Problem Set 5: Data Types

Problem 1

Define the predicate \((\text{list-of? } \text{pred})\) that, when applied to an argument \(\text{pred}\), returns a procedure that takes a list as argument. For example,

\[
> \text{(define list-of-numbers? (list-of? number?))}
> \text{(list-of-numbers? '(1 2 3 4 5))}
#t
> \text{(define list-of-symbols? (list-of? symbol?))}
> \text{(list-of-symbols? '(a b c d))}
#t
> \text{(list-of-symbols? '(a 2 c 3))}
#f
\]

Thus, \((\text{list-of? } \text{pred})\) has to return a procedure that when applied to a list returns \#t if the application of \(\text{pred}\) to all elements in that list returns \#t. Otherwise, this procedure has to return \#f.

Problem 2

Consider the following BNF specification:

\[
<\text{bin-search-tree}> \ ::= \ ()
\quad \mid \ (\text{<string> }<\text{bin-search-tree}> <\text{bin-search-tree}>)
\]

The procedure \((\text{path-to-key key bst})\), where \(\text{key}\) is a string and \(\text{bst}\) is a binary search tree that contains the string \(\text{key}\), returns a list of the symbols \(\text{left}\) and \(\text{right}\) showing how to find the node that contains the string \(\text{key}\). If the string \(\text{key}\) is found at the root, it returns the empty list. You can use the built-in Scheme string predicates \((\text{string<? string1 string2})\) and \((\text{string>? string1 string2})\) to implement the procedure \(\text{path-to-key}\). For example,

\[
(\text{path-to-key } "R" \ '("Q" ("H" () ("O" () ()))))
(\ "W" ("T" ("R" () ()))
() )
("Z" () () )
\]

\(\text{return}\)

\((\text{right left left})\)
Problem 3

Consider the following BNF specification:

\[
\text{Stack} ::= () \quad \text{empty-stack} \\
\text{Stack} ::= \text{<SchemeValue> Stack} \quad \text{extended-stack (value tail)}
\]

- Using the \texttt{define-datatype} abstraction, define the abstract syntax of data type \texttt{Stack} according to the BNF specification given above. Use as variant names the name given in the boxes.
- Using the data type \texttt{Stack} and the \texttt{cases} construct, define the procedure \texttt{(is-empty? stk)} that returns \#t if \texttt{stk} is empty. Otherwise this procedure returns \#f.
- Using the data type \texttt{Stack} and the \texttt{cases} construct, define the procedure \texttt{(push val stk)} that takes any Scheme value and a stack and returns a new stack with \texttt{val} as the top element. If the argument \texttt{stk} is not a \texttt{Stack} use the procedure \texttt{error} to print an error message (e.g. \texttt{(error "Argument is not a Stack!")}).
- Using the data type \texttt{Stack} and the \texttt{cases} construct, define the procedure \texttt{(pop stk)} that takes a stack \texttt{stk} and returns a new stack where the top element has been removed. If the argument \texttt{stk} is not a \texttt{Stack} or the \texttt{stk} is empty, use the procedure \texttt{error} to print an error message.
- Using the data type \texttt{Stack} and the \texttt{cases} construct, define the procedure \texttt{(top stk)} that takes a stack \texttt{stk} and returns the top element of \texttt{stk}. If the argument \texttt{stk} is not a \texttt{Stack} or the \texttt{stk} is empty, use the procedure \texttt{error} to print an error message.
Problem 4

Consider the following BNF specification:

\[<\text{RoseTree}> ::= () \mid (\text{<Number>} \{<\text{RoseTree}>\}^\star)\]

- Using the define-datatype abstraction, define the abstract syntax of data type \text{RoseTree} according to the BNF specification given above. Use as variant names the name given in the boxes. You need to use the predicate \text{list-of?} defined in Problem 2.

- Define the procedure \((\text{parse-rose-tree}\; \text{lst})\) that when applied to a list representation of a rose tree returns its abstract syntax tree. Note, it may be necessary to decompose the grammar (i.e., it is useful to define a separate procedure for the Kleene-Star component). You also need to check for correct input syntax. For example,

\[> (\text{parse-rose-tree}\; '(1 () (3 (4 (4 () (5))) (3) (2))))\]

\[(\text{rt-node}\;1\;((\text{empty-rt})\;((\text{rt-node}\;3\;((\text{rt-node}\;4\;((\text{empty-rt})\;((\text{rt-node}\;5\;())))))))\;((\text{rt-node}\;3\;())\;((\text{rt-node}\;2\;())))))\]

- Define the procedure \((\text{size}\;\text{rt})\) that determines the size of a rose tree according to the following definition: If \(\text{rt} \in <\text{RoseTree}>\), then

\[\begin{align*}
\text{rt} = () \quad (\text{empty-rt}) & : (\text{size}\;\text{rt}) = 1, \\
\text{rt} = (\text{n rts}) \quad (\text{rt-node}\;\text{n}\;\text{rts}) & : (\text{size}\;\text{rt}) = 2 + \sum_i^n (\text{size}\;\text{rt}_i), \text{where}\;\text{rt}_i \in \text{rts}.
\end{align*}\]

For example:

\[> (\text{size}\; (\text{parse-rose-tree}\; '(1 () (3 (4 (4 () (5))) (3) (2))))))\]

16
Problem 5

Most programming languages support data structures, called records (or structures), whose elements are accessed by name. Consider the following interface for a record data type:

\[
\begin{align*}
\text{(empty-record)} &= \{\emptyset\} \\
\text{(select-field } [f] s) &= (f s) \\
\text{(extend-record } [f] ((s_1 v_1) (s_2 v_2) \ldots (s_n v_n))) &= [g] \\
\end{align*}
\]

where \( (g s') = \begin{cases} v_i & \text{if } s' == s_i \text{ and } 1 \leq i \leq n \\ (f s') & \text{otherwise} \end{cases} \)

The procedure empty-record, applied to no argument, must produce a procedural representation of the empty record; select-field applies a procedural representation of a record to an argument (a field name), and \( \text{(extend-record } a-record '((s_1 v_1) (s_2 v_2) \ldots (s_n v_n))) \) produces a new (procedural) representation of a record that behaves like a-record, except that its value at symbol \( s_i \) is \( v_i \) (a name-value binding) for all \( i \) with \( 1 \leq i \leq n \). For example, the Pascal-like record

\[
dxy-record := \text{record end with } d = 6; x = 7; y = 8; \text{end};
\]

may be constructed and accessed as follows:

\[
> \text{(define dxy-record}
> \quad \text{(extend-record (empty-record) '((d 6)(x 7)(y 8))))}
> \text{(select-field dxy-record 'x)}
\]

7

Following the approach shown in EOPL2 and in the lecture notes, implement all three procedures (i.e., \( \text{(empty-record)}, \text{(select-field record name)}, \) and \( \text{(extend-record record pairs)} \)). Find a way to reuse the procedures \( \text{list-find-position} \) and \( \text{list-ref} \) (as shown in class), so that both can be applied to pairs of name-value bindings. At the moment, the procedure \( \text{list-find-position} \) expects a list of symbols whereas and the procedure \( \text{list-ref} \) expects a list of values. However, record extension uses a list of pairs of symbols and values. Do not change the procedures \( \text{list-find-position} \) and \( \text{list-ref} \). Instead, find a way to construct the right argument type.

Submission deadline: Thursday, March 22, 2007, 2:10 p.m.
Submission procedure: on paper in class.