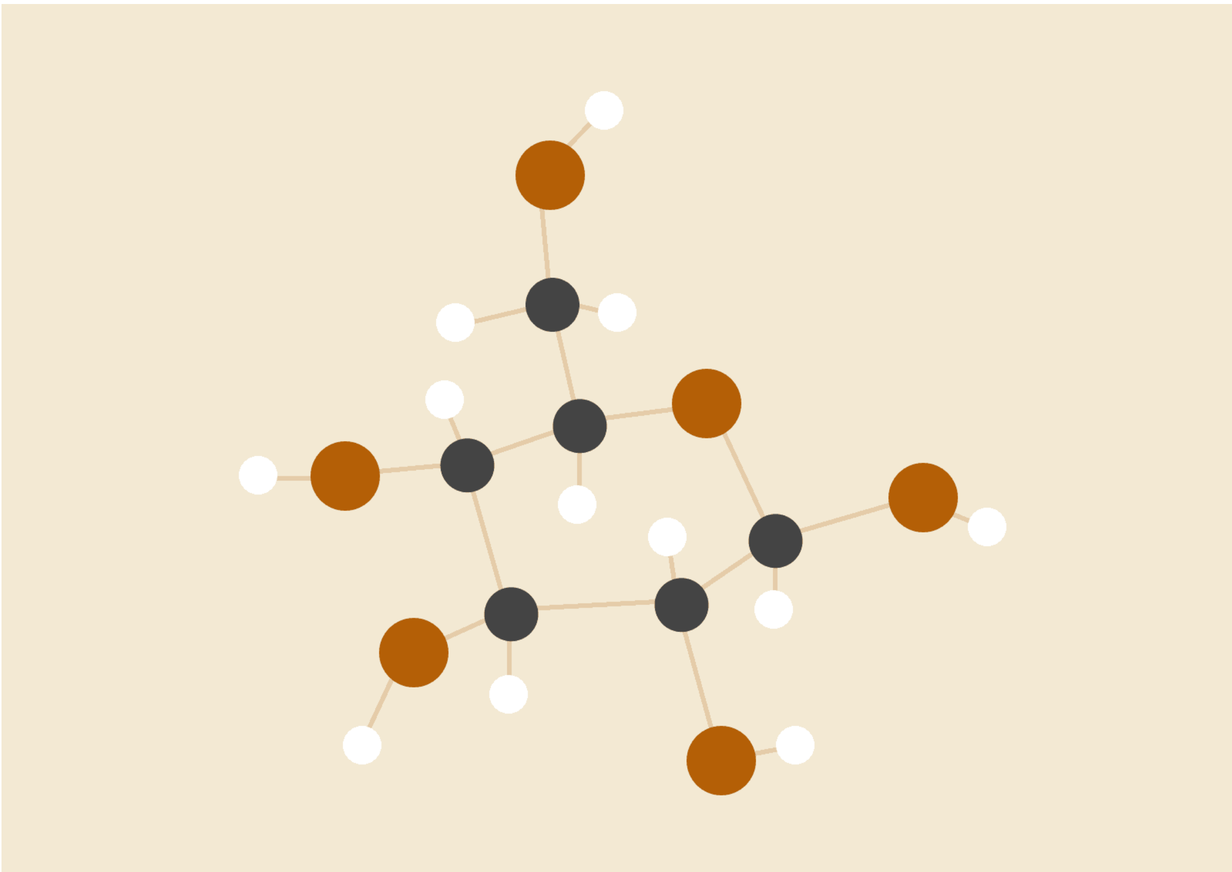
Compiler phase 1 report



Moaz Nabil - 18011824

Abdelrahman Kassem - 18010948

Abdelrahman Fawzy - 18010893

Ans Gomaa - 18010421

INTRODUCTION

What is lexical analysis? lexical analysis or tokenization is the process of converting a sequence of characters (lexemes)- such as in a computer program - into a sequence of tokens - strings with an assigned and thus identified meaning - What is the lexical analyzer generator? A lexical analyzer generator is a program designed to generate lexical analyzers, which recognize lexical patterns in text. The lexical analyzer generator is required to automatically construct a lexical analyzer from a regular expression description (rules) of a set of tokens.

