Compiler

Lexical Analyzer

Phase 1

Salma Ahmed , 31

Shrouk Ashraf , 35

Mahmoud Hussein , 67

Moamen Raafat , 51

24th March,2019

Table of contents

I. Problem Statement

This phase task is to design and implement a lexical analyzer generator tool.

The lexical analyzer generator is required to automatically construct a lexical analyzer from a regular expression description of a set of tokens.

The tool is required to construct a nondeterministic finite automata (NFA) for the given regular expressions, combine these NFAs together with a new starting state, convert the resulting NFA to a DFA, minimize it and emit the transition table for the reduced DFA together with a lexical analyzer program that simulates the resulting DFA machine.

The generated lexical analyzer has to read its input one character at a time, until it finds the longest prefix of the input, which matches one of the given regular expressions. It should create a symbol table and insert each identifier in the table. If more than one regular expression matches some longest prefix of the input, the lexical analyzer should break the tie in favor of the regular expression listed first in the regular specifications. If a match exists, the lexical analyzer should produce the token class and the attribute value.If none of the regular expressions matches any input prefix, an error recovery routine is to be called to print an error message and to continue looking for tokens.

II. Overall Design

The lexical analyzer consists of the following :

1. Parsers

- Grammar-parser
  a. ProductionParser which takes a set of productions rules and generate a postfix regex which can be used by the used by RegexToNfaConverter to generate the corresponding NFA. It returns a list of token each of which represents a certain regular expression. This part is isolated from the regex to NFA conversion so as to enable decoupling and modularity as every part of the project is responsible of a certain part.
  b. Properties: It is the same like the properties class in java it loads the format of the lexical rules file so that to be flexible and once the format of the file is changed, the project continues functioning according to the format specified in the properties file. This design is used such that to make the project as much as flexible as it can.
  c. Token: The token represents all the required data about a certain lexical rule such as the name of that rule, the postfix regex of that rule and the precedence of that rule.
- Code-parser which takes the java code file and deal with it as a stream of characters using some functions like getChar( ) and setStartIndex( ) in order to make it easier for the pattern matcher to deal with the stream of chars and retrieve a certain token.

2. Automata

While designing the structure of the graph we will be using, we tried to avoid redundancy in code between the NFA and DFA graphs, also the structure of the NFA node and DFA node has similarities, so we tried to define the common behaviour in the node and automata interfaces.

- Node
  a. Nfa Node which is a child class for the Node base class having some functions to deal with the NFA node like getTransitions( ) and

addTransitions( ).NFA node is considered the basic unit for constructing the NFA graph

   b. Dfa Node which is a child class for the Node base class having some functions to deal with the DFA node like getTransitions( ) and addTransitions( ).DFA node is considered the basic unit for constructing the DFA graph

- Automata
   a. NFA responsible for creation of NFA nodes either with certain acceptance states or ordinary intermediate nodes in order to build the NFA graph
   b. DFA responsible for creation of DFA nodes either with certain acceptance states or ordinary intermediate nodes in order to build the DFA graph
- RegexToNfaConvertor: It is responsible of converting a set of postfix regular expressions to its corresponding NFA depending on Thompson's construction algorithm.
- NfaToDfaConvertor which is responsible for converting the NFA to a DFA by applying subset construction on epsilon closures for each state.

3. **Dfa-minimizer**
   - DfaMinimzer which takes as an input a non minimized DFA , do the partitioning algorithm on it and output a new minimized DFA to be used later for the maximal munch.
4. **Pattern-matcher**
   - PatternMatcher which is responsible for analyzing a given code file and matching each token to a specific acceptance state , therefore specifying the class for this token in order to be saved to the analysis table . The pattern matcher performs the maximal munch algorithm on the given DFA
5. **File-writer**
   - FileWriter which is responsible for writing the required output to the file in a certain format.The file writer handles writing the minimized transition table,the Tokens and their classes and any errors.
6. **Utils**
   - SymbolTable which is used to store any identifier encountered in the source code in order to be used for the upcoming phase.It has an insertion function to perform this task.

