Problem Set 7:

Problem 1

Consider the following BNF specification:

\[
\text{<BExpression> ::= True}
\]

\[
| \quad \text{not <BExpression>}
\]

\[
| \quad \text{<BExpression> and <BExpression>}
\]

Where true, not, and and are keywords.

We can define the rank of an expression \( E \in \text{<BExpression>} \) by the following definition:

- \( \text{rank(true)} = 0 \)
- \( \text{rank(not E)} = \text{rank(E)} + 1 \)
- \( \text{rank(E_1 and E_2)} = \max\{\text{rank(E_1)}, \text{rank(E_2)}\} + 1 \)

Prove by structural induction that every expression \( E \in \text{<BExpression>} \) has a rank greater than or equal to 0.

Solution:

\( E = \text{true} \): immediate, \( \text{rank(E)} = 0 \) by def of rank.

\( E = \text{not } e \): By assumption, \( e \in \text{<Expression>} \). Hence \( \text{rank(e)} \geq 0 \). By a simple calculation, we have \( \text{rank(E)} \geq 1 \geq 0 \).

\( E = e_1 \text{ and } e_2 \):

By assumption, \( e_1, e_2 \in \text{<Expression>} \). Hence, \( \text{rank(e_1)} \geq 0, \text{rank(e_2)} \geq 0 \), and \( \max\{\text{rank(e_1)}, \text{rank(e_2)}\} \geq 0 \). By a simple calculation, we have \( \text{rank( e_1 and e_2 )} \geq 1 \), as required.

Q.E.D.

Problem 2

Start with “xmlscheme-10.scm”, which can be found on the course page. Add a “let*” construct to the defined language:

\[
\text{<expression> ::= “<let*” <declarations> <expressions> “/>”}
\]

let-star-exp (decls expr)

The “let*” should exhibit the same behavior as Scheme’s “let*” declaration.
Solution:

(define expression-spec
 '(
   (expression
    ("<let*" declarations expression "/">") let-star-exp)
   ...)
)

(define grammar-spec
 '(
   (expression
    ("<let*" declarations expression "/">") let-star-exp)
   ...)
)

(define eval-let-stars
 (lambda (ids exps env)
   (if (null? ids)
     env
     (eval-let-stars
      (cdr ids)
      (cdr exps)
      (extend-env (list (car ids))
       (list (eval-expression (car exps) env))
       env))
     )
   )
)

(define eval-expression
 (lambda (exp env)
   (cases expression exp
   ...
   (let-star-exp (decls body)
     (let ((new-env (eval-let-stars (get-ids decls) (get-rands decls) env)))
      (eval-expression body new-env))
     ...
   )))
)
Problem 3

Start with “xmlscheme-10.scm” or the language developed in Problem 2.
Define the procedure modulo for positive numbers in XMLScheme.

<invoke <reference value = modulo /><arguments <integer value = 13 /> <integer value = 4 /> /> />
⇒ 1
<invoke <reference value = modulo /> <arguments <integer value = 12 /> <integer value = 4 /> /> />
⇒ 0
<invoke <reference value = modulo /> <arguments <integer value = 12 /> <integer value = 0 /> /> />
⇒/: division by zero
Solution:
<let
  <declarations
    <declaration
      <variable value = modulo />
    <proc
      <params <param value = x /> <param value = y /> />
    % assume: x >= 0 && y >= 0
    <if
      <condition <reference value = y /> />
    <then
      % if x == 0 then loop immediately returns 0
      <loop
        <declarations
          <declaration
            <variable value = rest /> <reference value = x /> />
        <conditions
          <greater <arguments <reference value = rest /> <reference value = y /> />
          <equal <arguments <reference value = rest /> <reference value = y /> />
        />
        % identity
        <increments
          <reference value = rest />
        />
        <set
          <variable value = rest />
          <sub <arguments <reference value = rest /> <reference value = y /> />
        />
      />
      /
    /
  </else
  % force error (modulo x 0) is not defined
  <div <arguments <reference value = x /> <reference value = y /> />
  /
  /
  % return modulo
  <reference value = modulo />
/>
**Problem 4**

Start with “xmlscheme-10.scm” or the language developed in Problem 2 again. Now add modulo to the initial environment of the interpreter. That is, add a binding from the name “modulo” to a function (or closure) implementing the behavior of the modulo function to the initial environment init-env. Test your implementation with the samples given in Problem 3.

Solution:

```scheme
(define init-env
  (lambda ()
    (extend-env '(i v x modulo)
      (list 1 5 10 (closure-mod) )
      (empty-env))
  )
)

(define datatype procval procval?
  ...
  (closure-mod)
)

(define apply-procval
  (lambda (proc args calling-env)
    (cases procval proc
      ...
      (closure-mod ()
        (if (eqv? (length args) 2)
          (eval (append '(modulo) args))
          (eopl-error 'modulo "Illegal number of arguments!")
        ))
      )
    )
  )
)

> (read-eval-loop)
$ <invoke <reference value = modulo /> <arguments <integer value = 4/> <integer value = 3/> /> /> 1
30

Total: 10 + 20 + 30 + 30 = 90

**Submission deadline: Tuesday, April 10, 2007, 2:10 p.m.**

**Submission procedure: on paper in class.**