Essentials

Assigned: Tuesday March 27, 2012.
Due: on or before 9:00 PM on Monday April 9, 2012

When doing this homework, first create a directory named `HW3` somewhere in your home directory (e.g. as a subdirectory of a `cse305` directory). Place your solution to each question in a file or set of files, as indicated, in the `HW3` directory. When you're ready to submit, zip up the `HW3` directory and its contents (i.e. `zip -r HW3.zip HW3`), and use the `submit_cse305` command to submit your `HW3.zip` file.

It is very important that you pay close attention to the naming conventions for files and directories for your homework submissions in this course. Having uniform names for all student submissions makes grading submissions much easier. If you do not adhere to the naming conventions, grading of your work will be delayed, or it may simply be returned to you ungraded for you to correct the names.

Functional language interpreter, part II (100 points)

This homework is the second of a two-part homework in which you will build an interpreter for a Scheme-like language. In homework 2 you gained a hands-on understanding of how programming language execution works, in particular: environments, expression evaluations, primitive (built-in) function calls. You experienced how to implement the same functionality in several different languages, and gained breadth of experience. In this homework you will gain depth of experience, by extending a hw2 interpreter written in ML, to incorporate several additional features. You will incorporate lambda forms (user-defined functions), closures, syntactic sugar for let forms, and several new primitives. The new functionality is indicated in red text; you are responsible for all functionality, both black and red.

Write an interactive Read-Eval-Print-Loop (REPL) that handles the following expressions as input; note that whitespace (space, tab, newline) is permitted between any tokens of the language:

- Numbers (sequences of digits) are expressions. A number expression is evaluated to its base 10 interpretation. Note that all number literals are non-negative. To express a negative number you must compute it, as in (- 0 3).
- Boolean literals (#true and #false) are expressions. #true evaluates to #true, and #false to #false.
- Names (sequences of letters and digits, starting with a letter) are expressions. A name expression is evaluated by looking the name up in an environment, starting with the current environment, following static links, until the name is found. If it is not found, an error occurs; print '?error: unbound name' and re-prompt.
- Lambda forms, of the form '(' "lambda" '(' params ')' expr ')', where params is a whitespace-separated list of names, is an expression. A lambda form is evaluated by

creating a closure. A closure stores the current environment, the parameters, and the expression that is the body of the lambda form.

- nil is a literal expression; represent as Nil. Nil evaluates to itself, and prints as "Nil". (See below for use of Nil with Cons.)
- Applications, of the form '(' expr1 expr2 expr3 ')' where expr1, expr2 and expr3 are expressions, are expressions. An application is evaluated by first evaluating expr1, expr2 and expr3, and applying the value of expr1 to the values of expr2 and expr3 (in that order).

  A closure is applied by creating a binding for each parameter to its corresponding argument in a new environment, whose parent environment is the closure's environment. This new environment becomes new current environment; the body of the closure is evaluated with respect to the current environment (which is this new environment).

  A closure prints as "<closure>" (without the quotes).

  A primitive is applied as described; the following primitive functions must be supported:
  - Primitives: +, -, * and / are bound to the integer addition, subtraction, multiplication and division operations. A primitive arithmetic operator is applied as a binary operator by calling the corresponding built-in function; i.e. (/ 10 4) evaluates to 2.
  - Primitives: &, |, and ! are bound to the boolean *and*, *or* and *not* operations. Notice that while & and | are binary operators, ! is a unary operator. Each of these primitive operators is applied by calling the corresponding built-in function.
  - Primitives: < and = are bound to the less than and equals operations, respectively. Each of these primitive operators is applied by calling the corresponding built-in function.
  - Primitives: cons, car, and cdr are bound to the pair constructor, and first-member-of-pair and second-member-of-pair accessors, respectively. For example:
    - (cons e1 e2) returns a Cons(v1,v2) where v1 is the value of e1, v2 is the value of v2.
    - (car exp) returns v1 where the value of exp is Cons(v1,v2).
    - (cdr exp) returns v2 where the value of exp is Cons(v1,v2).
    - Treat Cons as an expression which evaluates to itself.
    - Pretty-printing rules for Cons(v1,v2) are the same as Scheme; a description of how to pretty-print a Cons is given below. In this description p1 is the pretty-print form of v1, p2 is the pretty-print form of v2, *p2 is the pretty-printed form of v2 without the outer parentheses. The pretty-print rules are:
      - if v2 is neither a Cons nor Nil then Cons(v1,v2) prints as (p1 . p2)
      - if v2 is a Cons then Cons(v1,v2) prints as (p1 *p2)
      - v2 is Nil then Cons(v1,v2) prints as (p1)
    - You are NOT expected to define your reader to recognize the pretty-printed Cons forms described above. Cons values must be created using the cons primitive.

