Assignment #1

Lexical Analyzer Generator

Names:

- Ahmed Alaa Shalaby (9)
- Ahmed Hesham Fathy (14)
- Mohamed Ahmed Sayed Ahmed (53)
- Mohamed Osama Abd-Elmged (56)
- Mohamed Osama Hassan Azmy (55)

Agenda

Problem Statement

Your 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. 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.

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.

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.

Assumptions:

Beside rules of input file that explained in the problem statement:

1- "to" operator (-)→ appears only in the regular definitions and not in regular expressions.
   a. Ex→ **letter= (a-z)** is allowed as it is defn, but **letter: (a-z)** not allowed as it appeared in expression.

2- Unlimited spaces can be used within regular definitions and regular expressions only.
   a. Ex→ **(a | z)** is allowed.

3- Keywords, punctuations, and symbols separated only by one or more spaces.
   a. Ex →**{if else}** is allowed, **{ if   else }** is also allowed. The same for [etc]

4- The user can represent concatenation operation using two ways
   a- By making one or more spaces between two operands
   Ex → {a   b|c} here a is concatenated with b and the result has OR operation with c
   b- By making one operand and then concatenate it with expression with (..)
   Ex → { a(b c) } here b is concatenated with c , and the result is concatenated with a

5- If the regular expression contains a word (EX : letter) , if this word was defined before as regular definition (EX: letter= a-z) then we substitute this regular definition in the regular expression that has the occurrence of the word .

6- In the last case , if we haven't a regular definition , then the word must be separated by spaces (character by character) (EX: "letter" should be → "l e t t e r" so each character should be treated as an input) NOTE that this case only happens if there was no regular definition for this word.

7- If there is a keyword , Its characters should be defined in an regular definition or the program will treat it as an error (EX : {int} , letter = a-c …. The program here will see that this word is error and will not use it as a keyword}

8- To write regular definition → pattern is : name then space then '=' then space then the definition {EX: letter : a-z}
9- To write regular expression → pattern is : name then space then ':' then space then the expression {EX: id : letter(letter | id)* }
10- Spaces can't be considered as key-words , as it always will be ignored in the file.

Parser:

Data structures used:

1- Vectors <string> → to separate regular definitions, regular expressions, punctuations, and key words.
2- Vector <token object>→ know as symbol table to hold tokens
3- two dimensional vector
4- Stack <char>→used in postfix generation
5- Vector < Vector < table object > > NFA → which has a row for every state , and for each row , we have a vector of table objects to say that in this state we can go to another state using this input.
6- User defined objects:
    a. Expression object→has:
        i. type (operand ,operator).
        ii. Value (expression itself).
        iii. Int start, end states.
    b. Token object→has:
        i. Token name.
        ii. Token value.
    c. Table object→has:
        i. Int next sate.
        ii. Input to that state.

General Algorithm:

First, reading one line at a time from file then:

1- Scanning each line to insert it in its specified vector by looking at special characters (: = { [)
    a. If the line is for keywords or punctuations :
        i. Scanning line and extract all keywords and punctuations to be inserted as tokens in symbol table
    b. If the line is for expressions or definitions:
        i. scan that line to remove unwanted spaces
2- after reading file is finished and all vectors are initialized:
    a. for each definition→if '-' found then expand it to hold string of OR's
    b. For each expression→replace all definition names with their generated expanded form.
3- For each regular expression→convert it to postfix form using stack implementation with the aid of operator precedence.
4- For each postfix form →split it to operators and operands and insert them into vector
5- So each regular expression in postfix form stored in vector.
6- For all regular expression→put their vectors in one vector.

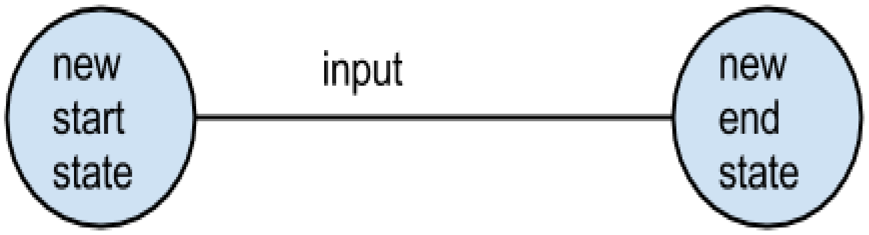
Construction of NFA from regular expressions :

In this phase the program tends to construct the NFA table from regular expressions , that will be used after that to construct the DFA table that will be minimized and used in the operation of accepting states.

