

# Lexical Analyzer Generator

Ahmed Yasser n: 10

Pierre Maged n: 22

Hossam Elkordi n: 24

Mina Ashraf n: 65

## Overview

It is required to implement a lexical analyzer generator tool that takes the grammar rules of a language and builds a minimized deterministic finite automata from them, then uses the resulting machine to parse tokens from an input file that follow the provided rules.

## Methodology

First the program reads the grammar file with path provided as a parameter to the main function or the default path "[grammar.txt](#)" if the user didn't specify one. The file contains 4 categories of rules (regular definitions, regular expression, punctuations and keywords). After parsing the language's rules, a map of regular expressions is produced that is used to create a non-deterministic finite automata which will be transformed to a deterministic one then it is minimized. The resulting automata is then used to parse tokens from the input file with path either provided as parameter to the main file or the default one is used "[input.txt](#)" using the maximal much techniques discussed in lectures. The resulting tokens and transition table of the automata along with their accepting states are the printed out to files.

## Classes

### InputParser

This class encapsulates the reading of the grammar file and generates a map of regular expressions with their priorities and a list for keywords and another one for the punctuations. The main function used in this class is `readFile()` which reads the file line by line and parses it depending on the pattern it takes

### Nfa

This class takes its input from the input parser class that takes its input from the grammar.txt file and creates a non deterministic automata that only accepts the inputs specified by the grammar.

### Dfa

This class produces the deterministic finite automata from the resulted nfa creating its transition table and a map of its accepting states. The main function in this class is `createDfa()` which implements the subset construction algorithm discussed in the lectures using two other helping functions:

- `epsilonClosure(list<int> u)`: This function adds all destination states under epsilon transition from each source state in the list.
- `move(list<int> t, string input, list<int> u)`: This function fills the list (u) with all destination states of transition from each state in (t) under input (input) which is one character, along with their epsilon closures.

## MinimizedDfa

Using the Equivalence Theorem, we minimize the DFA by:

- dividing the set of states into two sets: accepting and non accepting states, this partition is called  $P_0$
- we need to find  $P_k$  by partitioning the different sets of  $P_{k-1}$ . In each set of  $P_{k-1}$ , we will take all possible pairs of states. If two states of a set are distinguishable, we will split the sets into different sets in  $P_k$
- we stop when there is no change in partition ( $P_k = P_{k-1}$ )

Here is  $P_0$  partition, as  $P_0$  is divided into non-accepting states and accepting states, where the accepting states are represented in sets, each set contains accepting states for the same definition.

```
set<set<int>> finalStates;
set<set<int>> remainingStates;
set<string> s;
set<string>::iterator it;
set<set<set<int>>> temp;
for (itr = graph.begin(); itr != graph.end(); ++itr){
    if(dfaAccepted.find(itr->first) != dfaAccepted.end()){
        s.insert(dfaAccepted[itr->first]);
    }else{
        remainingStates.insert(itr->first);
    }
}
for (it = s.begin(); it != s.end(); ++it){
    set<set<int>> l;
    for (itr = graph.begin(); itr != graph.end(); ++itr){
        if(dfaAccepted.find(itr->first) != dfaAccepted.end() && dfaAccepted[itr->first]==(*it)){
            l.insert(itr->first);
        }
    }
    temp.insert(l);
}
```

Array **groups** is used to represent the mapping of the initial transitions to the groups of the previous partition


```
while(!finish){
    int groups[graph.size()][alphabet.size()];
    for(i=0;i<graph.size();++i){
        for(int j=0;j<alphabet.size();++j){
            groups[i][j]=-1;
        }
    }
    for (sss = partitions[p].begin(); sss != partitions[p].end(); ++sss){
        ++k;
        for(i=0;i<graph.size();++i){
            for(int j=0;j<alphabet.size();++j){
                if((*sss).find(dfaTable[i][j]) != (*sss).end()){
                    groups[i][j]=k;
                }
            }
        }
    }
}
```

Here we compose the next partition:

```
for (ss = (*sss).begin(); ss != (*sss).end(); ++ss){
    if(temporary.empty()){
        vector<set<int>> v;
        v.push_back(*ss);
        temporary.push_back(v);
    }else{
        for(i=0;i<temporary.size();++i){
            for(int x=0;x<alphabet.size();++x){
                if(groups[states[temporary[i][0]]][x] != groups[states[*ss]][x]){
                    identical=false;
                    break;
                }
            }
            if(identical){
                temporary[i].push_back(*ss);
                break;
            }
        }
        if(!identical){
            identical=true;
            vector<set<int>> v;
            v.push_back(*ss);
            temporary.push_back(v);
        }
    }
}
```

Then at the end of the while loop if  $P_k = P_{k-1}$  we will stop iterating

```
if(p!=0 && partitions[p]==partitions[p-1]){
    finish=true;
}
```



Then we update the **start** state, the **dfaAccepted** map and the **graph** map by removing the repeated states, and by updating each existence of the removed states by its new equivalent.

## InputLanguageParser

After having the minimized DFA, we need to parse the code using maximal munch algorithm. Given starting state, graph and token types; we parse the input file line by line, iterating through each line character by character. We follow the graph till arriving at an accepting state and carry on. If we find a new one the old one is deleted, otherwise we do a backtracking technique saving the last accepted state and then we start from the next character. All tokens are stored in a vector.

## OutputGenerator

This class prints out the transition table of the minimized dfa along with its accepting states. It also has a function to print out the parsed tokens.

