Programming Language, Assignment 5
Due: May 22nd, 11:59 pm

For this assignment, edit a copy of the attached hw5.rkt. 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 begging 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 e1 and e2 are MUPL expression, then (add e1 e2) is a MUPL expression (an addition).
- If s1 and s2 are Racket strings and e is a MUPL expression, then (fun s1 s2 e) is a MUPL expression (a function). In e, s1 is bound to the function itself (for recursion) and s2 is bound to the (one) argument. Also, (fun #f s2 e) is allowed for anonymous nonrecursive function.
- If e1, e2, e3 and e4 are MUPL expressions, then (ifgreater e1 e2 e3 r4) is a MUPL expression. It is a conditional where the result is e3 if e1 is strictly greater than e2 else the result is e4. Only one of e3 and e4 is evaluated.
- If e1 and e2 are MUPL expressions, then (call e1 e2) is a MUPL expression (a function call).
- If s is a Racket string and e1 and e2 are MUPL expressions, then (mlet s e1 e2) is a MUPL expression (a let expression where the value resulting e1 is bound to s in the evaluation of e2).
- If e1 and e2 are MUPL expressions, then (apair e1 e2) is a MUPL expression (a pair-creator).
- If e1 is a MUPL expression, then (fst e1) is a MUPL expression (getting the first part of a pair).
- If e1 is a MUPL expression, then (snd e1) 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 e1 is a MUPL expression, then (isaunit e1) 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)).

Submission

Write the functions in problem 1 – 4 in a single file, named "*sol5.rkt"*.
Then upload the file to the course homepage (under the assignment 5 menu). Make sure that you test your solution in the repl.

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 and 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 v1 and v2 respectively. If both values are intergers, it evaluates its third subexpression if v1 is a strictly greater integer than v2 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. 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 integer 1, else the result is the MUPL 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.

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 e1, e2, and e3. It returns a MUPL expression that when run evaluates e1 and if the result is MUPL's aunit then it evaluates e2 and that is the overall result, else it evaluates e3 and that is the overall 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 en+1. In each pair, assume si is a Racket string and ei is a MUPL expression. mlet* returns a MUPL expression whose value is en+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 an MUPL expression that acts like ifgreater except e3 is evaluated if and only if e1 and e2 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, e1 and e2 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).