

Exam 3a CIS 352: Programming Languages

Name: _____

Question	1	2	3	4	5	6	7	Total
Points Received								
Points Possible	20	16	20	9	8	15	12	100

Instructions — *Read them all!*

- (i) The quiz is closed book, closed notes, and closed friends.
- (ii) *Legibility counts!* Make sure that we can read *and find* your answers.
Use the backs of pages if you need more space for an answer.
- (iii) Make sure that your test paper has 7 pages total.
- (iv) Unless otherwise stated, you may use any built-in Scheme function you want—so long as it is spelled and used correctly.
- (v) In programming questions, define any helper functions you want.

Question 1 (20 points). Answer 4 out of the following 8 questions. *Only the first 4 answers I find will be graded!*

- (a) Briefly explain what a *thunk* is and how thunks are used in implementing programming languages.
- (b) Give a brief definition and an example of *variable aliasing*.
- (c) Briefly explain the difference between how `let` and `letrec` are implemented. *Hint: What goes wrong if we try to evaluate the following?*
- ```
let fact = proc(n) if zero?(n) then 1
 else *(n,(fact (sub1(n))))
in (fact 6)
```
- (d) Explain the difference between *expressed values* and *denoted values* and sketch a situation where they differ.
- (e) What are the *lexical addresses* of the underlined occurrences of `w` and `x` in the following expression?
- ```
(lambda (x y z)
  ((x y) (lambda (z w) (lambda (y) (x w))))))
= =
```
- (f) Briefly explain the distinction between *direct targets* and *indirect targets*, in the context of call-by-reference parameter passing.
- (g) Briefly explain the difference between *abstract syntax* and *concrete syntax* and the rôle *abstract syntax* in programming language implementation.
- (h) In the continuation-passing interpreter, *eval-expression* takes three arguments: an expression, an environment, and a continuation. Briefly explain the purpose of the *environment* and the *continuation*.

Question 2 (16 points). This question assumes that *call-by-value* parameter passing is used.

(a) (8 points) Consider the following code.

```
let x = 2  y = 4  z = 6
  in let f = proc(x) +(x,y)
      g = proc(t,y) *(t,y)
      y = 8
      in
        (f (g z 10))
```

(i) What is the value of this expression under *lexical scoping*?

(ii) What is the value of this expression under *dynamic scoping*?

(b) (8 points) Consider the following code.

```
let m = 0  q = 1  r = 10
  in let m = 4
      p = proc(x,t) begin
          set r = +(m,200);
          *(r,t)
        end
      in let q = 5
          k = proc(a,m) begin
              set m = 2;
              (p add1(m) *(a,q))
            end
          in
            (k m r)
```

(i) What is the value of this expression under *lexical scoping*?

(ii) What is the value of this expression under *dynamic scoping*?

Question 3 (20 points). Consider the following code, assuming *lexical scoping*.

```

let b = 7
    c = 2
    d = 0
in
  let f = proc(k) begin set d = +(d,1); *(k,10) end
      g = proc(j) begin set d = +(d,100); *(j,10) end
      h = proc(x,y,z,w) begin
          set y = x;
          set c = +(z,1);
          set y = *(x,x);
          set b = add1(b);
          write(y,z)
        end
  in begin
      (h (f 1) b c (g 2));
      write(b,c,d)
    end

```

Fill in the table below with the values printed under each of the listed parameter passing schemes.

	write(y,z) from h	write(b,c,d) from the main begin-end
(a) call-by-value		
(b) call-by-reference		
(c) call-by-name		
(d) call-by-need		
(e) call-by-value-result		

Question 4 (9 points). Suppose the definitions and statements below are executed in order.

```
> (define fox '((a) b c) )
> (define sox (cons (cdr fox) (car fox)) )
> (set-cdr! (car fox) '(10) )
> (define knox (cons (cdr (car sox)) (car sox)) )
> (set-car! (cdr (car sox)) '(y z) )
```

