Postmortem: Ex01

- Submissions: 104 out of 130 : 80%
- Fully Correct: 74 out of 104 : 71%
- Failed something: 30 out of 104 : 29%
  - Failed Tests: 27 out of 30 : 90%
  - Did not Compile: *3* out of 30 : 10%

Lazy Evaluation

Haskell uses *Lazy (non-Strict) Evaluation*

- Expressions are only evaluated *when* their value is needed
- In particular, argument expressions are not evaluated before a function is applied
- We find this approach allows us to write sensible programs not possible if strict-evaluation is used.
- However, it comes at a price . . .

Example: `isOdd`

- We define a function checking for 'oddness' as follows:
  `` isOdd n = n `mod` 2 == 1 ``
- Consider the call `isOdd (1+2)`
- A strict (non-lazy) evaluation would be as follows:

```
isOdd (1+2)
 = isOdd 3              evaluate arg first,
 = 3 `mod` 2 == 1       then apply function
 = 1 == 1
 = True
```

- A non-strict (lazy) evaluation would be as follows:

```
isOdd (1+2)
 = (1+2) `mod` 2 == 1   apply function first,
 = 3 `mod` 2 == 1       mod needs args evaluated
 = 1 == 1
 = True
```

`1+2` is only evaluated *when* `mod` needs its value to proceed.

`len` and `down`

- We have a length function `len`:

```
len xs = if null xs then 0 else 1 + len (tail xs)
```

- We have a function `down` that generates a list, counting down from its numeric argument:

```
down n = if n <= 0 then [] else n : (down (n-1))
```

  For example, `down 3 = [3,2,1]`
- We shall consider pattern matching versions shortly

Strict evaluation of `len (down 1)`

```
len (down 1)
= len (if 1 <= 0 then [] else 1 : (down (1-1))
= len (1 : (down (1-1))
= len (1 : (down 0))
= len (1 : (if 0 <= 0 then [] else 0 : (down (0-1))))
= len (1 : [])
= if null (1 : []) then 0 else 1 + len (tail (1 : []))
= 1 + len (tail (1 : []))
= 1 + len []
= 1 + len (if null [] then 0 else 1 + len (tail []))
= 1 + 0
= 1
```

We have 11 steps

Lazy evaluation of `len (down 1)` (part 1)

```
len (down 1)
= if null xs₁ then 0 else 1 + len (tail xs₁)
  where xs₁ = down 1
= if null xs₁ then 0 else 1 + len (tail xs₁)
  where xs₁ = if 1 <= 0 then [] else 1 : (down (1-1))
= if null xs₁ then 0 else 1 + len (tail xs₁)
  where xs₁ =  1 : (down (1-1))
= 1 + len (tail xs₁) where xs₁ =  1 : (down (1-1))
= 1 + ( if null xs₂ then 0 else 1 + len (tail xs₂)
        where xs₂ = tail xs₁ )
  where xs₁ =  1 : (down (1-1))
```

(continued overleaf)

Lazy evaluation of `len (down 1)` (part 2)

```
1 + ( if null xs₂ then 0 else 1 + len (tail xs₂)
        where xs₂ = tail xs₁ )
  where xs₁ =  1 : (down (1-1))
= 1 + ( if null xs₂ then 0 else 1 + len (tail xs₂)
        where xs₂ =  tail (1 : (down (1-1))))
= 1 + ( if null xs₂ then 0 else 1 + len (tail xs₂)
        where xs₂ =  down (1-1))
= 1 + ( if null xs₂ then 0 else 1 + len (tail xs₂)
        where xs₂ = (if (1-1) <= 0
                     then [] else (1-1) : (down ((1-1)-1))) )
= 1 + ( if null xs₂ then 0 else 1 + len (tail xs₂)
        where xs₂ = (if 0 <= 0
                     then [] else (1-1): (down ((1-1)-1))) )
= 1 + ( if null xs₂ then 0 else 1 + len (tail xs₂)
        where xs₂ = [] )
= 1 + 0
= 1   12 steps, each more expensive!
```

Why the $\texttt{xs}_1$ `= ...` ?

