UCLA CS 131 Midterm, Spring 2022
100 minutes total, open book, open notes, closed computer
Write answers on exam.

Name: Jeffrey Loewe Student ID: 105546173

1. Suppose we define the following type for binary trees, where each leaf has a label of type 'a and each internal node has a label of type 'b:

left right?

```
type ('a, 'b) bintree =
  | Bleaf of 'a
  | Bnode of 'b * ('a, 'b) bintree * ('a, 'b) bintree
```

1a (8 minutes). What is the type of the following function 'mystery'? Briefly explain how you inferred its type.

(some node or leaf)

```
let rec mystery w x = function
  | Bleaf(_) -> None
  | Bnode(y, a, b) ->
    match w x y with
      | 0 -> Some y
      | n -> mystery w x (if n < 0 then a else b)
```

some

Mystery is type int option. This is because of w x y, we know w allows for a ~~com~~ comparison ~~computation~~ between x and y, meaning they must be the same type. Then, because of the condition n<0, we know n is int because of the integer comparator <. Thus, we know mystery is type int option.

1b (8 minutes). What does 'mystery' do? Give a sample call to illustrate.

~~mystery finds the node with value 0 label x that matches label of~~

~~there is a~~

Mystery finds if there is a node with label y that equals x ~~for instance~~ after w is applied to x.

~~mystery will look~~

For instance, mystery _ x ~~[illegible]~~ some_tree finds a node in some_tree with label that was diff of 0 from x.

1c (12 minutes). Write a function 'flatten' that takes as an argument a binary tree and returns a list of labels on the tree, in left-to-right order so that all the labels of the left subtree of a node appear before the node's label, and all the labels of the right subtree appear after the node's label.

The function should take a bintree argument and return a binvisit list, where bintree is defined in (a) and binvisit is defined by:

```
type ('a, 'b) binvisit = | Vleaf of 'a | Vnode of 'b
```

For example, the expression:

```
flatten (Bnode ("a",
           Bnode ("b", Bleaf 1, Bnode ("c", Bleaf 2, Bleaf 3)),
           Bnode ("d",
              Bnode ("e", Bleaf 4, Bleaf 5),
              Bnode ("f", Bnode ("g", Bleaf 6, Bleaf 7), Bleaf 8))))
```

should return:

```
[Vleaf 1; Vnode "b"; Vleaf 2; Vnode "c"; Vleaf 3; Vnode "a"; Vleaf 4;
 Vnode "e"; Vleaf 5; Vnode "d"; Vleaf 6; Vnode "g"; Vleaf 7;
 Vnode "f"; Vleaf 8]
```

When writing 'flatten' do not use the OCaml standard library; just use builtin types and operators such as '::' and '@'.

let rec flatten ~~bintree~~ function =
| Bleaf (val) -> ~~[Bleaf]~~ [Vleaf val]
| Bnode(~~y~~ , ~~a~~ , ~~b~~) -> ~~(a @ [y] @ b);;~~
label left right
((left @ [Vnode label]) @ right);;

let rec flatten function =
| Bleaf(label) -> [Vleaf val]
| Bnode(label, left, right) -> ((flatten left)@
[Vnode label]) @ (flatten right);;

1d (10 minutes). Write a grammar 'flatten_grammar' in the style of Homework 2 that describes the values returned by the 'flatten' function on bintrees where all leaf values are the integers 1 through 8, and all node values are the strings "a" through "g".

let flatten_grammar =
~~(Node,~~ (VNode, (BNode,
~~function~~ function
~~INode->~~ | VNode ->
[1-8];

On back page

1e (10 minutes). Assume you have the aforementioned grammar 'flatten_grammar' and a solution to Homework 2, how would you go about writing a function 'unflatten' that is the inverse of 'flatten'? That is, (unflatten (flatten X)) should equal X. You need not implement 'unflatten', just describe how you would implement it if you had more time.

GO
~~Iterate throught throught list~~
~~of rules to find~~
of Vnodes,
To unflatten, we would be building
a tree so we iterate through
~~flatten grammar~~ the list and dfs
through flatten grammar to find
a matching derivation. If there
full
none, return None, otherwise build
the tree using the same methodology
as parse ~~through~~ tree through
using a list of applied rules from
flatten_grammar.

