Overview

We define the language A to be a particular set of strings (defined below) that represent valid arithmetic expressions operating on floating-point numbers, with the entire expression contained between %# and >. For this assignment you are to draw a state diagram of a PDA that recognizes this language and write a program that implements your PDA.

The Language A

To precisely define the language A, we first define the context-free grammar $G = (V, \Sigma, R, S)$, where $V = \{S, B, H, Y, N\}$ is the set of variables; the alphabet is

$$\Sigma = \{ ., 0, 1, 2, \ldots, 9, +, -, *, /, (,), \%, \#, > \}, \quad (1)$$

which includes a dot for float-point numbers; the starting variable is S; and the rules R are

$$\begin{aligned} S &\rightarrow \%\#B> \\ B &\rightarrow B+B \mid B-B \mid B*B \mid B/B \mid (B) \mid H \\ H &\rightarrow Y.Y \mid Y. \mid .Y \\ Y &\rightarrow NY \mid N \\ N &\rightarrow 0 \mid 1 \mid 2 \mid \cdots \mid 9 \end{aligned}$$

Then we define the language $A = L(G)$, which contains strings that begin with %# and end with >, and in between is an arithmetic expression over floating-point numbers. For example, the string “%#(15.-(6.312*.7))>” belongs to A, which we can show by using the derivation

$$\begin{aligned} S &\Rightarrow \%\#B> \Rightarrow \%\#(B)> \Rightarrow \%\#(B\text{-}B)> \Rightarrow \%\#(B\text{-}(B))> \\ &\Rightarrow \%\#(B\text{-}(B*B))> \Rightarrow \%\#(H\text{-}(B*B))> \Rightarrow \%\#(Y\ .\text{-}(B*B))> \Rightarrow \%\#(NY\ .\text{-}(B*B))> \Rightarrow \\ &\%\#(NN.\text{-}(B*B))> \Rightarrow \%\#(1N.\text{-}(B*B))> \Rightarrow \%\#(15.\text{-}(B*B))> \Rightarrow \%\#(15.\text{-}(H*B))> \Rightarrow \\ &\%\#(15.\text{-}(Y\ .Y*B))> \Rightarrow \%\#(15.\text{-}(N.Y*B))> \Rightarrow \%\#(15.\text{-}(6.Y*B))> \Rightarrow \\ &\%\#(15.\text{-}(6.NY*B))> \Rightarrow \%\#(15.\text{-}(6.NNY*B))> \Rightarrow \%\#(15.\text{-}(6.NNN*B))> \\ &\Rightarrow \%\#(15.\text{-}(6.3NN*B))> \Rightarrow \%\#(15.\text{-}(6.31N*B))> \\ &\Rightarrow \%\#(15.\text{-}(6.312*B))> \Rightarrow \%\#(15.\text{-}(6.312*H))> \\ &\Rightarrow \%\#(15.\text{-}(6.312*.Y\ ))> \Rightarrow \%\#(15.\text{-}(6.312*.N))> \\ &\Rightarrow \%\#(15.\text{-}(6.312*.7))> \end{aligned}$$

The above derivation is given so that you can get a better understanding of the language A, but the derivation cannot be directly used to design a PDA for A. The grammar G does not include the rule $B \rightarrow -B$ nor the rule $B \rightarrow \varepsilon$, so the strings “%#-45.0>” and “%#.8+-9.>” do not belong to A. Also, note that operands must be floating-point numbers, so integers (numbers without a dot) are not allowed; e.g., “%#45+3>” does not belong to A. The grammar G is ambiguous; e.g., the string

%#1.+2.*3.> ∈ *A* has two different parse trees.

PDA for *A*

First you are to construct a *deterministic* PDA $M = (Q, \Sigma, \Gamma, \delta, q_1, F)$ that recognizes *A*, where Σ is defined in equation (1). The PDA *M* must satisfy the following conditions:

The PDA must be defined with the alphabet Σ defined in equation (1). In other words the PDA must be able to handle any string of symbols from Σ. The PDA can handle certain strings not in *A* by crashing, i.e., the drawing does not have an edge leaving a state corresponding to particular symbols read and popped.

The PDA must have exactly one accept state.

The states in the PDA must be labeled $q_1, q_2, q_3, \ldots, q_n$, where q_1 is the start state and *n* is the number of states in the PDA. (It is also acceptable for the states to be labeled $q_0, q_1, \ldots, q_{n-1}$, with q_0 the start state.)

Each edge in the PDA must correspond to reading a symbol from Σ; i.e., no edge can correspond to reading *ε*. There is no restriction on pushing or popping *ε* on transitions.

Leaving the start state is exactly one edge with label "%, $\varepsilon \rightarrow$ %". Thus, the first thing the PDA does is it reads %, pops nothing, and pushes % on the stack.

Any edge going into the accept state has label ">, % → *ε*".

Your PDA must be deterministic; i.e., there must be exactly one way of processing each input string, which could be by crashing.

You will not be able to use the algorithm in Lemma 2.21 to convert the CFG *G* into a PDA for *A* since the resulting PDA will not be deterministic, as required. Implementing a deterministic PDA as a program should be straightforward. (Implementing a nondeterministic machine is more difficult since the program would need to check every branch in a tree of computation.)

The drawing of your PDA must include all edges that are ever used in accepting a string in *A*. But to simplify your drawing, those edges that will always lead to a string being rejected should be omitted. Specifically, when processing a string on your PDA, it might become clear at some point that the string will be rejected before reaching the end of the input. For example, if the input string is "%#34.5+*6.29>", then it is clear on reading the * that the string will not be accepted. Moreover, if an input string ever contains the substring +*, then the input string will be rejected. Thus, your drawing should omit the transition corresponding to reading the operator * in this case, so the PDA drawing will crash at this point.

In this project, the machine you design needs to only *recognize* the language *A*, not to *evaluate* the arithmetic expression. Because of this, your PDA does not need to distinguish between different Arabic numerals; e.g., there should be no difference

in reading the symbol 3 or the symbol 6 in the input. Thus, you can define notation such as $\Sigma_N = \{ 0, 1, 2, \ldots, 9 \}$ to denote the set of digits (Arabic numerals), and you can use this notation in labeling certain transitions. Also, all operators (+, -, *, /) can be handled similarly.

Test Cases

Test your program on each of the following input strings:

1. %#43.51386>
2. %#.78+27.-3.013/837.842+>
3. %#48622.+.41*1.2/00.1/521.23-.9+.53/7.>
4. %#382.89*14>
5. %#4.91-.*17.9>
6. %#44.88.6+3.208>
7. %#(1.2+(3.5-.9)/19).3>
8. %#(.4)64>
9. %#((824.23+(9.22-00.0)*21.2))+((.2/7.))>
10. %#(())>
11. %#((14.252+(692.211*(.39+492.1))/49.235)>
12. %#+6.5>
13. %#26.0*(.87/((4.+.2)/(23.1)-2.9)+6.)/(((823.*.333-57.*8.0)/.33+.0))>
14. %#.0*(32.922+.7-*9.))>
15. %#(4.+(.8-9.))/2.)*3.4+(5.21/34.2>