

CSE 341, Winter 2008, Assignment 5

Due: Thursday 28 February, 8:00AM

Last updated: February 20

Set-up: For this assignment, edit a copy of `hw5skeleton.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 MUPL (a Made Up Programming Language). MUPL programs are written directly in Scheme using the structs defined at the beginning of `hw5skeleton.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 e_1 and e_2 are MUPL expressions, then `(make-add e_1 e_2)` is a MUPL expression (an addition).
- If s_1 and s_2 are Scheme strings and e is a MUPL expression, then `(make-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, `(make-fun #f s_2 e)` is allowed for nonrecursive functions.
- If e_1 , e_2 , and e_3 , and e_4 are MUPL expressions, then `(make-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 `(make-app e_1 e_2)` is a MUPL expression (a function application).
- If e_1 and e_2 are MUPL expressions, then `(make-pair e_1 e_2)` is a MUPL expression (a pair-creator).
- If e_1 is a MUPL expression, then `(make-fst e_1)` is a MUPL expression (getting part of a pair).
- If e_1 is a MUPL expression, then `(make-snd e_1)` is a MUPL expression (getting part of a pair).
- `(make-unit)` is a MUPL expression (holding no data, much like `()` in ML or Scheme).
- If e_1 is a MUPL expression, then `(isunit e_1)` is a MUPL expression (testing for `(make-unit)`).
- `(make-closure f env)` is a MUPL value where f is 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, 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. (Expanding the language) MUPL is a very small language, but we can write some Scheme functions that act like MUPL macros.
 - (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. (Read ahead to the informal semantics on the next page to understand how `isunit` works, which you need to use.)

- (b) Write a Scheme function `mlet` that takes a string `s` and MUPL expressions `e1` and `e2`. It returns a MUPL expression that when run evaluates `e1` and then evaluates `e2` in an environment where `s` is bound to the result of `e1`. The result for `e2` is the overall result. Sample solution: 1 line. Hint: Use a function and call it.
- (c) Write a Scheme function `ifeq` that takes four MUPL expressions (`e1`, `e2`, `e3`, and `e4`) and returns a MUPL expression that acts like `ifgreater` except `e3` is evaluated if `e1` and `e2` are equal integers. Unfortunately, MUPL does not have hygiene and we want to evaluate `e1` and `e2` exactly once, so assume the MUPL expressions do not use the variables `_x` and `_y`. Sample solution is 7 (very short) lines and uses `mlet`.
2. (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 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 `v1` and `v2` respectively. Assuming both values are integers, it evaluates its third subexpression if `v1` is a strictly greater integer than `v2` 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 subexpression. It is an error if the result 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 `isunit` 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 `app` case is definitely the most complicated. In the sample solution, no case is more than 12 lines and several are 1 line.

3. (Using the language) We can write MUPL expressions directly in Scheme using the constructors for the structs and our solution to problem 1.
- (a) Bind to the Scheme 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. 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).

4. **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. Note: You will have to write a Scheme 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.

Turn-in Instructions

- Put all your solutions in one file, `lastname_hw5.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 5`.
- Go to <https://catalysttools.washington.edu/collectit/dropbox/djg7/1359> (link available from the course website), follow the “Homework 5” link, and upload your file.