Sample Language

Terms

$$e ::= x \mid \lambda x.e \mid e_1 e_2 \mid (e_1, e_2) \mid \text{fst } e \mid \text{snd } e \mid () \mid \text{fix } x.e \mid \text{if } e \text{ then } e \text{ else } e \mid \text{tt} \mid \text{ff} \mid \text{lcase } e \text{ of } (e, e) \mid e : e \mid \text{nil} \mid + \mid = \mid 0 \mid 1 \mid 2 \mid \ldots$$

Canonical Forms:  

$$c ::= \lambda x.e \mid () \mid (e_1, e_2) \mid e_1 : e_2 \mid \text{nil} \mid 0 \mid 1 \mid \ldots$$

Values:  

$$v ::= \lambda x.e \mid () \mid (v_1, v_2) \mid v_1 : v_2 \mid \text{nil} \mid 0 \mid 1 \mid \ldots$$
Example Programs

append ≜ fix r.λx.λy. 
   lcase x of (y, λh.λt.h : (r t y))

map ≜ fix r.λf.λx. 
   lcase x of (nil, λh.λt.(f h) : (r f t))

filter ≜ fix r.λx.λp. 
   lcase x of (nil, λh.λt.if p h then h : (r t p) else r t p)

reduce ≜ fix r.λf.λa.λx. 
   lcase x of (a, λh.λt.f h (r f a t))
Alternative Formulations

append ≜ \lambda x. \lambda y. \text{reduce } (\lambda h. \lambda t. h : t) y x

map ≜ \lambda f. \lambda x. \text{reduce } (\lambda h. \lambda t. (f h) : t) \text{ nil } x

filter ≜ \lambda x. \lambda p. \text{reduce } (\lambda h. \lambda t. \text{ if } p h \text{ then } h : t \text{ else } t) \text{ nil } x

reduce' ≜ \text{fix } r. \lambda f. \lambda a. \lambda x.
\text{lcase } x \text{ of } (a, \lambda h. \lambda t. r f (f h a) t)
More Example Programs

\[
\text{zip} \triangleq \text{fix } r. \lambda x. \lambda y. \\
\quad \text{lcase } x \text{ of } (\text{nil}, \lambda h_1. \lambda t_1. \\
\quad \quad \text{lcase } y \text{ of } (\text{nil}, \lambda h_2. \lambda t_2. (h_1, h_2): (r t_1 t_2)))
\]

\[
\text{eqlist} \triangleq \text{fix } r. \lambda x. \lambda y. \\
\quad \text{lcase } x \text{ of } (\text{lcase } y \text{ of } (\text{tt}, \lambda h. \lambda t. \text{ff}), \lambda h_1. \lambda t_1. \\
\quad \quad \text{lcase } y \text{ of } (\text{ff}, \lambda h_2. \lambda t_2. \\
\quad \quad \quad \quad \text{if } h_1 = h_2 \text{ then } r t_1 t_2 \text{ else } \text{ff}))
\]

\[
\text{search} \triangleq \text{fix } r. \lambda x. \lambda p. \lambda f. \lambda i. \\
\quad \text{lcase } x \text{ of } (i, \lambda h. \lambda t. \text{if } p h \text{ then } f h \text{ else } r t p f i)
\]
Call-by-value Evaluation (Scheme, ML)

\[
\begin{align*}
\lambda x.e & \leftrightarrow \lambda x.e \quad \text{ev}_\text{lam} \\

e_1 & \leftrightarrow \lambda x.e \quad e_2 \leftrightarrow v_1 \quad e[x := v_1] & \leftrightarrow v_2 \quad \text{ev}_\text{app} \\

