First-Class Functions in Racket

CS251 Programming Languages
Spring 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

\[(\text{define app-3-5 } (\lambda (f) (f 3 5)))\]
\[(\text{define sub2 } (\lambda (x y) (- x y)))\]

\[(\text{app-3-5 sub2})\]
\[\Rightarrow ((\lambda (f) (f 3 5)) \text{ sub2})\]
\[\Rightarrow ((\lambda (f) (f 3 5)) (\lambda (x y) (- x y)))\]
\[\Rightarrow ((\lambda (x y) (- x y)) 3 5)\]
\[\Rightarrow (- 3 5)\]
\[\Rightarrow -2\]
More Functions-as-Arguments

What are the values of the following?

\((\text{app-3-5 \ avg})\)

\((\text{app-3-5 \ pow})\)

\((\text{app-3-5 \ (\lambda \ (a \ b) \ a)})\)

\((\text{app-3-5 \ +})\)
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!

```python
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!
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

\[(\text{map} F \ (\text{list} \ v1 \ v2 \ \ldots \ vn))\]

\[
\begin{align*}
(v1) \rightarrow (v2) \rightarrow \cdots \rightarrow (vn) \\
\quad \downarrow F \quad \downarrow F \quad \downarrow F \\
\quad (F \ v1) \rightarrow (F \ v2) \rightarrow (F \ vn)
\end{align*}
\]

\[
\text{(define (mapF xs)} \\
\text{  (if (null? xs) \null \null)\n}\text{  (cons (F (first xs))} \\
\text{    (mapF (rest xs)))))\n\]
Express Mapping via Higher-order \texttt{my-map}:

\begin{center}
\begin{verbatim}
(define (my-map \texttt{f} \texttt{xs})
  (if (null? \texttt{xs})
      null
      (cons (\texttt{f} (first \texttt{xs}))
            (my-map \texttt{f} (rest \texttt{xs})))))
\end{verbatim}
\end{center}
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

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 \texttt{map} Function
Maps over Any Number of Lists

> (map (\lambda (x) (* x 2)) (range 1 5))
'(2 4 6 8)

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

> (map (\lambda (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))
\textbf{ERROR}: \texttt{map}: all lists must have same size;
arguments were: \#<procedure:pow> '(2 3 4 5) '(6 4 2)
Recall the List Filtering Pattern

\[(\text{filter} P \ (\text{list} \ v_1 \ v_2 \ \ldots \ v_n))\]

\[\text{(define (filter}\ P \ \text{xs})\]
\[\text{(if (null? } \text{xs})\]
\[\text{null}\]
\[\text{(if (P (first } \text{xs})\]
\[\text{(cons (first } \text{xs}) \ (\text{filter} P \ (\text{rest} \ \text{xs})))\]
\[\text{(filter} P \ (\text{rest} \ \text{xs})))\)]
Express Filtering via Higher-order `my-filter`

```scheme
(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

\[(\text{recf } (\text{list } v_1 \ v_2 \ ... \ v_n))\]

\[
\begin{align*}
  v_1 & \rightarrow v_2 & \rightarrow \cdots & \rightarrow v_n \\
  \text{combine} & & & \text{combine} & & & \text{combine} & & & \text{nullval}
\end{align*}
\]

\[
(\text{define } (\text{rec-accum } xs) \\
  (\text{if } (\text{null? } xs) \\
   \text{nullval} \\
   (\text{combine } (\text{first } xs) \\
    (\text{rec-accum } (\text{rest } xs))))))
\]
Express Recursive List Accumulation via Higher-order \textit{my-foldr}

\begin{verbatim}
(define (my-foldr combine nullval xs)
  (if (null? xs)
      nullval
      (combine (first xs)
                (my-foldr combine nullval (rest xs))))
\end{verbatim}
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`

```scheme
(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: '[(list 5 6 7)]
```
Compositional Programming

(sum-squares-of-multiples-of-3-or-5-up-to hi)
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