

# Programming Language, Assignment 4

## Due: June 4, 11:59 pm

For this assignment, edit a copy of the attached hw4.cc. In particular, replace occurrences of “TODO” to complete the problems.

### Overview

This homework has to do with MUPL (a Made Up Programming Language). MUPL programs are written directly in C++ by using the constructors defined by the structs defined at the beginning of hw4.cc. This is the definition of MUPL’s syntax:

- If *s* is a C++ string, then *Var(s)* is a MUPL expression (a variable use).
- If *n* is a C++ 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 C++ 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(“”, s2, e)* is allowed for anonymous non-recursive function.
- If *e1*, *e2*, *e3* and *e4* are MUPL expressions, then *IfGreater(e1, e2, e3, e4)* 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*. The results of the evaluation of *e1* and *e2* should be *Int* values. 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 C++ 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). 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 Int constant, a MUPL Closure, a MUPL AUnit, or a MUPL APair of MUPL values. Similar to ML, 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 - 3 in a single file, named *"sol4.cc"*. Then upload the file to the course homepage (under the assignment 4 menu). Make sure that you test your solution code compiles and works correctly.

### Problems

#### 1. Warm-Up:

- a. Write a C++ function `ToMuplList` that takes a `List<Expr>` (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 C++ function `FromMuplList` that takes a MUPL list (presumably of MUPL values but that will not affect your solution) and produces an analogous C++ `List<Expr>` (of MUPL values) with the same elements in the same order.

#### 2. Implementing the MUPL Language: Write a MUPL interpreter, i.e., a C++ function `eval` that takes a MUPL expression `e` and either returns the MUPL value that `e` evaluates to under the empty environment or throws `std::runtime_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 `std::map<string, Expr>` 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 (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  $v$ .
- 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 **an empty string, or ""**) 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  $e_1$  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.

3. **Using the Language:** We can write MUPL expressions directly in C++ using the constructors for the structs and also C++ functions that create and return MUPL expressions.

As the example of a C++ function that returns an MUPL expression, IfAUnit is defined in the skeleton code. The function takes three Expr values ( $e_1$ ,  $e_2$ , and  $e_3$ ) and returns the expression that tests if  $e_1$  is AUnit; if so,  $e_2$  is evaluated, otherwise  $e_3$  is evaluated. You can use this function to implement the following functions.

- Write the C++ function MuplMap that returns a MUPL function that acts like the map function (as we used extensively in ML). The returned map 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 APair where the second component is a MUPL list. The pseudo code (in ML) is shown in the comments of this function in the skeleton file.
- Write the C++ function MuplMapAddN that returns a (curried) MUPL function. The returned function 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 MuplMap that you implemented above. Also, the pseudo code is shown in the comments of this function in the skeleton file.

To test your implementation of MuplMap and MuplMapAddN, you can use makeIntList() function in the skeleton code. Example code for the function is in the main() function.