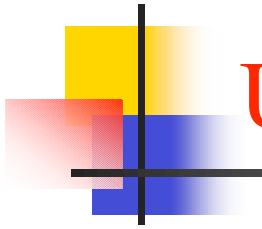
Clever Programming With Functions

Prof. Robert “Corky” Cartwright
Department of Computer Science
Rice University



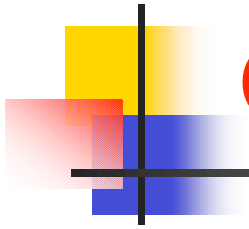
Using Functions to Represent Objects

- How can we represent a pair so that the only operations that code can perform on pairs are:

```
(create-pair x y)
(pair-first p)
(pair-second p)
(pair-equal? p1 p2)
```

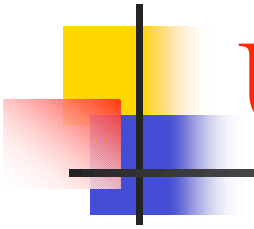
- What if we represent a pair as a list? As a struct? Structs are not as robust as you might think. In the advanced language level try:

```
(define-struct Pair (first second))
(define p (make-Pair 1 2))
(set-Pair-first! P 17)
p
```



Objects as closures

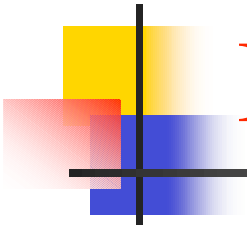
```
(define (make-pair x y)
  (lambda (msg)
    (cond [(equal? msg 'first) (lambda () x)]
          [(equal? msg 'second) (lambda () y)]
          [(equal? msg 'equal)
           (lambda (p)
             (and (equal? (pair-first p) x)
                  (equal? (pair-second p) y)))])))
(define (pair-first p) ((p 'first)))
(define (pair-second p) ((p 'second)))
(define (pair-equal? p1 p2) ((p1 'equal) p2))
```



Useful Functionals

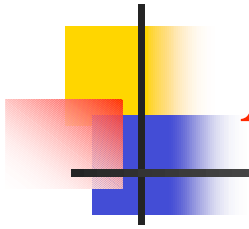
- What is a *functional*? A function that takes a function as a argument and often returns a function. The differential and integral operators in calculus are functionals.

- An important functional in functional programming: curry



Idea Behind curry

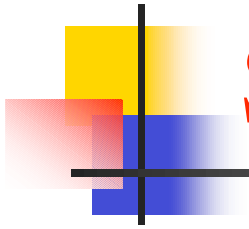
- Every function of the form
  ```
  A B -> C
  ```
  can be converted to a function of type
  ```
  A -> (B -> C)
  ```
  which is often more convenient.
- In set theory, here is an isomorphism between
  $A \times B \to C$
  and
  $A \to B \to C$
- This correspondence roughly holds for programming language types.



A Simple Example

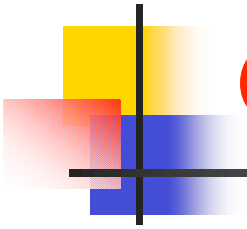
```
map : A          B    -> C
map : (X -> Y) [X] -> [Y]

map' : A       -> (B    -> C)
map' : (X->Y) -> ([X] -> [Y])
```



Standard Map

```
(define map
  (lamba (f l)
    (cond
      [(empty? l) empty]
      [else
        (cons (f (first l))
              (map f (rest l)))])))
```



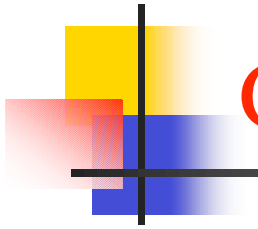
Curried Map

- A definition in terms of **map**

```
(define (map' f)
  (lambda (l) (map f l)))
```

- When written from scratch, it looks almost exactly like **map**:

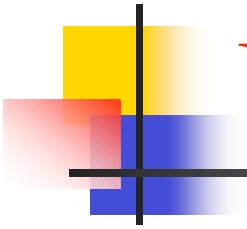
```
(define map'
  (lambda (f)
    (lambda (l)
      (cond
        [(empty? l) empty]
        [else
          (cons (f (first l))
                (map f (rest l)))]))))
```



Can We Define a Functional that Curries?

- Unfortunately, we need a separate curry function for each function arity >= 2.

```
(define (curry f)
  (lambda (x)
    (lambda (y) (f x y))))
```



Uncurry

- Challenge: See if you can write

```
uncurry : (A -> (B -> C)) -> (A B -> C)
```

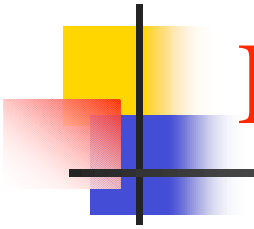
- Note the equational properties:

```
curry (uncurry f) = f
uncurry (curry f) = f
```

- These are laws in mathematics, but the second fails in programming languages. It doesn't hold in either CBN or CBV. Yet these equations are widely taught by PL experts as if PL domain theory was set theory
- What goes wrong? Error/exception behavior.

Bonus Material
Another Important Functional: **Y**

- lambda-notation (as in Scheme) indirectly supports recursion. How? A clever construction based on sophisticated mathematics.
- Short story: solutions to recursion equations are "fixed points". Given the equation
  $f(x) = E_f$, (which is equivalent to $f = \lambda x.\ E_f$)
  what is the least solution f^*? Under proper conditions,
  $f^* = \textbf{lub}\ F^i(\perp)$
  where $F(f) = \lambda f.\lambda x.\ E_f)$ and $\perp$ is the least-defined function (i.e., the function denoted by Ω). **Y** is defined by $\mathbf{Y}(F) = f^*$ where $F(f^*) = f^*$ (f^* is least solution)

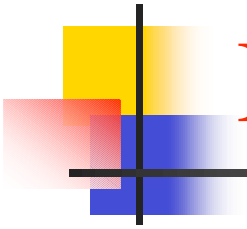


Defining **Y**

- λ-calculus programming trick:
  $(\lambda x.\, f(x\ x))\ (\lambda x.\, f(x\ x)) =$
  $f[(\lambda x.\, f(x\ x))\ (\lambda x.\, f(x\ x))] =$
  $f^2[(\lambda x.\, f(x\ x))\ (\lambda x.\, f(x\ x))] =$
  …
  $f^k[(\lambda x.\, f(x\ x))\ (\lambda x.\, f(x\ x))] =$
  …
- In CBN languages $\mathbf{Y} = \lambda f\,.\ (\lambda x.\, f(x\ x))\ (\lambda x.\, f(x\ x))$
- CBV is slightly harder and messier because $\mathbf{Y}F$ does not terminate
- Default for λ-calculus is CBN. Default for programming languages is CBV.

Bonus Material
Other Powerful Functionals: S,K

- Every closed λ-expression can be written without **any** variables given the three primitive functionals **S**, **K**, **I** where
  - **S** = $\lambda x.\lambda y.\lambda z.\ (xz)(yz)$
  - **K** = $\lambda x.\lambda y.\ x$
  - **I** = $\lambda x.x$
- In fact, you only need two because
  - **I** = **S**(**KK**)
- Functionals defined by closed λ terms (and nothing else are called **combinators**. Y (in all its varieties) is a combinator.



For Next Class

- Homework due Monday
- Reading:
  - Review for coming exam which will be distributed on Friday.