COMP.2030 HW 3: Scanner Fall 2019

**DUE: 10/14 (Mon) 11:59 PM**

In HW 2, the type of each character in an input string was determined. You are to write a scanner in this assignment. Scanner first reads a line from the console and stores characters into an input buffer of maximum 80 characters (getline()). Scanner then combines these characters into logical units called tokens, and returns the resulting table of tokens for the line.

Scanner puts the tokens into a table with two columns. The first column saves the assembled tokens of maximum 8 characters and the second column, their token types. Each entry in the token table has 12 bytes.

Token types are identical to character types used in HW 2, but do not include the blank char type any longer.

Token type 1 -- Number : 0 1 .. 9

Token type 2 -- Variable : Alphabet followed by alpha-numeric characters

Token type 3 -- Operator : \* + - /

Token type 4 -- Delimiter : . ( ) , : $

Token type 5 -- End of Line : #

For example, given a line of input below,

THISLOOP: LWU R2,63 #

the resulting token table becomes as follows (table header may be skipped in the program).

Token Token Type

---------------------------- ----------------

THISLOOP 2

: 4

LWU 2

R2 2

, 4

63 1

# 5

This type of output has to be repeated for EVERY line that is scanned, as in the previous assignment. In the program, when you use TabToken for the token table, you need to reserve an enough space for TabToken table corresponding to a single input line (20 entries of 3 words each should be sufficient). The same token table will be overwritten each time a new input line is processed.

The behavior of the Scanner can be described (defined) by a state (transition) diagram shown below. One could write a huge subroutine for the entire diagram. This results in a large monolithic program which may be quite difficult to debug. Furthermore, the approach locks the programmer to the specifics of the diagram and makes later modification difficult. It may, however, improve execution efficiency somewhat.

The approach we are going to take is more general, resulting in small, easily debuggable functions. We treat the diagram as a finite state automata. We simply encode the diagram into a state table and have Scanner simulate the finite automata. Namely, Scanner simply traces through the states in the diagram beginning in state Q0. At each state, Scanner calls the action subroutine associated with the state and them computes the next state to make a transition to based on the value of the variable T. The variable TOKEN holds the token string as it is being assembled. The variable, TokSpace, is initialized to 8, and is decremented each time a new character (byte) is appended to the current token in TOKEN.

curChar =Get next char, T=ChType(curChar) (ACT 1)

Q0

T=5

T=1

TOKEN=curChar, TokSpace=7

(ACT 2)

Q1

curChar =Get next char, T=ChType(curChar) (ACT 1)

T=6

T=3,4

T=2

curChar =Get next char, T=ChType(curChar) (ACT 1)

Save TOKEN

into TabToken

Clear TOKEN

(ACT 4)

Q4

Q5

Q3

Q2

RETURN

Q0

Save TOKEN

into TabToken

Clear TOKEN

(ACT 4)

T=1,2

T != 1,2

T != 1

T=1

Save TOKEN

into TabToken

Clear TOKEN

(ACT 4)

Q8

Q9

Q7

Q6

TOKEN=TOKEN+curChar TokSpace=TokSpace-1 (ACT 3)

TOKEN=TOKEN+curChar TokSpace=TokSpace-1 (ACT 3)

Q1

Q1

The same information specified through the state transition diagram of a state automation can be expressed in a tabular form. You should fill in missing entries in the state table.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| State\T |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Q0 | ACT1 | Q1 | Q1 | Q1 | Q1 | Q1 | Q1 | Q10 |
| Q1 | ACT2 | Q2 | Q5 | Q3 | Q3 | Q0 | Q4 | Q10 |
| Q2 | ACT1 | Q6 | Q7 | Q7 | Q7 | Q7 | Q7 | Q10 |
| Q3 | ACT4 |  |  |  |  |  |  |  |
| Q4 | RETURN | Q4 | Q4 | Q4 | Q4 | Q4 | Q4 | Q10 |
| Q5 | ACT1 |  |  |  |  |  |  |  |
| Q6 | ACT3 |  |  |  |  |  |  |  |
| Q7 | ACT4 |  |  |  |  |  |  |  |
| Q8 | ACT3 |  |  |  |  |  |  |  |
| Q9 | ACT4 |  |  |  |  |  |  |  |
| Q10 | ERROR | Q4 | Q4 | Q4 | Q4 | Q4 | Q4 | Q4 |

The value of variable T will be dependent upon the type of each character on the line according to the character types used in HW 2 (**not** to be confused with token types, although similar):

Characters Type Comment

------------------ ------ -------------

0 1 .. 9 1 Digits

A B .. Z a b .. z 2 Letters

\* + - / 3 Operators

. ( ) , : 4 Delimiters

blank 5 blank

# 6 End of the Line

The state transition table can be constructed as follows:

Q0: .word ACT1

.word Q1 # T=1

.word Q1 # T=2

.word Q1 # T=3

.word Q1 # T=4

.word Q1 # T=5

.word Q1 # T=6

.word Q11 # T=7

Q1: .word ACT2, Q2, Q5, ….