     Noting that there is another analysis table which contains each token in the code and its class (e.g identifier , reserved word , relop....)

7. **Error**
   - ErrorHandler which has enums representing different errors and their corresponding error messages such as lexical error

III. Data Structures

1. A vector of tokens `std::vector<Token *> tokens` to store the postfix regex generated by the production parser.

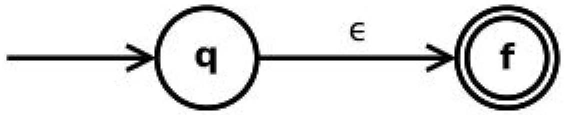
2. Vector of nodes representing the graph of an automata `std::vector<Node *> graph`.
3. A vector of state Ids to hold the transitions for a certain node `std::vector<std::vector<stateID>> transitions`
4. Analysis table which is a `std::multimap<std::string, std::string>` in order to keep track of every token and its token class once recognized by the pattern matcher.
5. Symbol table which is `std::vector<std::string>` to keep track of all identifiers.

IV. Algorithms and Techniques

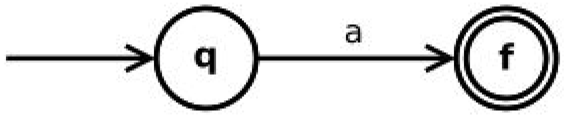
1. Converting regular expressions to NFA

The algorithm used to convert from regex to NFA is Thompson's algorithm. The algorithm consists of 4 main rules to build the NFA:

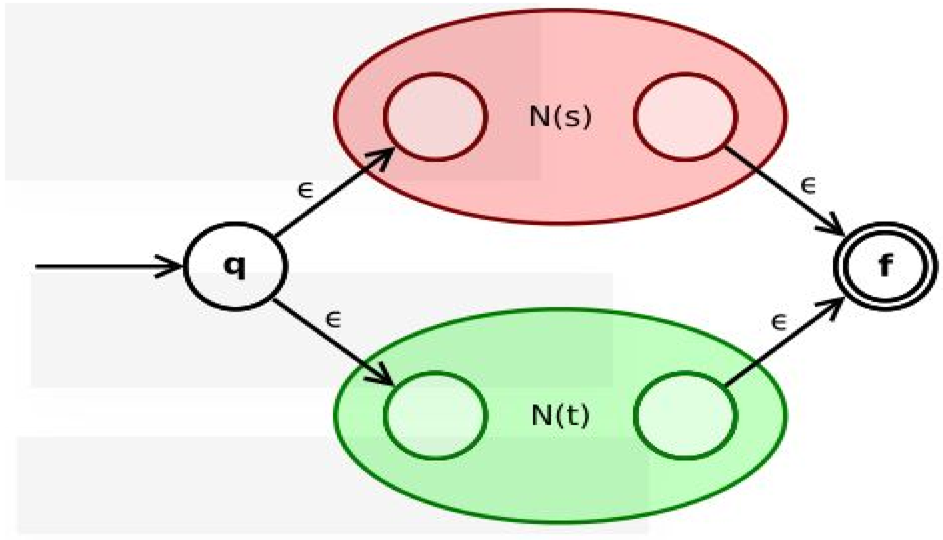
- The **empty-expression** ε is converted to



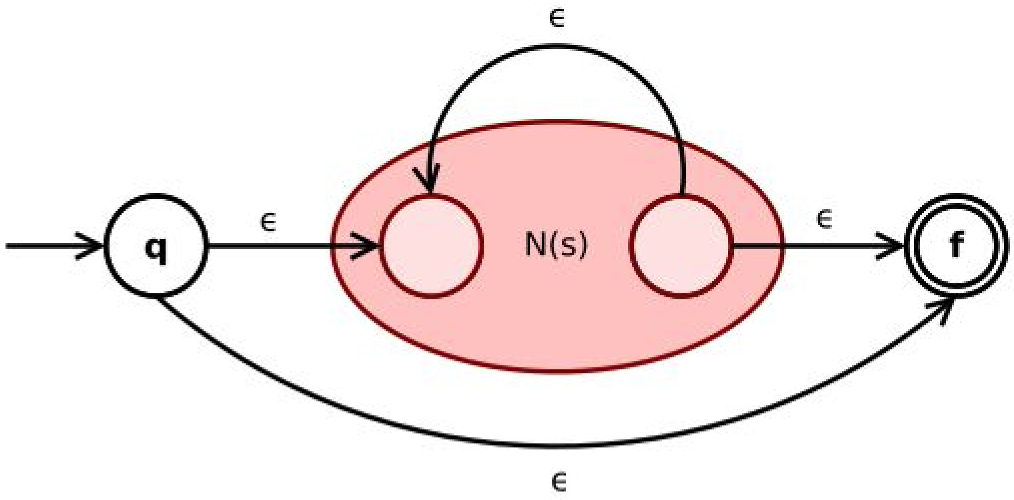
- A **symbol** *a* of the input alphabet is converted to



