Turk J Elec Eng & Comp Sci
() : –
© TÜBİTAK
doi:10.3906/elk-

Lexical analysis solution

Nazmican DİKMEN[1*], Zekiye DOĞAN[2†], Kasım GÖKMEN[3‡]
[1]Computer Engineering Department, Faculty of Engineering,
Dokuz Eylul University, Izmir, Turkey
[2]Computer Engineering Department, Faculty of Engineering,
Dokuz Eylul University, Izmir, Turkey
[3]Computer Engineering Department, Faculty of Engineering,
Dokuz Eylul University, Izmir, Turkey

Received: .201 • **Accepted/Published Online:** .201 • **Final Version:** ..201

Abstract: The study aims to find an approach for lexical analysis of a given input. Finding a solution to the given problem, and generating a valid output response is the essential purpose of the study.

Key words: Lexical analysis

1. Introduction

The project is a lexical analysis project. Lexical analysis is crucial for programming languages to function properly. It is the first layer of programming languages where commands go through. Within the boundaries of the given programming language Ceng++, a lexical analysis is applied to a given input source file ”code.Ceng”. During the analysis, results are printed as an output, and if there are any errors found, the program is terminated and they are printed and displayed to users as an output file ”code.lex”.

2. Project analysis and solutions

Lexical analysis for Ceng++ works as an interpreter to control each and every line of a given command input, and produce a valid output to the users. If in the given source code, a code block is written in a fashionable way that corresponds to the rules of Ceng++, the program expects to generate a log output. Else, within any error or faulty expression, it is also expected for the program to generate the same format of the log output, this time pointing out the faulty lines and/or expressions.

First and foremost of the boundaries is the identifier analysis. A typical command line starts with an identifier first in order to define the main behaviour of the said line. Whenever the program encounters an identifier, it checks its usage for whether if it's properly used or not. Function to check an identifier:

$$isvalidIdentifier(char * str) \tag{1}$$

Another checkpoint is operator controls. A command line has its meaning via operators. There will not be a initialization or defining of a value without operators. They also come in handy with string operations,

*Correspondence: nazmican.dikmen@ceng.deu.edu.tr
†Correspondence: zekiye.dogan@ceng.deu.edu.tr
‡Correspondence: kasim.gokmen@ceng.deu.edu.tr

This work is licensed under a Creative Commons Attribution 4.0 International License.
This work is licensed under a Creative Commons Attribution 4.0 International License.

e.g. string concatenation, as well as with arithmetical operations. Function to check an operator:

$$isValidOperator(char\ ch) \tag{2}$$

Third control would be the keywords. Keywords define the conditions, types, behaviours, and workflow of a code. Since there are naturally many keywords in Ceng++ as it is in any other programming languages, it is crucial to take their usages in consideration when implementing this control. For example loops behave differently then variable types etc. Function to check a keyword:

$$isValidKeyword(char * str) \tag{3}$$

Almost every arithmetic operations require numbers. Whether they're integer, float, or long type numbers, they should be taken into consideration. Function to check an integer:

$$isValidInteger(char * str) \tag{4}$$

Function to check a float:

$$isRealNumber(char * str) \tag{5}$$

Quite a lot amount of keywords require bracket usage. Whether they are normal brackets, curly brackets or angular brackets, they are needed. These brackets are controlled using matching-brackets method. All opening brackets are controlled whether if they're matching with the corresponding index to them. Function to control brackets:

$$*isValidBrackets(char\ ch) \tag{6}$$

Strings are often used in a general basis when using programming languages. Lexical analysis of strings require quotation marks as a delimiter for strings. Any code piece between two quotation marks are considered as strings. Function to control strings:

