Announcements

- Midterm #1 next Monday. More details to follow.
- "A computer is like a violin. You can imagine a novice trying first a phonograph and then a violin. The latter, he says, sounds terrible. Computer programs are good, [some] say, for particular purposes, but they aren't flexible. Neither is a violin, or a typewriter, until you learn how to use it."
  - *Marvin Minsky, "Why Programming Is a Good Medium for Expressing Poorly-Understood and Sloppily-Formulated Ideas", 1967*

Review

- Haskell is a functional programming language
- Strongly typed, but with type inference

  Bool
  Num, Int, Integer, Fractional, Floating, Double
  Eq, Ord
  Tuple, List, Function
- Classes, type variables
- List comprehension $[f\ \ x \mid x{<}{-}list,\ \ cond\ \ x]$
- $foldr\ \ \oplus\ \ v\ [a1, a2, ..an] = a1 \oplus (a2 \oplus (\ldots \oplus (an \oplus v)))$
- $foldl\ \ \oplus\ \ v\ [a1, a2, ..an] = (((v \oplus a1) \oplus a2) \oplus \ldots) \oplus an$

Clicker Question

- $foldr \; \oplus \; v \; [a1, a2, ..an] = a1 \oplus (a2 \oplus (\ldots \oplus (an \oplus v)))$

Given

```
ml = foldr (\ x y -> y+1) 0
```

what is the result of

```
ml [10,11,12,13,14,15]
```

A `[11,12,13,14,15,16]`

B `6`

C `11`

D `[True,True,True,True,True,False]`

E It gives a type error

Clicker Question

- $foldr \;\oplus\; v\; [a1, a2, ..an] = a1 \oplus (a2 \oplus (\ldots \oplus (an \oplus v)))$

Given

```
bar = foldr (\ x y -> x+1) 0
```

what is the result of

```
bar [10,11,12,13,14,15]
```

A `[11,12,13,14,15,16]`

B 6

C 11

D `[True,True,True,True,True,False]`

E It gives a type error

Clicker Question

- $foldr \;\oplus\; v\ [a1, a2, ..an] = a1 \oplus (a2 \oplus (... \oplus (an \oplus v)))$
- $map\ f\ [a1, a2, ..an] = [f\ a1, f\ a2, .., f\ an]$

Which of the following implement *map*

A `map f lst = foldr (\x y -> f x:y) [] lst`

B `map f lst = foldr (\x y -> f x: map f y) [] lst`

C `map f lst = foldr (\x y -> f x) [] lst`

D `map f lst = foldr (\x y -> x:f y) [] lst`

E None: *foldr* cannot be used to implement *map*

Clicker Question

$$foldr\ \oplus\ v\ [a1, a2, ..an] = a1 \oplus (a2 \oplus (... \oplus (an \oplus v)))$$
$$foldl\ \oplus\ v\ [a1, a2, ..an] = (((v \oplus a1) \oplus a2) \oplus ...) \oplus an$$

```
add1y x y = y+1
add1x x y = x+1
```

What returns the length of the list [7..9]?

- A (`foldr add1y 0 [7..9]`) and (`foldl add1y 0 [7..9]`)
- B (`foldr add1y 0 [7..9]`) and (`foldl add1x 0 [7..9]`)
- C (`foldr add1x 0 [7..9]`) and (`foldl add1y 0 [7..9]`)
- D (`foldr add1x 0 [7..9]`) and (`foldl add1x 0 [7..9]`)
- E all four (`foldr add1y 0 [7..9]`) and (`foldl add1y 0 [7..9]`) and (`foldr add1x 0 [7..9]`) and (`foldl add1x 0 [7..9]`)

Clicker Question

$$foldr \ \oplus \ v \ [a1, a2, ..an] = a1 \oplus (a2 \oplus (... \oplus (an \oplus v)))$$
$$foldl \ \oplus \ v \ [a1, a2, ..an] = (((v \oplus a1) \oplus a2) \oplus ...) \oplus an$$

