LEXICAL-ANALYZER-GENERATOR

(Compiler)

Mohamed Essam (64)
Mohamed Murad(66)
Mohamed Raafat (62)
Wessam Mohamed (81)

Project Report

TABLE OF CONTENTS

1] USED DATA STRUCTURE **..** ERROR! BOOKMARK NOT DEFINED.

2] ERROR! BOOKMARK NOT DEFINED.

ERROR! BOOKMARK NOT DEFINED.

ERROR! BOOKMARK NOT DEFINED.

ERROR! BOOKMARK NOT DEFINED.

1] USED DATA STRUCTURE

1] Rules Parser:

String and Vector is used in reading Rules file and parsing them

First: Rules File is read line by line, each line is read as a string and pushed in a vector. at the end of this stage you can get a vector of string which represents all Rules "lines"

Second: There exists a class called Rule, which contain 4 vector of vector of strings and a 5th vector of integer called order
each one of the 4 vector of string represent the type which this rule belong to Keywords, punctuation, definition and Expression.

after detecting type of each rule, it's parsed into tokens, each token is a string, and each token pushed into a vector of that rule line and after completing the rule line, this vector is pushed back to the vector of its type which is determined before.

2] Infix to Postfix Handler:

Stack is used in handling both the conversion of expressions from infix to postfix and the evaluation of the postfix form of expressions.

The main reason to use the stack is to handle the priority of *Regular Expressions Operations* while converting and evaluating the expressions.

3] NFA Builder:

Node which is used to store the edges from that node to any other node and has the priority of the node and the token that would be given if we reached this state.

Edge which is line connecting the first node with the second one and it has the condition for which this transition would apply if that condition has been met i.e. you need to have input “;” to go to the next state.

Graph which is a wrapper for all those above data structures and it is defined by its first and its last node in the graph we also define its size in the graph in order to convert the graph to adjacency list.

4] DFA Builder:

Transition table that would save the transition diagram and the list of the states that are in that table

State which save the token for the state if found and its priority of that state if it’s valid or invalid state.

5] Minimizer:

Unordered map, arrays, set and vectors is used in Minimization.

It's used Unordered map of integer as a key to refer to the number labeled to this state and an unordered map of char key as an input which determine which state we move as an Integer as a value.
this is the table of deterministic finite automata.

we use array of char to know all inputs we can move by them

set of integers is used with iterator to get the nodes of the group which it is represented as a set of integers.

we also use vector of char to prepare the array of used char which is the needed inputs

2] ALGORITHMS AND TECHNIQUES USED

<u>1] Infix to Postfix Handler:</u>

First (Infix to Postfix Converter) *"According to Geeks for Geeks"*:

1- Scan the infix expression from left to right.
2- If the scanned character is an operand, output it.
3- Else,
   …3.1 If the precedence of the scanned operator is greater than the precedence of the operator in the stack (or the stack is empty), push it.
   …3.2 Else, Pop the operator from the stack until the precedence of the scanned operator is less-equal to the precedence of the operator residing on the top of the stack. Push the scanned operator to the stack.
4- If the scanned character is an '(', push it to the stack.
5- If the scanned character is an ')', pop and output from the stack until an '(' is encountered.
6- Repeat steps 2-6 until infix expression is scanned.
7- Pop and output from the stack until it is not empty.

Second (Postfix Expression Evaluator) *"According to Geeks for Geeks"*:

1- Create a stack to store operands (or values).
2- Scan the given expression and do following for every scanned element.
   …2.1 If the element is an Operand, push it into the stack
   …2.2 If the element is an operator, pop operands for the operator from stack. Evaluate the operator and push the result back to the stack
3- When the expression is ended, the value in the stack is the final answer.

<u>2] NFA Builder:</u>

We used Thomson rules for construction of the NFA as in the lectures we support concatenation, or, star closure and positive closure.

Concatenation is done by just connecting the last node of the first graph by the first node of the second graph and return new graph whose first node is the first node in the first graph and the last node is the last node in the second graph with lambda transition.

Or is build by adding 2 new nodes and connect the first one with the first of each node with lambda transition and the last node of each graph with the second node with also lambda transition then we make new graph with first and last node are the new added nodes.

Star closure is build with adding also 2 nodes one for the first and the other for the last and connect the first with the start node of the graph with lambda transition and the last node of the graph with the second added node with lambda transition then we connect the first new-added node with the last new-add node with lambda transition, and the last node of the graph with the first one of the graph with lambda transition.

Positive closure is build like the star closure except that the transition from the new added node and the last added node is removed.

