Quotation for the day

... "computer science" is not a science and .. its significance has little to do with computers. The computer revolution is a revolution in the way we think and in the way we express what we think.

— Harold Abelson and Gerald Jay Sussman, "Structure and Interpretation of Computer Programs", 1985

Review

- Haskell Types:

  Bool (&&, ||, not)
  Num ($+$, $-$, $*$, abs)
      Integral (div, mod)
          Int
          Integer
      Fractional (/)
          Floating (log, sin, exp, ...)
              Double
  Eq (==, /=)
      Ord (>, >=, <=, <)
  List ([] :)
  Char
  String

Integral types

- Intergral types represent integers.
- They implement `+ * ^ - div mod abs negate`
- Two implementations:
  - Int - fixed-precision integers
  - Integer - arbitrary precision integers
- Integral is a class.
  Int and Integer are types in class Integral.
  Only types have implementations.
  (Haskell classes are like Java interfaces)

Fractional types

- Fractional types represent real numbers.
- They implement `+ * ^ - / abs negate`
- Floating types also implement `log sin exp` ...
- Multiple implementations:
  - Double - double precision floating-point numbers (64 bit)
  - Float - single precision floating-point numbers (32 bit) — don't use
  - Rational - exact rational numbers
- There are no types that are in both Integral and Fractional
- Num types implement `+ * ^ - abs negate`
  Num is a class (elements are types).
  Integral and Fractional are subclasses of Num.
  Floating is a subclass of Fractional.

Eq and Ord classes

- Eq types implement `==` `/=`
- Ord types implement `>` `>=` `<=` `<` `max` `min`
- Int, Integer, Double implement Eq and Ord
- Can you think of a Num type that isn't an Ord type?
  How about Complex?

Clicker Question

What is the inferred type of
`1.7 + fromIntegral (div 100 7)`

- A `Int`
- B `Fractional a => a`
- C `Double`
- D `Integral a => a`
- E `Num a => a`

Clicker Question

The inferred type of `==` is

`Eq a => a -> a -> Bool`

Based on this type signature, which of the following is not true:

- A `==` is a function that takes two arguments
- B the type of the first argument to `==` must be the same as the type of the second argument
- C the type of the first argument must be in the Eq class
- D the two arguments to `==` must be the same as each other
- E `x == y` must return either True or False

Lists

- A list is an ordered sequence of elements of the same type
- A list of type `[t]`, where `t` is a type, is either:
  - the empty list `[]`
  - of the form `h:r` where `h` is of type `t` and `r` is a list of type `[t]`
    `:` is an infix function.
- A list has a special syntax :
  [7] is an abbreviation for 7:[]
  [5,7] is an abbreviation for 5:7:[]
  [3,5,7] is an abbreviation for 3:5:7:[]
  : associates to the right
- both [...] and : notation can be used in patterns on the left side of $=$.
  `myelem e lst` is True whenever `e` is in `lst`

```
myelem e [] = False
myelem e (h:t) = e==h || myelem e t
```

Examples (Lists.hs)

- `myappend l1 l2`
  returns the list containing the elements of list `l1` followed by elements of `l2`
- This can also be defined as infix function
  `l1 ++++ l2`
  returns the list containing the elements of list `l1` followed by elements of `l2`
- A string is a list of characters
  `type String = [Char]   -- Defined in ‘GHC.Base’`
- `[1,2,3] ++++ ['a','b']`
  gives an error. Why?
- The standard Prelude defines ++ for append.

Examples (Lists.hs)

Let's define the following:

- `numeq e lst`
  returns the number of instances of *e* in list *lst*.
  `numeq 4 [7,1,4,5,4,6,7,4,8]` returns 3
- `numless x lst`
  returns the number of elements of list `lst` less than `x`

Clicker Question

The inferred type of `numeq` is

```
numeq :: (Num p, Eq t) => t -> [t] -> p
```

Based on this type signature, which of the following is not true:

- A the type of the first argument must implement == and /=
- B the type of the first argument must be the same as the type of every element of the list that is the second argument
- C `numeq` must return a value of a type in the Num class
- D `numeq` takes two arguments
- E `numeq` must return an Int

Examples (Lists.hs)

- `numeq e lst`
  returns the number of instances of *e* in list *lst*.
- `numless x lst`
  returns the number of elements of list `lst` less than `x`
- Define
  `numc c lst`
  returns number of elements in `lst` that have condition `c` true
- Define `numeq` using `numc`
- How can we use `numc` to count the number of items in a list that are less than 4?

List Examples (Lists.hs)

Define

- *numeq x lst* = number of instances of x in list *lst*.
- *numc c lst* = number of elements of *lst* for which *c* is True
- *filter c lst* = list of elements of *lst* for which *c* is True
- *filter* is the only one predefined. Why?
  More general definitions are easier to define, use and remember.
- How can numc and numeq be defined in terms of filter?
- *length*(*filter c lst*) does not create a list (with lazy evaluation).

Types Revisited

- Type declaration:

  $$exp :: cc \ \texttt{=>} \ te$$

  exp is an expression,
  cc is a class constraint of form $C\ a$ where C is the name of a class (e.g, Num, Integral, Show,...) and a is a type variable.
  te is a type expression.
- A function from type b to type c is of type `b -> c`
- A list of type b is of type $[b]$
- A 3-tuple (triple) of elements of type b, c, d is of type (b, c, d).
- What is the type of *mylen* that takes a list and returns an Int?
  `mylen :: [a] -> Int`
- What is the type of $+$ that adds two numbers?
  `(+) :: Num a => a -> a -> a`
- What is the type of *div* (integer division)?
  `div :: Integral a => a -> a -> a`