Fall CS 145 Final Review Guide.

- Prof: Gordon Cormack
- ISA: Ashish Mahto
- Author: Teresa Kang
- Edit: David Duan - j32duan@edu.uwaterloo.ca

==========

Topic. 7 Lambda Calculus

Subtopic. 7.1 Logic Operators

True = λx.λy.x
False = λx.λy.y

Note.
We think of True/False as making a choice between two things.
Choosing the first one between two things represents True; otherwise False.

```
<rkt>
  (define True (λ (yes) (λ (no) yes)))
  (define False (λ (yes) (λ (no) no)))
  (define (True? b) ((b 'yes) 'no))
  ;; if b is True, b returns the first value it consumes, which is yes; if b is False, it returns no.
</rkt>
```

Example. (IF <expr> <then-do> <else-do>)

```
<rkt>
  (define (IF test then-do else-do)
    ((test then-do) else-do))
</rkt>
```

Example. AND

```
<rkt>
  (define (AND a b) (IF a b False))
</rkt>
```

Example. OR

```
<rkt>
  (define (OR a b) (IF a True b))
</rkt>
```

Example. NOT

```
<rkt>
  (define (NOT a) (IF a False True))
</rkt>
```

Example. XOR

```
<rkt>
  (define (XOR a b) (IF a (NOT b) b))
</rkt>
```

Subtopic. 7.2 Y-combinator

Remark.
I'll only give the exact implementation here. Read my notes (a separate file under the same github directory) for further explanation

Implementation. Lazy

```
<rkt>
  (define Y
    (λ (f)
      (f (Y f))))
</rkt>
```

Implementation. Strict

```
<rkt>
  (define Y
    (λ (f)
      ((λ (self) (f (self self)))
       (λ (self) (f (self self))))))
</rkt>
```

Note. How to use it?

Step 1. Create an "almost" version of the function.
Step 2. Define y-combinator.
Step 3. Pass in the "almost" version of the function.
Step 4. Let Y do the dirty work for you.

Example. Factorial.

```
<rkt>
  (define (factorial n)
    (if (< n 2) 1 (* n (factorial (sub1 n)))))
</rkt>
```

Step 1. Create an "almost" version of the function.
We use lambdax to abstract out the explicit recursion.

```
<rkt>
  (define almost-factorial
    (lambda (f)
      (lambda (n)
        (if (< n 2) 1 (* n (f (sub1 n)))))))
</rkt>
```

Step 2. Define Y

```
<rkt>
  (define Y
    (lambda (f)
      ((f (x) (f (x x)))
```

```
            (f (x) (f (x x))))))
    </rkt>
```

Step 3. Pass in the "almost" version of the function.

```
    <rkt>
      (define factorial
        (Y almost-factorial))
    </rkt>
```

Step 4. Let Y do the dirty work for you.

Believe it or not, the above 2 lines is actually a factorial function lul.

If you don't have time to understand why it works, just remember these steps.

If you do want to understand the logic behind it, go read my "Y-combinator.pdf".

Subtopic. 7.3 List

Remark.

There is not much to explain in this section. Just read stepper and realize that everything defined like this actually work.

Note. Cons

```
<rkt>
  (define (Cons car cdr)
    (λ (selector) ((selector car) cdr)))
</rkt>

<stepper>
~> (Cons 1 3)
=> (λ (selector) ((selector 1) 3))
</stepper>
```

Note. Car/Cdr

```
<rkt>
  (define (Car pair) (pair True))
  (define (Cdr pair) (pair False))
</rkt>

<stepper>
~> (define my-pair (Cons 1 3)
=> (Car my-pair)  ;; expect 1
=> (my-pair True)
=> ((Cons 1 3) True)
=> ((λ(selector) ((selector 1) 3)) True)
=> ((True 1) 3)
=> (((λ(x) (λ(y) x)) 1) 3)
=> 1

=> (Cdr my-pair) ;; expect 3
=> (my-pair False)
=> ((Cons 1 3) False)
=> ((λ(selector) ((selector 1) 3)) False)
```

```
=> ((False 1) 3)
=> (((λ(x) (λ(y) y)) 1) 3)
=> 3
</stepper>
```

Subtopic. 7.4 Binary representation of integers

Note. Recursive definition of binary numbers

```
<rkt>
  ;; a = 0[A]: a is a binary number with last digit = 0
  ;; b = 1[B]: b is a binary number with last digit = 1

  ;; We use True to represent 1, and False to represent 0.
  ;; Also, we set zero as Empty, which is defined below:
  (define Empty (λ (x) True));; no matter what the argument is, always return True.

  ;; And we define the predicate Empty? as:
  (define (Empty? lst)
    (lst (λ (yes) (λ (no False)))))

  ;; What Empty? does, is to test if a number (which is represented as a list of True's or False's) is zero
  ;; (which is represented by Empty defined above). To see how Empty? works, see the stepper below.
</rkt>

<stepper>
  If we feed an Empty:
~> (True? (Empty? Empty))
=> (True? (Empty? (λ (x) True)))
=> (True? ((λ (x) True) (λ (yes) (λ (no) False))))
=> (True? True)
=> 'yes

  If we feed any other stuff:
~> (True? (Empty> (Cons 5 Empty))
</stepper>
```