In last phase we had a vector which contains many regular expressions one by one in each row in the postfix way (ex : for expression (a |b) the row of the vector should contain (a,b,|).

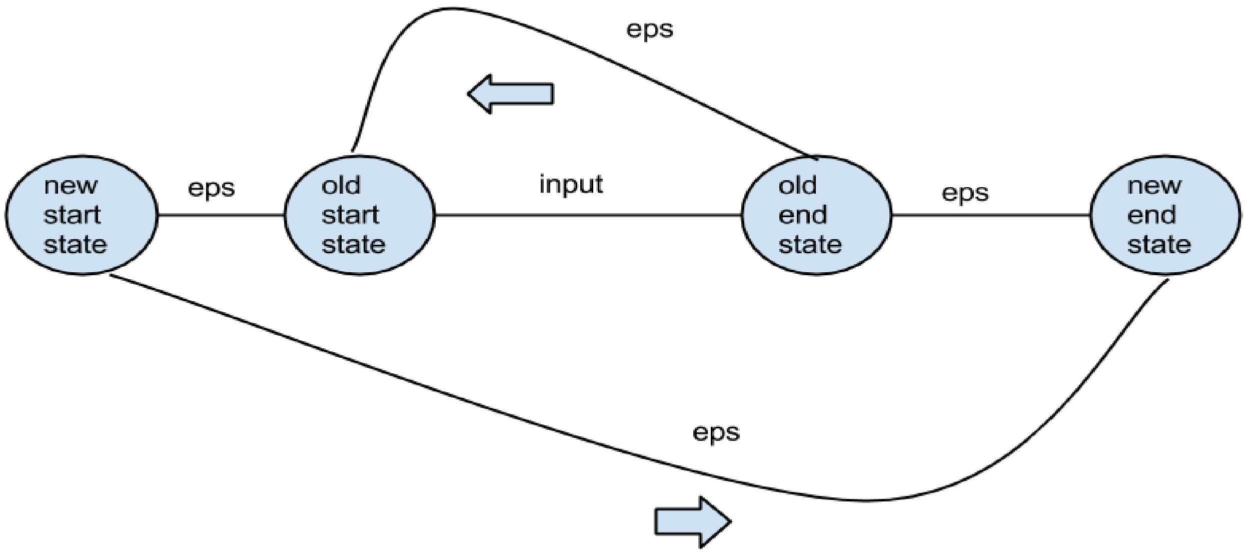
Now , we have a variable called (current state) which is incremented by one after every creation of state . We loop through the vector of postfix expression which has elements of type (Expression_obj) that has attributes for : type (operand or operator) , value (input string) , start state(initialized by one), and end state (initialized by one). And we have two cases while looping :

a- The expression is operand , so we call method (initiateExpression ()) which makes an expression for this operand consist of two states and an edge with this input as a label. We then replace the old expression in this vector with the new expression with new state and old states.