e_1 e_2 & \leftrightarrow v_2 \quad \text{ev}_\text{app} \\
\frac{e_1 \leftrightarrow v_1}{(e_1, e_2) \leftrightarrow (v_1, v_2)} \quad \frac{e_2 \leftrightarrow v_2}{(e_1, e_2) \leftrightarrow (v_1, v_2)} \quad \text{ev}_\text{pair} \\
\frac{e \leftrightarrow (v_1, v_2)}{\text{fst} e \leftrightarrow v_1} \quad \frac{e \leftrightarrow (v_1, v_2)}{\text{snd} e \leftrightarrow v_2} \quad \text{ev}_\text{fst} \quad \text{ev}_\text{snd} \\
\frac{e[x := \text{fix } x.e]}{\text{fix } x.e \leftrightarrow v} \quad \text{ev}_\text{fix}
\end{align*}
\]
Call-by-value contd.

\[
\begin{align*}
\text{tt} & \mapsto \text{tt} \quad \text{ev}_\text{tt} \\
\text{ff} & \mapsto \text{ff} \quad \text{ev}_\text{ff} \\
\text{e}_1 & \mapsto \text{tt} \quad \text{e}_2 \mapsto \text{v} \quad \text{ev}_\text{if}_\text{tt} \\
\text{if } \text{e}_1 \text{ then } \text{e}_2 \text{ else } \text{e}_3 & \mapsto \text{v} \quad \text{ev}_\text{if}_\text{tt} \\
\text{e}_1 & \mapsto \text{ff} \quad \text{e}_3 \mapsto \text{v} \quad \text{ev}_\text{if}_\text{ff} \\
\text{null} & \mapsto \text{null} \quad \text{ev}_\text{nil} \\
\text{e}_1 & \mapsto \text{null} \quad \text{e}_2 \mapsto \text{v} \quad \text{ev}_\text{cons} \\
\text{cons} \quad \text{e}_1 & \mapsto \text{null} \quad \text{e}_2 \mapsto \text{v} \quad \text{ev}_\text{lcase}_\text{nil} \\
\text{lcase } \text{e}_1 \text{ of } (\text{e}_2, \text{e}_3) & \mapsto \text{v} \quad \text{ev}_\text{lcase}_\text{cons} \\
\text{natural numbers} \\
\text{evaluate as expected}
\end{align*}
\]
\[ \textit{Call-by-value Example} \]

\[ \text{app} \triangleq \lambda x. \lambda y. \text{lcase } x \text{ of } (y, \lambda h. \lambda t. h : (\text{append } t \ y)) \]

\[
\begin{array}{c}
\text{app} \leftrightarrow \text{app} \quad \text{ev}_\text{lam} \\
\text{append} \leftrightarrow \text{app} \quad \text{ev}_\text{fix} \\
\text{append} \ (1: \text{nil}) \leftrightarrow \lambda y. \text{lcase } 1: \text{nil} \text{ of } (y, \lambda h. \lambda t. h : (\text{append } t \ y)) \quad \text{ev}_\text{app} \\
\text{append} \ (1: \text{nil}) \ (2:3: \text{nil}) \leftrightarrow 1:2:3: \text{nil}
\end{array}
\]

\[ D_1 = \]

\[
\lambda y. \text{lcase } 1: \text{nil} \text{ of } (y, \lambda h. \lambda t. h : (\text{append } t \ y)) \leftrightarrow \lambda y. \text{lcase } 1: \text{nil} \text{ of } (y, \lambda h. \lambda t. h : (\text{append } t \ y)) \quad \text{ev}_\text{lam}
\]

\[ D_2 = \]

\[
\begin{array}{c}
1 \leftrightarrow 1 \quad \text{ev}_1 \\
3 \leftrightarrow 3 \quad \text{ev}_3 \\
\text{nil} \leftrightarrow \text{nil} \quad \text{ev}_\text{nil} \\
2:3: \text{nil} \leftrightarrow 2:3: \text{nil} \quad \text{ev}_\text{cons}
\end{array}
\]

\[ D_3 = \]

\[
\begin{array}{c}
1 \leftrightarrow 1 \quad \text{ev}_1 \\
3 \leftrightarrow 3 \quad \text{ev}_3 \\
\text{nil} \leftrightarrow \text{nil} \quad \text{ev}_\text{nil} \\
2:3: \text{nil} \leftrightarrow 2:3: \text{nil} \quad \text{ev}_\text{cons}
\end{array}
\]
Call-by-value Example contd.

