Project 1 CS3361

Blane Brown, Tucker Hoffnagle, Samantha Holmes

1. Introduction

This is the CS3361 project created by Blane Brown who contributed the pseudo code, Tucker Hoffnagle who contributed the test cases and bug fixing and Samantha Holmes who contributed the data structures. This document covers the data structures used for this project, the pseudo code of the table-driven scanner algorithm used and some test cases to show our program is functioning properly. The pseudo code has been implemented in Java and the program runs perfectly with no errors.

2. Data Structures

There are three data structures present in our solution:

1. **nextStateTable**
- nextStateTable is a 2d integer array with two indexes. The first dimension is indexed from 1 to 16 for the number of states. The second is indexed from 1 to 14 with 1 representing space/tab, 2 representing newline, 3 representing the character /, 4 representing the character *, 5 representing the character (, 6 representing the character), 7 representing the character +, 8 representing the character -, 9 representing the character :, 10 representing the character =, 11 representing the character ., 12 representing a digit, 13 representing any letter and 14 representing any other character.
- for any integer i and j, nextStateTable[i][j] is either a number greater than 0 meaning move to that state, a 0 meaning there is nowhere else to move and to recognize the token, or a -1 meaning the character is invalid (such as a '$').

2. **tokenTypes**
- tokenTypes is a one-dimensional string array containing every possible token type. The index of each token type correlates to the final state it represents.
-for any state i, tokenTypes[i] gives either the type of token that was found for that final state or an error token if the state given was not a final state.

3. **tokens**
- tokens is an arraylist that contains all found tokens
- any token returned by the scan() function while scanning the string is added to this arraylist.

3. Algorithms

3.1 Main() Function

The main function is responsible for running the program until the end of the file is met, or for our implementation until the string is empty. It will continue to call scan until the string is empty or an error token is returned. Once completed it will print out the tokens gathered from scan()

Main()

Input: a file F containing a program.
Output: all token types discovered in F are printed to the user in the order they were found.
Side Effect: the file f is read into a string s so that other functions can act on s.

Given Data:

F: the file containing a program

Data:

s: input file characters written into a string (string)
token: type of token discovered (string)
tokenList: sequence of all tokens found (type of token not its content) (list)

Plan:

```
s := open file F and read its contents into s
while (s is not empty) //not at EOF
        token := Scan()
        if (token not equal to ERROR)
                if (token not equal to start) //ignore whitespace
                        add token to tokenList
        else
                print ERROR
                break loop
print tokenList
```

3.2 Scan() Function

The scan function is responsible for acting on string s and returning found tokens. A pointer to the first character in the string is created and once a token type is returned, every character included in the token is removed from the string and the scan function continues to be called until the string is empty meaning EOF or an error token is returned.

Scan()

Input: acts on global variable string s.
Precondition: s is not empty (EOF).
Output: a token; t is a token if the substring of s starting from the beginning to the index forms a token, otherwise t is an error.
Side Effect: if t is a token, remove substring from s corresponding to this token.

Data:

index: counter of where we are at in the input string s (integer)
currState: current state we are in (integer)
currChar: current character being looked at in the string (character)
token: name of the token (string)

Plan:

```
index := 0
currState := 1
currChar := character at index of s
token := “”
```

```
//while loop is used so we can get the longest possible token
while (index is less than the length of s – 1) //if index is equal to s – 1 we have one char left in s
        if (TransitionTable(currChar, currState) is not equal to 0)
                if(TransitionTable(currChar, currState) is equal to -1)
                        return ERROR //invalid character found
                index++
                currState := TransitionTable(currChar, currState)
                currChar := character at index of s
        else
                if (FinalStateToToken(currState, token) is not equals to ERROR)
                        token := substring of s from 0 to index
                        s := substring of s from index to the length of s
                        return FinalStateToToken(currState, token) //token found
                else
                        return ERROR

if (FinalStateToToken(currState, token) is equal to 0)
        token := substring of s from 0 to index
        s := substring of s from index to the length of s
        return FinalStateToToken(currState, token)

currState := TransitionTable(currChar, currState)
token := s
s := "" //empty
return FinalStateToToken(currState, token)
```

3.3 TransitionTable(character, integer) Function

The transition table function is responsible for finding out what state needs to be traveled to next given an input of a current state and current character.

TransitionTable(character currChar, integer currState)

Input: current character and current state.

Output: return next state; ns is the next state given from the transition table taking in the input of currChar and currState.

Data:

nextStateTable: determines where to go next (2d integer array)

currCharNumber: used to convert a character to a number since you can't index an array with a character (integer)

