Com S 342 Name: _____
Fall 2001 TA (or Section): _____

Principles of Programming Languages

Answers to Exam 4 on Interpreters and Language Semantics

This test has 8 questions and pages numbered 1 through 9.

Reminders

For this test, you can use one (1) page (8.5 by 11 inches, one (1) side, no less than 9pt font) of notes. Handwriting is okay. No photo-reduction is permitted. Don't use anything with printing on the other side, please. These notes are to be handed in at the end of the test. Have your name in the top right corner. Use of other notes or failure to follow these instructions will be considered cheating.

If you need more space, 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. Correct syntax also matters. Check your code over for syntax errors. You will lose points if your code has syntax errors.

You can use helping procedures whenever you like. If you write recursive helping procedures, please give a deftype declaration for them.

This page just contains reference material for problems on later pages.
Types of helpers from the chapter 3 interpreters used on this test. These ADTs correspond roughly to those in section 3.7 of the text.

```
eopl:error : (-> (symbol string datum ...) poof)

;; ---- ProcVal (procedure values) ADT --------
procval? : (type-predicate-for procval)
closure : (-> ((list-of symbol) expression environment) procval)
apply-procval : (-> (procval (list-of Expressed-Value)) Expressed-Value)

;; ---- Expressed-Value ADT --------
;; upcasts
number->expressed : (-> (number) Expressed-Value)
procval->expressed : (-> (Procval) Expressed-Value)
list->expressed : (-> ((list-of Expressed-Value)) Expressed-Value)
;; downcasts
expressed->number : (-> (Expressed-Value) number)
expressed->procval : (-> (Expressed-Value) Procval)
expressed->list : (-> (Expressed-Value) (list-of Expressed-Value))

;; ---- reference ADT --------
a-ref : (forall (T) (-> (number (vector-of T)) (ref-of T)))
deref : (forall (T) (-> ((ref-of T)) T))
setref! : (forall (T) (-> ((ref-of T) T) void))

;; ---- environment ADT ----------
;; type predicate
environment? : (type-predicate-for environment)
;; constructors
empty-env : (-> () environment)
extend-env : (-> ((list-of symbol) (list-of Expressed-Value) environment)
                 environment)
extend-env-recursively : (-> ((list-of symbol) (list-of (list-of symbol))
                              (list-of expression) environment)
                             environment)
;; observers
apply-env : (-> (environment symbol) Expressed-Value)
apply-env-ref : (-> (environment symbol) (ref-of Expressed-Value))
defined-in-env? : (-> (environment symbol) boolean)
```

1. (5 points) Complete the following definition of the defined language interpreter's `init-env` procedure so that it defines the name `five` to be the number 5. (You don't have to worry about the values of any names other than `five`.) Once this is done, in the defined language we would have the following examples:

```
--> five
5
--> +(five, five)
10
```

Your code must type check to receive full credit. Hint: look at the operations of the standard ADTs on page 2.

Answer:

```
(deftype init-env (-> () environment))
(define init-env
  (lambda ()
    (extend-env
      '(five)
      (map number->expressed '(5))
      (empty-env))))
```

2. (5 points) This is a question about local binding in the defined language. In this problem, the defined language is extended with lists, so that `list(1,2,3)` returns the list `(1 2 3)`. What is the result of the following expression?

```
let x = 3
    y = 4
in list(list(x, y),
        let x = y
            y = x
        in list(x, y),
        list(x, y))
```

Answer: The list ((3 4) (4 3) (3 4)).

3. (10 points) This is a question about the implementation of `let` expressions. Suppose we changed the interpreter to have a global variable, `the-env`, which was initialized to the value of `(init-env)`. And suppose that `the-env` was not passed to `eval-expression` as an argument, but that each `let` expression extended `the-env` as a side effect. That is suppose the interpreter was written as follows:

```
(deftype the-env environment)
(define the-env (init-env))

(deftype eval-expression (-> (expression) Expressed-Value))
(define eval-expression
  (lambda (exp)
    (cases expression exp
      (lit-exp (datum) (number->expressed datum))
      (var-exp (id) (apply-env the-env id))
      (primapp-exp (prim rands)
        (let ((args (eval-rands rands)))
          (apply-primitive prim args)))
      ;; ...
      (let-exp (ids rands body)
        (let ((args (eval-rands rands)))
          (begin
            (set! the-env (extend-env ids args the-env))
            (eval-expression body)))))))

(deftype eval-rands (-> ((list-of expression)) (list-of Expressed-Value)))
(define eval-rands
  (lambda (exps)
    (if (null? exps)
        '()
        (let ((first-ans (eval-expression (car exps))))
          (cons first-ans (eval-rands (cdr exps)))))))
```

(a) What would the defined language code in problem 2 (p. 3) return with this interpreter?
Answer: The list ((3 4) (4 3) (4 3)).

