<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(from 20)</td>
<td>2</td>
<td>(from 25)</td>
</tr>
<tr>
<td>3</td>
<td>(from 25)</td>
<td>4</td>
<td>(from 10)</td>
</tr>
<tr>
<td>5</td>
<td>(from 20)</td>
<td>total</td>
<td>(from 100)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>total</td>
<td>(from 100)</td>
</tr>
</tbody>
</table>
Recall reverse and add-at-end:

\[\text{reverse : list-of-numbers} \rightarrow \text{list-of-numbers}\]

\[
\begin{align*}
\text{(define (reverse l)} & \text{)} \\
& \text{(cond)} \\
& \quad \text{[(empty? l) empty]} \\
& \quad \text{[else (add-at-end (first l) (reverse (rest l)))]}) \\
\end{align*}
\]

\[\text{add-at-end : number list-of-numbers} \rightarrow \text{list-of-numbers}\]

\[
\begin{align*}
\text{(define (add-at-end ele l)} & \text{)} \\
& \text{(cond)} \\
& \quad \text{[(empty? l) (list ele)]} \\
& \quad \text{[else (cons (first l) (add-at-end ele (rest l)))]}) \\
\end{align*}
\]

Rewrite these functions, using fold:

\[\text{fold : list-of-X Y (X Y} \rightarrow \text{Y) } \rightarrow \text{Y}\]

\[
\begin{align*}
\text{(define (fold l base combine)} & \text{)} \\
& \text{(cond)} \\
& \quad \text{[(empty? l) base]} \\
& \quad \text{[else (combine (first l) (fold (rest l) base combine))]} \\
\end{align*}
\]

For each use of fold, identify what X and Y from fold’s type are.

Solution

\[
\begin{align*}
\text{(define (reverse l) (fold l empty add-at-end))} \\
\text{(define (add-at-end ele l) (fold l (list ele) cons))} \\
\end{align*}
\]

In both cases, X is number and Y is list-of-numbers.
Here is a data definition for a set of numbers. Unlike a list of numbers, a set of numbers should not contain any duplicated elements.

;; a set-of-numbers is either
;; - empty
;; - (cons number[n] set-of-numbers[l])
;; INVARIANT: the number n is not in the list-of-numbers l

Not all sets have unique representations. For example, the set of numbers \{1, 3\} can be represented as either

\[(\text{cons } 1 (\text{cons } 3 \text{ empty}))\]

or

\[(\text{cons } 3 (\text{cons } 1 \text{ empty}))\]

These should be thought of as equivalent sets.

Develop three functions:

;; start : number \to set-of-numbers
;; to build a new set of numbers that contains only n
(define (start n) ...)

;; extend : number set-of-numbers \to set-of-numbers
;; to build a bigger set of numbers, extending son.
(define (extend n son) ...)

;; test : number set-of-numbers \to boolean
;; to determine if n is in son.
(define (test n son) ...)

Solution

;; start : number \to set-of-numbers
;; to build a new set of numbers that contains only n
(define (start n) (list n))

;; extend : number set-of-numbers \to set-of-numbers
;; to build a bigger set of numbers, extending son.
(define (extend n son)
  (cond
   [(test n son) son]
   [else (cons n son)]))

;; test : number set-of-numbers \to boolean
;; to determine if n is in son.
(define (test n son)
  (cond
   [(empty? son) false]
   [else (or (= n (first son))
     (test n (rest son)))]))
Here is another data definition for a set of numbers:

;; a set of numbers is a function:
;; number → boolean

The intention is that applying the set to a number determines if the number is in the set. For example, this function:

(l lambda (x) (false))

represents the set with no numbers and this function:

(l lambda (x) (or (= x 2) (= x 1)))

represents the set that contains only the numbers 1 and 2.

Develop the same three functions from the previous page, but using the new data definition:

;; start : number → set-of-numbers
;; to build a new set of numbers that contains only n
(define (start n) ...)

;; extend : number set-of-numbers → set-of-numbers
;; to build a bigger set of numbers, extending son.
(define (extend n son) ...)

;; test : number set-of-numbers → boolean
;; to determine if n is in son.
(define (test n son) ...)

Solution

;; start : number → set-of-numbers
;; to build a new set of numbers that contains only n
(define (start n) (lambda (x) (= x n)))

;; extend : number set-of-numbers → set-of-numbers
;; to build a bigger set of numbers, extending son.
(define (extend n son)
  (lambda (y)
    (or (= n y)
        (son y))))

;; test : number set-of-numbers → boolean
;; to determine if n is in son.
(define (test n son)
  (son n))
;; merge-sort : list-of-numbers → list-of-numbers
(define (merge-sort l)
  (cond
   [(empty? l) empty]
   [else
    (merge (merge-sort (evens l))
           (merge-sort (odds l)))]))

;; merge : list-of-numbers list-of-numbers → list-of-numbers
(define (merge l1 l2)
  (cond
   [(empty? l1) l2]
   [(empty? l2) l1]
   [else
    (cond
     [(< (first l1) (first l2))
      (cons (first l1) (merge (rest l1) l2))]
     [else (cons (first l2) (merge l1 (rest l2)))]))])

;; evens : non-empty-list-of-numbers → list-of-numbers
;; to extract alternating elements of l, skipping the first one.
(define (evens l)
  (cond
   [(empty? (rest l)) empty]
   [else (odds (rest l))]))

;; odds : non-empty-list-of-numbers → list-of-numbers
;; to extract alternating elements of l, starting with the first one.
(define (odds l)
  (cond
   [(empty? (rest l)) l]
   [else (cons (first l) (evens (rest l)))]))

;; (some) examples
(evens (list 1 2 3 4)) = (list 2 4)
(odds (list 1 2 3 4)) = (list 1 3)

Is the function merge-sort generative or structurally recursive?

Solution
Generative

Is the function merge generative or structurally recursive?

Solution
Structural
The `merge-sort` function on the previous page does not terminate for all lists of numbers. Identify an input for which it fails to terminate. Provide a fix so that it will terminate for all lists of numbers.

**Hint:** try some (small) hand evaluations.

**Solution**

`merge-sort` doesn’t make progress for a list of numbers that only has one number in it. For example:

\[
(\text{merge-sort } (\text{list } 1))
\]

\[
= (\text{merge } (\text{merge-sort empty}) (\text{merge-sort } (\text{list } 1)))
\]

\[
= (\text{merge } (\text{merge-sort empty}) (\text{merge } (\text{merge-sort empty}) (\text{merge-sort } (\text{list } 1))))
\]

\[
= ...
\]

To fix, add a case for a singleton list to `merge-sort`.

```scheme
;; merge-sort : list-of-numbers → list-of-numbers
(define (merge-sort l)
  (cond
   [(empty? l) empty]
   [(empty? (rest l)) l]
   [else
    (merge (merge-sort (evens l))
            (merge-sort (odds l)))]))
```