Principles of Compiler Design (SEng 3043)

Chapter Two
Lexical Analysis

Debremarkos Institute of Technology
School of Computing
Software Engineering Academic Program

By Lamesginew A. (lame2002@gmail.com)

Introduction: Lexical analysis

- It is the first phase of compilation process.
- It takes the modified source code from language preprocessors.
- It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.

The role of lexical analyzer

- ❖ Its main purpose is to make life easier for the subsequent syntax analysis phase.
- ❖ The main task of the lexical analyzer
  - ✓ reads the input characters of the source program
  - ✓ groups them into lexemes
  - ✓ produces sequence of tokens for each lexeme as output
- ❖ The stream of tokens is sent to the parser for syntax analysis.
- ❖ The lexical analyzer also interacts with the symbol table.
  - ✓ When it discovers a lexeme, it enters into the symbol table.
  - ✓ It may read information from symbol table to determine token.

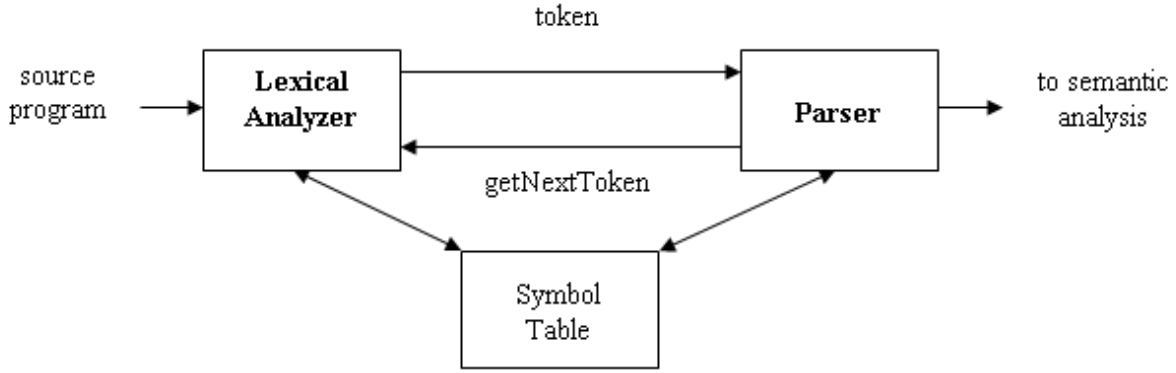


Figure: Interaction between lexical analyser and parser

- Additional tasks performed by lexical analyzer:
  - ✓ Stripping out comments and whitespace (blank, newline, tab).
  - ✓ Correlating error messages generated by the compiler with the source program.

Issues in Lexical Analysis

- Why to separate lexical analysis and parsing?
- Reasons for separating lexical analysis and syntax analysis phases.
  - ✓ Simplifies the design of the compiler.
  - ✓ To improve compiler efficiency
  - ✓ To enhance compiler portability.

Tokens, Patterns, and Lexemes

They are three related but distinct terms:

❖ ***Token***

- ✓ a pair containing token name and an optional attribute value.
- ✓ Tokens are units that are meaningful in the source language.
- ✓ Token are classification of lexical units; e.g.: keywords, constants, strings, numbers, operators, punctuation symbols, identifiers
- ✓ The token names are input symbols that the parser processes.

❖ ***Lexeme***

- ✓ Lexemes are specific character strings that make up a token.
- ✓ It is a sequence of characters in the source program that matches the pattern for a token.

❖ **Pattern**

- ✓ It is a rule describing the set of lexemes belonging to a token.
- ✓ It is a description of the form that lexemes of a token may take.
- ✓ In case of a keyword as a token, the pattern is the sequence of characters that forms the keyword.
- ✓ For identifiers and some other tokens, the pattern is a more complex structure that is *matched* by many strings.

