1 Chapter 1

1. Consider the function $double x = x + x$. When evaluating $double(double 2)$ we quickly run into the question of what order to evaluate the functions in. The two examples Hutton give demonstrate applicative order and normal order respectively. Another possible way to look at it would be to consider an order of operations scenario, where all *double* functions must be evaluated before + operators. Then we get the following calculation:

$$\begin{aligned} double(double 2) &= double(2+2) \\ &= (2+2)+(2+2) \\ &= 4+(2+2) \\ &= 4+4 \\ &= 8 \end{aligned}$$

2. Consider the function $sum[x]$. We see that:

$$\begin{aligned} sum[x] &= sum x : [] \\ &= x + sum[] \\ &= x + 0 \\ &= x \end{aligned}$$

   Therefore, $sum[x] = x \ \forall \ x :: Num$.

3. Consider the *product* function:

$$\begin{aligned} product[] &= 1 \\ product(x : xs) &= x \cdot product xsa \end{aligned}$$

   We will use this definition to perform a calculation:

$$\begin{aligned} product[2,3,4] &= 2 \cdot product[3,4] \\ &= 2 \cdot 3 \cdot product[4] \\ &= 2 \cdot 3 \cdot 4 \cdot product[] \\ &= 2 \cdot 3 \cdot 4 \cdot 1 \\ &= 2 \cdot 3 \cdot 4 \\ &= 2 \cdot 12 \\ &= 24 \end{aligned}$$

4. We can define a quick sort function that sorts from largest to smallest with the following definition:

```
    qsortrev [] = []
qsortrev (x:xs) = qsortrev  larger ++ [x] ++ qsortrev  smaller
                          where
                             smaller = [a | a ← xs, a =< x]
                             larger = [b | b ← xs, b > x]
```

5. Consider the definition of *qsort* given by Hutton. If the $\leq$ were replaces with a $<$ then during the evaluation of *qsort* any other values in the list, $a \in xs \mid a = x$ will be dropped. We can see the result

of this change in the following calculation:

$$\begin{aligned} qsort[2,2,3,1,1] &= qsort[1,1] ++ [2] ++ qsort[3] \\ &= qsort[1,1] ++ [2] ++ [3] \\ &= [1] ++ [2] ++ [3] \\ &= [1] ++ [2,3] \\ &= [1,2,3] \end{aligned}$$

2 Chapter 2

1. - $2 \uparrow 3 * 4 = (2 \uparrow 3) * 4$
   - $2 * 3 + 4 * 5 = (2 * 3) + (4 * 5)$
   - $2 + 3 * 4 \uparrow 5 = 2 + (3 * (4 \uparrow 5))$

2. This is a good place to mention, Hutton wrote the book with the Hugs system in mind. Hugs is no longer maintained, as such I have been using GHCi. So far, this doesn't seem to be a problem. One thing to note, any multiline input must be between :{ and :}.

3. Here are the syntactic errors I found.
   - The N was capitalized.
   - The *div* was not surrounded by back ticks, i.e. '*div*'.
   - The xs was not in the same column as the a above it.

4. The first definition for last I came up with is this:
   ```
   last xs = xs!!((length xs)-1)
   ```
   However the intention is not very clear. So I came up with this:
   ```
   last xs = head (reverse xs)
   ```
   Lastly, we could use some pattern matching.
   ```
   last (x:[]) = x
   last (x:xs) = last xs
   ```

5. Similiar to last, here are two possible definitions for *init*.
   ```
   init1 xs = take ((length xs)-1) xs

   init2 xs = reverse (tail (reverse xs))
   ```

3 Chapter 3

1. ```
   ['a','b','c'] :: [Char]

   ('a','b','c') :: (Char, Char, Char)

   [ (False , '0'), (True , '1') ]  :: [(Bool, Char)]

   ([False,True],[   0    ,    1   ])  :: ([Bool],[Char])

