CS 603 Final Exam **Spring 2006** **Solution**

1. Determine the output produced by each call to function f in this Scheme code.

```
(define f (L x) (if (null? L) x ((car L) (f (cdr L) x))))
(define inc (x) (+ x 1))
(define double (x) (+ x x))
(define square (x) (* x x))
(define dec (x) (- x 1))
(f (list inc double square dec) 6)    51
(f (list inc double square dec) 7)    73
(f (list inc double square dec) 8)    99
```

2. Determine the output produced by each call to function g in this ML code.

```
fun g [ ] x = x  |  g (h::t) x = g t (h x);
fun inc x = x+1;
fun double x = x+x;
fun square x = x*x;
fun dec x = x-1;
g [inc,double,square,dec]  4;    99
g [inc,double,square,dec]  5;   143
g [inc,double,square,dec]  6;   195
```

3. Write a Scheme function (applyall L x y) that takes a list L of binary functions, and values x and y. It should return the list of values obtained by applying each function in list L to arguments x and y. Example: (applyall (list + - *) 7 4) should return (11 3 28), because 7+4=11 and 7-4=3 and 7*4=28.

```
(define applyall (L x y)
    (if (null? L) '( )
          (cons ((car L) x y) (applyall (cdr L) x y))))
```

4. Write an ML function applyall(L,x,y) that takes a list L of binary functions, and values x and y. It should return the list of values obtained by applying each function in list L to arguments x and y. Use pattern matching; do not call the predefined hd or tl functions. Example: applyall([op+,op-,op*], 7, 4) should return [11,3,28], because 7+4=11 and 7-4=3 and 7*4=28.

```
fun applyall([ ],_,_) = [ ]
  |   applyall(h::t,x,y) = h(x,y)::applyall(t,x,y)
```

5. The complement of a binary string is obtained by changing all 0's to 1's and all 1's to 0's. For example, the complement of 01101001 is 10010110. Write a grammar that generates the set of binary strings such that the complement of the string equals the reverse of the string. Also, draw parse trees for the strings 01101001 and 10010110.

   $S \rightarrow 0S1 \mid 1S0 \mid \varepsilon$ **[Parse trees are not shown.]**

6. Complete the given μSmalltalk definition of the class Pair, so that the class Quadruple can be defined exactly as shown below, and so that the client code given on the right would then produce the output as shown in bold.

```
(class Pair Object (x y)
    (method x ( ) x)
    (method y ( ) y)
    ; provide additional methods here

    (classMethod new:: (a b)
        (init:: (new Pair) a b))

    (method init:: (a b)
        (set x a) (set y b) self)

    (method + (p)
        (new:: Pair (+ x (x p))
                    (+ y (y p))))

    (method println ( )
        (println x) (println y) self)

    (method = (p)
        (& (= x (x p)) (= y (y p))))




)

(class Quadruple Pair ( )
    (classMethod new:::: (a b c d)
        (new:: Pair (new:: Pair a b)
                    (new:: Pair c d)))
)
```

```
(val A (new:::: Quadruple 1 2 3 4))
(val B (new:::: Quadruple 1 2 3 5))
(val C (+ A B))
(val D (+ B A))
(println A)
    1
    2
    3
    4
(println B)
    1
    2
    3
    5
(println C)
    2
    4
    6
    9
(println D)
    2
    4
    6
    9
(= A B)
    <False>
(= B C)
    <False>
(= C D)
    <True>
```

7. In Smalltalk, the receiver of a message is known as self. Four possible implementations of self are described below. For each implementation, determine whether or not it works correctly, and briefly justify each answer.

   a. self is an implicit formal parameter that is used to automatically pass the address of the receiver to the function that implements the method.

      Yes, C++ and Java use this approach.

   b. self is an implicit global variable, and each time a method is invoked the global variable is automatically changed to point to the current receiver.

      No, because the global is not changed back upon returning from a method.

   c. self is an implicit instance variable of each object, and the constructor automatically initializes this field to point to the object when it is created.

      Yes, Smalltalk uses this approach.

   d. self is determined at run-time by mapping the address of the currently executing instruction to the instance of the class that contains this instruction.

      No, because the address would map to the class, not the instance of the class. Also, cannot even determine the correct class if the method was inherited.

8. Write a Prolog predicate inner(X,Y,Z) that is satisfied when Z is the inner product of lists X and Y with respect to binary operations + and *. This kind of inner product is often called a dot product. You may use helper predicates if you wish, and you may assume that lists X and Y have the same length. Example: inner([1,2,3], [4,5,6], 32), because 1*4+2*5+3*6 = 32.

```
inner([ ],[ ],0).
inner([A|B],[C|D],Z) :- inner(B,D,Y), Z is A*C + Y.
```

9. Write a Prolog predicate outer(X,Y,Z) that is satisfied when Z is the outer product of lists X and Y with respect to binary operation *. Note that Z is a list of length |X|, each of whose members is a sublist of length |Y|. The k^{th} element of the j^{th} sublist is obtained by multiplying the j^{th} element of X with the k^{th} element of Y. This kind of outer product is often called a multiplication table. You may use helper predicates if you wish. Example: outer([1,2,3], [4,5,6,7], [[4,5,6,7],[8,10,12,14],[12,15,18,21]]).

```
helper(_,[ ],[ ]).
helper(Q,[A|B],[X|Y]) :- X is Q*A, helper(Q,B,Y).

outer([ ],_,[ ]).
outer([A|B],L,[X|Y]) :- helper(A,L,X), outer(B,L,Y).
```

10. List five (or more) most significant ways that Prolog is not a pure logic programming language, and/or that Prolog does not provide the full power of logic programming.