## Token

A simple class to handle the tokens. It has 2 attributes type and lexeme, which are filled only through the constructor and getters as methods

## Data Structures

- `map<string, string>`: Used to provide regexes and their types to Nfa class.
- `map<string, int>`: identify priority value for each regex type.
- `list<string>`: Used to store the keywords and punctuations.
- `int start` that indicates the start state almost every time it is set to 1
- `int end state` it says the last state of the nfa
- `int number of states` it indicates the number of states in the nfa
- `vector<int>` accepting this is a vector that contains all the accepting states
- `map<int, map<string, vector<int>>>` this is the most important structure in the Nfa it contains all the transition in the following manner indexing the map with the the number of state returns a map that has as input a string that says what states to go to if the nfa was at this state and received the string
- `map<int, string>` tags that has all the tags for the accepting states
- `map<set<int>, map<string, set<int>>>`: Representation of the transition table of the dfa after applying the subset construction algorithm on the nfa.
- `map<set<int>, string>`: Identify the accepting states of the dfa and what type it accepts.
- `set<int>`: Represents a state in the dfa as a subset of the nfa states.
- `vector<Token>`: A vector that contains all the parsed tokens from the input file.

## Test Case

Input grammar file:

```
letter = a-z | A-Z
digit = 0 - 9
id: letter (letter|digit)*
digits = digit+
{boolean int float}
num: digit+ | digit+ . digits ( \L | E digits)
relop: \= \= | !\= | > | >\= | < | <\=
assign: =
{ if else while }
[; , \ ( \) { } ]
addop: \+ | -
mulop: \* | /
```

Minimized DFA transition table:

[https://drive.google.com/file/d/1sxSl9z1hg1JY4UdNeWu9uA\\_IAlYB-cwb/view?usp=sharing](https://drive.google.com/file/d/1sxSl9z1hg1JY4UdNeWu9uA_IAlYB-cwb/view?usp=sharing)

Accepting States:

[https://drive.google.com/file/d/1nHzvmOb8egjV5Je6F7pHxYnk6SL\\_4TzB/view?usp=sharing](https://drive.google.com/file/d/1nHzvmOb8egjV5Je6F7pHxYnk6SL_4TzB/view?usp=sharing)

Input program to test:

```
int sum , count , pass , mnt; while (pass !=
10)
{
pass = pass + 1 ;
}
```



Output file:

```
int      int
id       sum
'        '
id       count
'        '
id       pass
'        '
id       mnt
;        ;
while    while
(        (
id       pass
relop    !=
num      10
)        )
{        {
id       pass
assign   =
id       pass
addop    +
num      1
;        ;
}        }
```

### Assumptions:

The '\$' is reserved for the epsilon transition in the Nfa and input parser thus cannot be used as input.

## Automatic Lexical Analyzer Generator Using FLEX

The following steps are made on ubuntu system.

### Steps:

- Install flex from terminal using command:  
\$ Sudo apt-get install flex
- Flex takes as input a (.l) file which contains the lexical analyzer to be generated.
- The structure of the input file:
  - A. **Definition Section:** it is the part that contains the regular definitions and variable declarations. It is enclosed between curly brackets  

```
%{
% }
```
  - B. **Rules Section:** It is the part where the we define the regular expressions, It follows this pattern:  

```
%%
Pattern      {Action}
%%
```
  - C. **Code Section:** it contains the c user code and functions.
- The input file passes through the lex compiler and produces a (.c) file ready to be compiled by C compiler GCC. The (.c) file has a copy of the definition section and the flex uses a function named yylex() to run the rules section. Use this command to produce the (.c) file:  
\$ lex filename.l
- Compile the (.c) file named **lex.yy.c** to produce the executable file named **./a.out**, use this command:  
\$ gcc lex.yy.c

- Run the Executable File. Use this command:

\$/a.out

### Example:

We used a code that reads a file and counts the number of small letters, capital letters and number of lines, where the `yywrap()` function wraps the rule section and `yyin()` takes the input file pointer.

- grammar.l file

```
%{
int capCount = 0;
int smallCount = 0;
int numLines = 0;
}%

%%
[A-Z] {capCount++;}
[a-z] {smallCount++;}
\n {numLines++;}
%%
int yywrap(){}
int main(){
FILE *fp;
char filename[50];
printf("Enter the filename: \n");
scanf("%s",filename);
fp = fopen(filename,"r");
yyin = fp;
yylex();
printf("\nNumber of Captial letters %d\nNumber of Small letters - %d\n"
      "Number of Lines in file - %d\n", capCount,smallCount,numLines);
return 0;
}
```

- The test file being read:

```
Frame Table
First in load segment all allocate page and free page are replaced with equivalent ones in frames
FrameMap is a hashtable where the key is the physical address each entry contains the physical and virtual address and the thread that owns
the frame
We use palloc get page when creating it returns the virtual address then create a frame table entry for it
We use lock to insert the entry in eth frame map
Free we recieve the virtual address and create a temporary entry for it we store in it the physical after converting and search to find
similar physical address once we get it we store it and delete the temporary then delete the page and the original entry we want to delete
If it is not found we PANIC
```

- The Following commands shows the steps to run the program and the output:

```
omid@omid-Latitude-E5430-non-vPro:~$ cd Desktop
omid@omid-Latitude-E5430-non-vPro:~/Desktop$ lex grammar.l
omid@omid-Latitude-E5430-non-vPro:~/Desktop$ gcc lex.yy.c
omid@omid-Latitude-E5430-non-vPro:~/Desktop$ ./a.out
Enter the filename:
testFile

Number of Captial letters 14
Number of Small letters - 570
Number of Lines in file - 10
omid@omid-Latitude-E5430-non-vPro:~/Desktop$ |
```