CSE 341, Autumn 2020, Assignment 6
Due: Wednesday, December 2, 6:00PM

Set-up: For this assignment, grab the starter code. The first problem is about the untyped lambda calculus from lecture 20. The rest of the problems are Racket programming. For these problems, edit `hw6.rkt` to replace all occurrences of `"CHANGE"` with your solutions.

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

Overview: Most of this homework 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 `hw6.rkt`. This is the definition of MUPL's syntax:

- If s is a Racket string, then `(var` s`)` is a MUPL expression (a variable use).
- If n is a Racket integer, then `(int` n`)` is a MUPL expression (a constant).
- If e_1 and e_2 are MUPL expressions, then `(add` e_1 e_2`)` is a MUPL expression (an addition).
- 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 null` s_2 e`)` is allowed for anonymous nonrecursive functions.
- If e_1 and e_2 are MUPL expressions, then `(isgreater` e_1 e_2`)` is a MUPL expression (a comparison).
- If e_1, e_2, and e_3 are MUPL expressions, then `(ifnz` e_1 e_2 e_3`)` is a MUPL expression. It is a condition where the result is e_2 if e_1 is not zero else the result is e_3. Only one of e_2 and e_3 is evaluated.
- If e_1 and e_2 are MUPL expressions, then `(call` e_1 e_2`)` is a MUPL expression (a function call).
- 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 from evaluating e_1 is bound to s in the evaluation of e_2).
- If e_1 and e_2 are MUPL expressions, then `(apair` e_1 e_2`)` is a MUPL expression (a pair-creator).
- If e_1 is a MUPL expression, then `(first` e_1`)` is a MUPL expression (getting the first part of a pair).
- If e_1 is a MUPL expression, then `(second` e_1`)` is a MUPL expression (getting the second part of a pair).
- `(munit)` is a MUPL expression (holding no data, much like `()` in ML or `null` in Racket). Notice `(munit)` is a MUPL expression, but `munit` is not.
- If e_1 is a MUPL expression, then `(ismunit` e_1`)` is a MUPL expression (testing for `(munit)`).
- `(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 munit, 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 munit. 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 (munit) (int 7))` or `(first (int 7))`.

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

Turn-in Instructions

- Put your solutions to Problem 1 in the `hw6.ulc` file.
- Put all your solutions to the Racket problems into the `hw6.rkt` file.
- Put tests you wrote for the Racket part in `hw6tests.rkt`.
- Upload your `hw6.ulc`, `hw6.rkt`, and `hw6tests.rkt` to the CSE 341 Autumn 2020 Gradescope Homework 6 Assignment page.

Problems:

1. **Untyped Lambda Calculus**

   (a) Consider the following untyped lambda calculus terms (written in the ULC REPL syntax from lecture).

   ```
   pair = \x. \y. \f. f x y;;
   fst  = \p. p (\x. \y. x);;
   snd  = \p. p (\x. \y. y);;
   ```

   We claim these functions "encode" pairs into the untyped lambda calculus. The function `pair` takes two arguments `x` and `y` and returns an encoded pair. What an encoded pair does is take another argument `f` and calls it with the two components of the pair as arguments. The `fst` and `snd` function take a pair `p` as an argument and use the above encoding to extract either the first or second argument.

   Use `pair` to write an ULC expression that encodes the pair (1, 2). Use the encoding for numbers we learned in lecture. Bind your answer to the name `pair_one_two`.

   (b) When you bind an expression to a name, the ULC interpreter first evaluates the expression to a value, and it prints this value back to you. In a comment, copy in the value the ULC interpreter print back to you for your definition of `pair_one_two`. In the same comment, explain how you can "see" the numbers 1 and 2 inside this value.

