Overview: This homework mostly has to do with MUPL (a Made Up Programming Language). MUPL programs are written directly in Racket by using the constructors defined by the structs defined at the beginning of hw5.rkt. This is the definition of MUPL’s syntax:

- If $s$ is a Racket string, then `(var $s$)` is a MUPL expression (a variable use).
- If $n$ is a Racket integer, then `(int $n$)` is a MUPL expression (a constant).
- `(mtrue)` is a MUPL expression representing boolean truth. Notice `(mtrue)` is a MUPL expression, but `mtrue` is not.
- `(mfalse)` is a MUPL expression representing boolean falsity. Notice `(mfalse)` is a MUPL expression, but `mfalse` is not.
- If $e_1$ and $e_2$ are MUPL expressions, then `(add $e_1$ $e_2$)` is a MUPL expression (an addition).
- If $s_1$ and $s_2$ are Racket strings and $e$ is a MUPL expression, then `(fun $s_1$ $s_2$ $e$)` is a MUPL expression (a function). In $e$, $s_1$ is bound to the function itself (for recursion) and $s_2$ is bound to the (one) argument. Also, `(fun null $s_2$ $e$)` is allowed for anonymous nonrecursive functions.
- If $e_1$ and $e_2$ are MUPL expressions, then `(isgreater $e_1$ $e_2$)` is a MUPL expression (a comparison).
- If $e_1$, $e_2$, and $e_3$ are MUPL expressions, then `(mif $e_1$ $e_2$ $e_3$)` is a MUPL expression. It is a condition where the result is $e_3$ if $e_1$ is false else the result is $e_2$. Only one of $e_2$ and $e_3$ is evaluated.
- If $e_1$ and $e_2$ are MUPL expressions, then `(call $e_1$ $e_2$)` is a MUPL expression (a function call).
- If $s$ is a Racket string and $e_1$ and $e_2$ are MUPL expressions, then `(mlet $s$ $e_1$ $e_2$)` is a MUPL expression (a `let` expression where the value resulting from evaluating $e_1$ is bound to $s$ in the evaluation of $e_2$).
- If $e_1$ and $e_2$ are MUPL expressions, then `(apair $e_1$ $e_2$)` is a MUPL expression (a pair-creator).
- If $e_1$ is a MUPL expression, then `(first $e_1$)` is a MUPL expression (getting the first part of a pair).
- If $e_1$ is a MUPL expression, then `(second $e_1$)` is a MUPL expression (getting the second part of a pair).
- `(munit)` is a MUPL expression (holding no data, much like `()` in ML or `null` in Racket). Notice `(munit)` is a MUPL expression, but `munit` is not.
- If $e_1$ is a MUPL expression, then `(ismunit $e_1$)` is a MUPL expression (testing for `(munit)`).
- `(closure $env$ $f$)` is a MUPL value where $f$ is MUPL function (an expression made from `fun`) and $env$ is an environment mapping variables to values. Closures do not appear in source programs; they result from evaluating functions.

A MUPL value is a MUPL integer constant, MUPL true, MUPL false, a MUPL closure, a MUPL munit, or a MUPL pair of MUPL values. Similar to Racket, we can build list values out of nested pair values that end with a MUPL munit. Such a MUPL value is called a MUPL list.
You should assume MUPL programs are syntactically correct (e.g., do not worry about wrong things like (int "hi") or (int (int 37)). But do not assume MUPL programs are free of type errors like (add (munit) (int 7)) or (first (int 7)).

Warning: What makes this assignment challenging is that you have to understand MUPL well and debugging an interpreter is an acquired skill.

Turn-in Instructions: Turn in your modified hw5.rkt and hw5tests.rkt using the Catalyst dropbox link on the course website.

Problems:

1. Macro Practice: Define a (Racket) macro that is used like (while-greater e1 do e2) where e1 and e2 are expressions and while-greater and do are syntax (keywords). The macro should do the following:
   - It evaluates e1 exactly once.
   - It evaluates e2 at least once.
   - It keeps evaluating e2 until and only until the result is not a number greater than the result of the evaluation of e1.
   - Assuming evaluation terminates, the result is #t.
   - Assume e1 and e2 produce numbers; your macro can do anything or fail mysteriously otherwise.

   Hint: Define and use a recursive thunk. Sample solution is 9 lines. Example:
   (define a 7)
   (while-greater 2 do (begin (set! a (- a 1)) (print "x") a))
   (while-greater 2 do (begin (set! a (- a 1)) (print "x") a))

   Evaluating the second line will print "x" 5 times and change a to be 2. So evaluating the third line will print "x" 1 time and change a to be 1.

