CSE 341, Fall 2004, Assignment 5 (version 1)
Due: Friday 19 November, 9:00AM

Set-up: For this assignment, edit a copy of `hw5skeleton.scm`, which is on the course website. In particular, replace occurrences of `"CHANGE THIS"` to complete the problems.
Overview: This homework has to do with FROG (a frog is a grown-up tadpole). FROG programs are written 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 FROG expression (a variable).
- If n is a Scheme integer, then `(make-int` n`)` is a FROG expression (a constant).
- If e_1 and e_2 are FROG expressions, then `(make-add` e_1 e_2`)` is a FROG expression (an addition).
- If s is a Scheme string and e_1 and e_2 are FROG expressions, then `(make-alet` s e_1 e_2`)` is a FROG expression (a let-binding).
- If s_1 and s_2 are Scheme strings and e is a FROG expression, then `(make-fun` s_1 s_2 e`)` is a FROG 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 FROG expressions, then `(make-app` e_1 e_2`)` is a FROG expression (a function application).
- If *lst* is a Scheme list of FROG expressions, then `(make-alist` *lst*`)` is a FROG expression (a list).
- If e_1, e_2, and e_3, and e_4 are FROG expressions, then `(make-ifsameint` e_1 e_2 e_3 e_4`)` is a FROG expression (a conditional).
- If e_1 and e_2 are FROG expressions, then `(make-get` e_1 e_2`)` is a FROG expression (a list access).
- If e_1 and e_2 are FROG exressions, then `(make-handle` e_1 e_2`)` is a FROG expression (an exception handler).
- `(make-raise)` is a FROG expression (an exception-raise).

A FROG *value* is a constant, a closure (built with `make-closure`), or a FROG list (built with `make-alist`) of FROG values. We use lists to simulate multiple-argument functions (although currying would also work).

You should assume FROG programs are syntactically correct (e.g., do not worry about wrong things like `(make-int "hi")` or `(make-alist (make-int 37))`. But do *not* assume FROG programs are free of "type" errors like `(make-add (make-alist` e_1`) (make-int 7))` or `(make-get (make-int 7) (make-alist` e_1`))`.

Mutation rules: Don't need it; don't use it.

Warning: This assignment is difficult because you have to understand FROG well and debugging an interpreter is an acquired skill. Start early.

1. Write a Scheme function `getprim` that implements FROG's `make-get` operation. `getprim` takes a thunk `onraise` and two FROG values, `v1` and `v2`. `getprim` should call `onraise` if `v1` is not a FROG list, `v2` is not a FROG constant, or the length of `v1` is less than `v2`. Otherwise, `getprim` should return the `v2`th value in `v1`. Hint: `addprim` is similar: it implements FROG's `make-add` operation.
2. Write a FROG function that takes a FROG list and adds the first two elements in the list (relying on the implementation of `make-get` and `make-add` to raise an exception if necessary). Bind your FROG function to the Scheme variable `frog-addfun`. Hint: Even though your FROG function won't be recursive, you still have to make up a string for the first field in the `fun` struct you build.

3. Write a FROG interpreter, i.e., a Scheme function `eval-prog` that takes a FROG program `p` and either returns the FROG value `p` evaluates to or calls Scheme's error if evaluation encounters an unbound FROG variable or an uncaught FROG exception.

   A FROG expression is evaluated under an environment (for evaluating variables, as usual) and an exception handler (what to do to raise an exception). The interpreter uses a list of pairs for the former (starting with `initial-env` which binds the function you wrote to `"add"`). The interpreter uses a thunk for the latter (starting with a thunk that calls Scheme's `error`). Here is an informal semantics for FROG expressions:

   - A variable evalutes a value using the environment.
   - A constant is a value (evaluate to the FROG value, *not* the underlying Scheme integer).
   - An addition evaluates its subexpressions, and then evaluates to the sum of the results (using `addprim`).
   - A let-binding evaluates its second subexpression and then evaluates its third subexpression in an extended environment (as usual).
   - Functions are lexically scoped: A function evaluates to a closure holding the function and the current environment.
   - An application evaluates its first and second subexpressions to values. If the first is not a closure, it raises an exception. Else, it evaluates the closure's function's body in the closure's environment extended to map the function's name to the closure and the function's argument to the second value of the application.
   - A list evaluates to a list containing each expression in the list evaluated to a value.
   - A conditional evaluates its first two subexpressions to values. If either isn't a constant or they are different constants, the conditional evaluates the fourth subexpression and the result is the result for the conditional. Else the conditional evaluate the third subexpression to produce the result.
   - A get expression evaluates its subexpressions, and then evaluates to a result (using `getprim`).
   - A handle expression evaluates its first subexpression using a *different* exception handler (and if no exception is raised, uses the result): If invoked, this different handler "ignores what was happening" and evaluates the handle's second subexpression under the environment and exception handler that were current when the handle-expression was reached, and that is the handle's result. Hint: Use Scheme's `let/cc`.
   - A raise expression raises an exception (using the current exception handler).

   Hint: The `app` and `handle` cases are the trickiest. In the sample solution, no case is more than 10 lines and most are less than 5.

4. Write a FROG function for folding over a FROG list and bind the result to the Scheme variable `frog-fold`. Your function should take a list with at least 4 elements; let's call the first argument *f*, the second *accum*, the third *lst*, and the fourth *n*. If *n* is greater than the length of *lst*, return *accum*. Otherwise return the result of fold applied to the list holding *f*, *accum2*, *lst*, and *n+1*, where *accum2* is *f* applied to the two-element list holding *accum* and the n^{th} element of *lst*. Hint: To see if a list has *n* elements, use an exception handler and try getting the n^{th} element. Sample solution is 17 lines. You can test your solution with `frog-sum` (written for you) and `frog-has-num` (next problem).

5. Write a FROG function that uses `frog-fold` to see if a FROG list holds a FROG constant. Bind your function to the Scheme variable `frog-has-num`. Your function should take a list with two arguments: first a list and second a constant. It should evaluate to the FROG constant `(make-int 0)` if the constant *is* in the list and `(make-int 1)` otherwise. Hint: Start with an accumulator of `(make-int 1)`. Sample solution is 15 lines.

(Extra credit and turn-in instructions on next page.)

Extra Credit:

EC 1 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 FROG expression and computes its free variables.

EC 2 After doing the previous problem, use memoization to write a more efficient function for computing an expression's free variables.

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`.
- Email your solution to `daverich@cs.washington.edu`.
- The subject of your email should be *exactly* `[cse341-hw5]`.
- Your `.scm` file should be an *attachment*.