1. Trees

Red-black trees are a form of balanced binary search trees where each node has a colour: red or black, leaves are implicitly black; and the following properties hold:

Invariant 1 No red node has a red child;

Invariant 2 Every path from the root to a leaf has the same number of black nodes.

For these questions, you may assume we are implementing sets, i.e. interested whether an element exists or not, rather than a dictionary which maps keys to values.

1. Write a datatype to represent red-black trees, and a function to test if an element is present in the tree.

   [small]

2. We can implement insertion in two parts, first is finding the place to insert the element to, and inserting it not considering invariants; then fixing the tree so that the invariants hold.

   Hint 1 It is easier to fix up the invariants bottom-up (i.e. leaf first, then proceeding towards the root), because fixing the invariant at the current level might break the invariants at the level above.

   Hint 2 Which invariant we invalidate depends on what kind of node we insert: if we insert a red node then we invalidate invariant 1; if we insert a black node then we invalidate invariant 2. Out of these, inserting a red node is easier to fix up, as invariant 1 is a local property, whereas invariant 2 potentially needs to consider the whole (sub)tree. Code the insert function, you may assume that fixing the invariants is done by a function called `balance`, to be implemented in the next question.

   [medium, assuming I have given enough hints]

3. Code the `balance` function.

   Hint 1 When you fix up the current sub-tree, you have a choice of which invariant to invalidate for the parent sub-tree. It is easier if you only invalidate invariant 1 again.

   Hint 2 Related, it is easier to fix up a black node which has a red child which itself has a red child (invariant 1 invalidated), rather than fixing up a red node which has a red child (i.e. considering two nodes deep rather than just one node deep). This is because it allows keeping the number of black nodes the same in the sub-tree.

   Hint 3 We are invalidating invariant 1 in one place only, this can reduce the number of cases you need to consider.

   [medium]

2. Type-checking and inference

Type-checking and type-inference are very useful concepts, that aren't explicitly covered in the lectures, but they will help you work in OCaml much easier, as they give you an extra tool to check your programs for correctness, and types can even drive the solution to a problem. This section tries to go over some exercises bit-by-bit – do try to solve them without entering it into the computer.

1. What is the type of `let id x = x;;`? (Or more specifically, what is the type of `id` in the expression?)

   Hint The value `id` is a function returning its argument as-is.

2. What is the type of `let z = id 5;;`?

3. What is the type of `let f x y = x y;;`?

   Hint It might help to consider `f id 5`, which is a well-typed expression.

4. What is the type of `let f x y z = x y z;;`?
5. What is the type of `let f x y z = x y (z y);;`?
   Hint Consider `let g y z = z y;;` first
6. Why might `let f x = x x;;` not be well-typed?
7. If I define

   ```
   # type 'a ty = Wrapped of ('a ty -> 'a);;
   ```

   what is the type of the following expression?

   ```
   # let unwrap (Wrapped x) = x;;
   ```

8. What might go in the place of `todo` in the following expression: `let f x = unwrap todo x;;`? What is the type of `f`?
   Hint Make sure you have got the type of `unwrap` correct.
9. What is the type of `Wrapped f`?
10. What might go in the place of `todo` in `let g = todo (Wrapped f)`? Here `f` is the function defined previously. What is the type of `g`?
    Hint The value `todo` must be a function, given it is applied to a value `Wrapped f`. Look over your previous types.
11. What does the type of that expression tell us about the expression?
12. Can you think of another way of creating an expression of such type?

[each is tiny/small, but overall perhaps medium/big]

3. Full enumeration

Write a function that when given a list of colours, and a list of vertices, gives each possible assignment of colours to vertices. Colours and vertices need not be strings/integers, but for example:

```OCaml
all_pairings ["red"; "green"] [1; 2] =
[[("red", 1); ("red", 2)]; [("red", 1); ("green", 2)];
 [("green", 1); ("red", 2)]; [("green", 1); ("green", 2)]]
```

Note, the order of the outer list doesn't matter.

[medium]