- The **union expression** *s*|*t* is converted to



- The **Kleene star expression** *s** is converted to



- The plus operator will be the same like the star but without the first node q.

2. Converting NFA to DFA

The algorithm used is the subset construction, first the epsilon transition table is constructed for each node in the NFA graph, then a queue is used for implementing the algorithm, and a map where the key is a subset of NFA states and value is the stateID in the constructed DFA. first the start epsilon is pushed to the queue then each time a set of NFA states is popped and iterate over all the characters in the language then check if the set of the new NFA next states already has been identified as a state or not by checking the map, if no, create node for it and push it in the queue and the map. If yes. retrieve the key and add transition between the current node and this node directly.

3. DFA minimization

Before talking about the algorithm it is necessary to mention some data structures that are used in it.

- std::vector<std::set<stateID >> *prev = &sets1;
- std::vector<std::set<stateID>> *next = &sets2;

  Two pointers to vector of sets that will keep alternating in carrying the partitions.

- std::map<stateID, **int**> *stateToSetId;

  A map that will keep track of the place of every stateID in the sets vector.

The algorithm used in minimizing the DFA is divided into 3 steps:-

1. Initial Partitioning
   This step is like a preparation for the minimization algorithm. Its job is to partition the states to different sets depending in their type (Intermediate, Accepted, PHI) and on their tokens (id, num, ...).
2. Iterative steps
   The second step is an iterative step that keeps executing the same steps till a condition is true.
   The condition is that the “prev” and “next” vectors must have the same size after the iterative cycle finishes which indicates that the minimum dfa is reached.
   These steps are:
   a. Clear the “next” vector so the new generated partitions can be inserted to it.
   b. Perform the partitioning algorithm and add the new generated sets to the “next” vector.
      The partitioning algorithm is:
      i. Hold the first element in the set and erase it from the set.
      ii. Erase that element from the set and create a new set in the “next” vector.
      iii. Find as many element in that set that are equivalent to it (2 elements are equivalent if they go to the same state for the same transitions for all possible transitions)
      iv. For the remaining elements go to step i.
      v. Stop when this set is empty.
   c. Update the “stateToSetId” map after having the new partitions

d. Swap the “prev” and the “next” pointers so they are ready for the next iteration.

3. Minimum DFA Building

The Minimum DFA graph is to be built. During this step a node will be created to represent the corresponding set that it will represent. The root node is to be put at index zero which is the assumption that is used in the Maximal Munch step.

