Problem Set 5: Data Types

Problem 1

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

```scheme
> (define list-of-numbers? (list-of? number?))
> (list-of-numbers? '(1 2 3 4 5))
#t
> (define list-of-symbols? (list-of? symbol?))
> (list-of-symbols? '(a b c d))
#t
> (list-of-symbols? '(a 2 c 3))
#f
```

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

Solution:

```scheme
(define fold-left
  (lambda (f e lst)
    (if (null? lst)
        e
        (fold-left f (f e (car lst)) (cdr lst))
    ))
)

(define list-of?
  (lambda (pred)
    (lambda (lst)
      (fold-left (lambda (x y) (and x (pred y))) #t lst)
    ))
)
```

20
Problem 2

Consider the following BNF specification:

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

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,

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

Solution:

; PathToKey.scm

; key is an element of the bst
(define path-to-key
  (lambda (key bst)
    (if (null? bst)
        '()
      (if (equal? (car bst) key)
          '()
        (if (string<? (car bst) key)
            (cons 'right (path-to-key key (caddr bst)))
          (cons 'left (path-to-key key (cadr bst))))
    )
  )
)

Problem 3

Consider the following BNF specification:

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

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

(define-datatype Stack Stack?
  (empty-stack)
  (extended-stack (value (lambda (x) #t))
    (tail Stack?))) 5

(define is-empty? 5
  (lambda (stk)
    (cases Stack stk
      (empty-stack () #t)
      (else #f)
    ))
)

(define push 10
  (lambda (val stk)
    (if (Stack? stk)
      (extended-stack val stk)
      (error "Argument is not a Stack!")
    ))
)

(define pop 10
  (lambda (stk)
    (if (Stack? stk)
      (cases Stack stk
        (extended-stack (value tail) tail)
        (else "Stack is empty!")
      )
      (error "Argument is not a Stack!")
    ))
)

(define top 10
  (lambda (stk)
    (if (Stack? stk)
      (cases Stack stk
        (extended-stack (value tail) value)
        (else "Stack is empty!")
      )
      (error "Argument is not a Stack!")
    ))
)
Problem 4

Consider the following BNF specification:

\[ <\text{RoseTree}> ::= () | (\langle\text{Number}\rangle \{<\text{RoseTree}>\}^*) \]

- 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 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,

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

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

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

  - \(rt = ()\) (i.e., \text{empty-rt}): \(\text{size rt} = 1\),
  
  - \(rt = (n \text{ rts})\) (i.e., \text{rt-node n rts}): \(\text{size rt} = 2 + \sum_i^n \text{size rt}_i\), where \text{rt}_i \in \text{rts}.

For example,

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

16
Solution:
(define list-of?
  (lambda (pred)
    (lambda (lst)
      (if (null? lst)
          #t
          (and
           (pred (car lst))
           ((list-of? pred) (cdr lst))
         ))))

(define-datatype RoseTree RoseTree?
  (empty-rt)
  (rt-node
   (n number?)
   (rts (list-of? RoseTree?)))
)

(define parse-rose-tree
  (lambda (lst)
    (if (null? lst)
        (empty-rt)
        (rt-node (n (car lst))
                 (parse-rose-trees (cdr lst)))))
)

(define parse-rose-trees
  (lambda (lsts)
    (if (null? lsts)
        '()
        (cons (parse-rose-tree (car lsts))
              (parse-rose-trees (cdr lsts)))))
)

(define size
  (lambda (rt)
    (cases RoseTree rt
      (empty-rt () 1)
      (rt-node (n rts)
               (+ 2 (sum (map size rts)))))))
)

(define sum
  (lambda (lon)
    (if (null? lon)
        0
        (+ (car lon) (sum (cdr lon)))))
)

--
25
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:

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

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

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\((\text{extend-record} \text{ } a\text{-record}'((s_1 \text{ } v_1) (s_2 \text{ } v_2) \ldots (s_n \text{ } v_n)))\) 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

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

may be constructed and accessed as follows:

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

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

(define empty-record
  (lambda ()
    (lambda (s) (eopl:error 'select-field "No binding for ~s" s)))
  )
)

(define select-field
  (lambda (r s)
    (r s)
  )
)

(define extend-record
  (lambda (r pairs)
    (lambda (s)
      (let ((pos (list-find-position s (map car pairs))))
        (if (number? pos)
            (list-ref (map cadr pairs) pos)
            (select-field pairs s))
      )
    )
  )
)

Total: 20 + 20 + 40 + 25 + 30 = 135

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