Algorithms and techniques used

- Thompson's algorithm

  Given an NFA it concatenates the new to the current NFA.

```
NFABuilder &NFABuilder::Concatenate(NFA rhs) {
   nfa.end->addTransition(EPSILON, move(rhs.start));
   nfa.end = move(rhs.end);
   return *this;
}
```

  Given an NFA it builds a new NFA ORed with the current NFA.

```
NFABuilder &NFABuilder::Or(NFA rhs) {
   auto new_start{make_shared<NFA::Node>()};
   auto new_end{make_shared<NFA::Node>()};

   new_start->addTransition(EPSILON, move(nfa.start));
   new_start->addTransition(EPSILON, move(rhs.start));
   nfa.end->addTransition(EPSILON, new_end);
   rhs.end->addTransition(EPSILON, new_end);
   nfa.start = move(new_start);
   nfa.end = move(new_end);
   return *this;
}
```

This function builds a new NFA that accepts the positive or kleene closure of current NFA

```
NFABuilder &NFABuilder::PositiveClosure() {
   auto new_start{make_shared<NFA::Node>()};
   auto new_end{make_shared<NFA::Node>()};

   nfa.end->addTransition(EPSILON, nfa.start);
   new_start->addTransition(EPSILON, move(nfa.start));
   nfa.end->addTransition(EPSILON, new_end);
   nfa.start = move(new_start);
   nfa.end = move(new_end);
   return *this;
}

NFABuilder &NFABuilder::KleeneClosure() {
   this->PositiveClosure();
   nfa.start->addTransition(EPSILON, nfa.end);
   return *this;
}
```

- Subset construction

We start by finding starting states of all NFAs along with their epsilon closure then we make every possible transition and whenever we encounter a new state we add it to our queue data-structure to explore it latter.

```
DFA::DFA(const vector<RegularExpression> &regExps) {
   queue<NFA::Set> frontier;
   map<NFA::Set, int> visited;
   NFA::Set start;
   for (const auto &regEXP: regExps) {
       start.insert(regEXP.getNFA().getStart());
   }
   start = getEpsilonClosure(start);
   int stateID = 0;
   visited[start] = stateID;
   states.emplace_back(stateID++);
   frontier.push(start);
   while (!frontier.empty()) {
       auto current = move(frontier.front());
       frontier.pop();
```

```
        int index = visited.at(current);
        markAcceptingStates(states[index], current, regExps);
        for (char c = 1; c < CHAR_MAX; ++c) {
            NFA::Set next = getEpsilonClosure(Move(current, c));
            if (!visited.count(next)) {
                visited.insert({next, stateID});
                states.emplace_back(stateID++);
                frontier.push(next);
            }
            states[index].transitions[c] = visited.at(next);
        }
    }
    int emptySetIndex = visited.at(NFA::Set());
    for (auto &state: states) {
        state.transitions[0] = emptySetIndex;
    }
    this->minimizeDFA();
}
```

- Minimization

Finds the minimal set of states of the current DFA using classify method which divides the states into 2 sets of clases one set for the accepting states clases and a class for non-accepting states.

```
void DFA::minimizeDFA() {
    vector<int> statesClasses = classify();
    vector<State> newStates;
    for (int i = 0; i < states.size(); i++) {
        if (statesClasses[i] == newStates.size()) {
            newStates.emplace_back(move(states[i]));
            newStates.back().id = statesClasses[i];
            newStates.back().transitions =
renewTransitions(newStates.back().transitions, statesClasses);
        }
    }
    states = move(newStates);
}
```

- Maximum munch

Perform the maximal munch algorithm that receives a word and divides it into tokens.

```
void LexicalParser::applyMaximalMunch(const string &segment) {
   if (segment.empty()) return;

   int lastAcceptingState = -1;
   int lastAcceptingStateIndex = -1;
   int i = 0;

   const vector<DFA::State> &states = this->dfa.getStates();
   while (i < segment.length()) {
       const DFA::State *state = &states.at(0);
       for (int j = i; j < segment.length(); j++) {
           state = &states.at(state->transitions.at(segment[j]));
           if (state->isAccepting) {
               lastAcceptingStateIndex = j;
               lastAcceptingState = state->id;
           }
       }
       if (lastAcceptingStateIndex < i) {
           cerr << "Error in line " << lineNumber << " :" <<
segment.substr(i) << " Couldn't match\n";
           i++;
           continue;
       }
       this->tokenQueue.push({states.at(lastAcceptingState).regExp,
                              segment.substr(i, lastAcceptingStateIndex -
i + 1)});
       i = lastAcceptingStateIndex + 1;
   }
}
```

