PROGRAMMING LANGUAGE TRANSLATION

PHASE 1

LEXICAL ANALYZER

SANDRA SHERIF WASFI 3940

RIMON ADEL KARAM 4441

YOUSSEF MOHAMED 3812

ADEL ATEF MELEKA 3960

PROBLEM DESCRIPTION

The task in this phase of the assignment 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.

LEXICAL ANALYSER PROCEDURE

In order to construct a Lexical Analyzer Generator, the following steps must be achieved:

1. Read Regex file.
2. Convert Regex file into NFA.
3. Convert NFA into DFA.
4. Generate minimized DFA.
5. Extract Token class given an example code.

A detailed description for each step is explained below …

1. READ REGEX FILE

- For reading the file w created REGEX that accept the REGEX from the file and separate the terminal labels from the non-terminal labels
- We used hash map where the key was the pattern name and the values were an array of strings where each string was ordered with the next string.
- Main Data Structure Used : Hashmap.

2. CONVERT REGEX FILE TO NFA

- We constructed a nondeterministic finite automata (NFA) for the given regular expressions, combine these NFAs together with a new starting state as NFA starting Graph Node.

- Details are as follows:

Node Class:

- This class contain hashmap where the key is the input that is needed to go to certain node and the value as you can guess is that certain node

- Each node contain boolean named isGoal to determine weather or not the this node is goal or not.

SubGraph Class:

- This class have many static functions that return subgraph. These functions create the:

1) (or) between many SubGraph
2) (con-cat) between 2 SubGraphs or 1 SubGraph and Node.
3) Kleene closure for SubGraph
4) positive closure for SubGraph

NFA Class:

- This class receives the regex from the regex reader and then it process on it and generate the NFA starting from the NFA of the keywords then the NFA of non-terminal labels.
- we used stacks and hashmaps to generate the non-terminal Labels NFA
- Then we or all this subgraph together in main graph

Simulating a DFA

```
s := s0;
c := nextchar;
while c ≠ eof do
        s := move(s, c);
        c := nextchar
end;
if s is in F then
        return "yes"
else return "no";
```

3. CONVERT NFA TO DFA

- Given an input NFA Node & set of all possible inputs, it is required to convert it to DFA.
- In order to achieve this functionality, we have defined a base class **DFAstate** holding the base element of any state in our DFA graph.

- Conversion is done by the following algorithm:

Subset Construction (III)

initially, ϵ-*closure*(s_0) is the only state in *Dstates* and it is unmarked;
while there is an unmarked state T in *Dstates* **do begin**
    mark T;
    for each input symbol a **do begin**
        U := ϵ-*closure*(*move*(T, a));
        if U is not in *Dstates* **then**
            add U as an unmarked state to *Dstates*;
        Dtran[T, a] := U
    end
end

(ε-closure computation)

```
push all states in T onto stack;
initialize ϵ-closure(T) to T;
while stack is not empty do begin
        pop t, the top element, off of stack;
        for each state u with an edge from t to u labeled ϵ do
            if u is not in ϵ-closure(T) do begin
                add u to ϵ-closure(T);
                push u onto stack
            end
end
```

- Main Data Structures Used:
  Stack – ArrayList – Hashmap (Transition Table)
- The final output of the conversion is a **Transition Table** representing DFA state of a DFA graph.

4. GENERATE MINIMIZED DFA

- Given a DFA graph, we minimize it and emit the transition table for the reduced DFA together with a lexical analyzer program that simulates the resulting DFA machine.
- Minimize the number of states of a DFA by finding all groups of states that can be distinguished by some input string.

- Each group of states that cannot be distinguished is then merged into a single state.
- This is done by the following algorithm:

Algorithm: Minimizing the number of states of a DFA

- Input. A DFA M with set of states S, set of inputs $\sum$, transitions defined for all states and inputs, start state s_0, and a set of accepting states F.
- Output. A DFA M' accepting the same language as M and having as few states as possible.
- Method.

