CSE 413, Autumn 2008, Assignment 4
Due: Tuesday Oct. 28, 11:00 PM

Set-up: For this assignment, edit a copy of hw4skeleton.scm, which is on the course website. In particular, replace occurrences of "CHANGE" to complete the problems. Do not use any mutation (set!, set-car!, etc.) anywhere in the assignment.

Overview: This homework has to do with mupl (a Made Up Programming Language). mupl programs are written directly in Scheme using the structs defined at the beginning of hw4skeleton.scm, according to this syntax definition:

- If s is a Scheme string, then (make-var s) is a mupl expression (a variable use).
- If n is a Scheme integer, then (make-int n) is a mupl expression (a constant).
- If e1 and e2 are mupl expressions, then (make-add e1 e2) is a mupl expression (an addition).
- If s1 and s2 are Scheme strings and e is a mupl expression, then (make-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, (make-fun #f s2 e) is allowed for nonrecursive functions.
- If e1, e2, and e3, and e4 are mupl expressions, then (make-ifgreater e1 e2 e3 e4) is a mupl expression (a conditional evaluating to e3 if e1 is strictly greater than e2, otherwise to e4).
- If e1 and e2 are mupl expressions, then (make-app e1 e2) is a mupl expression (a function application).
- If e1 and e2 are mupl expressions, then (make-m-pair e1 e2) is a mupl expression (a pair-creator).
- If e1 is a mupl expression, then (make-fst e1) is a mupl expression (getting part of a pair).
- If e1 is a mupl expression, then (make-snd e1) is a mupl expression (getting part of a pair).
- (make-m-unit) is a mupl expression (holding no data, much like () in Scheme).
- If e1 is a mupl expression, then (is-m-unit e1) is a mupl expression (testing for (make-m-unit)).
- (make-closure env f) is a mupl value where f is a mupl function (an expression made from make-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 an integer constant (wrapped in make-int), a closure, unit, or a pair of values. Notice that like in Scheme we can build list values out of nested pair values that end with unit. You should assume mupl programs are syntactically correct (e.g., do not worry about wrong things like (make-int "hi") or (make-int (make-int 37))). But do not assume mupl programs are free of “type” errors like (make-add (make-unit) (make-int 7)) or (make-fst (make-int 7)).

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

1. (Implementing the language) Write a mupl interpreter, i.e., a Scheme function eval-prog that takes a mupl program p and either returns the mupl value that p evaluates to or calls Scheme’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). Use a list of (Scheme) pairs for the environment (starting with () so that you can use the provided envlookup function. Here is an informal semantics for mupl expressions:
• All values (including closures) evaluate to themselves. For example, (eval-prog (make-int 17)) would return (make-int 17), not 17.

• A variable evaluates to a value according to 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 application 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 second value of the application.

• A pair expression evaluates its two subexpressions and produces a (new) pair holding the results.

• A fst expression evaluates its subpexpression. It is an error if the result is not a pair of values. Else the result of the fst expression is the $e_1$ field in the pair.

• A snd expression is the same as a fst expression except the result is the $e_2$ field of the pair.

• An is-m-unit expression evaluates its subpexpression. 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 app case is definitely the most complicated. In the sample solution, no case is more than 12 lines and several are 1 line.

Note: You will need to use DrScheme’s “Pretty Big” language, which provides define-struct.

2. (Expanding the language) MUPL is a small language, but we can write Scheme functions that act like MUPL macros. They produce MUPL programs that other code could later pass to eval-prog. These Scheme functions you write produce MUPL expressions that you could have written by hand, i.e., they are “macros for MUPL.” In implementing these Scheme functions, do not use make-closure (which is only used internally in eval-prog) nor eval-prog (we are creating a program, not running it).

(a) Write a Scheme function ifunit 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 Scheme function mlet that takes a string $s$ and MUPL expressions $e_1$ and $e_2$. It returns a MUPL expression that when run evaluates $e_1$ and then evaluates $e_2$ in an environment where $s$ is bound to the result of $e_1$. The result for $e_2$ is the overall result. Sample solution: 1 line. Hint: Create a MUPL function and an application of that function to an argument.

(c) Write a Scheme 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 $e_1$ 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. Sample solution is 7 (short) lines and uses mlet.

3. (Using the language) We can write MUPL expressions directly in Scheme using the constructors for the structs and (for convenience) the functions we wrote in the previous problem.

(a) Bind to the Scheme variable mupl-map a MUPL function that acts like map (as we used extensively in Scheme). Your function should be curried: i.e., 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. A MUPL list is simply unit or a pair where the second component is a MUPL list. Sample solution: 7 lines.

(b) Bind to the Scheme variable `mupl-mapAddN` a MUPL function that takes an integer `i` and returns a MUPL function that takes a list of 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 easy). Sample solution is 4 lines (including the line given to you).

**Turn-in Instructions**

- Put all your solutions in one file, `lastname_hw4.scm`, where `lastname` is replaced with your last name.
- The first line of your `.scm` file should be a Scheme comment with your name and the phrase `homework 4`.
- Use the regular homework dropbox reached from the course home page to hand in your file.