   [ tail , init , reverse ] :: [[a]->[a]]
   ```
   ```

2. ```
   second :: [a]→a

   swap :: (a,b)→(b,a)

   pair :: a→b→(a,b)

   double :: Num a ⇒ a→a

   palindrome :: Eq a ⇒ [a]→Bool

   twice :: (a→a)→a→a
   ```

3. The above was checked using GHCi and, up to a variable name change, are correct.

4. To show that that two functions $f : A \to B$ and $g : A \to B$ are equal then we must prove that $f(x) = g(x)\ \forall\ x \in A$. This is not possible to do in general because A may not be a finite set. That is, to prove equality requires brute force unless there is more information available, so it would not be possible or practical unless A is small. So if $A = \{True, False\}$ then we could easily check for equality.

4 Chapter 4

1. My first attempt at the *halve* function:

   ```
   halve1 xs | even (length xs) = (take n xs, drop n xs) where n = (length xs) `div` 2
   ```

   Using a guard to check for even lists seems like the way to go, but I may revisit this question later.

2. ```
   safetail1 xs = if xs == [] then [] else tail where (_:tail) = xs

   safetail2 xs | xs == []  = []
                | otherwise = tail where (_:tail)=xs

   safetail3 []     = []
   safetail3 (x:xs) = xs
   ```

3. ```
   (||) :: Bool → Bool → Bool
   True  || True  = True
   True  || False = True
   False || True  = True
   False || False = False

   (||) :: Bool → Bool → Bool
   False || False = False
   _     ||    _  = True

   (||) :: Bool → Bool → Bool
   False || a    = a
   True  || True = True

   (||) :: Bool → Bool → Bool
   a    || False = a
   True || True  = True
   ```
   ```

4. ```haskell
   (&&) :: Bool → Bool → Bool
   a && b = if a == True then expr else False
            where
               expr = if b == True then True else False
   ```

5. It's possible that I misunderstood the previous question, since I only included one definition to avoid any patter matching. If we allow pattern matching against function definitions, but not patterns in the definition that we get:

   ```haskell
   (&&) :: Bool → Bool → Bool
   True && a  = if a == True then True else False
   False && _ = False
   ```

6. ```haskell
   mult = λx→(λy→(λz→x*y*z)))
   ```

5 Chapter 5

I took a long break before coming back to this. Sad face. Any who, this is a good place to mention that some of the functions Hutton assumes are available will actually need to be imported in order to work with prelude. I added the following import statement to make use of these functions.

```haskell
import Data.Char (ord, chr, isLower)
```

1. ```haskell
   sum [ x^2 | x ← [1..100]]
   ```

2. ```haskell
   replicate :: Int → a → [a]
   replicate n x = [ x | _←[1..n]]
   ```

3. ```haskell
   pyths :: Int → [(Int, Int, Int)]
   pyths n = [ (x,y,z) | x←[1..n], y←[1..n],
               let s = x^2 + y^2,
               let z = truncate (sqrt (fromIntegral s)),
               z*z == s, z ≤ n]
   ```

4. Here's one solution, that takes the easy approach.

   ```haskell
   perfects1 :: Int → [Int]
   perfects1 n = [i | i ← [1..n], sum (factors i) - i == i]
   ```

   Here's another, perhaps less efficient? approach using a filter and an anonymous function.

   ```haskell
   perfects2 :: Int → [Int]
   perfects2 n = [i | i ← [1..n], sum (filter (λx→x/=i) (factors i)) == i]
   ```

5. I may not have understood the question, but here's my answer.

   ```haskell
   [ x | x←zip (concat [replicate 3 i | i ← [1,2,3]]) (concat (replicate 3 [4,5,6]))]
   ```

6. ```haskell
   positions2 :: Eq a ⇒ a → [a] → [Int]
   positions2 a as = find a (zip as [1..])
   ```

7. ```haskell
   scalarproduct :: [Int] → [Int] → Int
   scalarproduct xs ys = sum [x*y | (x,y) ← zip xs ys]
   ```
   ```

8. I modified the let2int and the shift function. Since I did not have an expanded table with upper case frequencies, I converted all upper case characters to lower case characters before encoding, so that information is lost. I may revisit this so as to preserve case information while still using the original table.

```
let2int :: Char→Int
let2int c | isLower c = ord c - ord 'a'
          | isUpper c = let2int (toLower c)

shift :: Int→Char→Char
shift n c | isUpper c || isLower c = int2let ((let2int c + n) `mod` 26)
          | otherwise = c
```

6 Chapter 6

1. We could describe exponentiation as follows.

```
(^) :: Int → Int → Int
m^0 = 1
m^(n+1) = m * (m^n)
```

We will now show how this definition is used in evaluating 2^3.

$$\begin{aligned} 2^3 &= 2*(2^2) \\ &= 2*(2*(2^1)) \\ &= 2*(2*(2*(2^0))) \\ &= 2*(2*(2*(1))) \\ &= 2*(2*(2)) \\ &= 2*(4) \\ &= 8 \end{aligned}$$

