

# CSC 330

## Programming Languages

### Assignment 5

Note 1 **This assignment is to be done individually**

Note 2 You can discuss the assignment with others, but sharing and copying code is prohibited.

### A note on Academic Integrity and Plagiarism

Please review the following documents:

- Standards for Professional Behaviour, Faculty of Engineering:  
<https://www.uvic.ca/engineering/assets/docs/professional-behaviour.pdf>
- Policies Academic Integrity, UVic:  
<https://www.uvic.ca/students/academics/academic-integrity/>
- Uvic's Calendar section on Plagiarism:  
[https://www.uvic.ca/calendar/undergrad/index.php#/policy/Sk\\_0xsM\\_V](https://www.uvic.ca/calendar/undergrad/index.php#/policy/Sk_0xsM_V)

Note specifically:

#### Plagiarism

Single or multiple instances of inadequate attribution of sources should result in a failing grade for the work. A largely or fully plagiarized piece of work should result in a grade of F for the course.

Submissions will be screened for plagiarism **at the end of the term**.

You are responsible for your own submission, but you could also be responsible if somebody plagiarizes your submission.

### Objectives

After completing this assignment, you will:

- a better understanding of how languages are implemented,
- and how closures work.

### Introduction

This assignment 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:

1. If  $s$  is a Racket string, then  $(\text{var } s)$  is a MUPL expression (a variable use).

2. If  $n$  is a Racket integer, then `(int  $n$ )` is a MUPL expression (a constant).
3. If  $e_1$  and  $e_2$  are MUPL expressions, then `(add  $e_1$   $e_2$ )` is a MUPL expression (an addition).
4. 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.
5. 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.
6. If  $e_1$  and  $e_2$  are MUPL expressions, then `(call  $e_1$   $e_2$ )` is a MUPL expression (a function call).
7. 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  $e_2$ ).
8. If  $e_1$  and  $e_2$  are MUPL expressions, then `(apair  $e_1$   $e_2$ )` is a MUPL expression (a pair-creator).
9. If  $e_1$  is a MUPL expression, then `(fst  $e_1$ )` is a MUPL expression (getting the first part of a pair).
10. If  $e_1$  is a MUPL expression, then `(snd  $e_1$ )` is a MUPL expression (getting the second part of a pair).
11. `(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.
12. If  $e_1$  is a MUPL expression, then `(isaunit  $e_1$ )` is a MUPL expression (testing for `(aunit)`).
13. `(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))`.

## Warnings

- What makes this assignment challenging is that **you have to understand MUPL well** and debugging an interpreter is an acquired skill.
- **Do not use any mutation** (`set!`, `set-mcar!`, etc.) anywhere in the assignment.

## Your task, should you choose to accept it

### A. Warm-Up:

1. 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.
2. 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.

**Hint:** look at the test cases to see what a MUPL list looks like and what its equivalent Racket list is.

### B. Implementing the mupl Language:

3. 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 integers, 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  $v$ . **It must be placed at the beginning of the current environment.**
  - 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 (in that order). **Remember: it is a closure, hence, the evaluation of the function is in the environment where the function is defined, not the current environment (extended with the function name and its parameters—which should be placed at the beginning of its environment).**
  - A `apair` 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  $e_1$  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  $e_2$  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 1:** See the test cases to get a better understanding of what you have to create.

**Hint 2:** Inspect the implementation of `add` and `var`, which are provided (inside `eval-under-env`). Do not start the to code until you understand how they work.

**Hint 3:** The `call` case is the most complicated. In my solution, no case is more than 15 lines and several are 1 line.

- C. **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).
4. 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.
  5. Write a Racket function `mlet*` that takes a Racket list of Racket pairs `'((s1 . e1) ... (si . ei) ... (sn . en))` 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  $e_{i-1}$ .
  6. 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. **You have to figure out how to implement `ifeq` using `isgreater`.**
- D. **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.
7. 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.
  8. 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).

## Provided Code

For this assignment, edit `hw5.rkt`. In particular, replace occurrences of "CHANGE" to complete the problems.

## Restrictions

1. Do not use any mutation (`set!`, `set-mcar!`, etc.) anywhere in the assignment.

## Tests

Tests are divided into 10 sections (see bottom of `hw5tests.rkt`). I recommend you start by commenting all but the first one. Move one test group at a time. It will make your debugging faster. The tests are organized in such a way that they will help you complete your assignment.

## Evaluation

Solutions should be:

1. Correct. We might use more tests than the ones provided.
2. In good style, including indentation and line breaks

As usual, submit your solution in a single file via connex. No need to upload your test files.