National University of Singapore
School of Computing
CS3234 — Logic and Formal Systems

September 18, 2004

Assignment 1

(5% of final grade)

Deadline: October 10, 2004

This assignment is designed to introduce you to the logic programming paradigm, and to the programming language Prolog. You shall be required to write two small Prolog programs and, in the process, learn a few basic concepts about programming in Prolog. The Prolog interpreter that we use throughout this assignment is "ECLiPsE", and can be accessed through the CS5216 (Constraint and Logic Programming) module account, by running the following command on the `sunfire` system.

```
$ /home/course/cs5216/eclipse/bin/sparc_sunos5/eclipse
```

The `$` is the Unix prompt. The system will respond with the following prompt:

```
ECLiPSe Constraint Logic Programming System [kernel]
Copyright Imperial College London and ICL
Certain libraries copyright Parc Technologies Ltd
GMP library copyright Free Software Foundation
Version 5.7 #26, Mon Dec 15 00:13 2003
[eclipse 1]:
```

You can now input goals at the ECLiPsE prompt. For example you can try the following goal (`append` is a predefined predicate, whose definition has been given in the lecture notes).

```
append([a,b,c],[d,e,f],A).
```

The system replies with:

```
A = [a, b, c, d, e, f]
Yes (0.00s cpu)
```

The period '.' at the end of the goal is significant, acting as a separator between goals and clauses, so please be sure to include it in your goal. The `append` predicate takes three lists as parameters, and it is true only when the concatenation of the first two lists is equal to the third. If one of the parameters is a variable, the Prolog interpreter looks for an

answer substitution that will make the atom in the goal true. Using this property, we can perform useful computation. In the example above, using linear resolution, the system is able to compute the concatenation of two lists. Let's now try another goal:

```
append([a,b,c],A,[a,b,c,d,e,f]).
```

The system will reply with the following answer substitution:

```
A = [d, e, f]
Yes (0.00s cpu)
```

Interestingly enough, the append predicate can compute the *difference* between two lists, not only their concatenation. Let's try yet another goal.

```
append(A,B,[a,b,c]).
```

The system replies with

```
A = [ ]
B = [a, b, c]
Yes (0.00s cpu, solution 1, maybe more) ?
```

The system replies, as usual, with an answer substitution. Indeed, the empty list, which is the value of `A`, concatenated with `[a,b,c]`, which is the value of `B`, produces the list `[a,b,c]` given in the goal. However, this is not the only solution to the goal, and the system asks the user whether he/she is interested in seeing other solutions. The user has the choice of typing semicolon ';' or Enter. The semicolon will cause other solutions to be printed, while the Enter keypress returns the user to the ECLiPsE prompt. By pressing semicolon repeatedly, all possible solutions to the given goal shall be printed on the screen.

```
A = [ ]
B = [a, b, c]
Yes (0.00s cpu, solution 1, maybe more) ?  ;

A = [a]
B = [b, c]
Yes (0.00s cpu, solution 2, maybe more) ?  ;

A = [a, b]
B = [c]
Yes (0.00s cpu, solution 3, maybe more) ?  ;

A = [a, b, c]
B = [ ]
```

```
Yes (0.00s cpu, solution 4)
```

Let's now progress to writing our own programs. Use a text editor to type the following three clauses.

```
gcd(X,0,X).
gcd(X,Y,Z) :- X >= Y, Y > 0, W is X mod Y, gcd(Y,W,Z).
gcd(X,Y,Z) :- X < Y, gcd(Y,X,Z).
```

Then, save the three clauses in the file `gcd.prolog` . Now, we can load the file in ECLiPsE, by typing the following command at the ECLiPsE prompt.

```
["gcd.prolog"].
```

The system responds with

```
gcd.prolog compiled traceable 540 bytes in 0.00 seconds
Yes (0.01s cpu)
```

Now, we can try the following goal at the ECLiPsE prompt:

```
gcd(40,206,A).
```

The system answers with

```
A = 2
Yes (0.00s cpu, solution 1, maybe more) ?  ;

No (0.00s cpu)
```

The system produces the correct answer `A=2`, but then erroneously assumes that there may be another solution. When we press semicolon, the system discovers that there is no other solution, and replies 'No'. This behavior, while correct, may be annoying; we would very much prefer that the system guesses correctly when there are no more solutions, and refrains from offering to continue to search. However, learning how to avoid redundant searches is an advanced feature beyond the scope of this module, and throughout this assignment, we shall regard redundant searches as acceptable.

More information about Prolog and logic programming can be found on the CS5216 site:

```
http://www.comp.nus.edu.sg/~cs5216
```

There, you shall find links to full documentation on ECLiPsE Prolog, tutorials on logic programming, and example programs.

