Practice Problems for Section 6

NOTE: You do not need to handle corner cases, particularly receiving arguments of the wrong type, unless explicitly asked to do that in the problem statement.

The Usual Suspects

To practice a little with structs, we'll translate the definition of datatype for binary trees from Section 2's **Forest For The Trees** problem to Racket. Use the following definitions:

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

Write functions `btree-fold` and `btree-unfold` that implement folding and unfolding on these data structures. These functions should behave as described in Section 3 problem **Deeper Into The Woods**.

The first argument to `btree-fold` should be a function that accepts the result of the recursive call over the left tree as the first argument, value of node as the second argument, and the result of the recursive call over the right tree as the second argument. The second argument to `btree-fold` should be the initial value for leaves, and the third should be the tree to fold over.

The first argument to `btree-unfold` should be a function that takes the current state as its argument and produces either `#f` (to generate a leaf), or a cons cell, such that its `car` is the value for the new node, and the `cdr` is another cons cell containing states for recursive calls for left and right subtrees in `car` and `cdr` correspondingly.

Now use one of the functions you just implemented to write a function called `gardener`. `gardener` should take a single argument, a binary tree, and evaluate to a new binary tree, similar to the original one, but with all the nodes the value of which is `#f` pruned together with all their descendants. The behavior is similar (but not identical) to the `gardener` function from the **Forest For The Trees − 3** problem in Section 2.

So Dynamic

Crazy Sum

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, that it contains only numbers and binary functions suitable for operating on two numbers. First element of the given list will always be a number rather than a function. Consider all the functions used to be associative.

HINT: Check out the `procedure?` predicate.

EXAMPLE: `(crazy-sum (list 10 * 6 / 5 - 3))` evaluates to `9`

NOTE: While it may superficially look like the function implements infix syntax for arithmetic expressions, that's not really the case.

Universal Fold

Write a function `universal-fold` that takes a function, an initial value and a data structure. `universal-fold` should behave like `foldr` if the data structure passed to it is a list, and as a `btree-fold` otherwise. Write a function `universal-sum` that sums all the elements contained in a given data structure in terms of `universal-fold`.

Stomp!

Write a function `flatten` that accepts 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.

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)`

Lambda Madness

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 really just a *very* simple programming language – it doesn't have anything in it, apart from function expressions and function application! 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 this problem we'll do some of that in MUPL. We'll leave numbers and `aunit` alone, but we'll express `mlet` and MUPL pairs in terms of functions. Namely:

1. `(mlet name e body)` can be equivalently expressed as `(call (fun #f name body) e)`.

2. What about pairs? Closures actually carry some data with them, so we can keep elements of pairs inside closures. `(apair e1 e2)` would become `(fun #f "_f" (call (call (var "_f") e1) e2))`.

Try to figure out how does this work, and what should the definitions for `fst` and `snd` be. If you need to introduce any names, use `_x` and `_y` (with `_x` introduced first). You shouldn't need more than two names. Note that we're being unhygienic, be we're not going to bother generating safe names instead.

Write a function `simplify` that accepts a MUPL program as an argument and produces an equivalent program without `mlet`, `apair`, `fst` or `snd`.

NOTE: You'll need to copy the definitions of structs for representing MUPL programs from the code provided in this section's homework assignment.