Complex Arguments and Abstraction

Lecture 15
4 November 2019
Multiple complex arguments
Consider designing these three functions that work with lists:

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

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

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

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

```scheme
;; ListOfNumbers ListOfNumbers -> ListOfNumbers
(check-expect (append-lists '() '()) '())
(check-expect (append-lists (list 1 3 5) (list 0 4 6)) (list 1 3 5 0 4 6))
```
Consider designing these three functions that work with lists:

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

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

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

```scheme
;; ListOfNumbers ListOfNumbers -> ListOfNumbers
(check-expect (parallel-sum '() '()) '())
(check-expect (parallel-sum (list 1 3 5) (list 0 4 6))
  (list 1 7 11))
```
Consider designing these three functions that work with lists:

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

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

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

```scheme
;; ListOfNumbers ListOfNumbers -> ListOfNumbers
(check-expect (merge-lists '() '()) '())
(check-expect (merge-lists (list 1 3 5) (list 0 4 6))
  (list 0 1 3 4 5 6))
```
Consider designing these three functions that work with lists:

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

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

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

;; ListOfNumbers ListOfNumbers -> ListOfNumbers

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

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

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

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

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

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

<table>
<thead>
<tr>
<th></th>
<th>l1</th>
<th>l2</th>
</tr>
</thead>
<tbody>
<tr>
<td>'()'</td>
<td>(cons Number ListOfNumbers)</td>
<td>both lists are empty</td>
</tr>
<tr>
<td>'()'</td>
<td>both lists are empty</td>
<td>l1 is not empty, but l2 is empty</td>
</tr>
<tr>
<td>(cons Number ListOfNumbers)</td>
<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 NumberListOfNumbers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>l2</td>
<td>'()'</td>
<td>'()'</td>
</tr>
<tr>
<td>(cons NumberListOfNumbers)</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)) ...
        Both empty]
        [(and (empty? l1) (cons? l2)) ...
         (first l2) ...
         (merge-lists l1 (rest l2)) ...
         Only l1 is empty]
        [(and (cons? l1) (empty? l2)) ...
         (first l1) ...
         (merge-lists (rest l1) l2) ...
         Only l2 is empty]
        [(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 are non-empty)])
merge-lists: Cross-product table

<table>
<thead>
<tr>
<th></th>
<th>l1</th>
<th>l2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>'()</code></td>
<td>'(cons Number ListOfNumbers)'</td>
<td>l2</td>
<td>l1</td>
</tr>
<tr>
<td><code>(cons Number ListOfNumbers)</code></td>
<td>l2</td>
<td>add smaller first number to recursive call</td>
<td></td>
</tr>
</tbody>
</table>
[complex-arguments.rkt]
Abstraction
Functional abstraction
Many functions have the same basic shape.
The difference between two functions might just be a single value.
;; ListOfStrings -> Boolean
;; Does l contain "dog"?
(define (contains-dog? l)
  (and (not (empty? l))
    (or (string=? (first l) "dog")
      (contains-dog? (rest l))))

;; ListOfStrings -> Boolean
;; Does l contain "cat"?
(define (contains-cat? l)
  (and (not (empty? l))
    (or (string=? (first l) "cat")
      (contains-cat? (rest l)))))
;; ListOfStrings -> Boolean
;; Does l contain "dog"?
(define (contains-dog? l)
  (and (not (empty? l))
       (or (string=? (first l) "dog")
           (contains-dog? (rest l))))

;; ListOfStrings -> Boolean
;; Does l contain "cat"?
(define (contains-cat? l)
  (and (not (empty? l))
       (or (string=? (first l) "cat")
           (contains-cat? (rest l))))
We can *parametrize* the difference between the functions, namely the string we’re checking for:

```scheme
;; ListOfStrings String -> Boolean
;; Does l contain the string s?
(define (contains? l s)
  (and (not (empty? l))
       (or (string=? (first l) s)
           (contains? (rest l) s)))
```

```scheme
```

```scheme
```
This is a kind of *functional abstraction* — `contains-dog?` and `contains-cat?` can now be defined in terms of `contains?:`

```scheme
;; ListOfStrings -> Boolean
;; Does l contain "dog"?
(define (contains-dog? l)
  (contains? l "dog"))

;; ListOfStrings -> Boolean
;; Does l contain "cat"?
(define (contains-cat? l)
  (contains? l "cat"))
```
Or the difference between two functions might just be a function call.
;; ListOfNumbers Number -> ListOfNumbers
;; Select those numbers in l that are below t
(define (small l t)
  (cond [(empty? l) '()]
        [(< (first l) t)
         (cons (first l)
               (small (rest l) t))]
        [else
         (small (rest l) t)]))

;; ListOfNumbers Number -> ListOfNumbers
;; Select those numbers in l that are above t
(define (large l t)
  (cond [(empty? l) '()]
        [(> (first l) t)
         (cons (first l)
               (large (rest l) t))]
        [else
         (large (rest l) t)]))
;; ListOfNumbers Number -> ListOfNumbers
;; Select those numbers in l that are below t
(define (small l t)
  (cond [(empty? l) '()]
        [(<= (first l) t)
         (cons (first l)
               (small (rest l) t))]
        [else
         (small (rest l) t)]))

;; ListOfNumbers Number -> ListOfNumbers
;; Select those numbers in l that are above t
(define (large l t)
  (cond [(empty? l) '()]
        [(> (first l) t)
         (cons (first l)
               (large (rest l) t))]
        [else
         (large (rest l) t)]))
In this case we can parametrize the difference by specifying a function as a parameter:

;;; Function ListOfNumbers Number -> ListOfNumbers
;;; Select those numbers in l that satisfy function R with bound t
;;; R
(define (extract r l t)
  (cond 
    [(empty? l) '()] 
    [(r (first l) t) 
      (cons (first l) 
        (extract r (rest l) t))] 
    [else 
      (extract r (rest l) t)])))

(extract > '(1 2 3 4 5) 3) → '(4 5)
(extract < '(1 2 3 4 5) 3) → '(1 2)

To run this code, you’ll need to switch to the Intermediate Student language.
**Guideline:** Form an abstraction instead of copying and modifying any code.

Doing this gives you one place to make changes to the code, e.g., to fix a bug. If you make copies of code in many similar functions, you’ll have many places to update.
Abstracting data definitions
Many data definitions have the same basic shape.
Data definitions

;; A ListOfNumbers is one of:
;; - '()
;; - (cons Number ListOfNumbers)

;; A ListOfStrings is one of:
;; - '()
;; - (cons Number ListOfStrings)

Parametric data definition

;; A [List–of ITEM] is one of:
;; - '()
;; - (cons ITEM [List–of ITEM])

;; ListOfNumbers = [List–of Number]
;; ListOfStrings = [List–of String]
A [Maybe X] is one of:
- #false
- X

This parametric data definition lets us talk about:

- [Maybe String]
- [Maybe [List-of String]]
- [List-of [Maybe String]]
Acknowledgments

This lecture incorporates material from:

Matthias Felleisen
Robert Bruce Findler
Matthew Flatt
Shriram Krishnamurthi
Marc Smith