CSC324: Principles of Programming Languages

Lecture 5

Friday, 22 May 2020

Last time...

- We generalised our "function that operates on a list" template to a binary tree
- We saw how **higher-order functions** give us a new collection of primitives to make functions out of functions with.

Higher order list functions

Let's try to imagine some higher-order functions that make sense for lists:

- "Take a list of temperatures in *C and convert each to Kelvin."
- "Take a list of HTML elements and extract each's attributes."
- "Take a list of strings and strip each's trailing whitespace."

"Take a list of X and return the f() of each"

What does the type signature of such a function look like?

"Take a list of X and return the f() of each"

We need:

- A function that consumes **one** argument of type `a` and produces **one** `b`
  - `a and b could be the same, of course!  (eg. convert a *C to K)`
- A List of type a (eg. a list of ints)

```
;; (a -> b) -> listof a -> ??????
```

What should the return type of this function be?

map()

"Take a list of X and return the f() of each"

We need:

- A function that consumes **one** argument of type `a` and produces **one** `b`
  - `a and b could be the same, of course!  (eg. convert a *C to K)`
- A List of type a (eg. a list of ints)

```
;; (a -> b) -> listof a -> listof b
```

```
Prelude> :t map
map :: (a -> b) -> [a] -> [b]
```

Higher order list functions

Let's try to imagine some higher-order functions that make sense for lists:

- "Take a list of temperatures in *F and return the ones above freezing."
- "Take a list of light frequencies and return the ones in the UV range
- "Take a list of file names and return the ones compatible with a DOS computer"

"Take a list of X and return the ones that satisfy f()"

What does the type signature of such a function look like?

filter()

"Take a list of X and return the ones that satisfy f()"

We need:

- A function that consumes **one** argument of type `a` and produces **a boolean**
- A List of type a (eg. a list of ints)

```
;; (a -> boolean) -> listof a -> listof a
```

```
Prelude> :t filter
filter :: (a -> Bool) -> [a] -> [a]
Prelude>
```

list-process, revisited

Remember this one?

Does list-operate **need** to only consume numbers?

```
; list-operate consumes a binary operator on numbers,
; an initial number, and produces the result of applying
; the function f on the numbers.
;
; (num -> num -> num) -> num -> (listof num) -> num
(define (list-operate f init l)
  (match* (l)
    [('()) init]
    [((cons num xs)) (f num (list-operate f init xs))]))

(define (list-sum l) (list-operate + 0 l))
(define (list-prod l) (list-operate * 1 l))

(check-eq? (list-sum  '(1 2 3))  (+ 1 (+ 2 (+ 3 0))))
(check-eq? (list-prod '(1 2 3))  (* 1 (* 2 (* 3 1))))
```

list-process, revisited

Here we have a polymorphic function signature for list-process.

Is this the most general signature?

```
; list-operate consumes a binary operator on type "a"s,
; an initial a, and produces the result of applying
; the function f on the "a"s.
;
; (a -> a ->a) -> a -> (listof a) -> a
(define (list-operate f init l)
  (match* (l)
    [('()) init]
    [((cons num xs)) (f num (list-operate f init xs))]))

(define (list-sum l) (list-operate + 0 l))
(define (list-prod l) (list-operate * 1 l))
(define (list-concat-strings l) (list-operate string-append "" l))

(check-equal? (list-sum  '(1 2 3))  (+ 1 (+ 2 (+ 3 0))))
(check-equal? (list-prod '(1 2 3))  (* 1 (* 2 (* 3 1))))
(check-equal? (list-concat-strings '("hi" "csc" "324")) "hicsc324")
```

fold()

This is an example of a fold.

