Colin Gabrielson
CIS 425 Fall '14
Prof. Zena Ariola

Assignment 2

5.4)
a)

```
datatype 'a tree = LEAF of 'a | NODE of 'a tree * 'a tree;
fun f(x) = x + 1;

fun maptree f (LEAF(x)) = LEAF (f x)
|       maptree f (NODE(y,z)) = NODE (maptree f y, maptree f z);

val fir = NODE(NODE(LEAF 1,LEAF 2), LEAF 3);
val res = maptree f (fir);
```

function maptree takes in high order function f and in the first case when it's parameter is (LEAF(x)) it applies f to the value x in the leaf. Other wise it calls maptree on each subtree from the current NODE(y,z), and it passes along f.

b)
fn : ('a -> 'b) -> 'a tree -> 'b tree
The difference between this and the "expected type" are the 'b types.
This is because our function f(x), while it takes in x (of type 'a) it
could return another type, such as a char, so it is denoted 'b, making our tree a 'b tree

5.5)

```
datatype 'a tree = LEAF of 'a | NODE of 'a tree * 'a tree;
fun f(x : int, y : int) = x + y;

fun reduce f (LEAF(x)) = x
|       reduce f (NODE(y,z)) = f (reduce f y, reduce f z);

val fir = NODE(NODE(LEAF 1,LEAF 2), LEAF 3);
val res = reduce f (fir);
```

function reduce takes in a high order function f and in the first case returns the value within the leaf (f takes in a tuple containing two parameters, not one). In the second case, the high order function is called on the tuple of the subtrees at NODE(y,z), then we recursively call reduce on y and z, while passing f along.

5.6)

a)

```
fun f(x,y) = x + y;

fun curry f x y = f (x, y);
curry f 3 2;

fun z x y = x + y;

fun uncurry z (x,y) = z x y;
uncurry z (6,3);
```

b)

```
fun curry f x y = f (x, y);

fun uncurry z (x,y) = z x y;
```

uncurry(curry(f)) = f

first curry takes the function f, which is in the form f: ('a * 'b) -> 'c
more syntactically in the form: f x1 ... xn applying the curry function changes the signature to (x1, ...,xn). Then (x1, ... ,xn) is taken as a parameter for uncurry which produces x1 ... xn, which again is ('a * 'b) -> 'c, or f.

curry(uncurry(g)) = g

first uncurry takes the function g, which is in the form g: 'a -> ('b -> 'c)
more syntactically in the form: g(x1, ... ,xn) applying the uncurry function changes the signature to x1, ... ,xn. Then x1, ... ,xn is taken as a parameter for curry which produces (x1, ... ,xn), which again is 'a -> ('b -> 'c) or g.

5.7)

a)

if the if(...) statement returns false then x.i = 3 will not execute.
This means that the x.i variable will not be initialized to a value.
This causes whatever is in that memory location (junk) to continue to reside there. It is read as an int, and the run-time system will not catch the problem.

b)

The same bug cannot occur in this problem. The run time system catches the problem if "let val tag_int(m) ..." executes and "tag_int(3)" does not occur.
The message given is: "uncaught exception Bind [nonexhaustive binding failure]" which helps because it hints that a bind did not occur.

5.8

```
datatype Seq = Nil | Cons of int * (unit -> Seq)

fun head (Cons (x, _)) = x;

fun tail (Cons (_, xs)) = xs ();

(*fun ones = Cons(a, fn ( => )head();*)
```

a)

```
fun ones n = Cons(n, fn () => ones(n));
val alotof = ones 1;
```

b)

```
fun intList n = Cons(n, fn () => intList(n+1));
```

c)

```
fun takeN(0, _) = []
| takeN(_, Nil) = []
| takeN(i, Cons(n,xt)) = n :: takeN(i - 1, xt());

val lst =  intList 10;

takeN(4, lst);
```