Datastrucre used

- Struct component consists of a type and a string representing its regular definition or expression if it has one.
- Vector of strings to store regular expression literals keywords and punctuation.
- Vector of strings to store regular Expressions literals
- Vector of strings to store keywords names;
- Vector of strings to store punctuation names;
- vector<pair<string, vector<component>>> to store regular Definition Components;
- unordered_set<string> to store regular Definition names;
- Regular expression class has name, priority, and NFA.
- NFA class consists of start and end nodes.
- NFA::Node class has an id and transitions which are an unordered_map<char, vector<*Node>>
- Struct state has an id, boolean isAccepting, string regExp, vector<int> transitions
- DFA class has multiple states
- struct Token string regExp,string matchString

Minimal DFA Transition Table

Accepting states table

Test Sample

Input program

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

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: \* | /
```

Output result

```
int ==> int
sum ==> id
, ==> ,
count ==> id
, ==> ,
pass ==> id
```

```
, ==> ,
mnt ==> id
; ==> ;
while ==> while
( ==> (
pass ==> id
!= ==> relop
10 ==> num
) ==> )
{ ==> {
pass ==> id
= ==> assign
pass ==> id
+ ==> addop
1 ==> num
; ==> ;
} ==> }
```

Assumptions:

Expressions befroe and after the character (-) which indicate a 'to' operation for example (A-Z) means characters from A to Z, have to be each a single character, for example AZ-A isn't valid. Only ASCI characters ranging from ! (ASCI 33) to ~ (ASCI 126) are printed in the transition table.

Bonus Part:

The following steps are made on an ubuntu system.

Steps:

- Install flex from the terminal using the command:

  $ Sudo apt-get install flex

- 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 curly brackets%{%}

B. Rules Section:It is the part where we define the regular expressions, It follows this pattern: %% Pattern {Action} %%

C. Code Section:it contains the c user code and functions.

The input file passes through the lex compiler and produces 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 function named yylex() to run the rules section. Use this command to produce the (.c) file:

$ lex filename.l

● Compile the (.c) file named lex.yy.c to produce the executable file named ./a.out, use this command:

$ gcc lex.yy.c10

● Run the Executable File. Use this command:

$./a.out

```
%{
%}
DIGIT    [0-9]
ID       [a-zA-Z][a-zA-Z0-9]*
%%
[;|,|(|)|{|}] {printf("%s punctiuation\n", yytext);}
int|if|else|while|boolean|float {printf("%s Keyword\n", yytext);}
{DIGIT}    {printf( "%s digit\n", yytext);}
{DIGIT}+    {printf( "%s number\n", yytext);}
{DIGIT}+"."{DIGIT}*        {printf( "%s float number\n", yytext);}
{ID}        {printf( "%s An identifier\n", yytext );}
"<="|"<"|">="|">"|"!="|"=="   {printf( "%s A relop\n", yytext );}
"="   {printf( "%s Assign\n", yytext );}
"+"|"-"   {printf( "%s addop\n", yytext );}
"*"|"/"   {printf( "%s mulop\n", yytext );}
"{"[^{}\n]*"}"     /* eat up one-line comments */
[ \t\n]+          /* eat up whitespace */
.            {printf( "%s Unrecognized character\n", yytext );}
%%
int yywrap(){}
int main(){
yylex();
return 0;
}
```

Output:

```
Terminal:  Local
program.txt
int Keyword
sum An identifier
, punctuation
count An identifier
, punctuation
pass An identifier
, punctuation
mnt An identifier
; punctuation
while Keyword
( punctuation
pass An identifier
!= A relop
10 number
) punctuation
{ punctuation
pass An identifier
= Assign
pass An identifier
+ addop
1 digit
; punctuation
} punctuation
moaz@moaz:~/CSED-7/compilers/project/Compiler$
```