Function Abstraction

Type Abstraction

Anonymous Functions

Big Fish

A function that gets the big fish (> 5 lbs):

```scheme
; big : list-of-nums -> list-of-nums
(define (big l)
  (cond
    [(empty? l) empty]
    [(cons? l)
      (cond
        [(> (first l) 5)
         (cons (first l) (big (rest l)))]
        [else (big (rest l))])
    ])
(big empty) "should be" empty
(big '(7 4 9)) "should be" '(7 9)
```

Better with `local`:

```scheme
; big : list-of-nums -> list-of-nums
(define (big l)
  (cond
    [(empty? l) empty]
    [(cons? l)
      (local [(define big-rest (big (rest l)))]
        (cond
          [(> (first l) 5)
           (cons (first l) big-rest)]
          [else big-rest]))))
```

Suppose we also need to find huge fish...

Huge Fish

Huge fish (> 10 lbs):

```scheme
; huge : list-of-nums -> list-of-nums
(define (huge l)
  (cond
    [(empty? l) empty]
    [(cons? l)
      (local [(define h-rest (huge (rest l)))]
        (cond
          [(> (first l) 10)
           (cons (first l) h-rest)]
          [else h-rest]))))
```

How do you suppose I made this slide?

Cut and Paste!
The Trouble With Cut and Paste

; big : list-of-nums -> list-of-nums
(define (big l)
  (cond
   [(empty? l) empty]
   [(cons? l)
    (cond
      [(> (first l) 5)
       (cons (first l) (big (rest l)))]
      [else (big (rest l))])]))

; huge : list-of-nums -> list-of-nums
(define (huge l)
  (cond
   [(empty? l) empty]
   [(cons? l)
    (cond
      [(> (first l) 10)
       (cons (first l) (huge (rest l)))]
      [else (huge (rest l))])]))

After cut-and-paste, improvement is twice as hard
The Trouble With Cut and Paste

; big : list-of-nums -> list-of-nums
(define (big l)
  (cond
   [(empty? l) empty]
   [(cons? l) (local [(define big-rest (big (rest l)))]
     (if (> (first l) 5)
      (cons (first l) big-rest)
      [else big-rest]))]))

; huge : list-of-nums -> list-of-nums
(define (huge l)
  (cond
   [(empty? l) empty]
   [(cons? l) (local [(define h-rest (huge (rest l)))]
     (if (> (first l) 10)
      (cons (first l) h-rest)
      [else h-rest]))]))

After cut-and-paste, bugs multiply

How to Avoid Cut-and-Paste

Start with the original function...

; big : list-of-nums -> list-of-nums
(define (big l)
  (cond
   [(empty? l) empty]
   [(cons? l) (local [(define big-rest (big (rest l)))]
     (if (> (first l) 5)
      (cons (first l) big-rest)
      [else big-rest]))]))
How to Avoid Cut-and-Paste

... and add arguments for parts that should change

; bigger : list-of-nums num -> list-of-nums
(define (bigger l n)
  (cond
   [(empty? l) empty]
   [(cons? l)
    (local [(define r (bigger (rest l) n))]
      (cond
       [(> (first l) n)
        (cons (first l) r)]
       [else r]))])))

(define (big l) (bigger l 5))
(define (huge l) (bigger l 10))

Small Fish

Now we want the small fish:

; smaller : list-of-nums num -> list-of-nums
(define (smaller l n)
  (cond
   [(empty? l) empty]
   [(cons? l)
    (local [(define r (smaller (rest l) n))]
      (cond
       [(< (first l) n)
        (cons (first l) r)]
       [else r]))])))

(define (small l) (smaller l 5))

Sized Fish

; sized : list-of-nums num ... -> list-of-nums
(define (sized l n COMP)
  (cond
   [(empty? l) empty]
   [(cons? l)
    (local [(define r
                  (sized (rest l) n COMP))]
      (cond
       [(COMP (first l) n)
        (cons (first l) r)]
       [else r]))])))

(define (bigger l n) (sized l n >))
(define (smaller l n) (sized l n <))

Functions as Values

The definition

(define (bigger l n) (sized l n >))

works because functions are values

• 10 is a num
• false is a bool
• < is a (num num -> bool)

So the contract for sized is

; list-of-nums num (num num -> bool)
; -> list-of-nums
Sized Fish

; sized : list-of-nums num (num num -> bool) ; -> list-of-nums
(define (sized l n COMP)
  (cond
    [(empty? l) empty]
    [(cons? l)
      (local [(define r
          (sized (rest l) n COMP))]
        (cond
          [(COMP (first l) n)
            (cons (first l) r)]
          [else r]))])
  (define (tiny l) (sized l 2 <))
  (define (medium l) (sized l 5 =))

How about all fish between 3 and 7 lbs?