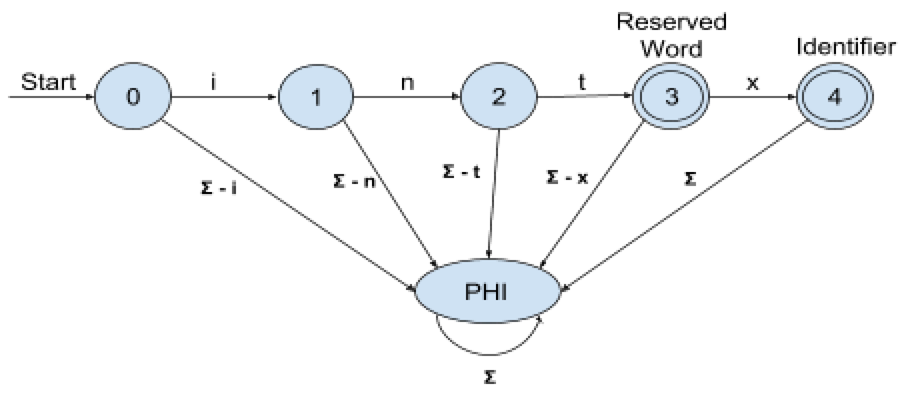
4. Maximal Munch

The algorithm applied for pattern matching in order to match a token with a certain acceptance state is the *maximal munch* where maximal munch is done on the minimized DFA starting from the root node in order to find the longest match.A token is considered a match if we reach an acceptance state .Once a delimiter is encountered or *PHI* state is reached , the found match and its token class are stored in the *analysis table* and then we redo trying to find a match from the DFA start .This process continues until there are no more characters in the stream.

An error recovery routine is called whenever a match can not be found in order to report the lexical error to the user , advance the pointer of the stream by one character then continue finding matches.This error handling is considered to be robust and efficient where the program does not stop after encountering an error.

Noting that When using DFA as a scanner , the DFA is considered stuck when it goes to PHI state where the match with the longest length is recorded along with the token type it matches.

Example :

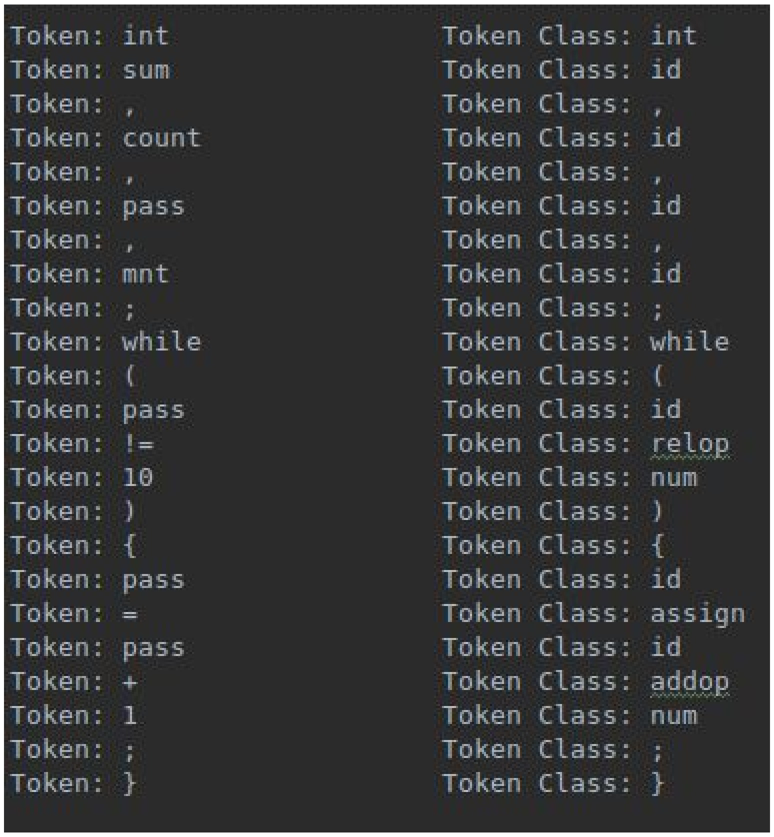


If the input to this DFA was intx then the output should be of type identifier and the matched token is intx since intx is a match longer than int according to maximal munch.

V. The Resultant Transition Table For The Minimal DFA.

The transition table of the minimal DFA after running the test program can be found here

VI. The Resultant Stream Of Tokens For the Example Test Program



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

VII. Assumptions

1. In order to construct a complete DFA if a state does not reach any other state through a certain transition , the next state in this case is assumed to be the PHI state where PHi is the state where the DFA gets stuck (like a DFA sink state).
2. The first node created by the Automata is always the root of the graph.

