CSE 310: Compiler Sessional
Assignment 2
Construction of a Lexical Analyzer [1]

April 5, 2015

1 Introduction

Lexical analyzer, also known as scanner, scans the source program as a sequence of characters and converts them into a sequence of tokens i.e., larger meaningful textual units [1, 3]. For example, scanner converts the character stream `var I : integer;` into the token stream `VAR ID COLON INTEGER SEMICOLON` as shown in Figure 1.

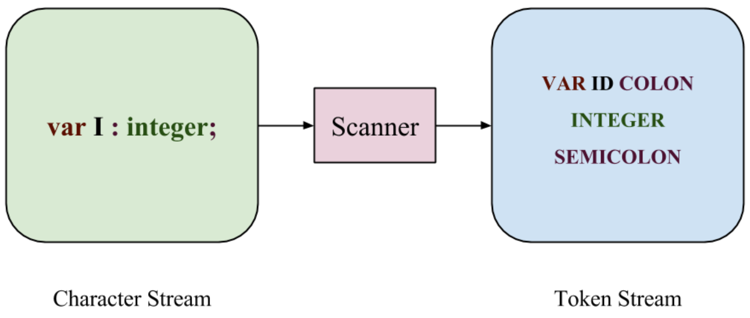


Figure 1: Lexical Analyzer in action

On our way to constructing a simple compiler for a subset of the language "Pascal", we have already implemented a rudimentary symbol-table in our last assignment. In this assignment, we will generate a lexical analyzer using `Flex` [2].

2 Input

The input to our program will be a text file containing Pascal source code. The source code may contain lexical errors. For example, please refer to the Appendix of this article. We will provide input file name as input from command line. Therefore, standard C command line arguments should be used for this assignment.

[1]Special Thanks to Sumaiya Nazeen

3 Output

For the given input source code, the lexical analyzer will produce two output files, one for the token stream and the other for logging the various operations of the lexical analyzer.

The token stream output file will list down all the tokens (in `<TYPE, VALUE>` or `<KEYWORD>` format) that are encountered in the input life. Appendix of this article contains some sample token stream output files for given input files.

The log output file, on the other hand, should record details of every operation of the scanner, state of the symbol-table at each step, messages generated by scanner for each tokens along with corresponding line numbers of the input file, and at last, total line count of the input file. Sample log files for given input files will be uploaded with this document.

We will provide both output file names as input from command line. Therefore, standard C command line arguments should be used for this assignment.

4 Implementation Isssues

Some general guidelines for the lexical analysis are as follows:

- **Keywords:** The keyword list for our subset of Pascal language is given in Table 1. For each keyword found in the input source code, we should print a token of the format, `<keyword_name>` in the token stream output file. In the log output file, the corresponding message will be, “`Line <line_no>: Keyword <keyword_name> is found.`” Keywords will not be inserted in the symbol-table.

Table 1: Subset of Pascal keywords

program	then	real
if	do	var
not	while	of
end	function	array
begin	procedure	write
else	integer	

- **Language Constructs:** The construct list for our subset of Pascal language is given in Table 2. For each construct found in the input source code, we should print a token of the format, `<Type,Value>` in the token stream output file. In the log output file, the corresponding message will be, “`Line <line_no>: <construct_value> is found.`” Construct symbol will be inserted in the symbol-table and after inserting the symbol, we should print the current contents of the symbol-table.

- **Pascal Strings:** Both single-line and multi-line double quoted strings are supported in Pascal as shown in Listing 1 and 2. For detailed example, please refer to the sample input files given

Table 2: Subset of Pascal Language Constructs

Type	Value	Pattern Description
ID	Any identifier (e.g., `i, Average, _t1, t2`)	letter or underscore followed by letters, digits, or underscores
NUM	Any real number (e.g., `0, 1, 1.2, 0.3E2, 12E2.3, 1.23E-5`)	Sequence of digits having optional fractional and signed or unsigned exponent part
RELOP	`<, <=, >, >=, =, <>`	
ADDOP	`+, -, or`	
MULOP	`*, /, div, mod, and`	
ASSIGNOP	`:=`	
BRACKET	`[, ]`	
PAREN	`(, )`	
DOTDOT	`..`	
COMMA	`,`	
SEMICOLON	`;`	
COLON	`:`	
DOT	`.`	

in the Appendix of this article. For each string found in the input source code, we should print nothing in the token stream output file. However, in the log output file, we should print a message like, "`Line <line_no>: String <text of the string> is found.`" Strings will not be inserted in the symbol-table.

```
"Hello, World!"
```

Listing 1: Single-line double quoted string.

```
"This is an example of\\
multiline string\\
in\\
pascal."
```

Listing 2: Multi-line double quoted string.

