Evaluation/Type checking [15 Points]

For each of the following expressions, give its type, or write "ill-typed" if it is not well-typed.

1. `fn x : bool => if x then ref x := false else ()` Type: ____________________

2. `fn x => if x then 0 else x + 1` Type: ____________________

3.
```
fn x => case x of
         (1.0, _) => 0
       | (x, y) => y x
```
Type: ____________________

For each of the following expressions, give its value:

1.
```
let
   fun return_if_found (v : int) (l : int list) =
         case l of
           nil    => 0
         | v :: _ => v
         | _ :: t => return_if_found v t
   in
    [return_if_found 1 [],
     return_if_found 1 [2, 3],
     return_if_found 1 [1, 2, 3]]
   end
```
Value: ______________

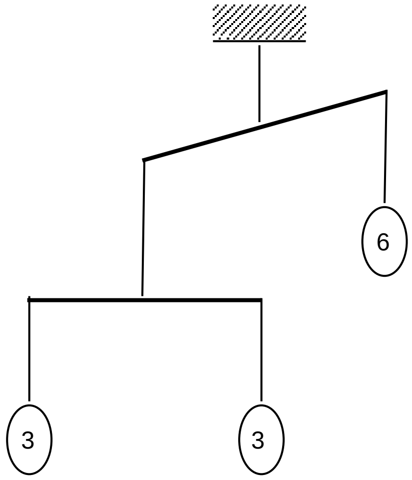
2.
```
let
   val rout = ref 1
   fun refadd () = let
                     val rin = ref 1
                   in
                     rin  := !rin + !rout;
                     rout := !rin + !rout
                   end
in
  refadd (refadd ());
  !rout
end
```
Value: ______________

Type Checking and Evaluation

1. `bool->unit`
2. *ill-typed*
3. `real * (real -> int) -> int`
4. `[0, 2, 1]`
5. `7`

Induction and Recursion [15 Points]

A *mobile* is a decorative construction built from beams, wires, and ornaments in the manner depicted below. At the end of each beam is a wire from which is hung either an ornament or another beam. The only significant property of an ornament is its weight, which is arbitrarily adjustable. A beam is attached by a wire fixed precisely at the middle of the beam. The beam itself, and the wire, are considered to be weightless. Thus the *weight* of a mobile is the sum of the weights of the ornaments attached to it. A mobile is *balanced* if the mobiles attached at the ends of each rod are of the same weight.



A mobile is represented in ML as a value of the following `datatype`:

```
datatype mobile = Ornament of real | Beam of mobile * real * mobile
```

An ornament is represented by its weight (a real number), and a beam is represented by the two mobiles attached at its ends, together with the total weight (a real number) of the mobile attached to that beam. Thus the `mobile` type satisfies the representation invariant that *the weight field associated with each beam is the sum of the weights of the two mobiles attached to that beam.*

In the following questions the representation invariant governing mobiles is assumed to hold on input and must be preserved on output.

1. Define a function `check_balance` of type `mobile -> true` which will check that the mobile is balanced.

2. Define a function `weight` which will return the total weight of a mobile.

3. Define a function `rebalance` of type `mobile -> mobile` which given a possibly unbalanced mobile will redistribute the weights among the ornaments so that the resulting mobile is balanced. *The total weight of the resulting mobile must be the same as that of the initial mobile.*

   Hint: Treat the weight attached to each beam of a mobile as the intended weight of the mobile at that beam. It is easy to derive the intended weights of the sub-mobiles attached to that beam from this. Write a helper function `redist` of type `real * mobile -> mobile` which given the intended final weight (a real) of a mobile and a possibly unbalanced mobile will return a balanced mobile with the given intended weight.

Induction and Recursion

```
datatype mobile = Beam of mobile * real * mobile | Ornament of real

fun check_balance_h (Ornament w) = SOME w
  | check_balance_h (Beam (m1, w, m2)) =
    case (check_balance_h m1, check_balance_h m2) of
         (SOME w1, SOME w2) => if w1 = w2 then SOME w else NONE
       | _                  => NONE

fun check_balance (mob : mobile) : bool = case check_balance_h mob of
                                                SOME _ => true
                                              | _ => false

fun weight (Ornament w) = w
  | weight (Beam(_, w, _)) = w

fun redist (intended, Ornament _ ) = Ornament intended
  | redist (intended, Beam(m1, w, m2)) =
    Beam(redist(intended / 2.0, m1),
         intended,
         redist(intended / 2.0, m2))

fun rebalance mob = redist (weight mob, mob)

val m1 = Beam (Beam(Ornament 1.0, 3.0, Ornament 2.0),
               9.0,
               Beam(Beam (Ornament 2.0, 3.0, Ornament 1.0),
                    6.0,
                    Ornament 3.0))
```