After building each expression, punctuation, and keyword we assemble all the graphs by adding start node and connect it will all other graphs with lambda transition and we return that start node.

3] DFA Builder (Subset builder):

We start from what the NFA builder finished by taking the start node from the assembling and make BFS from that node and get the adjacency list which contains numbers which is easier to deal with.

After that we start by making the lambda closure of the first node and push into a queue, so we would continue clustering the NFA and turn it to be a normal DFA. Each time we find new closure we push it into the queue and get its next states. While looping we ensure that this closure has not appeared before as if we didn’t make this check we would be in an infinite loop.

After clustering the nodes in closures, we give them numbers as normal states and define their transitions and states then put all this is transition table and define the number of the first state.

4] Minimization:

algorithm used in minimization is different a bit of that used in lecture.
to find minimized table we work as following:

```
while there is not any division of group at the last iteration of next loop >>> loop again
    while there isn't any division at searching based on input c >>> loop again
        for each initial group g in the list of all groups
            Num_of_group  <<< find_group_number(first element)
            add first element in group g1
            for each element e in group g
                if num of group of e is  Num_of_group
                    add to g1
                else
                        add to g2
        replace g1 and g2 with g
```

building minimized table algorithm is as following:

```
for each group in g
        num_of_group <<< find_group_number(g)
        set a new state as num_of_group
                add_next_states()

add_next_states()
        for each next state of group g for any element
                add a next state with input c to-
                the number of group this next state belongs to it
```

3] THE RESULTANT TRANSITION TABLE FOR THE MINIMAL DFA

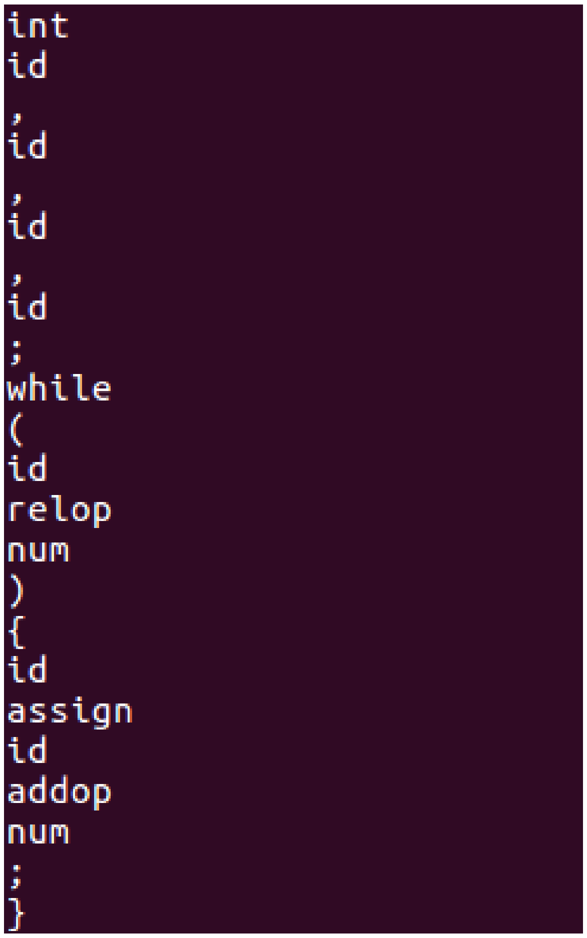
As the table is very long we put it a .txt file call table.txt with the report

The first line is the state count which is the number of states

Then its states and their tokens if the state has no token you will find blank area

Each state number and its possible inputs and the next state for each input.

4] THE RESULTANT STREAM OF TOKENS FOR THE EXAMPLE TEST PROGRAM



```
int
id
,
id
,
id
,
id
;
while
(
id
relop
num
)
{
id
assign
id
addop
num
;
}
```

5] ANY ASSUMPTIONS MADE AND THEIR JUSTIFICATION

We assumed that the keywords defined first then punctuation, then definitions then expressions which is not used in the LEX they define the ordering and the priority of the tokens by the order of the reading we made the 2 implementations, but we are using ours assumption as it’s logical that the keywords token would have higher priority than the expressions.

We assumed that we will not use the #, ?,% and the @ as we are using them in evaluating the regular expression.

6] BONUS – BUILDING LEXICAL ANALYZER USING FLEX

Flex (fast generator) is a alternative to . It is a that generates (also known as "scanners" or "lexers").

Steps:

1- Prepare **(lex_input.l)** file of type lex which contains all the detailed description of the input grammar of the language to be translated as shown in next .

2- After writing (.l) lex file compile it using the command `$lex lex_input.l`

3- The compilation will generate the file **(lex.yy.c)** which contains all the detailed code of the lexical analyzer generated.

