Part I
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:

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

would have to be

```scheme
(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 \texttt{a} and \texttt{b}
and produces \((\text{symbol}=? \ \texttt{a} \ \texttt{'apple})\)
... 
```

We can do this in \textbf{Intermediate with Lambda}
Lambda

An *anonymous function* value:

\[
\text{\texttt{(lambda (a b) (symbol=? a \texttt{\textquotesingle apple\textquotesingle})})}
\]

Using \texttt{lambda} the original example becomes

\[
\text{\texttt{(choose \texttt{\textquotesingle (apple banana)\textquotesingle}})
\text{\texttt{(cherry cherry)}})
\text{\texttt{(lambda (a b) (symbol=? a \texttt{\textquotesingle apple\textquotesingle})})}}
\]

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

In DrRacket:

```
> (lambda (x) (+ x 10))
(lamba (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:

\[
\textgreater \ ((\text{lambda } (x) (\text{+ } x \text{ 10})) \text{ 17})
\]

27

The function position of an \textit{application} (i.e., function call) is no longer always an identifier.

Some former syntax errors are now run-time errors:

\[
\textgreater \ (2 \ 3)
\]

\textit{procedure application: expected procedure, given 2}
Defining Functions

What’s the difference between

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

and

```
(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

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

How about functions that produce functions?

Here’s one:

\[
; \text{make-adder : num -> (num -> num)}
\]
\[
(\text{define (make-adder n)}
\]
\[
(\text{lambda (m) (+ m n))})
\]
\[
(\text{map (make-adder 10) ' (1 2 3))}
\]
\[
(\text{map (make-adder 11) ' (1 2 3))}
\]
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 \texttt{lambda} expression, we can abstract:

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

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

\text{mk-is-sym is a \textit{curried} version of symbol=？}
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)
```
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)))))

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

But we want non-symbols

; 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**:

; curry : (X -> (Y -> Z)) -> (X Y -> 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 Computing

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

text-contents : gui-item -> string

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 ""))
(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 "")))
Part II

( )

→

( )
Racket is a language for writing programs

DrRacket interprets Racket programs

DrRacket is a Racket program

HW 7: implement a mini DrRacket
Expr: (+ 2 3)

Evaluate

5
Mini DrRacket’s Input

For just numbers, plus, and minus:

; eval-expr : expr -> num

; An expr is either
;   - num
;   - (make-plus expr expr)
;   - (make-minus expr expr)

(check-expect (eval-expr 4) 4)
(check-expect (eval-expr (make-plus 2 3)) 5)
Mini DrRacket’s Input

For just numbers, plus, and minus:

; eval-expr : expr -> num

; An expr is either
; - num
; - (list '+ expr expr)
; - (list '- expr expr)

(check-expect (eval-expr '4) 4)
(check-expect (eval-expr '(+ 2 3)) 5)
Helpers for Mini DrRacket

Given "string.rkt" provides to-string, from-string, and many-from-string:

\[(\text{to-string} \ 5) \rightarrow "5"\]

\[(\text{from-string} \ "(+ \ 1 \ 2)") \rightarrow '(+ 1 2)\]

\[(\text{many-from-string} \ "1 \ (+ \ 2 \ 3) \ 8")\]
\[\rightarrow (\text{list} \ 1 \ '(+ \ 2 \ 3) \ 8)\]