Thought for the day

"...there are two ways of constructing a software design: One way is to make it so simple that there are *obviously* no deficiencies and the other way is to make it so complicated there are no *obvious* deficiencies. The first method is far more difficult."

— Tony Hoare, 1980 ACM Turing Award Lecture

Review

- Haskell Types:

  Bool (&&, ||, not)
  Num ($+$, $-$, $*$, abs)
  - Integral (div, mod)
    - Int
    - Integer
  - Fractional (/)
    - Floating (log, sin, exp, ...)
      - Double

  Eq (==, /=)
  - Ord (>, >=, <=, <)

  List ([] :)
  Char
  String

Guards

- Guards are used for if-then-else structure in definition of functions.
- General case:

```
name x1 x2 ... xk
    | g1 = e2
    | g2 = e2
     ...
    | gn = en
```

- evaluate g_1, g_2 in turn until the first one g_i evaluates to true, then return value of e_i.
- An Exception is raised if none of the guards are True
- Typical to have last condition to be `otherwise` which is a variable with value `True`.
- Haskell also has "if ... then ... else ..." structure

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 need to actually create a list.

Types Revisited

- Type declaration:

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

  exp is an expression,
  cc is a (tuple of) class constraint of form $C\ a$ where C is a class (e.g, Num, Integral,...) 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). (Similarly for other-length tuples).
- What is the type of *length* that takes a list and returns an Int?
  `length :: [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`

Types (cont)

- What is the inferred type of *numeq*?

```
numeq _ [] = 0
numeq x (h:t)
    | x==h = 1+numeq x t
    | otherwise = numeq x t
```

```
numeq :: (Num a, Eq a1) => a1 -> [a1] -> a
```

  Note: *a* and *a*1 could be same or different types.

- What is the inferred type of *numc*?

```
numc _ [] = 0
numc c (h:t)
    | c h       = 1+numc c t
    | otherwise = numc c t
```

```
numc :: Num a => (t -> Bool) -> [t] -> a
```

Clicker Question

The inferred type of `numeq` is

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

What is the inferred type of `numless`:

```
numless _ [] = 0
numless x (h:t)
    | h<x        = 1 + numless x t
    | otherwise = numless x t
```

- A `numless :: (Num p, Eq t) => t -> [t] -> p`
- B `numless :: (Num p, Ord t) => t -> [t] -> p`
- C `numless :: (Num p) => t -> [t] -> p`
- D `numless :: t -> [t] -> p`
- E `numless :: Int -> [Int] -> Int`

Clicker Question

What is the inferred type of *myelem* defined by

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

A `myelem :: Eq a => a -> [b] -> Bool`
B `myelem :: Eq t => t -> [t] -> Bool`
C `myelem :: a -> [b] -> Bool`
D `myelem :: a -> [b]`
E I have no idea

See
`http://cs.ubc.ca/~poole/cs312/2024/haskell/Lists2.pl`

Clicker Question

What is the inferred type of *mytake* defined by

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

A `mytake :: Int -> [Int] -> [Int]`

B `mytake :: (Num a, Eq a) => a -> [t] -> [t]`

C `mytake :: (Num a, Eq a) => a -> [t] -> t`

D `mytake :: (Num a, Eq a) => a -> t -> t`

E I have no idea

See
`http://cs.ubc.ca/~poole/cs312/2024/haskell/Lists2.pl`

Clicker Question

What is the inferred type of *numeqh* defined by

```
numeqh _ [] n = n
numeqh x (h:t) n
     | x==h       = numeqh x t (n+1)
     | otherwise = numeqh x t n
```

A `numeqh :: (Num b, Eq a) => (a, [a], b) -> b`

B `numeqh :: (Num a, Eq a) => a -> [a] -> a -> a`

C `numeqh :: (Eq a) => a -> [a] -> Int -> Int`

D `numeqh :: (Num b, Eq a) => a -> [a] -> b -> b`

E I have no idea

Clicker Question

What is the inferred type of `flip` defined by

```
flip f a b = f b a
```

A `flip :: (t1 -> t2) -> t2 -> t1`

B `flip :: (t -> t -> t) -> t -> t -> t`

C `flip :: (t -> t) -> t -> t`

D `flip :: (t1 -> t2 -> t) -> t2 -> t1 -> t`

E I have no idea

Clicker Question

Consider the functions `flip` and `hh` defined by

```
flip f a b = f b a
hh x y z = 10000*x + 100*y + z
```

What is the value of

```
flip hh 3 5 7
```

(It does not give an error.)

A 30507

B 70503

C 50307

D 30705

E 70305

Clicker Question

Consider the functions `flip` and `hh` defined by

```
flip f a b = f b a
hh x y z = 10000*x + 100*y + z
```

What is the value of

```
flip (hh 3) 5 7
```

(It does not give an error.)

A 30507

B 70503

C 50307

D 30705

E 70305

Clicker Question

filter defined by

```
filter _ [] = []
filter c (h:t)
    | c h       = h:filter c t
    | otherwise = filter c t
```

has type:

- A `filter :: t -> Bool -> [t] -> [t]`
- B `filter :: ([t] -> Bool) -> [t] -> [t]`
- C `filter :: (t -> Bool) -> t -> t`
- D `filter :: (t -> Bool) -> [t] -> [t]`
- E it does not have a legal type (and will result in a type error)

Clicker Question

filter, even are defined by:

```
filter _ [] = []
filter c (h:t)
    | c h       = h:filter c t
    | otherwise = filter c t
even n = 0 == mod n 2
```

what is the result of

```
filter even [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 the definitions:

```
filter _ [] = []
filter c (h:t)
    | c h       = h:filter c t
    | otherwise = filter c t
even n = 0 == mod n 2
nums = [1,2,4,5,6,7,8,10,11]
```

Which query will return the number of even elements of nums

- A `length filter even nums`
- B `filter length even nums`
- C `length (filter even nums)`
- D `filter even nums length`
- E None of the above

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]