1. Construct an initial partition $\prod$ of the set of states with two groups: the accepting states F and non-accepting states $S - F$.
2. Partition $\prod$ to $\prod_{new}$.
3. If $\prod_{new} = \prod$, let $\prod_{final} = \prod$ and go to step (4). Otherwise, repeat step (2) with $\prod := \prod_{new}$.
4. Choose one state in each group of the partition $\prod_{final}$ as the *representative* for that group.
5. Remove dead states.

5. EXTRACT TOKENS FROM A SAMPLE PROGRAM

- 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.
- 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.

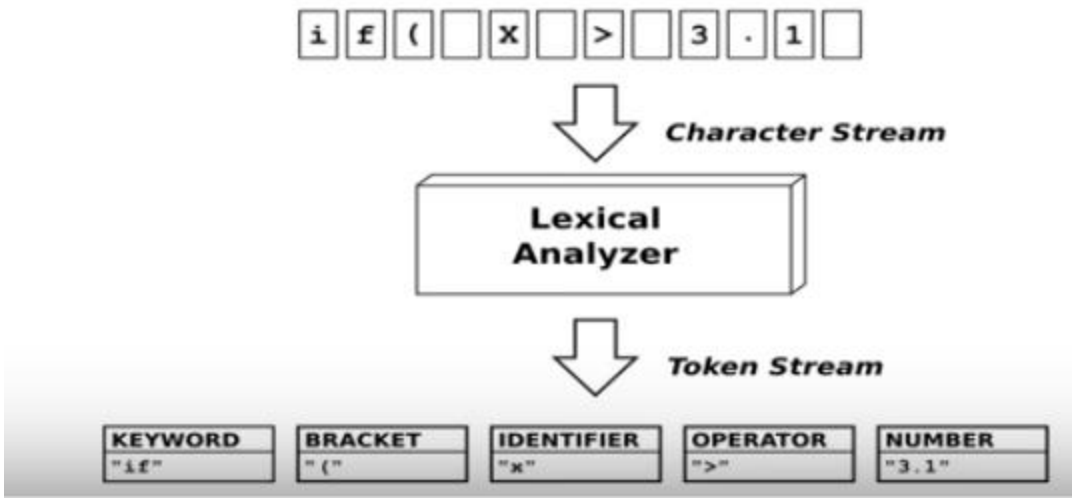
LEXICAL AUTOMATIC GENERATOR 'LEX'

WHAT IS LEX TOOL

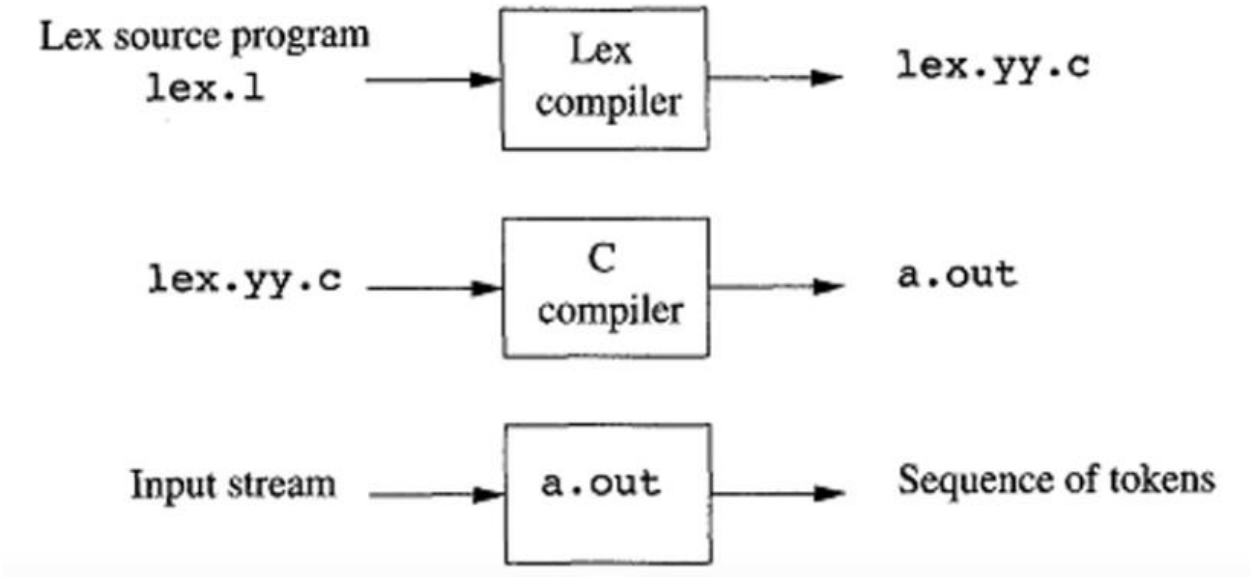
- Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem-oriented specification for character string matching, and produces a program in a general-purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed.
- The user supplies the additional code beyond expression matching needed to complete his tasks, possibly including code written by other generators. The program that recognizes the expressions is generated in the general-purpose programming language employed for the user's program fragments. Thus, a high-level expression language is provided to write the string expressions to be matched while the user's freedom to write actions is unimpaired. This

