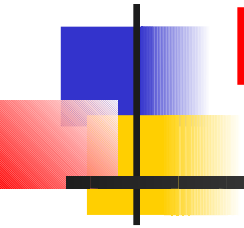
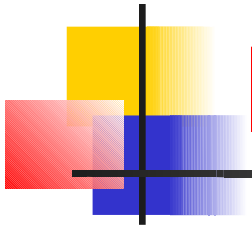


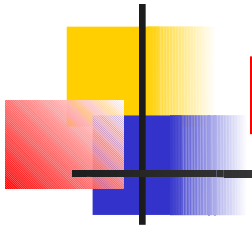
Functions as Values

Corky Cartwright
Stephen Wong
Rice University



Functional Abstraction

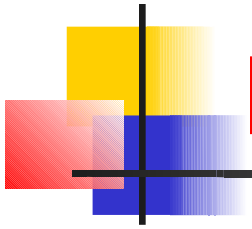
- A powerful tool
  - Makes programs much more concise
  - Avoids redundancy
  - Promotes "single point of control" (no code duplication)
- Generally involves polymorphic contracts (contracts containing type variables)
- What we cover today for lists applies to *any* recursive (self-referential) type



Look for the pattern

One function:

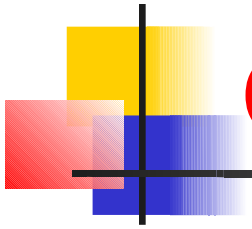
```
; add1-each : (list-of number) -> (list-of number)
; Purpose: adds one to each number in list
(define (add1-each l)
  (cond [(empty? l) empty]
        [else
           (cons (add1 (first l))
                 (add1-each (rest l)))]))
```



Look for the pattern

Another function:

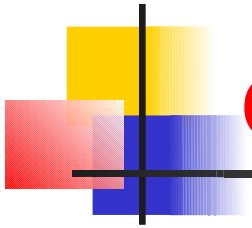
```
; not-each : (listOf boolean) -> (listOf boolean)
; Purpose: complements each boolean in the list
(define (not-each l)
  (cond [(empty? l) empty]
        [else (cons (not (first l))
                    (not-each (rest l)))]))
```



Codify the pattern

Abstracting with respect to `add1`, `not`, and the element type in the lists:

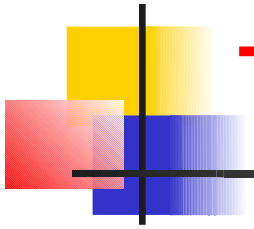
```
; map : (X -> X), (listOf X) -> (listOf X)
; applies f to each element in l
(define (map f l)
  (cond [(empty? l) empty]
        [else (cons (f (first l))
                    (map f (rest l)))]))
```



Generalize the pattern (and typing)

Do all occurrences of **X** in contract of **map** need to be of the same type?

```
; map : (X -> Y) (list-of X) -> (list-of Y)
; Purpose: (map f l) returns the list consisting of f
; applied to each element in l

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

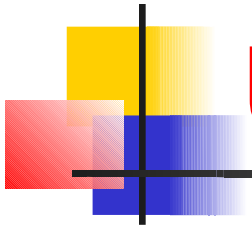


Tip on Generalizing Types

- When we generalize, we **only** replace
  - specific types (like **`number`** or **`symbol`**) or type variables (like **`X`** or **`Y`**)
  - by (other) type *variables*
- We almost **never** replace a type by the type **`any`**, which actually means

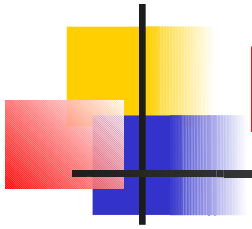
```
number | boolean | list-of number |
  list-of ... |  number -> number | ...
```

- What goes wrong if we use **`any`**? We cannot *instantiate* (bind) **`any`** as a custom type



Use the pattern

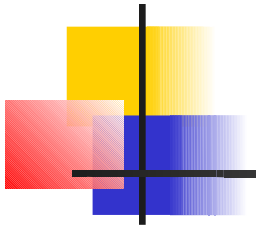
- `map` can be used with *any* unary function.
- `(map not l)`
- `(map sqr l)`
- `(map length l)`
- `(map first l)`
- `(map symbol? l)`
- Note: other recursive data types also have maps!



More about `map`

- Powerful tool for parallel computing!
- Aside: functional programming generally supports parallelism (a theme developed in Comp 322) because every disjoint sub-expression can be independently evaluated. In every function application **`(f arg1 .. . argn)`**, the arguments can be evaluated in parallel. In fact, the evaluation of **`f`** can be started as well, but it must wait for argument values (*futures*).
- Has elegant properties (from mathematics):
  - **`(map f (map g l)) = (map (compose f g) l)`**

  Soon we will see how to define **`compose`**
- For fun: Checkout Google's "map/reduce"



Better notation for function values

- Assume we want to square all of the elements in a list. How can we do this using `map` in a compact expression? We need simple notation for denoting new functions without the overhead of introducing a name for the function, *e.g.*, using `local`. Alonzo Church invented such an notation in the 1930's called *lambda*-notation. In Church's scheme

  $\lambda x \,.\, M$

  denotes the function *f* defined by the equation

  $f(x) = M.$

- Lisp (the progenitor of Scheme) adopted this notation for functions. In particular,

  (lambda (x$_1$ .. x$_n$) E)

  denotes the function ***f*** defined by:

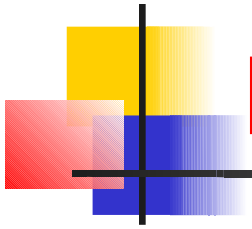
  (define (*f* x$_1$ .. x$_n$) E)

  In fact, a top-level function definition

  (define (*f* x$_1$ .. x$_n$) E)

  can also be written

  (define *f* (lambda (x$_1$ .. x$_n$) E))



Examples of `lambda`

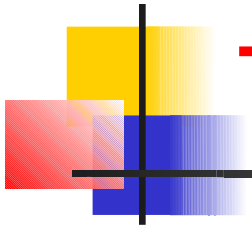
```
; square the elements in a list
    (map (lambda (x) (* x x)) '(1 2 3 4))
=>* '(1 4 9 16)