2. MUPL Warm-Up:

   (a) Write a Racket function racketlist->mupllist that takes a Racket list (presumably of MUPL values but that will not affect your solution) and produces an analogous MUPL list with the same elements in the same order.

   (b) Write a Racket function mupllist->racketlist that takes a MUPL list (presumably of MUPL values but that will not affect your solution) and produces an analogous Racket list (of MUPL values) with the same elements in the same order.
3. **Implementing the MUPL Language:** Write a MUPL interpreter, i.e., a Racket function `eval-exp` that takes a MUPL expression `e` and either returns the MUPL value that `e` evaluates to under the empty environment or calls Racket’s error if evaluation encounters a run-time MUPL type error or unbound MUPL variable.

A MUPL expression is evaluated under an environment (for evaluating variables, as usual). In your interpreter, use a Racket list of Racket pairs to represent this environment (which is initially empty) so that you can use **without modification** the provided `envlookup` function. Here is a description of the semantics of MUPL expressions:

- All values (including closures) evaluate to themselves. For example, `(eval-exp (int 17))` would return `(int 17)`, *not* 17.
- A variable evaluates to the value associated with it in the environment.
- An addition evaluates its subexpressions and, assuming they both produce integers, produces the integer that is their sum. (Note this case is done for you to get you pointed in the right direction.)
- An `isgreater` evaluates its two subexpressions to values `v_1` and `v_2` respectively. If both values are integers, then if `v_1 > v_2` the result of the `isgreater` expression is the MUPL value `(mtrue)`, else the result is the MUPL value `(mfalse)`.
- A `mif` evaluates its first expression to a value `v_1`. If that value is false, then `mif` evaluates its its third subexpression, else it evaluates its second subexpression.
- Functions are lexically scoped: A function evaluates to a closure holding the function and the current environment.
- An `mlet` evaluates its first expression to a value `v`. Then it evaluates the second expression to a value, in an environment extended to map the name in the `mlet` expression to `v`.
- A `call` evaluates its first and second subexpressions to values. If the first is not a closure, it is an error. Else, it evaluates the closure’s function’s body in the closure’s environment extended to map the function’s name to the closure (unless the name field is `null`) and the function’s argument-name (i.e., the parameter name) to the result of the second subexpression.
- An `apair` evaluates its two subexpressions and produces a (new) pair holding the results.
- A `first` evaluates its subexpression. If the result for the subexpression is a pair, then the result for the first expression is the `e1` field in the pair.
- A `second` evaluates its subexpression. If the result for the subexpression is a pair, then the result for the second expression is the `e2` field in the pair.
- An `ismunit` evaluates its subexpression. If the result is an munit expression, then the result for the `ismunit` expression is the MUPL value `(mtrue)`, else the result is the MUPL value `(mfalse)`.

**Hint:** The `call` case is the most complicated. In the sample solution, no case is more than 12 lines and several are 1 line.

4. **Expanding the Language:** MUPL is a small language, but we can write Racket functions that act like MUPL macros so that users of these functions feel like MUPL is larger. The Racket functions produce MUPL expressions that could then be put inside larger MUPL expressions or passed to `eval-exp`. In implementing these Racket functions, do not use `closure` (which is used only internally in `eval-exp`). Also do not use `eval-exp` (we are creating a program, not running it).

(a) Write a Racket function `ifmunit` that takes three MUPL expressions `e_1`, `e_2`, and `e_3`. It returns a MUPL expression that when run evaluates `e_1` and if the result is MUPL’s `munit` then it evaluates `e_2` and that is the result, else it evaluates `e_3` and that is the result. Sample solution: 1 line.
(b) Write a Racket function `mlet*` that takes a Racket list of Racket pairs `'((s₁ . e₁) ... (sᵢ . eᵢ) ... (sₙ . eₙ))` and a final MUPL expression `e_{n+1}`. In each pair, assume `sᵢ` is a Racket string and `eᵢ` is a MUPL expression. `mlet*` returns a MUPL expression whose value is `e_{n+1}` evaluated in an environment where each `sᵢ` is a variable bound to the result of evaluating the corresponding `eᵢ` for `1 ≤ i ≤ n`. The bindings are done sequentially, so that each `eᵢ` is evaluated in an environment where `s₁` through `sᵢ-1` have been previously bound to the values `e₁` through `eᵢ-1`.

