German University in Cairo
Department of Computer Science
Assoc. Prof. Haythem O. Ismail

CSEN 1003 Compiler, Spring Term 2018
Practice Assignment 2

Discussion: 05.02.19 - 11.02.19

Exercise 2-1

Consider the following input string: `aaabaabbababbb`

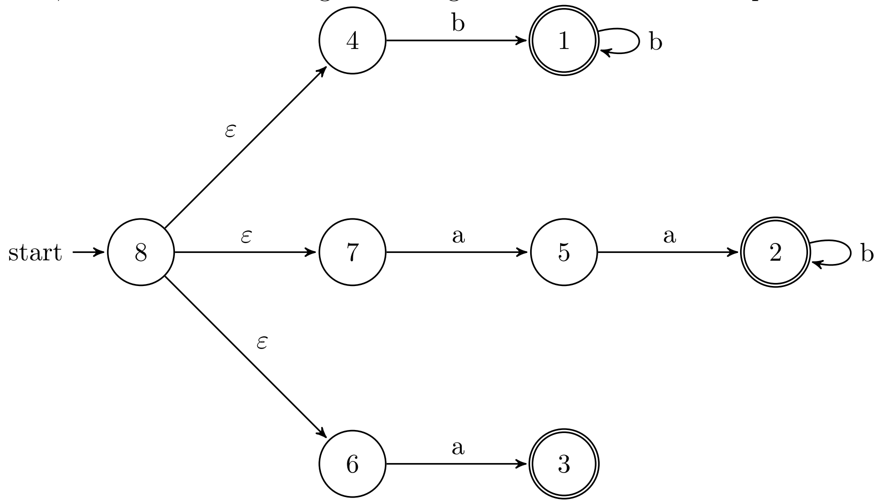
a) and the action-augmented regular definition:

$$\begin{array}{lll} 1 & \longrightarrow & \texttt{b}^+ & \{\texttt{printf("1")}\} \\ 2 & \longrightarrow & \texttt{aab}^* & \{\texttt{printf("2")}\} \\ 3 & \longrightarrow & \texttt{a} & \{\texttt{printf("3")}\} \end{array}$$

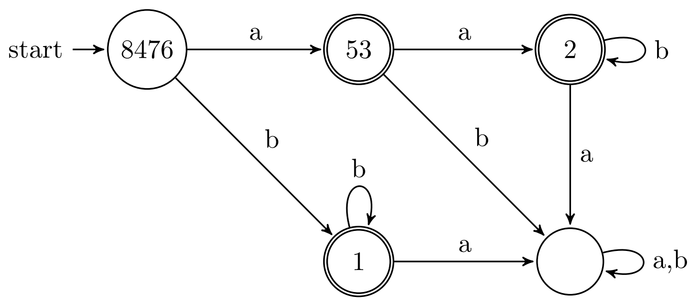
Draw the state diagram of an equivalent fallback DFA with actions. What will be printed when the DFA is run of the provided input?

Solution:

First, convert the action-augmented regular definition into the equivalent NFA



Then, convert the NFA into DFA with actions



The action of state 53 is {printf("3“)}, state 2 is {printf("2“)} and state 1 is {printf("1“)}
The main issue here is to use the two common practice rules of tokenization:

1. Pick the longest possible string that belong to the regular language, and
2. if two translation rules apply use the rule given first in the grammar.

Note that the order in which the common practice rules are applied matters!

The correct tokenization is: `aa a b aabb a b a bbb`

The output of the corresponding actions will be: `23123131`

b) Repeat for the following action-augmented regular definition.

$$\begin{array}{lcll} 4 & \longrightarrow & \texttt{(aa)}^*\texttt{b}^* & \{\texttt{printf("2")}\} \\ 3 & \longrightarrow & \texttt{a} & \{\texttt{printf("3")}\} \end{array}$$

Solution:

It differs from the rules above in the sense that we allow the first rule to match the empty string.

The tokenization is the same: `aa a b aabb a b a bbb`

And the resulting output is: `23223232`

Exercise 2-2

For the following action-augmented regular definition, give a regular expression describing the language of possible outputs. Assume that all inputs are strings of 0's and 1's only.

$$\begin{array}{lcll} 5 & \longrightarrow & \texttt{0} & \{\texttt{printf("c")}\} \\ 6 & \longrightarrow & \texttt{00} & \{\texttt{printf("a")}\} \\ 7 & \longrightarrow & \texttt{1} & \{\texttt{printf("b")}\} \end{array}$$

Solution:

An even length of 0's prints all a's, while an odd length string of 0's will have one c at the end (because of the maximal munch rule). Thus, strings of 0s generate the language $a^*c?$.