$$isValidString(char * str) \tag{7}$$

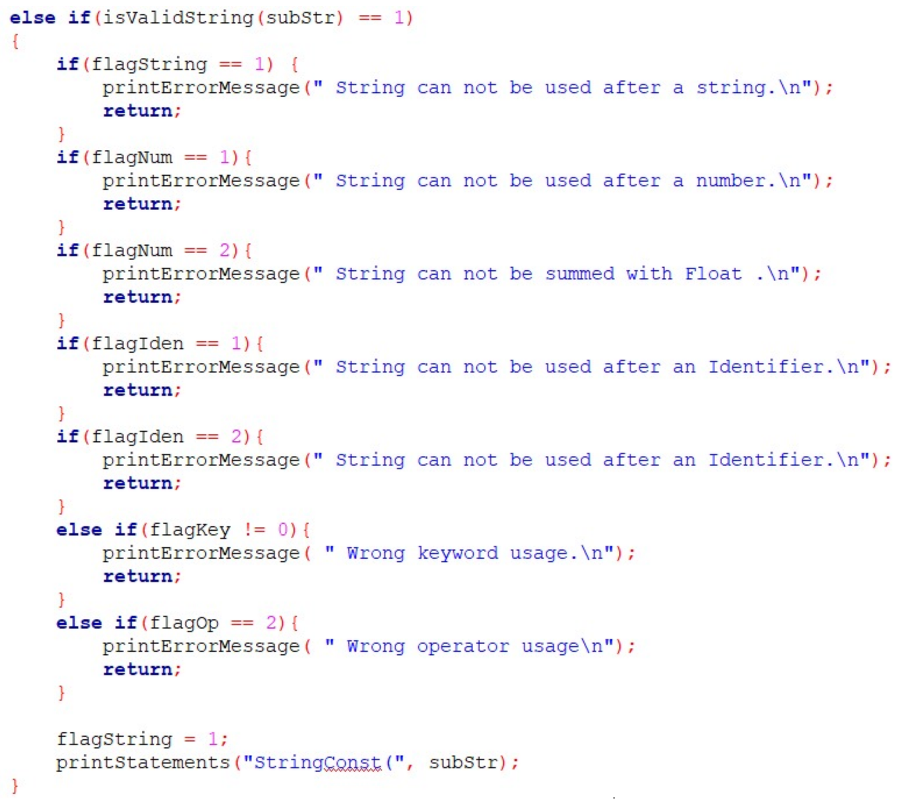
For the last but not least, comment lines are also present in Ceng++ with a delimiter of brackets with an asterisk. $(*example*)$

3. Algorithm analysis and solutions

Beginning lexical analysis starts with reading the input file and converting it into a single char array as a working instance. Throughout this buffer process, blank spaces are ignored, new lines are removed and all characters are converted to lower case since the Ceng++ programming language is a case insensitive language.

After the first phase is done, lexical analysis is processed on the created input char array. This array is iterated over through a loop, and gets parsed with corresponding delimiters until it yields a flag which indicates a type of lexeme. Any yielded flag is controlled using different varieties of if else conditions thoroughly. These tokens and errors if any exist that are encountered during analysis are sent to an output char array to be used in writing output file operations. If succeeded, a info message is written into an output .lex file, indicating its lexeme and token values. If any errors are found, they are also printed with additional information of the position and the source issue of the problem into the output .lex file from the previously generated output char array, breaking from the code afterwards.

Additional conditions which define flags for lexical analysis can be found above under **Project analysis and solutions**. A glossary for flags can be found in table Flags glossary.



```
else if(isValidString(subStr) == 1)
{
    if(flagString == 1) {
        printErrorMessage(" String can not be used after a string.\n");
        return;
    }
    if(flagNum == 1){
        printErrorMessage(" String can not be used after a number.\n");
        return;
    }
    if(flagNum == 2){
        printErrorMessage(" String can not be summed with Float .\n");
        return;
    }
    if(flagIden == 1){
        printErrorMessage(" String can not be used after an Identifier.\n");
        return;
    }
    if(flagIden == 2){
        printErrorMessage(" String can not be used after an Identifier.\n");
        return;
    }
    else if(flagKey != 0){
        printErrorMessage( " Wrong keyword usage.\n");
        return;
    }
    else if(flagOp == 2){
        printErrorMessage( " Wrong operator usage\n");
        return;
    }

    flagString = 1;
    printStatements("StringConst(", subStr);
}
```

Figure 1. A condition example.

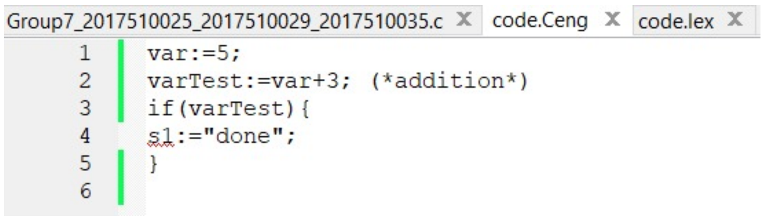
4. Tables and figures

Table 1. Sample lexical analysis result.

