

# Contents

<b>Exam</b>	<b>2</b>
Question 1 . . . . .	2
Question 2 & 3 . . . . .	2
Question 4 . . . . .	3
<b>Prelude</b>	<b>3</b>
2016 . . . . .	3
a . . . . .	3
b . . . . .	3
c . . . . .	4
d . . . . .	4
e . . . . .	4
f . . . . .	4
2015 . . . . .	4
a . . . . .	4
b . . . . .	4
c . . . . .	5
d . . . . .	5
e . . . . .	5
f . . . . .	5
<b>Abstraction &amp; HOF</b>	<b>5</b>
2016 . . . . .	6
a . . . . .	6
b . . . . .	7
<b>Runtime Errors</b>	<b>7</b>
2016 . . . . .	7
<b>Maybe</b>	<b>8</b>
2016 . . . . .	8

<b>ASTs</b>	<b>10</b>
A . . . . .	10
B . . . . .	10
C . . . . .	10
2016 . . . . .	11
<b>Proofs</b>	<b>12</b>
Induction . . . . .	12
2016 b . . . . .	13

## Exam

- Summer 2017: Same format as 13-15
  - 2hrs
  - 4 questions
  - Do 3
- Ignore Q5 on Summer 2016 (**WOO!**)
- Overall grade is just the sum of coursework percentage and exam percentage (25/75)
  - Don't need to pass coursework seperately

## Question 1

- Focus on *basics*
  - Pattern matching
  - Recursion

## Question 2 & 3

- 2-3 from 2016
- 2-4 from earlier years
- Focus on more advanced aspects
  - HOF
  - Maybe/Either
  - ASTs
  - Laziness, I/O (Monads)

## Question 4

- Not taught or examined before 2016
- Focus is on mathematical reasoning about program behaviour
  - Equational Reasoning
  - Recursion handled by induction
  - Laziness handled by co-induction

## Prelude

### Pay attention to the typing information

- Always think about the types and what the types are doing
- You lose most marks for writing something that doesn't type check

Can use previously implemented functions in the question!

Errors:

- You get runtime errors for free!
- Can handle errors if you like using `error` but not necessary
  - Don't need to implement `error`, uses some unsafe stuff...

Note: The answers at the revision lecture were similar to my own and I didn't take them down but I've included my answers for the last two years.

## 2016

**a**

```
head :: [a] -> a
head (x:xs) = x
```

**b**

```
init :: [a] -> [a]
init [x] = []
init (x:xs) = x:init xs
```

c

```
last :: [a] -> a
last [x] = x
last (x:xs) = last xs
```

d

```
span :: (a -> Bool) -> [a] -> ([a], [a])
span p [] = ([], [])
span p (x:xs)
  | p x      = (x:fst(z), snd(z))
  | otherwise = ([], x:xs)
  where
    z = span p xs
```

e

```
(!!) :: [a] -> Int -> a
(!!) (x:_) 0 = x
(!!) (x:xs) n = xs !! (n-1)
```

f

```
foldl1 :: (a -> a -> a) -> [a] -> a
foldl1 op [x] = x
foldl1 op (x:y:zs) = foldl1 op ((x `op` y):zs)
```

## 2015

a

```
repeat :: a -> [a]
repeat a = a:(repeat a)
```

b

```
replicate :: Int -> a -> [a]
replicate 0 _ = []
replicate n l = l:(replicate (n-1) l)
```

c

```
concat :: [[a]] -> [a]
concat [] = []
concat (x:xs) = x++(concat xs)
```

d

```
zip :: [a] -> [b] -> [(a, b)]
zip [] [] = []
zip [] _ = []
zip _ [] = []
zip (x:xs) (y:ys) = (x, y):(zip xs ys)
```

e

```
unzip :: [(a, b)] -> ([a], [b])
unzip [] = ([], [])
unzip ((a, b):xs) =
  (a:as, b:bs)
  where (as, bs) = unzip xs
```

f

```
fminimum :: (Ord a) => a -> [a] -> a
fminimum x [] = x
fminimum m (x:xs)
  | x < m      = fminimum x xs
  | otherwise  = fminimum m xs

minimum :: (Ord a) => [a] -> a
minimum [] = error "Empty list"
minimum (x:xs) = fminimum x xs
```