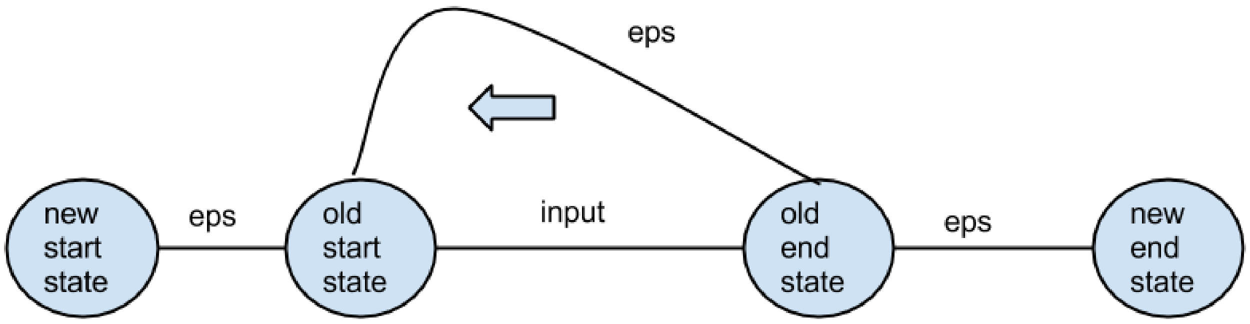


b- The expression is operator , then we have many cases :

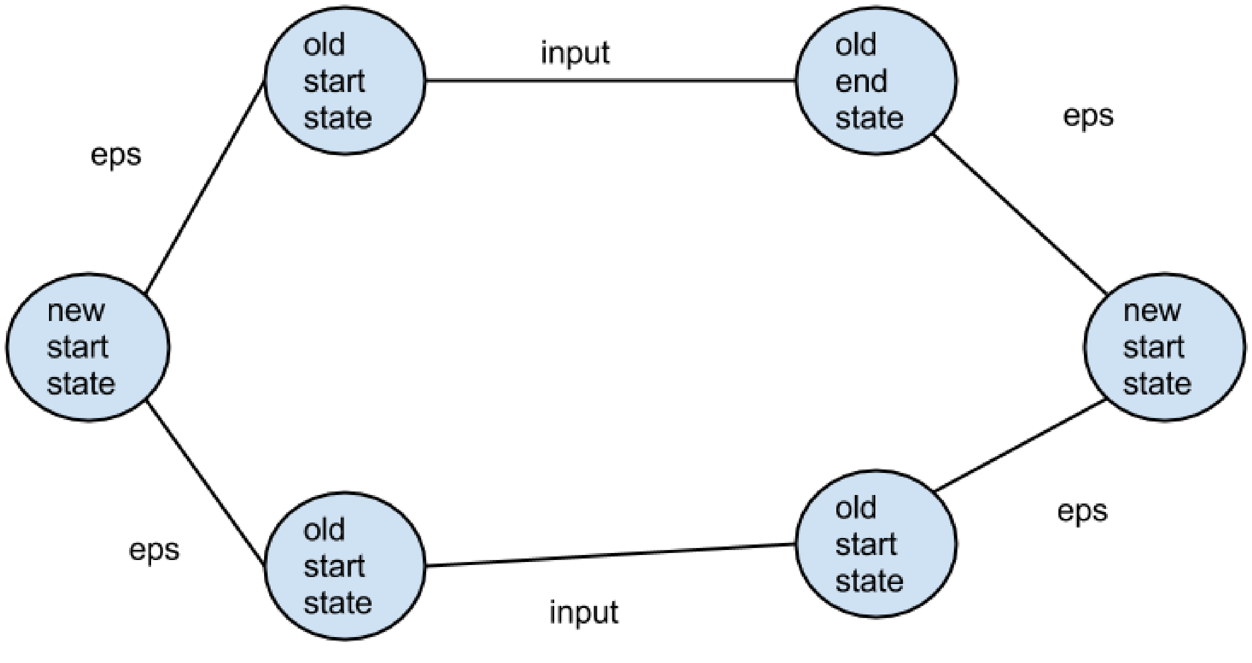
   1- The operator is (*) then we call method drawStar() which takes the last expression before the operator and makes a new expression for it , then push the new expression in the vector instead of the operator and the expression.



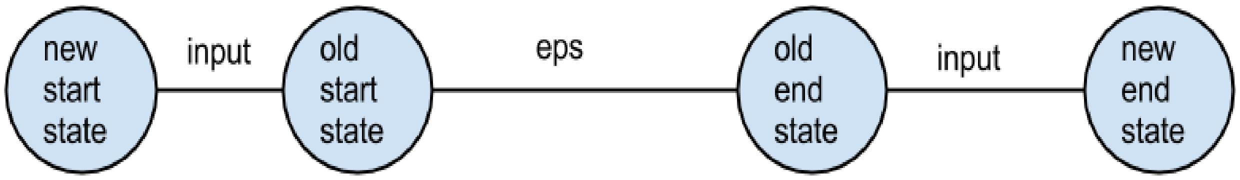
2- The operator is (+) which is treated as (*) but with different method drawPlus() and push back the new expression like before.



