Fall, 2000 Name: _____

22C:54 — Programming Language Concepts

Test on *EOPL* Chapters 2.2-2.3

This test has 3 questions and pages numbered 1 through 11.

Reminders

For this test, you can use one (1) page (8.5 by 11 inches, one (1) side, no less than 8pt font) of notes. No photo-reduction is permitted. These notes are to be handed in at the end of the test. Use of other notes or failure to follow these instructions will be considered cheating. Please put your name in the top right corner of your notes.

During the test, if you need more space for an answer, use the back of a page. Note when you do that on the front.

This test is timed. We will not grade your test if you try to take more than the time allowed. Therefore, before you begin, please take a moment to look over the entire test so that you can budget your time.

For programs, indentation is important to us for "clarity" points; if your code is sloppy or hard to read, you will lose points. We may take off a small amount if you do not give `deftype` declarations for recursive helping procedures. However, you do not have to write such comments for procedures for which the type is stated in the problem. Correct syntax also matters. Check your code over for syntax errors. You will lose points if your code has syntax errors.

Subset of Scheme You May Use.

Unless otherwise stated in a problem, when solving problems you may only use standard features of Scheme that we discussed in class, and helping functions that you define yourself. The standard is defined by the *Revised*^{5} *Report on the Algorithmic Language Scheme*.

Parts of Scheme You May *Not* Use.

Unless otherwise stated in a problem, you are prohibited from defining your own macros, and from using internal defines, all the input and output facilities, and the following keywords and procedures. (Don't worry if you don't know what these are.)

```
call-with-current-continuation  do      set!    set-car!        set-cdr!
string-set!     string-fill!    string->list
vector-set!     vector-fill!    vector->list
```

1. This is a problem about static properties of variables. Consider the following Scheme expression, in which `lambda` is the only special form used. (It has been written with extra space between the lines to aid in the drawings required below.)

```
((lambda (parms env)

   ((lambda (types)

      (check parms env))

    (map (lambda (p) (second p)) (rest parms))))

 (second expr) (rest env))
```

   (a) (5 points) In the above expression, draw arrows from each bound ⟨varref⟩ to the corresponding formal parameter declaration. (You're supposed to know what a "bound ⟨varref⟩" is.)

   (b) (5 points) In the above expression, also draw contours showing the regions of the declarations.

   (c) (5 points) Write, below, in set brackets, the entire set of the bound variables in above expression. For example, write $\{v, w\}$, if the bound variables are v and w. If there are no bound variables, write $\{\}$. (You're supposed to know what a "bound variable" is.)

   (d) (10 points) Write, below, in set brackets, the entire set of the free variables in above expression. For example, write $\{v, w\}$, if the free variables are v and w. If there are no free variables, write $\{\}$. (You're supposed to know what a "free variable" is.)

   (e) (15 points) Give the lexical address form of the above expression by filling in the rest of the expression below. (You can choose the position numbers for the free variables arbitrarily.)

```
((lambda (parms env)
   ((lambda (types)
```

2. (30 points) Write a procedure,

```
(deftype index-of-second (-> (char (vector char)) number))
```

that takes a character, `ch`, and a vector of characters, `vec`, and returns the index of the second occurrence of `ch` in `vec`, if there is one. If `ch` does not occur in `vec` twice, `index-of-second` should return -1.

The following are examples.

```
(index-of-second #\c (vector #\x #\c #\x #\c)) ==> 3
(index-of-second #\c (vector #\x #\c #\x)) ==> -1
(index-of-second #\a '#(#\b #\c)) ==> -1
(index-of-second #\x '#(#\a #\_ #\g #\o #\o #\d #\_ #\e #\g #\g)) ==> -1
(index-of-second #\o '#(#\a #\_ #\g #\o #\o #\d #\_ #\e #\g #\g)) ==> 4
(index-of-second #\g '#(#\a #\_ #\g #\o #\o #\d #\_ #\e #\g #\g)) ==> 8
```

Hints: Recall that a character literal is written `#\c` in Scheme. You can use `equal?` to compare characters for equality. Remember that Scheme vectors have indexes that start with zero (0). You can use the procedures

```
vector-length : (-> ((vector T)) number)
vector-ref : (-> ((vector T) number) T)
```

to get the length and to access an element of a vector, respectively. However, you may *not* use `vector->list`.

3. (30 points) Consider the following grammar for a multiple-argument lambda calculus with numeric literals and assignment expressions. Note that the variable being assigned to in an assignment expression is treated as a ⟨varref⟩; it does *not* declare a variable.

