Latest: HOWTO: Get tenure
Next: Tree transformations: Desugaring Scheme
Prev: 99 ways to say '(I love you) in Racket
Rand: Grammar: The language of languages (BNF, EBNF, ABNF)

Higher-order list operations

[article index] [email me] [@mattmight] [+mattmight] [rss]

There is a pattern with students learning functional programming.

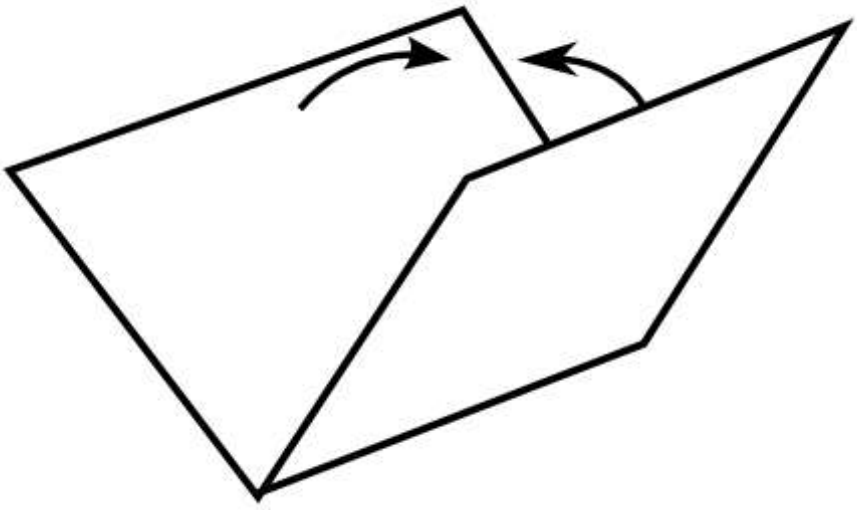
First, they try to use loops and mutation; this ends with awkward, broken programs. There is confusion and aggravation. Even hostility.

Eventually, they accept and embrace recursion.

But, then they write too much.

While recursion is better than iteration for functional programming, new functional programmers are unaware of powerful libraries to encapsulate recursion over common data structures like lists.

This post explains some of the common higher-order list operations in Racket and Haskell by re-implementing them.



To start, I abstract mapping out of adding and substracting to lists, and then I abstract folding and reducing out of mapping.

Just as with mutation, students are slow to give up on conditionals as well, but they eventually accept pattern-matching in its place.

To contrast the expressiveness in conditionals and pattern-matching, I've implemented some functions in both styles.

Where possible, I have also demonstrated each list operation using Racket's and Haskell's comprehension notations.

The post concludes with a brief example of using continuation-passing style to simplify multi-return-value list operations like zip and partition.

Read on for more.

Adding and subtracting one

Suppose you want to add one to every element of a list.

For programmers new to functional programming, it's tempting to write a recursive function for this:

```
; Racket:
(define (add1 lst)
  (if (null? lst)
      '()
      (cons (+ 1 (car lst))
            (add1 (cdr lst)))))

(add1 '(1 2 3))

-- Haskell:
add1 :: [Int] -> [Int]
add1 lst =
  if null lst
  then []
  else (head lst + 1) : (add1 (tail lst))
```

Now suppose you want to sustract one from every element of a list. Following the same strategy as before, you would create a new recursive function:

```
; Racket:
(define (sub1 lst)
  (if (null? lst)
      '()
      (cons (- (car lst) 1)
            (sub1 (cdr lst)))))

; Haskell:
sub1 :: [Int] -> [Int]
sub1 lst =
  if null lst
  then []
  else (head lst - 1) : (sub1 (tail lst))
```

While both `add1` and `sub1` are functionally correct, it is easier to use `map`:

```
 ; Racket:
 (map (λ (x) (+ x 1)) '(1 2 3)) ; yields '(2 3 4)

 ; Haskell:
 map (+1) [1,2,3] -- yields [2,3,4]
```

Abstracting into map

We can coax the definition of `map` out of `add1` by abstracting the addition operation into a functional parameter, `f`:

```
; Racket:
(define (map/test f lst)
  (if (null? lst)
      '()
      (cons (f (car lst))
            (map/test f (cdr lst)))))

; Haskell:
mapTest :: (a -> b) -> [a] -> [b]
mapTest f lst =
  if null lst
  then []
  else (f (head lst)) : (mapTest f (tail lst))
```

(I'm not using the name `map` to avoid clashing with the language-provided `map`.)

Map with matching

While the prior definition of `map` is acceptable, it is not the most natural way to express it in functional programming languages.

