First-Class Functions in Racket

CS251 Programming Languages
Fall 2016, Lyn Turbak

Department of Computer Science
Wellesley College

First-Class Values

A value is **first-class** if it satisfies all of these properties:

- It can be named by a variable
- It can be passed as an argument to a function;
- It can be returned as the result of a function;
- It can be stored as an element in a data structure (e.g., a list);
- It can be created in any context.

Examples from Racket: numbers, boolean, strings, characters, lists, … and **functions**!

Functions can be Named

```
(define dbl (λ (x) (* 2 x)))

(define avg (λ (a b) (/ (+ a b) 2))))

(define pow
  (λ (base expt)
    (if (= expt 0)
        1
        (* base (pow base (- expt 1))))))
```

Recall syntactic sugar:

```
(define (dbl x) (* 2 x))

(define (avg a b) (/ (+ a b) 2)))

(define (pow base expt) …)
```


Functions can be Passed as Arguments

```
(define app-3-5 (λ (f) (f 3 5))

(define sub2 (λ (x y) (- x y)))


(app-3-5 sub2)

⇒ ((λ (f) (f 3 5)) sub2)

⇒ ((λ (f) (f 3 5)) (λ (x y) (- x y)))

⇒ ((λ (x y) (- x y)) 3 5)

⇒ (- 3 5)

⇒ -2
```

More Functions-as-Arguments

What are the values of the following?

```
(app-3-5 avg)

(app-3-5 pow)

(app-3-5 (λ (a b) a))

(app-3-5 +)
```


Functions can be Returned as Results from Other Functions

```
(define make-linear-function
  (λ (a b) ; a and b are numbers
    (λ (x) (+ (* a x) b))))

(define 4x+7 (make-linear-function 4 7))

(4x+7 0)

(4x+7 1)

(4x+7 2)

(make-linear-function 6 1)

((make-linear-function 6 1) 2)

((app-3-5 make-linear-function) 2)
```


More Functions-as-Returned-Values

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

((flip2 sub2) 4 7)

(app-3-5 (flip2 sub2))

((flip2 pow) 2 3))

(app-3-5 (flip2 pow))

(define g ((flip2 make-linear-function) 4 7))

(list (g 0) (g 1) (g 2))

((app-3-5 (flip2 make-linear-function)) 2)
```


Functions can be Stored in Lists

```
(define funs (list sub2 avg pow app-3-5
                   make-linear-function flip2))

((first funs) 4 7)

((fourth funs) (third funs))

((fourth funs) ((sixth funs) (third funs)))

(((fourth funs) (fifth funs)) 2)

(((fourth funs) ((sixth funs) (fifth funs))) 2)
```

Functions can be Created in Any Context

- In some languages (e.g., C) functions can be defined only at top-level. One function cannot be declared inside of another.
- Racket functions like `make-linear-function` and `flip2` depend crucially on the ability to create one function inside of another function.

Python Functions are First-Class!

```
def sub2 (x,y):
  return x - y

def app_3_5 (f):
  return f(3,5)
```

```
def make_linear_function(a, b):
  return lambda x: a*x + b

def flip2 (binop):
  return lambda x,y: binop(y,x)
```

```
In [2]: app_3_5(sub2)
Out[2]: -2

In [3]: app_3_5(flip2(sub2))
Out[3]: 2

In [4]: app_3_5(make_linear_function)(2)
Out[4]: 11

In [5]: app_3_5(flip2(make_linear_function))(2)
Out[5]: 13
```


JavaScript Functions are First-Class!

```
function sub2 (x,y){
{ return x-y; }

function app_3_5 (f)
{ return f(3,5); }
```

```
function make_linear_function(a,b) {
  return function(x) {return a*x + b;};
}

function flip2(binop) {
  return function(x,y)
    { return binop(y,x); }
}
```

```
> app_3_5(sub2)
< -2

> app_3_5(flip2(sub2))
< 2

> app_3_5(make_linear_function)(2)
< 11

> app_3_5(flip2(make_linear_function))(2)
< 13
```

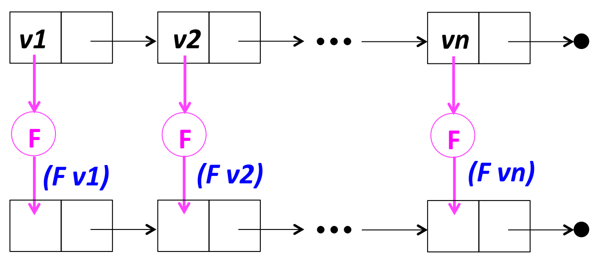

Higher-order List Functions

A function is **higher-order** if it takes another function as an input and/or returns another function as a result. E.g. `app-3-5`, `make-linear-function`, `flip2`.

We will now study **higher-order list functions** that capture the recursive list processing patterns we have seen.

Recall the List Mapping Pattern

`(mapF (list v1 v2 … vn))`




