Koç University
College of Engineering

COMP 301: Programming Language Concepts

Midterm Examination
Nov 19, 2016, Saturday 18:30-20:30
SOS-B07 & SOS-B10

Instructor: Deniz Yuret
Time Allowed: 120 minutes

Name:_____

Student Number:_____

NOTE: Explain your answers in full. Provide all the work in your exam paper, and make sure the final answer has been clearly marked for each question.

I pledge on my honor that I have neither given nor received unauthorized assistance on this exam.

Signature:_____

Question	Worth	Grade
1	10	
2	20	
3	10	
4	20	
5	10	
6	30	
Total	100	

1. (10 points) Implement a Scheme procedure, (swapper s1 s2 slist), which returns a list the same as slist, but with all occurrences of s1 replaced by s2 and all occurrences of s2 replaced by s1.

```
> (swapper 'a 'd '(a b c d))
(d b c a)
> (swapper 'a 'd '(a d () c d))
(d a () c a)
> (swapper 'x 'y '((x) y (z (x))))
((y) x (z (y)))
```

```
;;; ANSWER-1

(define (swapper x y lst)
  (cond ((eq? lst x) y)
        ((eq? lst y) x)
        ((pair? lst) (cons (swapper x y (car lst))
                           (swapper x y (cdr lst))))
        (else lst)))
```

2. (20 points) Consider the following binary tree datatype:

```
bintree ::= int | (symbol bintree bintree)
```

(a) (10 points) Implement the bintree datatype using `define-datatype`.

```
;;; ANSWER-2a

(define-datatype bintree bintree?
  (leaf (num integer?))
  (node (name symbol?)
        (left bintree?)
        (right bintree?)))
```

(b) (10 points) Implement a procedure `double` using `cases` that takes a bintree and returns another bintree where all the numbers have been doubled.

```
;;; ANSWER-2b

(define (double bt)
  (cases bintree bt
    (leaf (num) (leaf (* 2 num)))
    (node (name left right)
      (node name (double left) (double right)))))
```

3. (10 points) Draw the abstract syntax tree for the following lambda calculus expression:

```
(lambda (x)
  (lambda (y)
    ((lambda (x) (x y))
     x)))
```

You can assume the following grammar for lambda expressions:

```
(define-datatype lc-exp lc-exp?
  (var-exp (var identifier?))
  (lambda-exp (param identifier?) (body lc-exp?))
  (app-exp (rator lc-exp?) (rand lc-exp?)))
```

```
 ANSWER-3: the tree drawing of:

 (lambda-exp
   (param x)
   (body
     (lambda-exp
       (param y)
       (body
         (app-exp
           (rator
             (lambda-exp
               (param x)
               (body
                 (app-exp
                   (rator x)
                   (rand y)))))
           (rand x))))))
```

4. (20 points) Consider the following program:

```
let dbl = proc (x) x
    in let dbl = proc (x)
                   if zero?(x)
                   then 0
                   else -((dbl -(x,1)), -2)
       in (dbl 6)
```

(a) (10 points) What is the value of this program if we use lexical scoping? Please explain your answer.

```
ANSWER-4a
Let us call the first dbl dbl1 and second dbl2.
(dbl 6) calls dbl2 with x=6.
The environment of dbl2 is the one created by the first let where dbl=dbl1.
Therefore its body is evaluated within that environment.
-((dbl -(x,1)), -2) calls dbl1 with -(x,1)=5.
(dbl1 5) returns 5 and the final result is 7.
```

(b) (10 points) What is the value of this program if we use a dynamic scoping? Please explain your answer.

```
ANSWER-4b
Let us call the first dbl dbl1 and second dbl2.
(dbl 6) calls dbl2 with x=6.
The body of dbl2 is evaluated in the caller's environment where dbl=dbl2.
-((dbl -(x,1)), -2) calls dbl2 with -(x,1)=5.
In dynamic scoping the latest value of dbl=dbl2 is accessed.
The procedure runs recursively as intended and returns 12.
```

5. (10 points) Convert the following code to a nameless version by replacing `let` and `proc` by their nameless versions and replacing variable references with their lexical addresses:

```
let a = 3
in let p = proc (x) -(x,a)
   in let a = 5
      in -(a,(p 2))
```

```
;;; ANSWER-5

nlet 3
in nlet nproc -(#0,#1)
   in nlet 5
      in -(#0,(#1 2))
```

6. (30 points) Extend the lexical address translator and interpreter to handle the `cond` expression. Use the grammar

```
Expression ::= cond { Expression ==> Expression }* end
```

In this expression, the expressions on the left-hand sides of the `==>`'s are evaluated in order until one of them returns a true value. Then the value of the entire expression is the value of the corresponding right-hand expression. If none of the tests succeeds, the expression should report an error. Complete the implementation below.

(a) (10 points) What clause do we add to `define-datatype` for `cond-exp`?

```
(define-datatype expression expression?
  (const-exp (num number?))
  (diff-exp (exp1 expression?) (exp2 expression?))

  ...

  (cond-exp  ;; YOUR CODE HERE


 ;;; ANSWER-6a

    (conditions (list-of expression?))
    (actions    (list-of expression?))
```

(b) (10 points) How do we translate a `cond-exp` to a nameless expression?

```
(define (translation-of expr senv)
  (cases expression expr
    (const-exp (num) (const-exp num))
    (diff-exp (exp1 exp2)
      (diff-exp
        (translation-of exp1 senv)
        (translation-of exp2 senv)))

     ...

    (cond-exp ;; YOUR CODE HERE

;;; ANSWER-6b

      (conditions actions)
      (cond-exp
        (map (lambda (e) (translation-of e senv)) conditions)
        (map (lambda (e) (translation-of e senv)) actions))
```

(c) (10 points) How do we evaluate the resulting nameless expression? Feel free to use helper procedures.

```
(define (value-of expr nenv)
  (cases expression expr
    (const-exp (num) (num-val num))
    (diff-exp (exp1 exp2)
      (let ((exp1-val (value-of exp1 nenv))
            (exp2-val (value-of exp2 nenv)))
        (let ((exp1-num (expval->num exp1-val))
              (exp2-num (expval->num exp2-val)))
          (num-val
            (- exp1-num exp2-num)))))

     ...

    (cond-exp  ;; YOUR CODE HERE

 ;;; ANSWER-6c

    (cond-exp
      (conditions actions)
      (eval-cond conditions actions nenv)
    )))

(define (eval-cond conditions actions nenv)
  (cond ((null? conditions)
         (bool-val #f))
        ((expval->bool (value-of (car conditions) nenv))
         (value-of (car actions) nenv))
        (else
         (eval-cond (cdr conditions) (cdr actions) nenv))))
```