Example *„ Cont'd*

TOKEN	SAMPLE LEXEME	INFORMAL DESCRIPTION
id	abc, x, pi	Letter followed by letters and/or digits
num	123,-1.75, 3.14159	any numeric constant
string	"compiler design"	anything surrounded by " "
keyword	else	characters e,l,s,e
operator	+, -, *, /	Any arithmetic operators + or – or * or /
comparison	<=, >, < >, = =	Any relational operator < or > or <= or >= etc

- In most programming languages, the following classes cover tokens:
  - ✓ One token for each keyword. The pattern for a keyword is the same as the keyword itself.
  - ✓ Tokens for operators, either individually or in classes such as comparison as mentioned above.
  - ✓ One token for all identifiers
  - ✓ One or more tokens representing constants, such as numbers and literal strings
  - ✓ Tokens for each punctuation symbol, such as left and right parentheses, comma and semicolon

Attributes for Tokens

- When more than one lexeme can match a pattern, the lexical analyzer must provide additional information about the particular lexeme that matched.
- This is done by using attribute value that describes the lexeme represented by the token.
- Attribute is the value associated with the token
- The lexical analyzer collects information about tokens into their associated attributes.
- A token has usually only a single attribute - a pointer to the symbol-table entry in which the information about the token is kept; the pointer becomes the attribute for the token.
- The symbol table entry contains various information about the token such as the lexeme, its type, the line number in which it was first seen, etc.
- It is necessary to check if the token is already in the table

- Example, (in FORTRAN): **E = M * C ** 2**, the tokens and attributes are the following:
  - ✓ <**id**, pointer to symbol-table entry for E>
  - ✓ <**assign_op**>
  - ✓ <**id**, pointer to symbol-table entry for M>
  - ✓ <**mult_op**>
  - ✓ <**id**, pointer to symbol-table entry for C>
  - ✓ <**exp_op**>
  - ✓ <**number**, integer value 2>

Note: for operators, punctuations, and keywords; there is no need for an attribute value.

Lexical Errors

- Are errors thrown by lexer when unable to continue.
- When there's no way to recognize a lexeme as a valid token for lexer.
- Some errors are out of power of lexical analyzer to recognize, like:
  - fi (a = = f(x)) …
- However it may be able to recognize errors like:
  - d = 2r
- Such errors are recognized when no pattern for tokens matches a character sequence.
- When an error occurs, the lexical analyzer recovers by:
  - ✓ Skipping (deleting) successive characters from the remaining input
  - ✓ Deleting extraneous characters from the remaining input
  - ✓ Inserting missing characters from the remaining input
  - ✓ Replacing an incorrect character by a correct character
  - ✓ Transposing two adjacent characters

Input Buffering

- The lexical analyzer reads the program , character by character and generates valid tokes.
- Here the input string must be stored temporarily.
- We can say the input string must be buffered.
- The lexical analyzer has to have the information which are going to take place(future information).
- It has to know the characters ahead of the present character to make a decision whether it is a valid or not.
- It is true that we often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme.
- There are many situations where we need to look at least one additional character ahead.
- For instance, we cannot be sure we've seen the end of an identifier until we see a character that is not a letter or digit, and therefore is not part of the lexeme for **id**.
- Single-character operators like =, or < could also be the beginning of a two-character operator like = =, or <=; so there is confusion in judging the token.
- If we know the look ahead character, this problem can be solved to some extend.

- the input buffer in the case of lexical analyzer, is divided in to two.
  - One is traced by the current pointer (*lexemeBegin*), and
  - the second is traces by the forward pointer (*forward*).
- the current pointer marks the beginning of the current lexeme; and the forward pointer scans a head until a pattern match is found.

❑ In buffer pair buffering scheme,

- Once the lexeme is determined, *forward* is set to the character at its right end.
- After the lexeme is recorded as an attribute value of the token returned to the parser, *lexemeBegin* is set to the character immediately after the lexeme just found.
- Advancing *forward* requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move forward to the beginning of the newly loaded buffer.

❑ In Sentinel buffering scheme,

- The two test made for each character read in buffer pair scheme (buffer end test and current character test) can be combined together by extending each buffer to hold sentinel character at the end.
- The *sentinel* is a special character that cannot be part of the source program, and a natural choice is the character **eof**.

Specification of Tokens

- In theory of compilation, regular expressions are used to formalize the specification of tokens.
- Regular expressions are an important notation for specifying lexeme patterns.
- They cannot express all possible patterns
- But, they are effective in specifying those types of patterns that we need for tokens.

Recommendation : go back to your "Formal language and automata theory" course to recap more on regular expressions.