A primitive prints as <primitive:*name*>, where *name* is the name of the primitive; for example, <primitive:+> or <primitive:cons>.

- Special forms are evaluated according to special rules:
  - (define <var> <expr>) is evaluated by first evaluating <expr>, and binding, in the current environment, the name <var> to that value. A define special form does not have a value, and so nothing is printed as a result. Using define to bind a name that is already bound is an error occurs; print '?error: bound name' and re-prompt.
  - (load <string>) loads definitions and expressions from the file named by <string>. The value of a load special form is #true if the contents of the file loads properly, #false otherwise. N.B. for now the load special form is the only place a string can appear. A string is a sequence of characters enclosed in double quotation marks, as in "filename.txt"
  - (quit) exits the repl, and prints "Bye!" just before exiting.
  - (if expr1 expr2 expr3) is a conditional expression. The evaluation of the conditional proceeds but first evaluating expr1; if expr1 has value #true, the value of the form is the value of expr2, else it is the value of expr3. expr1 is always evaluated, but exactly one of expr2 and expr3 is evaluated.
- Syntactic sugar forms are translated into their underlying form
  - (let ((v1 e1) (v2 e2) … (vk ek)) exp) is syntactic sugar for ((lambda (v1 v2 … vk) exp) e1 e2 … ek)

The prompt of your repl must be 'repl>'. The repl must print the value of each expression it evaluates (see examples below).

Naming conventions

The programs you write may define several functions, named as you see fit, but you must define a public/exported function named repl of no arguments, as follows:

- ML: in a file named hw3.ml define a function named `repl` of no arguments. Ensure that `repl()` is called when the file is loaded.

Helpful hints

1. **Start early**.
2. You may again assume that all input will be well-formed. The provided interpreter does *some* error checking, and reports errors in *some* situations. You do not need to go beyond this, though you may.

3. You must complete the interpreter in ML. You may build your HW3 solution starting from your HW2 ML solution, or the provided ML solution. I highly recommend starting with the provided solution.

Examples

Assume the file "defs.scm" contains:

```
(define a 3)
(define b 5)
(define c (* a b))
(define foo (lambda (a b) (+ a b)))
(define e (foo a b))
(define fact (lambda (x)  (if (= x 0) 1 (* x (fact (- x 1))))))
```

The following examples show what the behavior of the repl should be for a selection of inputs; although this has been proofread quite carefully, note that this is hand-typed and may contain errors --- if in doubt, ask!

```
repl> (load "defs.scm")
repl> a
3
repl> b
5
repl> c
15
repl> e
8
repl> d
?error: unbound name
repl> (define d (* a (- c b)))
repl> d
30
repl> (define d 12)
?error: bound name
repl> foo
<closure>
repl> fact
<closure>
repl> (fact 3)
6
repl> (fact 5)
120
repl> (define locals (let ((a 12)) (lambda (b) (+ b a))))
repl> (locals 5)
17
repl> a
3
repl> (define free (lambda (b) (+ b a)))
repl> (free 5)
8
repl> (define adder (lambda (x) (lambda (y) (+ x y))))
repl> (define c1 (adder 3))
repl> (c1 4)
```

```
7
repl> (c1 1)
4
repl> (define c2 (adder 7))
repl> (c2 4)
11
repl> (c2 1)
8
repl> (c2 (c1 1))
11
repl> (define f (lambda (x) (+ x 1)))
repl> (define g (lambda (x) (* 2 x)))
repl> (define o (lambda (x y) (lambda (z) (x (y z)))))
repl> ((o f g) 3)
7
repl> ((o g f) 3)
8
repl> (define lst (cons 1 (cons 2 (cons 3 nil))))
repl> lst
(1 2 3)
repl> (define pr (cons 1 2))
(1 . 2)
repl> (define flist (cons f (cons g nil)))
repl> flist
(<closure> <closure>)
repl> (quit)
```