Mediumish Fish

; btw-3-and-7 : num num -> bool
(define (btw-3-and-7 a ignored-zero)
  (and (>= a 3)
       (<= a 7)))

(define (mediumish l) (sized l 0 btw-3-and-7))

- Programmer-defined functions are values, too
- Note that the contract of btw-3-and-7 matches the kind expected by sized

But the ignored 0 suggests a simplification of sized...

A Generic Number Filter

; filter-nums : (num -> bool) list-of-num ; -> list-of-num
(define (filter-nums PRED l)
  (cond
    [(empty? l) empty]
    [(cons? l)
      (local [(define r
          (filter-nums PRED (rest l)))
        (cond
          [(PRED (first l))
            (cons (first l) r)]
          [else r]))])])

(define (btw-3&7 n) (and (>= n 3) (<= n 7)))
(define (mediumish l) (filter-nums btw-3&7 l))

27-33
Big and Huge Fish, Again

(define (more-than-5 n)  
  (> n 5))
(define (big l)  
  (filter-nums more-than-5 l))

(define (more-than-10 n)  
  (> n 10))
(define (huge l)  
  (filter-nums more-than-10 l))

The more-than-5 and more-than-10 functions are really only useful to big and huge

We could make them local to clarify...

Big and Huge Fish, Improved

(define (big l)  
  (local [(define (more-than-5 n)  
    (> n 5))]
    (filter-nums more-than-5 l))))

(define (huge l)  
  (local [(define (more-than-10 n)  
    (> n 10))]
    (filter-nums more-than-10 l))))

Cut and paste alert!
You don’t think I typed that twice, do you?

Big and Huge Fish, Generalized

(define (bigger-than l m)  
  (local [(define (more-than-m n)  
    (> n m))]
    (filter-nums more-than-m l)))

(define (big l) (bigger-than l 5))
(define (huge l) (bigger-than l 10))

Big Example

...  
(define (bigger-than l m)  
  (local [(define (more-than-m n)  
    (> n m))]
    (filter-nums more-than-m l))))

(define (big l) (bigger-than l 5)) ...
(big '(7 4 9))
(huge '(7 4 9))

→

...  
(define (bigger-than l m)  
  (local [(define (more-than-m n)  
    (> n m))]
    (filter-nums more-than-m l))))

...  
(bigger-than '(7 4 9) 5)
(huge '(7 4 9))
(define (bigger-than l m)
  (local [[(define (more-than-m n)
                (> n m))]
            (filter-nums more-than-m l))
  (bigger-than '(7 4 9) 5)
(huge '(7 4 9))
→

(bigger-than '(7 4 9) 10)

...
... (define (bigger-than l m)
    (local ((define (more-than-m n)
                (> n m))
      (filter-nums more-than-m l)))
... (define (more-than-m42 n)
    (> n 5))
'(7 9)
(bigger-than '(7 4 9) 10)

→

... (define (more-than-m42 n)
    (> n 5))
'(7 9)
(local ((define (more-than-m n)
          (> n 10))
       (filter-nums more-than-m '(7 4 9)))

Abstraction

- Avoiding cut and paste is abstraction
- No real programming task succeeds without it
Symbols

Our favorite `list-of-sym` program:

```scheme
(define (eat-apples l)
  (cond
    [(empty? l) empty]
    [(cons? l)
      (local [(define ate-rest (eat-apples (rest l)))
        (cond
          [(symbol=? (first l) 'apple) ate-rest]
          [else (cons (first l) ate-rest)]))])))
```

° How about `eat-bananas`?
° How about `eat-non-apples`?

We know where this leads...

