UNIVERSITY OF TORONTO AT MISSISSAUGA
April 2006 Examination
CSC 324H5 S
Instructor: Richard Krueger
Duration — 3 hours
No Aids Allowed

Student Number: _____

Last (Family) Name(s): SOLUTIONS

First (Given) Name(s): SAMPLE

*It is an academic offence for students to possess the following items at their examination desks: cell phones, pagers of any kind, IPODs, MP3 players, wristwatch computers, personal digital assistants (e.g., palm pilots) or any other device that is electronic. If any of these items are in your possession now, please ensure they are turned off and put them with your belongings at the front (or side) of the room before the examination begins — no penalty will be imposed. A penalty **WILL BE** imposed if any of these items are kept at your examination desk. Please note, students are **NOT** allowed to petition to RE-WRITE a final examination.*

---

*Do **not** turn this page until you have received the signal to start.*
(In the meantime, please fill out the identification section above, and read the instructions below *carefully*.)

This final examination consists of 11 questions on 13 pages (including this one). *When you receive the signal to start, please make sure that your copy of the test is complete and write your student number at the bottom of every page, where indicated.*

Answer each question directly on the test paper, in the space provided, and use the reverse side of the pages for rough work. If you need more space for one of your solutions, use the reverse side of a page and *indicate clearly the part of your work that should be marked.*

Concise, well thought-out answers will receive higher credit than long rambling ones. Unreadable answers will be given zero (0) credit, so write legibly and clearly.

In the programming questions, you do not have to write pre-conditions/post-conditions. You do not need to comment your code, but use meaningful variable names. You may use helper procedures (unless otherwise indicated). Use recursion where possible.

MARKING GUIDE

# 1:	_____/ 8
# 2:	_____/ 10
# 3:	_____/ 10
# 4:	_____/ 10
# 5:	_____/ 10
# 6:	_____/ 10
# 7:	_____/ 14
# 8:	_____/ 10
# 9:	_____/ 10
# 10:	_____/ 10
# 11:	_____/ 8
TOTAL:	_____/110

Good Luck!

Question 1. Prolog Basics [8 MARKS]

Part (a) [2 MARKS]

Write `max/3` in Prolog (without using cuts). It takes three integers as arguments and succeeds if the third argument is the maximum of the first two.

SAMPLE SOLUTION

```
max(X,Y,Y) :- X =< Y.
max(X,Y,X) :- X > Y.
```

Part (b) [2 MARKS]

We can make this more efficient with a cut. Fill in *both* blanks below with the appropriate goals so that `max/3` works correctly when all arguments are instantiated. (Recall the goal `true` always succeeds. Use it instead of leaving a blank empty.)

```
max(X,Y,Y) :- ____X =< Y____, !, ____true____.
max(X,Y,X).
```

Part (c) [2 MARKS]

Queries of the form `?- max(3,2,X).` work fine, but there's a problem lurking. Give an example of a query that doesn't have correct results with the version of `max/3` from part (b).

SAMPLE SOLUTION

```
max(2,3,2).
```

This is matched as a fact whenever the first rule fails.

Part (d) [2 MARKS]

Write the following as Prolog clauses, queries or facts. Introduce any predicates, variables and constants you feel are necessary.

If a tree falls in the forest, it makes a noise if someone is nearby.

Many solutions are acceptable for these two questions.

```
noise-made(T) :- tree-in-forest(T), falls(T), nearby(_,T).
```

Is there no honest lawyer?

```
?- \+ (lawyer(X), honest(X)).
```

Question 2. Prolog Lists [10 MARKS]

Part (a) [5 MARKS]

Lexicographic order is like the order of words in a dictionary. Based on the ordering of letters (A through Z), words are sorted alphabetically. If one word is a prefix of another, the shorter word comes first, so `an` comes before `ant`.

For this question, we'll use lists instead of words, and we'll use integers instead of letters. Write a Prolog predicate `lexless/2` that takes two lists of integers and is true if its first argument is (strictly) lexicographically less than the second argument.

For example, `lexless([1,14], [1,14,20])` should succeed, but `lexless([14,15,20], [7,15,15,4])` should fail.

SAMPLE SOLUTION

```
lexless([],[_|_]).
lexless([X|_],[Y|_]) :- X < Y.
lexless([A|B],[C|D]) :- A = C, lexless(B,D).
```

Part (b) [5 MARKS]

Write a Prolog predicate `map(Pred, List1, List2)`. It takes the name of a binary predicate followed by two lists of equal length, and succeeds if `Pred` is true on all successive pairs of elements from `List1` and `List2`. You may assume that `Pred` is instantiated. (Do not use the built-in `maplist` in your solution.)