   (c) In lecture, we wrote a function `inc` to increment a (n encoded) number. Write a function `dec` to *decrement* a number. (For the purposes of this problem, "decrementing" zero should return zero.) This is harder than you might think, so follow these steps.

   i. Make an attempt on your own at writing such a function without doing anything fancy. In a comment in your `hw6.ulc` file, explain what you tried and where you got stuck.

   ii. Write a function `dec_helper` that takes a (n encoded) number n and returns a (n encoded) pair of n and $n-1$, unless n is zero, in which case it should return the pair $(0, 0)$. Compute this pair "recursively" by calling n with a base case of $(0, 0)$ and a `next` function that transforms the pair (x, y) into the pair $(x + 1, x)$.

   iii. Implement `dec` by calling `dec_helper` and extracting the second component.

   (d) **Challenge Problem:** Write a function in the untyped lambda calculus that takes an "encoded" natural number and computes its factorial.

   Hint: Use a similar "pair trick" as for `dec` above in order to keep track of both the current value of n and the accumulated result so far. Use the provided `mult` from lecture to multiply.

2. MUPL **Warm-Up:**

   (a) 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.

   (b) 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.

3. **Implementing the** MUPL **Language:** 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.)
   - An `isgreater` evaluates its two subexpressions to values v_1 and v_2 respectively. If both values are integers, then if $v_1 > v_2$ the result of the isgreater expression is the MUPL value `(int 1)`, else the result is the MUPL value `(int 0)`.
   - An `ifnz` evaluates its first expression to a value v_1. If it is an integer, then if it is not zero, then ifnz evaluates its second subexpression, else it evaluates its third subexpression.
   - Functions are lexically scoped: A function evaluates to a closure holding the function and the current environment.
   - 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 `null`) and the function's argument-name (i.e., the parameter name) to the result of the second subexpression.
   - A pair expression evaluates its two subexpressions and produces a (new) pair holding the results.
   - A first expression evaluates its subexpression. If the result for the subexpression is a pair, then the result for the first expression is the `e1` field in the pair.
   - A second expression evaluates its subexpression. If the result for the subexpression is a pair, then the result for the second expression is the `e2` field in the pair.
   - An `ismunit` expression evaluates its subexpression. If the result is an munit expression, then the result for the ismunit expression is the MUPL value `(int 1)`, else the result is the MUPL value `(int 0)`.

   Hint: The `call` case is the most complicated. In the sample solution, no case is more than 12 lines and several are 1 line.

4. **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).

   (a) Write a Racket function `ifmunit` 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 munit then it evaluates e_2 and that is the result, else it evaluates e_3 and that is the result. Sample solution: 1 line.

   (b) Write a Racket function `mlet*` 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 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}.

   (c) 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 `ifnz` except e_3 is evaluated if and only if e_1 and e_2 are equal integers. (An error occurs if the result of e_1 or e_2 is not an integer.) 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.

5. **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.

   (a) Bind to the Racket variable `mupl-filter` a MUPL function that acts like filter (as we used 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 with all the elements for which the function returns a number other than zero (causing an error if the function returns a non-number). Recall a MUPL list is munit or a pair where the second component is a MUPL list.

   (b) Bind to the Racket variable `mupl-all-gt` 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 containing the elements of the input list (in order) that are greater than i. Use `mupl-filter` (a use of `mlet` is given to you to make this easier).

6. **Challenge Problem:** Write a second version of `eval-exp` (bound to `eval-exp-c`) that builds closures with smaller environments: When building a closure, it uses an environment that is like the current environment but holds only variables that are free variables in the function part of the closure. (A free variable is a variable that appears in the function without being under some shadowing binding for the same variable.)

   Avoid computing a function's free variables more than once by writing a function `compute-free-vars` that takes an expression and returns a different expression that uses `fun-challenge` everywhere in place of `fun`. The new struct `fun-challenge` (provided to you; do not change it) has a field `freevars` to store exactly the set of free variables for the function. Store this set as a Racket set of Racket strings. (Sets are predefined in Racket's standard library; consult the documentation for useful functions such as `set`, `set-add`, `set-member?`, `set-remove`, `set-union`, and any other functions you wish.)

   You must have a top-level function `compute-free-vars` that works as just described — storing the free variables of each function in the `freevars` field — so the grader can test it directly. Then write a new "main part" of the interpreter that expects the sort of MUPL expression that `compute-free-vars` returns. The case for function definitions is the interesting one.