Here are some extra programming problems that can be done using the material in this module. Many are similar in difficulty and content to the homework, but they are not the homework, so you are free to discuss solutions, etc. on the discussion forum. Thanks to Pavel Lepin and Charilaos Skiadas for contributing most of these.

Racket structs:

To practice a little with structs, we can implement binary trees, with practice problems similar to those from Section 2 and Section 3, but of course in Racket instead of ML. Use the following definitions:

```
(struct btree-leaf () #:transparent)
(struct btree-node (value left right) #:transparent)
```

A binary tree is either (btree-leaf) or or a Racket value built from btree-node where the left and right fields are both binary trees.

- Write a function tree-height that accepts a binary tree and evaluates to a height of this tree. The height of a tree is the length of the longest path to a leaf. Thus the height of a leaf is 0.
- Write a function sum-tree that takes binary tree and sums all the values in all the nodes. (Assume the value fields all hold numbers, i.e., values that you can pass to +.
- Write a function prune-at-v that takes a binary tree t and a value v and produces a new binary tree with structure the same as t except any node with value equal to v (use Racket's equal?) is replaced (along with all its descendants) by a leaf.
- Write a function well-formed-tree? that takes any value and returns #t if and only if the value is legal binary tree as defined above.
- Write a function fold-tree that takes a two-argument function, an initial accumulator, and a binary tree and implements a fold over the tree, applying the function to all the values. For example, (fold-tree (lambda (x y) (+ x y 1)) 7 (btree-node 4 (btree-node 5 (btree-leaf) (btree-leaf)) (btree-leaf))) would evaluate to 18. You can traverse the tree in any order you like (though it does affect the result of a call to fold-tree if the function passed isn't associative).
- Reimplement fold-tree as a curried function.

Dynamic typing:

- Write a function crazy-sum that takes a list of numbers and adds them all together, starting from the left. There's a twist, however. The list is allowed to contain functions in addition to numbers. Whenever an element of a list is a function, you should start using it to combine all the following numbers in a list instead of +. You may assume that the list is non-empty and contains only numbers and binary functions suitable for operating on two numbers. Further assume the first list element is a number. For example, (crazy-sum (list 10 * 6 / 5 - 3)) evaluates to 9. Note: It may superficially look like the function implements infix syntax for arithmetic expressions, but that's not really the case.

- Write a function either-fold that is like fold for lists or binary trees as defined above except that it works for both of them. Give an appropriate error message if the third argument to either-fold is neither a list nor a binary tree.
- Write a function flatten that takes a list and flattens its internal structure, merging all the lists inside into a single flat list. This should work for lists nested to arbitrary depth. For example, (flatten (list 1 2 (list (list 3 4) 5 (list (list 6) 7 8)) 9 (list 10))) should evaluate to (list 1 2 3 4 5 6 7 8 9 10).

Using lambda-calculus ideas to remove features from MUPL programs:

Like Racket itself, MUPL (the programming language from this section's homework assignment) is essentially a superset of untyped lambda calculus. "Lambda calculus" may sound scary, but it's essentially a *very* simple programming language -- it really doesn't have anything in it, apart from anonymous functions and function calls! Of course, that makes it very inconvenient to program in, which is also why real programming languages usually supply all sorts of bells and whistles, like additional language constructs and data types like booleans and numbers.

Nonetheless, untyped lambda calculus is Turing-complete, so we can actually represent things like numbers and booleans using nothing but functions. In these problems we'll do some of that in MUPL.

- Notice that MUPL doesn't need mlet: Anywhere we have (mlet name e body), we can use (call (fun #f name body) e) instead and get the same result. Write a Racket function remove-lets that takes a MUPL program and produces an equivalent MUPL program that does not contain mlet.
- [more challenging] Now we will do something even more clever: remove pairs by using closure environments as another way to "hold" two pieces of data. Instead of using, (apair e1 e2), we can use (mlet "_x" e1 (mlet "_y" e2 (fun #f "_f" (call (call (var "_f") (var "_x")) (var "_y"))))) (assuming "_x" isn't already used in e2 -- we will assume that). This will evaluate to a closure that has the result of evaluating e1 and e2 in its environment. When the closure is called with a function, that function will be called with the result of evaluating e1 and e2 (in curried form). So if we replace every apair expression as described above, then we can, rather cleverly, replace (fst e) with (call e (fun #f "x" (fun #f "y" (var "x")))). Extend your remove-lets, renaming it remove-lets-and-pairs so that it removes all uses of apair, fst, and snd. (We are leaving it to you to figure out how to replace (snd e). Note 1: Remember you need to remove things recursively inside of apair, fst, etc., else an expression like (fst (snd (var "x"))) won't have the snd removed. Note 2: The resulting program should produce the same result when evaluated if the (original) result doesn't contain any pair values. If the original result does contain pair values, the result after removal will contain corresponding closures. Note 3: A slightly more challenging approach is to change how apair is removed so that we do not need to assume "_y" is not used in e2.

More MUPL functions:

In the first problem, we treat (int 1) as true in MUPL and (int 0) as false in MUPL.

- Define a Racket binding mupl-all that holds a MUPL function that takes a MUPL list and evaluates to (MUPL) true if all the list elements are (MUPL) true, else it evaluates to (MUPL) false.
- Define a Racket binding mupl-append that holds a MUPL function that takes two MUPL lists (in curried form) and appends them.
- Define a Racket binding mupl-zip that holds a MUPL function that takes two MUPL lists (in curried form) and returns a list of pairs (much like ML's zip). If the MUPL lists are different lengths, ignore a suffix of the longer list (so the returned list of pairs has a length equal to the shorter of the argument lengths).
- Redo the previous two problems with the MUPL functions taking a pair with the arguments rather than

using currying.

- Define a Racket binding mupl-curry that holds a MUPL function that is like ML's fun curry f = (fn x y => f (x,y)).
- Define a Racket binding mupl-uncurry that holds a MUPL function that is like ML's fun uncurry f = (fn (x,y) => f x y).

More MUPL macros:

As above, as needed, we treat (int 1) as true in MUPL and (int 0) as false in MUPL.

- Define a Racket binding if-greater3 that is a MUPL macro (a Racket function) that takes 5 MUPL expressions and produces a MUPL program that, when evaluated, evaluates the 4th subpexression as the result if the 1st subexpression is greater than the 2nd and the 2nd is greater than the 3rd, else it evaluates the 5th subexpression as the result. When the MUPL program is evaluated, it should always evaluate the 1st, 2nd, and 3rd subexpressions exactly once each and then the 4th subexpression or 5th subexpression but not both.
- Define a Racket binding call-curried that is a MUPL macro (a Racket function) that takes a MUPL expression e1 and a *Racket* list of MUPL expressions e2 and produces a MUPL program that, when evaluated, calls the result of evaluating e1 as a curried function with all of the results of evaluating the expressions in e2. For example, instead of writing (call (call e1 ea) eb), you can write (call-curried e1 (list ea eb)).

完成