(c) Write a Racket function `ifeq` that takes four MUPL expressions `e₁`, `e₂`, `e₃`, and `e₄` and returns a MUPL expression that acts similar to `mif` except that `e₃` is evaluated if and only if `e₁` and `e₂` are equal integers; otherwise, `e₄` is evaluated. (An error occurs if the result of `e₁` or `e₂` is not an integer.) Assume none of the arguments to `ifeq` use the MUPL variables `_x` or `_y`. Use this assumption so that when an expression returned from `ifeq` is evaluated, `e₁` and `e₂` are evaluated exactly once each.

5. **Using the Language:** We can write MUPL expressions directly in Racket using the constructors for the structs and (for convenience) the functions we wrote in the previous problem.

   (a) Bind to the Racket variable `mupl-filter` a MUPL function that acts like `filter` (as we used in ML). Your function should be curried: it should take a MUPL function and return a MUPL function that takes a MUPL list and applies the function to every element of the list returning a new MUPL list with all the elements for which the function returns a value other than `mfalse`. Recall a MUPL list is `munit` or a pair where the second component is a MUPL list.

   **Note:** An earlier version of this problem incorrectly stated that the function argument to `mupl-filter` should return numbers instead of booleans. Since this error was caught only a few days before the deadline, we will accept either version for full credit. However, since MUPL has booleans, the boolean version is better, so please do that version if possible.

   (b) Bind to the Racket variable `mupl-all-gt` a MUPL function that takes an MUPL integer `i` and returns a MUPL function that takes a MUPL list of MUPL integers and returns a new MUPL list of MUPL integers containing the elements of the input list (in order) that are greater than `i`. Use `mupl-filter` (a use of `mlet` is given to you to make this easier).

(Challenge problems on next page.)
6. Challenge Problems: Pursue the following extensions to mupl. Do not modify your existing interpreter at all. Instead, make several new interpreters. An empty definition for the new interpreter for the first challenge problem has been provided to you. For the other challenge problems, make yet another copy of the interpreter as needed.

(a) Write a second version of eval-exp (bound to eval-exp-c) that builds closures with smaller environments: When building a closure, it uses an environment that is like the current environment but holds only variables that are free variables in the function part of the closure. (A free variable is a variable that appears in the function without being under some shadowing binding for the same variable.)

Avoid computing a function’s free variables more than once. Do this by writing a function compute-free-vars that takes an expression and returns a different expression that uses fun-challenge everywhere in place of fun. The new struct fun-challenge (provided to you; do not change it) has a field freevars to store exactly the set of free variables for the function. Store this set as a Racket set of Racket strings. (Sets are predefined in Racket’s standard library; consult the documentation for useful functions such as set, set-add, set-member?, set-remove, set-union, and any other functions you wish.)

You must have a top-level function compute-free-vars that works as just described — storing the free variables of each function in the freevars field — so the grader can test it directly. Then write a new “main part” of the interpreter that expects the sort of mupl expression that compute-free-vars returns. The case for function definitions is the interesting one.

(b) Add symbols (or strings, if you prefer) to mupl, including a way to create symbols, and a way to compare symbols for equality.

(c) Add some side-effecting input-output operations to mupl. print is a good example. Evaluating an mupl print expression should actually print to Racket’s standard output. You may choose to make your print function accept any mupl value, or just strings; it’s up to you.

(d) Using symbols (or strings), mupl programmers can “simulate” structs by using lists and symbol tags. Use such a simulation to encode all of mupl as a simulated mupl struct. Then implement a mupl interpreter in mupl. (This should be a straightforward (if somewhat tedious) translation of the mupl interpreter you wrote in Racket. If you don’t handle tediousness, you may want to consider ways to make your life easier, instead of doing this directly.)

(e) Add mutable pairs to mupl, including a way to create a new mutable pair, access the first and second components of a mutable pair, and modify the first and second component of a mutable pair. As an additional challenge, do this without using Racket’s mutable features. (This latter challenge will require changes to the setup of the interpreter. Think carefully and ask questions if necessary before attempting this.)

(f) Implement (in Racket) a compiler from mupl to Racket. At a high level, your compiler should take a single mupl expression as an argument, and return a (quoted) Racket expression that “does the same thing”. You will need to understand quoting to do this problem, please see the last few pages of the Unit 6 reading notes, and lookup quoting in the Racket documentation. Ask questions.

(g) Implement (in Racket) a compiler from (the mupl-like subset of) Racket to mupl. Your compiler should take a single (quoted) Racket expression as an argument and return a mupl expression that “does the same thing”. If your compiler encounters a feature of Racket that mupl does not have, report an error. This problem also requires understanding quoting; see the resources mentioned in the previous challenge problem.

(h) Do something else interesting related to mupl.