I. Bonus

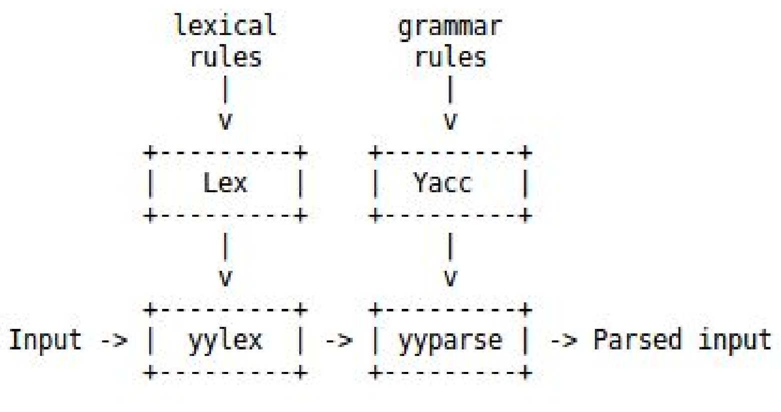
What is LEX

LEX is a software tool designed to generate scanners, also known as tokenizers, which recognize lexical patterns in text.

LEX main purpose is to perform lexical analysis, the processing of character sequences such as source code to produce tokens for use as input to other programs such as parsers.

LEX reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language.

This source code is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into tokens which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a DFA generated by Lex. The programs segments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.



```
          lexical        grammar
           rules          rules
             |              |
             v              v
        +---------+    +---------+
        |   Lex   |    |  Yacc   |
        +---------+    +---------+
             |              |
             v              v
        +---------+    +---------+
Input -> |  yylex  | -> | yyparse | -> Parsed input
        +---------+    +---------+
```

Lex with Yacc

Required Steps For Using LEX

1. Create a new .lex file for el lexical rules following the LEX file format which is

```
     {definitions}
%%
{rules}
%%
{user subroutines}
```

The created file contains our definitions and described lexical rules and their corresponding actions then in the subroutines section, there is a main in order to read a code file and send it to the generated lexical analyzer to generate tokens as follows :

```
%{
#include <stdio.h>
%}
letter [a-z|A-Z]
digit  [0-9]
digits {digit}+
num {digit}+|({digit}+\.{digit}+E{digits})|({digit}+\.{digit})
ID  {letter}({letter}|{digit})*
relop  (\=\=)|(!\=)|(>)|(>\=)|(<)|(<\=)
assign [\=]
escape_chars [; , \( \) \{ \}]
addop  [\+ | \-]
mulop  [* | /]
%%
" "    ;
if printf("%s is a reserved word\n",yytext);
else printf("%s is a reserved word\n",yytext);
while printf("%s is a reserved word\n",yytext);
```

```
{ID} printf("%s is an Identifier\n", yytext);

{num} printf("%s is a Number\n", yytext);

{assign} printf("%s is an assignment operation\n", yytext);

{relop} printf("%s is an operation\n",yytext);

{escape_chars} printf("%s is an escape character\n",yytext);

{addop} printf("%s is an addition or subtraction operation\n",yytext);

{mulop} printf("%s is a multiplication or division operation\n",yytext);

%%

main(int argc, char* argv[]) {

       FILE *fh;

       if (argc == 2 && (fh = fopen(argv[1], "r")))

       yyin = fh;

       yylex();

       return 0;

}
```

