Compositional Programming

CS251 Programming Languages
Fall 2016, Lyn Turbak

Department of Computer Science
Wellesley College

Motivating problem: ssm35

`(sum-of-squares-of-multiples-of-3-or-5 n)`

Return the sum of the squares of the all the multiples of 3 and 5 between 1 and n, inclusive.

Since `sum-of-squares-of-multiples-of-3-or-5` is a very long name, we'll abbreviate it to `ssm35`.

For example, what should `(ssm35 10)` return?

A monolithic recursive solution

This starts at n, counts down to 0, and then sums up the squares of the multiples of 3 and 5 on the way out of the recursion.

```
(define (ssm35-monolithic-count-down n)
  (if (= n 0)
      0
      (if (or (divisible-by? n 3)
              (divisible-by? n 5))
          (+ (* n n)
             (ssm35-monolithic-count-down (- n 1)))
          (ssm35-monolithic-count-down (- n 1)))))
```

```
> (ssm35-monolithic-count-down 10)
251
```


A monolithic solution that counts up

This version uses a helper function to generate the numbers from 1 up to n. But it sums the squares from highest to lowest rather than lowest to highest.

```
(define (ssm35-monolithic-count-up n)
  (define (helper num)
    (if (> num n)
        0
        (if (or (divisible-by? num 3)
                (divisible-by? num 5))
            (+ (* num num)
               (helper (+ num 1)))
            (helper (+ num 1)))))
  (helper 1))
```

```
> (ssm35-monolithic-count-up 10)
251
```

Signal-processing style of programming

This version decomposes the problem into steps that generate, map, filter, and accumulate intermediate lists. It uses higher-order list operators to manipulate the lists.

```
(define (ssm35-holo n)
  (foldr + 0
         (map (λ (x) (* x x))
              (filter (λ (num) (or (divisible-by? num 3)
                                   (divisible-by? num 5)))
                      (range 1 (+ n 1))))))
```

```
> (ssm35-holo 10)
251
```


Composition in Racket

```
(define (o f g)
  (λ (x) (f (g x))))

(define (inc x) (+ x 1))
(define (dbl y) (* y 2))

> ((o dbl inc) 5)

> ((o inc dbl) 5)
```


Composition style of programming

```
(define ssm35-compose
   (o (λ (squares)
        (foldr + 0 squares))
      (o (λ (filtered-nums)
           (map (λ (x) (* x x)) filtered-nums))
         (o (λ (nums)
              (filter (λ (num) (or (divisible-by? num 3)
                                   (divisible-by? num 5)))
                      nums))
            (o (λ (hi) (range 1 hi))
               inc)))))
```

```
> (ssm35-compose 10)
251
```


The identity function `id`

```
(define id (λ (x) x)))

> ((o id inc) 5)

> ((o dbl id) 5)
```

Composing lists of functions

```
(define (o-list funlist)
  (foldr o id funlist))

(define (dbl x) (* x 2))
(define (inc y) (+ y 1))
(define (sq z) (* z z))
```

```
> ((o-list (list dbl inc sq)) 5)
```


ssm35 with o-list

```
(define ssm35-compose-list
  (o-list (list (λ (squares)
                 (foldr + 0 squares))
                (λ (filtered-nums)
                  (map (λ (x) (* x x)) filtered-nums))
                (λ (nums)
                  (filter (λ (num) (or (divisible-by? num 3)
                                       (divisible-by? num 5)))
                          nums))
                (λ (hi) (range 1 hi))
                inc)))
```

```
> (ssm35-compose-list 10)
251
```


Recall Currying

A curried binary function takes one argument at a time.

Haskell Curry

```
(define (curry2 binop)
   (λ (x) (λ (y) (binop x y))))

(define curried-mul (curry2 *))

> ((curried-mul 5) 4)

> (my-map (curried-mul 3) (list 1 2 3))

> (my-map ((curry2 pow) 4) (list 1 2 3))

> (my-map ((curry2 (flip2 pow)) 4) (list 1 2 3))

> (define lol (list (list 2 3) (list 4) (list 5 6 7)))

> (map ((curry2 cons) 8) lol)

> (map (??? 8) lol)
  '((2 3 8) (4 8) (5 6 7 8))
```


Racket's built-in `curry` function

```
> (((curry *) 2) 5)
10

> ((curry * 2) 5)
10

> (map (curry * 3) '(7 2 5))
'(21 6 15)

> (define (triple a b c) (list a b c))

> (map (curry triple 1 2) '(7 2 5))
'((1 2 7) (1 2 2) (1 2 5))

> (map (curry triple 8 9) '(7 2 5))
'((8 9 7) (8 9 2) (8 9 5))

> (map (curry triple 8) '(7 2 5))
'(#<procedure:curried> #<procedure:curried> #<procedure:curried>)
```

Uncurrying (no built-in Racket function)

```
(define (uncurry2 curried-binop)
  (λ (x y) ((curried-binop x) y)))
```

```
> (define curried-* (curry2 *))

> (map (curried-* 3) (range 10))
'(0 3 6 9 12 15 18 21 24 27)

> (define mul (uncurry2 curried-*))

> (mul 3 4)
12
```

```
(define (uncurry3 curried-ternop)
  (λ (x y z) (((curried-ternop x) y) z)))
```


Defining functions without any λs

```
(define map-scale
  (uncurry2 (o (curry2 map) (curry2 *))))
```

```
> (map-scale 5 (range 10))
'(0 5 10 15 20 25 30 35 40 45)
```

```
(define map-cons
  (uncurry2 (o (curry2 map) (curry2 cons))))
```

```
> (map-cons 17 '((1 2 3) (4) () (5 6)))
'((17 1 2 3) (17 4) (17) (17 5 6))
```


Sometimes argument flipping is helpful

```
(define (flip2 binop)
  (λ (x y) (binop y x)))
```

```
> (filter ((curry2 divisible-by?) 5)
          (range 1 21))
'(1 5)

> (filter ((curry2 (flip2 divisible-by?)) 5)
          (range 1 21))
'(5 10 15 20)
```


Handling functions using same arg > once

```
(define (dup-arg curried-binop)
  (λ (x) ((curried-binop x) x)))
```

```
> ((dup-arg (curry2 *)) 5)
```

`and` and `or` need special handling (why?)

```
> (((curry2 and) (> 251 100)) (divisible-by? 251 3))
and: bad syntax in: and

> (((curry2 (λ (b1 b2) (and b1 b2))) (> 251 100))
   (divisible-by? 251 3))
#f

> (((curry2 (λ (b1 b2) (and b1 b2))) (< 251 100))
    (divisible-by? 251 0))
remainder: undefined for 0
```


`o-and` and `o-or`

```
(define (o-and f g)
   (λ (x) (and (f x) (g x))))

(define (o-or f g)
   (λ (x) (or (f x) (g x))))
```

```
> ((o-and (λ (n) (> n 100))
          (λ (n) (divisible-by? n 3)))
   251)
#f

> ((o-and (λ (n) (< n 100))
          (λ (n) (divisible-by? n 0)))
   251)
#f
```


Defining ssm35 without any λs

```
(define ssm35-no-lambdas
  (o-list (list (curry foldr + 0)
                ((curry2 map) (dup-arg (curry2 *)))
                ((curry2 filter)
                 (o-or ((curry2 (flip2 divisible-by?)) 3)
                       ((curry2 (flip2 divisible-by?)) 5)))
                ((curry2 range) 1)
                ((curry2 +) 1))))
```

```
> (ssm35-no-lambdas 10)
251
```