Lexical-Analyzer-Generator

(Compiler)



Project Report

Mohamed Essam (64)

Mohamed Murad(66)

Mohamed Raafat (62)

Wessam Mohamed (81)

Table of contents

[1] Used data structure](#_Toc509262469)  [3](#_Toc509262471)

2]  [Explanation of all algorithms and techniques used: 4](#_Toc509262474)

[3] The resultant transition table for the minimal DFA: 5](#_Toc509262478)

[4] The resultant stream of tokens for the example test program: 6](#_Toc509262478)

[5] Any assumptions made and their justification: 5](#_Toc509262478)

**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 also 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] 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 integer is used with iterator to get the nodes of the group which it is  
represented as a set of integer.  
  
we also use vector of char to prepare the array of used char which is the needed inputs  
  
  
  
  
**2] algorithms   
and techniques used**

1] Infix to Postfix Handler:

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

2] 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**

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

**5] ANY ASSUMPTIONS MADE AND THEIR JUSTIFICATION**

**6] BONUS – BUILDING LEXICAL ANALYZER USING FLEX**

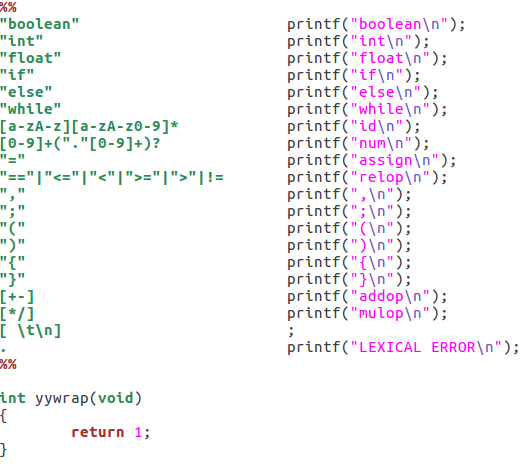
**Flex** (fast [lexical analyzer](https://en.wikipedia.org/wiki/Lexical_analyzer) generator) is a [free and open-source software](https://en.wikipedia.org/wiki/Free_and_open-source_software) alternative to [**Lex**](https://en.wikipedia.org/wiki/Lex_programming_tool). It is a [computer program](https://en.wikipedia.org/wiki/Computer_program) that generates [lexical analyzers](https://en.wikipedia.org/wiki/Lexical_analysis) (also known as "scanners" or "lexers").

Steps:

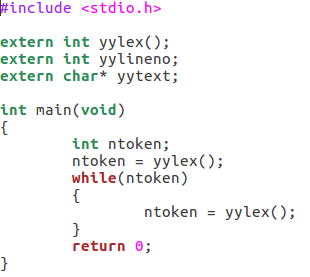
1. Prepare **(lex\_input.l)** file of type lex which contains all the detailed description of the input grammar of the language to be translated as shown in next .
2. After writing (.l) lex file compile it using the command $lex lex\_input.l
3. The compilation will generate the file **(lex.yy.c)** which 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:

Scanner.c



Sample Input:

int sum , count , pass ,

mnt; while (pass == 10)

{

pass = pass + 1 ;

}

Sample Output:

