CSE 341, Autumn 2012, Assignment 4
Due: Friday October 26, 10:00PM

60 points total (5 points for Question 1, 25 points for Question 2, 15 points each for Questions 3 and 4); up to 10% extra for the extra credit question.

You can use up to 3 late days for this assignment.

Set-up: For this assignment, edit copies of `hw4provided.rkt` and `hw4tests.rkt`, which are on the course website. In particular, replace occurrences of `"CHANGE"` to complete the problems. Do not use any mutation (`set!`, `set-mcar!`, etc.) anywhere in the assignment.

Overview: This homework 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 `hw4provided.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).
- 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 #f` s_2 e`)` is allowed for anonymous nonrecursive functions.
- If e_1, e_2, and e_3, and e_4 are MUPL expressions, then `(ifgreater` e_1 e_2 e_3 e_4`)` is a MUPL expression (a conditional meaning e_1 is strictly greater than e_2).
- 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 of 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 `(fst` e_1`)` is a MUPL expression (getting part of a pair).
- If e_1 is a MUPL expression, then `(snd` e_1`)` is a MUPL expression (getting part of a pair).
- `(aunit)` is a MUPL expression (holding no data, much like `()` in ML or `null` in Racket).
- If e_1 is a MUPL expression, then `(isaunit` e_1`)` is a MUPL expression (testing for `(aunit)`).
- `(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, a MUPL closure, a MUPL `aunit`, or a MUPL pair of MUPL values. Similar to Racket, we can build list values out of nested pair values that end with aunit. 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))`. Think of `(int 3)` as part of the program's syntax — it's how you write the number 3 in MUPL. But do *not* assume MUPL programs are free of "type" errors like `(fst (int 7))` or `(add (aunit) (int 7))`.

Warning: This assignment is difficult because you have to understand MUPL well and debugging an interpreter is an acquired skill. Start early.

Turn-in Instructions

- Put all your solutions in one file, `lastname_hw4.rkt`, where `lastname` is replaced with your last name. Put unit tests in `lastname_hw4_tests.rkt`. You should have tests for each kind of MUPL expression, including testing for bad arguments. (See the provided tests for `int` and `add` for examples.) In addition, test that lexical scoping is working properly by testing some nested `mlet` expressions, that functions are properly capturing and using their environment of definition, and that recursion is working properly.
- Turn in your files using the Catalyst dropbox link on the course website.

Problems:

1. **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.

2. **Implementing the mupl Language:** Write a MUPL interpreter, i.e., a Racket function `eval-prog` that takes a MUPL program `p` and either returns the MUPL value that `p` evaluates to 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-prog (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.)
   - Functions are lexically scoped: A function evaluates to a closure holding the function and the current environment.
   - An `ifgreater` evaluates its first two subexpressions to values v_1 and v_2 respectively. Assuming both values are integers, it evaluates its third subexpression if v_1 is a strictly greater integer than v_2 else it evaluates its fourth subexpression.
   - An mlet expression 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 `#f`) and the function's argument to the result of the second subexpression.
   - A pair expression evaluates its two subexpressions and produces a (new) pair holding the results.
   - A fst expression evaluates its subexpression. It is an error if the subexpression is not a pair of values. Else the result of the fst expression is the `e1` field in the pair.

- A snd expression is the same as a fst expression except the result is the `e2` field of the pair.
- An `isaunit` expression evaluates its subexpression. If the result is unit, then the result for the isunit expression is the integer 1, else the result is the integer 0.

Hint: The `call` case is the most complicated. In the sample solution, no case is more than 12 lines and several are 1 line. Start writing and running the unit tests very quickly, adding tests as you implement more functionality. Comment out the last test (of `mupl-mapAddN`), and run the tests as soon as you have the top-level function `eval-prog` fixed and a case for `int` defined. Then tackle variables and `mlet`, and only after that's working start on functions and `call`.

3. **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-prog`. In implementing these Racket functions, do not use `closure` (which is only used internally in `eval-prog`) nor `eval-prog` (we are creating a program, not running it).

   (a) Write a Racket function `ifaunit` 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 unit then it evaluates e_2 and that is the overall result, else it evaluates e_3 and that is the overall result. Sample solution: 1 line.

   (b) Write a Racket function `mlet*` that takes a Racket list of Racket pairs '($(s_1\ .\ e_1) \ldots (s_i\ .\ e_i) \ldots (s_n\ .\ e_n)$) and a final MUPL expression e_{n+1}. In each pair, assume s_i is a Racket string and e_i is a MUPL expression. `mlet*` returns a MUPL expression whose value is e_{n+1} evaluated in an environment where each s_i is a variable bound to the result of evaluating the corresponding e_i for $1 \leq i \leq n$. The bindings are done sequentially, so that each e_i is evaluated in an environment where s_1 through s_{i-1} have been previously bound to the values e_1 through e_{i-1}.

   (c) Write a Racket function `ifeq` that takes four MUPL expressions e_1, e_2, e_3, and e_4 and returns a MUPL expression that acts like `ifgreater` except e_3 is evaluated if and only if e_1 and e_2 are equal integers. Unfortunately, MUPL does not have hygiene and we want to evaluate e_1 and e_2 exactly once, so assume the MUPL expressions do not use the variables `_x` and `_y` (i.e., you can use these variables to implement `ifeq`).

4. **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-map` a MUPL function that acts like map (as we used extensively 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. Recall a MUPL list is aunit or a pair where the second component is a MUPL list.

   (b) Bind to the Racket variable `mupl-mapAddN` a MUPL function that takes a MUPL integer i and returns a MUPL function that takes a list of MUPL integers and returns a new list that adds i to every element of the list. Use `mupl-map` (a use of `mlet` is given to you to make this easier).

5. **Challenge Problem:** Write a second version of `eval-prog` (bound to `eval-prog2`) that builds closures with smaller environments: When building a closure, it uses an environment that is like the current environment but only holds 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.) Note: You will have to write a Racket function that takes a MUPL expression and computes its free variables.

   For full challenge-problem credit, use memoization (yes, you should use mutation for this) to avoid computing any function's free variables more than once.

   Warning: The sample solution does not include a solution to the extra credit.