\[ \mathcal{D}_3 = \]

\[
\begin{array}{c}
D_3 = \\
\frac{1 \leftarrow 1 \text{ ev-1}}{\text{nil} \leftarrow \text{nil} \text{ ev-nil}}
\end{array}
\]

\[
\begin{array}{c}
\frac{1 : \text{nil} \leftarrow 1 : \text{nil} \text{ ev-cons}}{\frac{2 \text{ ev-1}}{\text{nil} \leftarrow \text{nil} \text{ ev-nil}}}
\end{array}
\]

\[
\begin{array}{c}
\frac{\frac{\left(\lambda h.\lambda t. h : (\text{append } t (2:3:\text{nil}))\right) \text{ nil} \leftarrow 1:2:3:\text{nil}}{\lambda \text{ cons} \text{ 1:nil of (2:3:nil, } \lambda h.\lambda t. h : (\text{append } t (2:3:\text{nil}))\text{) } \leftarrow 1:2:3:\text{nil}}}{\text{lcase 1:nil of (2:3:nil, } \lambda h.\lambda t. h : (\text{append } t (2:3:\text{nil}))\text{) } \leftarrow 1:2:3:\text{nil}}
\end{array}
\]

\[
\begin{array}{c}
\frac{D_4}{\frac{\lambda h.\lambda t. h : (\text{append } t (2:3:\text{nil})) \leftarrow \lambda h.\lambda t. h : (\text{append } t (2:3:\text{nil})) \text{ ev-lam}}{1 \leftarrow 1 \text{ ev-1}}}
\end{array}
\]

\[
\begin{array}{c}
D_4 = \\
\frac{\left(\lambda h.\lambda t. h : (\text{append } t (2:3:\text{nil}))\right) 1 \leftarrow \lambda t.1 : (\text{append } t (2:3:\text{nil}))}{\text{app} \leftarrow \text{app} \text{ ev-lam}}
\end{array}
\]

\[
\begin{array}{c}
D_5 = \\
\frac{\frac{\text{app} \leftarrow \text{app} \text{ ev-lam}}{\frac{\text{append} \leftarrow \text{app} \text{ ev-fix}}{\frac{\text{nil} \leftarrow \text{nil} \text{ ev-nil}}{D_7}}}}{D_7 \text{ ev-app}}
\end{array}
\]

\[
\begin{array}{c}
\frac{1 \leftarrow 1 \text{ ev-1}}{\text{append nil} \leftarrow \lambda y. \text{lcase nil of (y, } \lambda h.\lambda t. h : (\text{append } t y)\text{)}\\
\text{append nil (2:3:nil) } \leftarrow 2:3:\text{nil} \text{ ev-cons}}
\end{array}
\]

\[
\begin{array}{c}
\frac{\text{append nil (2:3:nil) } \leftarrow 2:3:\text{nil} \text{ ev-cons}}{1 : (\text{append nil (2:3:nil)) } \leftarrow 1:2:3:\text{nil}}
\end{array}
\]
Call-by-value Example contd.

\[ D_7 = \]
\[ \lambda y.\text{lcase nil of } (y, \lambda h.\lambda t.h:(\text{append } t y)) \leftrightarrow \lambda y.\text{lcase nil of } (y, \lambda h.\lambda t.h:(\text{append } t y)) \]

\[ D_8 = \]
\[ 2 \leftrightarrow 2 \quad ev_2 \quad 3 \leftrightarrow 3 \quad ev_3 \quad \text{nil} \leftrightarrow \text{nil} \quad ev_{\text{nil}} \quad 3:\text{nil} \leftrightarrow 3:\text{nil} \quad ev_{\text{cons}} \quad \text{ev}_{\text{cons}} \]
\[ 2:3:\text{nil} \leftrightarrow 2:3:\text{nil} \quad ev_{\text{cons}} \]

