Project 1 . Building a Scanner

1. Specify the important data structure(s) you will use in your scanner algorithm. The specification should be independent of any specific programming language such as JAVA

   The chosen data structure in order to create the scanner algorithm is a two-dimensional array `transitionTable`. This transition table represents the available states and inputs that are generated through a text file. The first dimension of the transition table is from 1 to 16 where 1 represents `state 1`, 2 represents `state 2` and so on. The second dimension represents the character that is being fetched through a text file. This goes on from 1 to 14 where 1 represents the `tab` and `space` and 2 represent the `newline` character respectively up to 14 where it represents `non-valid characters`. As the scanner start fetching for an input character, it is then paired to the current `state` and by matching it up in the transition table, one can obtain a `value` which represents the `next state` if such input is recognized by the automata. If not recognized, it would have a value of 0, which can indicated `invalid token` or `tokenless state`. Another data structure that is used to support the `transitionTable` is a one-dimensional array named `stateToken`. This array corresponds to the name of the final state valid token after it evaluates the inputs successfully through the transition table. This is patterned in the given automata ($figure$ 2.6) of the textbook where `state 2` is represented as `div` and `state 6` as `lparen` and any other non-token as `error`.

2. The pseudo-code. For each function in your pseudo-code, you MUST have input, output and how output is related to input.

Algorithm 1: scan

input : fp: file pointer to the input
output : $token$: the valid token found from the file pointer; return $error$ otherwise.
side effect: file pointer fp will be moved into position based on the extracted valid token.
plan :

```
i := 1..n where n is the number of states
j := 1..m where m is the number of input character
transitionTable := array [i][j]
tokenTable := array [i] // represents the final state tokens
word := string // placeholder for input characters
state := 1 // start state
x := character // placeholder for an input char
do
    x := input character from fp
    num := colNum(x) // gets the correct column number for input
    lastState := state // storing last state to go back
    token := "error" // initally setting up token
    if transitionTable[state][num] != 0 // longest possible token rule
    then
        state := transitionTable[state][num] // exhaust possible input in a state
        put input char to word
    end
    else
        if tokenTable[state] is error // invalid token
        then
            token := "error" // error flag
        end
        else
            // valid token or tokenless state
            if tokenTable[lastState] is error // tokenless state
            then
                token := "error" // error flag
            end
            else if word is "read" // distinguish read token
            then
                token := "read"
            end
            else if word is "write" // distinguish write token
            then
                token := "write"
            end
            else
                token := tokenTable[lastState] // valid token
            end
            go back to start state // reset to start state again
            unread a character
            return token
        end
    end
while true;
```

Algorithm 2: colNum

input : *c*: character
output : *col*: column number of the character
side effect: N.A.
plan :

```
switch x do
    case space do
        col := 1
    end
    case newline do
        col := 2
    end
    case '/' do
        col := 3
    end
    case '*' do
        col := 4
    end
    case '(' do
        col := 5
    end
    case ')' do
        col := 6
    end
    case '+' do
        col := 7
    end
    case '-' do
        col := 8
    end
    case ':' do
        col := 9
    end
    case '=' do
        col := 10
    end
    case '.' do
        col := 11
    end
    case digit do
        col := 12
    end
    case letter do
        col := 13
    end
    otherwise do
        col := 14
    end
end
```

Algorithm 3: main

input : F: text file
output : N.A.
side effect: prints valid tokens stored in a $tokenVect$; if invalid, prints "$error$"
plan :

```
fp := file pointer // points to the start of the text file
tokenVect := vector
while fp is not End Of File do
    token := scan(fp) // grabs the token
    if token == error then
        print error and exit program
    end
    else
        tokenVect.push() // pushes the valid token in tokenVect
    end
end
print tokenVect
```

3. The test cases to test the correctness of your program with explanation why you select them
   To check the correctness of the program, we can test at least 3 test cases that would exhaust almost all possibilities of outcomes.

   First Test Case:

   ```
   /
   (
   )
   +
   -
   *
   :=
   .1
   1
   1.1
   abc123
   read
   write
   ```

   Exhausting all the states with valid token is one way to make sure that the program recognizes all of the valid tokens based on the automata on the textbook. Additionally, this will prove if the program follows the longest possible token rule for the identifier `abc123`. This will also make sure that the $transitionTable$ record has correct information on where the next state should go based on the given character input.

   Second Test Case:

   ```
   read
   /* foo
   bar */
   *
   five 5
   ```

   This is the given format in the pdf where the answer should be $(read, times, id, number)$. This will not only check to see if valid tokens are read correctly, but also confirm if the program does indeed ignore comments.

Third Test Case:

```
abc
123
=
```

Using this test case will prove that the program should print out $error$ as the automata would not accept an equal sign as is for it is not recognized.

4. Acknowledgment of people and their contribution to your project.

   I did this project by myself.