Filtering Symbols

```scheme
(define (filter-syms PRED l)
  (cond
    [(empty? l) empty]
    [(cons? l)
      (local [(define r
        (filter-syms PRED (rest l)))
        (cond
          [(PRED (first l))]
          [else r]))])))
```

This looks really familiar

Last Time: Filtering Numbers

```scheme
(define (filter-nums PRED l)
  (cond
    [(empty? l) empty]
    [(cons? l)
      (local [(define r
        (filter-nums PRED (rest l)))
        (cond
          [(PRED (first l))]
          [else r]))]))
```

How do we avoid cut and paste?

Filtering Lists

We know this function will work for both number and symbol lists:

```scheme
(define (filter PRED l)
  (cond
    [(empty? l) empty]
    [(cons? l)
      (local [(define r
        (filter PRED (rest l)))
        (cond
          [(PRED (first l))]
          [else r]))])))
```

But what is its contract?
The Contract of Filter

How about this?

\[(\text{num-OR-sym} \rightarrow \text{bool}) \ \text{list-of-num-OR-list-of-sym} \rightarrow \text{list-of-num-OR-list-of-sym}\]

; A num-OR-sym is either
; - num
; - sym

; A list-of-num-OR-list-of-sym is either
; - list-of-num
; - list-of-sym

This contract is too weak to define \textit{eat-apples}

; \textit{eat-apples} : list-of-sym \rightarrow list-of-sym
(define (eat-apples l)
  (filter not-apple? l))

; not-apple? : sym \rightarrow bool
(define (not-apple? s)
  (not (symbol=? s 'apple)))

\textit{eat-apples} must return a list-of-sym, but by its contract, \textit{filter} might return a list-of-num

The reason \textit{filter} works is that if we give it a list-of-sym, then it returns a list-of-sym

Also, if we give \textit{filter} a list-of-sym, then it calls \textit{PRED} with symbols only

A better contract:

\textit{filter} :
\((\text{num} \rightarrow \text{bool}) \ \text{list-of-num} \rightarrow \text{list-of-num})
\textbf{OR}
\((\text{sym} \rightarrow \text{bool}) \ \text{list-of-sym} \rightarrow \text{list-of-sym})

But what about a list of images, posns, or snakes?
The True Contract of Filter

The real contract is

\[ \text{filter} : ((X \to \text{bool}) \to \text{list-of-X} \to \text{list-of-X}) \]

where \(X\) stands for any type

- The caller of \text{filter} gets to pick a type for \(X\)
- All \(X\)s in the contract must be replaced with the same type

Data definitions need type variables, too:

; A list-of-X is either
; - empty
; - (cons X empty)

Looking for Other Built-In Functions

Recall \text{inflate-by-4\%}:

\[
\text{define (inflate-by-4\% l)}
\begin{cases}
\text{empty? l) empty} \\
\text{else (cons (* (first l) 1.04) (inflate-by-4\% (rest l)))}
\end{cases}
\]

Is there a built-in function to help?

Yes: \text{map}

Using Filter

The \text{filter} function is so useful that it's built in

New solution:

\[
\begin{align*}
\text{(define (eat-apples l))} \\
\text{(local [(define (not-apple? s))} \\
\text{ (not (symbol=? s 'apple)))]} \\
\text{(filter not-apple? l))}
\end{align*}
\]

Using Map

\[
\begin{align*}
\text{(define (map CONV l)} \\
\text{(cond} \\
\text{[(empty? l) empty]} \\
\text{[else (cons (CONV (first l)) (map CONV (rest l)))])})
\end{align*}
\]

(\text{define (inflate-by-4\% l)}
\begin{cases}
\text{(local [(define (inflate-one n)} \\
\text{ (* n 1.04))]} \\
\text{(map inflate-one l))})
\end{cases}
\]

; negate-colors : list-of-col -> list-of-col
\text{(define (negate-colors l)}
\text{(map negate-color 1))}
The Contract for Map

```
(define (map CONV 1)
  (cond
    [(empty? 1) empty]
    [else (cons (CONV (first 1))
               (map CONV (rest 1)))])
```

- The 1 argument must be a list of X
- The CONV argument must accept each X
- If CONV returns a new X each time, then the contract for map is
  map : (X -> X) list-of-X -> list-of-X

The True Contract of Map

Despite the contract mismatch, this works!

```
(define (distances 1)
  (map distance-to-0 1))
```