2. Generate the C file lex.yy.c for the scanner using the following command :

   “ lex lexical-rules.lex “

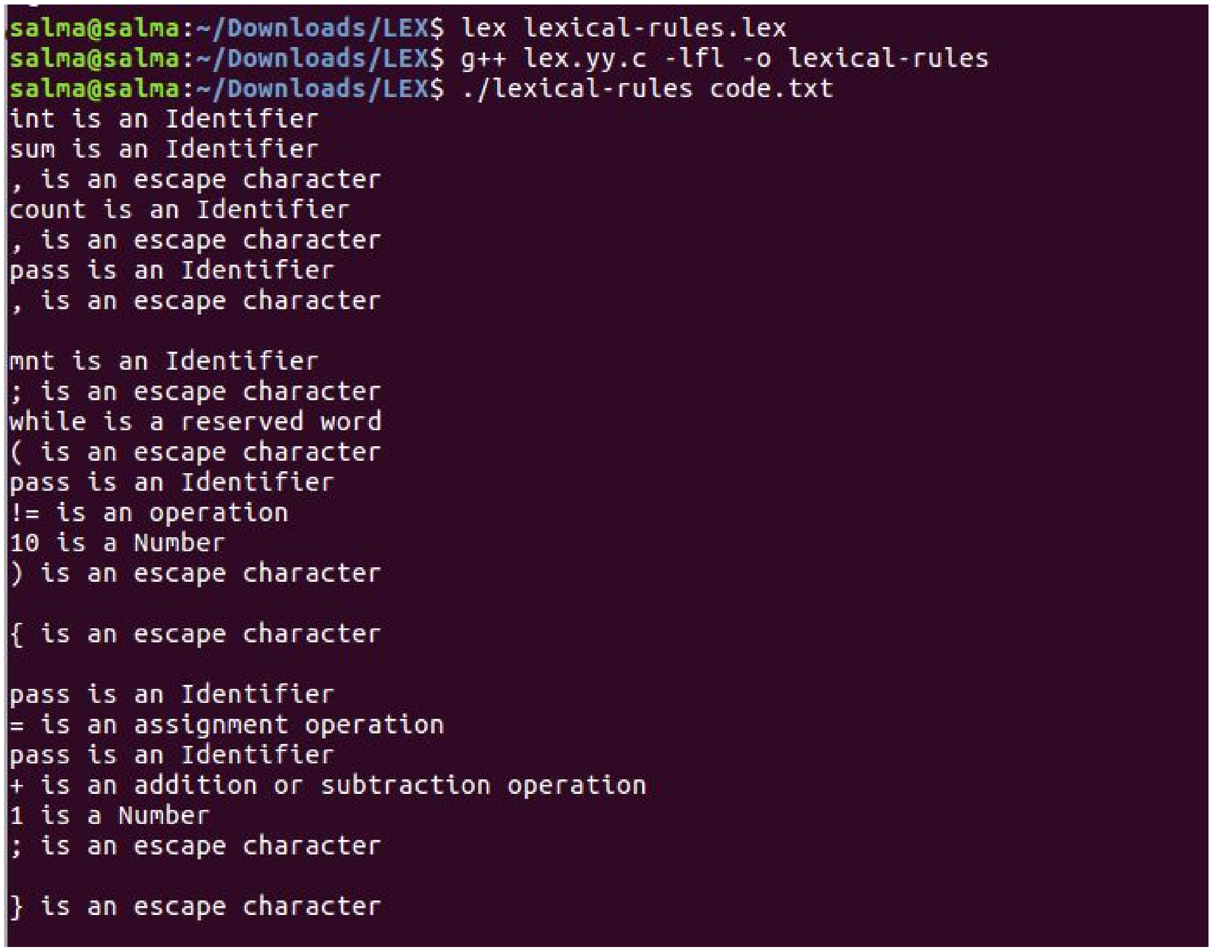
3. Generate the executable file for the the above C file using this command :

   “ g++ lex.yy.c -lfl -o lexical-rules “

4. Run the generated executable file in order to analyze the code in a given file named *“ code.txt”* using the following command :

   “./lexical-rules code.txt “

5. The output for the test program is as follows :



```
salma@salma:~/Downloads/LEX$ lex lexical-rules.lex
salma@salma:~/Downloads/LEX$ g++ lex.yy.c -lfl -o lexical-rules
salma@salma:~/Downloads/LEX$ ./lexical-rules code.txt
int is an Identifier
sum is an Identifier
, is an escape character
count is an Identifier
, is an escape character
pass is an Identifier
, is an escape character

mnt is an Identifier
; is an escape character
while is a reserved word
( is an escape character
pass is an Identifier
!= is an operation
10 is a Number
) is an escape character

{ is an escape character

pass is an Identifier
= is an assignment operation
pass is an Identifier
+ is an addition or subtraction operation
1 is a Number
; is an escape character

} is an escape character
```