- **Pascal Comments:** Both single-line and multi-line comments are supported in Pascal as shown in Listing 3 and 4. Pascal comments start with a { and end with a }, no { appears inside the text of an pascal comment. For each comment found in the input source code, we should print nothing in the token stream output file. However, in the log output file, we should print a message like, "`Line <line_no>: Comment <text of the comment> is found.`" Comments will not be inserted in the symbol-table.

```
{Single Line Pascal Comment Example}
```

Listing 3: Single-line pascal comment.

```
{
Multi-line Pascal
Comment Example,
}
```

Listing 4: Multi-line pascal comment.

- **White Space:** We will ignore all white spaces in the input file.
- **Lexical Error Reporting:** If the input source code file contains following lexical errors we should report corresponding errors along with line numbers in the log file.
  - Unrecognized characters, `@, ~`
  - Ill-formed identifiers and numbers, `1abc, 1.2.3, .5`
  - Unfinished Strings
  - Ill-formed Strings
  - Unfinished Comments
  - Ill-formed Comments
  - Any other lexical errors
- **Symbol-table Dump:** After every insertion in our symbol-table, we should print or dump the contents of the symbol-table. While dumping the symbol-table contents, we will only print non empty buckets unlike previous assignment. At the end, when the scanning of the input file is completed, we should dump the symbol-table one last time.
- **Line Count:** We should also count the total number of lines in the input file and print it in the log file at the end.

5 Flex Compilation Commands

Command sequence to compile Flex source codes is as follows:

- `flex test.l`
- `g++ -lfl lex.yy.c`
- `./a.out in.txt token.txt log.txt`

6 Deadline

Deadline to submit this assignment is the **next sessional class**. Submissions after the deadline will not be accepted.

References

[1] Lan Gao. Flex tutorial. `http://alumni.cs.ucr.edu/~lgao/teaching/flex.html`. [Online; last accessed April 4, 2015].

[2] V Paxson, W Estes, and J Millaway. The flex manual-lexical analysis with flex. `http://courses.softlab.ntua.gr/compilers/flex.pdf`. [Online; last accessed April 4, 2015].

[3] Wikipedia. Lexical analysis — wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Lexical_analysis`, 2015. [Online; last accessed April 4, 2015].

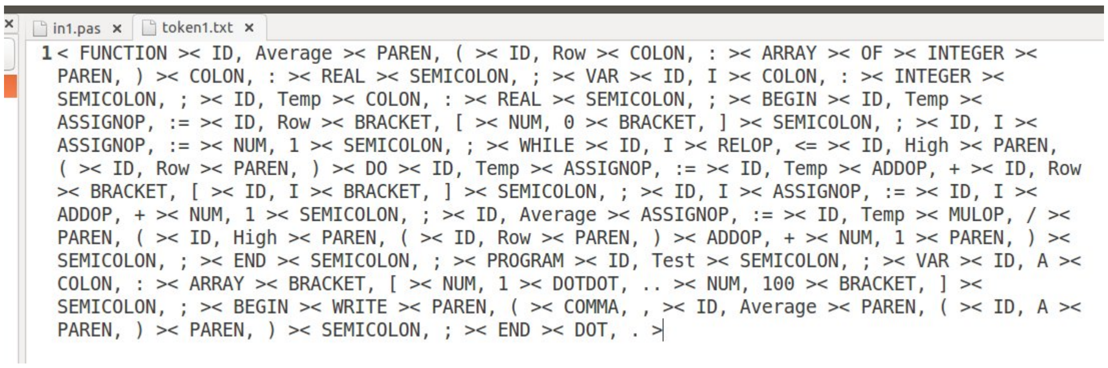
Appendices

A Sample Input Output Files

In this appendix, we provide few examples of sample input and output for this assignment.A sample input source code file is shown in Listing 6. The token stream output file for this source code is shown in Figure 2. The corresponding *log* output file `log1.txt` is not shown due to it's size constraint.

```
function Average (Row : array of integer) : real;
    var I : integer;
        Temp : real;
    begin
        Temp := Row[0];
        I := 1;
        while I <= High(Row) do
            Temp := Temp + Row[I];
            I := I+1;
            Average := Temp / (High(Row)+1);
    end;

{This program calculates the average of first 100 natural numbers
and prints it to the console.}

program Test;
    var A : array[1..100];
    begin
        write("Average of 100 numbers: ",Average(A));
    end.
```

