Lazy Evaluation
Evaluating Functions

Consider:

\[
\text{add } x \ y = x + y; \quad \text{square } x = x \times x \\
\text{add } (\text{square } 2) \ (\text{add } 2 \ 3)
\]
add x y = x + y;  square x = x * x

add (square 2) (add 2 3)

-- apply square:
add (2 * 2) (add 2 3)
-- apply *:
add 4 (add 2 3)
-- apply inner add :
add 4 (2 + 3)
-- apply + :
add 4 5
-- apply add
4+5
-- apply +
9

-- apply add:
(square 2) + (add 2 3)
-- apply inner add :
(square 2) + (2 + 3)
-- apply inner + :
(square 2) + 5
-- apply square:
(2 * 2) + 5
-- apply *
4+5
-- apply +
9
There are many possible orders to evaluate a function

```
head (1:(reverse [2,3,4,5]))  head (1:(reverse [2,3,4,5]))
-- apply reverse                   -- apply head
-- ... many steps omitted here      1
head (1 : [5,4,3,2])
-- apply head

1
```

In Haskell, any two different ways of evaluating the same expression will produce the same final value if they terminate.
There are many possible orders to evaluate a function

head (1:(reverse [2,3,4,5]))

-- apply reverse

-- ... many steps omitted here

head (1 : [5,4,3,2])

-- apply head

1

In Haskell, any two different ways of evaluating the same expression will produce the same final value if they terminate.

The same isn’t true of imperative programming languages.
Two styles:

**Strict Evaluation**

In C/C++/Java a call such as:
\[ f(btho(whom\_ever),\text{increment\_count()}) \]
we know that `whom\_ever` will be beaten, then the global variable will be updated. This is leftmost, innermost evaluation.

It would be very confusing if we couldn’t count on those being evaluated!

Eager, call-by-value evaluation.

**Lazy Evaluation**

In Haskell an expression:
\[ \text{fst} \ (5, \ (\text{factorial} \ 10000)) \]
the second, expensive computation won’t even be performed. Outermost evaluation, only when needed.

Some optimizations (via memoization):
\[ \text{foo} \ (\text{factorial} \ 10) \ \text{“meh”} \ (\text{factorial} \ 10) \]

Call-by-name evaluation.
Two styles:

**Strict Evaluation**

In C/C++/Java a call such as:

\[ f(btho(whom\_ever),increment\_count()) \]

we know that `whom\_ever` will be beaten, then the global variable will be updated. This is leftmost, innermost evaluation.

It would be very confusing if we couldn’t count on those being evaluated!

Eager, call-by-value evaluation.

---

In C/C++/Java have you ever thought about what happens when `first_cond()` returns true in:

```java
if (first_cond() || second_cond()) {
    ...
}
```

or:

```java
if (!(first_cond()) && second_cond()) {
    ...
}
```

Compare to:

\[ \text{True} \land \text{True} = \text{True} \]

\[ _ \land _ = \text{False} \]
Haskell’s call-by-name evaluation

What about lambda expressions:

*Main> (\x -> 3 * 2 + x) (4+2)
12

It looks like the following are possible evaluations:

(\x -> 3 * 2 + x) (4+2)
3 * 2 + (4+2)
3 * 2 + 6
6 + 6
12

(\x -> 3 * 2 + x) (4+2)
\x -> 3 * 2 + x
6
6 + 6
12

Haskell will not simplify inside lambda expression (i.e., reducing as on the example on the right) because the lambda’s are treated as black boxes. A lambda’s only simplification operation is to apply it.
Strict Evaluation

Eager, call-by-value evaluation.

- May fail to halt working on some computation that is later discarded.
- May use fewer evaluations when it does halt.

Lazy Evaluation

Call-by-name evaluation.

- Can halt even on infinite structures. But care is needed
  *cf.* 3 `elem` [2,4..]
- May need more evaluations, but saving intermediate results helps mitigate these costs.

Haskell has the $!$ operator to force strict evaluation; this can be helpful to save stack space.
Examples:

```
let g x y = y

g 2 4
4

g (2 `div` 0) 10
10

(g $! (2 `div` 0)) 10
*** Exception: divide by zero
```

Note that these three uses force strictness in different ways:

```
(g $! x) y
(g x) $! y
(g $! x) $! y
```

### Strict Evaluation

### Lazy Evaluation

**Call-by-name evaluation.**

- Can halt even on infinite structures. But care is needed _cf._ 3 `elem` [2,4..]
- May need more evaluations, but saving intermediate results helps mitigate these costs.

Haskell has the $! operator to force strict evaluation; this can be helpful to save stack space.
Strict Evaluation

Eager, call-by-value evaluation.

- May fail to halt working on some values. But care is needed more evaluations, when it does halt.

Lazy Evaluation

Call-by-name evaluation.

Haskell has the $!$ operator to force strict evaluation; this can be helpful to save stack space.