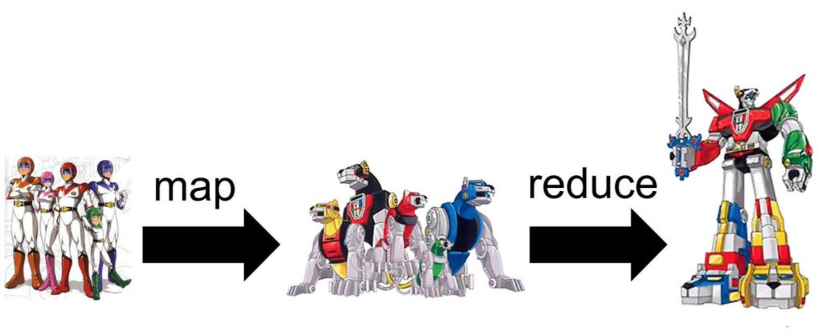
CS 314 Lecture 10

Functional programming

February 26, 2019

Map and reduce



Higher-order Functions: map

```
(define map
  (lambda (f l)
    (if (null? l)
     '()
     (cons (f (car l)) (map f (cdr l))))))
```

- map takes two arguments: a function and a list
- map builds a new list by applying the function to every element of the (old) list

Higher-order Functions: map

Example:

- (map abs '(-1 2 -3 4)) $\Rightarrow$ (1 2 3 4)
- (map (lambda (x) (+ 1 x)) '(-1 2 -3)) $\Rightarrow$ (0 3 -2)

Actually, the built-in map can take more than two arguments:

- (map + '(1 2 3) '(4 5 6)) $\Rightarrow$ (5 7 9)

More on Higher Order Functions

reduce: a higher order function that takes a binary, associative operation and uses it to “roll-up” a list

```
(define reduce
  (lambda (op l id)
    (if (null? l)
        id
        (op (car l) (reduce op (cdr l) id)) )))
```

More on Higher Order Functions

Example:

- (reduce + '(10 20 30) 0)
- $\Rightarrow$ (+ 10 (reduce + '(20 30) 0))
- $\Rightarrow$ (+ 10 (+ 20 (reduce + '(30) 0)))
- $\Rightarrow$ (+ 10 (+ 20 (+ 30 (reduce + '() 0))))
- $\Rightarrow$ (+ 10 (+ 20 (+ 30 0)))
- $\Rightarrow$ 60

More on Higher Order Functions

Now we can compose higher order functions to form compact powerful functions.

Examples:

```
(define sum
  (lambda (f l)
    (reduce + (map f l) 0) ))
```

- (sum (lambda (x) (* 2 x)) '(1 2 3))
- (reduce (lambda (x y) (+ 1 y)) '(a b c) 0)

Lexical Scoping and let, let*, and letrec

All are variable binding operations:

(here LET = one of let, let*,letrec):

```
(LET ((v1 e1)
      (v2 e2)
      ...
      (vn en))
  e)
```

Lexical Scoping and let, let*, and letrec

- let: binds variables to values (no specific order), and evaluates body e using the bindings; new bindings are not effective during evaluation of any e i .
- let*: binds variables to values in textual order of write-up (left to right, or here: top down); new binding is effective for next e i (nested scopes).
- letrec: bindings of variables to values in no specific order; independent evaluations of all e i to values have to be possible; new bindings effective for all e i ; mainly used for recursive function definitions.

Lexical Scoping and let, let*, and letrec

```
(let ((x 10)
      (y 20))
  (+ x y))
```

Lexical Scoping and let, let*, and letrec

```
(let* ((x 10)
       (y (* 2 x)))
  (+ x 1))
```

Lexical Scoping and let, let*, and letrec

```
(letrec ((f ...g...)
         (g ...f...))
  (+ (f 10) (g 20)))
```

Lexical Scoping and let, let*, and letrec

```
(define even? (lambda (x)
  ...))

(define odd? (lambda (x)
  ...))
```

Lexical Scoping and let, let*, and letrec

```
(letrec (even? (...)
          odd? (...))
  (even? 10))
```

1. numElements: '(a b (c d)) $\rightarrow$ 4, '(a b c) $\rightarrow$ 3
2. flatten: '(a (b c) d) $\rightarrow$ '(a b c d)
3. rev: '(a (b c) d) $\rightarrow$ '(d (c b) a)
4. double: '(a (b c) d) $\rightarrow$ '(a a (b b c c) d d)
5. delete: 'c '(a (b c) c d) $\rightarrow$ '(a (b) d)
6. minSquareVal: '(-5 3 -7 10 -11 8 7) $\rightarrow$ 9

Does this make sense?

f $\equiv$ ... f ...

In lambda calculus, such an equation does not define a term. How to find a λ-term that does “satisfy” the recursive definition?

Does this make sense?

```
add = \mn.if (isZero? n) then m
                 else (add (succ m) (pred n))
```

This is not a valid definition of a λ-term. What about this one?

```
add = \f.(\mn.if (isZero? n) then m
                      else (f (succ m) (pred n)))
```

Claim: The fixed point of the above function is what we are looking for.