Exercise 1: Towers of Hanoi

Write a Prolog program that solves the Towers of Hanoi puzzle. You need to define a predicate `hanoi` taking 5 parameters: a natural number `N`, three constants representing poles, and a list `L` containing terms of the form `move(pole1,pole2)`. Each term of the form `move(pole1,pole2)` represents the move of a disk from `pole1` to `pole2`. The predicate `hanoi` must be true whenever the sequence of moves in `L` leads to a solution to the puzzle. For example, the goal

```
hanoi(3,src,aux,dest,L).
```

must be answered with

```
L = [ move(src, dest), move(src, aux), move(dest, aux), move(src, dest),

      move(aux, src), move(aux, dest), move(src, dest) ]
```

Exercise 2: Non-Attacking Queens

This question requires you to solve the n-queens puzzle. The problem is to find all ways of placing n non-taking queens on a $n \times n$ chess board. A queen attacks all cells in its same row, column, and either diagonal. Therefore, the objective is to place n queens on an $n \times n$ board in such a way that no two queens are on the same row, column or diagonal. Below we show a solution on a standard 8×8 chessboard.

					Q		
		Q					
				Q			
						Q	
Q							
			Q				
	Q						
							Q

We will represent a solution to the n-queens puzzle as a permutation of the set of numbers $\{1, \ldots, n\}$. Such a representation is sound, because we will have *exactly one queen* on each row and column, and so if permutation $p : \{1 \ldots n\} \mapsto \{1 \ldots n\}$ is a solution to the puzzle, then $(k, p(k))$ will specify the position of the k^{th} queen, for $1 \leq k \leq n$.

In Prolog, such a permutation will be encoded as a list in which the numbers $1, \ldots, n$ appear *exactly once*, in some order.

We shall solve this question in several steps.

2.1 The first step is to define a predicate `sel`, which takes three parameters, a list, a constant, and another list, and which is true if the second parameter is a member of the list that makes up the first parameter, and if the third parameter is the first parameter, with the element making up the second parameter removed. The system should produce the following answers to the following two goals:

```
[eclipse 2]:  sel([2,1,4],A,B).

A = 2
B = [1, 4]
Yes (0.00s cpu, solution 1, maybe more) ?  ;

A = 1
B = [2, 4]
Yes (0.00s cpu, solution 2, maybe more) ?  ;

A = 4
B = [2, 1]
Yes (0.00s cpu, solution 3, maybe more) ?  ;

No (0.00s cpu)


[eclipse 4]:  sel(A,a,[1,2]).

A = [a, 1, 2]
Yes (0.00s cpu, solution 1, maybe more) ?  ;

A = [1, a, 2]
Yes (0.00s cpu, solution 2, maybe more) ?  ;

A = [1, 2, a]
Yes (0.00s cpu, solution 3)
```

2.2 Define a predicate `gen` that takes as parameters a positive number `N` and a list of integers, and which is true if the second parameter is the list of numbers `[N,...,2,1]`. The following is an example goal, and the corresponding answers.

```
[eclipse 5]:  gen(5,L).

L = [5, 4, 3, 2, 1]
Yes (0.00s cpu, solution 1, maybe more) ?  ;

No (0.00s cpu)
```

2.3 Define a predicate `not_attack` that takes four numbers as parameters. Each pair of numbers, the first two, and the last two, represent the coordinates of a queen, and the predicate should be true if the two queens do not attack each other. The following are example goals, and the corresponding answers.

```
[eclipse 6]:  not_attack(2,3,3,2).

No (0.00s cpu)

[eclipse 7]:  not_attack(2,3,5,4).

Yes (0.00s cpu)
```

2.4 Define a predicate `not_attack_list`, which takes four parameters:

- A list `L` of integers, representing y (vertical) coordinates of some queens;
- An integer `K` which encodes the x (horizontal) coordinates of the queens in list `L`: the first queen in `L` has the x coordinate equal to `K`, the second element of `L` has the x coordinate equal to `K+1`, the third element of `L` has the x coordinate equal to `K+2`, and so on.
- the last two parameters are positive integers representing the coordinates of another queen.

The predicate `not_attack_list` is true if the queen whose coordinates are the last two parameters does not attack any of the queens represented by the list `L`. In defining this predicate, it shall be useful to use the predicate `not_attack` defined above. Here is an example goal for `not_attack_list`.

```
[eclipse 8]:  not_attack_list([4,8,3],6,1,2).

Yes (0.00s cpu)

[eclipse 9]:  not_attack_list([5,8,3],6,1,2).

No (0.00s cpu)
```

2.5 You have now all the tools to solve the N-Queens puzzle. Use the following strategy. The solution will be a permutation of `[1,2,...,N]`, or `[N,...,2,1]`, which can be generated by the predicate `gen`. Construct the solution starting with an empty board, which is represented by the empty list. Select (using `sel`) a number from the list `[N,...,2,1]` (or whatever is left of it), and see if placing that number at the beginning of the partial solution found so far produces a board of non-attacking queens (using `not_attack_list`). If it does, add the selected number to the partial solution, and continue the process (i.e. call your predicate recursively). You don't need to worry about what happens if your new queens attacks one of the other. The predicate will simply not be true, and no solution will be returned by the system. Here's a sample call, and a sample answer (there are 92 solutions for the 8×8 case; we shall not list all of them here).

```
[eclipse 11]:  nqueens(8,L).

L = [5, 7, 2, 6, 3, 1, 4, 8]
Yes (0.00s cpu, solution 1, maybe more) ?
```

This solution represents the board given in the figure at the beginning of this exercise.