Prelude.hs

Prelude.hs is the standard library which is loaded when you start Haskell, and it contains all the functions you have learned so far. Here's my pick of the most useful. Note that just knowing about these functions ain't gonna help you; knowing *how to use them* is the key! So, lots of practice is essential.

List processing basics

- (:) (list constructor)
- ++
- head
- last
- tail
- init
- length
- !! (list index)

Extra list processing

- maximum
- minimum
- reverse
- elem
- notElem
- concat
- take
- drop
- takeWhile
- dropWhile
- words
- unwords

Arithmetic

- div
- mod
- gcd
- lcm
- even
- odd
- sum
- product

Higher Order Functions
Map, fold and filter are so powerful, they've been put on their own page instead.

Tuples

- zip
- unzip
- fst
- snd

Other functions

- show
- read
- succ
- pred

List processing basics

No examples needed here I hope :)

1 - :
The basic list constructor. Adds an element to the front of a list.

2- ++
The list concatenator. Adds a list onto the front of another list. Make sure you understand the difference between : and ++

3- head
Returns the first element of a list.

4 - last
The opposite of head; last returns the last element of a list.

5 - tail

The tail of a list is everything except the first element. Returns an error if the list is empty.

6 - init
The opposite of tail. Given a list, init returns the list without the last element.

7 - length
Returns the length of the list

8 - !!
Returns the element of a list located at the specified index. Note that an 'index' starts counting from zero.

Extra List Processing

1 - maximum and minimum
These return the largest and smallest elements of a list repectively. Don't worry too much about the `Ord a` part of the type signature, but if you must know, then it means that this function only takes inputs of a Type which is `deriving Ord`. (1) That means that Haskell knows how to put these things in Order. (e.g. 1 < 2, 2 < 5, 'a' < 'z', False < True)

```
Prelude> :t maximum
maximum :: Ord a => [a] -> a
Prelude> :t minimum
minimum :: Ord a => [a] -> a

Prelude> maximum ['a'..'z']
'z'
Prelude> maximum [True, True, False]
True
Prelude> minimum ["ABCD", "ABCE", "ABC"]
"ABC"
```

2 - reverse
reverse takes a list and, um, reverses it

```
Prelude> :t reverse
reverse :: [a] -> [a]

Prelude> reverse [1..10]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
Prelude> reverse ['a'..'k']
"kjihgfedcba"
```

3 - elem and notElem
elem tells you if a specified element is in a list, and notElem is simply the opposite of that. Note that this only works for types which are `deriving Eq`, (2) meaning that Haskell knows how to say "that element is the same as this one".

```
Prelude> :t elem
elem :: Eq a => a -> [a] -> Bool
Prelude> :t notElem
notElem :: Eq a => a -> [a] -> Bool

Prelude> elem 'a' "milkshake"
True
Prelude> notElem 'a' "milkshake"
False
Prelude> elem "ABC" ["ABCD", "AB", "CBA"]
False
```

4 - concat
concat takes in a list of lists and combines them all into one list. (3)

```
Prelude> :t concat
concat :: MonadPlus a => [a b] -> a b
(Think of this as [[a]] -> [a]
Prelude> concat ["ABCD", "AB", "CBA"]
"ABCDABCBA"
```

5 - take and drop
`take n [a]` gives you the first n elements of the list. Likewise, `drop n [a]` gives you everything back *except* the first n elements of a list.

```
Prelude> :t take
take :: Int -> [a] -> [a]
Prelude> :t drop
drop :: Int -> [a] -> [a]

Prelude> take 4 "milk - legendary stuff"
"milk"
Prelude> drop 6 "milk - legendary stuff"
" legendary stuff"
Prelude> (take 4 string) ++ " is" ++ (drop 6 string) where string = "milk - legendary stuff"
"milk is legendary stuff"
```

(Yes, you can type 'where' clauses into Haskell!)

6 - takeWhile and dropWhile
takeWhile and dropWhile are similar to normal take and drop, but are more powerful - they take in a function (only certain types though) and uses it to 'test' elements of the list starting from the beginning. It will continue to take or drop elements from the list until an element fails the test.

The function you feed takeWhile or dropWhile must be of type `(a -> Bool)`. That is, it takes one input of any type and returns a Bool. Examples of valid functions are isVowel (which you wrote) or the inbuilt functions isUpper, isLower, isDigit, even, odd. Less obvious but most useful are (==) and (>) and (<).

```
Prelude> :t takeWhile
takeWhile :: (a -> Bool) -> [a] -> [a]
Prelude> :t dropWhile
dropWhile :: (a -> Bool) -> [a] -> [a]

Prelude> takeWhile (<=4) [1..10]
[1, 2, 3, 4]
Prelude> takeWhile (isDigit) "5984357one2three234"
"5984357"
Prelude> dropWhile (isDigit)  "5984357one2three234"
"one2three234"
```

7 - words and unwords
words takes in a single String and breaks it up wherever there are spaces, into a list of Strings. And unwords does the opposite.

```
Prelude> :t words
words :: String -> [String]
Prelude> :t unwords
unwords :: [String] -> String

Prelude> words "this is a cgi string after you decode it"
["this", "is", "a", "cgi", "string", "after", "you", "decode", "it"]
Prelude> unwords ["this", "is", "a", "cgi", "string", "after", "you", "decode", "it"]
"this is a cgi string after you decode it"
```

