A programming language designer should be responsible for the mistakes made by programmers using the language. It is a serious activity; not one that should be given to programmers with 9 months experience with assembly; they should have a strong scientific basis, a good deal of ingenuity and invention and control of detail, and a clear objective that the programs written by people using the language would be correct, free of obvious errors and free of syntactical traps.

— Tony Hoare, Null References: The Billion Dollar Mistake, 2009
`https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/`

- Assignment 1 solution and
- Assignment 2 on schedule tab of web page.

Review

- Haskell Types:

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

Some Predefined list definitions (Lists2.hs)

- $[e1..en]$ is the list of elements from e1 to en (inclusive)
  $[e1, e2..em]$ is the list of elements from e1 to em, where
  $e2 - e1$ gives step size
  $[e..]$ is the list of all numbers from e
- take n lst   first n elements of lst
- head lst   is the first element of lst
  tail lst   is the rest of the list
- lst !! n   nth element of lst
- lst1 ++ lst2   append lst1 and lst2
- $sum\ [a1, a2, ..an] = a1 + a2 + ... + an$
- zip [a1,a2,...,an] [b1,b2,...,bn] = [(a1,b1),(a2,b2),...,(an,bn)]
- map f [a1,a2,...,an] = [f a1,f a2,...,f an]

Lambda

- How can we find elements of a list that are less than 3 or greater than 7 (using filter)?
- Lambda lets us define a function without giving it a name.

  ```
  \ x -> (x < 3) || (x > 7)
  ```

  is a function true of numbers less than 3 or greater than 7
- `filter (\ x -> (x < 3) || (x > 7)) [1..10]`
  is easy to read and work out what it is saying
- A definition

  ```
  foo x = exp
  ```

  is an abbreviation for

  ```
  foo = \ x -> exp
  ```

- `foo x y = exp` is an abbreviation for
  `foo = \ x -> \y -> exp` also written
  `foo = \ x y -> exp`
- `myadd = \x y -> x+y`

Local Definitions

- `where` can be used for local definitions the definition of functions:

```
fun args = exp
   where
     local = val
```

  is an abbreviation for

```
fun args =
   ((\ local -> exp ) val)
```

- `let` can be used anywhere an expression is used:

```
let local = val
   in
     exp
```

  is an abbreviation for

```
((\ local -> exp ) val)
```

List Comprehensions

- In mathematics, what is

$$\{x^2 \mid x \in \{1,2,3,4,5,6,7\},\ x \bmod 2 = 1\}$$

- This is written in Haskell as

  ```
  [x^2 | x <- [1..7], x `mod` 2 == 1]
  ```

  "List Comprehension"

- List comprehensions can do everything filter and map can do.
- This can use pattern matching, e.g.,

  ```
  [x+y | (x,y) <- [(1,2),(4,3),(5,6)]]
  [x+y | (x,y) <- [(1,2),(4,3),(5,6)], x<y]
  ```

- Implement dot-product of $[a_1, \ldots, a_n]$ and $[b_1, \ldots, b_n]$

$$\sum_i a_i * b_i$$

Clicker Question

Given

```
even n = 0 == mod n 2
```

what is the result of

```
[even x | x <- [1,2,3,4,5,6]]
```

A `[2,4,6]`

B `[2,4,6,8,10,12]`

C 3

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

E It gives a type error

Clicker Question

Given

```
even n = 0 == mod n 2
```

what is the result of

```
[x | x <- [1,2,3,4,5,6], even x]
```

- A `[2,4,6]`
- B `[2,4,6,8,10,12]`
- C 3
- D `[False,True,False,True,False,True]`
- E It gives a type error

List Definitions (foldr and friends) Lists3.hs

Define:

- $sum\ [a1, a2, ..an] = a1 + a2 + ... + an$
- $product\ [a1, a2, ..an] = a1 * a2 * ... * an$
- $or\ [a1, a2, ..an]$ is True when one the ai is *True*
- $append\ [a1, a2, ..an]\ l2 = a1 : a2 : ... : an : l2$
- generalized to
  $foldr\ \oplus\ v\ [a1, a2, ..an] = a1 \oplus (a2 \oplus (... \oplus (an \oplus v)))$
- How can we define *sum*, *product*, *or*, *and* using foldr?
- What does the following return?
  `foldr (:) [5,6,7] [1,2,4]`
  How can we define *append* using foldr?
  Haskell append is written as infix $++$
- Define dot-product using foldr and zip.

```
-- dotprod [x1,..,xn] [y1,..,yn] = x1*y1+...+xn*yn
dotprod v1 v2 = foldr (\ (x,y) s -> x*y+s) 0
                                (zip v1 v2)
```