The true contract of map is

map : (X -> Y) list-of-X -> list-of-Y

The caller gets to pick both X and Y independently

Posns and Distances

Another function from HW 4:

```
; distances : list-of-posn -> list-of-num
(define (distances l)
  (cond
    [(empty? l) empty]
    [(cons? l) (cons (distance-to-0 (first l))
                      (distances (rest l)))])
```

The distances function looks just like map, except that

distances-to-0 is

```
posn -> num
```

not

```
posn -> posn
```

More Uses of Map

```
; modernize : list-of-pipe -> list-of-pipe
(define (modernize l)
  ; replaces 4 lines:
  (map modern-pipe l))
```

```
; modern-pipe : pipe -> pipe
...;
```

```
; rob-train : list-of-car -> list-of-car
(define (rob-train l)
  ; replaces 4 lines:
  (map rob-car l))
```

```
; rob-car : car -> car
...;
```
Folding a List

How about sum?

\[
\text{sum : list-of-num} \rightarrow \text{num}
\]

Doesn’t return a list, so neither filter nor map help

But recall combine-nums...

\[
; \text{combine-nums : list-of-num num}
; (\text{num num} \rightarrow \text{num}) \rightarrow \text{num}
\]

\[
\text{(define (combine-nums l base-n COMB)}
\]

\[
\begin{align*}
[(\text{empty? l)} & \text{ base-n}] \\
[(\text{cons? l)} & \text{ COMB)} \\
(\text{first l)} & \text{ (combine-nums (rest l) base-n COMB)))]
\end{align*}
\]

The Foldr Function

\[
; \text{foldr : (X Y } \rightarrow \text{ Y) Y list-of-X } \rightarrow \text{ Y}
\]

\[
\text{(define (foldr COMB base l)}
\]

\[
\begin{align*}
[(\text{empty? l)} & \text{ base]} \\
[(\text{cons? l)} & \text{ COMB (first l)} \\
(\text{foldr COMB base (rest l)}))]
\end{align*}
\]

The \textit{sum} and \textit{product} functions become trivial:

\[
\text{(define (sum l) (foldr + 0 l))}
\]

\[
\text{(define (product l) (foldr } \ast \text{ 1 l))}
\]

The Foldr Function

\[
; \text{foldr : (X Y } \rightarrow \text{ Y) Y list-of-X } \rightarrow \text{ Y}
\]

\[
\text{(define (foldr COMB base l)}
\]

\[
\begin{align*}
[(\text{empty? l)} & \text{ base]} \\
[(\text{cons? l)} & \text{ COMB (first l)} \\
(\text{foldr COMB base (rest l)}))]
\end{align*}
\]

Useful for HW 5:

\[
; \text{total-blue : list-of-col } \rightarrow \text{ num}
\]

\[
\text{(define (total-blue l)}
\]

\[
\text{(local [(define (add-blue c n)}
\]

\[
(\text{+ (color-blue c) n)])}
\]

\[
(\text{foldr add-blue 0 l)}))
\]

The Foldr Function

\[
; \text{foldr : (X Y } \rightarrow \text{ Y) Y list-of-X } \rightarrow \text{ Y}
\]

\[
\text{(define (foldr COMB base l)}
\]

\[
\begin{align*}
[(\text{empty? l)} & \text{ base]} \\
[(\text{cons? l)} & \text{ COMB (first l)} \\
(\text{foldr COMB base (rest l)}))]
\end{align*}
\]

In fact,

\[
\text{(define (map f l)}
\]

\[
\text{(local [(define (comb i r)}
\]

\[
(\text{cons (f i) r)])}
\]

\[
(\text{foldr comb empty l))}
\]
The Foldr Function

; foldr : (X Y -> Y) Y list-of-X -> Y
(define (foldr COMB base l)
  (cond
    [(empty? l) base]
    [(cons? l)
     (COMB (first l)
           (foldr COMB base (rest l)))]))

Yes, filter too:

(define (filter f l)
  (local [(define (check i r)
            (cond
              [(f i) (cons i r)]
              [else r)])]
    (foldr check empty l)))

The Source of Foldr

How can foldr be so powerful?

