Multiple Complex Inputs

12 November 2020
Consider designing these three functions that work with lists:

- **append-lists**, which takes two lists and returns a list with all of the elements from the first list followed by all of the elements from the second list.

- **parallel-sum**, which takes two lists of numbers (of the same length) and returns a list of sums.

- **merge-lists**, which takes two sorted lists of numbers and returns a sorted list with all of the numbers.
Consider designing these three functions that work with lists:

**append-lists**, which takes two lists and returns a list with all of the elements from the first list followed by all of the elements from the second list.

**parallel-sum**, which takes two lists of numbers (of the same length) and returns a list of sums.

**merge-lists**, which takes two sorted lists of numbers and returns a sorted list with all of the numbers.

```racket
;; [List-of X], [List-of X] -> [List-of X]
(check-expect (append-lists '() '()) '())
(check-expect (append-lists '(1 2 3) '(4 5 6)) '(1 2 3 4 5 6))
```
Consider designing these three functions that work with lists:

append-lists, which takes two lists and returns a list with all of the elements from the first list followed by all of the elements from the second list.

parallel-sum, which takes two lists of numbers (of the same length) and returns a list of sums.

merge-lists, which takes two sorted lists of numbers and returns a sorted list with all of the numbers.

;; [List-of X], [List-of X] -> [List-of X]
(check-expect (append-lists '() '()) '())
(check-expect (append-lists '(1 2 3) '(4 5 6)) '(1 2 3 4 5 6))
Consider designing these three functions that work with lists:

**append-lists**, which takes two lists and returns a list with all of the elements from the first list followed by all of the elements from the second list.

**parallel-sum**, which takes two lists of numbers (of the same length) and returns a list of sums.

**merge-lists**, which takes two sorted lists of numbers and returns a sorted list with all of the numbers.

```scheme
;; [List-of Number], [List-of Number] -> [List-of Number]
(check-expect (parallel-sum '() '()) '())
(check-expect (parallel-sum '(1 2 3) '(4 7 10)) '(5 9 13))
```
Consider designing these three functions that work with lists:

- **append-lists**, which takes two lists and returns a list with all of the elements from the first list followed by all of the elements from the second list.

- **parallel-sum**, which takes two lists of numbers (of the same length) and returns a list of sums.

- **merge-lists**, which takes two sorted lists of numbers and returns a sorted list with all of the numbers.

```
;; [List-of Number], [List-of Number] -> [List-of Number]
(check-expect (parallel-sum '() '()) '())
(check-expect (parallel-sum '(1 2 3) '(4 7 10))
  '(5 9 13))
```
Consider designing these three functions that work with lists:

**append-lists**, which takes two lists and returns a list with all of the elements from the first list followed by all of the elements from the second list.

**parallel-sum**, which takes two lists of numbers (of the same length) and returns a list of sums.

**merge-lists**, which takes two sorted lists of numbers and returns a sorted list with all of the numbers.

```scheme
;; [List-of Number], [List-of Number] -> [List-of Number]
(check-expect (merge-lists '() '()) '())
(check-expect (merge-lists '(1 3 5) '(0 4 6)) '(0 1 3 4 5 6))
```
Consider designing these three functions that work with lists:

- **append-lists**, which takes two lists and returns a list with all of the elements from the first list followed by all of the elements from the second list.
- **parallel-sum**, which takes two lists of numbers (of the same length) and returns a list of sums.
- **merge-lists**, which takes two sorted lists of numbers and returns a sorted list with all of the numbers.

```ml
;; [List-of Number], [List-of Number] -> [List-of Number]
```

What template do we use for a function that consumes two lists?
The shape of a function consuming multiple complex arguments will depend on what the function needs to do with those arguments.
append-lists

Takes two lists and returns a list with all of the elements from the first list followed by all of the elements from the second list.