For example, `map(square, [1,2,3], [1,4,9])` should succeed if `square` is suitably defined. *Hint:* We need some way to call the predicate with arguments. The Prolog goal `Q =.. [p,x,y]` unifies `Q` with `p(x,y)`, and the goal `call(p(x,y))` succeeds if Prolog can prove `p(x,y)`.

SAMPLE SOLUTION

```
map(_,[],[]).
map(P,[X|L],[Y|M]) :- Q =.. [P,X,Y], call(Q), map(P,L,M).
```

Question 3. Prolog Data Structures [10 MARKS]

Suppose a directed graph $G = (V, E)$ is represented in Prolog using the predicate `vertex(X)`, which is true if the value of `X` is a vertex, and predicate `edge(X,Y)`, which is true if there is an edge from `X` to `Y` in G. A path is a list of vertices such that every pair is an edge of G.

Part (a) [4 MARKS]

Assume the graph is acyclic (contains no cycles). Write a predicate `path(X,Y,P)` which is true if `P` is a path in G from vertex `X` to `Y` (`P` must start with `X` and end with `Y`). Do not use any helper predicates.

SAMPLE SOLUTION

```
path(X,X,[X]).
path(X,Y,[X|P]) :- edge(X,N), path(N,Y,P).
```

Part (b) [6 MARKS]

Graphs, in general, may contain cycles. Our specification of `path/3` above might get caught forever in a cycle. Rewrite `path/3` using a helper predicate such that only simple paths are found (a path is simple if it never visits a vertex more than once).

SAMPLE SOLUTION

```
path(X,Y,P) :- path(X,Y,P,[]).
path(X,X,[X],V) :- \+ member(X,V).
path(X,Y,[X|P],V) :- edge(X,N), \+ member(N,V), path(N,Y,P,[X|V]).
```

Question 4. Prolog Search Trees [10 MARKS]

Consider the following Prolog program.

```
a(X) :- b(X), c(X).
a(X) :- e(X), d(Z).
b(X) :- f(X).
c(X) :- d(X).
d(2).
d(3).
e(4).
e(1).
f(1).
f(2).
```

Part (a) [8 MARKS]

Draw the search tree for the Prolog query `?- a(X).` and list, in the order they are produced, all the answers Prolog would give.

SAMPLE SOLUTION

```
X = 2 ;

X = 4 ;

X = 4 ;

X = 1 ;

X = 1 ;

No
```

Part (b) [2 MARKS]

Suppose the first rule is replaced with the rule

```
a(X) :- b(X), !, c(X).
```

List any answers to for the query `?- a(X).` that Prolog would find using this modified program.

Prolog will report:

```
?- a(X).

No
```

Question 5. Unification [10 MARKS]

What is the term resulting from unifying the following pairs of expressions? Write the answer in the appropriate box, or write "cannot" if the two expressions cannot be unified.

Expression 1	Expression 2	Unification Result
`p(X, f(b))`	`p(a, Y)`	`p(a, f(b))` bindings: `X = a, Y = f(b)`
`q(f(a), _, g(h(c)))`	`q(f(_), b, X)`	`q(f(a), b, g(h(c)))` bindings: `X = g(h(c))`
`q(X, f(g(Y)), Y)`	`q(a, f(Z), h(b))`	`q(a, f(g(h(b))), h(b))` bindings: `X = a, Y = h(b), Z = g(h(b))`
`X + 3`	`7`	cannot
`[[Y|_], b, c]`	`[X,b,Y]`	`[[c|_G181], b, c]` bindings: `Y = c, X = [c|_G181]`

Question 6. Context-Free Grammars [10 MARKS]

Suppose we are writing an new programming language "MUL6" language (Made-Up Language 2006). One task is to formally specify the language's grammar. Suppose we want several operators of various precedence and associativity:

Precedence	Operators	Operator Type	Associativity
highest	`~`, `!`	prefix unary	right to left (i.e., right associative)
	`@`	infix binary	none
	`#`	infix binary	left to right (i.e., left associative)
lowest	`%`, `&`	infix binary	left to right (i.e., left associative)

Recall that higher precedence operators are executed first, and associativity determines which operators of equal precedence are executed first.