Function fixed points

The fixed points of a function g is the set of values

$\text{fix}_g = \{x | x = g(x)\}$

Examples:

function g	fix_g
λx.6	6
λx.(6 - x)	3
λx.((x*x) + (x-4))	-2, 2
λx.x	entire domain of f
λx.(x+1)	{}

The Y-combinator

Is there a λ-term Y that "computes" a fixed point of a function $F = \lambda f.(\ldots f \ldots)$, i.e., $(YF) = (F(YF))$?

YES. Y is called the fixed point combinator.

$Y \equiv (\lambda f.((\lambda x.f(x\ x))\ (\lambda x.f(x\ x))))$

- (YF)
- $= ((\lambda f.((\lambda x.f(x\ x))\ (\lambda x.f(x\ x))))\ F)$
- $= ((\lambda x.F(x\ x))\ (\lambda x.F(x\ x)))$
- $= (F((\lambda x.F(x\ x))\ (\lambda x.F(x\ x))))$
- $= (F(YF))$

The Y-combinator

Example: $F \equiv \lambda f.(\lambda mn.$
if (isZero? n) then m else (f (succ m) (pred n)))

- $((YF)\ 3\ 2) =$
- $(((\lambda f.((\lambda x.f(x\ x))\ (\lambda x.f(x\ x))))\ F)\ 3\ 2) =$
- $((F((\lambda x.F(x\ x))\ (\lambda x.F(x\ x))))\ 3\ 2) =$
- $((\lambda mn.$if (isZero? n) then m else
  $(((\lambda x.F(x\ x))\ (\lambda x.F(x\ x)))$ (succ m) (pred n))) 3 2) $=$
- if (isZero? 2) then 3 else
  $(((\lambda x.F(x\ x))\ (\lambda x.F(x\ x)))$ (succ 3) (pred 2)) $=$
- $(((\lambda x.F(x\ x))\ (\lambda x.F(x\ x)))\ 4\ 1) =$
- $((F((\lambda x.F(x\ x))\ (\lambda x.F(x\ x))))\ 4\ 1) =$

The Y-combinator

Example: $F \equiv \lambda f.(\lambda mn.$
if (isZero? n) then m else (f (succ m) (pred n)))

- $((F((\lambda x.F(x\ x))\ (\lambda x.F(x\ x))))\ 4\ 1) =$
- if (isZero? 1) then 4 else
  $(((\lambda x.F(x\ x))\ (\lambda x.F(x\ x)))$ (succ 4) (pred 1)) =
- $(((\lambda x.F(x\ x))\ (\lambda x.F(x\ x)))\ 5\ 0) =$
- $((F((\lambda x.F(x\ x))\ (\lambda x.F(x\ x))))\ 5\ 0) =$
- if (isZero? 0) then 5 else
  $(((\lambda x.F(x\ x))\ (\lambda x.F(x\ x)))$ (succ 5) (pred 0)) =
- 5

The Y-combinator example (cont.)

Note: Informally, the Y-combinator allows us to get as many copies of the recursive procedure body as we need. The computation "unrolls" recursive procedure calls one at a time.

This notion of recursion is purely syntactic.

Recursion

```
(define fact (lambda (n)
  (if (zero? n)
      1
      (* n (fact (- n 1))))))
```

Recursion

```
(fact 4)
= (* 4 (fact 3))
= (* 4 (* 3 (fact 2)))
= (* 4 (* 3 (* 2 (fact 1))))
= (* 4 (* 3 (* 2 (* 1 (fact 0)))))
= (* 4 (* 3 (* 2 (* 1 1))))
= (* 4 (* 3 (* 2 1)))
= (* 4 (* 3 2))
= (* 4 6)
= 24
```

Recursion

```
(define fact-tail (lambda (n)
  (fact-tail-acc n 1)))

(define fact-tail-acc (lambda (n a)
  (if (zero? n)
      a
      (fact-tail-acc (- n 1) (* n a)))))
```

Recursion

```
(fact-tail 4)
= (fact-tail-acc 4 1)
= (fact-tail-acc 3 4)
= (fact-tail-acc 2 12)
= (fact-tail-acc 1 24)
= (fact-tail-acc 0 24)
= 24
```

Tail recursion

When the recursive call is the last value returned by a function, we say it's tail recursive.

- Reuse stack frame
- Control data we need is constant

Closures

A function as a value where free variables capture their local environment is known as a closure:

```
(define plusn
  (lambda (n)
    (lambda (y)
      (+ n y))))
```

Thunks

We can use lambdas to delay evaluation:

```
(define foo1 (+ 2 3))
(define foo2 (lambda () (+ 2 3)))
```

This is known as a *thunk*.

Thunks

Recall the example lambda expression with no normal form:

```
(let ((foo
        ((lambda (x) (x x)) (lambda (x) (x x)))))
  (+ 1 2))
```

Thunks

```
(let ((foo
         (lambda ()
           ((lambda (x) (x x)) (lambda (x) (x x))))))
  (+ 1 2))
```

Streams

Let's write a function that generates all positive integers.

```
(define allInts
  (lambda ()
    (list 0 1 2 ...)))
```

Streams

Let's write a function that generates all positive integers.

```
(define allIntsFrom
  (lambda (n)
    (cons n (allIntsFrom (+ n 1)))))

(define allInts
  (allIntsFrom 0))
```

Streams

Let's write a function that generates all positive integers.

```
(define allIntsFrom
  (lambda (n)
    (cons n (lambda () (allIntsFrom (+ n 1))))))

(define allInts
  (allIntsFrom 0))
```