Which of the following gives a type error at compilation time

```
(i)  foldr (:) [] [1,2,3,4,5]
(ii) foldl (:) [] [1,2,3,4,5]
```

- A neither give an error
- B (i) gives an error and (ii) doesn't
- C (ii) gives an error and (i) doesn't
- D they both give an error

Call-by-name and Call-by-value

- Recall: Definition
  `foo x = exp` is an abbreviation for
  `foo = \ x -> exp`
  Writing `foo` is same as `\ x -> exp`
- `foo x y = exp` is an abbreviation for
  `foo = \ x -> \y -> exp`
- Reduction:
  `(\ x -> f(x)) a` reduces to `f(a)`
  substitute argument for formal parameter.
- Example:

```
m x y = x*y
m (10-5) (m 10 5)
```

- Call-by-value: evaluate arguments before reduction: `m 5 50`
- Call-by-name: reduction of function first: `(10-5)*(m 10 5)`

Call-by-name and Call-by-value

- Call-by-value: evaluate arguments before reduction
- Call-by-name: reduction of function first
- What does the following do?

  ```
  inf = 1+inf
  ```

- Does following halt?

  ```
  inf = 1+inf
  fst (x,y) = x
  fst (3+2, inf)
  ```

- ```
  sq x = x*x
  sq (55+45)
  ```

- If they both halt, they give same answer
  ```

Lazy Evaluation

- Lazy evaluation: evaluate argument only once, only if needed
- Evaluation Order:
  - Evaluation from outside in
  - Otherwise (if it knows both arguments need to be evaluated) from left to right
- Example:

```
from1 a =  a: from1 (a+1)
mytake 0 _ = []
mytake _ [] = []
mytake n (x:xs) = x:mytake (n-1) xs
-- mytake 2 (from1 10)
```

Lazy Evaluation

- It is possible to evaluate all arguments that need to be evaluated in parallel.
- One could build a compiler that memorizes the results of all previous function calls.
  GHC does not do that. It just caches locally.
- Lazy evaluation enables forms of programming that are not possible with call by value. E.g., definition of if-then-else

```
myif True  then_exp else_exp = then_exp
myif False then_exp else_exp = else_exp
fac n = myif (n==0) 1 (n*fac (n-1))
```

Lazy Computation Examples (Lazy.hs)

- ```
  foldr f v []     = v
  foldr f v (x:xs) = f x (foldr f v xs)
  foldr(\ x y -> x+1) 0 [10..]
  ```
- ```
  lstto 0 = []
  lstto n = n:lstto (n-1)
  mysum [] = 0
  mysum (h:t) = h+mysum t
  ```
- `mysum (lstto 5)`

Lazy Computation Examples: finding primes (Lazy.hs)

- Eratosthenes of Cyrene (276 BCE – c.195/194 BCE) estimated circumference of Earth (accurately!), founded geography, and defined one of the first non-trivial algorithms.
- Sieve of Eratosthenes
  start with the list of all numbers ≥ 2,
  when found a prime, cross off the multiples of that prime from the rest of the list. The next element on the list is prime.
  - - sieve (p:xs) is the list of all primes from p, given all of a multiples of primes less than p have been removed.

```
primes = sieve [2..]
  where sieve (p:xs) =
            p : sieve [x | x <- xs, x `mod` p /= 0]
take 100 primes
```

Computing Fibonacci numbers (super fast(?))

Fibonacci numbers: $f_n = f_{n-1} + f_{n-2}$
Naive Fibonacci n takes time exponential in n.
Fast Fibonacci n takes time linear in n
Can we compute the Fibonacci n in time logarithmic in n?

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} f_n \\ f_{n-1} \end{pmatrix} = \begin{pmatrix} f_n + f_{n-1} \\ f_n \end{pmatrix}$$

$$\begin{pmatrix} f_n \\ f_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

We can compute x^n in logarithmic time....
see Lazy.hs