```
(define (mapF xs)
  (if (null? xs)
      null
      (cons (F (first xs))
            (mapF (rest xs)))))
```


Express Mapping via Higher-order `my-map`

```
(define (my-map f xs)
  (if (null? xs)
      null
      (cons (f (first xs))
            (my-map f (rest xs)))))
```


`my-map` Examples

```
> (my-map (λ (x) (* 2 x)) (list 7 2 4))

> (my-map first (list (list 2 3) (list 4) (list 5 6 7)))

> (my-map (make-linear-function 4 7) (list 0 1 2 3))

> (my-map app-3-5 (list sub2 + avg pow (flip pow)
                        make-linear-function))
```


Your turn

`(map-scale n nums)` returns a list that results from scaling each number in `nums` by `n`.

```
> (map-scale 3 (list 7 2 4))
'(21 6 12)

> (map-scale 6 (range 0 5))
'(0 6 12 18 24)
```

Currying

Haskell Curry

A curried binary function takes one argument at a time.

```
(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))
```


Mapping with binary functions

```
(define (my-map2 binop xs ys)
  (if (not (= (length xs) (length ys)))
      (error "my-map2 requires same-length lists")
      (if (or (null? xs) (null? ys))
          null
          (cons (binop (first xs) (first ys))
                (my-map2 binop (rest xs) (rest ys))))))
```

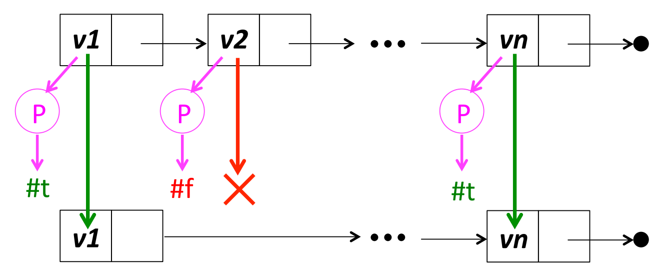
```
> (my-map2 pow (list 2 3 5) (list 6 4 2))
'(64 81 25)

> (my-map2 cons (list 2 3 5) (list 6 4 2))
'((2 . 6) (3 . 4) (5 . 2))

> (my-map2 cons (list 2 3 4 5) (list 6 4 2))
ERROR: my-map2 requires same-length lists
```


Built-in Racket `map` Function Maps over Any Number of Lists

```
> (map (λ (x) (* x 2)) (range 1 5))
'(2 4 6 8)

> (map pow (list 2 3 5) (list 6 4 2))
'(64 81 25)

> (map (λ (a b x) (+ (* a x) b))
       (list 2 3 5) (list 6 4 2) (list 0 1 2))
'(6 7 12)

> (map pow (list 2 3 4 5) (list 6 4 2))
ERROR: map: all lists must have same size;
arguments were: #<procedure:pow> '(2 3 4 5) '(6 4 2)
```


Recall the List Filtering Pattern

(filter**P** (list ***v1 v2*** … ***vn***))




```
(define (filterP xs)
  (if (null? xs)
      null
      (if (P (first xs))
          (cons (first xs) (filterP (rest xs)))
          (filterP (rest xs)))))
```

Express Filtering via Higher-order `my-filter`

```
(define (my-filter pred xs)
  (if (null? xs)
      null
      (if (pred (first xs))
          (cons (first xs)
                (my-filter pred (rest xs)))
          (my-filter pred (rest xs)))))
```

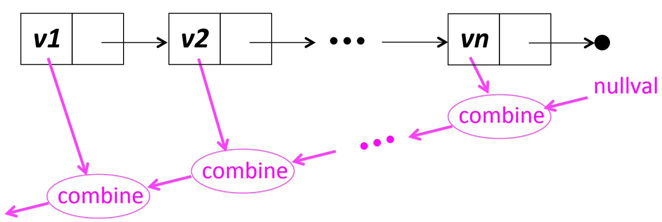
Built-in Racket `filter` function acts just like `my-filter`

`filter` Examples

```
> (filter (λ (x) (> x 0)) (list 7 -2 -4 8 5))

> (filter (λ (n) (= 0 (remainder n 2)))
          (list 7 -2 -4 8 5))

> (filter (λ (xs) (>= (len xs) 2))
          (list (list 2 3) (list 4) (list 5 6 7))

> (filter number?
          (list 17 #t 3.141 "a" (list 1 2) 3/4 5+6i))