   ⟨exp⟩ ::= ⟨varref⟩
   | ⟨lit⟩
   | `(` `lambda` `(` {⟨var⟩}* `)` ⟨exp⟩ `)`
   | `(` ⟨exp⟩ {⟨exp⟩}* `)`
   | `(` `assign` ⟨varref⟩ ⟨exp⟩ `)`
   ⟨varref⟩ ::= ⟨var⟩
   ⟨var⟩ ::= ⟨symbol⟩
   ⟨lit⟩ ::= ⟨number⟩

   You can use the following helping procedures for this grammar given on page 6. (Hint, just look at their types.) Using the these helping procedures, and helping procedurs for sets given on page 9. Write a procedure, `free-vars`

   ```
   (deftype free-vars (-> (exp) (set symbol)))
   ```

   that returns the set of free variables in a given ⟨exp⟩. There are some examples on the next page.

The following are examples for the problem on the previous page.

```
(free-vars (parse-exp 'x))
  = (set-of 'x)

(free-vars (parse-exp 3))
  = (set-of )

(free-vars (parse-exp '(f x 3 x)))
  = (set-of 'f 'x)

(free-vars (parse-exp '(lambda (f g h) (f x 3 x))))
  = (set-of 'x)

(free-vars (parse-exp '(assign x 3)))
  = (set-of 'x)

(free-vars (parse-exp '(assign x (plus 3 x))))
  = (set-of 'x 'plus)

(free-vars (parse-exp '(assign g (lambda (f g h) (f x 3 x)))))
  = (set-of 'g 'x)

(free-vars
 (parse-exp '(hmm (assign g (lambda (f g h) (lambda (x) (f x 3 x)))))))
  = (set-of 'hmm 'g)

(free-vars
 (parse-exp
  '((assign
     h
     (assign
      r
      (assign g (lambda (f g h) (lambda (x) (f x 3 x))))))
    (plus 3 (assign x (plus 4 0))))))
  = (set-of 'h 'r 'g 'x 'plus)

(free-vars
 (parse-exp
  '(lambda (h r g x)
     ((assign
       h
       (assign
        r
        (assign g (lambda (f g h) (lambda (x) (f x 3 x))))))
      (plus 3 (assign x (plus 4 0)))))))
  = (set-of 'plus)
```

```
;;; $Id: lambda-multiple-assign.scm,v 1.1 2000/10/25 01:58:34 leavens Exp leavens $
;;; An ADT for the untyped lambda multiple-argument calculus
;;; with numeric literals and assignment expressions

;;;     <exp> ::= <varref>
;;;             | <lit>
;;;             | ( lambda ( {<var>}* ) <exp> )
;;;             | ( <exp> {<exp>}* )
;;;             | ( assign <varref> <exp> )
;;;     <varref> ::= <var>
;;;     <var> ::= <symbol>
;;;     <lit> ::= <number>

trustme!  ;; suppress spurious type errors

(deftype exp? (-> (datum) boolean))
(deftype varref? (-> (exp) boolean))
(deftype lit? (-> (exp) boolean))
(deftype lambda? (-> (exp) boolean))
(deftype app? (-> (exp) boolean))
(deftype assign? (-> (exp) boolean))
(deftype varref->var (-> (exp) symbol))
(deftype lit->number (-> (exp) number))
(deftype lambda->formals (-> (exp) (list symbol)))
(deftype lambda->body (-> (exp) exp))
(deftype app->rator (-> (exp) exp))
(deftype app->rands (-> (exp) (list exp)))
(deftype assign->name (-> (exp) symbol))
(deftype assign->value (-> (exp) exp))
(deftype make-varref (-> (symbol) exp))
(deftype make-lit (-> (number) exp))
(deftype make-lambda (-> ((list symbol) exp) exp))
(deftype make-app (-> (exp (list exp)) exp))
(deftype make-assign (-> (symbol exp) exp))
(deftype parse-exp (-> (datum) exp))

(load-quietly-from-lib "all.scm")

(defrep exp (list datum))

(define exp?
  (lambda (exp)
    (or (varref? exp)
        (lit? exp)
        (and (lambda? exp)
             (exp? (lambda->body exp)))
        (and (app? exp)
             (exp? (app->rator exp))
```

```
             (all exp? (app->rands exp)))
        (and (assign? exp)
             (exp? (assign->value exp))))))

(deftype is?-maker (-> (datum (-> ((list datum)) boolean))
                       (-> ((list datum)) boolean)))
(define is?-maker
  (lambda (tag predicate)
    (lambda (ld)
      (and (list? ld)
           (not (null? ld))
           (eq? tag (car ld))
           (predicate ld)))))

(deftype no-additional-constraints (-> (T) boolean))
(define no-additional-constraints
  (lambda (ld) #t))

(define varref? (is?-maker 'varref no-additional-constraints))
(define lit? (is?-maker 'lit no-additional-constraints))

(define lambda?
  (is?-maker 'lambda
             (lambda (ld)
               (and (= (length ld) 3)
                    (let ((formals (cadr ld)))
                      (and (list? formals)
                           (all symbol? formals)))))))

(define app?
  (is?-maker 'app
             (lambda (ld)
               (>= (length ld) 2))))

(define assign?
  (is?-maker 'assign
             (lambda (ld)
