# UNIVERSITY OF VICTORIA
# EXAMINATION DECEMBER 2007
# CSc 330 F01

NAME:______________________________ STUDENT NO:______________________________

SIGNATURE:______________________________________________________ SECTION: F01

INSTRUCTORS: Dr. M. Cheng DURATION: 3 Hours

TO BE ANSWERED ON EXAMINATION PAPER. NO CALCULATORS ALLOWED.

STUDENTS MUST COUNT THE NUMBER OF PAGES IN THIS EXAMINATION PAPER BEFORE BEGINNING TO WRITE, AND REPORT ANY DISCREPANCY IMMEDIATELY TO THE INVIGILATOR.

THIS EXAMINATION PAPER HAS 14 PAGES. THERE ARE 6 QUESTIONS. ANSWER ALL QUESTIONS.

| For Use of Examiner | | Marker |
|---|---|---|
| 1 | /20 | |
| 2 | /15 | |
| 3 | /15 | |
| 4 | /20 | |
| 5 | /20 | |
| 6 | /10 | |
| Total | /100 | |

1. (20%) **Lambda Calculus and Type Inference**

   (a) (10%)
   Reduce the following $\lambda$-expression to normal form (i.e., contains no more $\beta$-redexes.). Show your steps clearly.

$$( \ (\lambda f.\lambda g.\lambda x.f(g\ x)) \ \ (\lambda f.\lambda x.f\ x) \ ) \ \ (\lambda f.\lambda x.f\ x)$$

(b) (3%)
The notion of *bound* versus *free* variables in Lambda Calculus is related to *lexical scoping* in programming languages (e.g., Java). Explain.

(c) (7%)
Given the following Haskell function,

```
foldr  f  x  l  =
    if l == [] then x
    else  f  (head l)  (foldr f x (tail l))
```

What is the most general type of `foldr`? The type language of Haskell is:

```
<Type> ::= <Var>  |  <Type> "->" <Type>  |  "[" <Type> "]"  |  "Int"
<Var> ::=  "a"  | "b"  | "c"   | "d"
```

2. (15%) **Programming Language Concepts**

   (a) (3%) Name *three* concepts in logic programming (e.g., Prolog) that are unique and different as compared to Java.

   (b) (3%) Name *three* concepts in functional programming (e.g., Haskell) that are unique and different as compared to Java.

   (c) (3%) Name *three* features of Pascal that are different as compared to Java.

(d) (3%) Almost every programming language has an associated BNF grammar. Why? Name *three* reasons.

(e) (3%) Most programming languages (e.g., Java, Pascal, Haskell) use *static-typing* except Prolog. What are the pros and cons of static-typing?

3. (15%) **Compiler and PL/0**

   (a) (3%) Discuss the basic architecture of a compiler. (You may use PL/0 compiler as an example.)

   (b) (3%) A friend of yours decided to write a Haskell to Java compiler. But, he cannot make up his mine about writing this compiler in Java or Haskell. Could you explain to him which is a better choice and why?

(c) (3%) Name *three* features of a programming language (e.g., PL/0) that need the runtime support of *activation records*.

(d) (3%) What does the PL/0 compiler generate after it successfully parses an error-free PL/0 program? Be more specific.(**Hint**: Java.)

(e) (3%) The PL/0 compiler is originally written in Pascal using a parsing technique called *recursive-descent*. Summarize what this technique is about.

4. (20%) **Functional Programming**

   (a) (5%)
   Write a Haskell function `subset u v` which returns true if the list `u` is a subset of list `v`, otherwise false. (Note: All elements in each list are unique; that is, each list represents a set.) e.g., `subset [3,1] [1,5,4,3] => true` and `subset [3,1] [4,1,5] => false`.

   (b) (5%)
   Write a Haskell function `delete e l` which returns a list with *all* occurrences of element `e` in list `l` removed, e.g., `delete 3 [1,3,2,5,3,4] => [1,2,5,4]`. Your solution **must** use the following function `filter`.

```
filter p l =
     if (p (head l))
     then (head l) : (filter p (tail l))
     else filter p (tail l)
```

(c) (5%)
Write a Haskell function `alldiff l1 l2` which returns true if *all* elements of lists `l1` and `l2` are different, false otherwise. For example, `alldiff [6,3] [2,4,1,5] => true`.

