"I think that it's extraordinarily important that we in computer science keep fun in computing. When it started out, it was an awful lot of fun. Of course, the paying customers got shafted every now and then, and after a while we began to take their complaints seriously. We began to feel as if we really were responsible for the successful, error-free perfect use of these machines. I don't think we are. I think we're responsible for stretching them, setting them off in new directions and keeping fun in the house. I hope the field of computer science never loses its sense of fun."

– *Alan J. Perlis, 1977 (quoted in dedication to "Structure and Interpretation of Computer Programs", 1985)*

Announcements

- Midterm #1 next Monday.
  - You can write your personalized exam in class time or in any 50 minutes in the 24 hours before the end of the exam.
  - Open-book. You can use Google and ghci. (But won't help ;)
  - All predefined functions you can use will be described (type and their API).
  - A practice (previous) exam is on course web page.

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 $[f\ x \mid x\texttt{<-}list,\ cond\ x]$
- $foldr\ \oplus\ v\ [a1, a2, ..an] = a1 \oplus (a2 \oplus (... \oplus (an \oplus v)))$
- $foldl\ \oplus\ v\ [a1, a2, ..an] = (((v \oplus a1) \oplus a2) \oplus ...) \oplus an$
- reduction
- Call by value, call by name, lazy evaluation

Computing Fibonacci numbers (super fast(?))

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

Type

- `type` defines a type name as an abbreviation for other types
- Example:

  ```
  type String = [Char]
  ```

- Example matrices and matrix multiplication:

```
type Matrix = Int -> Int -> Integer
mm:: Matrix -> Matrix -> Matrix
```

Types

- type can be parametrized by type variables.
- Example

```
type Matrix t = Int -> Int -> t
mm:: Num t => Matrix t -> Matrix t -> Matrix t

-- m1110 is a particular 2x2 matrix
m1110 :: Matrix Integer
m1110 2 2 = 0
m1110 _ _ = 1
```

The mathematical definition of matrix multiplication:

```
mm:: Num t => Matrix t -> Matrix t -> Matrix t
mm m1 m2 i j = sum [(m1 i k)*(m2 k j) | k <- [1..size]]
size = 2
```

(see Lazy.hs)

data

- `data` defines new data structures (and a type). Example:

  ```
  data APerson = Person String Int
  ```

  defines
  - a new type: `APerson`
  - a constructor: `Person`
- The constructors have dual roles:
  - constructors with arguments can be used as functions
    `sam = Person "Sam" 27` is object of type `APerson`
  - constructors with no arguments are constants
  - constructors can be used patterns for deconstructing the type (accessors) on left side of = or in function definitions.

```
showPerson (Person name age) = name ++ show age
```

data (cont). (Dataeg.hs)

`data` definitions can be recursive:

```
data MyListInteger =
                Empty
              | ConsI Integer MyListInteger
   deriving Show
```

defines

- a new type: `MyListInteger`
- 2 new constructors: `Empty`, `ConsI`
- defines `show :: MyListInteger -> String`
- Constructors can be used to define entities of the type:
  `ConsI 27 Empty` is a list of 1 elements
  `ConsI 234 (ConsI 27 Empty)` is a list of 2 elements
- constructors can be used patterns for deconstructing the type (accessors) on left side of $=$ or in function definitions.

```
myApp Empty lst = lst
myApp (ConsI h t) lst = ConsI h (myApp t lst)
```

Type and data (Dataeg.hs)

```
data FValue =  BooleanF Bool
             | NumberF Int
             | StringF [Char]
             | MissingF
```

defines

- a new type: *FValue*
- 4 new constructors: *BooleanF*, *NumberF*, *StringF*, *MissingF*.

Clicker Question

Given

```
data MD = Fun Int
        | Bar
```

Which of the following is a type?

A MD

B Fun, Bar

C MD, Fun, Bar

D Fun

Clicker Question

Given

```
data MD = Fun Int
        | Bar
```

Which of the following is a function?

A MD

B Fun, Bar

C MD, Fun, Bar

D Fun

Clicker Question

Given

```
data MD = Fun Int
        | Bar
```

Consider the following

(i) `Fun 34`

(ii) `Bar`

(iii) `57`

(iv) `MD Fun Bar`

Which define an expression of type *MD*?

A (i) and (ii)

B (iii)

C (i), (ii), (iii)

D (iv)

E All of them

Parameterized types

In the Prelude is the definition:

```
data Maybe t = Nothing
             | Just t
```

- `t` is a type variable. Just as in `[t]`.
- `Maybe t` is a type for all types `t`.
- It is useful when a function is partial and doesn't always return a value.

Clicker Question

In the Prelude is the definition:

```
data Maybe t = Nothing
             | Just t
```

Which of the following not true:

A `Maybe Int` is a type

B `Nothing` and `Just` are both functions of type `Maybe`

C `Nothing` is a constant of type `Maybe t` for all types t

D `Just` has type

```
Just :: a -> Maybe a
```

Recursive data types (BSTree.hs)

- `data` definitions can be recursive: Example:

```
-- a binary search tree that maps integers to strings
data BSTree = Empty
            | Node Int String BSTree BSTree
```

  defines a new type, *BSTree*, and two new constructors *Empty* and *Node*.

- The constructors can be used as
  - constants (Empty) or functions (Node) to create a BSTree
  - patterns for deconstructing the type
- The data structures can be parametrized by types:

```
-- a binary search tree
--   k is the key type; v is the value type
data BSTree k v = Empty
                | Node k v (BSTree k v) (BSTree k v)
```