\[ D_9 = \]
\[ \text{nil} \leftrightarrow \text{nil} \quad ev_{\text{nil}} \quad 2 \leftrightarrow 2 \quad ev_2 \quad 3 \leftrightarrow 3 \quad ev_3 \quad \text{nil} \leftrightarrow \text{nil} \quad ev_{\text{nil}} \quad \text{ev}_{\text{cons}} \quad \text{ev}_{\text{cons}} \]
\[ \text{ev}_{\text{cons}} \quad \text{ev}_{\text{cons}} \quad \text{ev}_{\text{lc-nil}} \]

\[ \text{lcase nil of } (2:3:\text{nil}, \lambda h.\lambda t.h:(\text{append } t 2:3:\text{nil})) \leftrightarrow 2:3:\text{nil} \]
**Eager Evaluation**

Evaluates each argument once.

Sometimes call-by-value is too eager:

\[
\text{search} \triangleq \text{fix } f. \lambda x. \lambda p. \lambda g. \lambda i. \\
\text{lcase } x \text{ of } (i, \lambda h. \lambda t. \text{if } p h \text{ then } g h \text{ else } f t p g i)
\]

failure expression, \(i\), is always evaluated in recursive call. This can be fixed by “thunking”:

\[
\text{search} \triangleq \text{fix } f. \lambda x. \lambda p. \lambda g. \lambda i. \\
\text{lcase } x \text{ of } (i(), \lambda h. \lambda t. \text{if } p h \text{ then } g h \text{ else } f t p g i)
\]

\(i\) must now be a function which takes a dummy, \((\)\), argument.

Eager evaluation can also cause “unnecessary” non-termination.
Call-by-name Evaluation

\[ \lambda x.e \leftarrow \lambda x.e \quad \text{ev\_lam} \]

\[ e_1 \leftarrow \lambda x.e \quad e[x := e_2] \leftarrow v \quad \text{ev\_app} \]

\[ (e_1, e_2) \leftarrow (e_1, e_2) \quad \text{ev\_pair} \]

\[ (\_ ,\_ ) \leftarrow (\_ ,\_ ) \quad \text{ev\_unit} \]

\[ \begin{array}{c}
  e \leftarrow (v_1, v_2) \\
  \text{ev\_fst}
\end{array} \]

\[ \begin{array}{c}
  \text{fst } e \leftarrow v_1 \\
  \text{ev\_fst}
\end{array} \]

\[ \begin{array}{c}
  e \leftarrow (v_1, v_2) \\
  \text{ev\_snd}
\end{array} \]

\[ \begin{array}{c}
  \text{snd } e \leftarrow v_2 \\
  \text{ev\_snd}
\end{array} \]

\[ e[x := \text{fix } x.e] \leftarrow v \quad \text{ev\_fix} \]

\[ \text{fix } x.e \leftarrow v \quad \text{ev\_fix} \]
**Call-by-name contd.**

\[ tt \leftrightarrow tt \text{ ev}_{tt} \]
\[ ff \leftrightarrow ff \text{ ev}_{ff} \]
\[ e_1 \leftrightarrow tt \quad e_2 \leftrightarrow v \quad \text{ev}_\text{if}_{tt} \]
\[ \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \leftrightarrow v \]
\[ e_1 \leftrightarrow ff \quad e_3 \leftrightarrow v \quad \text{ev}_\text{if}_{ff} \]
\[ \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \leftrightarrow v \]
\[ \text{nil} \leftrightarrow \text{nil} \quad \text{ev}_\text{nil} \]
\[ e_1 : e_2 \leftrightarrow e_1 : e_2 \quad \text{ev}_\text{cons} \]
\[ e_1 \leftrightarrow \text{nil} \quad e_2 \leftrightarrow v \quad \text{ev}_\text{lc}_{\text{nil}} \]
\[ \text{lcase } e_1 \text{ of } (e_2, e_3) \leftrightarrow v \quad \text{ev}_\text{lc}_{\text{nil}} \]
\[ e_1 \leftrightarrow v_1 : v_2 \quad e_3 v_1 v_2 \leftrightarrow v \]
\[ \text{lcase } e_1 \text{ of } (e_2, e_3) \leftrightarrow v \]