> (filter (lambda (binop) (>= (app-3-5 binop)
                              (app-3-5 (flip2 binop))))
          (list sub2 + * avg pow (flip2 pow)))
```


Recall the Recursive List Accumulation Pattern

(recf (list *v1* *v2* … *vn*))




```
(define (rec-accum xs)
  (if (null? xs)
      nullval
      (combine (first xs)
               (rec-accum (rest xs)))))
```


Express Recursive List Accumulation via Higher-order `my-foldr`

```
(define (my-foldr combine nullval xs)
  (if (null? xs)
      nullval
      (combine (first xs)
               (my-foldr combine
                         nullval
                         (rest xs)))))
```

my-foldr Examples

```
> (my-foldr + 0 (list 7 2 4))

> (my-foldr * 1 (list 7 2 4))

> (my-foldr - 0 (list 7 2 4))

> (my-foldr min +inf.0 (list 7 2 4))

> (my-foldr max -inf.0 (list 7 2 4))

> (my-foldr cons (list 8) (list 7 2 4))

> (my-foldr append null
            (list (list 2 3) (list 4)(list 5 6 7)))
```


More my-foldr Examples

```
;; This doesn't work. Why not?
> (my-foldr and #t (list #t #t #t))

> (my-foldr (λ (a b) (and a b)) #t (list #t #t #t))

> (my-foldr (λ (a b) (and a b)) #t (list #t #f #t))

> (my-foldr (λ (a b) (or a b)) #f (list #t #f #t))

> (my-foldr (λ (a b) (or a b)) #f (list #f #f #f))
```

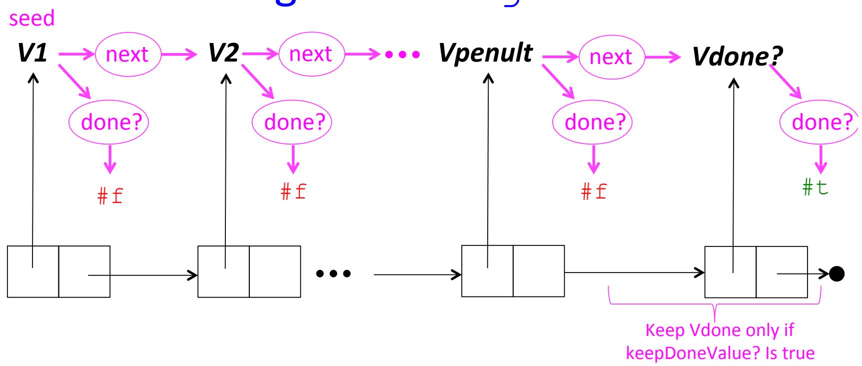

Mapping & Filtering in terms of my-foldr

```
(define (my-map f xs)
  (my-foldr ???

            ???
            xs))

(define (my-filter pred xs)
  (my-foldr ???

            ???
            xs))
```


Built-in Racket foldr Function Folds over Any Number of Lists

```
> (foldr + 0 (list 7 2 4))
13
> (foldr (lambda (a b sum) (+ (* a b) sum))
         0
         (list 2 3 4)
         (list 5 6 7))
56
> (foldr (lambda (a b sum) (+ (* a b) sum))
         0
         (list 1 2 3 4)
         (list 5 6 7))
ERROR: foldr: given list does not have the same size
as the first list: '(5 6 7)
```

Creating lists with `genlist`




```
(define (genlist next done? keepDoneValue? seed)
  (if (done? seed)
      (if keepDoneValue? (list seed) null)
      (cons seed
            (genlist next done? keepDoneValue? (next seed)))))
```


Simple `genlist` examples

```
(genlist (λ (n) (- n 1))
         (λ (n) (= n 0))
         #t
         5)
```

```
(genlist (λ (n) (- n 1))
         (λ (n) (= n 0))
         #5
         5)
```

```
(genlist (λ (n) (* n 2))
         (λ (n) (> n 100))
         #t
         1)
```

```
(genlist (λ (n) (* n 2))
         (λ (n) (> n 100))
         #f
         1)
```


Your Turn

(my-range ***lo hi***)

```
> (my-range 10 20)
'(10 11 12 13 14 15 16 17 18 19)

> (my-range 20 10)
'()
```

(halves ***num***)

```
> (halves 64)
'(64 32 16 8 4 2 1)

> (halves 42)
'(42 21 10 5 2 1)

> (halves 63)
'(63 31 15 7 3 1)
```


Digression: Iteration Example (upcoming lecture)

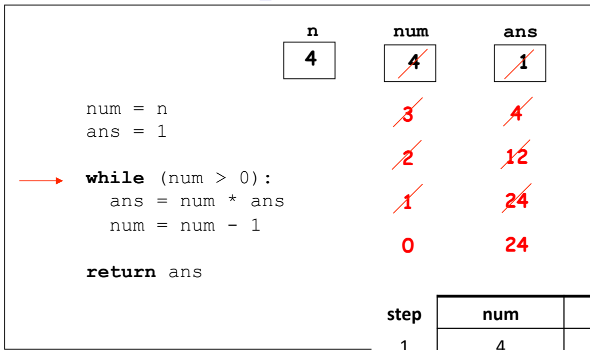
Iteration Rules:

- next num is previous num minus 1.
- next ans is previous num times previous ans.

