Design and Report for CS3361 Project 1

Team Members: Stephen Devaney, Christian Gonzalez, and Samuel Williamson.

1. Introduction

This project is an implication of the DFA Scanner discussed in the Concept of Programming Languages Course. The scanner is intended to find tokens (including read and write) based upon the regular expression for a calculator language given on page 54 of the course textbook (Programming Languages Pragmatics 4th Ed by Michael L Scott). The Scanner receives a file from the command line, runs the scanner over the input file, and outputs error if an improper token is found, otherwise it outputs the list of tokens found from the input file. This project utilizes a table and driver style scanner instead of the case-switch style scanner. As for the token data type this project will simply utilize the String data type to represent a token.

2. Data Structure

a) outputQueue: is a dynamically allocated queue of tokens. This queue will hold all tokens generated by the scanner from the input file for output after the scanner has successfully extracted all tokens from the file. The idea for the queue is to keep from having to reallocate memory in programming languages that require prior declarations of data types and size of data (ex C, C++). For languages that do not require prior declarations of data types and size of data (ex python) you can replace the queue with an array/list of tokens and just append the tokens on the end of the array/list. As required the queue will follow the normal definition of a queue (FIFO) and have the standard functions isempty, enqueue, and dequeue. The structure for the output Queue holds a head and tail pointers and an additional structure for the node that holds a token and a pointer (called next) to the next element.

b) transitionTable: is a two-dimensional array. The first dimension (i) is indexed from 0 to 17 to represent the states from 1 to 18. The second dimension (j) is indexed from 1 to 14 representing the characters/character types. When this transition table is index ie. transitionTable[i][j] the state at the corresponding index will indicate the next state. States with dashes in figure 2.12 of the textbook will either be replaced with a -1, 0 or a +1. If the next state is -1 the scanner is stuck and will output an error token and end the program as indicated in the project description. If the next state is 0 the scanner has recognized a token. If the next state is +1 the scanner has found white space or a comment and will reset the scanner back to the starting state in order to continue scanning for tokens.

c) tokenTable: is an one-dimensional array that holds the final output tokens based on the index. The token table in the textbook defines empty tokens as errors. An error token will be replaced in this data structure to allow for easier implementation of the scanner.

d) keywordTable: is an one-dimensional array that is used to check if the id equals a certain keyword that will be used as a token in its place such as read and write.

3. Algorithms (in pseudocode)

3.1 Function scan(...)

Input:
  inFile: file pointer from main function

Output:
  the token to be output to main program

Precondition:
  The current pointer of the input file is not at the end of the file.

Data:
  idCharacters: a list of characters to be compared to keywordTable if the token is an id
  cur_char: current character read in from inFile
  cur_state: current state of the scanner
  cur_char_index: the preset index the character represents in the transitionTable
  transitionTable: holds the next state based based off of the state index and character index
  tokenTable: holds the final output token based of the state
  keywordTable: holds keywords that also need to be represented as tokens (read, write)

Plan:

```
idCharacters := empty string
Prev_state := cur_state := 1 // start state
while file pointer is not at the end of the file and (cur_state is not error (-1) or recognize (0))
    cur_char_index := 0
    cur_char := current character read in from file
    case cur_char
        \n : cur_char_index := 1
        / : cur_char_index := 2
        * : cur_char_index := 3
```

```
            ( : cur_char_index := 4
            ) : cur_char_index := 5
            + : cur_char_index := 6
            - : cur_char_index := 7
            : : cur_char_index := 8
            = : cur_char_index := 9
            . : cur_char_index := 10
            digit: cur_char_index := 11
            letter: cur_char_index := 12
            Otherwise: cur_char_index := 13
        case transitionTable[cur_state-1, cur_char_index]
            Any number besides 0 or -1: // move to next state
                cur_state := transitionTable[cur_state-1, cur_char_index]
                if tokenTable[cur_state-1] is id
                    if idCharacters current length is less than 5 // 5 is the longest keyword length
                        append the cur_char on to the end of the string
            0: // recognize token
               // scan will ignore comment and white space and move on to next function
                exit loop as a precondition to while loop
            -1: // error! token not found
                exit loop as a precondition to while loop
    if tokenTable[prev_state-1] is id
        iterate through keywordTable
            if keywordTable[index] matches idCharacters
                return keywordTable[Index]
    return tokenTable[prev_state-1]
```

3.2 function isempty()

input:
    queue: queue to be checked to verify weather or not that it is empty

Precondition:
    queue: is initialized

Output:
    returns a boolean value indicating that both head and tail pointers in the queue are null

Data:
    queue.head: is the head pointer for the queue
    queue.tail: is the tail pointer for the queue

```
Plan:{
return if queue.head and queue.tail pointers are null
}
```

3.3 function enqueue()

input:
    queue: list of tokens that an additional token needs to be added to
    token: token to be added to queue

Precondition:
    queue: is initialized

Output:
    adds a token to the end of the queue

Data:
    newNode: is the node to be added to the queue
    queue.head: is the head pointer for the queue
    queue.tail: is the tail pointer for the queue

```
Plan:{
newNode’s data = token
newNode’s next element = null
if queue is empty
    queue.head and queue.tail pointers = newNode
else
    queue tail’s next element = newNode
    queue tail pointer = newNode
}
```

3.4 function dequeue()

input:
    queue: list of tokens that an additional token needs to be added to

Precondition:
    queue: is initialized and not empty

Output:

```
    outputToken: token that is removed from the queue
    tempNode: temporary node to hold the element being removed

Data:
    queue.head: is the head pointer for the queue
    queue.tail: is the tail pointer for the queue

Plan:{
outputToken = empty token
if queue is not empty
    outputToken = queue front's token
    tempToken = queue front
    Queue front = queue front's next element
    free memory allocated for outputToken
    if queue front is null
        queue rear = null //  sets queue to empty
return outputToken
}
```

3.5 Main algorithm:

```
Input:
    fileName: File name given from command line on the console

Output:
    outputQueue: queue that holds all tokens identified by the scanner
    curToken: curToken received from the scanner used to check for an error

Data:
    inFile: file pointer to hold the address of the file in memory given to be scanned

Plan:
    inFile := open file fileName
    outputQueue := empty queue
    while not at the end of inFile
        curToken := scan(inFile) // if an error is raised the function will return an error token
        if curToken != error
            outputQueue enqueues tempToken
        else:

             break from loop
```

```
if curToken == error
    print curToken
else
    while outputQueue is not empty:
        print and remove first element in outputQueue
```

4. Test Cases

An example for one of our test cases, where we have our Input File, Command Line, and Output:

1.

<u>Input</u>: test1.txt

```
write
         /* test
           line */
*
three 3
```

<u>Command Line:</u>

```
scanner text1.txt
```

<u>Output</u>:

```
(write, times, id, number)
```

2.

<u>Input</u>: test2.txt

```
read
         /* test
           line */
*
four 4
```

<u>Command Line:</u>

```
scanner text2.txt
```

<u>Output</u>:

```
(read, times, id, number)
```

3.

Input: test3.txt

```
read
/* test
        line */
5 *  5 @
```

Command Line:

```
scanner text3.txt
```

Output:

```
error.
```

Acknowledgement

Thank you Dr. Zhang for slides and lectures on scanners in Concepts of Programming Languages. Thank you Michael L. Scott for writing the course textbook that teaches about scanner applications.