natural numbers

evaluate as expected
Lazy Evaluation

Call-by-name only evaluates an expression when needed.

\[
eqlist (1:2:\cdots:100:\nil) (0:1:\cdots:100:\nil)
\]

is much faster in call-by-name than call-by-value.

Avoids non-termination from unused arguments, but very inefficient:

\[
(\lambda x. x + x + x + x) \text{long\_computation}
\]

computes \text{long\_computation} four times.

Actual implementations (e.g. Haskell) use call-by-need, or lazy, evaluation which waits to evaluate until necessary, but caches the value a variable is bound to so that further uses of the variable do not have to recompute its value.
Infinite Structures

Laziness allows for directly handling infinite structures:

\[
\text{zeros} \triangleq \text{fix } l. 0 : l \\
\text{nats} \triangleq \text{fix } l. 0 : (\text{map } (\lambda x. x + 1) l) \\
\text{enum} \triangleq \lambda l. \text{zip nats } l
\]

Have to be careful when applying functions to infinite lists.

- append, map, zip behave as expected on infinite lists.

- \text{reduce'} always diverges on an infinite lists.

- filter, reduce, search, eqlist might diverge on an infinite list.
Some Haskell Syntax

fix is implicit in all Haskell definitions.

let zeros = 0:zeros
let nats = 0:(map (\x -> x + 1) nats)
let enum = \l -> zip nats l

Some alternatives for nats:

let nats = 0:(map (+ 1) nats)
let nats = [0 ..]

Notes:
(+ 1) ≡ (\x -> x + 1)
[0 .. 5] ≡ 0:1:2:3:4:5:[] ≡ [0, 1, 2, 3, 4, 5]
Haskell has pattern matching. Note that this forces evaluation of argument to the shape of pattern:

let pair_ones = (1,1):pair_ones
\((x,y) \rightarrow ()\) pair_ones!!1000000

Takes a while to evaluate even though argument isn’t used.

Some useful Haskell built-in functions:
[1,2,3,4,5]!!3 \equiv 4.  (Note !! is an infix function.)
take 3 [1,2,3,4,5] \equiv [1,2,3]
Lazy Evaluation

Consider expression:

```haskell
let sums = (1,1):(map (\(i,n) -> (i+1, n+i+1)) sums)
```

Consider an explicitly delayed version:

```haskell
let ones = 1:ones
let sums' =
    (1,1):(map (\(i,n) -> (i+(ones!!1000000), n+i+1)) sums')
```

First evaluation of `sums'!!1000` takes a while, but subsequent calls need not reevaluate the list.
Homework 2

1. [20 points]
   (a) What does:
       \[ \text{append (1:nil) (2:3:nil)} \]
   evaluate to using call-by-name?
   
   (b) Write a complete call-by-name evaluation derivation for:
       \[ \text{append (1:nil) (2:3:nil)} \mapsto v \]
   where \( v \) is your answer to part a.

2. [20 points]
   Write an expression for an infinite list representing the graph of the factorial function:
   \[ e = (1, 1) : (2, 2) : (3, 6) : (4, 24) : (5, 120) : (6, 720) : (7, 5040) : \ldots \]

3. [30 points]
   Write an expression for an infinite list representing the graph of the Fibonacci function:
   \[ e = (1, 1) : (2, 1) : (3, 2) : (4, 3) : (5, 5) : (6, 8) : (7, 13) : \ldots \]