Implementation. (ADD1 a)

```
<rkt>
  ;; There are two cases:
  ;; Case 1. If the last digit of a is zero, then simply add one to the last digit, then cons the last digit
  ;;         to the rest of a. That is, if (car a) is zero, we cons 1 to (cdr a).
  ;; Case 2. If the last digit of a is one, then adding one to a would cause the last digit to become zero
  ;;         and a one would be carried over to the next digit. That is, if (car a) is one, we cons 0 to
  ;;         (ADD1 (cdr a)).
  ;; Also remember that IF = if, Empty? = zero?, Car = car, Cdr = cdr, Cons = cons.

  (define (ADD1 a)
    (IF (Empty? (Car a))
      (Cons True (Cdr a))
      (Cons False (ADD1 (Cdr a)))))
</rkt>
```

Implementation. (ADD a b)

```
<rkt>
  ;; Base case: if a = 0, return b.
  ;; Recursive case:
  ;; Case 1. If the last digit of a = 0, then we cons the last digit of b (whether it's 0 or 1) to the
  ;;         recursive result. That is, if (car a) = 0, we cons (car b) to (ADD (cdr a) (cdr b)).
  ;; Case 2. If the last digit of a = 1 and b = 0, the last digit of result would be 1, so we cons a zero
  ;;         to the recursive result. That is, if (car a) = 1 (note that if the program skips the first
  ;;         case then the last digit of a must not be zero, thus when writing the actual program we don't
  ;;         need to write (True? (car a)) for this condition) and (car b) = 0, we cons 1 to (ADD (cdr a)
  ;;         (cdr b)).
  ;; Case 3. If the last digit of a = 1 and b = 1, the last digit of result would be 0, and a one is carried
  ;;         over to the recursive result. That is, if none of the above case matches, we cons 0 to
  ;;         (ADD1 (ADD (cdr a) (cdr b))).

  (define (ADD a b)
    (IF (Empty? a) b
      (If (Not (Car a)) (Cons (Car b) (ADD (Cdr a) (Cdr b)))
        (If (Not (Car b)) (Cons True (ADD (Cdr a) (Cdr b)))
          (Cons False (ADD1 (ADD (Cdr a) (Cdr b))))))))
</rkt>
```

Implementation. (SUB1 a)

```
<rkt>
  ;; Error case: if a = 0, return 'undefined
  ;; Base case: if a = 1, return 0
  ;; Recursive case:
  ;; Case 1. If the last digit of a is 1, cons 0 to (cdr a).
  ;; Case 2. If the last digit of a is 0, cons 1 to (SUB1 (cdr a)).

  (define (SUB1 a)
    (IF (Empty? a) 'undefined
      (IF (AND (Car a) (Empty? (Cdr a))) Empty
        (If (Car a) (Cons False (Cdr a))
          (Cons True (SUB1 (Cdr a)))))))
</rkt>
```

Implementation. (SUB a b)

```
<rkt>
  ;; Note: (SUB a b) returns b - a.
  ;; Base case: when a = 0, return b.
  ;; Recursive case:
  ;; Case 1. When the last digit of a is zero, cons (car b) to (SUB (cdr a) (cdr b)).
  ;; Case 2. When the last digit of a is one and the last digit of b is one, cons
  ;;         0 to (SUB (cdr a) (cdr b)).
  ;; Case 3. When the last digit of a is one and the last digit of b is zero, cons
  ;;         1 to (SUB1 (SUB (cdr a) (cdr b))).

  (define (SUB a b)
    (IF (Empty? a) b
      (If (Not (Car a)) (Cons (Car b) (SUB (cdr a) (cdr b)))
        (If (Car b) (Cons False (SUB (cdr a) (cdr b)))
          (Cons True (SUB1 (SUB (cdr a) (cdr b))))))))
```

```
</rkt>
```

Implementation. (MUL a b)

```
<rkt>
  ;; Base case: when a = 0 or b = 0, return 0.
  ;; Recursive case:
  ;; Case 1. When a is of the form 0[A] (we don't care about b) then we can write a = 2A.
  ;;         Multiplying a = 2A with b, we get 2A*b, or 0[A*b]. That is, if (car a) = 0,
  ;;         we cons 0 to (MUL (cdr a) b).
  ;; Case 2. When a is of the form 1[A] (we still don't care about b) then we can write
  ;;         a = 2A+1, thus a*b = (2A+1)*b = 2Ab + b. That is, if (car a) = 1, we add b
  ;;         to the result of (MUL (cdr a) b).

  (define (MUL a b)
    (IF (Empty? a) Empty
      (IF (Empty? b) Empty
        (If (Not (Car a)) (Cons False (MUL (cdr a) b))
          (ADD b (Cons True (MUL (cdr a) b)))))))
</rkt>
```

Implementation. TAN & NAT

```
<rkt>
  ;; TAN: binary -> decimal
  ;; Base case: when x = 0, return 0.
  ;; Else: if last digit = 0, multiply 2 to the recursive result; if last digit = 1,
  ;;       multiply 2 to the recursive result then plus one.

  (define (TAN x)
    (IF (Empty? x) 0
      (IF (Not (Car x)) (* 2 (TAN (Cdr x)))
        (add1 (* 2 (TAN (Cdr x)))))))

  ;; NAT: decimal -> binary
  ;; Base case: when n = 0, return 0.
  ;; Else: simple tail recursion.

  (define (NAT x)
    (IF (zero? n) Empty
      (ADD1 (NAT (sub1 n)))))
</rkt>
```

==========

END