Programming Languages & Compilers

Phase1: Lexical Analyzer Generator

Mostafa Nabil Mohamed (54)
Tarek Mohamed Kamal (23)
Mohamed Mostafa Alnashar (46)
Merna Mohamed Mostafa (56)

Lexical Analyzer Generator

Objective

This phase of the assignment aims to practice techniques for building automatic lexical analyzer generator tools.

Problem Statement

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 Page ¼ to be called to print an error message and to continue looking for tokens.

The lexical analyzer generator is required to be tested using the given lexical rules of tokens of a small subset of Java. Use the given simple program to test the generated Lexical analyzer.

Lexical Analyzer Generator

Keep in mind that the generated lexical analyzer will integrate with a generated parser which you should implement in phase 2 of the assignment such that the lexical analyzer is to be called by the parser to find the next token.

Lexical Rules

- The tokens of the given language are: identifiers, numbers, keywords, operators and punctuation symbols.
- The token id matches a letter followed by zero or more letters or digits.
- The token num matches an unsigned integer or a floating- point number. The number
- consists of one or more decimal digits, an optional decimal point followed by one or
- more digits and an optional exponent consisting of an E followed by one or more digits.
- keywords are reserved. The given keywords are: int, float, boolean, if, else, while.
- Operators are: +, -, *, /, =, <=, <, >, >=, !=, ==
- Punctuation symbols are parentheses, curly brackets, commas and semicolons.
- Blanks between tokens are optional.

Lexical Rules Input File Format

- Regular definitions are lines in the form LHS = RHS
- Regular expressions are lines in the form LHS: RHS
- Keywords are enclosed by { } in separate lines.
- Punctuations are enclosed by [ ] in separate lines
- \L represents the Lambda symbol.
- The following symbols are used in regular definitions and regular expressions with the meaning discussed in class: - | + * ( )
- Any reserved symbol needed to be used within the language, is preceded by an
- escape backslash character.

Lexical Analyzer Generator

Used Data Structure

Graph: It's graph contains the nodes and edges of the automaton. Each graph contains some attributes like:

- Node start ➜ it holds the first node of the graph.
- Accept_state ➜ it holds the last state of the graph that accepts definition or expression in that automaton the graph represents.

Node: It represents one of the states that an automaton consists of. Each node contains some attributes like:

- Int id ➜ it holds the number of this node with respect to all automaton's nodes.
- String accepted_input ➜ it holds the pattern that was accepted if it was an accept node and if not then the value would be non to represent that it's not accept node.
- Int priority ➜ it holds the priority of the accepted node depending on the order of the lines entered in the input file.
- Vector<Edges> entering, leaving ➜ representing the edges between nodes.

Edge: It represents one of the edges that an automaton has between states. Each edge contains some attributes like:

- Node src ➜ it holds the source node.
- Node dest ➜ it holds the destination node.
- RegularDef weight ➜ it holds the weight of each edge.

RegularDef: It's mainly a graph. it holds the weight of each edge. It can be a one character weight or a whole graph representing a regular definition. Each regularDef contains some attributes like:

- Graph regular_def ➜ graph that holds the definition sequence.

Lexical Analyzer Generator

NFA: It represents the Non Deterministic Automaton that will represent the rules of the language. Mainly it's a graph. Also, It's a singleton as there's only one NFA that represents all the rules and many classes will need to use it. Each NFA contains some attributes like:

- Graph automaton ➔ it holds the Non Deterministic Automaton graph.

DFA: It represents the Deterministic Finite Automaton that will represent the rules of the language after converting the NFA to DFA. Mainly it's a table that represents the transitions from one set of nodes to another set under some input. Also, It's a singleton as there's only one DFA that represents all the rules and many classes will need to use it. Each DFA contains some attributes like:

- vector<pair<Node*, map<RegularDef*,Node*>>> transition_table ➔ it holds the transition table of the NFA after converting it. Each pair in the vector represents a row in the table. Each row has Node as the state and map that holds the inputs to that state as RegularDef and a Node that represents the state to reach under that input.
- vector<set<Node*>> stateMappingTable ➔ it's a vector to keep track of the states we have while building the final transition table.