```
def fact_while(n):

  num = n      } Declare/initialize local
  ans = 1      } state variables

  while (num > 0):
    ans = num * ans   } Calculate product and
    num = num - 1     } decrement num

  return ans   } Don't forget to return answer!
```

Digression: iteration tables

Execution frame for fact_while(4)



step	num	ans
1	4	1
2	3	4
3	2	12
4	1	24
5	0	24

Using genlist to generate iteration table

```
(define (fact-table n)
  (genlist (λ (num&ans)
             (let ((num (first num&ans))
                   (ans (second num&ans)))
               (list (- num 1) (* num ans))))
           (λ (num&ans) (<= (first num&ans) 0))
           #t
           (list n 1)))
```

```
> (fact-table 4)
'((4 1) (3 4) (2 12) (1 24) (0 24))
> (fact-table 5)
'((5 1) (4 5) (3 20) (2 60) (1 120) (0 120))
```

```
> (fact-table 10)
'((10 1)
  (9 10)
  (8 90)
  (7 720)
  (6 5040)
  (5 30240)
  (4 151200)
  (3 604800)
  (2 1814400)
  (1 3628800)
  (0 3628800))
```


Your turn: sum-list iteration table

```
(define (sum-list-table ns)
  (genlist (λ (nums&ans)



                                                   )
           (λ (nums&ans)                           )
           #t
           (list                  )))
```

```
> (sum-list-table '(7 2 5 8 4))
'(((7 2 5 8 4) 0)
  ((2 5 8 4) 7)
  ((5 8 4) 9)
  ((8 4) 14)
  ((4) 22)
  (() 26))
```


`genlist` can collect iteration table column!

```
; With table abstraction
(define (partial-sums ns)
  (map second (sum-list-table ns)))
```

```
; Without table abstraction
(define (partial-sums ns)
  (map second
       (genlist (λ (nums&ans)
                  (let ((nums (first nums&ans))
                        (ans (second nums&ans)))
                    (list (rest nums) (+ (first nums) ans))))
                (λ (nums&ans) (null? (first nums&ans)))
                #t
                (list ns 0))))
```

```
> (partial-sums '(7 2 5 8 4))
'(0 7 9 14 22 26)
```

Moral: ask yourself the question
"Can I generate this list as the column of an iteration table? "

Racket's `apply`

```
(define (avg a b)
  (/ (+ a b) 2))
```

```
> (avg 6 10)
8

> (apply avg '(6 10))
8
```

`apply` takes (1) a function and (2) a **list** of arguments
and returns the result of applying the function to the arguments.

`genlist-apply`: a kinder, gentler `genlist`

```
(define (genlist-apply next done? keepDoneValue? seed)
  (if (apply done? seed)
      (if keepDoneValue? (list seed) null)
      (cons seed
            (genlist-apply next done? keepDoneValue?
                           (apply next seed)))))
```

Example:

```
(define (partial-sums ns)
  (map second
       (genlist-apply
         (λ (nums ans)
          (list (rest nums) (+ (first nums) ans)))
         (λ (nums ans) (null? nums))
         #t
         (list ns 0))))
```


Your turn: `partial-sums-between`

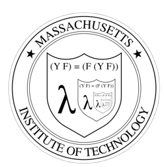
```
(define (partial-sums-between lo hi)
  (map second
       (genlist-apply
          ; Flesh out parts









          )))
```

```
> (partial-sums-between 3 7)
'(0 3 7 12 18 25)

> (partial-sums-between 1 10)
'(0 1 3 6 10 15 21 28 36 45 55)
```


Summary (and Preview!)

Data and procedures and the values they amass,
Higher-order functions to combine and mix and match,
Objects with their local state, the messages they pass,
A property, a package, a control point for a catch —
In the Lambda Order they are all first-class.
One Thing to name them all, One Thing to define them,
One Thing to place them in environments and bind them,
In the Lambda Order they are all first-class.

Abstract for the *Revised4 Report on the Algorithmic Language Scheme (R4RS)*, MIT Artificial Intelligence Lab Memo 848b, November 1991



Emblem for the Grand Recursive Order of the Knights of the Lambda Calculus