Report

# Introduction

I am writing this report to lay out the important data structures of a program I will write, and the code’s general structure. The program must implement a DFA based on the DFA from book *Programming Language Pragmatics FOURTH EDITION* in figure 2.6.

# Data Structures

These are the data structures I will be using for my project.

**token\_list**

* + vector
  + Holds the recognized tokens from input
  + Exists in DFA class.
  + Will be printed out once the DFA destructor is called.

**transition\_table**

* + 2D Array which columns 0-13 will have the characters to be recognized. There will be a bijection from the numbers 0-13 to the recognizable tokens which is mapped by *token\_table*.
  + Will have rows indexed 0-17 which will represent the current state of the DFA.
    - Let *i* be the row index. At row index *i*, the current state represented by *i* is *i+1*.
  + The elements are the transition states.
  + The function “scan” will use this table to return the transition state. By reading a character, and checking the current state of the DFA, the “scan” function will return the transition state.
    - If a character is scanned that does not lead to anything else in the array, *the scan function will determine if it is the final state (a valid token), or if an invalid token has been encountered.*
    - An important note is that the driver in figure 2.12 does not always consider the longest possible token rule.
      * To remedy this, I used the number -2 to mean that an error state has been encountered, not just a dead-end state when needed.
      * For more details, please refer to the appendix at the end of this document.
  + Exists in DFA class

**token\_table**

* + array
  + the list of recognizable tokens which will be used
  + Exists in DFA class
  + Bijection from chars to 0-13 exists here.
  + “error” tokens will be in indices that represent a dead state from the Scanner table.

**DFA**

* + Members
    - token\_list
    - transition\_table
    - token\_table
    - cur\_state
    - prev\_state
    - token
  + Description
    - Class
    - Holds most components necessary to implement the DFA

# Pseudo-code

## Pseudo-code for DFA Structure

**data**

token\_list: vector which elements are tokens scanned

token\_table: array of recognizable tokens except "read" and "write"

transition\_table: 2D Array w/ transition info

**plan**

**Function** get\_index

**Pass In:** character

**Pass Out:** corresponding integer for transition table

Map out each of the possible symbols of the DFA with integers 0- 13. Once a match has been found for the symbol, return the corresponding column index (integer) that symbol maps to.

**Endfunction**

**Function** scan

**PRECONDITION:** input\_stream cannot be end of file

**Pass In:** input\_stream

**Pass Out:** token to token\_list

current\_state: current state of DFA

cur\_char: current character being scanned

prev\_state: state before finding dead-end

WHILE (no error flag) **or** (token found)

cur\_char := input\_stream.get\_char()

prev\_state := current\_state

current\_state := transition\_table[index(current\_state)][get\_index(cur\_char)]

//index() just subtracts 1 from the input state to index transition //table

IF (current\_state = dead\_end) and (token\_table[index(prev\_state)] = final\_state) THEN

token\_list.push(token\_table[index(prev\_state)]) //push token

ELSEIF [(current\_state = dead\_end AND token\_table(prev\_state) =/= final state) OR (current\_state = dead\_state)] THEN

token\_list.push(“error”) //return error flag to token\_list

PRINT "error."

EXIT\_PROGRAM

ENDIF

ENDWHILE

**Endfunction**

## Pseudo-code for Main Function

**given data**

file\_name := INPUT user\_input

**data**

input\_stream := convert file input into a stream of characters

DFA: DFA object from data class with scan function & more

**plan**

WHILE input\_stream is not empty

DFA.scan(input\_stream)

ENDWHILE

PRINT DFA.token\_list //happens in DFA destructor

**endofplan**

# Test Cases

For my test cases, I will choose the following cases to test my program’s correctness.

1. A file with multiple lines with all possible valid tokens. (true positive)
2. Many files with valid tokens, and reasonable amount of invalid tokens. (true negative)
3. A file with only invalid tokens. (true negative)
4. An empty file to test if “scan” function recognizes that the precondition of EOF is true. (true negative). The scan function is not called if file is at EOF.

# Acknowledgement

Thanks to Dr. Yuanlin Zhang for the useful sample report.

# Appendix

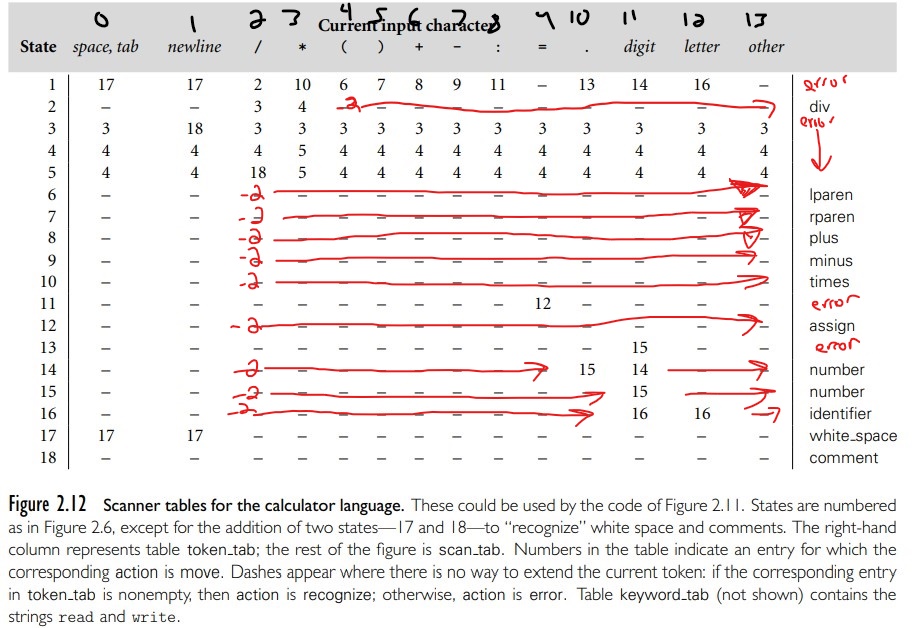


Figure 1: Modified Scanner Table for my Program. The red arrows represent what dashes should be replaced with -2’s. The dashes by default mean dead-end which I represent In my program with -1; however, a -1 is not enough to determine if it is either a final state or a dead-state. For example: if we are in state 6, and we scan a mult character immediately after, then we are for sure at a dead-end. But since this would follow the “longest token” rule, this should be a dead state, and not lparen. By leaving -1 in whitespace and -2 for everything else, I am better able to implement my driver located in my scan function.