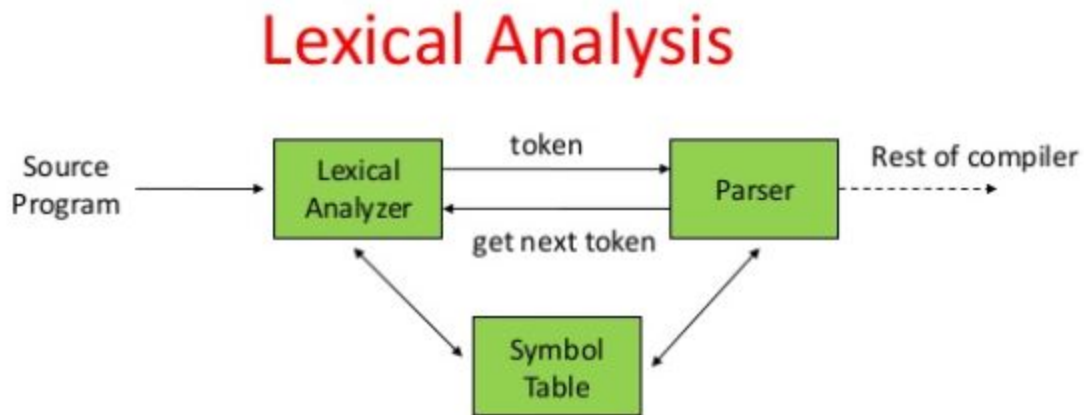


Phase1: Lexical Analyzer



Developed by:

1. Rita Samir 26
 2. Shereen El kordi 33
 3. Merit Victor 69
 4. Nancy Abdel Kareem 70
-

Introduction

This is our implementation of a lexical analyzer as the first phase of a compiler developed mainly for java bytecode and it's more general and can be extended for other languages as long as the grammar given to the lexical analyzer follows the same format defined.

Used Data Structures

We defined the following class models:

1. AcceptedState.h

Represents the accepted state which contains the final state number and its token type that it accepts.

2. Token.h

Represents the token returned by lexical analyzer which contains the lexeme value and token type.

3. NFA.h

Represents the structure of the NFA including its transitions, final state, total number of states and the token type associated with the NFA.

4. UnionedNFA.h

Represents the structure of the final NFA to be converted to DFA. It has almost the same structure as any ordinary NFA except that it has may final states.

The difference between the NFA and UnionedNFA modules should become clearer when the NFA parsing algorithm is explained.

We used the following data structures:

1. **map<int,map<string,int>>**

Used for representing DFA Transition table. The key of the first map is the **fromState** and its value is a map that represents all transitions that the fromState makes on each symbol. The key of the second map is the **input** and its value is the **toState** that the fromState goes to on this input.

2. **vector<AcceptedState>**

Used to contain all the **final/accepted states**. Each accepted state is a class that attributes the final state number and its token type that it accepts.

3. **Queue**

Used in DFA algorithm to get the transitions between states in BFS path.

4. **Set**

Used to remove duplicated states.

5. **vector<vector<int>>**

Used to represent the equivalent states, as each vector considered as a state.

6. **Transition struct**

Represents the transitions of the NFA, and is considered the main structure that forms the NFA.

```
struct transition {  
  
    int fromState;  
  
    int toState;  
  
    string trans_symbol;  
  
};
```

7. map <string, string> regularExpressions;

The name of the regular expression vs its string value.

8. vector<string> keywords;

9. vector<string> punctuations;

Algorithms and Techniques Used

Lexical Rules Input File Parsing 's Algorithm

Implemented in LexicalRulesParser.cpp

It is simple string parsing and with the use of the built in regex library from std.

Mainly the algorithm takes each line from the rules file, compare it against some regex to detect the pattern specified in the assignment pdf, it then categorize it into one of four possibilities:

- 1) Regular definition
- 2) Regular expression
- 3) Keyword
- 4) Punctuation.

And it stored in the associated data structure mentioned early as they are global and will be used in the next stage (NFA parsing).

NFA's Algorithm (Thompson's construction)

Implemented in ReToNFA.cpp

The algorithm takes each regular expression, keyword and punctuation and parse it using Thompson's algorithm.

The algorithm takes each input symbol, convert it to NFA then using induction it applies on of the operations: Kleene closure, positive closure, concatenation or union. Finally it returns one NFA represents the regular expression.

Note that, those NFAs constructed for each regular expression are unionized in the final step constructing the UnionedNFA module that represents the grammar as whole.

DFA's Algorithm

Implemented in DFA.cpp

We used **BFS** algorithm in converting NFA to DFA to get all DFS states and its transitions.

Initially, we get the epsilon closure of the initial state then push it in the queue, mark it as checked to avoid processing it more than once. Each time, we get a state from the queue and pop it, get all the states that this state goes to on each symbol and combined them in one state, get its epsilon closure, check if this new state marked before if not, mark it and push it on the queue.

We stop when the queue is empty. So we generate the DFA transition table represented by `map<int, map<string, int>>` and generate accepted states.

Minimized DFA 's Algorithm

Implemented in Minimization.cpp

The algorithm used in minimization is the partitioning algorithm.

