Values and Names

Some Values:

- Numbers: \(1, 17.8, 4/5\)
- Booleans: \texttt{true, false}\n- Lists: \texttt{empty, \texttt{(cons 7 empty)}}
- ...

- Function \texttt{name}s: \texttt{less-than-5, first-is-apple?}
  
  given
  
  (\texttt{define (less-than-5? n) ...})
  
  (\texttt{define (first-is-apple? a b) ...})
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:

\[
(\text{local } [(\text{define } (\text{first-is-apple? } a \ b) \\
  \text{(symbol=? } a \ \text{'apple}))] \\
(\text{choose } '(\text{apple banana}) \\
  '(\text{cherry cherry}) \\
  \text{first-is-apple?}))
\]

would have to be

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

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

Can we avoid naming functions?

In other words, instead of writing

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

we’d like to write

```lisp
... function that takes \texttt{a} and \texttt{b} and produces (symbol=? a 'apple)
...```

Naming Nothing

Can we avoid naming functions?

In other words, instead of writing

\[
\begin{align*}
  \text{(local } & \text{ [(define (first-is-apple? a b)} \\
  & \text{(symbol=? a 'apple)]}) \\
  & \text{ ... first-is-apple? ...})
\end{align*}
\]

we’d like to write

\[
\begin{align*}
  \ldots \\
  \text{function that takes } a\text{ and } b \\
  \text{ and produces (symbol=? a 'apple)} \\
  \ldots
\end{align*}
\]

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)))
```
Lambda

An *anonymous function* value:

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

Using `lambda` the original example becomes

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

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

In DrRacket:

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

\[
(\text{lambda} \ (a1) \ ...)
\]

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:

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

27

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

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

and

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

?
Defining Functions

What’s the difference between

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

and

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

?  

Nothing — the first one is (now) a shorthand for the second
Lambda and Built-In Functions

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

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

(define (inflated-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

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

How about functions that \textit{produce} functions?
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?

Here’s one:

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

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

\[
; \text{lambda (m) (+ m n))}
\]

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

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

Suppose that we need to filter different symbols:

```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:

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

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

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

\[
\begin{align*}
\text{(filter } & \text{ (mk-is-sym } \ 'apple\text{) } l) \\
\text{(filter } & \text{ (mk-is-sym } \ 'banana\text{) } l) \\
\text{(filter } & \text{ (mk-is-sym } \ 'cherry\text{) } l)
\end{align*}
\]
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 lambda expression, we can abstract:

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

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

\[
\text{mk-is-sym is a } \text{curried} \text{ version of symbol=?}
\]
! 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

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

\[
(\text{define} (\text{compose } f \ g)
  (\lambda x. (f (g x))))
\]

\[
(\text{filter} (\text{compose}
  \not\quad
  (\text{curry symbol=}?) \ '\text{apple})
  1)
\]