Assignment 4    B521    Fall 2005
Representation Independent Interpreters /
Logic Programming I

Revised 22 September
Due Sunday, September 25 at 6 p.m.

• Make sure you have read chapter three of EOPL3 before beginning Part I, and chapters one and nine of The Reasoned Schemer thoroughly before beginning Parts II and III of this assignment.

• This is a long assignment. Get started early!

• Place your code in a file named a4.ss and submit it to Vincent.

Part I: Interpreters

Write the following four interpreters:

1. An interpreter value-of/proc/proc that uses procedural representation for closures and procedural representation for environments.

2. An interpreter value-of/proc/ds that uses procedural representation for closures and data structure representation for environments.

3. An interpreter value-of/ds/proc that uses data structure representation for closures and procedural representation for environments.

4. An interpreter value-of/ds/ds that uses data structure representation for closures and data structure representation for environments.
Your interpreters should handle the following expression types.

```scheme
(define-datatype expression expression?
  (const-exp
   (n number?))
  (bool-exp
   (b boolean?))
  (var-exp
   (v symbol?))
  (if-exp
   (t expression?)
   (c expression?)
   (a expression?))
  (zero?-exp
   (e expression?))
  (diff-exp
   (e1 expression?)
   (e2 expression?))
  (prod-exp
   (e1 expression?)
   (e2 expression?))
  (proc-exp
   (v symbol?)
   (e expression?))
  (call-exp
   (e1 expression?)
   (e2 expression??)))
```

These expression types are described in chapter three of *EOPL3*, with the exception of `bool-exp`, which is the type of the boolean constants #t and #f, and `prod-exp`, which represents the product of two expressions, both of which must evaluate to constant (numeric) values. See chapter three for the semantics of this language. Essentially all of the code for the representation-dependent interpreter can be found in chapter three (and in your notes from class).

For each representation-independent interpreter, make sure you define the five following helper procedure: `base-env`, `extend-env`, `apply-env`, `make-closure`, and `apply-closure`. To make sure all of your helper functions have unique names, suffix the helpers for a given interpreter with the same suffix as the interpreter (for example, `value-of/proc/proc` has `base-env/proc/proc`, `extend-env/proc/proc`, etc).
Make sure you include the `expressed-value→scheme-value` helper procedure in your definition.

Insert the following code into your file. When it loads, the results of the test cases should confirm that all four interpreters correctly compute the factorial of five.

```
(define-syntax test
  (syntax-rules ()
    [(_ name e1 e2)
      (let ([v1 e1] [v2 e2])
        (if (equal? v1 v2)
          (printf "Test \s passed.\n" name)
          (error 'test "\nTest \s failed.\n" name \e1 \v1 \e2 \v2))))])

(define fact-5
  (call-exp
    (call-exp
      (proc-exp '!
        (call-exp
          (call-exp
            (var-exp '!))
        (var-exp '!))
      (var-exp 'n))))
    (proc-exp '!
      (proc-exp 'n
        (if-exp (zero?-exp (var-exp 'n))
          (const-exp 1)
          (prod-exp
            (var-exp 'n)
            (call-exp
              (call-exp
                (var-exp '!))
            (var-exp '!))
          (diff-exp (var-exp 'n) (const-exp 1)))))))
  (const-exp 5)))
```
(test "proc/proc"
   (expressed-value→scheme-value (value-of/proc/proc fact-5 (base-env/proc/proc)))
   120)

(test "proc/ds"
   (expressed-value→scheme-value (value-of/proc/ds fact-5 (base-env/proc/ds)))
   120)

(test "ds/proc"
   (expressed-value→scheme-value (value-of/ds/proc fact-5 (base-env/ds/proc)))
   120)

(test "ds/ds"
   (expressed-value→scheme-value (value-of/ds/ds fact-5 (base-env/ds/ds)))
   120)

In total, you should have four interpreters, five helper procedures for each, and the expressed-value→scheme-value helper procedure, for a total of twenty-five procedure definitions for Part I.
Part II: Simple run expressions

Write the answer to the following problems using your knowledge of the Mini-Kanren logic programming system. With each answer, give a full-sentence intuitive justification as to why and how you came up with your answer. Answers with no justification receive no credit.

Remember that #s is written as succeed, #u is written as fail, ≡ is written as ==, cond is written as conde, run* is written as run*, run5 is written as run 5, and anyo is written as anyo.

1. What is the value of
   \( \text{run}^* (q) \\
   (≡ 5 q) \)

2. What is the value of
   \( \text{run}^* (q) \\
   (≡ 5 q) \\
   (≡ 6 q) \)

3. What is the value of
   \( \text{run}^* (q) \\
   \text{conde} \\
   (≡ 5 q) #s \\
   (≡ 6 q) \)

4. What is the value of
   \( \text{run}^1 (q) \\
   \text{conde} \\
   (≡ 5 q) #s \\
   (≡ 6 q) \)

5. What is the value of
   \( \text{run}^* (q) \\
   \text{conde} \\
   (≡ 5 q) #s \\
   (≡ 6 q) \)
6. What is the value of
   \((\text{run}^2 (q))\)
   \((\equiv 5 \; q)\)
   \((\text{cond}^e ((\text{cond}^e ((\equiv 5 \; q) (\equiv 6 \; q)))) (\equiv 5 \; q))))\)

7. What is the value of
   \((\text{let} ((x (\text{var} \ 'x)) (y (\text{var} \ 'y)) (\text{unify} \; x \; 5 \; '((,y \ . \ 4) (,x \ . \ y))))))\)

8. What is the value of
   \((\text{let} ((x (\text{var} \ 'x)) (y (\text{var} \ 'y)) (z (\text{var} \ 'z)) (\text{reify} (\text{walk}^* \; x \; '((,y \ . \ z) (,x \ . \ y))))))\)

**Part III: always and never**

Suppose *always* and *never* are defined as
\[
\begin{align*}
\text{(define any}^o & \text{)} \\
\text{(lambda (g))} & \text{)} \\
\text{(cond}^e & \text{)} \\
(g \text{ } #s) & \text{)} \\
\text{(else (any}^o g)))) & \text{)} \\
\text{(define always (any}^o #s)) & \text{)} \\
\text{(define never (any}^o #u)) & \text{)}
\end{align*}
\]

Write the answer and give a justification to the following questions.

1. What is the value of
   \((\text{run}^5 (q) \text{ always}))\)
2. What is the value of
   \( \text{run}^5 (q) \)
   \( \text{never} \)

3. What is the value of
   \( \text{run}^1 (q) \)
   \( \text{always} \)
   \( \#u \)

4. What is the value of
   \( \text{run}^1 (q) \)
   \( \#u \)
   \( \text{always} \)

5. What is the value of
   \( \text{run}^1 (q) \)
   \( \text{cond}^e \)
   \( (never) \)
   \( (#s)) \)

6. What is the value of
   \( \text{run}^1 (q) \)
   \( \text{cond}^e \)
   \( (#s) \)
   \( (never)) \)

**Brainteaser**

The brainteaser does not count unless the rest of the assignment is attempted.

Write one interpreter \textit{value-of} that accepts as additional arguments the representation-dependent helper functions (seven total arguments) for a given closure/environment representation and works the same as the interpreter above.