Lexical analyzer phase 1

Team members:

Mahmoud Saied 60

Abd El Rahman Atef 41

Mohamed Ramdan 56

Outline:

1. Introduction.
2. Data structures.
3. Algorithms and Techniques.
4. Transition table.
5. Tokens.
6. Bonus task.

1 – Introduction:

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.

2 – Data structures:

```
class InputTokenizer
{
    public:
        InputTokenizer(void);
        virtual ~InputTokenizer();
        void parseFile(char * filePath);
        void printContents(void);

        std::map <std::string,std::string> regDefn;
        std::map <std::string,std::string> tokens;
        std::vector <std::string> kwds;
        std::vector <std::string> pncs;
        std::vector <std::string> tokens_ordered;
    protected:

    private:


        void parseLine(std::string line);
        void clrSpcs(std::string &line);

        void parseExp(std::string line, int pos);
        void parseDef(std::string line, int pos);
        void parsePunc(std::string line);
        void parseKwd(std::string line);

        std::string expand(std::string  str);
        std::string noSpaces(std::string str);
        std::string substitue(std::string str);
        std::string addSpaces(std::string str);
};
```

The InputTokenizer class contains the main functions used, maps for the regular expressions and tokens and vectors of the final sorted tokens.

```
class Node{
    /*
    the smallest unit of a graph a Node that keeps
    track of the next Nodes.
    */
public:
    void addEdge(Node * n1, char inp);
    bool isFinal = false;                              /* is this Node final ? */
    std::vector <std::pair<Node *,char> > edges;       /* connected Nodes */
};


struct graph {
    /* a graph is characterized by its starting and ending points*/
    Node * start, * end;
};
```

This class describes how the graph will be represented isFinal variable shows if the node is terminal node or not.

Edges vector contains the nodes that can be reached directly from the current node.

```
class NFABuilder
{
    public:
        NFABuilder(InputTokenizer& inTok);
        virtual ~NFABuilder();
        aGraph * buildFinal();
        void test();
        graph * pntrGraph;
        std::map<Node *,std::string> finalNodes;     /* what does every final Node define ? */
        std::map<int,std::string> finalNodesA;       /* the same as finalNodes but is accessed using Nodes numbering */
        std::map<std::string,int> priorities;        /* priority of every identifier */
        void printGraph(graph * g);
        void printAGraph(aGraph * g);
    protected:
    private:
        std::string delms = "|\\()*+";               /* language delimiters */
        InputTokenizer inTok;
        graph * kleenClosure(graph * g);
        graph * kleenPlus(graph * g);
        graph * concat(graph * g1,graph * g2);
        graph * alter(graph * g1,graph * g2);
        graph * alter2(graph * g1,graph * g2);
        graph * construct(char a);
        graph * toPostfix(std::string lhs, std::string x);
        graph * doOperation(char op, std::stack <graph *> &expr);
        graph * connect(graph* g1, graph* g2);
        graph * connect2(graph * g1);
        graph * build();
        aGraph * adapt(graph * g1);
        void buildPriority();
};
```

This class contains the main functions used for building the NFA graph, maps for final nodes and what it represents, also contains map for the priority of every token.

```
class outputParser
{
    public:
        void match(char * path);
        outputParser(aGraph * g1, std::map<int,std::string> *fs);
        virtual ~outputParser();
    protected:
    private:
    aGraph * g;
    std::map<int,std::string> *finalStates;
    int findEdge(int node,char ch);
};
```

This class contains final states of the graph.

3 – Algorithms and techniques:

In this section we will describe how the analyzer take the input file and convert it to and automata.

1-Converting the input file to regular expressions:

In this part we first take input file then convert every line of the file to regular expression then save the result to a data structure "map".

When we find a regular definition depend on another one defined before we search for this one in our data structure as we saved this definition before.
If we have for example letter : [a-z] we expand it to a|b|c…..|z.

