CSE 341, Spring 2004, Assignment 5
Due: Friday 14 May, 9:00AM

Last updated: 5 May

As discussed in class, you will implement closure-conversion and write some examples in `minfun`, a tiny language created for this homework. Several key definitions are in `hw5provided.scm`. This assignment may make little sense without notes from class. **Warning: This assignment is difficult.**

Concrete Syntax: The *concrete syntax* for `minfun` is the following:

```
num ::= <<any Scheme number>>
id  ::= <<any Scheme symbol>>
e ::= num
    |  id
    |  (fun (id1 ... idn) e)
    |  (let id e1 e2)
    |  (app e1 e2 ... en)
    |  (if1 e1 e2 e3)
    |  (mul e1 e2)
    |  (add e1 e2)
    |  (is-eq e1 e2)
    |  (pr e1 e2)
    |  (fst e1)
    |  (snd e2)
    |  (set-fst! e1 e2)
    |  (set-snd! e1 e2)
    |  (is-pr e1)
```

To use concrete syntax to create a `minfun` program, put a `'` in front and pass it to the Scheme function `parse`. For example, `(parse '(let x 5 (pr 3 x)))`. **However**, `parse` only accepts functions with one argument (i.e., `(fun (`*id*`)` *e*`)`) and applications of one argument (i.e., `(app` *e*1 *e*2`)`). *Remember to write* ***app****, unlike in Scheme.*

Abstract Syntax: The *abstract syntax* for `minfun` is defined by the `define-struct` definitions in `hw5provided.scm`. Every field must be a `minfun` expression, with these exceptions:

- `fun-args` is a list of symbols (`minfun` identifiers)
- `app-args` is a list of `minfun` expressions

For the most part, the result of parsing is obvious. **However**, the parser desugars `(let` *id* *e*1 *e*2`)` to `(app (fun (`*id*`)` *e*2`)` *e*1`)`.

Semantics: The *semantics* for `minfun` is largely like Scheme. For example, variables are lexically scoped and all constructs eagerly evaluate their arguments except `if1`. The primitives have this meaning:

- `(if1` *e*1 *e*2 *e*3`)` evaluates *e*1. If the result is 1, it evaluates to *e*2, else it evaluates to *e*3. (We use 1 because we do not have booleans.)
- `mul` is multiplication and `add` is addition.
- `is-eq` must take two expressions that evaluate to numbers. It produces 1 if the numbers are the same, else 0.
- The remaining primitives are exactly like corresponding primitives in Scheme: `pr` is like `cons`, `fst` is like `car`, `snd` is like `cdr`, `set-fst!` is like `set-car!`, `set-snd!` is like `set-cdr!`, and `is-pr` is like `pair?` (except it returns 1 or 0).

Encodings: Because `minfun` is so small, we must *encode* some common idioms:

- There is no special empty-list. By convention, `minfun` programmers (all 56 of them), use 99 for the empty-list. For example, `(pr 3 (pr 5 99))` is a list of length 2.
- There is no explicit recursion, but we can fake it with mutation. See `hw5tests.scm` for two examples.

Evaluation: The provided function `evaluate` is an almost-correct interpreter for `minfun` programs. It even handles functions and applications with any number of arguments. **However**, it does not allow free variables in functions, which is particularly problematic since the parser desugars let to functions. Therefore, calling `evaluate` on the provided tests and most other examples will cause an error. *You are not to change the `evaluate` function.*

Printing: To view results and help with debugging, the provided function `to-sexp` is useful. It's roughly the opposite of parsing. **However**, if its argument has a *cycle*, then it will go an infinite loop, so don't call it! Warning: evaluating programs that fake recursion with mutation can produce cycles.

Problems:

1. (Writing `minfun` programs) Hint: Sample solution is 14 lines
   (a) Using `parse`, define a Scheme variable `minfun-append` that holds the abstract syntax for a `minfun` function that appends two lists. In other words, write a Scheme expression of the form `(define minfun-append (parse '(...)))` for an appropriate `...`
      - You will have to use currying because `parse` requires one-argument functions.
      - You will have to fake recursion with mutation.

   (b) Using `parse`, define Scheme variables `lst1` and `lst2`, each holding a `minfun` list. `lst1` should hold a list holding 1, 2, and 3 in that order. `lst2` should hold a list holding 4 and 5 in that order. Remember 99 "is" the empty-list.

   (c) *Without using parse*, define a Scheme variable `ans` that holds a `minfun` program that applies `minfun-append` to `lst1` and `lst2`. You should build abstract syntax directly (by calling the Scheme function `make-app`). This will give you a good third test (though of course you should write more).

2. (Closure Conversion) Hint: Sample solution including helper functions is 70 lines

   You must write a Scheme function `convert` that takes a `minfun` program (in abstract syntax) and produces an equivalent `minfun` program (in abstract syntax). You may assume the input program has no cycles (though running it may make cycles), has only one-argument functions, and has no undefined variables. Your output must have no free variables (so you can call `evaluate`). You should write these Scheme helper functions (you may write others):

   (a) `convert-body` does the actual work (recursively). It takes 4 arguments:
      - `e`, the `minfun` program to be converted
      - `arg`, the argument name for the nearest enclosing function, or `#f` if `e` is not in any function. Note that `evaluate` allows using this `minfun` variable.
      - `arg-stack`, an ordered list of argument names for the enclosing functions, *not including* the nearest enclosing function. So the first element of `arg-stack` is the name of the second nearest function's argument. The list is empty unless `e` is in at least two functions.
      - `env-var`, the `minfun` variable name used in the result to access the environment of free variables.

      Hints:

      - You need a case for every kind of `minfun` expression. Only 3 such cases are difficult.

- The case for variables should use `get-env-exp`, described below, unless the variable is the same as `arg`. (You can compare `minfun` variables with Scheme's `eq?`.)
- The case for functions should translate the function body using rather different arguments for `convert-body`. Remember the result for the function case is a pair. In particular, we need a new `env-var`; call the Scheme primitive `(gensym)` to get one.
- The case for applications should create a function of two arguments and apply this function (because we do not have let). Use `(gensym)` to create fresh names for the parameters.

(b) `get-env-exp` does the actual work of converting a free variable into the correct environment-access expression. It should take a `minfun` variable (the one being converted), an `arg-stack` (as described above), and a `minfun` expression `env-exp`. When `convert-body` calls `get-env-exp`, it will use its `env-var` for `env-exp`. If the `minfun` variable is the n^{th} element of `arg-stack`, then `get-env-exp` returns a `minfun` expression that, when evaluated, gets the n^{th} element of the environment in `env-var`. (However, there is no need to compute what n is.) You can call `error` if the variable is not in `arg-stack`.

3. **(Extra Credit)** Write a second version of closure conversion (called `convert2`) that does not put unused variables in function environments. That is, if `x` is in scope in the body of a function but the function body does not actually have an occurrence of `x`, then the environment for the converted function should not have a "slot" for `x`. Warning: The sample solution does not do the extra credit.

Turn-in Instructions

- Put all your solutions in one file, `lastname_hw5.scm`, where `lastname` is replaced with your last name.
- Line 1 of your `.scm` file should include 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*.