# Chapter 1 Homework:

12/6/2010

**Q:** Give another possible calculation for the result of "double (double 2)". The two already given were:

- `double (double 2) ⇒`
  `double (2 + 2) ⇒`
  `double 4 ⇒`
  `4 + 4 ⇒`
  `8`

- `double (double 2) ⇒`
  `double 2 + double 2 ⇒`
  `(2 + 2) + double 2 ⇒`
  `4 + double 2 ⇒`
  `4 + (2 + 2)`
  `4 + 4 ⇒`
  `8`

**A:** This seems like kind of a lame answer, but there are not very many possibilities that aren't trivial. This is derived from the first way of calculating "double (double 2)" where the inner double is applied first.

```
double (double 2) ⇒
double (2+2) ⇒
(2+2) + (2+2) ⇒
4 + (2+2) ⇒
```

```
4 + 4 ⇒
8
```

Where the arrows represent the application of the inner double, the outer double, the first addition, the last addition and the middle addition respectively.

**Q:** Show that sum[x] = x for any number x.

**A:** Lets make up a definition for sum:

```
mysum :: Num a ⇒ [a] → a
mysum [] = 0
mysum [x] = x
mysum (x:xs) = x + mysum xs
```

In this definition it is clear from the pattern matching that mysum [x] = x HAS to be true in this definition, since without it it would never be able to pattern match the list of one element, and since this is a recursive definition would fail for all lists passed in (except the empty list) without this part of the definition. Obviously this must be true since in real life it would be like performing the identity function for addition on a number, which always results in the same number.

**Q:** Define a function "product" that produces the product of a list of numbers, and show using your definition that product [2,3,4] = 24.

**A:** MyProd

```
myprod :: Num a ⇒ [a] → a
myprod xs = foldr (∗) 1 xs
```

To show that it gets 24 from the input of [2,3,4] I will step through the program by hand:

```
myprod [2,3,4] ⇒
foldl (∗) 1 [2,3,4] ⇒
foldl (∗) (1 ∗ 2) [3,4] ⇒
foldl (∗) (1 ∗ 2 ∗ 3) [4] ⇒
foldl (∗) (1 ∗ 2 ∗ 3) [] ⇒
1 ∗ 2 ∗ 3 ∗ 4 ∗ 1 ⇒
24
```

Although I am mildly unsatisfied with this or other definitions that I came up with insofar as they always produce 1 for the result of passing the empty list (because that is the identity function for multiplication), though it seems to me like the result should actually be 0. The only way I can think of fixing that would be to use conditionals:

```
myprod2 :: Num a ⇒ [a] → a
myprod2 xs | xs == [] = 0
           | otherwise = foldl (∗) 1 xs
```

Which while it accomplishes the task at hand is neither elegant nor concise.

**Q:** How should the following definition of the function qsort be modified so that it produces a reverse sorted version of a list?

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

**A:** Now to reverse this we would want to still put x in the middle with a list before and after but the larger would have to come first, and the smaller second. Other than that I don't think any change is necessary.

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

**Q:** What would be the effect of replacing $\leq$ by $<$ in the definition of qsort?

**A:** I think that the result would be the simultaneous removal of duplicated elements anywhere in the list in addition to sorting.

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

Let's Test it out... I am correct!