LEXICAL ANALYSER

THIRD REVIEW REPORT

Submitted by

Vinti Bodhwani (15BCE0719)

Harshit Kedia (15BCE0)

Ratnasambhav Priyadarshi (15BCE0646)

Prepared for

Theory of Computation and Compiler Design (CSE2002) – Project Component

Submitter to

Senthil Kumar

Assistant Professor

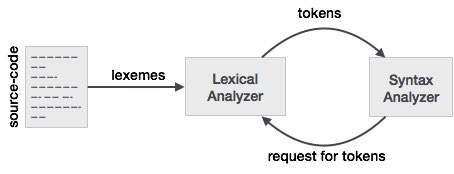
School of Computer Science and Engineering



Introduction

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

If the lexical analyzer finds a token 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.



Lexemes are said to be a sequence of characters (alphanumeric) in a token. 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.

For example, in C language, the variable declaration line

int value = 100;

Contains the tokens:

int (keyword), value (identifier), = (operator), 100 (constant) and ;

Let’s look at another example. Consider this expression in the C programming language:

sum = 3 + 2;

Tokenized and represented by the following table:

|  |  |
| --- | --- |
| **Lexeme** | **Token category** |
| sum | "Identifier" |
| = | "Assignment operator" |
| 3 | "Integer literal" |
| + | "Addition operator" |
| 2 | "Integer literal" |
| ; | "End of statement" |

Tokenization is the process of demarcating and possibly classifying sections of a string of input characters. The resulting tokens are then passed on to some other form of processing. The process can be considered a sub-task of parsing input.

'Tokenization' has a different meaning within the field of computer security.

Take, for example,

The quick brown fox jumps over the lazy dog

The string isn't implicitly segmented on spaces, as an English speaker would do. The raw input, the 43 characters, must be explicitly split into the 9 tokens with a given space delimiter (i.e. matching the string " " or regular expression /\s{1}/).

The tokens could be represented in XML,

**<sentence>**

**<word>**The**</word>**

**<word>**quick**</word>**

**<word>**brown**</word>**

**<word>**fox**</word>**

**<word>**jumps**</word>**

**<word>**over**</word>**

**<word>**the**</word>**

**<word>**lazy**</word>**

**<word>**dog**</word>**

**</sentence>**

Project Scope:

Our lexical analyzer is designed to accept an input sting and tokenize it into the various kinds of tokens. It will be able to detect the following types of tokens:

1. Words
2. Numbers
3. Escape Character
4. Whitespaces
5. Comments
6. Punctuations
7. End of File
8. Keywords

This will be done with the help of Regular Expressions. The output will include the position of each token, i.e. there line number and position in that line.

Code:

We have written the program in PHP. The tokens are detected using the following regular expressions:

"T\_LATEX\_COMMAND" => array(

"regex" => '/^\\\([^\_$#%&][a-zA-Z0-9\.\_-]+)(({[^}]\*})|(\[[^\]]\*\]))\*/s',

"store" => true

),

"T\_WORD" => array(

"regex" => '/^[a-zA-ZÀ-ÿ][a-zA-Z0-9À-ÿ]\*/',

"store" => true

),

"T\_NUMBER" => array(

"regex" => '/^[0-9]([0-9\.,]\*[0-9])?/',

"store" => true

),

"T\_ESCAPED" => array(

"regex" => '/^\\\[\_$#%&]/',

"store" => true

),

"T\_WHITESPACE" => array(

"regex" => '/^\s+/',

"store" => false

),

"T\_COMMENT" => array(

"regex" => '/^%[^\n]\*\n/',

"store" => false

),

"T\_PUNCTUATION" => array(

"regex" => '/^[,\.\_!\?\(\)-:;\'"–<>|@\\*\{\}$#%&=+\[\]`]+/',

"store"=> true

),

"T\_EOF"=> array(

"regex" => '/^$/',

"store"=> false

)