Template:

(define (func-for-loX l)
  (cond
    [(empty? l) ...]
    [(cons? l) ... (first l)
     ... (func-for-loX (rest l)) ...]))

Fold:

(define (foldr COMB base l)
  (cond
    [(empty? l) base]
    [(cons? l)
     (COMB (first l)
           (foldr COMB base (rest l)))]))

Other Built-In List Functions

More specializations of foldr:

ormap : (X -> bool) list-of-X -> bool
andmap : (X -> bool) list-of-X -> bool

Examples:

; got-milk? : list-of-sym -> bool
(define (got-milk? l)
  (local [(define (is-milk? s)
            (symbol=? s 'milk))]
    (ormap is-milk? s)))

; all-passed? : list-of-grade -> bool
(define (all-passed? l)
  (andmap passing-grade? l))
What about Non-Lists?

Since it's based on the template, the concept of fold is general

```scheme
; fold-ftn : (sym num sym Z Z Z Z) Z ftn -> Z
(define (fold-ftn COMB base ftn)
  (cond
   [(empty? ftn) base]
   [(child? ftn)
     (COMB (child-name ftn) (child-date ftn) (child-eyes ftn)
       (fold-ftn COMB BASE (child-father ftn)))
     (fold-ftn COMB BASE (child-mother ftn)))]
)

(define (count-persons ftn)
  (local [(define (add name date color c-f c-m)
               (+ 1 c-f c-m))]
    (fold-ftn add 0 ftn)))

(define (in-family? who ftn)
  (local [[(define (here? name date color in-f? in-m?)
             (or (symbol=? name who) in-f? in-m?)]]
    (fold-ftn here? false ftn)))
```

Function Abstraction

Type Abstraction

Anonymous Functions

Values and Names

Some Values:

- **Numbers:** 1, 17.8, 4/5
- **Booleans:** true, false
- **Lists:** empty, (cons 7 empty)
- ...
- **Function names:** less-than-5, first-is-apple? given
  (define (less-than-5? n) ...)  
  (define (first-is-apple? a b) ...)

Why do only function values require names?

Naming Everything

Having to name every kind of value would be painful:

```
(local [[(define (first-is-apple? a b)
             (symbol=? a 'apple))]
         (choose '(apple banana) '(cherry cherry)
           first-is-apple?)])
```

Would have to be

```
(local [[(define (first-is-apple? a b)
             (symbol=? a 'apple))]
         (define al '(apple banana))
         (define bl '(cherry cherry))
         (choose al bl first-is-apple?)])
```

Fortunately, we don't have to name lists
Naming Nothing

Can we avoid naming functions?
In other words, instead of writing

```scheme
(define (first-is-apple? a b)
  (symbol=? a 'apple))
```

... first-is-apple? ...

we’d like to write

```scheme
... function that takes a and b
    and produces (symbol=? a 'apple)
    ...
```

We can do this

```
Lambda

An *anonymous function* value:

```
(lambda (a b) (symbol=? a 'apple))
```

Using `lambda` the original example becomes

```
(choose '(apple banana) '(cherry cherry)
  (lambda (a b) (symbol=? a 'apple)))
```

Why the funny keyword `lambda`?

It’s a 70-year-old convention: the Greek letter \( \lambda \) means "function"

Using Lambda

In DrScheme:

```scheme
> (lambda (x) (+ x 10))
(lambda (a1) ...)
```

Unlike most kinds of values, there’s no one shortest name:

- The argument name is arbitrary
- The body can be implemented in many different ways

So DrScheme gives up — it invents argument names and hides the body

```
Using Lambda

In DrScheme:

```scheme
> ((lambda (x) (+ x 10)) 17)
27
```

The function position of an *application* (i.e., function call) is no longer always an identifier

Some former syntax errors are now run-time errors:

```scheme
> (2 3)
procedure application: expected procedure, given 2
```
Defining Functions

What’s the difference between

```scheme
(define (f a b)
  (+ a b))
```

and

```scheme
(define f (lambda (a b)
  (+ a b)))
```

Nothing — the first one is (now) a shorthand for the second.