    Order of rules within the database matters, so not purely logical OR.
    Order of clauses on right side of each rule matters, so not purely logical AND.
    The "not" predicate is different from purely logical NOT.
    Assignment, relational, and arithmetic operators force an order of evaluation.
    The cut ("!") is not a logical function.
    Only Horn clauses are permitted.
    No existential quantification; universal quantification is assumed for each rule.
    The "assert", "retract", and "print" predicates have side effects.

11. Suppose a Prolog database currently contains only facts of the form parent(x, y), which means that x is a parent of y. Write Prolog rules for each of the following:

    a. child(A, B), which means that A is a child of B.

       child(A, B) :- parent(B, A).

    b. sibling(A, B), which means that A is a brother or sister of B.

       sibling(A, B) :- not(A=B), parent(C, A), parent(C, B).

    c. grandchild(A, B), which means that A is a grandchild of B.

       grandchild(A, B) :- child(A, C), child(C, B).

    d. cousin(A, B), which means that A is a (first) cousin of B.

       cousin(A, B) :- parent(C, A), parent(D, B), sibling(C, D).

    e. descendant(A, B), which means that A is a descendant of B.

       descendant(A, B) :- child(A, B).
       descendant(A, B) :- child(A, C), descendant(C, B).

12. Suppose this definition of predicate p is found in Prolog's database:

    p(X, X).

    Determine and explain the output that would appear upon entering this query:

    ?- p(L, [a|L]).

    a. First assume that the "occurs check" <u>is</u> implemented in this version of Prolog.

       The answer is "no" because L cannot unify with [a|L].

    b. Next assume that the "occurs check" <u>is not</u> implemented in this version of Prolog.

       The answer is "yes" but the list L is infinite: L = [a, a, a, a, a, …].

13. Rewrite each logical expression as an equivalent completely simplified expression.

    a. $\neg\ (\exists x)\ (\forall y)\ (P(x) \leftrightarrow Q(y))$

       $(\forall x)\ (\exists y)\ (P(x) \oplus Q(y))$

    b. $p \wedge q \wedge (p \rightarrow \neg r) \wedge (s \rightarrow \neg q) \wedge (r \vee \neg t) \wedge (s \vee t)$

       false

    c. $p \vee q \vee ((p \rightarrow \neg r) \wedge (s \rightarrow \neg q))$

       true

14. Determine the result of applying resolution to these two rules. Also state the substitution that must be applied so that unification can succeed.

    $A(w, x) \leftarrow B(f(w), x)\ ,\ C(w, g(x))$.
    $C(h(y), z)\ ,\ D(y, k(z)) \leftarrow E(y, z)$.

    Substitution: w := h(y) and z := g(x).
    Result: $A(h(y), x)\ ,\ D(y, k(g(x))) \leftarrow B(f(h(y)), x)\ ,\ E(y, g(x))$.

15. Complete this natural semantics rule for Scheme's primitive cons function.

$$\frac{\langle e, \rho, \sigma_0\rangle \Downarrow \langle \text{PRIMITIVE}(\text{cons}), \sigma_1\rangle \quad \langle \mathbf{e_1, \rho, \sigma_1}\rangle \Downarrow \langle \mathbf{v_1, \sigma_2}\rangle \quad \langle \mathbf{e_2, \rho, \sigma_2}\rangle \Downarrow \langle \mathbf{v_2, \sigma_3}\rangle \quad \ell_1 \notin \text{dom } \sigma_3 \quad \ell_2 \notin \text{dom } \sigma_3}{\langle \text{APPLY}(e, e_1, e_2), \rho, \sigma_0\rangle \Downarrow \langle \mathbf{CONS}(\ell_1, \ell_2), \sigma_3\{\ell_1 \rightarrow \mathbf{v_1}, \ell_2 \rightarrow \mathbf{v_2}\}\rangle} \text{ (CONS)}$$

16. Complete this axiomatic semantics rule for a C-like DO_WHILE statement in Impcore. Expression S is the body of the loop, and expression e is the condition.

If **P { S } Q**

and **Q ∧ e → P**

then **P** { (do_while S e) } **Q ∧ ¬ e**

17. Write a denotational semantics rule for a C-like DO_WHILE statement in Impcore. Command c is the body of the loop, expression e is the condition, and s is the store.

execute [(do_while c e)] s =

execute [(while e c)] (execute [c] s)

18. Write a translational semantics rule for a C-like DO_WHILE statement in Impcore. Command c is the body of the loop, and expression e is the condition.

execute [(do_while c e)] =

L: execute [c]
evaluate [e]
JUMPT L