Higher-Order Functions [15 Points]

Consider the following data type of binary trees

```
datatype 'a bt = Lf of 'a | Nd of 'a bt * 'a * 'a bt
```

The items at the leaves and nodes of a tree are called the *elements* of the tree. The function `flatten : 'a bt -> 'a list` that returns a list of all the elements in the tree may be defined as follows:

```
fun flatten (Lf x) = [x]
  | flatten (Nd(l,x,r)) = flatten l @ (x :: flatten r)
```

Because of the use of the append (`@`) function on each recursive call, the running time of `flatten` is not linear in the number of elements of the tree.

1. What is the complexity of the function `flatten` defined above as a function of the number n of elements in the tree (express it in "big-O" notation). Explain your answer in one or two sentences.

2. Define another implementation of `flatten` that is linear in the number of elements in the tree. Proceed as follows. Define a function `fl : 'a bt -> 'a list -> 'a list` that, when applied to a tree t, yields a function that, when applied to a list l, appends the list of elements of t to l (so that the elements of l come after all the elements of t in the list). Thus the function `flatten` may be defined in terms of the function `fl` as follows:

```
fun flatten t = fl t []
```

Higher-Order Functions

1. $O(n^2)$ is the worst case complexity. The worst case is the degenerate case when the tree is completely unbalanced.

2.
```
fun fl (Lf x) = (fn xs => x::xs)
  | fl (Nd(l,x,r)) = (fn xs => (fl l) (x::fl r xs))
```

Data Abstraction [20 Points]

You are to implement the abstract data type of *integer multisets*. An integer multiset is similar to an integer set, except that a given element may occur more than once in the multiset. Thus a given multiset may have 2 copies of 17, one copy of 3, and no copies of any other integer.

A signature for the ADT of integer multisets is as follows:

```
signature MULTISETS =
  sig
    type multiset
    val empty     :  multiset
    val singleton :  int ->  multiset
    val union     :  multiset * multiset -> multiset
    val diff      :  multiset *  multiset ->  multiset
    val all_alike :  multiset -> bool
  end
```

The value `empty` is the multiset with no elements. The function `singleton` returns a multiset with one copy of the given element. The function `union` combines the elements of two multisets, adding the number of copies of each element occurring in either multiset. The function `diff` computes the difference of two multisets defined by striking out the elements in the second multiset from those in the first. If an element occurs more often in the first than the second, then the number of copies in the result is the difference between the two; otherwise, the element is removed entirely. The function `all_alike` checks whether the multiset consists of any number of copies of a single element.

You are to implement all operations *except* `union` according to the following guidelines. Represent multisets as lists of pairs `(`e`,` n`)`, where e is an integer element of the multiset and $n \geq 0$ is the count of the number of times that e occurs in the multiset, satisfying the following three invariants:

1. The pairs are sorted in non-decreasing order on their first element.
2. No two pairs in the list have the same first element.
3. The second element of every pair in the list is strictly greater than 0.

1. Give the representation type of the multiset:
   `type multiset =`
2. Give the value of `empty`:
   `val empty =`
3. Give the definition of `singleton`:
   `fun singleton e =`

