CSE 341, Spring 2020, Assignment 5
Due: Tuesday, May 26, 11:59PM

Set-up: For this assignment, edit a copy of `hw5.rkt`, which is available in the zip archive `hw5.zip` on the course website. In particular, replace occurrences of `"CHANGE"` to complete the problems. *You may not use any mutation (`set!`, `set-mcar!`, etc.) anywhere in the assignment.*

Overview: This homework has to do with the programming language MUPL (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 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 `(ifnz` e_1 e_2 e_3`)` is a MUPL expression. It is a condition where the result is e_2 if e_1 is not zero else the result is e_3. 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 one of the following:

- a MUPL integer constant
- a MUPL closure
- a MUPL munit
- a MUPL pair of MUPL values

Similar to Racket, we can build MUPL list values out of nested MUPL pair values that end with a MUPL munit. Such a MUPL value is called a MUPL list.

You can assume MUPL programs are syntactically correct (e.g., do not worry about wrong things like `(int "hi")` or `(int (int 37))`. But you can *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 keeping track of what code is Racket and what code is MUPL. Debugging an interpreter is an acquired skill. Take your time, think carefully, and don't try to write code until you understand *exactly* what that code is supposed to do.

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:** Complete 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. If an error occurs, provide a meaningful error message in terms of the MUPL error, not a Racket error.

   A MUPL expression is evaluated under an environment (for evaluating variables, as usual). In our interpreter, we will use a Racket list of Racket pairs to represent this environment (which is initially empty). You can use the provided `envlookup` function ***without modification*** to lookup variables in an environment.

   You should not modify the `eval-exp` function, but instead complete the implementation of `eval-under-env` by adding cases to evaluate each type of MUPL expression described below. Two cases (variables and `add`) have been provided to get you started– do not change these cases.

   The semantics for MUPL expressions are given on the next page.

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. (This case has been provided for you.)
- An addition evaluates its subexpressions and, assuming they both produce integers, produces the integer that is their sum. (This case has been provided for you.)
- Functions are lexically scoped: A function evaluates to a closure holding the function and the current environment.
- A pair expression evaluates its two subexpressions and produces a (new) pair holding the results.
- A first expression 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 expression 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` expression evaluates its subexpression. If the result is an munit expression, then the result for the ismunit expression is the MUPL value `(int 1)`, else the result is the MUPL value `(int 0)`.
- 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 `(int 1)`, else the result is the MUPL value `(int 0)`.
- An `ifnz` evaluates its first expression to a value v_1. If it is an integer, then if it is not zero, then ifnz evaluates its second subexpression, else it evaluates its third 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 `null`) and the function's argument-name (i.e., the parameter name) to the result of the second subexpression.

Hints: The `call` case is the most complicated. The `fun` case is surprisingly *not* complicated. In the sample solution, no case is more than 12 lines and several are 1 line, though you do not need to match these.

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).

   Hint: Pay close attention to when you are writing Racket code and when you are writing MUPL code. Be careful to use proper MUPL syntax, especially around variables and function calls.

   (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 $'((s_1 \ . \ e_1) \ldots (s_i \ . \ e_i) \ldots (s_n \ . \ e_n))$ 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}. Sample solution: 5 lines.

(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 `ifnz` except e_3 is evaluated if and only if e_1 and e_2 are equal integers. (An error occurs if the result of e_1 or e_2 is not an integer.) Assume none of the arguments to `ifeq` use the MUPL variables `_x` or `_y`. Use this assumption to ensure that when an expression returned from `ifeq` is evaluated, e_1 and e_2 are evaluated exactly once each. Sample solution: 7-8 lines.

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.

   In these problems, you are binding Racket variables to MUPL expressions. Your solutions should use only MUPL syntax (except for the initial Racket `define`.

   (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 number other than zero (causing an error if the function returns a non-number). Recall a MUPL list is munit or a pair where the second component is a MUPL list. Sample solution: 11 lines.

   (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). Sample solution: 5 lines.

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.

Testing

In addition to implementing the functions described above, you must write a suite of tests to verify that your functions work correctly. Tests should be written using the approach shown in section, and should be comprehensive enough to fully verify that your functions work as indicated. Be sure to consider edge cases and unusual (but valid) inputs. Truly exceptional test suites may receive a small amount of extra credit.

Assessment

To receive full credit, your solutions should be:

- Functionally correct
- Written in good style according to the style guide, including indentation and line breaks
- Written using only features discussed in class through Monday, May 18 (Lecture 17).
- Written **without** using mutation.

Turn-in Instructions

- Add all your solutions to the main problems and challenge problems (if you worked on them) to the file `hw5.rkt`.
- Add all your tests to the file `hw5tests.rkt`.
- Follow the link on the course website to submit your files to Gradescope.
- The Gradescope autograder will confirm that you have submitted the correct file and that your code compiles. **Submissions that do not compile will receive a 15% penalty!**