2-Build NFA:

In this part we take every regular expression separately and convert it to its NFA graph.
Then we make only one start node for all NFAs built before so now we have one NFA for all expressions.

3-Convert NFA to DFA:

In this part we convert NFA to DFA by removing all epsilon edges.
First we begin from the start node and save all nodes reached from epsilon edges in some data structure and make all this nodes in one new node (state) and then start from every node contained in this state and repeat the first step until all nodes in the NFA graph covered.
The of this algorithm is "every group of nodes in NFA graph represent a single node in the DFA graph".

4-Minimize DFA:

In this part we minimized the DFA graph derived from the NFA.

We combine every group of states that go to the same states in the transition table in one state and then derive minimized transition table that we can build from it the minimized graph.

5-Matching:

In this part we scan the input code character by character and use munch algorithm for matching.

Munch algorithm works as follow:

Go from the current state to the next state which can be reached if we have the current character until we stuck then we take the last matched token if there is a tie we take the token with highest priority.

4 – Tokens:

Attached file describes the tokens of the sample program.

5 – Transition table:

Attached file describes the transition table.

6 – Bonus task:

Steps for using the tool:

1 – Download the lex.exe for windows.

2 – After installing the program open it.

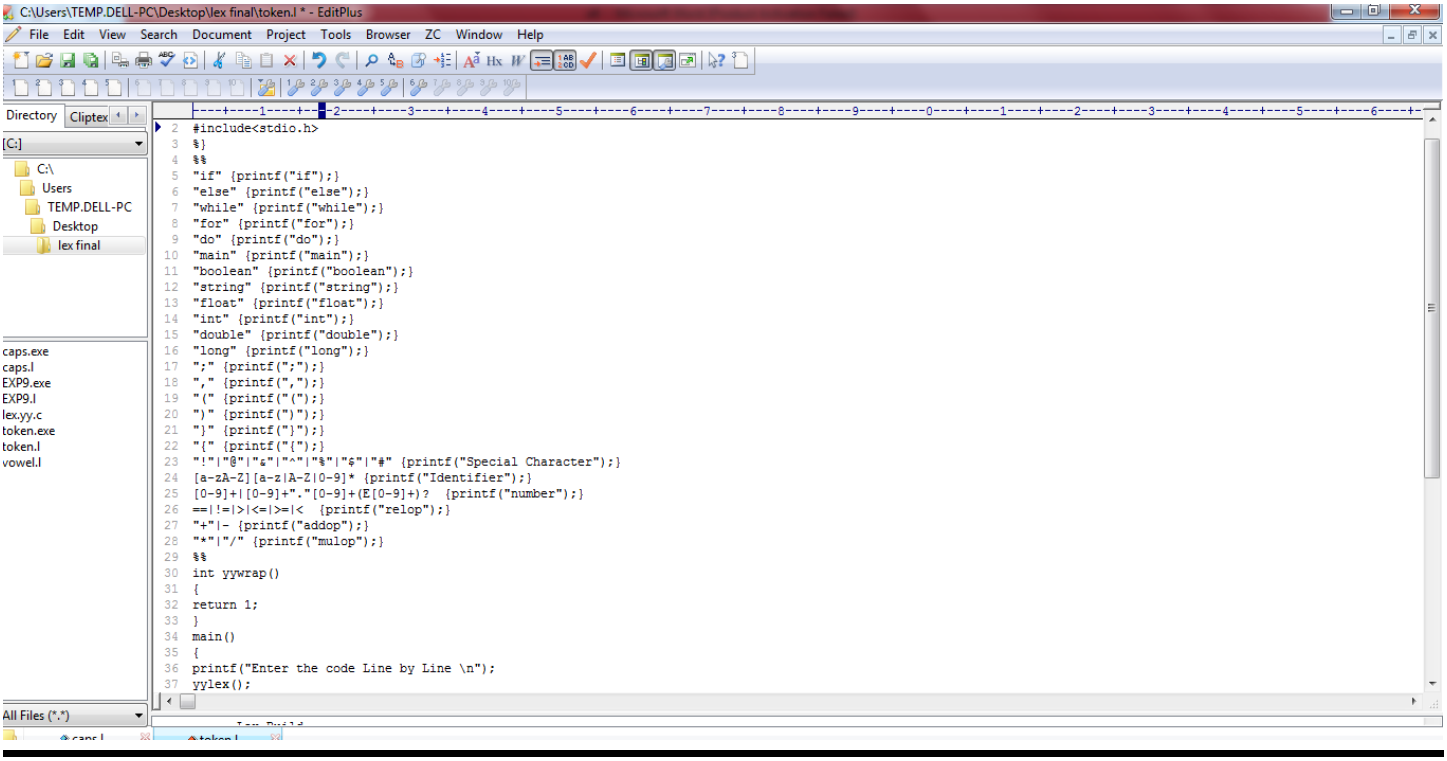
3 – Choose the source code file from (file -> open).