- if the list is empty, the result is the initial value
- else, apply f to the first element and the result of folding the rest

```
; list-process consumes a binary operator on a
; and a list of a, and produces the result of
; applying that operator on the "a"s.
;
; (a -> b -> b) -> b -> list a -> b
(define (foldr f init l)
  (match* (l)
    [('()) init]
    [((cons x xs)) (f x (foldr f init xs))]))

(define (list-sum l) ...)
(define (list-prod l) ...)
```

foldr is a right fold

the r stands for "right". Look at how the expression grows to the right as the list is traversed.

```
; foldr takes a binary operation, an initial value,
; and a list of elements, and combines the items
; calling f on them from left to right.
(define (foldr f init l)
  (match l
    ('() init)
    ((cons x xs) (f x (foldr f init xs)))))

(foldr + 0 '(1 2 3))
(+ 1 (foldr + 0 '(2 3)))
(+ 1 (+ 2 (foldr + 0 '(3))))
(+ 1 (+ 2 (+ 3 (foldr + 0 '()))))
(+ 1 (+ 2 (+ 3 0)))
(+ 1 (+ 2 3))
(+ 1 5)
6
```

foldl is a left fold

the l stands for "left".

Notice that the initial value acts as an accumulator here (see what the base case returns)

```
; foldl consists of a binary function to apply,
; an accumulator, and a list of a, and applies
; the binary operator in turn to each of a, with the
; accumulator passed to the function.
; (a -> b -> b) -> b -> list a -> b
(define (foldl f acc l)
  (match* (l)
    [('()) acc]
    [((cons x xs)) (foldl f (f x acc) xs)]))

(foldl + 0 '(1 2 3))
(foldl + (+ 1 0) '(2 3))
(foldl + 1 '(2 3))
(foldl + (+ 2 1) '(3))
(foldl + 3 '(3))
(foldl + (+ 3 3) '())
(foldl + 6 '())
6
```

foldl is a left fold

the l stands for "left".

Notice that the initial value acts as an accumulator here (see what the base case returns)

```
(foldl + (+ 3 (+ 2 (+ 1 0))) '())
```

```
; foldl consists of a bi
; an accumulator, and a list of
; the binary operator in turn to each of
; accumulator passed to the function.
; (a -> b -> b) -> b -> list a -> b
(define (foldl f acc l)
  (match* (l)
    [('()) acc]
    [((cons x xs)) (foldl f (f x acc) xs)]))

(foldl + 0 '(1 2 3))
(foldl + (+ 1 0) '(2 3))
(foldl + 1 '(2 3))
(foldl + (+ 2 1) '(3))
(foldl + 3 '(3))
(foldl + (+ 3 3) '())
(foldl + 6 '())
6
```

Ordering of folds in Racket vs Haskell

Notice that the expression that ends up getting evaluated in Racket with a left fold is different than that of a right fold.

Haskell's output is more consistent, but requires that the type signature for the two functions differ.

```
Prelude> :t foldl
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
Prelude> :t foldr
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
Prelude>
```

Last time...

We saw a function that takes two ints and produces an int

We saw a function that takes an int and produces a function, that takes an int and produces an int

Compare how we call them...

```
;; (int -> int) -> int
(define/match (slow-add m n)
  [(m 0) m]
  [(m n) (slow-add (+ m 1) (- n 1))])

;; int -> (int -> int)
(define (add-n n)
  (lambda (x) (+ x n)))

(check-eq? (slow-add 4 2) 6)
(check-eq? ((add-n 4) 2) 6)
```

Associativity of type signatures

Morally, I put it to you that the parentheses around the function arguments are ultimately unnecessary because they are "equivalent".

Why should we think about things in this way?

```
;; int -> int -> int
(define/match (slow-add m n)
  [(m 0) m]
  [(m n) (slow-add (+ m 1) (- n 1))])

;; int -> int -> int
(define (add-n n)
  (lambda (x) (+ x n)))

(check-eq? (slow-add 4 2) 6)
(check-eq? ((add-n 4) 2) 6)
```

Last time...

Function application, as it relates to the type signature, can be thought of "eating" each type in the signature for each argument that gets passed in.

```
;; int -> int -> int
(define/match (slow-add m n)
  [(m 0) m]
  [(m n) (slow-add (+ m 1) (- n 1))])

;; int -> int -> int
(define (add-n n)
  (lambda (x) (+ x n)))

(check-eq? (slow-add 4 2) 6)
(check-eq? ((add-n 4) 2) 6)
```

Currying

Definition: A function is said to be **curried** if only some of its arguments have been applied.

(Haskell automatically currys functions, so let's use Haskell for a bit here.)