Functional programmers prefer structural pattern matches over explicit conditional tests:

```
; Racket:
(define (map/match f lst)
  (match lst
    ['()            '()]
    [(cons hd tl)    (cons (f hd) (map/match f tl))]))

; Haskell:
mapMatch :: (a -> b) -> [a] -> [b]
mapMatch f [] = []
mapMatch f (hd:tl) = (f hd):(mapMatch f tl)
```

Mapping with comprehensions

Racket provides special `for` forms (comprehensions) which can often replace uses of higher-order list operations like `map`.

For example, to add one to every list element, try:

```
(for/list ([x '(1 2 3 4 5)])
  (+ x 1))                      ; yields '(2 3 4 5 6)
```

Haskell also provides a comprehension notation for lists:

```
[ x + 1 | x <- [1,2,3,4,5] ] -- yields [2,3,4,5,6]
```

Filtering lists

The `filter` function offers another chance to see the difference between explicit conditional tests and structural pattern matching.

The `filter` function returns a list in which every element satisfies a predicate:

```
; Racket:
(define (filter/test p? lst)
  (cond
    [(null? lst)      '()]
    [(p? (car lst))    (cons (car lst)
                             (filter/test p? (cdr lst)))]
    [else              (filter/test p? (cdr lst))]))

-- Haskell:
filterTest :: (a -> Bool) -> [a] -> [a]
filterTest p lst =
 if null lst
   then []
 else if (p (head lst))
   then (head lst) : (filterTest p (tail lst))
 else
   (filterTest p (tail lst))
```

or, with structural pattern matching:

```
; Racket:
(define (filter/match p? lst)
  (match lst
    ['() '()]
    [(cons (? p?) tl)   (cons (car lst) (filter/match p? tl))]
    [(cons hd tl)       (filter/match p? tl)]))

; Haskell:
filterMatch :: (a -> Bool) -> [a] -> [a]
filterMatch p [] = []
filterMatch p (hd:tl) | p hd = hd:(filterMatch p tl)
                      | otherwise = filterMatch p tl
```

With these:

```
 ; Racket:
 (filter/match even? '(1 2 3 4 5 6)) ; yields '(2 4 6)

 -- Haskell:
 filterMatch even [1,2,3,4,5,6] -- yields [2,4,6]
```

Filtering with comprehensions

Racket's `for` forms accept predicates to allow fusion of mapping and filtering:

For example, to select the odd elements and add one:

```
(for/list ([x '(1 2 3 4 5)]
           #:when (odd? x))
  (+ x 1))                     ; yields '(2 4 6)
```

Haskell's comprehension notation also accepts filters:

```
 [ x + 1 | x <- [1,2,3,4,5], odd x ] -- yields [2,4,6]
```

Abstracting map

Returning to `map`, we find two more opportunities for abstraction: we can parameterize both `cons` and the empty list, `'()`:

```
; Racket:
(define (abstract-map kons nil f lst)
  (if (null? lst)
      nil
      (kons (f (car lst))
            (abstract-map kons nil f (cdr lst)))))

-- Haskell:
abstractMap :: (c -> b -> b) -> b -> (a -> c) -> [a] -> b
abstractMap kons nil f lst =
  if null lst
  then nil
  else kons (f (head lst)) (abstractMap kons nil f (tail lst))
```

By supplying the list constructor, the empty list and the identity, it recovers the original list:

```
; Racket:
(abstract-map cons '() identity '(1 2 3 4)) ; yields '(1 2 3 4)

-- Haskell:
abstractMap (:) [] id [1,2,3,4] -- yields [1,2,3,4]
```

But, by supplying addition, 0 and the identity, it sums the list:

```
; Racket:
(abstract-map + 0 identity '(1 2 3 4)) ; yields 10

-- Haskell:
abstractMap (+) 0 id [1,2,3,4] -- yields 10
```

By changing the identity to the function that squares its argument, `abstract-map` could compute the vector norm of a list:

```
; Racket:
(abstract-map + 0 (λ (x) (* x x)) '(3 4)) ; yields 25

-- Haskell:
abstractMap (+) 0 (\x -> x*x) [3,4] -- yields 25
```

From mapping into folding

Functional programming languages do not supply abstract mapping operations. Rather, they supply *folding* operations.

To derive folding from abstract mapping, consider that the `kons` parameter could apply an operation to each element if desired.