4- Write **(scanner.c)** which takes input from the standard input to be sent to the lexical analyzer generated.

5- Compile the files **(lex.yy.c)** and **(scanner.c)** together and make the output is **(output.exe)** using the command `$ gcc scanner.c lex.yy.c -o output`

6- To test the lexical analyzer generated write a file **(config.in)** containing the source code to be translated using lexical analyzer .

7- Output is the matching translation of the lexical analyzer. If there is a token that not matching any of the expressions it will print **error**

Source Code:

Source Code:

Lex_input.l:

```
%%
"boolean"                               printf("boolean\n");
"int"                                   printf("int\n");
"float"                                 printf("float\n");
"if"                                    printf("if\n");
"else"                                  printf("else\n");
"while"                                 printf("while\n");
[a-zA-z][a-zA-z0-9]*                    printf("id\n");
[0-9]+("."[0-9]+)?                      printf("num\n");
"="                                     printf("assign\n");
"=="|"<="|"<"|">="|">"|!=               printf("relop\n");
","                                     printf(",\n");
";"                                     printf(";\n");
"("                                     printf("(\n");
")"                                     printf(")\n");
"{"                                     printf("{\n");
"}"                                     printf("}\n");
[+-]                                    printf("addop\n");
[*/]                                    printf("mulop\n");
[ \t\n]                                 ;
.                                       printf("LEXICAL ERROR\n");
%%

int yywrap(void)
{
        return 1;
}
```

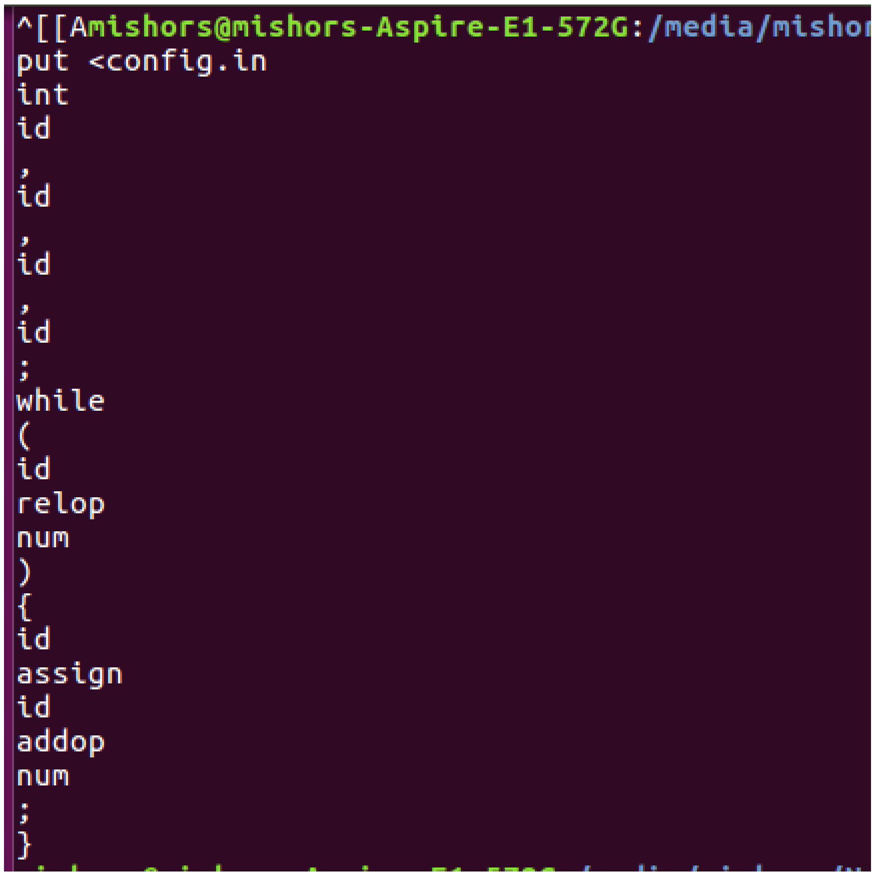
Scanner.c

```
#include <stdio.h>

extern int yylex();
extern int yylineno;
extern char* yytext;

int main(void)
{
        int ntoken;
        ntoken = yylex();
        while(ntoken)
        {
                ntoken = yylex();
        }
        return 0;
}
```

Sample Input:

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

Sample Output:



```
^[[Amishors@mishors-Aspire-E1-572G:/media/mishor
put <config.in
int
id

,
id

,
id

,
id
;
while
(
id
relop
num
)
{
id
assign
id
addop
num
;
}
```