Listing 5: Sample input source code file, `in1.pas`



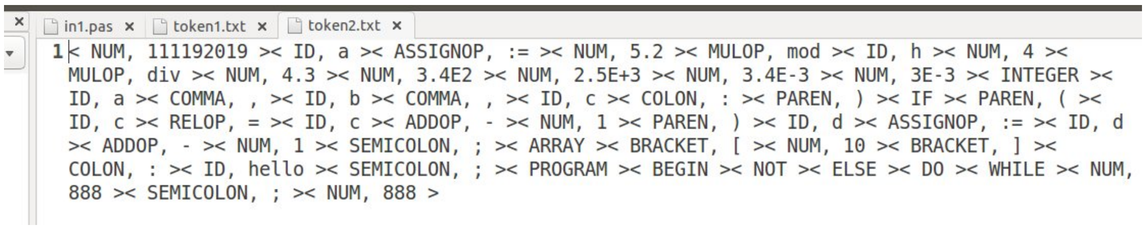
```
< FUNCTION >< ID, Average >< PAREN, ( >< ID, Row >< COLON, : >< ARRAY >< OF >< INTEGER ><
PAREN, ) >< COLON, : >< REAL >< SEMICOLON, ; >< VAR >< ID, I >< COLON, : >< INTEGER ><
SEMICOLON, ; >< ID, Temp >< COLON, : >< REAL >< SEMICOLON, ; >< BEGIN >< ID, Temp ><
ASSIGNOP, := >< ID, Row >< BRACKET, [ >< NUM, 0 >< BRACKET, ] >< SEMICOLON, ; >< ID, I ><
ASSIGNOP, := >< NUM, 1 >< SEMICOLON, ; >< WHILE >< ID, I >< RELOP, <= >< ID, High >< PAREN,
( >< ID, Row >< PAREN, ) >< DO >< ID, Temp >< ASSIGNOP, := >< ID, Temp >< ADDOP, + >< ID, Row
>< BRACKET, [ >< ID, I >< BRACKET, ] >< SEMICOLON, ; >< ID, I >< ASSIGNOP, := >< ID, I ><
ADDOP, + >< NUM, 1 >< SEMICOLON, ; >< ID, Average >< ASSIGNOP, := >< ID, Temp >< MULOP, / ><
PAREN, ( >< ID, High >< PAREN, ( >< ID, Row >< PAREN, ) >< ADDOP, + >< NUM, 1 >< PAREN, ) ><
SEMICOLON, ; >< END >< SEMICOLON, ; >< PROGRAM >< ID, Test >< SEMICOLON, ; >< VAR >< ID, A ><
COLON, : >< ARRAY >< BRACKET, [ >< NUM, 1 >< DOTDOT, .. >< NUM, 100 >< BRACKET, ] ><
SEMICOLON, ; >< BEGIN >< WRITE >< PAREN, ( >< COMMA, , >< ID, Average >< PAREN, ( >< ID, A ><
PAREN, ) >< PAREN, ) >< SEMICOLON, ; >< END >< DOT, . >
```

Figure 2: Token stream output file `token1.txt` for the input file, `in1.pas`

Another set of source code input file and token stream output file is shown in Listing 6 and Figure 3 respectively.

```
{This is a

program}

111192019
a := 5.2 mod h

4 div 4.3

3.4E2
2.5E+3
3.4E—3
3E—3


integer a,b,c
"This \\
is a multi—line \"quoted \\
string."

"This is a single—line quoted string" :)
if (c = c — 1)
  // do nothing :)
d := d —1; {This is an assignment statement}

array[10] : hello;


program begin not else do while



8.abc ab.c 888;888
```

Listing 6: Another sample input source code file, `in2.pas`



```
< NUM, 111192019 >< ID, a >< ASSIGNOP, := >< NUM, 5.2 >< MULOP, mod >< ID, h >< NUM, 4 ><
MULOP, div >< NUM, 4.3 >< NUM, 3.4E2 >< NUM, 2.5E+3 >< NUM, 3.4E-3 >< NUM, 3E-3 >< INTEGER ><
ID, a >< COMMA, , >< ID, b >< COMMA, , >< ID, c >< COLON, : >< PAREN, ) >< IF >< PAREN, ( ><
ID, c >< RELOP, = >< ID, c >< ADDOP, - >< NUM, 1 >< PAREN, ) >< ID, d >< ASSIGNOP, := >< ID, d
>< ADDOP, - >< NUM, 1 >< SEMICOLON, ; >< ARRAY >< BRACKET, [ >< NUM, 10 >< BRACKET, ] ><
COLON, : >< ID, hello >< SEMICOLON, ; >< PROGRAM >< BEGIN >< NOT >< ELSE >< DO >< WHILE >< NUM,
888 >< SEMICOLON, ; >< NUM, 888 >
```

Figure 3: Token stream output file `token2.txt` for the input file, `in2.pas`