Strings and Languages

❖ *Alphabet* ($\sum$): any finite set of symbols; e.g. of symbols: letters, digits, punctuation.

- The set {0, 1} is the binary alphabet.
- ASCII is an important example of an alphabet.

❖ *String:* a finite sequence of symbols over an alphabet (drawn from alphabet).

- The length of a string *s*, written as |s|, is the number of occurrences of symbols in *s*.
- The empty string, denoted by ε, is the string of length zero.

❖ *Language*: any countable set of strings over some fixed alphabet.

- For example, the set of strings consisting of n 0's followed by n 1's { ε ,01, 0011,…}
- the set of binary numbers whose value is prime {10, 11, 101, 111, 1011, …} are languages.

Operations on Languages

- In lexical analysis, the most important operations on languages are *union*, *concatenation*, and *closure*.
- *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.

✓ Union of L and M: $L \cup M = \{s \mid s \in L \text{ or } s \in M\}$

✓ Concatenation of L and M: $LM = \{xy \mid x \in L \text{ and } y \in M\}$

✓ Exponentiation of L: $L^0 = \{\varepsilon\}$; $L^i = L^{i-1}L$

✓ Kleene closure of L: $L^* = \cup_{i=0,\ldots,\infty} L^i$

✓ Positive closure of L: $L^+ = \cup_{i=1,\ldots,\infty} L^i$

Regular expressions

- The regular expressions are built recursively out of smaller regular expressions, using the rules described below.
- Basis rules:
  - ✓ ε is a regular expression, and L (ε) is {ε}
  - ✓ If *a* is a symbol in Σ, then *a* is a regular expression, and L(*a*) = {*a*}
- Suppose **r** and **s** are regular expressions denoting languages L(**r**) and L(**s**), respectively.
  - ✓ (r) | (s) is a regular expression denoting the language L(r) U L(s).
  - ✓ (r) (s) is a regular expression denoting the language L(r) L(s) .
  - ✓ $(r)^*$ is a regular expression denoting $(L(r))^*$.
  - ✓ (r) is a regular expression denoting L(r).

➢ A language that can be defined by a regular expression is called a regular set or regular language.

➢ If two regular expressions r and s denote the same regular set, we say they are equivalent and write as r = s.

Regular definition

Regular definitions introduce a naming convention to certain regular expressions, in the form

$d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

…

$d_n \rightarrow r_n$

Where

- Each d_i is a new symbol, not in $\sum$ and not the same as any other of the d's, &
- Each r_i is a regular expression over the alphabet $\sum \cup \{ d_1, d_2, \ldots, d_{i-1} \}$

- C identifiers are strings of letters, digits, and underscores.
- Here is a regular definition:
  - ✓ *letter* → A | B | . . . | Z | a | b | . . . | z
  - ✓ *letter_* → A | B | . . . | Z | a | b | . . . | z | _
  - ✓ *digit* → 0 | 1 | . . . | 9
  - ✓ *id* → *letter_*(*letter_* | *digit*)*

Extensions of Regular Expressions

- Many extensions have been added to regular expressions to enhance their ability to specify string patterns.
- Some regular expression variants that use today are the following.

❖*One or more instances*

- ✓ $r^+ = r\ r^* = r^*\ r$
- ✓ $r^* = r^+ \mid \varepsilon$

❖*Zero or one instance*

- ✓ $r? = r \mid \varepsilon$, or put another way, $L(r?) = L(r) \cup \{\varepsilon\}$.

❖*Character classes*

- A regular expression $a_1 \mid a_2 \mid \ldots \mid a_n = [a_1 a_2 \ldots a_n]$.
- Consecutive letters and digits can be expressed by $[a_1 - a_n]$; just the first and the last separated by hyphen.
- Thus [abc] is shorthand for a | b | c, and [a-z] is shorthand for a | b | . . . | z

- Using these short hands, we can rewrite the above regular definitions as follows.
  - ✓ letter_ → [A – Za – z _]
  - ✓ digit → [0 - 9]
  - ✓ id → letter_ (letter_ | digit)*