; compose: (Y -> Z) (X -> Y) -> (X -> Z)
; Purpose: (compose f g) returns the composition
;   of unary functions f and g;
(define (compose f g) (lambda (x) (f (g x))))

    (map (compose add1 square) '(1 2 3 4))
=>* '(2 5 10 17)
```

Expressing **`lambda`** using **`local`** is straightforward, but ugly

`(lambda (`x_1 `...` X_n`) M) <=>`

`(local [(define (new-v `x_1 `...` x_n`) M)] new-v)`

where **`new-v`** is a fresh variable.

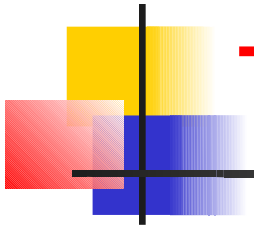


Templates as functions

- Recall the template for lists:

```
; (define (f l)
;  (cond
;    [(empty? l) ...]
;    [else ... (first l)
;          ... (f (rest l)) ... ]))
```

Can we construct a function **`foldr`** that takes the "..." for **`empty?`** and the "..." for **`else`** as parameters **`init`** and **`op`**? Yes. The **`op`** parameter must be a function because it must process **`(first l)`** and **`(fn (rest l))`**.



Templates as functions

The abstraction looks just like this:

```
; the contract is not obvious;
  (define (foldr op init l)
     (cond [(empty? l) init]
           [else
             (op (first l)
                 (foldr op init (rest l)))]))
```

Intuitively,

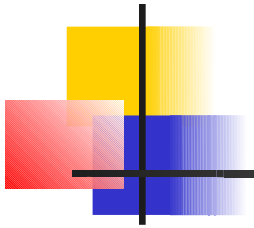
```
     (foldr op init (list e1 ... en))
=>* (op e1 (op e2 ... (op en init) ...)))
```

which is

```
     e1 op ( ... (en op init) ... ))
```

in infix notation.

Can we express all functions we've written using **`foldr`**?
What is the type of **`foldr`**?

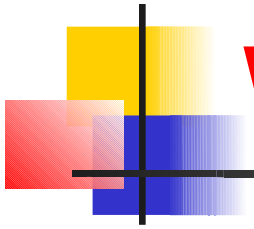


map in terms of **foldr**

Can we write **map** in terms of **foldr**? Yes.

```
; map: (X->Y) list-of-X -> list-of-Y
(define (map f l)
  (foldr (lambda (x l)(cons (f x) l))
         empty
         l))
```

Note that **foldr** performs the recursion.



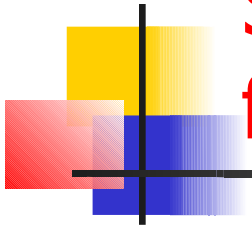
What is the type of `foldr`?

```
; foldr: (X X -> X) X list-of-X -> X
```

Reasoning: in **(foldr op init alox)**, **alox** is a **list-of-X** for some type **X** , implying (in simple cases) that **op** is a binary operation on values of type **X** and **init** is a value of type **X**.

But there is a more general type for cases when **op** returns a different type **Y** than its first input type **X**. Since **op** takes its output type as its second argument type, **op** must have type **X Y -> Y**. Similarly, **init** must have type **Y** and the output of **foldr** must have type **Y**.

```
; foldr: (X Y -> Y)  Y  (list-of X) -> Y
; (foldr op init (list e1 ... en)) returns
; (op e1 ( ... (op en init) ... )) which is
; e1 op ( ... (en op init) ... )) in infix notation
```



Should all our template-based functions be written using `foldr`?

Some functional programmers would say yes.

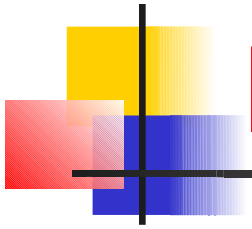
But the two justifications for introducing abstractions are:

- to eliminate duplication of code that conceivably could be changed
- to simplify reasoning about programs

Could the definition of `foldr` conceivably change. No.

Is the `foldr` abstraction helpful in reasoning about functions defined using it? Debatable.

Is the `foldr` definition of `map` easier to understand? I think not.



For Next Class

- Homework due next Monday. Don't dally.

- Reading:
  - Ch 21-22: Abstracting designs and first class functions