Maps, Filter, Polymorphism

CS 151: Introduction to Computer Science I

Implementing Lists

How to make a recursive definition of `(Listof Number)`:

```
(define-struct Cons-Number
  ([first :  Number]
   [rest :  (Listof-Number)]))

(define-type Listof-Number (U 'empty Cons-Number))
```

Our representation of the list '(5 15):

```
(Cons-Number 5 (Cons-Number 15 'empty))
```

Implementing Lists

Write a function that adds 1 to every element of a Listof-Number.

```
(:  add-one :  Listof-Number -> Listof-Number)
```

Implementing Lists

Write a function that adds 1 to every element of a Listof-Number.

```
(: add-one : Listof-Number -> Listof-Number)
```

```
(: add-one : Listof-Number -> Listof-Number)
;; adds 1 to every element in the list
(define (add-one list)
  (match list
    ['empty 'empty]
    [(Cons-Number head tail)
     (Cons-Number (+ 1 head) (add-one tail))]))
```

Implementing Lists

A *pair* is a data structure that holds two values.

```
(cons 42 43)
(cons "see" "saw")
(cons 'this 'that)
(cons + -)
```

A list is recursively defined in Racket: it is either the constant null, or it is a pair whose second value is a list.

Implementing Lists

Actually, our definition of `Listof-Number` pretty much matches Racket's definition of `(Listof Number)`:

```
(: add-one : Listof-Number -> Listof-Number)
(define (add-one list)
  (match list
    ['empty 'empty]
    [(Cons-Number head tail)
     (Cons-Number (+ 1 head) (add-one tail))]))
```

```
(: add-one : (Listof Number) -> (Listof Number))
(define (add-one list)
  (match list
    ['() empty]
    [(cons head tail)
     (cons (+ 1 head) (add-one tail))]))
```

Map

Very common task: given a list, do something to every element

```
(map f my-list)
```

Examples: add one to every number in a list

```
(:  add1 :  Number -> Number)
;; add 1 to the input
(define (add1 x) (+ x 1))
```

```
(map add1 (list 1 2 3 4)) ⟹ '(2 3 4 5)
```

Map

Append a ! to every `String` in a list,

```
(: append-! : String -> String)
;; append a ! character
(define (append-! str) (string-append str "!"))
```

```
    (map append-! (list "no" "good"))
⟹ '("no!" "good!")
```

Given a list of `Point`s, replace each one by its x-coordinate

```
(: my-point : Point)
(define my-point (Point -3 2))
(map Point-x (list my-point my-point)) ⟹ '(-3 -3)
```

Filter

Very common task: given a list, pick out some of the elements

```
(filter f? my-list)
```

Examples: pick out the odd numbers from a list of `Integers`

```
(filter odd? (list 1 2 3 4 5)) ⟹ '(1 3 5)
```

Pick out the `Strings` starting with ‘S’,

```
(: starts-with-S? : String -> Boolean)
(define (starts-with-S? title)
  (and
    (> (string-length title) 0)
    (char=? #\S (string-ref title 0))))
```

```
(filter starts-with-S? (list "Emma" "Sense and
Sensibility")) ⟹ '("Sense and Sensibility")
```

Foldr and Foldl

```
(foldl func initial-value list )
(foldr func initial-value list )
```

```
(: foldl : (All (X Y) (X Y -> Y) Y (Listof X) ->
Y))
;; fold a reduction function across a list
left-to-right

(: foldr : (All (X Y) (X Y -> Y) Y (Listof X) ->
Y))
;; fold a reduction function across a list
right-to-left
```

Foldr and Foldl

Fold Right successively applies a two-argument function to every element in a list from right to left starting with a initial value

```
;; Output should be 10
(foldr + 0 (list 1 2 3 4))
;; Output should be "hello"
(foldr string-append "" (list "h" "e" "l" "l" "o"))
```

Fold Left performs the same action in the opposite direction

```
;; Output should be 10
(foldl + 0 (list 1 2 3 4))
;; Output should be "olleh"
(foldl string-append "" (list "h" "e" "l" "l" "o"))
```

Functions are Data too

A `Phonebook` is a function from names to phone numbers,

```
(define-type Phonebook (String -> Integer))
```

Here's a function that looks up names in the book:

```
(:  lookup :  Phonebook String -> Integer)
(define (lookup book name)
  (book name))
```

A function that takes another function as input is called a *higher order function*

Functions are Data too

The computer science department uses two Phonebooks to keep track of people's numbers, CS Dir and FuncBook. Make a function that checks if someone's number is the same in both books.

```
(: same-number? : Phonebook Phonebook String ->
Boolean)
(define (same-number? cs-dir funcbook name)
  (=
    (cs-dir name)
    (funcbook name)))
```

Functions are Data too

Your teacher gives you two functions f and g and a number t and asks you, “which function has a larger value at t?"

```
(:  max-func :
    (Real -> Real) (Real -> Real) Real
 -> (Real -> Real))
(define (max-func f g t)
  (if (> (f t) (g t))
      f
      g))
```

Begs the question: what are the types of map and filter?

