Principles of Programming, Spring 2006
Practice 6

Daejun Park, Heejong Lee
Programming Research Lab.@SNU

April 13, 2006

1. Consider the following set operators:

$$\begin{array}{rcl} \texttt{member?} & : & \mathit{element} \times \mathit{set} \rightarrow \mathit{bool} \\ \texttt{adjoin} & : & \mathit{element} \times \mathit{set} \rightarrow \mathit{set} \\ \texttt{union} & : & \mathit{set} \times \mathit{set} \rightarrow \mathit{set} \\ \texttt{intersection} & : & \mathit{set} \times \mathit{set} \rightarrow \mathit{set} \end{array}$$

   `member?` is a predicate that determines whether a given element is a member of a set. `adjoin` takes an object and a set as arguments and returns a set that contains the elements of the original set and also the adjoined element. `union` computes the union of two sets and `intersection` computes the intersection of two sets.

   One way to represent a set is as a list of its elements in which no element appears more than once. Another way to speed up our set operations is to change the representation so that the set elements are listed in increasing order. We can do better than the ordered-list representation by arranging the set elements in the form of a tree.

   Define three kinds of the above set operators, unordered list, ordered list, binary tree, respectively. Furthermore define the procedure `list2tree` that convert a set as unordered list into the set as binary tree.

$$\texttt{list2tree} \;:\; \mathit{set}_{\mathit{unordered\ list}} \rightarrow \mathit{set}_{\mathit{binary\ tree}}$$

2. Consider the following procedures:

$$\begin{array}{rclcl} d & \in & \mathit{digit} & \rightarrow & 0 \mid 1 \mid \cdots \mid 9 \\ s & \in & \mathit{string} & \rightarrow & \epsilon \mid d \mid d \cdot s \\ c & \in & \mathit{code} & \rightarrow & \epsilon \mid d \mid c \cdot c \mid c \texttt{|} c \mid c\texttt{+} \end{array}$$

$$\begin{array}{rcl} \texttt{match} & : & \mathit{string} \times \mathit{code} \rightarrow \texttt{bool} \\ \texttt{matchs} & : & \mathit{string} \times \mathit{code\ set} \rightarrow \texttt{bool} \\ \texttt{first} & : & \mathit{code} \rightarrow \mathit{digit\ set} \\ \texttt{rest} & : & \mathit{digit} \times \mathit{code} \rightarrow \mathit{code\ set} \end{array}$$

Informally, the `matchs` extends the `match` to *code set*, the `first(`c`)` means a set of first digit of a string that matches with c, and the `rest(`d,c`)` means a set of code representing the string s, such that $d \cdot s$ matches with c. The following is the formal definitions of these procedures.

$$\begin{aligned}
\parallel \epsilon \parallel &\doteq \{\epsilon\} \\
\parallel d \parallel &\doteq \{d\} \\
\parallel c_1 \cdot c_2 \parallel &\doteq \{s_1 \cdot s_2 \mid s_1 \in \parallel c_1 \parallel \ \wedge\ s_2 \in \parallel c_2 \parallel\} \\
\parallel c_1 \,|\, c_2 \parallel &\doteq \parallel c_1 \parallel \ \cup\ \parallel c_2 \parallel \\
\parallel c\texttt{+} \parallel &\doteq \parallel c \parallel \ \cup\ \{s \cdot s \mid s \in \parallel c \parallel\} \ \cup\ \{s \cdot s \cdot s \mid s \in \parallel c \parallel\} \ \cup\ \cdots
\end{aligned}$$

$$\begin{aligned}
\texttt{match}(s, c) &\doteq \begin{cases} \mathit{true}, & s \in \parallel c \parallel \\ \mathit{false}, & \text{otherwise} \end{cases} \\
\texttt{matchs}(s, C) &\doteq \textstyle\bigvee_{c \in C} \texttt{match}(s, c) \\
\texttt{first}(c) &\doteq \{d \mid d \cdot s \in \parallel c \parallel\} \\
\texttt{rest}(d, c) &\doteq \{c' \mid d \cdot s \in \parallel c \parallel \ \wedge\ s \in \parallel c' \parallel\}
\end{aligned}$$

By the above definitions, we can conclude the following equality.

$$\parallel c \parallel \ = \ \{d \cdot s \mid d \in \texttt{first}(c) \ \wedge\ c' \in \texttt{rest}(d, c) \ \wedge\ s \in \parallel c' \parallel\}$$

However we cannot use the above definitions to implement the procedures, since the size of $\parallel c \parallel$ is infinite. Therefore we have to modify the definition into a computable one. Compelete modified definitions in the following.

$$\begin{aligned}
\texttt{match}(s, \epsilon) &\doteq s = \epsilon \\
\texttt{match}(s, d) &\doteq s = d \\
\texttt{match}(s, c_1 \cdot c_2) &\doteq s = d \cdot s' \ \wedge\ d \in \texttt{first}(c_1 \cdot c_2) \ \wedge\ \texttt{matchs}(s', \texttt{rest}(d, c_1 \cdot c_2)) \\
\texttt{match}(s, c_1 \,|\, c_2) &\doteq \texttt{match}(s, c_1) \ \vee\ \texttt{match}(s, c_2) \\
\texttt{match}(s, c\texttt{+}) &\doteq \boxed{}
\end{aligned}$$

$$\begin{aligned}
\texttt{matchs}(s, \phi) &\doteq \boxed{} \\
\texttt{matchs}(s, C) &\doteq \textstyle\bigvee_{c \in C} \texttt{match}(s, c)
\end{aligned}$$

$$\begin{aligned}
\texttt{first}(\epsilon) &\doteq \{\epsilon\} \\
\texttt{first}(d) &\doteq \{d\} \\
\texttt{first}(c_1 \cdot c_2) &\doteq \boxed{} \\
\texttt{first}(c_1 \,|\, c_2) &\doteq \boxed{} \\
\texttt{first}(c\texttt{+}) &\doteq \boxed{}
\end{aligned}$$

$$\begin{aligned}
\texttt{rest}(d, \epsilon) &\doteq \phi \\
\texttt{rest}(d, d') &\doteq \begin{cases} \{\epsilon\}, & d = d' \\ \phi, & d \neq d' \end{cases} \\
\texttt{rest}(d, c_1 \cdot c_2) &\doteq \{c \cdot c_2 \mid c \in \texttt{rest}(d, c_1)\}^\dagger \\
\texttt{rest}(d, c_1 \,|\, c_2) &\doteq \boxed{} \\
\texttt{rest}(d, c\texttt{+}) &\doteq \texttt{rest}(d, c) \cup \boxed{}
\end{aligned}$$

† Note that if $\texttt{rest}(d, c) = \phi$, then $\{c_1 \cdot c_2 \mid c_1 \in \texttt{rest}(d, c)\} = \phi$.