Jahir Montes

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
      ELSE IF [(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

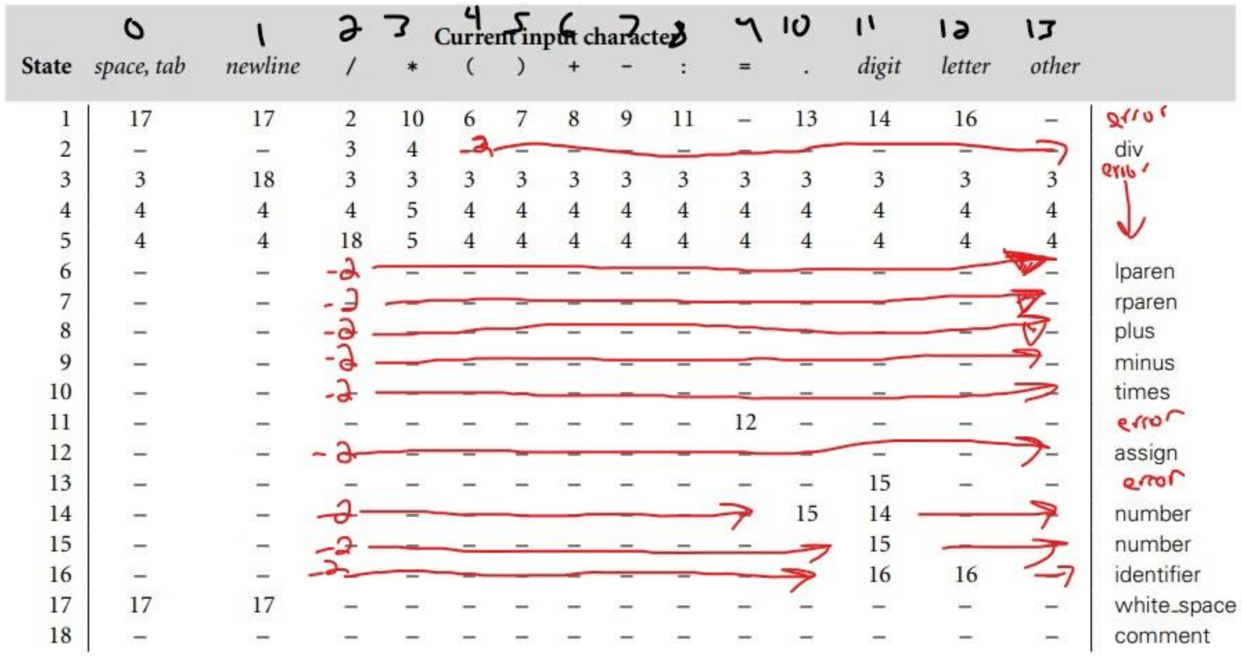
For my test cases, I will choose the following cases to test my program's correctness.

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



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
	Current input character														
State	*space, tab*	*newline*	/	*	(	)	+	−	:	=	.	*digit*	*letter*	*other*	
1	17	17	2	10	6	7	8	9	11	–	13	14	16	–	error
2	–	–	3	4	-2	–	–	–	–	–	–	–	–	–	div
3	3	18	3	3	3	3	3	3	3	3	3	3	3	3	error
4	4	4	4	5	4	4	4	4	4	4	4	4	4	4	
5	4	4	18	5	4	4	4	4	4	4	4	4	4	4	
6	–	–	-2	–	–	–	–	–	–	–	–	–	–	–	lparen
7	–	–	-2	–	–	–	–	–	–	–	–	–	–	–	rparen
8	–	–	-2	–	–	–	–	–	–	–	–	–	–	–	plus
9	–	–	-2	–	–	–	–	–	–	–	–	–	–	–	minus
10	–	–	-2	–	–	–	–	–	–	–	–	–	–	–	times
11	–	–	–	–	–	–	–	–	–	12	–	–	–	–	error
12	–	–	-2	–	–	–	–	–	–	–	–	–	–	–	assign
13	–	–	–	–	–	–	–	–	–	–	–	15	–	–	error
14	–	–	-2	–	–	–	–	–	–	–	15	14	–	–	number
15	–	–	-2	–	–	–	–	–	–	–	–	15	–	–	number
16	–	–	-2	–	–	–	–	–	–	–	–	16	16	–	identifier
17	17	17	–	–	–	–	–	–	–	–	–	–	–	–	white_space
18	–	–	–	–	–	–	–	–	–	–	–	–	–	–	comment

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.