3- The operator is (|) then we call method drawOR() which takes the last two expressions before the operator and makes a new expression which consists of OR of two expressions
And push this new expression in the vector with new start state and old state instead of the operator and the two expression.



4- The operator is ( )"space character" which is treated as (|) but with different method drawConcatenation() and push the new expression in the vector



We have method (addTransation()) that takes parameters the first state , the second state , and the label of the edge , and then add this information as a Table_obj (end state, input) to the vector of the NFA's in the row of the start state.

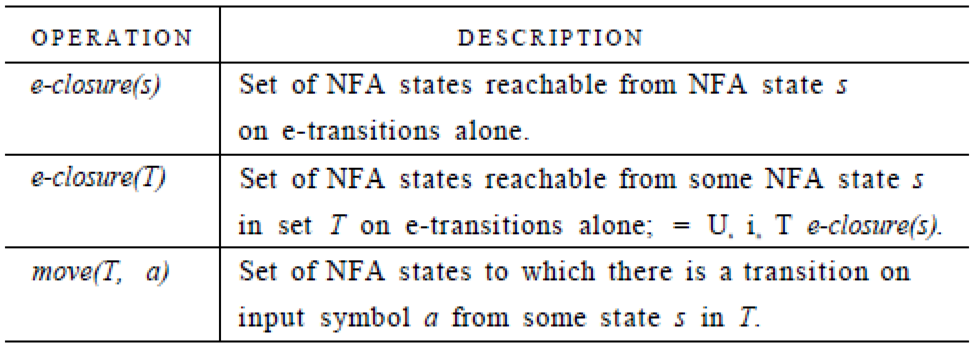
Also we have a method (addRows()) which add vectors for all the states that was created before .

Subset Construction Algorithm (Conversion from NFA to DFA)

Given the NFA in the form of vector of vectors of non-deterministic Nodes we generate the DFA in the form of vector of vectors of deterministic Nodes.

Each node either deterministic or non-deterministic has a unique identifier and the input of the transition.

The deterministic state has a vector of non-deterministic states that creates the deterministic one



OPERATION	DESCRIPTION
e-closure(s)	Set of NFA states reachable from NFA state *s* on e-transitions alone.
e-closure(T)	Set of NFA states reachable from some NFA state *s* in set *T* on e-transitions alone; = U, i, T *e-closure(s)*.
move(T, a)	Set of NFA states to which there is a transition on input symbol *a* from some state *s* in *T*.

The Algorithm consists of four main methods

1. Move (T, a)
   given input a and set of non-deterministic states T we return a set of states that I can reach from any state in T with input = a.

```
For (all states in the set T) {
      For (all transitions from state S in set T) {
         if (input = a)
             Add this state to the output states

      }
}
```

2. Epsilon-closure(T)

```
push all states of T onto stack;
Put all states of set T into output state
while (!stack is empty ) {
    pop the top element off stack;
        For ( each state u with an edge from t to u labeled e )
       If ( u is not in e-closure(T) ) {
          Add u to e-closure(T);
          push u onto stack;
       }
}
```

3. Generate-DFA(NFA,inputs)

```
initially, e-closure(s0) is the only state in Dstates, and it is unmarked;
while ( there is an unmarked state T in Dstates ) {
   mark T;
   for ( each input symbol a ) {
       U = e-closure(move(T,a));
       if ( U is not in Dstates )
          add U as an unmarked state to Dstates;
     }
}
```

4. Construct-DFA(DStates,inputs)

```
Let DFA be a vector of vectors
for(each state in Dstates ){
      let v a vector for each state transitions
      for(all inputs in the input sets){
           get the epsilon-closure of each non-deterministic state in D
           add this state in the vector v;
```

```
        }
        add v to the vector of vectors (DFA Graph)
}
```

The Minimization Of DFA :

In this phase we should minimize the DFA which we have it from the previous step.