What are the final values of:

fox _____

sox _____

knox _____

Question 5 (8 points). Suppose we have the following definitions in the version of our defined language that includes exceptions.

```
define f = proc (n) if zero?(sub1(n)) then 50 else raise n ;
define h1 = proc (y) +(y,100) ;
define h2 = proc (z) +(z,500) ;
define h3 = proc (g) proc (w) +((g w), w) ;
```

Give the values for each of the following expressions:

(a) `try *(10, raise 7) handle h1`

(b) `+(12,try (f 7) handle proc (w) *(w,1000))`

(c) `try +(try (f 1) handle h1, 1000) handle h2`

(d) `try ((h3 f) 99) handle h2`

Question 6 (15 points). Consider adding to our call-by-value interpreter a *swap-two-variables* expression that has the following concrete and abstract syntax.

$$\langle \text{exp} \rangle ::= \langle \text{identifier} \rangle ::= \langle \text{identifier} \rangle$$

$$\text{swap-exp } (\text{id1 id2})$$

The intended semantics of an expression such as $x ::= y$ is that the variables x and y have their values swapped; *furthermore*, the expression should return the original value of y . For example, the following should print out: 20 10 21 40:

```
let a = 10  b = 20  c = 30  d = 40
  in let f = proc(x) +(a,x)
      in begin
          a ::= b
          write(a,b,(f 1), c ::= d)
        end
```

A relevant portion of `eval-expression` from the EOPL §3.7 interpreter is given on the right. Provide the necessary code to implement the `swap` construct.

```
(define eval-expression
  (lambda (exp env)
    (cases expression exp
      (lit-exp (datum) datum)
      (var-exp (id) (apply-env env id))
      (varassign-exp (id rhs-exp)
        (begin
          (setref!
            (apply-env-ref env id)
            (eval-expression rhs-exp env))
          1))
      (primapp-exp (prim rands)
        (let ((args (eval-rands rands env)))
          (apply-primitive prim args)))
      (proc-exp (ids body) (closure ids body env))
      (app-exp (rator rands)
        (let ((proc (eval-expression rator env))
              (args (eval-rands rands env)))
          (if (procval? proc)
              (apply-procval proc args)
              (eopl:error 'eval-expression
                "Attempt to apply non-procedure ~s"
                proc))))
      (swap-exp (id1 id2)
        ... YOUR CODE WILL GO HERE ...
        ) )))
```

Question 7 (12 points). For this question we take our continuation-passing interpreter and add *while-expressions* which have the following concrete and abstract syntax.

```
⟨exp⟩ ::= while ⟨exp⟩ do ⟨exp⟩
        while-exp (test body)
```

The intended meaning of “while *test* do *body*” is to:

1. Evaluate *test*. (Call the resulting value *t*.)
2. If $t \neq 0$, then first evaluate *body* and second go back to step 1.
3. If $t = 0$, then (the loop terminates) and the while-expression returns the value 1.

For example, the following should print out:

```
2 4 8 16:
```

```
let x = 1 k = 4
  in while notZero?(k) do begin
    set x = *(x,2);
    set k = sub1(k);
    write(x)
  end
```

A relevant portion of `eval-expression` from the continuation-passing interpreter is given on the right. Write the necessary code to implement the while construct.

```
(define eval-expression
  (lambda (exp env cont)
    (cases expression exp
      (lit-exp (datum) (cont datum))
      (var-exp (id) (cont (apply-env env id)))
      (proc-exp (ids body)
        (cont (closure ids body env)))
      (if-exp (test-exp true-exp false-exp)
        (eval-expression test-exp env
          (lambda (val)
            (if (true-value? val)
                (eval-expression true-exp env cont)
                (eval-expression false-exp env cont))))))
      (primapp-exp (prim rands)
        (eval-rands rands env
          (lambda (args)
            (cont (apply-primitive prim args))))))
      (app-exp (rator rands)
        (eval-expression rator env
          (lambda (proc)
            (eval-rands rands env
              (lambda (args)
                (if (procval? proc)
                    (apply-procval proc args cont)
                    (eopl:error 'eval-expression
                      "Attempt to apply non-procedure ~s"
                      proc)))))))
      (while-exp (test body)
        ... YOUR CODE WILL GO HERE ...
        ) )))
```