(b) Briefly explain why this interpreter's implementation of let expressions does not correctly implement the usual semantics for let expressions. (That is, what's wrong with its behavior?)

Answer: The problem is that it doesn't restore the environment when it's finished evaluating the body of a let expression. That happens implicitly when the environment is passed in the interpreter, because the extension only applies to the body of the let, and is "forgotten" automatically by the rest of the evaluation, which never sees it. A fix to this implementation is to use a stack of environments, and to push the new environment on the stack and pop it off when done evaluating the body, restoring the old environment with another `set!`.

4. (10 points) Consider an interpreter for the defined language extended with lists. For this interpreter your task is to add a new built-in primitive, `second`. Its semantics is that `second(`*le*`)` evaluates *le*, which should be a list, and returns the second element of that list. (You can assume that the value of *le* is a list with at least two elements; that is, you don't have to write code to check that.) For example, once this is done, in the defined language we would have the following examples:

```
--> second(list(1,2,3))
2
--> +(second(list(3,4,2,5)), second(list(80,90)))
94
```

Your code should type check, but you don't have to check for the proper number of arguments to the primitive in the defined language. Hint: look at the operations of the standard ADTs on page 2.

Please fill in your answer in the appropriate places below. We have already completed the concrete syntax input for SLLGEN.

Answer:

```
(define the-grammar
  '((program (expression) a-program)
    ;; ... assume the other parts of the grammar are done
    (primitive ("second") second-prim)))

(define-datatype primitive primitive?
   ;; ... assume the other primitives are done and add yours below ...
  (second-prim)
  )

(deftype apply-primitive
  (-> (primitive (list-of Expressed-Value)) Expressed-Value))
(define apply-primitive
  (lambda (prim args)
    (cases primitive prim
       ;; ... assume the other primitive cases are done,
       ;; and add yours below...
      (second-prim () (cadr (expressed->list (car args))))
```

5. This is a question about scoping. Consider the following code in the defined language.

```
let chief = 0
    maxwell = 86
in let secret = proc() maxwell
   in let agent = proc(chief, maxwell) list(chief, maxwell, (secret))
      in (agent 5 10)
```

   (a) (5 points) What kind of scoping is the interpreter using if this code returns `(5 10 86)`?
   Answer: static scoping

   (b) (5 points) What kind of scoping is the interpreter using if this code returns `(5 10 10)`?
   Answer: dynamic scoping

6. This is a problem about parameter passing mechanisms. Consider the following code, written in the defined language with static scoping, assignment, and lists.

```
let x = 3
    y = 4
in let capture = proc (n, m) begin
                               set n = sub1(x);
                               set y = -(m,y);
                               list(n, m, x, y)
                             end
   in let ans = (capture x +(y, 6))
      in cons(x, cons(y, ans))
```

   (a) (5 points) What is the result of the above program if call-by-value is used as the parameter passing mechanism?
   Answer: The list (3 6 2 10 3 6).

   (b) (10 points) What is the result of the above program if call-by-reference is used as the parameter passing mechanism?
   Answer: The list (2 6 2 10 2 6).

7. (20 points) In this problem you will implement the following syntax in the defined language.

   ⟨expression⟩ ::= `implies` ⟨expression⟩ `==>` ⟨expression⟩

   Use the following as the abstract syntax for the `implies`-expression.

```
(define-datatype expression expression?
  ;; ...
  (implies-exp (left-exp expression?) (right-exp expression?)))
```

   The meaning of this syntax is supposed to be that of a short-circuit logical implication operator. The following are examples in the defined language:

```
--> implies 1 ==> 0
0
--> implies 0 ==> 1
1
--> implies -(2,1) ==> 1
1
--> implies +(0,0) ==> -(2,2)
1
--> letrec forever() = (forever) in implies 0 ==> (forever)
1
```

   That is, `implies` e_1 e_2 is equivalent to `if` e_1 `then` e_2 `else 1`. However, you are *not* to implement this as a syntactic sugar. That is, do *not* use `if-exp`, `run`, or `scan&parse` in your solution. Instead you will implement this in `eval-expression` directly, by filling in the code for the `implies-exp` case of `eval-expression` below.