The minimization algorithm :

From the pervious step we have a DFA which have a final states and non -final ones we can minimize the DFA to make it easier to use it in the simulation after this step .

In the first step we should divide the states to final and non-final list of sates , each time we should check if each state in the list goes to state in the same in the list ,if not we should divide the list and we should do this until we have lists of sates and in each state in the same list goes to the same list at the same input .

To implement this algorithm we should have in the first some vectors contain the statue of the states and map to the list of states to inform us the final state and the non final ones and the accepted patterns from the regular expression in the first step.

Data structures :

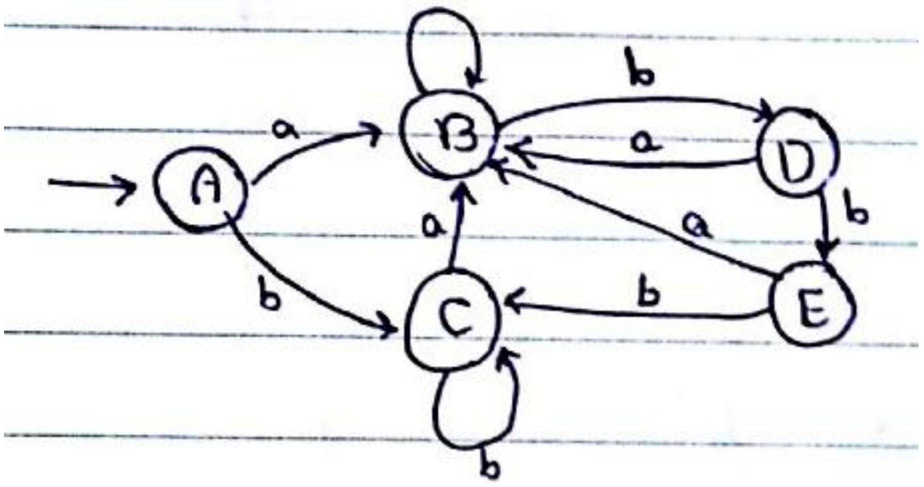
The main data structure we use it in the minimization phase is Disjoint set we use this data structure to union states with each other in a list and then check the inputs of each one of them to determine if they are In the same list or not .

We have also some vectors and lists to use them as a container to the final sates and the patterns …etc

Like :

vector<string> *patterns;

vector<bool> *isFinalVector;



We will explain the algorithm at an example of this figure .

In this DFA we have 5 states 1 final and 4 non finals , first we iterate on the list of states which I have from the pervious step and at the same time I iterate on the vector of sates which indicate whether state is final or not and have its accepted pattern and if the states are final states and have same accepted pattern put them in the same list by disjoint set and if it’s not final union it in the non final list also by the disjoint sets , after this step we have more than one list one of them is the non final states and the rest of them are final states with the same accepting pattern.

As we have only one accepting state in this example so the lists are {A,B,C,D} and {E}

We have main list which we put the lists in it after each step , at the first we pop a list from the main list and check if it will be divided or not if it divided we put the divided list to the main list ,if not we put the list to final list l called minimized list and we check the main list if it is empty then we finished the algorithm .After the first iteration the lists in the main list will be {A,B,C} ,{D},{E}

Then it will be {A,C},{B},{D},{E} and it will be the final minimized list .

Simulation of DFA:

In this phase we have the minimized table for the DFA , and there is an input file containing a program that we want to apply the states to it and get the accepted patterns , and indicate that there is an error if we go to unaccepted state.The algorithm goes as follow:

- We read the input character by character, we have a start state that we take the character as an input from it to another state.
- We take the longest prefix of the input , until there is a separator (space) and insert this lexeme to the symbol table.
- If the longest prefix reached to a state that we can’t go further , we should use panic error mode as will be discussed later.
- We have method (move ()) that returns the next state from the current one using that input.
- **NOTE** there are predefined tokens in symbol table { punctuations and key-words} so if the lexeme that is accepted { id} we search if it is already in the symbol table and insert it if not . For example {int } should be treated as regular id until we try to insert it in the symbol table and if it was recorded as key-word , it will not be inserted again and the token in this case is key-word.