Complete the following BNF definition to enforce the operator precedence and associativity rules. Assume that the nonterminals `<identifier>` and `<number>` are defined for you and represent variables and numerical values. You do not have to handle parentheses.

```
<elem> ::= <identifier> | <number>
<expr> ::=
```

Sample Solution

```
<expr> ::= <expr> % <term1> | <expr> & <term1> | <term1>
<term1> ::= <term1> # <term2> | <term2>
<term2> ::= <term2> @ <term3> | <term3>
<term3> ::= ~ <term3> | ! <term3> | <elem>
```

Question 7. Interpretation [14 MARKS]

Suppose we are implementing an interpreter for our "MUL6" language in Scheme. Don't worry about the full details about this fantastic new language, as we'll only touch on a few points in this exam. Be aware though that its syntax is somewhat different from that of Scheme.

Part (a) [5 MARKS]

The following Scheme code implements a Scheme interpreter.

```
(define read-eval-print
  (lambda ()
    (display "--> ")
    (write (eval (read)))
    (newline)
    (read-eval-print)))
```

Note that `read` will read a Scheme expression from standard input and return it, `eval` takes a legal Scheme expression and evaluates it (in the current environment), and `write` prints a Scheme object to standard output.

Explain precisely the most important changes (if any) that must be made to each of the `read`, `eval` and `write` procedures to convert this to an interpreter for MUL6. (Explain what the new code must do, don't try to write the code.)

Answers will vary.

Part (b) Environments [6 MARKS]

Our MUL6 interpreter needs some way to keep track of variables and their values. Our *environment* will keep track of this as a list of lists of variable names and values. For example, the following environment has a bound to 1 and b bound to 2 (notice that the first a shadows the later a):

```
((a 1) (b 2) (a 5))
```

Complete the following Scheme procedures to implement our environment representation.

```
(define empty-env            ; returns an empty environment
  (lambda () '() ))

(define extend-env           ; returns an environment that extends "env" and
  (lambda (vars vals env)    ; binds the "vals" to the "vars"

    (if (null? vars) env
        (cons (list (car vars) (car vals)) (extend-env (cdr vars) (cdr vals) env)))))

(define apply-env            ; returns the value of the variable "var" in "env"
  (lambda (env var)          ; or prints an error if "var" not bound in "env"

    (if (null? env) (error 'apply-env "No binding for ~s" sym)
        (if (eq? (caar env) var) (cadar env)
            (apply-env (cdr env) var)))))
```

Part (c) [3 MARKS]

Suppose MUL6 includes a "`let`" statement (with the expected semantics) as follows (in BNF):

```
<expr> ::= let <identifier> = <expr> {, <identifier> = <expr>}* in <expr>
```

Recall that the Extended BNF notation "`{ A }*`" means `A` can be repeated zero or more times. Suppose the following MUL6 code was presented:

```
let x = 1 in                    /* 1 */
  let v = 5, x = 7 in           /* 2 */
    2
3                               /* 3 */
```

What should the environment look like at the three indicated points in execution? Use the procedures from part (b) and assume the initial environment is stored in the Scheme variable `starting-env`.

`/* 1 */`:

`/* 2 */`:

`/* 3 */`:

Question 8. Parameter Passing [10 MARKS]

Our new programming language "MUL6" uses *pass-by-name* parameter passing. Suppose the following is a procedure declaration in MUL6:

```
procedure swap (x, y)
    var t = x
  begin
    x = y
    y = t
  end
```

Assume that `var` declares a list of variables local to the procedure, and `begin` and `end` delimit the procedure body.

Part (a) [3 MARKS]

Should MUL6 use static or dynamic scope for variables? Explain.

> Dynamic scope. We want the names passed as arguments to be evaluated in the environment of the caller.

Part (b) [4 MARKS]

What happens if we try to use this procedure to swap the values of variables `s` and `t`? That is, what problem do we encounter when we try to call `swap(s,t)`? How could we solve this problem?

> The names `t` will clash. A solution would be to ensure that the "local" `t` gets a new, unique name when `t` appears within one of its arguments.

Part (c) [3 MARKS]

Explain what problem occurs if we try to call `swap(i, a[i])`, where `a` is an array. How can we get `swap` to exchange the values of `i` and `a[i]`?

> The value of `i` will already have changed, so the final mention of `a[i]` will refer to an unexpected array location. A solution would be to call `swap(a[i], i)` instead. We need to ensure that the binding of `x` does not appear within the expression `y`.