2 (10 minutes). If a Java program does an exit-monitor operation E followed by a normal-load operation L, a Java implementation can do L before E. However, the reverse is not true: if a Java program does a normal load L before an exit monitor E, the implementation cannot do E before L. Explain why this is, giving an example of reordering that is OK and why it is OK, and reordering that is not OK and why it is not OK.

OK ordering

Shape a = new shape();
Shape b = new shape();
int l = 6;

a.y = l
l = b.x
these can be switched

This ordering is okay because with optimizations there is no difference in results between the two orders

NOT OK ordering

Shape a = new shape();
Shape b = new shape();
int l = 6;

l = b.x
a.y = l
cannot be switched

This ordering is not okay because the load changes l, meaning a volatile store is invalid when LE->EL

3 (4 minutes). Which role of types is more important in OCaml: type annotation or type inference? Briefly explain.

Type inference is more important in OCaml since what seems to be function invocations are treated as a generic type and the code that you provide via aspects like pattern-matching means you can typically avoid specifying types in code since it can be inferred from your code via rules

4 (14 minutes). Write a simple, unambiguous grammar for OCaml that specifies only the following OCaml constructs.

Expressions (or patterns) that contain:

* identifiers
* literal integer constants
* the empty list
* the '::' constructor
* parenthesized subexpressions (or subpatterns)

Also, expressions that contain:

* function definitions using the 'function' keyword and with '|' immediately after 'function'
* function calls

Your grammar should behave the way OCaml does; for example, (a b c::d::e) should be equivalent to (((a b) c)::(d::e)).

expression = type

[]

::= []

::= [type] "::" [type]

digit = 0|1|2|3|4|5|6|7|8|9

type = int | float | string | bool ...

parens = "(expn)"

5a (4 minutes). Briefly explain any extra work you had to do to make the grammar unambiguous.

I ~~had to~~ ~~[illegible]~~ would have to handle associativity via extra rules

5b (10 minutes). Diagram your grammar, with boxes around nonterminals and nothing around terminals. Keep your diagram as simple as possible.

6. OCaml lacks support for shared memory parallel programming. Suppose we fix this, by adding the following concurrency primitives to the subset of OCaml studied in class:

```
type 'a thread (* The type of threads that eventually return
                  values of type 'a.  *)

val create : ('a -> 'b) -> 'a -> 'b thread
   (* (create func arg) creates a new thread that executes
      (func arg) concurrently with the rest of the program.
      The thread terminates when 'func' returns a value.  *)

val wait : 'a thread -> 'a
   (* 'wait t' suspends execution of the calling thread until the
      thread 't' terminates; it then returns the value that t's
      function returned.  *)
```

6a (3 minutes). Give an example trivial use of these concurrency primitives, by multiplying 3*5 in one new thread, and 7*9 in another, and then adding the results.

wait(create (fun x -> 3*x) 5) +
wait(create (fun x -> 7*x) 9);;

6b (7 minutes). What would need to go into an OCaml Memory Model (OMM) that is like the Java Memory Model (JMM) except tailored for this variant of OCaml? Briefly explain the differences between the JMM and your OMM. Or if it's not feasible to design an OMM this way, explain why not.

There would be no garbage collection since every variable would be used

~~In addition~~

let flatten_grammar =
(BNode,
| BNode -> [[~~Bleaf~~ Vleaf Bleaf ~~3~~]; [VNode BNode];
[Vleaf Bleaf];
[Vnode BNode; Vnode ; Bnode; Vnode];
[Vleaf Bleaf; Vnode Bnode; Vnode Bnode];
[Vnode BNode; Vnode; Bnode; Bnode];
[Vnode "a"]... [Vnode "b"]]; Vleaf Bleaf]]
| ~~VNode~~ BLeaf ->
[[V~~Node~~Leaf 1] ~~2~~]; [V~~Node~~Leaf 2]; [V~~Node~~Leaf 3];
...[V~~Node~~Leaf 8]]