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?)
```

should 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
(local [(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 in **Intermediate with Lambda**
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))))
```

The funny keyword `lambda` is an 80-year-old convention: the Greek letter \( \lambda \) means “function”
Using Lambda

In DrRacket:

```racket
> (lambda (x) (+ x 10))
(l 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 DrRacket gives up — it invents argument names and hides the body.
Using Lambda

In DrRacket:

```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

\[
\text{(define } f \text{ (a b)}
\]
\[
\text{(+ a b))}
\]

and

\[
\text{(define } f \text{ (lambda (a b)}
\]
\[
\text{(+ a b)))}
\]

?  

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 (inflating-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

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

How about functions that \textit{produce} functions?

Here’s one:

\[
; \text{make-adder} : \text{ num} \rightarrow (\text{ num} \rightarrow \text{ num})
\]

\[
\text{(define (make-adder } n) \\
(\lambda (m) (+ m n)))
\]

\[
(\text{map (make-adder } 10) ' (1 2 3))
\]

\[
(\text{map (make-adder } 11) ' (1 2 3))
\]
Using Functions that Produce Functions

Suppose that we need to filter different symbols:

\[
\begin{align*}
\text{(filter \ (lambda \ (a) \ (symbol=? \ a \ 'apple)) \ l)} \\
\text{(filter \ (lambda \ (a) \ (symbol=? \ a \ 'banana)) \ l)} \\
\text{(filter \ (lambda \ (a) \ (symbol=? \ a \ 'cherry)) \ l)}
\end{align*}
\]

Instead of repeating the long \textit{lambda} expression, we can abstract:

\[
\begin{align*}
; \text{mk-is-sym : sym -> (sym -> bool)} \\
\text{(define \ (mk-is-sym \ s)} \\
\text{\quad (lambda \ (a) \ (symbol=? \ s \ a)))}
\end{align*}
\]

\[
\begin{align*}
\text{(filter \ (mk-is-sym \ 'apple) \ l)} \\
\text{(filter \ (mk-is-sym \ 'banana) \ l)} \\
\text{(filter \ (mk-is-sym \ 'cherry) \ l)}
\end{align*}
\]

\text{mk-is-sym is a \textit{curried} version of symbol=?}
Currying Functions!

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

```scheme
; curry : (X Y -> Z) -> (X -> (Y -> Z))
(define (curry f)
  (lambda (v1)
    (lambda (v2)
      (f v1 v2)))))

(define mk-is-sym (curry symbol=?))

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

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

; curry : (X Y -> Z) -> (X -> (Y -> Z))
(define (curry f)
  (lambda (v1)
    (lambda (v2)
      (f v1 v2)))))

(filter ((curry symbol=? 'apple) l)
(filter ((curry symbol=? 'banana) l)
(filter ((curry symbol=? 'cherry) l)
! Composing Functions!

But we want *non*-symbols

```scheme
; compose (Y -> Z) (X -> Y) -> (X -> Z)
(define (compose f g)
  (lambda (x) (f (g x))))

(filter (compose
         not
         ((curry symbol=?) 'apple))
1)
```
! Uncurrying Functions!

Sometimes it makes sense to **uncurry**:

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

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

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

(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

Button click ⇒ update bottom text
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

```scheme
(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 ""))
```
(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 ""))