Problem Set 2: Introduction to Scheme & Induction

Problem 1
Implement the procedure count that, when applied to a list, returns the number of elements in the list. For example, \( \text{count '(1 2 3 4 5 6)} \) returns 6. The procedure \text{count} can be implemented using the built-in procedures \text{null?}, \text{car}, \text{cdr}, +, and the procedure \text{count} itself (i.e., \text{count} is a recursive procedure).

Solution:

\[
\text{Solution:} \\
\begin{align*}
\text{(define count} & \text{(lambda (lon) )} \\
& \text{(if (null? lon) 0) } \\
& \text{(+ 1 (count (cdr lon))) )} \\
\end{align*}
\]

Problem 2
Implement the procedure \( \text{(take lst n)} \) that returns the first \( n \) elements of \( \text{lst} \). If \( n \) is greater than the length of \( \text{lst} \), then simply return \( \text{lst} \). Examples:

\[
\begin{align*}
> \text{(take '(1 2 3 4) 3)} \\
& (1 2 3) \\
> \text{(take '(1 2) 4)} \\
& (1 2)
\end{align*}
\]

Solution:

\[
\text{Solution:} \\
\begin{align*}
\text{(define take} & \text{(lambda (l n) )} \\
& \text{(if (null? l) ')} \\
& \text{(' } \\
& \text{(if (< n 0) '}) \\
& \text{(')} \\
& \text{((car l) (take (cdr l) (- n 1))) )} \\
\end{align*}
\]

Problem 3
Implement the procedure \( \text{(functor proc obj)} \) that applies the procedure \( \text{proc} \) to \( \text{obj} \). If \( \text{obj} \) is a list the procedure \text{functor} has to apply \text{proc} to each element of the list denoted by \( \text{obj} \) using the Scheme procedure \text{map}. Examples:

\[
\begin{align*}
> \text{(functor (lambda (x) (+ x 2)) 2)} \\
& 4
\end{align*}
\]
> (functor (lambda (x) (* x 2)) '(2 3 4))
(4 6 8)
Solution:
(define Functor
(lambda (f obj)
  (if (list? obj)
      (map f obj)
      (f obj))
)
)

Problem 4
Define the procedure \(\text{zip \, lst}_1 \, \text{lst}_2\) that takes two lists and returns a list of pairs of corresponding elements. For example: However, the two lists are not required to have the same length. That is, if the length of one list is shorter than the other one, then the length of the zipped list (i.e., the list of pairs) is the same as the length of the shorter list. To solve this problem, you have to use the Scheme procedure \text{map}.
Examples:

> (zip '(1 2 3 4 5 6) '(a b c d e f))
((1 a) (2 b) (3 c) (4 d) (5 e) (6 f))
> (zip '(1 2 3 4 5 6) '(a b c d))
((1 a) (2 b) (3 c) (4 d))
> (zip '(1 2 3 4) '(a b c d e f))
((1 a) (2 b) (3 c) (4 d))
> (zip '(1 2 3 4) '())
()
> (zip '() '(a b c d e f))
()
Solution: (give also 10 points for simple \text{‘map’})
(define zip
  (lambda (lst1 lst2)
    (let ((c1 (count lst1)) (c2 (count lst2)))
      (if (< c1 c2)
        (map list lst1 (take lst2 c1))
        (map list (take lst1 c2) lst2)
      )
    )
  )
)
Problem 5
Consider the following definition of a procedure \( h \):
\[
\text{(define h}
\begin{align*}
&\text{(lambda (op e lst)} \\
&\quad \text{(if (null? lst)} \\
&\quad\quad e \\
&\quad\quad (op (car lst) (h op e (cdr lst)))) \\
&\quad ) \\
&\text{)}
\end{align*}
\)

The procedure \( h \) works by taking a list, replacing '() by \( e \) and combining the elements of the list using the operator \( op \) from right to left. The procedure \( h \) is called fold-right. A specific property of fold-right is that it requires stack space proportional to the size of the list. An alternative implementation that requires constant space is called fold-left, which combines the elements of the list from left to right.

Define the procedure \((\text{fold-left } lst \ OP \ e)\) that, when applied to a list \( lst \), an operator \( op \), and a neutral element \( e \) combines the first element of \( lst \) with \( e \) and combines the result with remaining elements in \( lst \) to the right.

