In [1]:

```
import $file.hw3stdlib_new
import hw3stdlib_new._
```

Out[1]:

```
import $file.$

import hw3stdlib_new._
```

Homework 3

Due 9/22 at 11:59pm

Submission Instructions

Upload only this .ipynb file to Canvas. Do not add anything to hw2stdlib since you can't submit it.

In this homework we will develop more higher order functions and learn how to use fold.

See This link (https://www.notion.so/Guidelines-for-Programming-Homework-dbd25efa7bb24915ae6bcb06827fc5b6) for what is and isn't allowed in your code.

Problem 1 (5 Points)

Write the filter function. Remember that this should be polymorphic. So:

$$filter : (a \to \mathbb{B}) \to List\ a \to List\ a$$

Do not curry the parameters in the Scala version. If the predicate (The function we give it) is true for an element, then that element will be in the output list. If it's false leave it out.

In [2]:

```
// BEGIN SOLUTION
def filter[A](p : (A => Bool), xs : List[A]) : List[A] = xs match {
    case Empty        => Empty
    case Cons(x, xs) => p(x) match {
        case True  => Cons(x, filter(p, xs))
        case False => filter(p, xs)
    }
}
// END SOLUTION
```

Out[2]:

```
defined function filter
```

In [3]:

```
assert(filter((n: Nat) => lte(n, two), Cons(three, Cons(two, Empty))) ==
assert(filter((n: Nat) => lte(n, two), Empty) == Empty, 2)
assert(filter((n: Nat) => lte(n, four), Cons(three, Cons(two, Empty))) =
passed(4)
```

```
*** Tests Passed (4 points) ***
```

In [4]:

```
// HIDDEN TEST (1 pts)
// BEGIN HIDDEN TESTS
assert(filter((n: Nat) => lte(n, five), Cons(three, Cons(three, Cons(two
assert(filter((n: Nat) => lte(n, one), Cons(three, Cons(two, Empty))) ==
// END HIDDEN TESTS
```

Problem 2 (5 Points)

Implement the same filter function using a `fold`. Name it `filterWithFold`. (Hint, take a look at the `append` and / or `reverse` functions in the standard library).

In [5]:

```
// END SOLUTION
def filterWithFold[A](p : (A => Bool), xs : List[A]) : List[A] =
    reverse(fold((x: A, kept: List[A]) => p(x) match {
        case True  => Cons(x, kept)
        case False => kept
    }, Empty, xs))
// END SOLUTION
```

Out[5]:

```
defined function filterWithFold
```

In [6]:

```
assert(filterWithFold((n: Nat) => lte(n, two), Cons(three, Cons(two, Emp
assert(filterWithFold((n: Nat) => lte(n, two), Empty) == Empty, 2)
assert(filterWithFold((n: Nat) => lte(n, four), Cons(three, Cons(two, Em
passed(4)
```

```
*** Tests Passed (4 points) ***
```

In [7]:

```
// HIDDEN TEST (1 pts)
// BEGIN HIDDEN TESTS
assert(filterWithFold((n: Nat) => lte(n, five), Cons(three, Cons(three,
assert(filterWithFold((n: Nat) => lte(n, one), Cons(three, Cons(two, Emp
// END HIDDEN TESTS
```

Problem 3 (5 points)

Implement a function

$$ifThenElse : \mathbb{B} \to a \to a \to a$$

which chooses either the first A given if the bool is true or the second if it's false.

In [8]:

```
// BEGIN SOLUTION
def ifThenElse[A](test: Bool)(then: A)(otherwise: A): A = test match {
    case True => then
    case False => otherwise
}

// Alternatively:
def ifThenElse_alt[A](test: Bool): A => A => A =
    test match {
        case True => (then => otherwise => then)
        case False => (then => otherwise => otherwise)
    }
// END SOLUTION
```

Out[8]:

```
defined function ifThenElse
defined function ifThenElse_alt
```

In [9]:

```
assert(ifThenElse(True)(one)(two) == one)
assert(ifThenElse(False)(one)(two) == two)
passed(4)
```

```
*** Tests Passed (4 points) ***
```

In [10]:

```
// HIDDEN TEST (1 pts)
// BEGIN HIDDEN TESTS
assert(ifThenElse[Bool](True)(True)(False) == True)
assert(ifThenElse[Bool](False)(False)(True) == True)
// END HIDDEN TESTS
```

Problem 4 (5 points)

Implement the `Maybe` type:

$$Maybe\ a \ := \ \text{None} \ | \ \text{Just}\ a$$

Take a look at the definition of `List` in the stdlib as a starting point

In [11]:

```
sealed trait Maybe[+A]
case object None extends Maybe[Nothing]
case class Just[A](x : A) extends Maybe[A]
```

Out[11]:

```
defined trait Maybe
defined object None
defined class Just
```

In [12]:

```
val mx: Maybe[Nat] = None
val my = Just(three)
(None: Maybe[Nat]) match {
    case None => two
    case Just(n) => three
}
passed(4)
```

```
*** Tests Passed (4 points) ***
```

Out[12]:

```
mx: Maybe[Nat] = None
my: Just[Succ] = Just(Succ(Succ(Succ(Zero))))
res11_2: Succ = Succ(Succ(Zero))
```

In [13]:

```
// HIDDEN TEST (1 pts)
// BEGIN HIDDEN TESTS
val my = Just(Just(None))
// END HIDDEN TESTS
```

Out[13]:

```
my: Just[Just[None.type]] = Just(Just(None))
```

Problem 5 (5 points)

Implement

$$map : (a \to b) \to Maybe\ a \to Maybe\ b$$

Don't curry the function in the Scala implementation. Similarly to lists, it should return `None` if given `None` and should return `Just(f(value))` if it contains a value.

In [14]:

```
// BEGIN SOLUTION
def map[A,B](f : (A => B), mx : Maybe[A]) : Maybe[B] = mx match {
    case None => None
    case Just(x) => Just(f(x))
}
// END SOLUTION
```

Out[14]:

```
defined function map
```

In [15]:

```
assert(map(plus(_: Nat, four), None) == None)
assert(map(plus(_: Nat, four), Just(one)) == Just(five))
passed(4)
```

```
*** Tests Passed (4 points) ***
```

In [16]:

```
// HIDDEN TEST (1 pts)
// BEGIN HIDDEN TESTS
assert(map(plus(_: Nat, four), None) == None)
assert(map(map(plus(_: Nat, three), _: Maybe[Nat]), Just(Just(one))) == 
// END HIDDEN TESTS
```