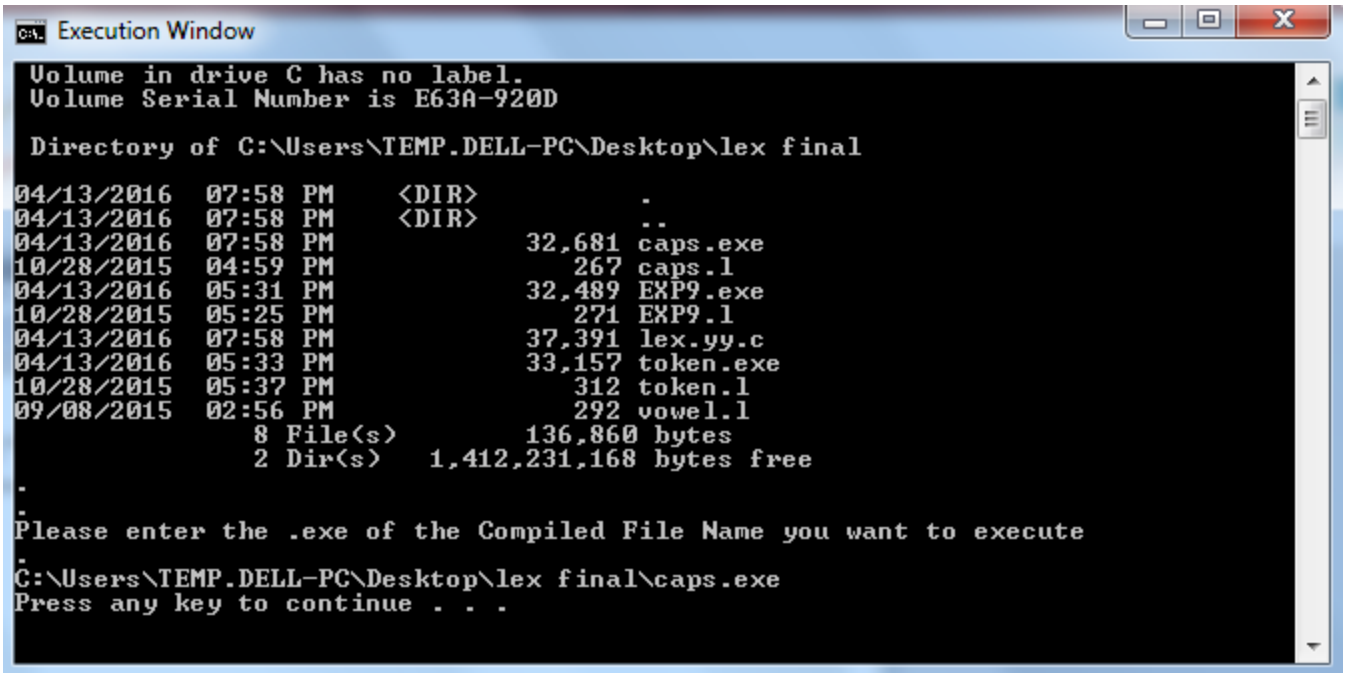
4 – From tools (Lex file compile).