Lambda and Built-In Functions

Anonymous functions work great with `filter`, `map`, etc.:

```scheme
(define (eat-apples l)
  (filter (lambda (a)
    (not (symbol=? a 'apple))) l))

(define (inflate-by-4% l)
  (map (lambda (n) (* n 1.04)) l))

(define (total-blue l)
  (foldr (lambda (c n)
    (+ (color-blue c) n)) 0 l))
```

Functions that Produce Functions

We already have functions that take function arguments

```scheme
map : (X -> Y) list-of-X -> list-of-Y
```

How about functions that produce functions?

Here’s one:

```scheme
; make-adder : num -> (num -> num)
(define (make-adder n)
  (lambda (m) (+ m n)))

(map (make-adder 10) '(1 2 3))
(map (make-adder 11) '(1 2 3))
```

Using Functions that Produce Functions

Suppose that we need to filter different symbols:

```scheme
(filter (lambda (a) (symbol=? a 'apple)) l)
(filter (lambda (a) (symbol=? a 'banana)) l)
(filter (lambda (a) (symbol=? a 'cherry)) l)
```

Instead of repeating the long `lambda` expression, we can abstract:

```scheme
; mk-is-sym : sym -> (sym -> bool)
(define (mk-is-sym s)
  (lambda (a) (symbol=? s a)))

(filter (mk-is-sym 'apple) l)
(filter (mk-is-sym 'banana) l)
(filter (mk-is-sym 'cherry) l)
```

`mk-is-sym` is a [curried](https://en.wikipedia.org/wiki/Currying) version of `symbol=?`
! Currying Functions!

This **curry** function curries any 2-argument function:

\[
\text{curry} : (X \ Y \to Z) \to (X \to (Y \to Z))
\]

```scheme
(define (curry f)
  (lambda (v1)
    (lambda (v2)
      (f v1 v2)))))
```

```scheme
(define mk-is-sym (curry symbol=?))
(filter (mk-is-sym 'apple) l)
(filter (mk-is-sym 'banana) l)
(filter (mk-is-sym 'cherry) l)
```

! Composing Functions!

But we want *non*-symbols

\[
\text{compose} : (Y \to Z) (X \to Y) \to (X \to Z)
\]

```scheme
(define (compose f g)
  (lambda (x)
    (f (g x)))))
```

```scheme
(filter (compose not
           ((curry symbol=?)) 'apple)) l)
```

! Uncurrying Functions!

Sometimes it makes sense to **uncurry**:

\[
\text{uncurry} : (X \to (Y \to Z)) \to (X \ Y \to Z)
\]

```scheme
(define (uncurry f)
  (lambda (v1 v2)
    ((f v1) v2)))
```

```scheme
(define (map f l)
  (foldr (uncurry (compose (curry cons) f))
         empty l))
```

```scheme
(define (total-blue l)
  (foldr (uncurry (compose (curry +) color-blue))
         0 l))
```
Lambda in Math

; derivative : (num → num) → (num → num)
(define (derivative f)
  (lambda (x)
    (/ (- (f (+ x delta)))
       (f (- x delta)))
    (* 2 delta))))
(define delta 0.0001)

(define (square n) (* n n))
((derivative square) 10)

Produces roughly 20, because the derivative of $x^2$ is 2x

Lambda in Real Life

Graphical User Interfaces (GUIs) often use functions as values, including anonymous functions

Java equivalent: inner classes

GUI Library

make-text : string → gui-item

make-message : string → gui-item

draw-message : gui-item string → bool

make-button : string (event → bool) → gui-item

create-window : list-of-list-of-gui-item → bool

GUI Example

(define (greet what)
  (draw-message greet-msg
    (string-append
      what " ", "
      (text-contents name-field)))))

(define name-field
  (make-text "Name:"))

(define hi-button
  (make-button "Hello" (lambda (evt) (greet "Hi"))))

(define bye-button
  (make-button "Goodbye" (lambda (evt) (greet "Bye"))))

(define greet-msg
  (make-message "__________________________"))
GUI Example Improved

(define (mk-greet what)
  (lambda (evt)
    (draw-message greet-msg
      (string-append
        what "", "
        (text-contents name-field)))))

(define name-field
  (make-text "Name:"))
(define hi-button
  (make-button "Hello" (mk-greet "Hi")))
(define bye-button
  (make-button "Goodbye" (mk-greet "Bye")))
(define greet-msg
  (make-message "__________________________"))