(d) (5%)
What is the answer of the following Haskell expression? (Note: `foldr` is defined in Question 1c and `filter` is defined in Question 4b.)

```
foldr  (++)  []  (map  (\u-> filter  (\x-> x==u) [1,2,3])  [2,3,4])
```

where

```
map f l =
     if l == []
     then []
     else (f (head l)) : (map f (tail l))
```

5. (20%) **Logic Programming**

   (a) (5%)
   You are given the following predicate:

```
concat( [], Y, Y ).
concat( [U|X], Y, [U|Z] ) :- concat( X, Y, Z ).
```

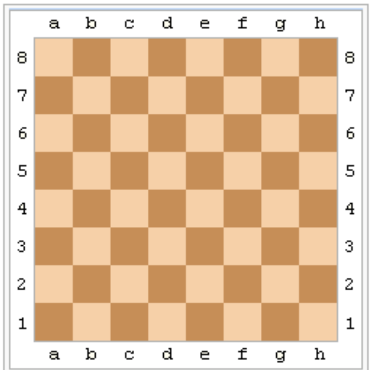
   Show the entire search space (the derivation tree) with all substitutions for the query:

```
?- concat( X, [U|Y], [1,2] ).
```

(b) (5%) Write a Prolog predicate `minimum( L, U )` which holds if `U` is the *minimum* element of list `L`. For example, `minimum( [3,5,2,4], 2 )` is true.

(c) (5%) Write a Prolog predicate `delete(X, L, U)` which holds if `U` is the list after removing an element `X` from the set `L`. (Note: Every element in `L` is unique.) Your solution **must** use `concat` as defined above. For example, `delete( 2, [3,5,2,4], [3,5,4] )` is true.

(d) (5%)
You are given the following predicates where a Knight moves on a 8x8 chess board. `kmove((X1,Y1), (X2,Y2))` which holds if a Knight can move from the position



Chessboard

`(X1,Y1)` to the position `(X2,Y2)` in a single move. And `kpath(P1, P2, Path)` holds if there exists a *non-cyclic* `Path` from position `P1` to `P2` inclusively. (You may assume `length(L, N)` is predefined, which holds if `N` is the length of list `L`.) Write a query or a predicate to answer each of the following.

i. (1%) "Can a Knight move from `(a,1)` to `(b,7)` in 3 moves?"

ii. (1%) "Is there a path from `(a,1)` to `(h,1)`?"

iii. (1%) "Are there two distinct paths from `(a,1)` to `(h,8)` and back?"

iv. (1%) "Is there a path from `(a,1)` to `(h,8)` in *exactly* 5 moves?" (**Note**: The path [(a,1),(b3)] is a single move.)

v. (1%) "Is there a path from `(a,1)` to `(h,8)` that *must* visit `(d,4)`?"

6. (10%) **Problem Solving**

   (a) (5%) Use the Knight's move on a chess board as a problem. Write a Prolog program which answers the question whether "there exists a path that visits all positions along a diagonal *exactly* once", that is, all the positions `(a,1), (b,2), (c,3), (d,4), (e,5), (f,6), (g,7), (h,8)`. Such a path must contain *at least* all these positions, but could contain many more. (You may use any predicate defined or given earlier.)

(b) (5%) A *Mersenne* number is a number that is one less than a power of 2. That is,

$$M_n = 2^n - 1$$

for all $n > 1$. A *Mersenne* prime number is a *Mersenne* number which is also prime. For example, $M_1 = 1, M_2 = 3, M_3 = 7, M_5 = 31$ are all prime numbers, where $M_4 = 15, M_6 = 63$ are not. Modify or extend the following prime number generator `allprimes` (based on *Sieve*) and Mersenne number generator `mersennes` so that one can generate an infinite sequence of *Mersenne* prime numbers. (Note: `n^m` computes $n^m$ in Haskell.)

```
allprimes = 1: (primes [2..])

primes l = (head l) : (primes (filter (notDivisibleBy (head l)) (tail l))

notDivisibleBy n = \x -> ((mod x n) /= 0)

mersenne n = 2^n -1

mersennes = map mersenne [1..]
```

**END**