   To save time, only give the code for the `implies-exp` case, and any auxiliary procedures that you call in that case.

   Answer:

```
(deftype eval-expression (-> (expression environment) Expressed-Value))
(define eval-expression
  (lambda (exp env)
    (cases expression exp
      (lit-exp (datum)
        (number->expressed datum))
      (var-exp (id)
        (apply-env env id))
      ;; ... assume that the rest of this is done
      (implies-exp (left-exp right-exp)
        (if (true-value? (eval-expression left-exp env))
            (eval-expression right-exp env)
            (number->expressed 1)))
      )))
```

8. (25 points) In this problem you will implement a new kind of procedure expression in the defined language. This new kind of procedure expression has the following syntax:

   ⟨expression⟩ ::= `varargsproc ( ⟨identifier⟩ )` ⟨expression⟩

   Its semantics is that its evaluation forms a new kind of procedure closure that remembers the current environment, e. When called, this new closure extends the remembered environment, e, by binding the list of its actual parameters to its formal parameter, and then evaluating the expression in its body in that extended environment. Assume that the interpreter has already been extended to support lists. So the following are examples.

```
--> let mklist = varargsproc (x) x
    in (mklist 3 4 2 (mklist (mklist (mklist 5) 4) 1))
(3 4 2 (((5) 4) 1))
--> let g = varargsproc (x) cons(add1(car(x)), cdr(x))
        h = varargsproc (x) car(x)
    in list((g 1 10), (g 3 20 30), (g 40), (g (h 3)), (g (h 5 10)), (h (g 1)))
((2 10) (4 20 30) (41) (4) (6) (2))
--> let f = varargsproc (x) varargsproc (y) x
    in list(((f)), ((f ((f 3) 4) 5 6)), ((f 9 10 11 12 13 14 15 16) 17 18))
(() ((3) 5 6) (9 10 11 12 13 14 15 16))
```

   Your task is to implement the `varargsproc` expression, by writing code in the appropriate places below. On this page write your code for the ProcVal ADT. On the following page write your code to define the abstract syntax and to do evaluation of the new expressions.

   Answer:

```
(define-datatype procval procval?
  (closure
    (ids (list-of symbol?))
    (body expression?)
    (env environment?))
  (varargs-closure
    (id symbol?)
    (body expression?)
    (env environment?)))

(deftype apply-procval
  (-> (procval (list-of Expressed-Value)) Expressed-Value))
(define apply-procval
  (lambda (proc args)
    (cases procval proc
      (closure (ids body env)
        (eval-expression body (extend-env ids args env)))
      (varargs-closure (id body env)
        (eval-expression body
                         (extend-env (list id)
                                     (list (list->expressed args))
                                     env))))))
```

Please fill in your code below for the other changes to the interpreter for the problem on the previous page. We have already completed the concrete syntax input for SLLGEN.

Answer:

```
(define the-grammar
  '((program (expression) a-program)
    ;; ... assume the other parts of the grammar are done
    (expression
      ("varargsproc" "(" identifier ")" expression)
      varargsproc-exp)))

(define-datatype expression expression?
  ;; ... assume the other parts are done
  (varargsproc-exp
  (varargsproc-exp
   (id symbol?)
   (body expression?))
  )

(deftype eval-expression (-> (expression environment) Expressed-Value))
(define eval-expression
  (lambda (exp env)
    (cases expression exp
      (var-exp (id)
        (apply-env env id))
      (app-exp (rator rands)
        (let ((proc (eval-expression rator env))
              (args (eval-rands rands env)))
          (if (procval? proc)
            (apply-procval (expressed->procval proc) args)
            (eopl:error 'eval-expression
              "Attempt to apply non-procedure ~s" proc))))
      ;; ... assume the other parts of this are done
      (varargsproc-exp (id body)
          (procval->expressed (varargs-closure id body env)))
      )))
```