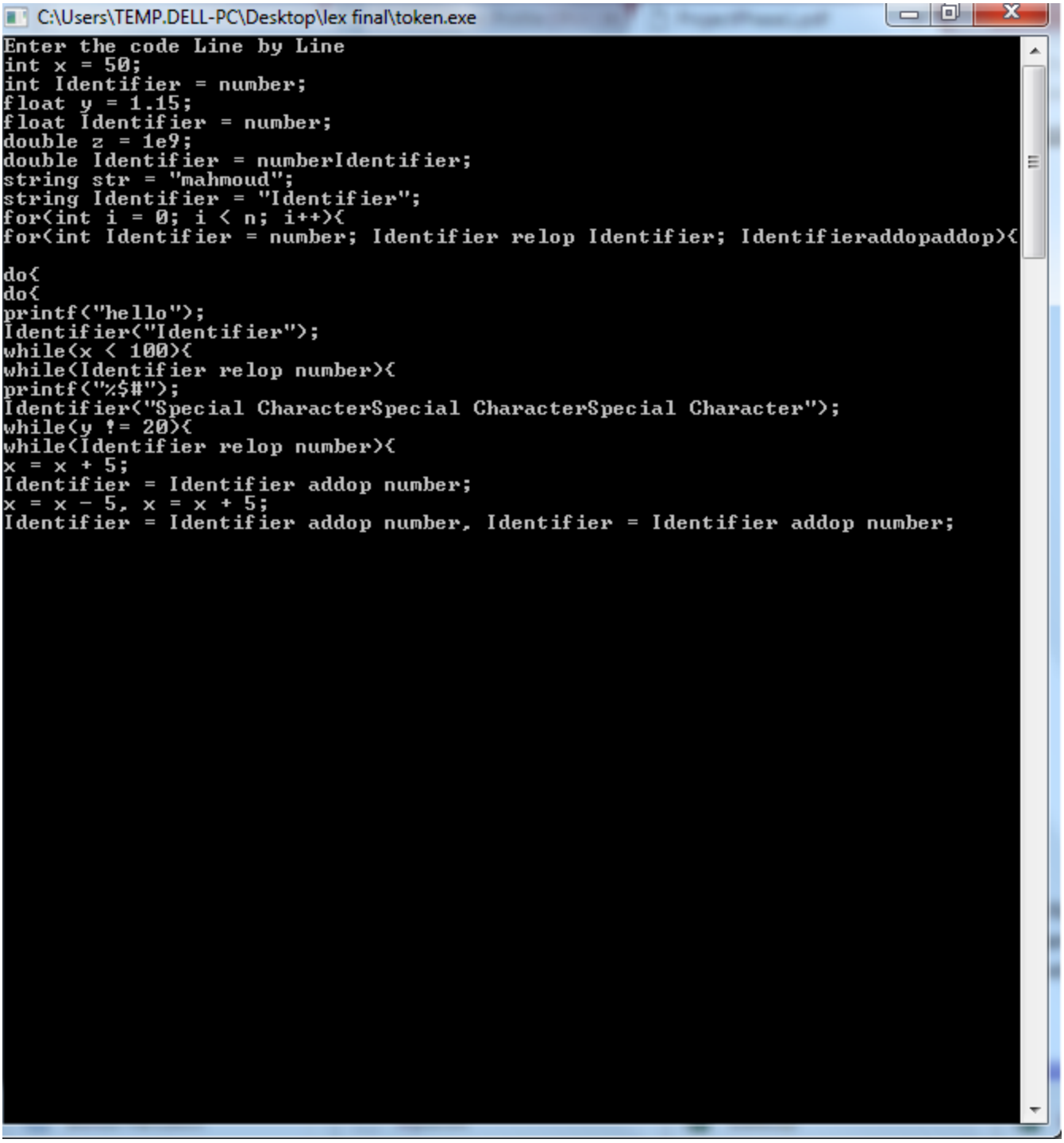
5 – From tools (Lex build).