- Consider the first step:

  ```
  len (down 1)
  = if null xs₁ then 0 else 1 + len (tail xs₁)
    where xs₁ = down 1
  ```

- We don't evaluate `down 1` — we bind it to formal parameter $\texttt{xs}_1$
- Parameter `xs` occurs twice, but we don't copy:

  `...down 1 ...down 1 ...`

  Instead we share the reference, indicated by the `where` clause:

  `...xs₁ ...xs₁ ...where xs₁ = down 1`

- Function `len` is recursive, so we get different instances of `xs` which we label as $\texttt{xs}_1$, $\texttt{xs}_2$, ...
- The grouping of an (unevaluated) expression (`down 1`) with a binding ($\texttt{xs}_1$ `= down 1`) is called either a *"closure"*, or a *"thunk"*.
- Building thunks is a *necessary* overhead for implementing lazy evaluation.

Lazy Evaluation: the costs

- Lazy evaluation has an overhead: building thunks
- Memory consumption per reduction step is typically slightly higher
- In our examples so far:
  `isOdd (1+2)`
  `len (down 1)` we needed to evaluate almost everything
- So far we have observed no advantage to lazy evaluation . . .

Advantages of Laziness (I)

- Imagine we have a function definition as follows:

```
myfun carg struct1 struct2
 = if f carg
   then g struct1
   else h struct2
```

  where `f`, `g` and `h` are internal functions
- Consider the following call:

```
myfun val s1Expr s2Expr
```

  where both `s1Expr` and `s2Expr` are very expensive to evaluate.
- With strict evaluation we would have to compute both before applying `myfun`
- With lazy evaluation we evaluate `f val`, and then only evaluate one of either `s1Expr` or `s2Expr`, and then, only if `g` or `h` requires its value.

Laziness and Pattern Matching

- Consider a pattern matching version of `len`

```
len [] = 0
len (x:xs) = 1 + len xs
```

- How is call `len aListExpression` evaluated ?
- In order to pattern match we need to know if `aListExpression` is empty, or a cons-node.
- We evaluate `aListExpression`, but only to the point were we know this difference
  If it is not null, we do not evaluate the head element, or the tail list.
- e.g. if `aListExpression = map f (1:2:3:[])`, where

```
map f [] = []
map f (x:xs) = f x : map f xs
```

  then we only evaluate `map f` as far as

```
f 1 :  map f (2:3:[])
```

Advantages of Laziness (II)

- Prelude function `take n xs` returns the first `n` elements of `xs`

```
take 0 _ = []
take _ [] = []
take n (x:xs) = x : (take (n-1) xs)
```

- Function `from n` generates an *infinite* ascending list starting with `n`.

```
from n = n : (from (n+1))
```

- Evaluating `from n` will fail to terminate for any `n`.
- Evaluation of `take 2 (from 0)` depends on the evaluation method.

Strict Evaluation of `take 2 (from 0)`

```
take 2 (from 0)
= take 2 (0 : from 1)
= take 2 (0 : 1 : from 2)
= take 2 (0 : 1 : 2 : from 3)
= take 2 (0 : 1 : 2 : 3 : from 4)
= take 2 (0 : 1 : 2 : 3 : 4 : from 5)
= ...
```

(You get the idea . . .)

Lazy Evaluation of `take 2 (from 0)`

Here we don't bother to show the closures explicitly (using `where` $\texttt{xs}_1$ `= ...`).

```
take 2 (from 0)
= take 2 (0 : from 1)
= 0 : (take 1 (from 1))
= 0 : (take 1 (1 : from 2))
= 0 : (1 : take 0 (from 2))
= 0 : (1 : [])
```

We are done ! We only built the bit of `from 0` that we actually needed.

Evaluation Strategy and Termination

We can summarise the relationship between evaluation strategy and termination as:

- There are programs that simply do not terminate, no matter how they are evaluated
  e.g. `from 0`
- There are programs that terminate if evaluated lazily, but fail to terminate if evaluated strictly
  e.g. `take 2 (from 0)`
- There are programs that terminate regardless of chosen evaluation strategy
  e.g. `len (down 1)`
- However, there are *no* programs that terminate if evaluated strictly, but fail to terminate if evaluated lazily.