avoids forcing the user who wishes to use a string manipulation language for input analysis to write processing programs in the same and often inappropriate string handling language.

- Lex is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called ``host languages." Just as general-purpose languages can produce code to run on different computer hardware, Lex can write code in different host languages. The host language is used for the output code generated by Lex and for the program fragments added by the user. Compatible run-time libraries for the different host languages are also provided. This makes Lex adaptable to different environments and different users. Each application may be directed to the combination of hardware and host language appropriate to the task, the user's background, and the properties of local implementations. At present, the only supported host language is C, although Fortran. Lex itself exists on UNIX, GCOS, and OS/370; but the code generated by Lex may be taken anywhere the appropriate compilers exist.
- It’s a tool which generate lexical analyzer.
- Lexical analyzer is the first phase of compiler which take input as source code and generate output as tokens.



- The input notation for the LEX tool is referred as the LEX language and the itself is the LEX compiler.
- The LEX compiler transforms the input patterns into a transition diagram and generates code in a file called lex.yy.c .
- Here is a simple diagram representing phases done the LEX tool…



WORKING

- An input file, called **lex.l** , is written in the LEX language and describes the lexical analyzer to be generated .
- The LEX compiler transforms lex.l to a C program, in a file that is always name **lex.yy.c** .
- The later file is compiled by the C compiler into file called **a.out** ,as always.
- The C-compiler output is a working lexical analyser that can take a stream of input characters and produce a stream of tokens.

FORMAT

A LEX program has the following form…

```
{declarations}
%%
{translation rules}
%%
{auxiliary functions}
```

- The declarations section includes declarations of variables, i.e. #include<…>.
- The translations rules have the form: **Pattern {Action}.**
- The third section holds whatever auxiliary functions are used in the actions. Alternatively, these functions can be compiled separately and loaded with the lexical analyzer.

LEX PATTERNS

Here is attached part of supported LEX patterns…

.	any character except new line
\n	new line
*	zero or more copies of preceeding expression
+	one or more copies of preceeding expression
?	zero or one copies of preceeding expression
$	end of file
a|b	a or b

SAMPLE PROGRAM

```
declaration
letter [A-Z a-z]
digit [0-9]
id {letter}({letter|digit })*
number  {digit}({digit}|{.digit})
%% (translation )
if                  {return (IF);}
else                {return(ELSE);}
{id}                {yylval = (int) installID(); return(ID);}
{number}      {yylval = (int) installID(); return(ID);}
%% (auxiliary function)
int installID()
Int installNum()
```

THANK YOU