Panic mode:

In this technique we take the longest prefix of the input until there is a state that we can’t go further. And if this state is final then we this lexeme is accepted. If not, then we remove characters from the current lexeme until we reach an accepting state, and treat the removed characters a new lexeme.

Sample Run:

For the following sample program

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

And the following regular expressions

```
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: \* | /
```

The minimization of DFA is

In the file " **minimization table output.txt**"

The output file

```
keyword
id
punc
id
punc
id
punc
id
punc
keyword
punc
id
relop
num
punc
punc
id
```

```
assign
id
addop
num
punc
punc
```

The output Symbol table

```
token name --> keyword  | token value -->  boolean
token name --> keyword  | token value -->  int
token name --> keyword  | token value -->  float
token name --> keyword  | token value -->  if
token name --> keyword  | token value -->  else
token name --> keyword  | token value -->  while
token name --> punc  | token value -->  ;
token name --> punc  | token value -->  ,
token name --> punc  | token value -->  (
token name --> punc  | token value -->  )
token name --> punc  | token value -->  {
token name --> punc  | token value -->  }
token name --> id  | token value -->  sum
token name --> id  | token value -->  count
token name --> id  | token value -->  pass
token name --> id  | token value -->  mnt
token name --> relop  | token value -->  !=
token name --> num  | token value -->  10
token name --> assign  | token value -->  =
token name --> addop  | token value -->  +
token name --> num  | token value -->  1
```

For the following sample program

int sum , count , pass , mnt;

for (int i=0;i< sum;i++)

{

pass = pass + 1 ;

}

And the following regular expressions

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: * | /

The output file

```
keyword

id

punc

id

punc

id

punc

id

punc

keyword

punc

keyword

id

assign

num

punc

id

relop

id

punc

id

addop
```

```
addop

punc

punc

id

assign

id

addop

num

punc

punc
```

The output Symbol table

```
token name --> keyword  | token value -->  boolean
token name --> keyword  | token value -->  int
token name --> keyword  | token value -->  float
token name --> keyword  | token value -->  if
token name --> keyword  | token value -->  else
token name --> keyword  | token value -->  while
token name --> keyword  | token value -->  for
token name --> punc  | token value -->  ;
token name --> punc  | token value -->  ,
token name --> punc  | token value -->  (
token name --> punc  | token value -->  )
token name --> punc  | token value -->  {
token name --> punc  | token value -->  }
token name --> id  | token value -->  sum
token name --> id  | token value -->  count
token name --> id  | token value -->  pass
token name --> id  | token value -->  mnt
token name --> id  | token value -->  i
token name --> assign  | token value -->  =
token name --> num  | token value -->  0
token name --> relop  | token value -->  <
token name --> addop  | token value -->  +
token name --> num  | token value -->  1
```

For the following sample program

```
for?? (int i=0;i< sum;i++)

{

pass = pass
```

And the following regular expressions

```
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: \* | /
```

The output file

```
keyword
error in : ?
error in : ?
punc
keyword
id
assign
num
punc
id
relop
id
punc
id
addop
addop
punc
punc
id
assign
```

The output Symbol table

```
token name --> keyword  | token value -->  boolean
token name --> keyword  | token value -->  int
token name --> keyword  | token value -->  float
token name --> keyword  | token value -->  if
token name --> keyword  | token value -->  else
token name --> keyword  | token value -->  while
token name --> keyword  | token value -->  for
token name --> punc  | token value -->  ;
token name --> punc  | token value -->  ,
token name --> punc  | token value -->  (
token name --> punc  | token value -->  )
token name --> punc  | token value -->  {
token name --> punc  | token value -->  }
token name --> id  | token value -->  i
token name --> assign  | token value -->  =
token name --> num  | token value -->  0
token name --> relop  | token value -->  <
token name --> id  | token value -->  sum
token name --> addop  | token value -->  +
token name --> id  | token value -->  pass
```