CMPSC 461, Project MUL461 Part B
Due: Friday April 30rd, 11:59PM

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

This assignment is built on top of MUL461 Part A. You will need to copy over your implementation for Part A before proceeding with the project to be able to pass the tests.

Overview: This project has to do with MUL461 (a Made Up Language for CMPSC 461). MUL461 programs are written directly in Racket by using the constructors defined by the structs defined at the beginning of `mulb.rkt`. This is the definition of MUL461's syntax for Part A:

- If s is a Racket string, then `(mul461-var` s`)` is a MUL461 expression (a variable use).
- If n is a Racket integer, then `(mul461-int` n`)` is a MUL461 expression (an integer constant).
- If e_1 and e_2 are MUL461 expressions, then `(mul461-+` e_1 e_2`)` is a MUL461 expression (an addition).
- If e_1 and e_2 are MUL461 expressions, then `(mul461--` e_1 e_2`)` is a MUL461 expression (a subtraction).
- If e_1 and e_2 are MUL461 expressions, then `(mul461-*` e_1 e_2`)` is a MUL461 expression (a multiplication).
- If b is a Racket boolean, then `(mul461-bool` b`)` is a MUL461 expression (a boolean constant).
- If e_1 and e_2 are MUL461 expressions, then `(mul461-and` e_1 e_2`)` is a MUL461 expression. e_1 and e_2 should both evaluate to `mul461-bool` values.
- If e_1 and e_2 are MUL461 expressions, then `(mul461-or` e_1 e_2`)` is a MUL461 expression. e_1 and e_2 should both evaluate to `mul461-bool` values.
- If e is a MUL461 expression that evaluates to a `mul461-bool` value, then `(mul461-not` e`)` is a MUL461 expression.
- If e_1 and e_2 are MUL461 expressions, then `(mul461-<` e_1 e_2`)` is a MUL461 expression that evaluates to a `mul461-bool` value if e_1 and e_2 evaluate to `mul461-int` values.
- If e_1 and e_2 are MUL461 expressions, then `(mul461-=` e_1 e_2`)` is a MUL461 expression that evaluates to a `mul461-bool` value if e_1 and e_2 evaluate to `mul461-int` values.
- If e_1, e_2, and e_3 are MUL461 expressions, then `(mul461-if` e_1 e_2 e_3`)` is a MUL461 expression. It is a condition where the result is e_2 if e_1 is a boolean constant of true else the result is e_3. Only one of e_2 and e_3 is evaluated.
- If s is a Racket string and e_1 and e_2 are MUL461 expressions, then `(mul461-let` s e_1 e_2`)` is a MUL461 expression (a let expression where the value resulting from evaluating e_1 is bound to s in the evaluation of e_2).

This is additional syntax of the language for Part B. In Part B we are adding support for pairs and function (lambda expressions) and function calls into our MUL461 language.

- If e_1 and e_2 are MUL461 expressions, then `(mul461-cons` e_1 e_2`)` is a MUL461 expression (a pair-creator).
- If e is a MUL461 expression, then `(mul461-car` e`)` is a MUL461 expression (getting the first part of a pair).

- If e is a MUL461 expression, then `(mul461-cdr` e`)` is a MUL461 expression (getting the second part of a pair).
- `(mul461-null)` is a MUL461 expression (holding no data, much like `()` in ML or `null` in Racket). Notice `(mul461-null)` is a MUL461 expression, but `mul461-null` is not.
- If e is a MUL461 expression, then `(mul461-isnull` e`)` is a MUL461 expression (testing for `(mul461-null)`) that should evaluates to MUL461 expression of boolean value `(mul461-bool #t)` or `(mul461-bool #f)`.
- If s_1 and s_2 are Racket strings and e is a MUL461 expression, then `(mul461-fun` s_1 s_2 e`)` is a MUL461 expression (a function). In e, s_1 is bound to the function itself (for recursion) and s_2 is bound to the (one) argument.
- If e_1 and e_2 are MUL461 expressions, then `(mul461-call` e_1 e_2`)` is a MUL461 expression (a function call).
- `(mul461-closure` f env`)` is a MUL461 value where f is MUL461 function (an expression made from `mul461-fun`) and env is an environment mapping variables to values. Closures do not appear in source programs; they result from evaluating functions.

In Part A, A MUL461 *value* is a MUL461 integer constant, a or a MUL461 boolean constant.

In Part B, A MUL461 value from Part A is still a value, and for the case of pairs, the definition of a MUL461 *value* is recursive: a MUL461 expression `(mul461-null)` is a MUL461 value. If v_1, v_2 are MUL461 values, then `(mul461-cons` v_1 v_2`)` is also a MUL461 value. A MUL461 `(mul461-fun ...)` evalutes to a MUL461 value as a `(mul461-closure f env)` with its current environment. The env for f in the closure is implemented as static scoping with deep-binding.

You should assume MUL461 programs are syntactically correct (e.g., do not worry about wrong things like `(mul461-int "hi")` or `(mul461-int (mul461-int 37))`. But do *not* assume MUL461 programs are free of type errors like `(mul461-+ (mul461-bool #t) (mul461-int 7))` or `(mul461-not (mul461-int 3))`.

Warning: What makes this assignment challenging is that you have to understand MUL461 well and debugging an interpreter is an acquired skill.

Turn-in Instructions: Turn in your modified `mulb.rkt` to gradescope.