This was the main classes structure we have in this project that will be used as data structures in the whole project along with the others like: vector, map, set, and array.

Lexical Analyzer Generator

Algorithms And Techniques

Overview of the program

The lexical analyzer generator starts by reading the input file which represents the rules of the language and after that it starts to parse it line by line and represent it as tokens after that we send the tokens of each line to be represented as an Non Deterministic Automaton by building each token as nodes in a graph then we merge the automatons of all lines together by ORing them and send the NFA to be converted into DFA and then minimize it. Finally we read the program that is needed to be tested character by character and test it through our minimized DFA and check if it works matches any of the patterns we created from the rules file and print each token's type after that.

Tokenizing the rules

After reading the input file each line is tokenized into a vector<string>.
We use **ReadRules** class to read the file line by line and then send it to **ExpNfa** class to function **construct_automata** which send it at the beginning to **Tokenizing** class to split the line after each space and check if it contains special cases that need further splits like having operation like * + - | = at the end or in between of a word and that is achieved by parsing each word character by character to check if there's any operation so we will split the word before it. Finally we return a vector of tokens back to the constuct_automata to continue building it.

Creating NFA

We used Thompson's construction algorithm to create the NFA by representing each char as automata then merging the automatons together as the pattern state.

After the line is tokenized we have a vector of tokens that we start working recursively by splitting each in its constituent sub expressions.

Lexical Analyzer Generator

First, we check the type of the line if it's a definition, expression, keyword, or punctuation. then depending on the type we build the automata as follows:

> Definition: we split the definition into the simple forms as if it's A-Z we create automata for each character and then merge them together to create the automata. After that we add it to a map<string, Definition*> to keep track of these definitions and to use them again if needed.
>
> Expression: we create the automata of that expression using the definitions created or creating new ones and then store it in a vector of graphs to be ORed later to create the last NFA.
>
> Keyword/Punctuation: create automata by oring the characters together.

We create the graphs of these automatons by splitting the tokens over * , + , | , . and each character is created as a single automata. If it's 'a' then we have 0---a--->0 and then merge graphs together depending on the operation using the rules of thompson.

Finally after creating all the NFAs of all the lines we build the final NFA by adding another node at the beginning and edges to each NFA's starting node using LAMBDA.

Deriving DFA

After creating the NFA we send it to the **DFA** class to start building the transition table of the DFA after converting the NFA. We do that by using the subset construction algorithm.

For each state in the DFA transition table not visited yet, we get its destination set for each transition, get their epsilon closure and convert it to a new state in the DFA transition table. When no state for a certain transition is available, we direct the transition to a dummy state representing no accepted state.

Lexical Analyzer Generator

Minimizing DFA

For each regular language, there also exists a minimal automaton that accepts it, that is, a DFA with a minimum number of states and this DFA is unique (except that states can be given different names). The minimal DFA ensures minimal computational cost for tasks such as pattern matching. There are two classes of states that can be removed or merged from the original DFA without affecting the language it accepts to minimize it.

- Unreachable states are the states that are not reachable from the initial state of the DFA, for any input string.
- Non Distinguishable states are those that cannot be distinguished from one another for any input string.

First,The main partition ,which has all the states in the beginning, is divided into accepted and non-accepted partitions .Then,take each partition of them and check each two states with the same input definition to divide that partition into other partitions in condition that having the output states in the same partition.

Then check every time the previous number of partitions equals the next number of partitions if yes so we finish the minimization and begin to build the Minimized transition state table with the new partitions .

With conditions that:

- The starting group is the partition that has the starting state in it.
- The accepted input for each group is the accepted input for the first state in that partition.

Lexical Analyzer Generator

Parse Program

We take the program and split into tokens after each space then parse it character by character and send it to the minimized DFA to check if it matches any pattern by following the states under the inputs given until it reaches accept state if it matches. If it does we will print the matched pattern. If not we will print not matched.