At first we divided states (set of all states) into number of sets. One set for each accepted state and other set will contain non-accepted states. This partition is called P_0 . Finding P_k by partitioning the different sets of P_{k-1} . In each set of P_{k-1} (K is integer starting at value 1), we took all possible pair of states. If two states of a set are distinguishable, we will split the sets into different sets in P_k . Stopping when $P_k = P_{k-1}$ (No change in partition). All states of one set are merged into one and also renaming some states to its equivalence states is necessary in the final minimized DFA. No. of states in minimized DFA will be equal to no. of sets in P_k . Two states are distinguishable in partition P_k if for any input symbol those states will go to different sets in partition P_{k-1} .

Maximal Munch's Algorithm

Implemented in LexcalOutput.cpp

Starting with scanning the first character of the input file, considering it as input and the from state will be the start state and getting the from state from it, we will have four cases:

- The input symbol doesn't exist in our set of inputs, in that case it may be a space or new line character so we will add new token according to last accepted state, initialize the from state to starting state and scanning next char until getting different one. Otherwise printing illegal symbol error message.
- The to state is dead state which means it stuck, in this case we add new token and initialize the from state to starting state.
- The to state is accepted state, we update the lexeme and to state, remembering that accepted state type and scan the next character.

-
- The last case is if the from state is intermediate state, we keep tracking the lexeme, update from state and scan the next character.

Looping through those four conditions until we reach the end of the input file.

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: * | /

The resultant transition table for the minimal DFA

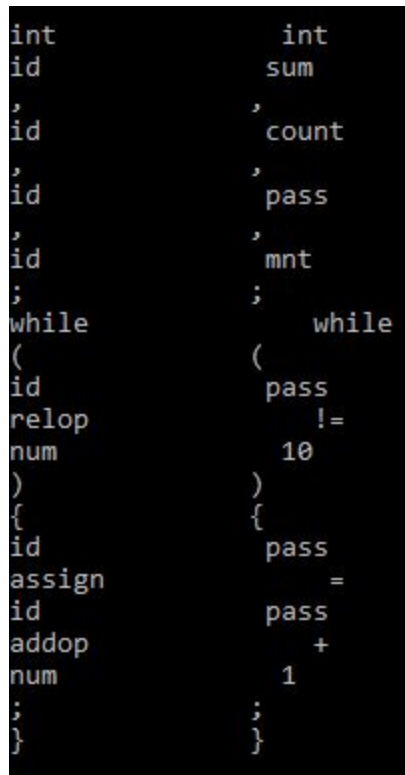
This is a link for the minimized dfa table for the above input grammar file:

<https://drive.google.com/file/d/17Ad8JrZgMTe3Cx0p79fBxwOBkwsI9eCi/view?usp=sharing>

Test program

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

The resultant stream of tokens for the example test program



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

- 1) **Keywords and punctuations are of higher precedence than the regular expressions as they are reserved symbols/words.**
- 2) **The lexical Rules input form should be written to provide the closest form of standard regex for example:**

- Each grouping of symbols for some operation should be included between parenthesis ()

For example:

num: digit+ | (digit+ . digits (\L | (E digits)))

That is because we parse the RE into NFA using some variation in the traditional **infix evaluation algorithm** and the stacks used require parenthesis to detect which symbols should be grouped together for some applied operation.

- letter = [a-z A-Z] digit = [0 - 9]

Are written to be easily parsed to find the associated letters of the group as follows:

```
for (int j = 0; j < 128; ++j) {  
  
    if(regex_match(string(1, (char)j), reg1)) {  
  
        replacement += string(1, (char)j) + "|";  
  
    }  
  
}
```

We check the ANCI code against the group and extract the letters that match to be ored result in the expression: (0|1|2|3|4|5|6|7|8|9) which is easily evaluated in the NFA parsing.

Applying those assumptions results in:

Token Type : String Value

boolean : boolean int : int float : float

if : if else : else while : while

::: ,: , (: \ () : \) { : { } : }

addop : \+|-

assign : =

id :

(A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z)((A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z)|(0|1|2|3|4|5|6|7|8|9))*

mulop : *|/

num :

(0|1|2|3|4|5|6|7|8|9)+|((0|1|2|3|4|5|6|7|8|9)+.(0|1|2|3|4|5|6|7|8|9)+(\L|(E(0|1|2|3|4|5|6|7|8|9)+)))

relop : (==|!=|>|>=|<|<=)

Bouns

INTRODUCTION

It is required to describe the steps for using a tool to automatically generate a lexical analyzer for the given regular expression. We used Flex on a Ubuntu based system.

Steps

1. Install flex on your system from the terminal using this command:
`$ sudo apt-get install flex`
2. 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 these 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.
3. The input file passes through the lex compiler and produce 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 functions named `yylex()` to run the rules section.
Use this command to produce the .c file:
`$ lex filename.l`
 4. Compile the .c file named `lex.yy.c` to produce the executable file named `./a.out`
Use this command:
`$ gcc lex.yy.c`
 5. Run the Executable File. Use this command:
`$./a.out`

Example:

1. We used a code that reads a file and counts the number of small letters, capital letters and number of lines.

Note:

The yywrap() function wraps the rule section and yyin() takes the input file pointer.

```
*grammar.l x
%{
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;
}
```

-
2. The Following is the test file being read..

```
testFile x
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 pallocc 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
```

3. The Following commands shows the steps to run the program and the output.

```
Terminal
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$ |
```