Assignment - 1: Lexical Analyser in Traditional Compilation Process

Kaustav Goswami*, Chandan Kumar‘
kaustavgoswami.2013@gmail.com, cchaudhary278@gmail.com
Computer Science Engineering Department
Indian Institute of Information Technology Guwahati

I. Introduction

The DFA based approach for Lexical Analysis in C++ can be accurately modeled as a separate program. The accuracy and correctness of the program is strictly dependent on the extensiveness of the code written.

In this report we have presented, we discuss the design that we've proposed into a package of programs which modularly generates tokens from a C++ program and analyse the same. The modular approach is choosen in order to provide a better understanding of the working of a typical lexical analyser.

First task of the program is to iterate through a given C++ code and generate tokens. The second part is to analyse the given tokens with the assistance of a *symbol table* and finally to put it in a output file.

II. Algorithm

Following segment of code represents the working of both token generation as well as parsing of the same by the program.

Algorithm 1 Parser

```
1: tokens = (string)in_file.split("")
2: while tokens do
3:     if tokens in keywords then
4:         fileWrite (out_file, "token,keyword")
5:     else if tokens in operators then
6:         fileWrite (out_file, "token,operator")
7:     else if tokens in brackets then
8:         fileWrite (out_file, "token,brackets") ‘
9:     else if tokens in headers then
10:        fileWrite (out_file, "token,headers")
11:    else
12:        if tokenStructure (tokens) == float then
13:            fileWrite (out_file, "token,brackets")
14:        else if checkVariable (tokens) then
15:            fileWrite (out_file, "token,variable")
16:        else if checkStructure (tokens) == string then
17:            fileWrite (out_file, "token,string")
18:        else
19:            fileWrite (out_file, "token,integer")
20:        end if
21:    end if
22:    if tokens.contains (";") or token.contains (",") then
23:        fileWrite("out_file,
   tokens[tokens.length()],delimiter")
24:    end if
25: end while
```

A. Analysis of Algorithm 1

The complexity of generating the tokens is clearly $O(n)$. No further optimizations are possible for the generating method.

While there are valid number of tokens incoming, we will check the given token in DFAs designed as $keywords$; containing all keywords of C/C++, $operators$; containing all valid list of operators, $brackets$; for brackets, $headers$ for all valid set of header files. The error state of these DFAs ends up in a list of checker methods which essentially checks the structure of a given token. This results in accurate detection of various datatypes and delimiters. The space complexity for processing all the tokens is exactly the combined size of all the tokens.

The time complexity for the parsing part is mainly dependent upon data structures used for implementing the DFAs.

The data structure we've used is a string trie. The advantage of using a trie is that it models an exact DFA and had a search complexity of $O(TokenLength)$. Also the space complexity of a trie is very efficient as the there are no repeatations for sequential series of repeated characters. For example, if A = "std" and B = "stdio" then the total space used in this case are $5 + 2$ (2 for '\0' characters) memory locations instead of 10 characters in most common data structures.

This results in a time complexity of $O(Numberoftokens + TokenLength)$.

III. Implementation

For implementation purpose, in addition to the above, we've used another method which is basically a raw form of token generator which formats the input strings according to our needs.

A. Analysis for Algorithm 2

Algorithmically, the raw version of token generator has a complexity of the total number of characters. These tokens are again iterated over in order to properly split out all the

For C/C++ programs
*Roll Number 1601028‘Roll Number 1601015

Algorithm 2 Raw Token Generator

```
1: while lines in input_file do
2:     words = string.split (lines, " ")
3:     if words.contains ("//") or words.contains ("/*") then
4:         Discard Lines Till "*/" is found
5:     else
6:         out_file.write (word)
7:     end if
8: end while
```

tokens for correctness. This overhead seems unnecessary but is implemented to increase robustness, correctness and to provide a better understanding of the token generation procedure. The actual token generator cost is $O(n)$ where n is the number of characters in the given C/C++ file. Implementation overhead is additional with the previous analysis. This method however is responsible for comment deductions.

B. Symbol Table

The implemented symbol table is a 1D string vector. Whenever a call is made to the symbol table, it first checks for the presence of the variable in the list (Costly only in worst case). If found, the corresponding details of the variable is extracted out. Else, a new entry is created in the table.

Insertion into this vector is $O(1)$ and search complexity os $O(n)$ where n is length of the string formatted as *Datatype,Variable Name,Size*.

Also we have assumed the size of variables as 4 bytes for integer and void, 8 for float and double, 1 for bool, string and char types.

IV. Installation

The program distribution comes in a *tar* archive which contains
a C++ file,
a Python Script, and
a Shell Script to build and launch the program.
The dependencies for this program are python2.7 and g++ with C++11 standard.

Use *./launcher.sh [filename]* in unix to build the program as well as to run the same. Please note that the program can only be launched with the shell script provided.

Please use *chmod +x launcher.sh* in case of denial of permission.

V. Additional Notes

The code we've written has a few points to keep in mind.

Tokens are being fed into the system sequentially in a stream. This logically is a queue of tokens which serves our purpose without going for an unnecessary queue maintenance, to implement the same, as it would have increased the complexity more. Superficial analysis of the same would say that the complexity of using an additional queue would be same as the implementation method we've used.

The trie data structure that we have used is to our believe is one of the most efficient data structure specifically for insertion, search as well as for avoiding repeatations. Hence we would also like to point that our method is properly optimized in terms of efficiency of both space as well as time complexities.

Another thing is that a string trie does not enable us to insert "/" into the trie. This forced us to manually handle this case.

Also we would like to mention about handling of all the delimiting characters. We are not assuming ';' is preceeded by a blank space. This made us to check for ';' in all *sentence ending* cases. This is another manually checking part of the program which ensures correctness with the flexibility to use blank spaces before ';'. Please note that a blank space would also detect the same without any issues.

The output of the program is a list of all the tokens redirected to a file and also displaying the same with parsing information on the console screen along with the symbol table.

An extended feature of our design is that this program checks for brackets mismatch.

VI. Limitations

This implementation works correctly as well as robustly but it has its own limitations too. The entire database for all keywords, header files, operators etc. present in C/C++ is not entered but has complete provision to do so.

Another limitation is that certain sequence of valid C/C++ programs, suppose the use of scope, might fail to get detected in this program in its current stage. However it has a scope for further development.

VII. Conclusion

We've successfully designed a program which can be termed as a primitive version of a Lexical Analyser with robust and correct results with an optimized method for searching. There are certainly a few limitations but for most common cases, our design works flawlessly.