

Distribution of Scores

26-30:	26	
31-35:	31 31 32 33 35	
36-40:	37 37 38	Average: 38.41
41-45:	41 43	Median: 37
46-50:	46 48 50 50 50	

Problem 1: Short Answers (20 points)

Pick **four** of the following questions to answer. If you do more than four, all will be graded, *but* only the *lowest* four scores will count. (*Hint: Keep your examples simple.*)

- (a) Give an example program that produces different answers under (i) *call-by-value* and (ii) *call-by-reference* parameter passing.

An answer: Consider

```
let x=0
  in let p = proc(z) set z=10
      in begin (p x); x end
```

This returns 0 under call-by-value and returns 10 under call-by-reference.

- (b) Give an example program that produces different answers under (i) *call-by-value* and (ii) *call-by-name* parameter passing.

An answer: The example of (a) works for this part too, with call-by-name returning 10.

- (c) Give an example program that produces different answers under (i) *lexical* and (ii) *dynamic* scoping.

An answer: Consider

```
let y = 0
  in let p = proc(x) y
      in let y = 10
          in (p 0)
```

This returns 0 under lexical scoping and returns 10 under dynamic scoping.

- (d) What is the difference between *concrete syntax* and *abstract syntax*?

An answer: Concrete syntax is what the user types for the program and abstract syntax is the interpreter/compiler's data structure representation of the program's syntax.

- (e) What is the job of the *scheduler* in a multi-threaded programming system?

An answer: The scheduler keeps track of the currently executing thread and how much time it has currently used up and the threads waiting to run. When the currently executing thread either quits or runs out of its current time slice, then if the current thread has more work to do, it is put at the end of the ready queue and the front of the ready queue become the currently executing thread.

- (f) Define *tail-call* and give an example of one.

An answer: A tail call is a procedure call which is followed by a return to the calling code (see http://en.wikipedia.org/wiki/Tail_call). The calls of `value-of/k` and `apply-cont` in the code given for problem 3 are tail-calls.

- Problem 2 (15 points)** Suppose that the following definitions are made in our defined language:

```
define f = proc (n) if zero?(n) then raise 50 else +(n,1)
define h1 = proc (y) +(y,200)
define h2 = proc (z) +(z,300)
define h3 = proc (g,w) +((g +(w,-1)), w)
```

Give the values for each of the following expressions:

- (a) `try (f 10) catch proc (w) +(w,100)`

Answer:

```
try +(f 10) catch proc (w) +(w,100)
  ~> try +(10,1) catch proc (w) +(w,100) ~> 11
```

- (b) `try +(3, raise 20) catch h1`

Answer:

```
try +(3, raise 20) catch h1
  ~> (h1 20) ~> +(20,200) ~> 220
```

- (c) `try +(try (f 0) catch h1, 1000) catch h2`

Answer:

```
try +(try (f 0) catch h1, 1000) catch h2
  ~> try +( (h1 50) , 1000) catch h2
  ~> try +( +(50,200) catch h1, 1000) catch h2
  ~> try +(250, 1000) catch h2 ~> 1250
```

(d) `try (h3 f 1) catch h2`

Answer: `350`
`try (h3 f 1) catch h2 ~> try +((f +(1,-1)),1) catch h2`
`~> try +((f 0),1) catch h2 ~> (h2 50) ~> +(50,300)`
`~> 350`

(e) `try +(4,try +(3,raise 0) catch f) catch h1`

Answer: `250`
`try +(4,try +(3,raise 0) catch f) catch h1`
`~> try +(4,(f 0)) catch h1 ~> (h1 50) ~> +(50,200)`
`~> 250`

Problem 3 (15 points) An and-expression takes two arguments e_1 and e_2 , and:

Step 1. First evaluates e_1 .

Step 2. Then this value is tested.

Step 2a. If this value is `#f`, then the and-expression returns `#f` without evaluating e_2 .

Step 2b. Otherwise, the and-expression returns the result of evaluating e_2 .

For example, `and(zero?(0),zero?(12))` should return `#f` and `and(zero?(0),zero?(0))` should return `#t`. Below are portions of the `value-of/k` and `apply-cont` procedures from our continuations-base interpreter with the code to handle and-expressions partially filled in. Your job is to supply the remaining parts.

```
(define value-of/k
  (lambda (exp env cont)
    (cases expression exp
      (const-exp (num)
        (apply-cont cont (num-val num)))
      (var-exp (var)
        (apply-cont cont (apply-env env var)))
      ...
```

```
(if-exp (exp1 exp2 exp3)
  (value-of/k exp1 env
    (if-test-cont exp2 exp3 env cont)))
(and-exp (exp1 exp2)
   )
... )))
```

```
(define apply-cont
  (lambda (cont val)
    (cases continuation cont
      (end-cont () val)
      (if-test-cont (exp2 exp3 env cont)
        (if (expval->bool val)
          (value-of/k exp2 env cont)
          (value-of/k exp3 env cont)))
      ...
      (and-test-cont (exp2 saved-env saved-cont)
        (if (expval->bool val)
          
           ) )
      ... )))
```

(a) Your code for Step 1. (*Hint: You should create an and-test-cont continuation as part of this.*)

Answer: `(value-of/k exp1 env (and-test-cont exp2 env cont))`

(b) Your code for Step 2b.

Answer: `(value-of/k exp2 saved-env saved-cont)`

(c) Your code for Step 2a.

Answer: `(apply-cont saved-cont (bool-val #f))`