2. We will evaluate length $[1, 2, 3]$

$$\begin{aligned} \text{length } [1,2,3] &= 1 + \text{length } [2,3] \\ &= 1 + (1 + \text{length } [3]) \\ &= 1 + (1 + (1 + \text{length } [])) \\ &= 1 + (1 + (1 + 0)) \\ &= 1 + (1 + (1)) \\ &= 1 + (2) \\ &= 3 \end{aligned}$$

We will evaluate drop $3[1, 2, 3, 4, 5]$

$$\begin{aligned} \text{drop } 3[1,2,3,4,5] &= \text{drop } 3[1,2,3,4,5] \\ &= \text{drop } 2[2,3,4,5] \\ &= \text{drop } 1[3,4,5] \\ &= \text{drop } 0[4,5] \\ &= [4,5] \end{aligned}$$

We will evaluate init $[1,2,3]$

$$\begin{aligned}\text{init } [1,2,3] &= 1 : (\text{init } [2,3]) \\ &= 1 : (2 : (\text{init } [3])) \\ &= 1 : (2 : ([])) \\ &= [1,2]\end{aligned}$$

3.
```
and :: [Bool] → Bool
and [] = True
and (False : _ ) = False
and (True : bs) = and bs

concat :: [[a]] → [a]
concat [] = []
concat (x:xs) = x ++ (concat xs)

replicate :: Int → a → [a]
replicate 0 _ = []
replicate n a = a:(replicate (n-1) a)

(!!) :: [a] → Int → a
(x:xs) !! 0 = x
(x:xs) !! n = xs !! (n-1)

elem :: Eq a ⇒ a → [a] → Bool
elem _ [] = False
elem a (x:xs) = if a==x then True else elem a xs
```

4.
```
merge :: Ord a ⇒ [a] → [a] → [a]
merge xs [] = xs
merge [] xs = xs
merge (x:xs) (y:ys) = if x ≤ y then x:(merge xs (y:ys)) else y:(merge ys (x:xs))
```

5.
```
halve :: [a] → ([a],[a])
halve xs = (take h xs, drop h xs)
            where
                h = (length xs) `div` 2

mergesort :: Ord a ⇒ [a] → [a]
mergesort []  = []
mergesort [a] = [a]
mergesort xs  = merge (mergesort h1) (mergesort h2) where (h1,h2) = halve xs
```

6.
```
sum :: Num a ⇒ [a] → a
sum [] = 0
sum (x:xs) = x + sum xs

take :: Int → [a] → [a]
take 0 _ = []
take n (x:xs) = x : (take (n-1) xs)

last :: [a] → a
last [a] = a
last (x:xs) = last xs
```

7 Chapter 7

1. We see that prob1 and prob2 are equivalent.

```
prob1 :: (a → b) → (a → Bool) → [a] → [b]
prob1 f p xs = [f x | x ← xs, p x]

prob2 :: (a → b) → (a → Bool) → [a] → [b]
prob2 f p xs = map f (filter p xs)
```

2. ```
all :: (a → Bool) → [a] → Bool
all p xs = and (map p xs)

any1 :: (a → Bool) → [a] → Bool
any1 p xs = or (map p xs)

takeWhile :: (a → Bool) → [a] → [a]
takeWhile _ [] = []
takeWhile p (x:xs) = if p x then x : takeWhile p xs else []

dropWhile :: (a → Bool) → [a] → [a]
dropWhile _ [] = []
dropWhile p (x:xs) = if p x then dropWhile p xs else (x:xs)
```

3. ```
map :: (a → b) → [a] → [b]
map f = foldr (λx xs → f x : xs) []

filter :: (a → Bool) → [a] → [a]
filter p = foldr (λx xs → if p x then x:xs else xs) []
```

4. ```
dec2int :: [Int] → Int
dec2int = foldl (λd x → x + 10*d) 0
```

5. As I understand it, the problem is that there is no such compose function, one that can take a list of functions and return a composition of all of them. The issue with this is in the type system: how would one describe the generalized type of such a function? I suspect this is possible with more advanced machinery, that can support errors, but the type system will not know for sure that any arbitrary list of functions is composable. Instead, we have to use the compose operator one at a time.