6 – From tools (Execute exe directly).

7 – after each step from 4 to 6 the state of the operation appears below and it should always be "Normal Termination".


C:\Users\TEMP.DELL-PC\Desktop\lex final\token.l * - EditPlus
File Edit View Search Document Project Tools Browser ZC Window Help
Directory Cliptex
[C:]
C:\
Users
TEMP.DELL-PC
Desktop
lex final
caps.exe
caps.l
EXP9.exe
EXP9.l
lex.yy.c
token.exe
token.l
vowel.l
All Files (*.*)
#include<stdio.h>
%}
%%
"if" {printf("if");}
"else" {printf("else");}
"while" {printf("while");}
"for" {printf("for");}
"do" {printf("do");}
"main" {printf("main");}
"boolean" {printf("boolean");}
"string" {printf("string");}
"float" {printf("float");}
"int" {printf("int");}
"double" {printf("double");}
"long" {printf("long");}
";" {printf(";");}
"," {printf(",");}
"(" {printf("(");}
")" {printf(")");}
"}" {printf("}");}
"{" {printf("{");}
"!"|"@"|"&"|"^"|"%"|"$"|"#" {printf("Special Character");}
[a-zA-Z][a-z|A-Z|0-9]* {printf("Identifier");}
[0-9]+|[0-9]+"."[0-9]+(E[0-9]+)? {printf("number");}
==|!=|>|<=|>=|< {printf("relop");}
"+"|- {printf("addop");}
"*"|"/" {printf("mulop");}
%%
int yywrap()
{
return 1;
}
main()
{
printf("Enter the code Line by Line \n");
yylex();


Execution Window
Volume in drive C has no label.
Volume Serial Number is E63A-920D
Directory of C:\Users\TEMP.DELL-PC\Desktop\lex final
04/13/2016 07:58 PM <DIR> .
04/13/2016 07:58 PM <DIR> ..
04/13/2016 07:58 PM 32,681 caps.exe
10/28/2015 04:59 PM 267 caps.l
04/13/2016 05:31 PM 32,489 EXP9.exe
10/28/2015 05:25 PM 271 EXP9.l
04/13/2016 07:58 PM 37,391 lex.yy.c
04/13/2016 05:33 PM 33,157 token.exe
10/28/2015 05:37 PM 312 token.l
09/08/2015 02:56 PM 292 vowel.l
8 File(s) 136,860 bytes
2 Dir(s) 1,412,231,168 bytes free
.
.
Please enter the .exe of the Compiled File Name you want to execute
.
C:\Users\TEMP.DELL-PC\Desktop\lex final\caps.exe
Press any key to continue . . .


C:\Users\TEMP.DELL-PC\Desktop\lex final\token.exe
Enter the code Line by Line
int x = 50;
int Identifier = number;
float y = 1.15;
float Identifier = number;
double z = 1e9;
double Identifier = numberIdentifier;
string str = "mahmoud";
string Identifier = "Identifier";
for(int i = 0; i < n; i++){
for(int Identifier = number; Identifier relop Identifier; Identifieraddopaddop){
do{
do{
printf("hello");
Identifier("Identifier");
while(x < 100){
while(Identifier relop number){
printf("%$#");
Identifier("Special CharacterSpecial CharacterSpecial Character");
while(y != 20){
while(Identifier relop number){
x = x + 5;
Identifier = Identifier addop number;
x = x - 5, x = x + 5;
Identifier = Identifier addop number, Identifier = Identifier addop number;