Programming Languages (Coursera / University of Washington) Assignment 5

Set-up: For this assignment, edit a copy of `hw5.rkt`, which is 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 `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).
- 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. It is a conditional where the result is e_3 if e_1 is strictly greater than e_2 else the result is e_4. Only one of e_3 and e_4 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 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 the first part of a pair).
- If e_1 is a MUPL expression, then `(snd` e_1`)` is a MUPL expression (getting the second part of a pair).
- `(aunit)` is a MUPL expression (holding no data, much like `()` in ML or `null` in Racket). Notice `(aunit)` is a MUPL expression, but `aunit` is not.
- 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 a MUPL 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))`. But do *not* assume MUPL programs are free of type errors like `(add (aunit) (int 7))` or `(fst (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 (same as for previous assignments): First, follow the instructions on the course website to submit your solution file (not your testing file) for auto-grading. Do not proceed to the peer-assessment submission until you receive a high-enough grade from the auto-grader: Doing peer assessment

requires instructions that include a sample solution, so these instructions will be "locked" until you receive high-enough auto-grader score. Then submit your same solution file again for peer assessment and follow the peer-assessment instructions.

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-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.)
   - 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. If 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-name (i.e., the parameter name) 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. If the result for the subexpression is a pair, then the result for the fst expression is the `e1` field in the pair.
   - A snd expression evaluates its subexpression. If the result for the subexpression is a pair, then the result for the snd expression is the `e2` field in the pair.
   - An `isaunit` expression evaluates its subexpression. If the result is an aunit expression, then the result for the isaunit expression is the MUPL value `(int 1)`, else the result is the MUPL value `(int 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.

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-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 `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 MUPL's aunit 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. 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, e_1 and e_2 are evaluated exactly once each.

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 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 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-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.