Sample Runs

1. Test 1

   Rules:

```
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: \* | /
```

   Program:

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

Lexical Analyzer Generator

Minimization Table:

```
------This is Minimized DFA -----
Def  ! , . / 0 1 2 3 4 5 6 7 8 9 ; < = > A B C D E F G H I J K L M N O P Q R S T U V W X Y Z \( \) \* \+ \- a b c d e eps f g h i j k l m n o p q r s t u v w x y z {  Accepted
40:  6 7 6 6 12 9 6 6 20 6 6 6 6 6 6 6 6 6 6 6 6 11 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 4 4 4 4 4 4 4 4 4 4 42 1 41 2 2 5 0 26 27 29 30 28 28 3 3 <none>
41:  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 31 42 42 42 42 42 42 42 42 42 42 42 42 42 <none
>
42:  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <none
>
0:  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <,>
1:  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 31 42 42 42 42 42 42 42 42 42 42 42 42 42 <assig
n>
2:  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 31 42 42 42 42 42 42 42 42 42 42 42 42 42 <relop
>
3:  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <mulop
>
```

```
4:  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 4 4 4 4 4 4 4 4 4 4 43 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <num>
5:  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <;>
6:  6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <id>
26:  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <\(>
27:  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <\)>
28:  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <addo
p>
7:  6 6 6 6 6 6 6 6 6 6 6 6 6 6 8 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <id>
12:  6 6 6 6 6 6 6 6 6 6 6 16 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <id>
9:  6 6 6 6 6 6 6 6 6 6 6 13 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <id>
20:  6 6 6 6 6 32 6 6 6 6 6 6 6 21 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <id>
11:  6 6 6 6 6 6 6 14 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <id>
29:  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <{>
30:  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <}>
31:  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <relo
p>
```

```
43:  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 33 33 33 33 33 33 33 33 33 33 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <none
>
8:  6 6 6 6 6 6 6 6 6 6 6 6 6 6 10 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <id>
16:  6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 23 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <id>
13:  6 6 6 6 6 6 6 6 6 6 6 6 6 6 17 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <id>
32:  6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <if>
21:  6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 34 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <id>
14:  6 6 6 6 6 6 6 6 18 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <id>
33:  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 44 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 33 33 33 33 33 33 33 33 33 33 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <num>
10:  6 6 6 6 6 6 6 6 6 6 6 15 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <id>
23:  6 6 6 6 35 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <id>
17:  22 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <id>
34:  6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <int>
18:  6 6 6 6 6 6 6 6 6 6 6 24 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <id>
44:  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 36 36 36 36 36 36 36 36 36 36 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <none
>
```

```
15:  6 6 6 6 19 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <id>
35:  6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <else>
22:  6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 37 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <id>
24:  6 6 6 6 38 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <id>
36:  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 36 36 36 36 36 36 36 36 36 36 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <num>
19:  25 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <id>
37:  6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <float>
38:  6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <while>
25:  6 6 6 6 6 6 6 6 6 6 6 6 6 39 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <id>
39:  6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 <boolean>
```

Lexical Analyzer Generator

Output:

```
int
id
,
id
,
id
,
id
;
while
(
id
relop
num
)
{
id
assign
id
addop
num
;
}
```

2. Test 2

Rules:

```
letter = a-z | A-Z
digit = 0 - 9
id: letter (letter|digit)*
digits: digit+
```

Program:

```
mostafa tarek 1245 nashar 1212 merna ada21 as2d1 2assad 5421
```

Lexical Analyzer Generator

Minimized Table:

:

------This is Minimized DFA -----

Def	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	a	b	c	d	e	eps	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	Accepted
0:	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	0	0	0	0	0	0	0	0	0	0	<digits>
2:	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	<none>
1:	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	<id>
3:	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	<none>

Output:

```
id
id
digits
id
digits
id
id
id
digits
id
digits
```