4. Give the implementation of multiset difference:
   `fun diff (s1, s2) =`

5. Give the definition of `all_alike` in such a way that it runs in time independent of the size input multiset:

Data Abstraction

```
signature MULTISETS =
  sig
     type multiset
     val empty :  multiset
     val singleton :  int ->  multiset
     val union    :  multiset *  multiset ->  multiset
     val diff    :  multiset *  multiset ->  multiset
     val all_alike :  multiset -> bool
  end

structure MS : MULTISETS =
    struct
        type multiset = (int * int) list
        val empty = nil
        fun singleton e = [(e, 1)]
        fun union (nil : multiset, m) = m
          | union (m, nil) = m
          | union ((e, n) :: t , (e', n') :: t') =
            if e < e' then
                (e, n) :: union (t, (e', n') :: t')
            else
                if e > e' then
                    (e', n') :: union ((e, n) :: t, t')
                else  (* e = e' *)
                    (e, n + n') :: union (t, t')
        fun diff (nil : multiset, _) = nil
          | diff (m, nil) = m
          | diff ((e, n) :: t , (e', n') :: t') =
            if e < e' then
                (e, n) :: diff (t, (e', n') :: t')
            else
                if e > e' then
                    diff ((e, n) :: t, t')
                else  (* e = e' *)
                    if n <= n' then
                        diff (t, t')
                    else
                        (e, n - n') :: diff (t, t')
        fun all_alike [h] = true
          | all_alike _ = false
    end
```

Computability [15 Points]

Consider the problem of defining a function `Find` of type `(int -> bool) -> bool` which for an ML function p of type `int -> bool` (i.e., a predicate) returns true iff $p\,n$ evaluates to true for some $n \geq 0$.

Show that it *not* possible to define the function `Find` in ML in such a way that

1. `Find` p always returns either `true` or `false`, and
2. `Find` p evalustes to `true` iff there is some $n \geq 0$ such that $p\ n$ evaluates to `true`.

(Note that the obvious sequential search testing $p\ n$ for $n = 0, 1, 2, \ldots$ doesn't work as an implementation of `Find` because p may diverge on m even though there is an $n > m$ such that $p\ n$ evaluates to `true`.)

Hint: Show that if one indeed could write `Find` then one could write a function `H:(unit ->` τ `) -> bool`, where τ is a non-empty type, which is a decision procedure for the halting problem.

Computability

Let `tau` be an arbitrary an arbitrary non-empty type.

```
fun H (f: unit-> tau) : bool =
  let fun g(x: int) : bool = (f (); x = x)
  in
    Find g
  end
```

Assume `Find` is a decision procedure. Then

1. `H` either returns true or false
2. `H` returns true iff `f ()` halts, since `g x` returns true iff `f ()` halts).

Thus `H` would solve the halting problem (for supsensions), contradicting Theorem 2.1 in the Computability Notes.

Grammars [20 **Points**]

In the TEX typesetting system one uses ^ for superscript and _ for subscript in mathematical formulas. For example, the text `x^3` is formatted as x^3, and `x_i^2` is formatted as x_i^2. The parentheses (and) are used for grouping as usual. We here consider a fragment of this notation, defined by the following grammar with the start symbol e.

$$
\begin{array}{lcl}
e & \rightarrow & e \; \hat{} \; e \\
e & \rightarrow & e \; \_ \; e \\
e & \rightarrow & (\; e \;) \\
e & \rightarrow & a \\
a & \rightarrow & \mathtt{x} \\
a & \rightarrow & 0
\end{array}
$$

For simplicity we only allow one letter `x` and only one digit `0`.

1. Show that the above grammar is ambiguous by exhibiting two leftmost derivations for the string `x ^ x _ x`.

2. The above grammar is unsuitable for parsing because of the ambiguity and because it is left recursive. Rewrite the grammar to express that `^` and `_` are left associative with the same precedence so that **`x ^ x _ x`** parses as (**`x ^ x`**) **`_ x`**. Ensure that your grammar is not left recursive.

   Hint: proceed by considering the form of the strings that e generates and then write new grammar rules that generates the same set of strings.

Grammars

1. The string `x _ x _ x` has two leftmost derivations:

$$\begin{array}{rcl} e & \to & e \,\_\, e \\ & \to & x \,\_\, e \\ & \to & x \,\_\, e \,\_\, e \\ & \to & x \,\_\, x \,\_\, e \\ & \to & x \,\_\, x \,\_\, x \end{array}$$

and

$$\begin{array}{rcl} e & \to & e \,\_\, e \\ & \to & e \,\_\, e \,\_\, e \\ & \to & x \,\_\, e \,\_\, e \\ & \to & x \,\_\, x \,\_\, e \\ & \to & x \,\_\, x \,\_\, x \end{array}$$

2.

$$\begin{array}{rcl} e & \to & a \\ e & \to & a \; o \; e \\ o & \to & \uparrow \\ o & \to & \_ \\ a & \to & \mathtt{x} \mid (e) \\ a & \to & 0 \mid 1 \mid \cdots \mid 9 \end{array}$$