## Abstraction & HOF

1. Create a higher order function `hof _ _ _ = ...`
2. Implement `f1, f2, f3, ...` using `hof`
  - `f1 = hof ...`
  - `f2 = hof ...`

Go back to question 1 for a brief second:

```
head :: [a] -> a
head [] = ???
```

Not forced to live in a world with arbitrary types.

```
f1 :: Num a => [a] -> a
f1 [] = 0
```

**Don't need a type signature unless asked for.**

Top tip: When asked for what prelude function this HOF is, check the Prelude Reference!

## 2016

**a**

```
fn [] = e                -- 1, 0, 0, [], 0
fn (x:xs) = f xs
```

Look at what the operator is. Focus on the operator being **that**.

```
fn [] = e                -- 1, 0, 0, [], 0
fn (x:xs) = x `fop` (fn xs) -- fop = funny op
```

Whatever funny op is doing, it takes in two arguments.

```
f1: x `fop` y = x*y
f2: x `fop` y = 1+y
f5: x `fop`   = x*x+y
```

Can also think of it as pre-processing of x, i.e.

```
fn [] = e                -- 1, 0, 0, [], 0
fn (x:xs) = (f x) `op` (fn xs)
```

Both are perfectly valid answers.

Right now, referring to things - fop, e. Where do they come from? A higher order function is a function that wraps up all this info.

```
hof e fop [] = e
hof e fop (x:xs) = x `fop` (hof e fop xs)
```

Type signature (not needed unless explicitly asked for):

```
hof :: a -> (b -> a -> a) -> [b] -> a
```

b

Don't do this:

```
f1 hof e fop [] = e
f1 hof e fop (x:xs) = x `fop` f1 xs
```

What to actually do:

```
f1 = hof 1 (*)
f2 = hof 0 f2op
    where f2op x y = 1+y
f3 = hof 0 (+)
f4 = hof [] (++)
f5 = hof 0 f5op
    where f5op x y = (x*x)+y
```

## Runtime Errors

How can the function fail with Haskell runtime errors?

```
search :: Tree -> Int -> String
search x (Many left i s right)
```

This is a compile time error! (also a typo)

What you want to see is how can you break this? If you focus on the code, you'll miss something obvious. It's all about the types!

## 2016

```
data Tree = Empty
          | Single Int String
          | any Tree Int String Tree
```

Look at `Empty`

- Code will fail if there is an `Empty` tree

Look at `Single`

```
search x (Single i s)
  | x == i = s
```

- If `x != i` the code will fail

Look at `Many`

```
search x (Many left i s right)
  | x == i = s
  | x > i = search x right
```

- If `x < i` the code will fail

`search x right` can induce any of these conditions. You'll get one of these three runtime errors. Only safe path is to only go right down the tree.

## Maybe

Only code of interest is supplied. If code isn't supplied, it's assumed to be correct.

Has sometimes asked to use `Either` in the past.

## 2016

```
type Dict = [(String, Int)]
ins :: String -> Int -> Dict -> Dict
lkp :: String -> Dict -> Maybe Int

data Expr = K Int
          | V String
          | Add Expr Expr
          | Dvd Expr Expr
          | Let String Expr Expr

eval :: Dict -> Expr -> Int
eval _ (K i) = i
eval d (V s) = fromJust $ lkp s d
eval d (Add e1 e2) = eval d e1 + eval d e2
eval d (Dvd e1 e2) = eval d e1 `div` eval d e2
eval d (Let v e1 e2) = eval (ins v i d) e2
                      where i = eval d e1
```



Code has a bunch of runtime errors! Will fail if `lkp` returns `Nothing`. Turn this into a function without any runtime errors! Rather than returning an `Int`, return a `Maybe Int`.

