Module 2: More Functional Abstraction

Topics:

- Making abstract list functions
- Making single-use, unnamed functions
- Making functions that produce functions

Readings: HtDP 21, 22.1, 22.2, Intermezzo 4

Recall: Implementation of **map**

```
;; Produces a list of values
;; created by applying f to
;; each value in lst
(define (map f lst)
  (cond
    [(empty? lst) empty]
    [else (cons (f (first lst))
                (map f (rest lst)))]))
```


Design Recipe: What is the contract for **map**?

- What is the type of a function?
  - Its contract
- For example:

  `;; sqr: num -> num[>=0]`

  So type of **sqr** is **(num->num[>=0])**

  `;; string-length: string -> nat`

  So type of **string-length** is **(string -> nat)**

- But, **map** must work for many types of functions

Using **map**

Example	Map consumes
`(map sqr lst)`	`(num -> num[>=0]) and (listof num)`
`(map not lst)`	`(boolean->boolean) and (listof boolean)`
`(map string-length los)`	`(string -> nat) and (listof string)`

Using variables in contracts

Contract for **map** must show relationships among the parameters:

- Let X be the type of data in the consumed list
- Let Y be the type of data in the produced list

```
;; map: (X -> Y) (listof X)
;;        -> (listof Y)
```


Exercises

Complete the contracts for

- **filter**
- **foldr**

Another abstract list function: **build-list**

`build-list` can be used for tasks such as:

- Creating a list with the first 10 even numbers, starting from 0.
- Creating a list of the first 25 perfect squares, starting from 0.
- Creating

```
(list (make-posn 0 0) (make-posn 1 1)
  ... (make-posn 999 999))
```


Solving without **build-list**

- First, solve a related, simpler problem:
  create **`(list 0 1 2 ... N-1)`** for any nat **N**
- Recall: N is a natural number if
  – N = 0, or
  – N=K+1, where K is a natural number
- Recall the "count-up" template from CS115

Creating a "count-up" list

```
(define (count-up n)
 (local
  [(define (count-from-i i)
     (cond
       [(= i n) empty]
       [else (cons i
                   (count-from-i
                       (add1 i)))]))]
  (count-from-i 0)))
```

Use **count-up** to produce list of first n evens

- Approach 1:
  - Modify **count-up** directly to create an even number **2*i** instead of just **i**

A solution for **first-n-evens**

```
(define (first-n-evens n)
 (local
  [ (define (double k) (* k 2))
    (define (evens-from-i i)
     (cond
       [(= i n) empty]
       [else (cons (double i)
                   (evens-from-i
                       (add1 i)))]))]
  (evens-from-i 0)))
```


Use **count-up** to produce list of first evens

- Approach 2:
  - Use **count-up** as a helper function and map a function from **i** to **2*i**.

```
(define (first-n-evens n)
  (local
    [(define (double k) (* 2 k))]
    (map double (count-up n))))
```

Using **`build-list`** directly

```
(define (first-n-evens n)
   (local
      [(define (double k) (* 2 k))]
      (build-list n double)))
```

Contract:

```
;; build-list: nat (nat -> X)
;;              -> (listof X)
```


Other abstract list functions

- See section 21.2 (Figure 57) for summaries of Scheme's abstract list processing functions

`lambda` and unnamed, single-use functions

Consider the function **`double`** in **`first-n-evens`**:

- It only exists inside **`first-n-evens`**
- It has a name only so that it can be passed to **`build-list`**

➔New language feature: **`lambda`**

➔New language level: **`Intermediate Student with lambda`**

`lambda`

- creates a function value
- function can have any number of parameters

```
(lambda (x) (+ 1 (sqr x)))
```

⇒A function value that consumes a single parameter and produces its square plus 1

Its value is the function itself.

Using `lambda`

```
(define (first-n-evens n)
  (build-list n
     (lambda (k) (* k 2))))
```


Defining functions with `lambda`

```
(define double
   (lambda (k) (* k 2)))
```

is equivalent to

```
(define (double k)
    (* k 2))
```

Evaluating a **lambda** expression

What is the value of

```
((lambda (x) (* x 2)) 4) ?
```

- Since the first part is a function value, apply function simplification rules

```
=> (* 4 2)
=> 8
```


Evaluating another **lambda** expression

What is the value of

```
((lambda (x y)
    (cond [(> x y) x]
          [else y])) 3 2)?
=> (cond [(> 3 2) 3]
         [else 2])
=> 3
```


Our **lambda** Rule

Use **lambda** when the function is

- Single use
- Reasonably short (2-3 lines)

Complete **multiples-of** which consumes a nat **n** and a list of integers, and produces a list of only those which are multiples of `n`.

Solution

```
;; multiples-of: nat[>0] (listof int)
;;               ->(listof int)
;; produces multiples of n in lst
;; example:
;; (multiples-of 10 (list 9 -10 81 20 0))
;;           => (list -10 20 0)
(define (multiples-of n lst)
  (filter
     (lambda (k) (zero? (remainder k n)))
     lst))
```


Format of `lambda`

```
(lambda
    (list-of-parameters)
   body-of-function)
```


Functions as values

We can use function values anywhere:

- As arguments to other functions
- In lists
- As the value produced by another function

Functions as parameters: another example

```
;; transformed?: (X -> Y) X Y -> boolean
;; Produces true if the result of
;; applying f to s equals t, else false.
;; Examples:
;; (transformed? sqr 5 25) => true
;; (transformed? (lambda (s)
;;  substring s 0 1))"ABC" "a") => false
(define (transformed? f s t)
  (equal? (f s) t))
```


Function values in lists

Consider

```
(define arithmetic (list  + - * /))
(define fun-with-1-2
    (lambda (f) (f 1 2)))
```

- What is the type of **arithmetic**?
- What is the contract for **fun-with-1-2**
- What is the value of
  ((second arithmetic) 1 2) ?
- What is the value of
  (map fun-with-1-2 arithmetic) ?

Functions that make functions

Consider the function **adder-maker**:

```
(define (adder-maker addend)
    (lambda (x) (+ x addend)))
```

What does it produce?

=> a one-parameter function

Writing Examples and Tests for **adder-maker**

- **adder-maker** produces a function
- We can't test two functions for equality
- Instead, we write examples and tests for what the produced function does, e.g. show that **(adder-maker 5)** is a function that adds 5 to its argument.

More details for **adder-maker**

```
;; adder-maker: num -> (num -> num)
;; produces a function that consumes a
;; number and adds addend to it
;; Examples: ((adder-maker 5) 8) => 13
;; ((adder-maker -3) 0) => -3
(define (adder-maker addend)
    (lambda (x) (+ x addend)))
(check-expect ((adder-maker 5) 8) 13)
(check-expect ((adder-maker -3) 0) -3)
```


Goals of Module 2

- Understand use of **build-list**
- Implement functions that consume or produce functions
- Understand **lambda** and how and when it should be used
- Understand that functions are like other values in Scheme