = or >= or == or !=	<=
id	Letter followed by letters and digits	Pi, score, a123
Number	Any numeric constant	3.14, 9.08
Literal	Anything by ", within ""	"Seg fault"

Token Attributes

A token is represented along with its attribute. A typical attribute is a pointer to the symbol-table entry in which the information about the token is populated. Consider the following example

E=M*C**2 → (3.1)

For the statement in (3.1), the lexemes are “E”, “=”, “M”, “*”, “C”, “**”, “2”.

The tokens and the their attributes are listed below

<**id**, pointer to symbol-table entry for E>

<**assign_op**,> - attribute since it is a keyword indicating the operator

<**id**, pointer to symbol-table entry for M>

<**multi_op**,> - attribute since it is a keyword indicating the operator

<**id**, pointer to symbol-table entry for C>

<**exp_op**,> - no attribute since it is a keyword indicating the operator

<**num**,integer value 2>

Patterns: The lexical phase, decides the token based on defining patterns. Let us look at how to define patterns using regular expression. Table 3.2 gives the basic idea about the basic patterns of a regular expression.

Table 3.2 Basic regular expression

Basic Patterns	**Matching**	**Description**
x	The character x	Only one string “x”
.	Any character except newline	Any character
[xyz]	Any of the characters x, y, z	Strings are “x” or “y” or “z”
R?	An optional R, where R corresponds to any expression.	If R = a, then the possible strings are either “a” or “ϵ”
R*	Kleene Closure operator. Zero	If R = a, R* = $\bigcup_{i=0}^{\infty} R^i$ which indicates

	or more occurrences of R	infinite number of strings where „i' indicates the number of occurrences of R. R^i is formed as a concatenation of „i' R"s. R^2 = R R. $R^* = \{ R^0, R^1, R^2, \ldots R^\infty \}$ So for the example R = a, R* = {ε, a, aa, aaa, …}, where ε indicate a string of length 0. If R = ab, then R* = { ε, ab, abab, ababab, …}
R+	Positive closure operator. One or more occurrences of R	This is also like Kleene Closure except that it doesn"t have the string ε
R_1R_2	R_1 followed by R_2	R1 concatenated by R2
$R_1\|R_2$	Either R_1 or R_2	R1 or the string R2.
(R)	R itself	Just R with precedence for parenthesis.

Regular expressions are used to define the patterns using various regular expression operators. As far as the operators are considered, * has the highest precedence, followed by "**.**" and "|" has the least precedence. Consider the example of defining Pascal Language identifiers. The identifier can start with alphabets followed by 0 or more combinations of alphabets or numbers.

L(Pascal Identifier) = letter **.** (letter | digit)* → (3.2)

In statement (3.2), letter indicates any alphabet and digit indicates any integer between 0 and 9. The expression (letter | digit)* can be extended as { ε, letter, digit, letter.digit, digit.letter, letter.letter, digit.digit, ….}

Properties of Regular Expression (RE)

If L(r) and L(s) are regular expressions then L(r) U L (s) is also a RE and so is L(r) . L(s). L(r*) is also a RE if L(r) corresponds to a regular expression. For example, if the input alphabet, ∑ = {a, b}, then

- a* = {ε, a, aa, aaa, …}

- a | b = {a, b}
- (a|b)* = { ε, a, b, aa, ab, ba, bb, …}

Regular Definitions

Regular definitions are the names given to certain regular expressions and these names are later used for constructing the patterns. Regular definition is a sequence of the form

d1 → r1, d2 → r2, d3 → r3…

where each "di" is a symbol that is not in the input alphabet and each "ri" corresponds to a regular expression. To generate a regular expression, the left hand side "di" is replaced by the corresponding right hand side "ri". Consider the example of defining identifiers in pascal.

letter → A | B | …| z

digit → 0 | 1 | 2 | 3 … | 9

id → letter . (letter | digit)* → (3.3)

From statement (3.3), the identifier "id" is typically replaced with its RHS definition, where the constituent of the RHS is again stated as regular definition. Another example for defining number constant is given in statement (3.4) while the definition of statements is given in (3.5).

digit → 0 | 1 | … | 9

digits → digit **.** digit *

optionalFraction → **.**digits | ε

optionalExponent → (E(+ | - | ε)digits) | ε

number → digits**.** optionalFraction**.** optionalExponent → (3.4)

Statement regular definition is given in (3.5)

Stmt → if expr then Stmt | if expr then Stmt else Stmt | ε

expr → term relop term | term

term → id | number

id → letter . (letter|digits)*

relop → < | > | <= | >= | == | !=

number → digits → (3.5)

However, it may be hard to specify regular expressions for certain constructs like Strings and comments but constructing automata may be easier. In this context, it is to be understood that the language generated by an automata and the regular expression refers to a class of languages called regular language. Defining automata is easier rather than stating regular expressions. There is another option to combine both and specify partial automata with regular expressions on the edges. This doesn‟t require us to specify all the possible states but specify different actions at different states. As far as the automata is concerned a deterministic finite automata (DFA) is faster in string matching. However, constructing a DFA is difficult. Therefore, we create non-deterministic finite automata (NDFA) from every regular expression using an algorithm. Merge all the automata using epsilon moves (like the | construction). Then from the NDFA construct a DFA using another algorithm and then use a procedure to minimize the automaton starting with separate accepting states.

Automata

Automata typically refer to a Deterministic one. It is a five tuple representation (Q, ∑, δ, q0, F), where q0 belongs to Q, Q is a finite set of states, F is a subset of Q indicating final states,

δ is a mapping from Q x ∑ to Q. The transition function states, for every state on every input symbol there is exactly one transition. We define the language of the automata as

L = {w | δ (q0, w) = r, r is a subset of F}.

The interpretation of this definition is that, the string “w” should take you from the start state q0 to any one of the final state. Hence, every string has exactly one path and so ~~a~~ DFA is faster for string matching. On the other hand, a NFA is similar to a DFA, but it gives some flexibility. It is a five tuple representation (Q, ∑ , δ, q0, F), q0 belongs to Q, F is a subset of Q and δ is a mapping from Q x ∑ to 2^Q . The transition function here indicates that for every state on every input symbol there could be 0 or more transitions.

We define the language of the NFA as

L = {w | δ (q0, w) = R, where any one state of R is a subset of F}.

Since, multiple paths exists because of the existence of multiple transitions, a NFA takes more time for string matching. ε -NFA is same as NFA but with more flexibility in allowing to change state without consuming any input symbol.

The definition of ε -NFA is different from NFA only in the transition function which is defined as δ a mapping from Q x ∑ U { ε } to 2^Q. Hence, it is slower than NFA for string matching.

Summary: This module focused on constructing regular expressions and regular definitions as a way of defining pattern for regular expressions which will be used by the lexical phase of the compiler.