(check-expect (append-lists '() '())
 '())

(check-expect (append-lists (list 1 2 3) (list 4 5 6))
 (list 1 2 3 4 5 6))
append-lists

Sometimes a complex argument is just “along for the ride”, so you should use the template for the other argument:

```scheme
(define (append-lists l1 l2)
  (cond [(empty? l1) ...]
        [(cons? l1)
          (... (first l1)
               (append-lists (rest l1) l2) ...]))
```
parallel-sum

Takes two lists of numbers (of the same length) and returns a list of sums.

(check-expect (parallel-sum '() '()) '())

(check-expect (parallel-sum (list 1 3 5) (list 0 4 6)) (list 1 7 11))
parallel-sum

Sometimes the arguments are exactly the same shape, so you should essentially use the one-argument template:

```
(define (parallel-sum l1 l2)
  (cond [(empty? l1) ...]
        [(cons? l1)
         (... (first l1)
              (first l2)
              (parallel-sum (rest l1)
                            (rest l2)) ...]))
```
merge-lists

Takes two sorted lists of numbers and returns a sorted list with all of the numbers.

(check-expect (merge-lists () '()) '())

(check-expect (merge-lists (list 1 3 5) (list 0 4 6)) (list 0 1 3 4 5 6))
merge-lists

Sometimes you have to consider all possible combinations, so you use a template that considers all combinations.
merge-lists

Sometimes you have to consider all possible combinations, so you use a template that considers all combinations.

Remember our definition for \textit{ListOfNumbers}:

\begin{quote}
;;;; A ListOfNumbers is one of:
;;;; - '()
;;;; - (cons Number ListOfNumbers)
\end{quote}
merge-lists: Cross-product table

<table>
<thead>
<tr>
<th>l1</th>
<th>l2</th>
</tr>
</thead>
<tbody>
<tr>
<td>'('</td>
<td>'('</td>
</tr>
<tr>
<td>both lists are empty</td>
<td>both lists are empty</td>
</tr>
<tr>
<td>l1 is not empty, but l2 is empty</td>
<td>l1 is not empty, but l2 is empty</td>
</tr>
<tr>
<td>(cons NumberListOfNumbers)</td>
<td>(cons NumberListOfNumbers)</td>
</tr>
<tr>
<td>l2 is not empty, but l1 is empty</td>
<td>both lists are non-empty</td>
</tr>
</tbody>
</table>
**merge-lists: Cross-product table**

<table>
<thead>
<tr>
<th></th>
<th>'(())</th>
<th>(cons Number ListOfNumbers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>'()</td>
<td>'()</td>
<td>l1</td>
</tr>
<tr>
<td>'(())</td>
<td>'(())</td>
<td>l1</td>
</tr>
<tr>
<td>(cons Number ListOfNumbers)</td>
<td>l2</td>
<td>add smaller first number to recursive call</td>
</tr>
</tbody>
</table>
merge-lists

(define (merge-lists l1 l2)
  (cond [(and (empty? l1) (empty? l2))
          ...
          ]
        [(and (empty? l1) (cons? l2))
         (... (first l2)
              (merge-lists l1 (rest l2)) ...)]
        [(and (cons? l1) (empty? l2))
         (... (first l1) ...
              (merge-lists (rest l1) l2) ...)]
        [(and (cons? l1) (cons? l2))
         (... (first l1)
              (first l2)
              (merge-lists (rest l1) l2)
              (merge-lists l1 (rest l2))
              (merge-lists (rest l1) (rest l2)) ...)])

Both empty

Only l1 is empty

Only l2 is empty

Both are non-empty
**merge-lists**: Cross-product table

<table>
<thead>
<tr>
<th></th>
<th>l1</th>
<th>(cons <strong>Number</strong> <strong>List</strong>Of<strong>Numbers</strong>)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(cons <strong>Number</strong> <strong>List</strong>Of<strong>Numbers</strong>)</td>
<td>l2</td>
<td>add smaller first number to recursive call</td>
</tr>
<tr>
<td>l2</td>
<td>l2</td>
<td>l1</td>
</tr>
<tr>
<td>l1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Acknowledgments

This lecture incorporates material from:

Matthias Felleisen
Robert Bruce Findler
Matthew Flatt
Shriram Krishnamurthi
Marc Smith