Order of arguments in currying

We saw that currying in Haskell happens with the first argument first, and then the second argument next, and so on:

```
Prelude> :t filter (\ x -> x > 0) l
filter (\ x -> x > 0) l :: (Ord a, Num a) => [a]
Prelude> :t filter (\ x -> x > 0)
filter (\ x -> x > 0) :: (Ord a, Num a) => [a] -> [a]
```

Order of arguments in currying

We saw that currying in Haskell happens with the first argument first, and then the second argument next, and so on:

This means that when we choose what arguments go in what order, we **generally favour the most general arguments first**, since those are the ones most likely to be curryed.

Order of arguments in currying

We saw that currying in Haskell happens with the first argument first, and then the second argument next, and so on:

If we want to curry arguments out of order, we have to do it the manual way:

```
Prelude> l = [-2, 4, 1, 0, 4, -2, -4]
Prelude> filter_l f = filter f l
Prelude> filter_l isodd
[1]
```

Manipulating abstract syntax trees

We've seen, over the last few lectures, how to write functions that operate on various kinds of data.

Now, let's do the same with one particular datatype: the abstract syntax tree of the language we are programming in!

ASTs in Racket & The Quote Operator

We previously saw that we defined the empty list in kind of a funny way:

```
'() ;; an empty expression, prefaced by a quote
```

The time has come to talk about what that quote means.

Datum

In Racket, a quoted expression is called a **Datum**, which you saw in Lab 2 and Exercise 2. Quoting **delays evaluation** of the list (ie. prevents function application). A quoted *distributes* over all elements within the list.

```
    '(1 2 (3 4))
=> (list '1 '2 '(3 4))
=> (list '1 '2 (list '3 '4))
=> (list 1 2 (list 3 4))
```

Evaluating a datum (in the REPL only!)

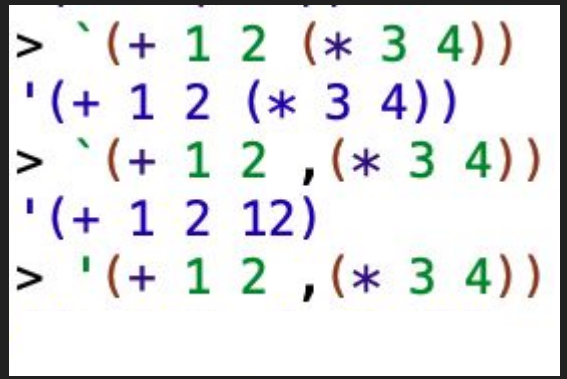
```
(eval (cons slow-add '(1 2)))

=> (eval '(slow-add 1 2))

=> 6
```

Notice that eval implicitly "knew" where to look up functions like slow-add. We will have a lot more to say about eval once we start talking about **environments**.

Quasiquoting

Quoting an expression delays evaluation of the entire expression.

Quasiquoting with the backtick operator allows us to **unquote** certain subexpressions to be evaluated as normal.



Quasiquoting

Quoting an expression delays evaluation of the entire expression.

Quasiquoting with the backtick operator allows us to **unquote** certain subexpressions to be evaluated as normal.

Quasiquoting

```
> '(+ 1 2 (* 3 4)) ; quote: all evaluation deferred
'(+ 1 2 (* 3 4))
> `(+ 1 2 (* 3 4)) ; quasi-quote: all evaluation deferred, again
'(+ 1 2 (* 3 4))
> `(+ 1 2 ,(* 3 4)) ; quasi-quote with unquote: unquoted expression evaluated
'(+ 1 2 12)
> '(+ 1 2 ,(* 3 4)) ; quote with unquote: what does this do?
```

Quasiquoting

```
> '(+ 1 2 (* 3 4)) ; quote: all evaluation deferred
'(+ 1 2 (* 3 4))
> `(+ 1 2 (* 3 4)) ; quasi-quote: all evaluation deferred, again
'(+ 1 2 (* 3 4))
> `(+ 1 2 ,(* 3 4)) ; quasi-quote with unquote: unquoted expression evaluated
'(+ 1 2 12)
> '(+ 1 2 ,(* 3 4)) ; quote with unquote: what does this do?
'(+ 1 2 ,(* 3 4))
> ; it quotes the unquote!
```