Lexeme	Token
Keyword	if
LeftPar	(
Identifier	var
RightPar	)
LeftCurlyBracket	{
Identifier	s1
Operator	:=
StringConst	done
EndOfLine	;
RightCurlyBracket	}

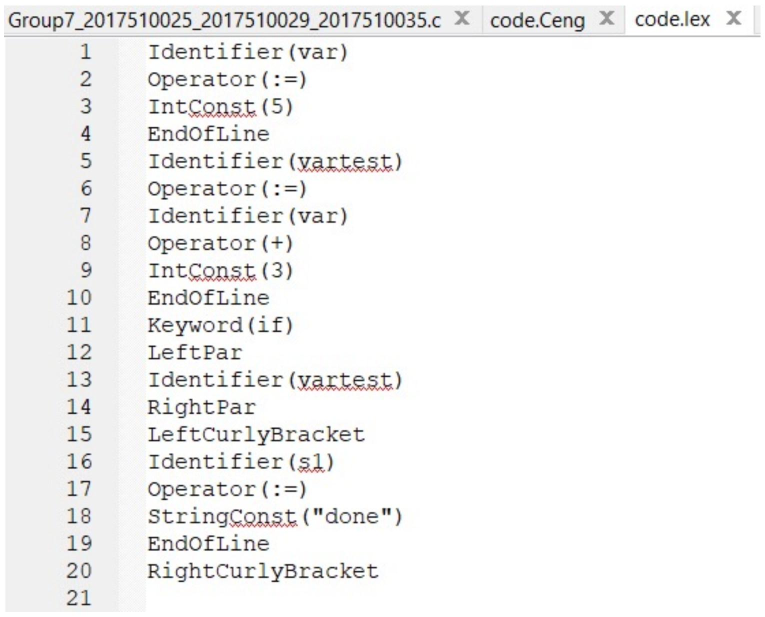
Table 2. Flags glossary.

	flagString	flagNum	flagIden	flagKey	flagOp	flagDo
0	Prev not String.	Prev not int.	Prev not ident.	Prev not keyword.	Prev not operator.	Curly br. after while.
1	Prev is String.	Prev is int.	Prev is ident.	while, for, if	+, -, *, /	EOL after while.
2	Str Concat.	+, -, *, / needs.	-	int, long, float, goto	++, - -	-
3	-	++, - - needs.	-	break, continue	:=	-
4	-	-	-	Curly br. after 1.	-	-
5	-	-	-	do	-	-



```
var:=5;
varTest:=var+3;  (*addition*)
if(varTest){
s1:="done";
}
```

Figure 2. Input code.Ceng file.



```
Identifier(var)
Operator(:=)
IntConst(5)
EndOfLine
Identifier(vartest)
Operator(:=)
Identifier(var)
Operator(+)
IntConst(3)
EndOfLine
Keyword(if)
LeftPar
Identifier(vartest)
RightPar
LeftCurlyBracket
Identifier(s1)
Operator(:=)
StringConst("done")
EndOfLine
RightCurlyBracket
```

Figure 3. Output code.lex file.

Acknowledgment

The project is concluded matching each and every expectation that was mentioned from the assignment document. Apart from the expected features, the project is moved to a more advanced level by adding additional error checks within the scope of the project. Project weight is distributed equally amidst the project members.

References

[1] Patrick Hanks. Lexical Analysis Norms and Exploitations. The MIT Press. January 25, 2013.

[2] Introduction of Lexical Analysis - https://www.geeksforgeeks.org/introduction-of-lexical-analysis/

[3] Online LaTeX Editor - https://www.overleaf.com/learn

[4] Lexical Analysis In 4 Simple Points - https://www.jigsawacademy.com/blogs/business-analytics/lexical-analysis/

[5] Lexical Analysis - https://en.wikibooks.org/wiki/Compiler_Construction/Lexical_analysis

[6] LATEX Tutorials A PRIMER, Indian TEX Users Group, Trivandrum, India, 2003 September, EDITOR: E. Krishnan, COVER: G. S. Krishna. - https://www.tug.org/twg/mactex/tutorials/ltxprimer-1.0.pdf