Problems:

1. **Implementing the** MUL461 **Language:** Write a MUL461 interpreter, i.e., a Racket function `eval-exp` that takes a MUL461 expression `e` and either returns the MUL461 value that `e` evaluates to under the empty environment or calls Racket's `error` if evaluation encounters a run-time MUL461 type error or unbound MUL461 variable.

   A MUL461 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 MUL461 expressions:

   - All values evaluate to themselves. For example, `(eval-exp (mul461-int 17))` would return `(mul461-int 17)`, *not* 17.
   - A variable evaluates to the value associated with it in the environment.(This case for var is done for you.)
   - An addition/subtraction/multiplication evaluates its subexpressions and, assuming they both produce integers, produces the `mul461-int` n that is their sum/difference/product. (Note the case for addition is done for you to get you pointed in the right direction.)
   - An $<$ or $=$ comparison evaluates its subexpressions and, assuming they both produce integers, produces the `mul461-bool` b that is the result of comparing the two integer values.
   - An and/or evaluates its subexpressions and, assuming they both produce booleans, produces the `mul461-bool` b that is their and/or boolean operation result.
   - A not evaluates its subexpression and, assuming it produces a boolean, produces the `mul461-bool` b that is the logical negation of its subexpression result.
   - An `if` evaluates its first expression to a value v_1. If it is a boolean, then if it is `mul461-bool #t`, then if evaluates to its second subexpression, else it evaluates its third subexpression. Only one of e_2 and e_3 should be evaluated. Do not evaluate both e_2 and e_3.
   - A `let` 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 let expression to v.

   Here is additional description of the semantics of MUL461 expressions from Part B:

   - All values evaluate to themselves. For example, `(mul461-null)`, `(mul461-closure f env)` are already MUL461 values.
   - For a pair creater expression `(mul461-cons` e_1 e_2`)`, you should first evaluate the sub-expressions e_1 and e_2 to MUL461 values v_1 and v_2, and the original pair will evaluate to `(mul461-cons` v_1 v_2`)`
   - For `mul461-car` and `mul461-cdr`, you should first evaluate the subexpression e to MUL461 value v. If v is not of the form `(mul461-cons` v_1 v_2`)`, then you should raise a type error similar to when adding non-integers. Assume e evaluates to `(mul461-cons` v_1 v_2`)`, then `(mul461-car e)` should evaluate to v_1 and `(mul461-cdr e)` should evaluate to v_2.
   - For `(mul461-isnull e)`, you should first evaluate e to a value v. If v is `(mul461-null)`, then the testing expression should evaluate to `mul461-bool #t`; other wise the testing expression should evaluate to `mul461-bool #f`.
   - For a MUL461 expression e that is `(mul461-fun fname formal body)`, it should be evaluated to a closure `(mul461-closure e env)` where env is the current env that provides mapping for all potential free vars in e.

- For a function call MUL461 expression `(mul461-call f actual)`, you should first evaluate f to v_1 and `actual` to v_2, and then make sure that v_1 is a closure (or otherwise a type error should be raised). Assume v_1 is a closure of the form `(mul461-closure (mul461-fun fname formal body) env)`, then you should add these two bindings to the env in the closure (formal: v_2, fname:v_1, env) and evaluate `body` with this new environment that binds the formal parameter to the actual argument value and binds the function name to the function closure for recursive functions.

2. **Expanding the Language:** MUL461 is a small language, but we can write Racket functions that act like MUL461 macros so that users of these functions feel like MUL461 is larger. The Racket functions produce MUL461 expressions that could then be put inside larger MUL461 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 `makelet*` that takes a Racket list of Racket pairs '$((s_1\ .\ e_1) \ldots (s_i\ .\ e_i) \ldots (s_n\ .\ e_n))$ and a final MUL461 expression e_{n+1}. In each pair, assume s_i is a Racket string and e_i is a MUL461 expression. `makelet*` returns a MUL461 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}.

   (b) Write a Racket function `makelist` that takes a Racket list of MUL461 expressions, and creates a MUL461 expression that is essentially a list of these MUL461 expressions using `mul461-cons` and `mul461-null`. Do not evaluate any of the expressions.

3. **Using the Language:** We can write MUL461 expressions directly in Racket using the constructors for the structs and (for convenience) the functions we wrote in the previous problem.

   (a) Write a function `makefactorial` that gives a racket integer n, returns a MUL461 expression that is similar to $n * ((n-1) * (n-2) ... * 1)$ using MUL461 constructs. For example `(makefactorial 3)` should return this value:

   `(mul461-* (mul461-int 3) (mul461-* (mul461-int 2) (mul461-int 1)))`.

   (b) Bind to the Racket variable `mul461-filter` a MUL461 function that acts like filter (as we used in Racket). Your function should be curried: it should take a MUL461 function and return a MUL461 function that takes a MUL461 list and applies the function to every element of the list returning a new MUL461 list with all the elements for which the function returns `(mul461-bool #t)`. Recall a MUL461 list is `(mul461-null)` or a pair where the second component is a MUL461 list.

   (c) Bind to the Racket variable `mul461-map` a MUL461 function that acts like map (as we used in Racket). Your function should be curried: it should take a MUL461 function and return a MUL461 function that takes a MUL461 list and applies the function to every element of the list, returning a new MUL461 list with all the elements be the return value of the function.