Alternative Evaluation Orders:
Delay and laziness

When are expressions evaluated?

Bonus: memoization

Eager evaluation: arguments first
call-by-value semantics

When do arguments/subexpressions evaluate (ML, Racket)?
- Function arguments: once, before calling function
- Conditional branches: only one branch, after checking condition

```
fun iffy x y z =
  if x then y else z

fun facty n =
  iffy (n = 0)
    1
    (n * (facty (n - 1)))
```

Delay and Laziness

Delayed evaluation with thunks
explicit emulation of lexically-scoped call-by-name semantics

Thunk fn () => e
  - n. a zero-argument function used to delay evaluation
  - v. to create a thunk from an expression:
    "thunk the expression"

No new language features.

```
fun if_by_name x y z =
  if x () then y () else z ()

fun fact n =
  if_by_name (fn () => n = 0)
    (fn () => 1)
    (fn () => n * (fact (n - 1)))
```

Eager... not eager...

Thunk: evaluate when value needed
explicit emulation of lexically-scoped call-by-name semantics

```
fun f1 th =
  if ... then 7 else ... th() ...

fun f2 th =
  if ... then 7 else th() + th()

fun f3 th =
  let val v = th ()
  in if ... then 7 else v + v end

fun f4 th =
  if ... then 7 else
  let val v = th () in v + v end
```

See code examples

- # evaluations?
- Faster?
- Slower?
- Side effects?
Lazy evaluation: first time value is needed

*call-by-need semantics*

Argument/subexpression **evaluated zero or one times**, no earlier than first time result is actually needed.

**Result reused** (not recomputed) if needed again **anywhere**.

Benefits of delayed evaluation, with minimized costs.

Explicit laziness with **promises**:
- Promise.delay (fn () => x * f x)
- Promise.force p

Promises: explicit laziness

(a.k.a. suspensions)

**signature** PROMISE =

```haskell
sig

  (* Type of promises for 'a. *)
  type 'a t

  (* Take a thunk for an 'a and make a promise to produce an 'a. *)
  val delay : (unit -> 'a) -> 'a t

  (* If promise not yet forced, call thunk and save. Return saved thunk result. *)
  val force : 'a t -> 'a

end
```

Promises: delay and force

(a.k.a. suspensions)

**structure** Promise => PROMISE =

```haskell
struct
datatype 'a promise = Thunk of unit -> 'a
  | Value of 'a

type 'a t = 'a promise ref

fun delay thunk = ref (Thunk thunk)

fun force p =
  case !p of
  Value v => v
  | Thunk th =>
    let val v = th ()
    in val_ = p := Value v
    end
```

Stream: infinite sequence of values

- Cannot make all the elements now.
- Make one when asked, delay making the rest.

- Interface/idiom for division of labor:
  - Stream producer
  - Stream consumer
  - Interleave production / consumption in time, but *not in code*.

- Examples:
  - UI events
  - UNIX pipes: git diff delay.sml | grep "thunk"
  - Sequential logic circuit updates (CS 240)
Streams in ML: false start

Let a **stream** be a thunk that, *when called*, returns a pair of
- the next element; and
- the rest of the stream.

\[
\text{fn} () \Rightarrow (\text{next\_element}, \text{next\_thunk})
\]

Given stream \(s\), get elements:
- First: \(\text{let val } (v1,s1) = s ()\)
- Second: \(\text{val } (v2,s2) = s1 ()\)
- Third: \(\text{val } (v3,s3) = s2 ()\)

Delay and Laziness

Streams in ML: recursive types

Single-constructor datatype allows recursive type:

\[
\begin{align*}
\text{datatype } \ 'a \ \text{scons} & = \ Scons \ of \ 'a \ast (\text{unit} \rightarrow \ 'a \ \text{scons}) \\
\text{type } \ 'a \ \text{stream} & = \ \text{unit} \rightarrow 'a \ \text{scons}
\end{align*}
\]

Given a stream \(s\):
- First: \(\text{let val Scons(v1,s1) = s ()}\)
- Second: \(\text{val Scons(v2,s2) = s1 ()}\)
- Third: \(\text{val Scons(v3,s3) = s2 ()}\)

Stream consumers

Find index of first element in stream for which \(f\) returns true.

\[
\begin{align*}
\text{fun firstindex } f \ \text{stream } = \\
&\text{let fun consume stream acc } = \\
&\text{let val Scons (v,s) = stream () } \\
&\text{in } \\
&\text{if } f \ v \\
&\text{then acc } \\
&\text{else consume s (acc } + 1) \\
&\text{end } \\
&\text{in consume stream 0 end } \\
\end{align*}
\]

: ('a \rightarrow \text{bool}) \rightarrow 'a \text{ stream} \rightarrow \text{int}

Delay and Laziness

Stream producers

\[
\begin{align*}
\text{fun ones } () & = \text{Scons (1,ones)} \\
\text{val rec ones } & = \text{fn } () \Rightarrow \text{Scons (1,ones)}
\end{align*}
\]

Create next thunk via *delayed recursion!*
- Return a thunk that, when called, calls the outer function recursively.

\[
\begin{align*}
\text{val nats } & = \\
&\text{let fun f x } = \text{Scons (x, fn } () \Rightarrow f (x + 1) ) \\
&\text{in fn } () \Rightarrow f \ 0 \ \text{end}
\end{align*}
\]

\[
\begin{align*}
\text{val powers2 } & = \\
&\text{let fun f x } = \text{Scons (x, fn } () \Rightarrow f (x \ast 2) ) \\
&\text{in fn } () \Rightarrow f \ 1 \ \text{end}
\end{align*}
\]
Getting it wrong

Tries to use a variable before it is defined.

```plaintext
val ones_bad = Scons (1, ones_bad)
```

Would call `ones_worse` recursively **immediately** (infinitely).

Does not type-check.

```plaintext
fun ones_worse () = Scons (1, ones_worse ())
```

**Correct:** thunk that returns `Scons` of value and stream (thunk).

```plaintext
fun ones () = Scons (1, ones)
val rec ones = fn () => Scons (1, ones)
```

---

Bonus: Lazy by default?

**ML:**
- Eager evaluation. Explicitly emulate laziness when needed (promises).
- Immutable data, bindings. Explicit mutable cells when needed (refs).
- Side effects anywhere.

**Pros:**
- avoid unnecessary work, build elegant infinite data structures.

**Cons:**
- difficult to control/predict evaluation order:
  - Space usage: when will environments become unreachable?
  - Side-effect ordering: when will effects execute?

**Haskell:**
- canonical real-world example

**Pros:**
- Non-strict evaluation, except pattern-matching. Explicit strictness when needed.
- Usually implemented as lazy evaluation.
- Immutable everything. Emulate mutation/state when needed.
- Side effects banned/restricted/emulated.

---

Bonus: Memoization

see memo.sml

Not delayed evaluation, but...
- Promises (call-by-need) are memoized thunks (call-by-name), though memoization is more general (multiple arguments).
- Can use an indirect recursive style similar to streams (without delay)
  - Actually fixpoint...

**Basic idea:**
- Save results of expensive pure computations in mutable cache.
- Reuse earlier computed results instead of recomputing.
- Even for recursive calls.

**Benefits:**
- Save time when recomputing.
- Can reduce exponential recursion costs to linear (and amortized by repeated calls with same arguments).

**See also:** dynamic programming (CS 231)