- Could define an instance for `Maybe Int`
  - Do you treat `Nothing` as 0?
- Could use Monads

```
eval :: Dict -> Expr -> Maybe Int
eval _ (K i) = Just i
eval d (V s) = lkp s d
eval d (Add e1 e2) = case (eval d e1, eval d e2) of
    (Nothing, _)    -> Nothing
    (_, Nothing)    -> Nothing
    (Just x, Just y) -> Just (x+y)
eval d (Dvd e1 e2) = case (eval d e1, eval d e2) of
    (Nothing, _)    -> Nothing
    (_, Nothing)    -> Nothing
    (Just x, Just y)
        | i2 == 0    = Nothing
        | otherwise = Just (x `div` y)
eval d (Let v e1 e2) = case (eval d e1) of
    Nothing -> Nothing
    Just i  -> eval (ins v i d) e2
```

or

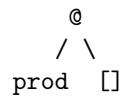
```
eval :: Monad m => Dict -> Expr -> m Int
eval _ (K i) = return i
eval _ (V s) = case lkp s d of
    Nothing -> fail " "
    Just i   -> return i
eval d (Add e1 e2) = do i1 <- eval d e1
    i2 <- eval d e2
    return (i1+i2)
eval d (Div e1 e2) = do i1 <- eval d e1
    i2 <- eval d e2
    if i2 == 0 then fail " "
    else (i1 `div` i2)
```

Careful about what type asked! In 2016 `Maybe` type was explicitly asked for. Can use Monads and at the end return with the right type signature.

## ASTs

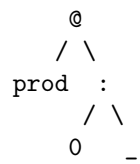
### A

prod []



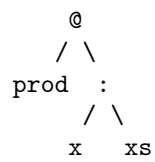
### B

prod (0:\_)

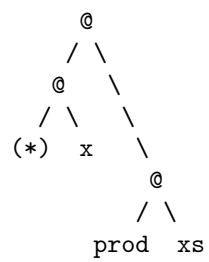


### C

prod (x:xs)

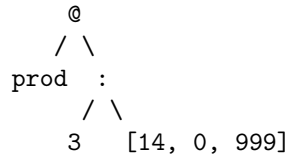


x \* prod xs = (\*) x (prod xs)



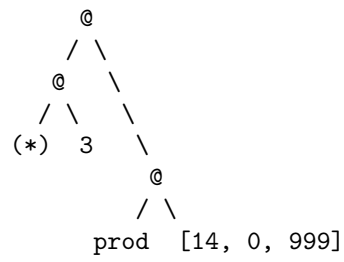
2016

prod [3, 14, 0, 999]



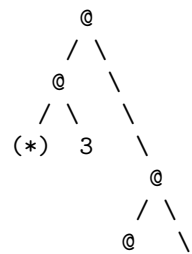
Step 1:

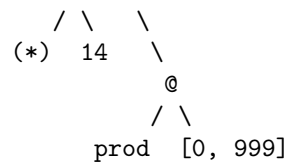
- vs A = fail
- vs B = fail
- vs C
  - x -> 3
  - xs -> [14, 0, 999]



Step 2:

- vs A = fail
- vs B = fail
- vs C
  - x -> 14
  - xs -> [0, 999]





etc.

## Proofs

### Induction

`prod (ms++ns) == prod ms * prod ns`

Induction is closely related to recursion. Look at where all the recursions are.

`[] ++ ys = ys`  
`(x:xs) ++ ys = x:(xs++ys)`

Induction on `xs`

- `xs = []`
- `x:xs`

**Base Case:**

`prod ([]++ys) = prod [] * prod ys`  
`= prod ys = 1 * prod ys`

Definition of `prod` and `++`

`= prod ys = prod ys`

**Induction Step:**

`prod ((x:xs)++ys) = prod (x:xs) * prod ys`  
`= prod (x:(xs++ys)) = x * prod xs * prod ys`  
`= x * prod (xs++ys) = x * (prod xs * prod ys)`

Apply your inductive hypothesis...

`x * prod (xs++ys) = x * prod (xs++ys)`

## 2016 b

Don't have to do the proof - state what is to be proved.

Induction on `ys`.

**Base Case:**

`mbr x (rem x []) == False`

**Step Case:**

`mbr x (rem x (y:ys)) == False`

given that `mbr x (rem x ys) == False`

**Case split:** Look at the case where `x==y` and `x!=y`.