PPL – Recitation Six

Abdall

March 5, 2019

Goals for this Recitation

Goals

- Go over the lambda calc booleans one more time
- Let's get scheming

Lambda Calc Booleans

Let's Recall

- $\lambda x.\lambda y.x \equiv$ true (pick first)
- $\lambda x.\lambda y.y \equiv$ false (pick second)
- This has no meaning – we just picked arbitrarily but have to derive some meaning from them

And

p	q	$p \wedge q$
T	T	T
T	F	F
F	T	F
F	F	F

- What can we notice here?

And

p	q	$p \wedge q$
T	T	T
T	F	F
F	T	F
F	F	F

- Some values of p are deterministic.
- In this way our lambda expressions become easy: $\lambda p.\lambda q.(p\ q\ F)$

This is sometimes hard...

p	q	$p\ XOR\ q$
T	T	F
T	F	T
F	T	T
F	F	F

Let's get shcemin'

Fundamentals

- Not really a big deal but we have $(+\ 1\ 2)$
- We will work a lot with lists

List Basics

- Let us say we have: (define L '(1 2 3 4))
- Then (car L) $\equiv$ '(1 2 3 4)
- Then (cdr L) $\equiv$ '(1 2 3 4)
- What is super important about this?

List Basics

- Let us say we have: (define L '(1 2 3 4))
- Then (car L) $\equiv$ '(1 2 3 4)
- Then (cdr L) $\equiv$ '(1 2 3 4)
- What is super important about this?
- We also have (cons '15 L) $\rightarrow$ '(15 1 2 3 4)

Sanity check

- (car '(3 4))
- (cdr '(3))
- (cons (car '(3 4)) (cdr '(2 3 4)))
- (cons '15 (cons '12 '(1 2 3)))
- (cons '15 '(cons '12 '(1 2 3)))
- ((car (cons + '())) 2 5)

Functions

- Let us talk function definitions:
  (define *name* (lambda (*params*) *expression*))
- A quick example:

```
(define length_func
    (lambda (l)
     (cond
         ((null? l) 0)
         (else (+ 1 (length (cdr l)))))))

```

Let's take a step back

- Control Sequence:

```
(define absval
  (lambda (x)
     (if (>= x 0) x (* -1 x))))

```

- Where have we seen this?!

else if else if else if ...

- If we have to have multiple conditions:

```
(cond
  (<cond 1 > < result 1 >)
  (< cond 2 > < result 2>)
  . . .
  (< cond N > < result N >)
  ( e l s e <else-result >)
)
```

Example Central

- The factorial function:

Example Central

- The factorial function:

```
(define fact
    (lambda (x)
     (cond
        ((zero? x) 1)
        ((eq? x 1) 1)
        (else (* x (fact (- x 1)))))))

```

A very common structure

- Let us say we want to do some sort of comprehension on a list
- Typically we want to process the first element, do something, then recursively do this to the rest of the list:

```
(define delete
    (lambda (x l)
      (cond
        ((null? l) '())
        ((eq? x (car l)) (delete x (cdr l)))
        (else (cons (car l) (delete x (cdr l))))))
  )
```

Let's practice

- We want to write a function that doubles the given parameter:

```
(define double
  (lambda (a alist)
    ...

```

WE HAVE A PROBLEM

- We can't always expect "flat" lists
- What if we have a list that is like: '(1 2 3 '(4 5 1 3) 12 9)

WE HAVE A PROBLEM

- We can't always expect "flat" lists
- What if we have a list that is like: '(1 2 3 '(4 5 1 3) 12 9)
- There is a slightly different way to handle this...
- Do the double example but handle nested lists

To higher orders we go!

Map

- To me this one makes more sense lol
- It takes a function and a list
- Takes each element, applies the function, adds back to a list
- For a thought exercise, can you write this one?

Reduce

- This one is a bit more tricky
- Need a binary (associative) operation, a list, and an ID
- We roll our function down
- For a thought exercise, can you write this one?