Removing the functional parameter `f` simplifies the function to `foldr`:

```
; Racket:
(define (foldr/test kons nil lst)
  (if (null? lst)
      nil
      (kons (car lst)
            (foldr/test kons nil (cdr lst)))))

-- Haskell:
foldrTest :: (a -> b -> b) -> b -> [a] -> b
foldrTest kons nil lst =
  if null lst
  then nil
  else kons (head lst) (foldrTest kons nil (tail lst))

; Racket:
(foldr/test cons '() '(1 2 3 4)) ; yields '(1 2 3 4)
(foldr/test +     0  '(1 2 3 4)) ; yields 10

-- Haskell:
foldrTest (:) [] [1,2,3,4] -- yields [1,2,3,4]
foldrTest (+) 0  [1,2,3,4] -- yields 10
```

The *r* in `foldr` comes from its application of the operation from right to left:

```
foldr (:) [] [1,2,3,4] = 1:(2:(3:(4:[])))
```

Folding is the right list operation when you need to track a running accumulation of results from previous iterations.

Folding with tail recursion

In strict functional programming languages, proper programmming practice dictates tail recursion for efficiency.

Transforming `foldr` to use tail recursion yields `foldl`:

```
-- Racket:
(define (foldl/test kons nil lst)
  (if (null? lst)
      nil
      (foldl/test kons (kons (car lst) nil) (cdr lst))))

-- Haskell:
foldlTest :: (a -> b -> b) -> b -> [a] -> b
foldlTest kons nil lst =
  if null lst
  then nil
  else foldlTest kons (kons (head lst) nil) (tail lst)
```

Now, folding applies the operation left to right:

```
foldl (:) [] [1,2,3,4] = 4:(3:(2:(1:[])))
```

which means:

```
; Racket:
(foldl/test cons '() '(1 2 3 4)) ; yields '(4 3 2 1)

-- Haskell:
foldlTest (:) [] [1,2,3,4] -- yields [4,3,2,1]
```

Folding with comprehensions in Racket

Racket provides a general `for/fold` form to express folds and combinations thereof with filters and maps.

For examples, to sum the elements of a list:

```
(for/fold ([sum 0])
  ([x '(1 2 3 4)])
  (+ x sum))         ; yields 10
```

And, `for/fold` supports multiple accumulators as well:

```
(for/fold ([sum 0] [product 1])
  ([x '(1 2 3 4)])
  (values (+ x sum) (* x product)))  ; yields 10 24
```

Reducing

Reducing is a special case of folding in which no initial accumulation element is supplied and the folding operation is an associative, commutative binary operator over a set:

```
; Racket
(define (reduce op lst)
  (match lst
    ['()             (error "no elements in list")]
    [(list a)         a]
    [(cons hd tl)    (op hd (reduce op tl))]))

-- Haskell:
reduce :: (a -> a -> a) -> [a] -> a
reduce op []  = error "no elements in list"
reduce op [x] = x
reduce op (x:tl) = op x (reduce op tl)
```

And, then:

```
; Racket:
(reduce + '(1 2 3 4)) ; yields 10

-- Haskell:
reduce (+) [1,2,3,4] -- yields 10
```

Zipping

Zipping combines two lists into a single list of pairs element-wise.

Were we to write `zip` by hand, it would move through two lists in tandem, pairing the elements:

```
; Racket:
(define (zip lst1 lst2)
  (match* [lst1 lst2]
    [{'() '()} '()]
    [{(cons hd1 tl1) (cons hd2 tl2)}
     (cons (list hd1 hd2)
           (zip tl1 tl2))]))

-- Haskell:
myZip :: [a] -> [b] -> [(a,b)]
myZip [] [] = []
myZip (hd1:tl1) (hd2:tl2) = (hd1,hd2):(myZip tl1 tl2)
```

Haskell has a `zip` function, but Racket does not, because Racket programmers can zip by supplying extra arguments to `map`:

```
(map list '(1 2 3 4) '(4 5 6 7))
  ; yields '((1 4) (2 5) (3 6) (4 7))
```

Zipping with Racket comprehensions

The `for` notation in Racket can also zip lists; for example:

```
(for/list ([x '(1 2 3 4)]
           [y '(4 5 6 7)])
  (list x y))
  ; yields '((1 4) (2 5) (3 6) (4 7))
```

Unzipping

Unzipping a list of pairs into two lists is trickier.