```
sumsqreven = sum.map (^2).filter even
```

6. ```
curry :: ((a,b) → c) → a → b → c
curry f = λx y → f (x,y)

uncurry :: (a → b → c) → (a,b) → c
uncurry f = λ (x,y) → f x y
```

7. ```
chop8 :: [Bit] → [[Bit]]
chop8 = unfold (null) (take 8) (drop 8)

map :: (a → b) → [a] → [b]
map f = unfold (null) (λxs → f (head xs)) tail

iterate :: (a → a) → a → [a]
iterate f = unfold (λx→False) id f
```
```

8. Here are the changes made. Notable, a function that prepends a parity bit, and function that checks and removes that bit, and new encode and decode functions that use the parity functions.

```
addparity :: [Bit] → [Bit]
addparity bs = (sum bs) ‘mod‘ 2 : bs

checkparity :: [Bit] → [Bit]
checkparity bs = if (sum bs) ‘mod‘ 2 == 0 then tail bs else error "Parity check failed..." bs

encodeP :: String → [Bit]
encodeP = concat.map (addparity.make8.int2bit.ord)

chop9 :: [Bit] → [[Bit]]
chop9 [] = []
chop9 bs = take 9 bs : chop9 (drop 9 bs)

decodeP :: [Bit] → String
decodeP = map (chr.bit2int.checkparity) ◦ chop9
```

9. Here's the channel stuff

```
channel :: [Bit] → [Bit]
channel = id

badchannel :: [Bit] → [Bit]
badchannel s = tail s
```

Here's some sample output.

```
Main> (decodeP.channel.encodeP) "abcde"
"abcde"
*Main> (decodeP.badchannel.encodeP) "abcde"
"λ176*** Exception: Parity check failed...
*Main> (decodeP.channel.encodeP) "bad?"
"bad?"
*Main> (decodeP.badchannel.encodeP) "bad?"
"λ177λ176*** Exception: Parity check failed...
```

8 Chapter 10

1.
```
mult :: Nat → Nat → Nat
mult Zero n     = Zero
mult (Succ m) n = add (mult m n) n
```

2. Here is our new definition of occurs. It is more efficient because it only does one comparison, instead of at most three.

```
occurs’ :: Ord a ⇒ a → Tree a → Bool
occurs’ a (Leaf b) = a == b
occurs’ a (Node b tl tr) = case compare a b of
                           EQ → True
                           LT → occurs’ a tl
                           GT → occurs’ a tr
```

3.
```
leafCount :: Btree → Int
leafCount (Bleaf _)     = 1
leafCount (Bnode t1 t2) = leafCount t1 + leafCount t2

isBalanced :: Btree → Bool
isBalanced (Bleaf _)     = True
isBalanced (Bnode t1 t2) = (leafCount t1 - leafCount t2 ≤ 1) && isBalanced t1 && isBalanced t2
```

4.
```
halve :: [a] → ([a],[a])
halve xs = (take n xs, drop n xs) where n = (length xs) ‘div‘ 2

balance :: [Int] → Btree
balance [a] = Bleaf a
balance xs  = Bnode (balance a) (balance b) where (a,b) = halve xs
```

5.
```
data Prop = Const Bool
          | Var Char
          | Not  n
          | And Prop Prop
          | Or Prop Prop
          | Imply Prop Prop
          | Equiv Prop Prop

eval :: Subst → Prop → Bool
eval _ (Const a)     = a
eval s (Var c)       = find c s
eval s (Not p)       = not (eval s p)
eval s (And p1 p2)   = (eval s p1) && (eval s p2)
eval s (Or p1 p2)    = (eval s p1) || (eval s p2)
eval s (Imply p1 p2) = (eval s p1) ≤ (eval s p2)
eval s (Equiv p1 p2) = (eval s p1) == (eval s p2)

findVars :: Prop → [Char]
findVars (Const _)     = []
findVars (Var c)       = [c]
findVars (Not p)       = findVars p
findVars (And p1 p2)   = findVars p1 ++ findVars p2
findVars (Or p1 p2)    = findVars p1 ++ findVars p2
findVars (Imply p1 p2) = findVars p1 ++ findVars p2
findVars (Equiv p1 p2) = findVars p1 ++ findVars p2
```

9 Chapter 12

5
```
pairSum :: Num a ⇒ (a, a) → a
pairSum (x, y) = x + y

fib :: [Integer]
fib = 0 : 1 : (map pairSum (zip fib (tail fib)))
```