               (and (= (length ld) 3)
                    (symbol? (cadr ld))))))

(define varref->var cadr) ;; REQUIRES: the argument is a varref
(define lit->number cadr) ;; REQUIRES: the argument is a lit

(define lambda->formals cadr) ;; REQUIRES: the argument is a lambda
(define lambda->body caddr)   ;; REQUIRES: the argument is a lambda

(define app->rator cadr) ;; REQUIRES: the argument is an application
(define app->rands caddr) ;; REQUIRES: the argument is an application
```

```
(define assign->name cadr) ;; REQUIRES: the argument is an assign
(define assign->value caddr) ;; REQUIRES: the argument is an assign

(define make-varref (lambda (sym) (list 'varref sym)))
(define make-lit (lambda (num) (list 'lit num)))
(define make-lambda (lambda (formals body) (list 'lambda formals body)))
(define make-app (lambda (rator rands) (list 'app rator rands)))
(define make-assign (lambda (name value) (list 'assign name value)))

(define parse-exp
  (lambda (exp)
    (cond
     ((symbol? exp) (make-varref exp))
     ((number? exp) (make-lit exp))
     ((lambda? exp) (make-lambda (cadr exp) (parse-exp (caddr exp))))
     ((assign? exp) (make-assign (cadr exp) (parse-exp (caddr exp))))
     ((and (list? exp) (<= 1 (length exp)))
      (make-app (parse-exp (car exp))
                (map parse-exp (cdr exp))))
     (else (error "parse-exp: bad syntax in expression: " exp)))))
```

```
;;; $Id: set-ops.scm,v 1.1 2000/10/23 18:13:42 leavens Exp leavens $

(deftype the-empty-set (set T))
(deftype set-empty? (-> ((set T)) boolean))
(deftype set-size (-> ((set T)) number))
(deftype set-of (-> (T ...) (set T)))
(deftype set-member? (-> (T (set T)) boolean))
(deftype set-one-elem (-> ((set T)) T))
(deftype set-rest (-> ((set T)) (set T)))
(deftype set-subset? (-> ((set T) (set T)) boolean))
(deftype set-equal? (-> ((set T) (set T)) boolean))
(deftype set-add (-> (T (set T)) (set T)))
(deftype set-remove (-> (T (set T)) (set T)))
(deftype set-union (-> ((set T) (set T)) (set T)))
(deftype set-minus (-> ((set T) (set T)) (set T)))
(deftype set-intersect (-> ((set T) (set T)) (set T)))
(deftype set-union-list (-> ((list (set T))) (set T)))
(deftype set-union* (-> ((set T) ...) (set T)))

(defrep (set T) (list T))

(define the-empty-set '())

(define set-empty?
  (lambda (s) (null? s)))

(define set-size (lambda (s) (length s)))

(define set-of
  (lambda args
    (if (null? args)
        the-empty-set
        (set-add (car args) (apply set-of (cdr args))))))

(define set-member?
  (lambda (e S)
    (and (not (set-empty? S))
         (or (equal? e (car S))
             (set-member? e (cdr S))))))

(define set-one-elem
  (lambda (s)
    ;; REQUIRES: s is not empty
    (car s)))

(define set-rest
  (lambda (s)
    ;; REQUIRES: s is not empty
```

```
    (cdr s)))

(define set-subset?
  (lambda (S1 S2)
    (or (set-empty? S1)
        (and (set-member? (set-one-elem S1) S2)
             (set-subset? (set-rest S1) S2)))))

(define set-equal?
  (lambda (S1 S2)
    (and (set-subset? S1 S2)
         (set-subset? S2 S1))))

(define set-add
  (lambda (e S)
    (if (set-member? e S)
        S
        (cons e S))))

(define set-remove
  (lambda (e S)
    (if (set-empty? S)
        S
        (if (equal? e (car S))
            (cdr S)
            (cons (car S) (set-remove e (cdr S)))))))

(define set-union
  (lambda (S1 S2)
    (if (set-empty? S1)
        S2
        (set-add (car S1) (set-union (cdr S1) S2))
        )))

(define set-minus
  (lambda (S1 S2)
    ;; ENSURES: result contains just the elements of S1 that are NOT in S2
    (if (set-empty? S2)
        S1
        (set-remove (car S2) (set-minus S1 (cdr S2)))
        )))

(define set-intersect
  (lambda (S1 S2)
    ;; ENSURES: result contains just the elements in both S1 AND S2
    (if (set-empty? S1)
        '()
        (if (set-member? (car S1) S2)
```

```
            (cons (car S1) (set-intersect (cdr S1) S2))
            (set-intersect (cdr S1) S2)
            ))))

(define set-union-list
  (lambda (lset)
    (if (null? lset)
        the-empty-set
        (set-union (car lset)
                   (set-union-list (cdr lset))))))

(define set-union*
  (lambda lset
    (set-union-list lset)))
```