# CSE216 Programming Abstractions

## Church Boolean

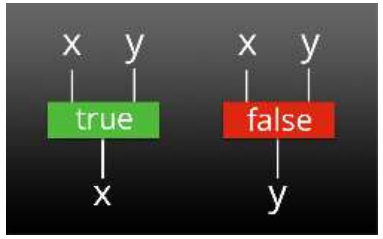
YoungMin Kwon

# Anonymous Functions for Boolean

- Goal
  - Replace Boolean literals and operators with anonymous functions (lambda)

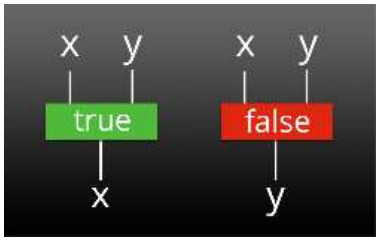
- Representations of Boolean literals


```
(*true / false*)
let true_  = fun x y -> x
let false_ = fun x y -> y

let _ = assert(1 = true_  1 0)
let _ = assert(0 = false_ 1 0)
```



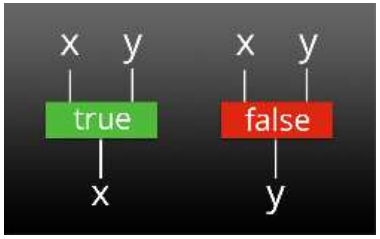
# Logical Operators

```
(*logical operators*)
let and_ = fun x y -> x y false_
let _ = assert(1 = (and_ true_  true_)   1 0)
let _ = assert(0 = (and_ true_  false_)  1 0)
let _ = assert(0 = (and_ false_  true_)  1 0)
let _ = assert(0 = (and_ false_  false_) 1 0)

(*TODO: implement the logical operators*)
let or_  = fun x y ->
let _ = assert(1 = (or_  true_  true_)   1 0)
let _ = assert(1 = (or_  true_  false_)  1 0)
let _ = assert(1 = (or_  false_  true_)  1 0)
let _ = assert(0 = (or_  false_  false_) 1 0)

let not_ = fun z ->
let _ = assert(0 = (not_ true_)  1 0)
let _ = assert(1 = (not_ false_) 1 0)
```



# Conditional Operator

```
(*conditional operator*)
(*TODO: implement conditional operator*)
let if_ = fun p t f ->

let _ = assert(1 = (if_ true_  1 2))
let _ = assert(2 = (if_ false_ 1 2))
```



# Comparators

```
(*comparators*)
let ge  = fun x y -> if x >= y then true_ else false_
let abs = fun x -> if_ (ge 0 x) (- x) x
let _ = assert((3,3) = (abs (- 3), abs 3))

let le  = fun x y -> ge y x
let abs = fun x -> if_ (le x 0) (- x) x
let _ = assert((3,3) = (abs (- 3), abs 3))
```

```
(*TODO: using ge, le and, logical operators,
        implement gt, lt, eq, ne*)
let gt  = fun x y ->
let abs = fun x -> if_ (gt 0 x) (- x) x
let _ = assert((3,3) = (abs (- 3), abs 3))

let lt  = fun x y ->
let abs = fun x -> if_ (lt x 0) (- x) x
let _ = assert((3,3) = (abs (- 3), abs 3))

let eq  = fun x y ->
let abs = fun x -> if_ (eq (x * abs x) (x * x)) x (- x)
let _ = assert((3,3) = (abs (- 3), abs 3))

let ne  = fun x y ->
let abs = fun x -> if_ (ne (x * abs x) (x * x)) (- x) x
let _ = assert((3,3) = (abs (- 3), abs 3))
```

# Implementing Functions

- Applicative order evaluation
  - The following implementation does not work.
  - Why?

```
(*implementing functions*)
let rec fact x =
    (if_ (eq x 0)
        1
        (x * fact (x - 1)))

let _ = fact 4
```

# Implementing Functions

- Delaying the immediate function application
  - Thunk: wrap a function invocation with another function definition

```
(*thunk: delay function call*)
let rec fact x =
    (if_ (eq x 0)
        (fun u -> 1)
        (fun u -> x * fact (x - 1)))
    ()

let _ = assert(24 = fact 4)
```

# Implementing Functions

```
(*
let rec gcd a b =
    (if_ (gt a b)
        (gcd (a - b) b)
    (if_ (lt a b)
        (gcd (b - a) a)
        a))
let _ = gcd 12 16
*)

(*TODO: implement gcd using thunk*)
let rec gcd a b =

let _ = assert(4 = gcd 12 16)
```