Plan:

nextStateTable[states][characters] := 2d array containing transition states

```
{ 1,  1,  2,  10,  6,  7,  8,  9,  11,  -1,  13,  14,  16,  -1 },
{ 0,  0,  3,   4,  0,  0,  0,  0,   0,   0,   0,   0,   0,  -1 },
{ 3,  1,  3,   3,  3,  3,  3,  3,   3,   3,   3,   3,   3,   3 },
{ 4,  4,  4,   5,  4,  4,  4,  4,   4,   4,   4,   4,   4,   4 },
{ 4,  4,  1,   5,  4,  4,  4,  4,   4,   4,   4,   4,   4,   4 },
{ 0,  0,  0,   0,  0,  0,  0,  0,   0,   0,   0,   0,   0,  -1 },
{ 0,  0,  0,   0,  0,  0,  0,  0,   0,   0,   0,   0,   0,  -1 },
{ 0,  0,  0,   0,  0,  0,  0,  0,   0,   0,   0,   0,   0,  -1 },
{ 0,  0,  0,   0,  0,  0,  0,  0,   0,   0,   0,   0,   0,  -1 },
{ 0,  0,  0,   0,  0,  0,  0,  0,   0,   0,   0,   0,   0,  -1 },
{ 0,  0,  0,   0,  0,  0,  0,  0,   0,  12,   0,   0,   0,  -1 },
{ 0,  0,  0,   0,  0,  0,  0,  0,   0,   0,   0,   0,   0,  -1 },
{ 0,  0,  0,   0,  0,  0,  0,  0,   0,   0,   0,  15,   0,  -1 },
{ 0,  0,  0,   0,  0,  0,  0,  0,   0,   0,  15,  14,   0,  -1 },
{ 0,  0,  0,   0,  0,  0,  0,  0,   0,   0,   0,  15,   0,  -1 },
{ 0,  0,  0,   0,  0,  0,  0,  0,   0,   0,   0,  16,  16,  -1 },
```

//-1 in the array means error should be returned

switch case of currChar
- case space/tab: currCharNumber := 1
- case newline: currCharNumber := 2
- case slash: currCharNumber := 3
- case asterisk := 4
- case left parenthesis := 5
- case right parenthesis := 6
- case plus := 7
- case minus := 8
- case colon := 9
- case equals := 10
- case period := 11
- case number := 12
- case identifier: currCharNumber := 13
- case other: currCharNumber := 14 //character found is not in our languages alphabet so return ERROR unless in a comment

return nextStateTable[currState - 1][currCharNumber - 1]

3.4 FinalStateToToken(integer, string) Function

The final state to token function is responsible for turning an input of a final state into a string of a token type and returning it.

FinalStateToToken(integer currState, string token)

Input: current state we are at and the characters that make up the found token.

Output: a token type; tt is a token type corresponding to the current state were in. If were not in a final state, then tt will be “ERROR”. If a keyword was given as input then tt is that keyword.

Data:

tokenTypes: used to determine what token name to return (string array)

Plan:

```
tokenTypes := {"start", "div", "ERROR", "ERROR", "ERROR", "lparen", "rparen", "plus", "minus",
"times", "ERROR", "assign", "ERROR", "number", "number", "id"}
if (currState == -1)
        return ERROR
else if (token == read) //recognize read or write as their own token types
        return read
else if (token == write)
        return write
else
        return tokenTypes[currState - 1]
```

4. Test Cases

Some test cases we have used while testing the program are:

1. you see me
   // you don't see me
   total := (a + 3.14 / 2 * 2 - 1) write total

2. /* invalid $ character allowed
   In comment */ but not
   Out51d3 of it

3. This should give error $ =

The first test allows us to see that newline comments are working along with different operators and that write is returned as a token "write" instead of "id". The second test allows us to see that multiline comments work and that invalid characters such as $ are allowed inside comments. It also demonstrates the longest possible token working as "Out51d3" is returned as just "id". The third test shows that incomplete tokens such as "=" which is missing its ":" or invalid characters such as $ returns error.

5. Unit Test

We know this wasn't required but we accidentally made this table because we originally thought this is what was wanted for test cases. We decided to leave it in our design because we like how it turned out and spent quite a while making it. It also helps confirm our program works as intended.

Tested Function	Reason	Input	Expected Output
Main()	This test would make sure the program can properly find a listed file.	scanner testtxt.txt	String s = testtxt.txt contents

Main()	This test would make sure the program throws an error if a listed file is not found.	scanner anything.txt	Prints ERROR File not found.
Main()	This test would make sure the main function properly prints the correct tokens.	token = id token = number	Prints [id, number]
Main()	This test would make sure the program properly prints an ERROR message if an improper token is read from the file.	token = ERROR	Prints ERROR
Scan()	This test would make sure a file with valid tokens returns the tokens one at a time. This test also makes sure that comments are not recognized as tokens.	/* This is a comment */ //this is a comment 3	Returns number
Scan()	This test makes sure a file with invalid tokens returns an error.	$	Returns ERROR
TransitionTable(character, integer)	This test makes sure the transition table works properly by returning the next state to travel to given your current character and current state.	TransitionTable('a', 1)	Returns 16
TransitionTable(character, integer)	This test makes sure the transition table returns negative one because the given character is not in the automata's alphabet.	TransitionTable('$', 1)	Returns -1
FinalStateToToken(integer, string)	This test makes sure the proper token type is returned given the final state and the	FinalStateToToken(16, "abc")	Returns id

	characters that make up the current token.		
FinalStateToToken(integer, string)	This test makes sure that the error token is returned given that the state we are in is not a final one.	FinalStateToToken(-1, “$”)	Returns ERROR
FinalStateToToken(integer, string)	This test makes sure that the keyword token “read” is returned instead of “ID”, given that we are in a final ID state and the characters that make up the current token is read.	FinalStateToToken(16, “read”)	Returns read
FinalStateToToken(integer, string)	This test makes sure that the keyword token “write” is returned instead of “ID”, given that we are in a final ID state and the characters that make up the current token is write.	FinalStateToToken(16, “write”)	Returns write