Arithmetic

1 - `div` and `mod`
`div` is how many whole times the first number can be divided by the second number. `mod` is the remainder after the first number is divided by the second number. Note that `div` and `mod` only process integers.(4)

```
Prelude> :t div
div :: Integral a => a -> a -> a
Prelude> :t mod
mod :: Integral a => a -> a -> a

Prelude> 17 `mod` 6
5
Prelude> 17 `div` 6
2
```

2 - gcd and lcm
gcd stands for Greatest Common Divisor of two numbers. lcm is the Lowest Common Multiple of two numbers. Useful for some Discrete Mathematics algorithms, but not much else.

```
Prelude> :t gcd
gcd :: Integral a => a -> a -> a
Prelude> :t lcm
lcm :: Integral a => a -> a -> a

Prelude> gcd 21 5
1
Prelude> gcd 21 14
7
Prelude> lcm 4 8
8
Prelude> lcm 12 8
24
```

3 - even and odd
How many times have you written isEven and isOdd, not realising that they were already inbulit?

```
Prelude> :t even
even :: Integral a => a -> Bool
Prelude> :t odd
odd :: Integral a => a -> Bool

Prelude> even 5
False
Prelude> odd 5
True
```

3 - sum and product
Simply add up or multiply all the contents of a list of numbers

```
Prelude> :t sum
sum :: Num a => [a] -> a
Prelude> :t product
product :: Num a => [a] -> a

Prelude> sum [1..10]
55
Prelude> product [1..4]
24
```

Higher Order functions (important!!)

1 - map
2 - filter
3 - fold
Map, fold and filter are so powerful, they've been put on their own page instead.

Tuples

1 - fst and snd
fst and snd take in a two-element tuple and return the first and second element of that tuple respectively. They have a major drawback in that they only work for two-part tuples. *HOWEVER* it's an excellent idea to write your own versions of fst and snd for larger tuples when needed, like part2of5, part3of5, part4of5 and so on.

```
Prelude> :t fst
fst :: (a,b) -> a
Prelude> :t snd
snd :: (a,b) -> b

Prelude> fst ("Star", "Craft")
"Star"
Prelude> snd ("Star", "Craft")
"Craft"
```

2 - zip and unzip
zip takes in two lists of (possibly different) things and turns them into a list of 2-element tuples. The first element of each of these tuples comes from the first list, and the second element comes from the second list. The length of this list equals the length of the smaller of the two lists. This is great for storing information which is related to another bit of data, such as Candidates and their number of Votes. And to reverse the process, you feed the tuples into unzip.

```
Prelude> :t zip
zip :: [a] -> [b] -> [(a,b)]
Prelude> :t unzip
unzip :: [(a,b)] -> ([a],[b])

Prelude> zip "ABCD" [1,6,3,23,12]
[('A', 1), ('B', 6), ('C', 3), ('D', 23)]
Prelude> zip "ABCD" [1,6,3,23]
[('A', 1), ('B', 6), ('C', 3), ('D', 23)]
Prelude> zip [1,6,3,23] "ABCD"
[(1, 'A'), (6, 'B'), (3, 'C'), (23, 'D')]
Prelude> maximum (zip [1,6,3,23] "ABCD")
(23, 'D')
Prelude> snd (maximum (zip [1,6,3,23] "ABCD"))
'D'
Prelude> unzip [(1, 'A'), (6, 'B'), (3, 'C'), (23, 'D')]
([1, 6, 3, 23], "ABCD")
```

Other functions

1 - show
show converts anything which has a Show function defined, into a String.

2 - read
read is the opposite of show; it reads in a String and converts it to whatever type you specify. Note the way I've used it below!

```
Prelude> read "2"
ERROR: Unresolved overloading
*** type : Read a => a
*** expression : read "2"
```

```
Prelude> read "2" :: Int
2
Prelude> read "2" :: Float
2.0
```

3 - succ and pred

For types with an Enum function defined, succ gives the successor and pred gives the predecessor.

```
Prelude> :t succ
succ :: Enum a => a -> a
Prelude> :t pred
pred :: Enum a => a -> a

Prelude> succ 2
3
Prelude> succ 20
21
Prelude> pred 21
20
Prelude> succ 'Z'
'['
Prelude> pred 'Z'
'Y'
Prelude> succ "Letters"
ERROR: [Char] is not an instance of class "Enum"
```

Footnotes

(1) This works for types deriving Ord (where Haskell works it out for you), or which have had Ord written by the user. Haskell normally puts elements in the order in which you typed them in. For example if you had

```
data TextNumber = One | Two | Four | Three
     deriving Ord
```

Then Haskell would think that One is less than Two, Two is less than Four, and Four is Less than Three. Solution: type them in properly or write your own Ord function.

(2) Likewise, you can let Haskell define Eq automatically (deriving Eq) or you can write your own. This is true for all classes (Eq, Ord, Enum, Show, Read ...)

(3) Monads are complicated.... and not studied in COMP1011.

(4) `div` and `mod` are examples of what are called "infix" operators. That is, you can put the function name between the inputs in forward quotes.
So `17 `div` 6 is` exactly the same as `div 17 6.` Use whichever you are more comfortable with (probably the infix version).

---

paull@cse.unsw.edu.au