**Solution:**
\[
\text{(define fold-left}
\begin{align*}
&\text{(lambda (lst op e)} \\
&\quad \text{(if (null? lst)} \\
&\quad\quad e \\
&\quad\quad (fold-left (cdr lst) op (op e (car lst)))) \\
&\quad ) \\
&\text{)}
\end{align*}
\)

Problem 6
The union of two sets \( A \cup B \) is defined as \{\( x \mid x \in A \lor x \in B \}\}. Define the procedure \((\text{union } lst_1 \ OP \ lst_2)\) that builds the union of \( lst_1 \) and \( lst_2 \). We assume that both \( lst_1 \) and \( lst_2 \) denote proper sets, that is, they do not contain duplicates. Use the procedure \( \text{member} \) and define the procedure \( \text{union} \) in a way, such that the order of the argument lists is preserved. Examples:

\[
\begin{align*}
&\text{> (union '(a b c) '(c d e))} \\
&\quad (a b c d e) \\
&\text{> (union '(c d e) '(a b c))} \\
&\quad (c d e a b) \\
&\text{> (union '(f e d c b a) '(u v w x y z))} \\
&\quad (f e d c b a u v w x y z)
\end{align*}
\]
Solution:

(define union
  (lambda (lst1 lst2)
    (if (null? lst2)
        lst1
        (if (member (car lst2) lst1)
            (union (remove lst1 (car lst2)) (cdr lst2))
            (union (append lst1 (list (car lst2))) (cdr lst2)))))

Problem 7

The difference of two sets $A \setminus B$ is defined as $\{x \mid x \in A \land x \notin B\}$. Define the procedure $(\text{difference } lst_1 \; lst_2)$ that builds the difference of $lst_1$ and $lst_2$. We assume that both $lst_1$ and $lst_2$ denote proper sets, that is, they do not contain duplicates. Use the procedure member and define the procedure difference in a way, such that the order of the argument lists is preserved. Examples:

> (difference '(a b c d e) '(a b c))
(d e)
> (difference '(a b c d e) '(c d e))
(a b)
> (difference '(c d e a b) '(a b c))
(d e)
> (difference '(c d e a b) '(c d e))
(a b)
> (difference '(f e d c b a u v w x y z) '(z y x w v u))
(f e d c b a)

Solution:

(define difference
  (lambda (lst1 lst2)
    (if (null? lst2)
        lst1
        (if (member (car lst2) lst1)
            (difference (remove lst1 (car lst2)) (cdr lst2))
            (difference lst1 (cdr lst2)))))

(define remove
  (lambda (lst elem)
    (if (null? lst)
        '()
        (if (equal? (car lst) elem)
            (cdr lst)
            (cons (car lst) (remove (cdr lst) elem)))))
Problem 8
Given a list of numbers (i.e., \(lon\)) that starts with 2, the procedure \((\text{sieve } lon)\) returns a list of prime numbers that are contained in \(lon\).

Sieve of Eratosthenes:

We start with the integers beginning with 2, which is the first prime. To determine the rest of the primes, we start by filtering the multiples of 2 from the rest of the integers. This leaves a list beginning with 3, which is the next prime. Now we filter the multiples of 3 from the rest of that list. This leaves a list beginning with 5, which is the next prime, and so on.

In other words, we construct the primes by a sieving process, described as follows: To sieve a list of numbers \(lon\), form a list of numbers whose first element is the first element of \(lon\) and the rest of which is obtained by filtering all multiples of the first element of \(lon\) out of the rest of \(lon\) and sieving the result.

Note, given two numbers \(n\) and \(m\), then \((\text{modulo } n \ m)\) returns 0 if \(n\) is a multiple of \(m\).

Example:

\[
> (\text{sieve } '(2 3 4 5 6 7 8 9 10))
\]

\((2 3 5 7)\)

Implement the procedure \text{sieve}. Note, you should actually implement two procedures to solve the problem.

Solution:

\[
\begin{align*}
(\text{define sieve} & \quad \text{(lambda } (lon)) \\
& \quad \text{(if } (\text{null? } lon) \\
& \quad \quad ')' \\
& \quad \quad \text{(cons } (\text{car } lon) \quad (\text{sieve } (\text{filter } (\text{car } lon) \quad (\text{cdr } lon)))))) \\
& \quad ) \\
\end{align*}
\]

\[
(\text{define filter} & \quad \text{(lambda } (n \ lon)) \\
& \quad \text{(if } (\text{null? } lon) \\
& \quad \quad ')' \\
& \quad \quad \text{(if } (= 0 \ (\text{modulo } (\text{car } lon) \ n)) \\
& \quad \quad \quad (\text{filter } n \ (\text{cdr } lon)) \\
& \quad \quad \quad \text{(cons } (\text{car } lon) \quad (\text{filter } n \ (\text{cdr } lon)))))) \\
& \quad ) \\
\end{align*}
\]
Total: $10 + 10 + 10 + 10 + 15 + 15 + 25 + 40 = 135$

Submission deadline: Thursday, January 25, 2007, 2:10 p.m.

Submission procedure: on paper in class.