Question 9. Scheme Macros [10 MARKS]

The following is an "object" implemented in Scheme:

```
(define 2d-vector
  (lambda (inx iny)
    (let ((x inx) (y iny))
      (lambda args
        (let ((msg (car args)))
          (apply (cond ((equal? msg 'x) (lambda () x))
                       ((equal? msg 'y) (lambda () y))
                       ((equal? msg 'stretch)
                        (lambda (factor)
                          (set! x (* x factor)) (set! y (* y factor))))
                       ) (cdr args))))))
```

We want a Scheme syntactic form (macro) named `class` so that we can write the above as:

```
(define 2d-vector
  (class (inx iny)
    ; list of pairs of instance variable and initial value
    ((x inx) (y iny))
    ; pairs of method header and body
    ((x) x)
    ((y) y)
    ((stretch factor) (set! x (* x factor)) (set! y (* y factor)))))
```

Fill in the blanks below to complete the `class` macro. It should work for any "class" with any number of instance variables, any number of methods, and any number of arguments for methods.

SAMPLE SOLUTION

```
(define-syntax class
  (syntax-rules ()
    ((_ initials vars ((method-name margs ...) body ...) ...)
     (lambda initials
       (let vars
         (lambda args
           (let ((msg (car args)))
             (apply (cond
                      ((equal? msg 'method-name) (lambda (margs ...) body ...))
                      ...)
               (cdr args)))))))))
```

Question 10. Sorting in Scheme [10 MARKS]

Recall that the *Insertion Sort* algorithm inserts elements, one at a time, into a sorted list. For simplicity in this question, we will only consider lists of integers.

Part (a) [5 MARKS]

Suppose we have a procedure `insert` that inserts an element into a sorted list. For example,
`(insert 3 '(1 5 7))` $\Longrightarrow$ `(1 3 5 7)`

Write a Scheme procedure `isort` that takes a list of numbers and performs an *insertion sort*, returning the sorted list. (*Hint:* Insert the first element into a sorted list.)

SAMPLE SOLUTION

```
(define isort
  (lambda (x)
    (if (null? x) x
        (insert (car x) (isort (cdr x))))))
```

Part (b) [5 MARKS]

Write the procedure `insert` in Scheme.

SAMPLE SOLUTION

```
(define insert
  (lambda (a l)
    (cond ((null? l) (list a))
          ((< a (car l)) (cons a l))
          (else (cons (car l) (insert a (cdr l)))))))
```

Question 11. Prolog in Scheme [8 MARKS]

Suppose we are implementing a Prolog interpreter in Scheme. Right now our main interpreter loop looks like this:

```
(define (prolog-rep)
  (display "?- ")
  (if (prove (read-prolog-query))
      (display "Yes\n")
      (display "No\n"))
  (prolog-rep))
```

Assume `read-prolog-query` reads a query from the user and returns it in the form required by `prove`. If `prove` is given a query containing no variables, it returns true if the list of goals can be proven, and false otherwise. If `prove` is given a query containing variables, it outputs each satisfying instantiation of the variables (by calling the `print-instantiation` procedure each time) and (like Prolog) returns false once the search tree has been exhausted.

We want to allow computation of the search tree to be stopped after a satisfying instantiation is found, as we can in SWI-Prolog. Rewrite the `prolog-rep` and `print-instantiation` procedures using Scheme continuations so that "Yes" will be printed if another solution might exist, and "No" if the search tree has been exhausted. Define any global variables you might need. You may reference the old `print-instantiation` procedure via the name `old-print-instantiation` (remember to pass on any arguments!). Calling the Scheme procedure `more-solutions?` with no arguments will prompt the user and return true if another solution is desired, and false otherwise (*i.e.*, like pressing `;` or pressing Enter in Prolog, respectively).

Note: Your goal is to have this work just like SWI-Prolog, but without having to return through all the levels of `prove` recursion.

SAMPLE SOLUTION

```
(define done #f)

(define (prolog-rep)
  (display "?- ")
  (if (call/cc (lambda (k) (set! done k) (prove (read-prolog-query))))
      (display "Yes\n")
      (display "No\n"))
  (prolog-rep))

(define print-instantiation
  (lambda args
    (let ((val (apply old-print-instantiation args)))
      (if (not (more-solutions?)) (done #t)
          val))))
```