Interspersed 1's generate b's, so the full language is: $(a^*c?b^+)^*a^*c?$

A common mistake might be to incorrectly account for the priority between the rules for 0 and 00.

Exercise 2-3

Give a regular definition for non-negative integers without leading zeros. A zero is represented by a single `0`.

Solution:

$$\begin{array}{lcl} \textit{digit} & \rightarrow & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \\ \textit{nonzerodigit} & \rightarrow & \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \\ \textit{integer} & \rightarrow & \texttt{0} \mid \textit{nonzerodigit}\ \textit{digit}^* \end{array}$$

Exercise 2-4 (Based on an exercise from the textbook)

Write an action-augmented regular definition for a C-style string literal. A string literal starts and ends with double-quotes (`"`) and any character in between. Any `"` appearing between the initial and final double-quotes must be escaped by preceding it with a backslash (\). Hence, a backslash in the string must be represented by two backslashes. The actions should produce a token $\langle \textbf{lit}, s \rangle$, where s is the string without the enclosing double-quotes and the escape backslashes.

Solution:

$Char$	$\longrightarrow$	$[a-zA-Z0-9]$	
$String$	$\longrightarrow$	$char^*$	A(String) = if table.isOpen() table.append(String) else do-something-else
DBS	$\longrightarrow$	\\	A(DBS) = if table.isOpen() table.add('\') else do-something-else
DQ	$\longrightarrow$	"	A(DQ) = if table.isOpen() {table.close(); return <lit, table.last()>} else table.open()
$BSDQ$	$\longrightarrow$	\"	A(BSDQ) = if table.isOpen() table.add('"') else do-something-else

Exercise 2-5

In this exercise, you will write an action-augmented regular definition to process sequences of Haskell-style lists of non-negative integers. A list of non-negative integers has two alternate representations:

a) A comma-separated sequence of non-negative integer literals between `[` and `]`.

b) A non-negative integer literal, followed by a `:` , followed by a list of non-negative integers.

The actions should produce a sequence of tokens for `[`, `]`, `,`, and non-negative integers, converting lists of the second form to those of the first. For example, on input `1:2:[3]`, the output should be

$$\langle \mathbf{LB} \rangle, \langle \mathbf{num}, \mathtt{1} \rangle, \langle \mathbf{comma} \rangle, \langle \mathbf{num}, \mathtt{2} \rangle, \langle \mathbf{comma} \rangle, \langle \mathbf{num}, \mathtt{3} \rangle, \langle \mathbf{RB} \rangle$$

Show the output on input `[12,13]4:[16][]`.

Solution:

Num	$\longrightarrow$	$[0-9]^+$	
$Head1$	$\longrightarrow$	$[Num$	$A(Head1) = \{return(\langle \mathbf{LB} \rangle, \langle \mathbf{num}, Num \rangle)\}$
$Head$	$\longrightarrow$	Num	$A(Head) = \{return(\langle \mathbf{LB} \rangle, \langle \mathbf{num}, Num \rangle)\}$
$NestedHead$	$\longrightarrow$	$:[Num$	$A(NestedHead) = \{return(\langle \mathbf{comma} \rangle, \langle \mathbf{num}, Num \rangle)\}$
$Body1$	$\longrightarrow$	$:Num$	$A(Body1) = \{return(\langle \mathbf{comma} \rangle), \langle \mathbf{num}, Num \rangle)\}$
$Body2$	$\longrightarrow$	$,Num$	$A(Body2) = \{return(\langle \mathbf{comma} \rangle), \langle \mathbf{num}, Num \rangle)\}$
$EmptyList$	$\longrightarrow$	$[]$	$A(EmptyList) = \{return(\langle \mathbf{LB} \rangle, \langle \mathbf{RB} \rangle)\}$
$NestedEmptyList$	$\longrightarrow$	$:[]$	$A(EmptyList) = \{return(\langle \mathbf{RB} \rangle)\}$
RB	$\longrightarrow$	$]$	$A(RB) = \{return(\langle \mathbf{RB} \rangle)\}$

The output of input `[12,13]4:[16][]` will be:

$\langle \mathbf{LB} \rangle, \langle \mathbf{num}, \mathtt{12} \rangle, \langle \mathbf{comma} \rangle, \langle \mathbf{num}, \mathtt{13} \rangle, \langle \mathbf{RB} \rangle, \langle \mathbf{LB} \rangle, \langle \mathbf{num}, \mathtt{4} \rangle, \langle \mathbf{comma} \rangle, \langle \mathbf{num}, \mathtt{16} \rangle, \langle \mathbf{RB} \rangle, \langle \mathbf{LB} \rangle, \langle \mathbf{RB} \rangle$