Values and Names

Some Values:

• Numbers: 1, 17.8, 4/5
• Booleans: #true, #false
• Lists: '(), (cons 7 '())
• ...

• Function names: less-than-5, first-is-apple?
  given
  (define (less-than-5? n) ...)
  (define (first-is-apple? a b) ...)
Values and Names

Some Values:

• Numbers: 1, 17.8, 4/5
• Booleans: #true, #false
• Lists: '(), (cons 7 '())
• ...
• 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

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

we’d like to write

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

Naming Nothing

Can we avoid naming functions?

In other words, instead of writing

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

we’d like to write

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

We can do this in Intermediate with Lambda
Lambda

An *anonymous function* value:

$$\lambda (a\ b)\ (\text{symbol=}?\ a\ 'apple)$$

Using \texttt{lambda} the original example becomes

$$\text{choose}\ '\text{(apple banana)}$$
$$\quad\text{'}\text{(cherry cherry)}$$
$$\quad\lambda (a\ b)\ (\text{symbol=}?\ a\ 'apple))$$
Lambda

An *anonymous function* value:

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

Using `lambda` the original example becomes

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

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

In DrRacket:

```scheme
> (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:

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

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

In DrRacket:

\[
\begin{align*}
& \texttt{> } ((\text{lambda } (x) (+ x 10)) 17) \\
& \text{27}
\end{align*}
\]

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:

\[
\begin{align*}
& \texttt{> } (2 \ 3) \\
& \textit{procedure application: expected procedure, given 2}
\end{align*}
\]
Defining Functions

What’s the difference between

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

and

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

?
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 \texttt{filter, map, \textit{etc.}}:

\begin{verbatim}
(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))
\end{verbatim}
Functions that Produce Functions

We already have functions that take function arguments

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

How about functions that *produce* functions?
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 produce functions?

Here’s one:

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

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

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

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

Suppose that we need to filter different symbols:

```lisp
(filter (lambda (a) (symbol=? a 'apple)) l)
(filter (lambda (a) (symbol=? a 'banana)) l)
(filter (lambda (a) (symbol=? a 'cherry)) l)
```
Using Functions that Produce Functions

Suppose that we need to filter different symbols:

(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:

; 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)
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 \text{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)
```
! 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

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

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