Polymorphism

We've already seen examples of functions that work for many types:

```
(length (list 5 10 100)) ⟹ 3
(length (list "Hyde" "Park")) ⟹ 2
```

```
(append (list 1 2 3) (list 4 5 6)) ⟹ '(1 2 3 4 5
6)
(append (list "some" "words") (list "other"
"words") ⟹ '("some" "words" "other" "words")
```

What are the types of the functions `length` and `append`?

```
(: length : (All (A) (Listof A) -> Integer))
```

Polymorphism

What are the types of the `first` and `rest` functions?

```
(first (list 1 2 3)) ⟹ 1
(rest (list 1 2 3)) ⟹ '(2 3)
```

```
(:  first :  (All (A) (Listof A) -> A))
(:  rest :  (All (A) (Listof A) -> (Listof A))
```

What is the type of the `list-ref` function?

```
(list-ref (list 8 9) 1) ⟹ 9
```

```
(:  list-ref :  (All (A) (Listof A) Integer -> A))
```

list length

In fact, the `length` function you implemented in HW2 was already polymorphic! All you have to do is change the type

```
(:  my-length :  (All (A) (Listof A) -> Integer))
;; computes the length of a list
(define (my-length list)
  (cond
    [(empty?  list) 0]
    [else (+
           1
           (length (rest list)))]))
```

Writing the type can be the hardest part...

list=?

How can we check if two lists are equal?

list=?

How can we check if two lists are equal?

```
(: list=? :
    (All (A) (A A -> Boolean) (Listof A) (Listof A)
    -> Boolean))
(define (list=? eq list-one list-two)
  (cond
    [(and (empty? list-one) (empty? list-two))
#t]
    [(or (empty? list-one) (empty? list-two)) #f]
    [else (and (eq (first list-one) (first
list-two)) (list=? eq (rest list-one) (rest
list-two)))]))
```

Implementing map

Question: what is the type of map?

Implementing map

Question: what is the type of map?
Polymorphic in two arguments!

```
(:  map :  (All (A B) (A -> B) (Listof A) ->
(Listof B)))
(define (map f my-list)
  (cond
    [(empty?  my-list) empty]
    [else (cons (f (first my-list)) (map f (rest
my-list)))]))
```

map is a polymorphic function and also a higher order function

Local

- Supports local definitions inside expressions
- Provide local names for intermediate values
- Provide a way to limit scope.

```
(:  distance :  Real Real Real Real -> Real)
(define (distance x1 y1 x2 y2)
  (sqrt (+ (sqr (- x2 x1)) (sqr (- y2 y1))) ))

(:  distance-local :  Real Real Real Real -> Real)
(define (distance-local x1 y1 x2 y2)
  (local
    {(define delta-x (- x2 x1))
    (define delta-y (- y2 y1))}
    (sqrt (+ (sqr delta-x) (sqr delta-y)))))
```

Local to return function as outputs

```
(define-type Int-Set (Integer -> Boolean))
```

We can view a set of integers as a function Integer -> Boolean where the function is true if the input is in the set.

```
(define-type Int-Set (Integer -> Boolean))
(:  union-int-set :  Int-Set Int-Set -> Int-Set)
;; takes the union of two sets of integers
(define (union-int-set set1 set2)
  (local
    {
    (:  union-set :  Int-Set)
    (define (union-set n)
    (or (set1 n) (set2 n)))}
    union-set))
```

Implementing Lists

```
(define-struct Cons-Number
  ([first :  Number]
   [rest :  (Listof-Number)]))

(define-type Listof-Number (U 'empty Cons-Number))
```

```
(define-struct (Cons A)
  ([first :  A]
   [rest :  (Listof A)]))

(define-type (Listof A) (U 'empty (Cons A)))
```

Binary Trees

```
(define-struct Tree
  ([value :  Number]
   [left-child :  (U 'none Tree)]
   [right-child :  (U 'none Tree)]))
```

```
(define-struct (Tree A)
  ([value :  A]
   [left-child :  (U 'none Tree)]
   [right-child :  (U 'none Tree)]))
```

Could be a tree of numbers, or a family tree, or an expression tree…

Size

Write a function that finds the number of nodes in a `Tree`.

Size

Write a function that finds the number of nodes in a `Tree`.

```
(:  size :  (All (A) (Tree A) -> Integer))
(define (size t)
  (+
   1
   (if (Tree?  (Tree-left-child t))
               (size (Tree-left-child t))
               0)
   (if (Tree?  (Tree-right-child t))
               (size (Tree-right-child t))
               0)))
```

Searching

Write a function that looks for a node of the tree with a given requirement. If there isn't a node, return `'none`

Inputs: a binary tree and a requirement function
For family trees,

```
(: my-family : (Tree String))
(define my-family a-very-complicated-expression)

(: named-barney? : String -> Boolean)
(define (named-barney? name)
  (string=? name "Barney"))

(search my-family named-barney?)
```

Could do a lot more interesting things (on Friday)

Searching

```
(:  search :  (All (A) (Tree A) (A -> Boolean) ->
    (U 'none (Tree A))))
(define (search tree req)
  (cond
    [(req (Tree-value tree)) tree]
    [(Tree?  (search (Tree-left-child tree) req))
     (search (Tree-left-child tree) req)]
    [(Tree?  (search (Tree-right-child tree) req))
     (search (Tree-right-child tree) req)]
    [else 'none]))
```

What to know

- How to implement Lists
- map, filter
- Polymorphic functions and types
- Binary trees

Lab today: 4-6 in CSIL