….

The algorithm for tracing through the states and a section of the state table are given below. CUR holds the current state and T has the current value of character type.

Scanner Algorithm

1. Call getLine
2. CUR = Q0; T=1
3. ACT = STAB[CUR][0]

CALL ACT

1. CUR = STAB[CUR][T]
2. GO TO 3

**Use $s0 and $s1 to hold the value of T and CUR**, respectively. Steps 2 through 4 can be coded in MIPS as follows:

la $s1, Q0

li $s0, 1

nextState: lw $s2, 0($s1)

jalr $v1, $s2 # Save return addr in $v1

sll $s0, $s0, 2 # Multiply by 4 for word boundary

add $s1, $s1, $s0

sra $s0, $s0, 2

lw $s1, 0($s1)

b nextState

In this assignment, you need to write four short functions (ACT1, ACT2, ACT3 and ACT4) and include the following state table, STAB, in the data section. You are free to handle ERROR function.

**Notes:**

* In order to print the token table one row at a time, the last byte of the third word of a token (the word for the token type) can be set to ‘\n’ to force the printer to move to the next line.
* Printing stops as soon as a byte in the output string has Null (0x00). The variable TOKEN and TabToken therefore can be initialized to all ‘ ‘ (blank, 0x20) instead of all 0’s.

STAB:

Q0: .word ACT1

.word Q1 # T1

.word Q1 # T2

.word Q1 # T3

.word Q1 # T4

.word Q1 # T5

.word Q1 # T6

.word Q11 # T7

Q1: .word ACT2

.word Q2 # T1

.word Q5 # T2

.word Q3 # T3

.word Q3 # T4

.word Q0 # T5

.word Q4 # T6

.word Q11 # T7

Q2: .word ACT1

.word Q6 # T1

.word Q7 # T2

.word Q7 # T3

.word Q7 # T4

.word Q7 # T5

.word Q7 # T6

.word Q11 # T7

Q3: .word ACT4

.word Q0 # T1

.word Q0 # T2

.word Q0 # T3

.word Q0 # T4

.word Q0 # T5

.word Q0 # T6

.word Q11 # T7

Q4: .word ACT4

.word Q10 # T1

.word Q10 # T2

.word Q10 # T3

.word Q10 # T4

.word Q10 # T5

.word Q10 # T6

.word Q11 # T7

Q5: .word ACT1

.word Q8 # T1

.word Q8 # T2

.word Q9 # T3

.word Q9 # T4

.word Q9 # T5

.word Q9 # T6

.word Q11 # T7

Q6: .word ACT3

.word Q2 # T1

.word Q2 # T2

.word Q2 # T3

.word Q2 # T4

.word Q2 # T5

.word Q2 # T6

.word Q11 # T7

Q7: .word ACT4

.word Q1 # T1

.word Q1 # T2

.word Q1 # T3

.word Q1 # T4

.word Q1 # T5

.word Q1 # T6

.word Q11 # T7

Q8: .word ACT3

.word Q5 # T1

.word Q5 # T2

.word Q5 # T3

.word Q5 # T4

.word Q5 # T5

.word Q5 # T6

.word Q11 # T7

Q9: .word ACT4

.word Q1 # T1

.word Q1 # T2

.word Q1 # T3

.word Q1 # T4

.word Q1 # T5

.word Q1 # T6

.word Q11 # T7

Q10: .word RETURN

.word Q10 # T1

.word Q10 # T2

.word Q10 # T3

.word Q10 # T4

.word Q10 # T5

.word Q10 # T6

.word Q11 # T7

Q11: .word ERROR

.word Q4 # T1

.word Q4 # T2

.word Q4 # T3

.word Q4 # T4

.word Q4 # T5

.word Q4 # T6

.word Q4 # T7