Recognition of Tokens

- Starting point is the language grammar to understand the tokens.
- Consider the following grammar fragment:

$$
\begin{aligned}
\textit{stmt} &\rightarrow \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \\
&\quad \mid \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \textbf{ else } \textit{stmt} \\
&\quad \mid \varepsilon \\
\textit{expr} &\rightarrow \textit{term} \textbf{ relop } \textit{term} \\
&\quad \mid \textit{term} \\
\textit{term} &\rightarrow \textbf{id} \\
&\quad \mid \textbf{num}
\end{aligned}
$$

- where the terminals **if, then, else, relop, id**, and **num** generate sets of strings given by the following regular definitions:

if	→ if
then	→ then
else	→ else
relop	→ < | <= | <> | > | >= | =
id	→ letter (letter | digit)*
num	→ digit+ (. digit+)? (E (+|-)? digit+)?

- where **letter** and **digit** are as defined previously.

- For this language fragment the lexical analyzer will recognize the keywords **if, then, else**, as well as the lexemes denoted by **relop, id**, and **num**.

➢ Lexemes are separated by white space, consisting of non-null sequences of blanks, tabs, and newlines.

➢ Lexical analyzer will strip out white space.

➢ It will do so by comparing a string against the regular definition **ws**, below.

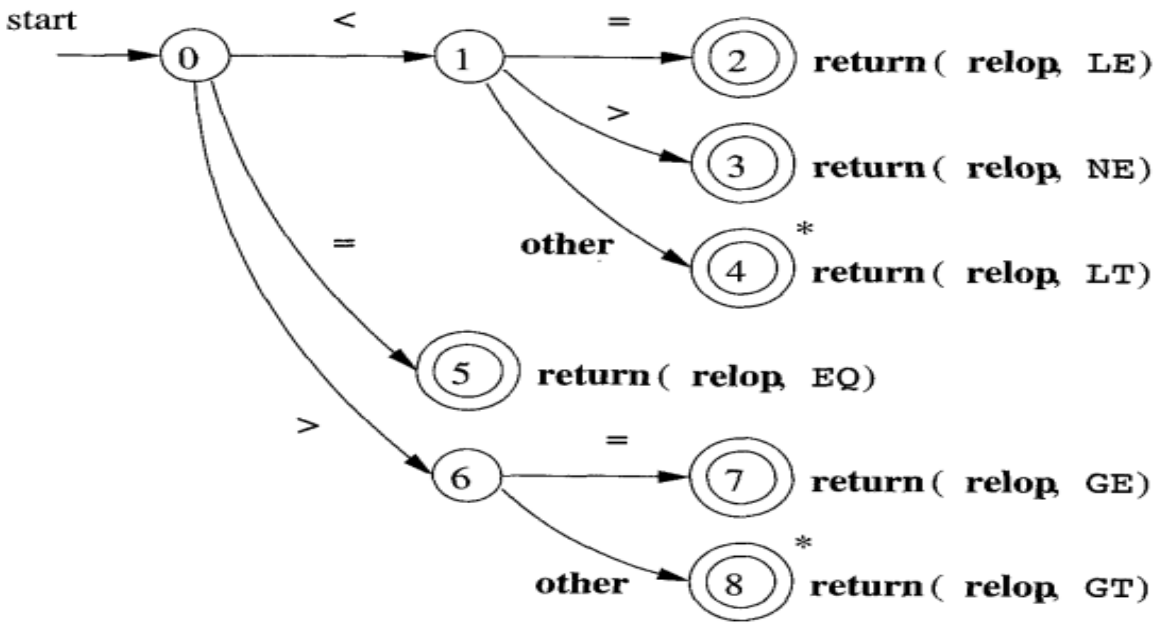
- ✓ *delim* → (blank | tab | newline)
- ✓ *ws* → delim^+

➢ If a match for **ws** is found, the lexical analyzer does not return a token to the parser.

➢ Rather, it proceeds to find a token following the white space and returns that to the parser.

Transition diagram

- It is diagrammatic representation like Flowchart, used for lexical analyzers to recognize tokens.
- It has a collection of nodes called states and edges directed from one state of the transition diagram to another.
- Transition diagram for **relop** token is the following



- Note, that state 4 and 8 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 the fact from state 5.


THANK YOU
VERY
MUCH !!