Since it returns two values, it can make the function awkward to write:

```
; Racket:
(define (unzip/values lst)
  (match lst
    ['() (values '() '())]
    [(cons (list a b) tl)
     (define-values (as bs) (unzip/values tl))
     (values (cons a as) (cons b bs))]))

-- Haskell:
myUnzip :: [(a,b)] -> ([a],[b])
myUnzip [] = ([],[])
myUnzip ((x,y):tl) =
 let (xs,ys) = myUnzip tl
  in (x:xs,y:ys)
```

Unzipping with continuations

The awkwardness in `unzip` comes from capturing multiple return values.

Capturing multiple return values is easier with continuation-passing style.

We're going to pass a callback--a continuation--to unzip that will accept the two unzipped lists:

```
; Racket:
(define (unzip/callback lst k)
  (match lst
    ['() (k '() '())]
    [(cons (list a b) tl)
     (unzip/callback tl (λ (as bs)
       (k (cons a as) (cons b bs))))]))

-- Haskell:
unzipk :: [(a,b)] -> ([a] -> [b] -> d) -> d
unzipk [] k = k [] []
unzipk ((x,y):tl) k =
 unzipk tl (\ xs ys -> k (x:xs) (y:ys))
```

To use this form, the programmer must supply the callback:

```
; Racket:
(unzip/callback '((1 2) (3 4) (5 6)) (λ (as bs)
  as)) ; yields '(1 3 5)

-- Haskell:
unzipk [(1,2),(3,4),(5,6)] (\ as bs -> as)  -- yields [1,3,5]
```

Partitioning

Partitioning is like filtering, except that it returns two lists: one list contains the elements matching the predicate; the other list contains those that do not.

Once again, the need to return multiple values makes the implementation feel awkward:

```
; Racket:
(define (partition/values p? lst)
  (match lst
    ['() (values '() '())]
    [(cons hd tl)
     (let-values ([{ins outs} (partition/values p? tl)])
       (if (p? hd)
           (values (cons hd ins) outs)
           (values ins (cons hd outs))))]))

-- Haskell:
partition :: (a -> Bool) -> [a] -> ([a],[a])
partition p [] = ([],[])
partition p (hd:tl) =
 let (ins,outs) = partition p tl
  in if p hd
     then (hd:ins,outs)
     else (ins,hd:outs)
```

Partitioning with continuations

Converting partitioning to continuation-passing style makes it easier to write and more convenenient to use:

```
; Racket:
(define (partition/callback p? lst k)
  (match lst
    ['() (k '() '())]
    [(cons hd tl)
     (partition/callback p? tl (λ (ins outs)
       (if (p? hd)
           (k (cons hd ins) outs)
           (k ins (cons hd outs)))))]))

-- Haskell:
partitionk :: (a -> Bool) -> [a] -> ([a] -> [a] -> d) -> d
partitionk p [] k = (k [] [])
partitionk p (hd:tl) k =
 partitionk p tl (\ ins outs ->
  if p hd
  then k (hd:ins) outs
  else k ins (hd:outs))
```

Further reading

I bought Learn You a Haskell for Great Good! as a resource for my lab to learn Haskell, and I recommend it for newcomers:

My colleagues David Van Horn and Matthias Felleisen teamed up to author Realm of Racket, a guide to learning programming in Racket with games:

The go-to resource on functional data structures and operations remains Chris Okasaki's Purely Functional Data Structures:

It's one of the classics that every functional programmer has on their shelf.

Code

The Racket code and the Haskell code are both available.

Exercises

1. Rewrite abstract mapping and folding using matching.
2. Rewrite mapping, filtering and folding using continuations.

Related pages

- Church encodings and the Y Combinator in Python
- Introducing QuickCheck: Number theory and red-black trees
- Understanding and implementing laziness

- Implementing Java as a CESK machine, in Java
- Writing an interpreter, CESK-style
- Writing CEK-style interpreters in Haskell
- Compiling up to the λ-calculus
- Parsing with derivatives (Yacc is dead: An update)
- Deleting from Okasaki's red-black trees
- By example: Continuation-passing style in JavaScript
- Self-inlining anonymous closures in C++
- 7 lines of code, 3 minutes: Implement a programming language
- Self-inlining anonymous functions in C++
- Lambda-style anonymous functions in C++
- Lambda-calculus in C++ templates
- Lazy-list-based streams in Scala
- Church encodings in Scheme
- An interpreter for Lambdo
- Okasaki red-black tree maps in Scala
- Non-termination without loops, iteration or recursion in Javascript
- Memoizing recursive functions in Javascript with the Y combinator
- Advanced programming languages

Latest: HOWTO: Get tenure
Next: Tree transformations: Desugaring Scheme
Prev: 99 ways to say '(I love you) in Racket
Rand: Grammar: The language of languages (BNF, EBNF, ABNF)