CS 131 Programming Languages

October 29, 2009

Midterm Grade Distribution

Grades: 80-89: 2, 70-79: 5, 60-69: 13, 50-59: 6, 40-49: 12, 30-39: 12, 20-29: 6, 10-19: 2. Mean is 48.7, median is 46.

Question 6 from the midterm (implementing currying using Java or C++):

```
Cfo f (int x) {
        return new Cfo(x);
}
public class Cfo {
        int x;
        public Cfo (int x0) {
                x = x0;
        }
        public int fun (int y) {
                return x + y;
        }
}
```

Homework 4 and Prolog

Kenken is a puzzle like Sudoku, from New York Times. In Kenken there are extra constraints to solve when filling the puzzle. The assignment is to write Kenken solver in Prolog.

Logic Programming and Prolog

In *imperative style*, such as in Java and C++, we have assignments and calls and the computation proceeds from these. In *functional style*, such as OCaml, there are no assignments to make. In *logic style* such as Prolog, there are no assignments and no calls.

There are function symbols but they're never called. You "talk about functions" but you never invoke them. The philosophy of Prolog is "Algorithm = Logic + Control." You write down the *logic* of the program (series of axioms and rules in which the system can infer properties of the wanted solution) and the system decides how to solve the problem. It always follows the logic but it can try many different ways to solve the problem. *Control* is the advice to the system about the most efficient way of formulating the answer. The idea is that the problem is partitioned in two pieces. *Logic piece* worries about the correctness of problem specification (programmer's responsibility). *Control piece* can be defaulted to let the system make all the decisions. Often, the system does a bad job of deciding and comes up with an inefficient solution to the problem. Control piece only affects efficiency of the solution, without affecting correctness. The advice to the solution can be supplied by the programmer.

Let's look at Prolog code that sorts a list. L is the unsorted list and S is the sorted list. Instead of a function we have to define a predicate that's true if S is the sorted version of L. S is the sorted version of L if L is a permutation of S and S is sorted. The *:-* operator signifies *if* statement.

```
sort(L, S) :- perm (L, S), sorted (S).

sorted([]).
sorted([_]).
sorted([X, Y | L]) :- X =< Y, sorted([Y | L]).

perm([], []).
perm([X | L], AXB) :-  perm(L, AB),
                       append(A, B, AB),
                       append(A, [X | B], AXB).

append([], L, L).
append([X | L], M, [X | LM])  :- append(L, M, LM).

Example run:
        append([3,4],[],R).
                append([4],[],R1).
```

```
                        append([],[],R2).
```

Comparing OCaml and Prolog syntax:

OCaml	**Prolog**
`h::t`	`‘.’(H,T)`
`[3;7;9]`	`[3,7,9]`
`3::7::9::[]`	`‘.’(3, ‘.’(7, ‘.’(9, ‘[]’)))`

Let’s try to find whether X is a member of a list.

```
member(X, [X|_]).
member(X, [_|L]) :- member(X, L).

member (3, [X, 4, Y]).
X = 3;
Y = 3;
member(3, L).
L = [3, _1967];
L = [_1463, 3 | _1291];
L = [_91, _47, 3 | _19672];
```

Let’s try to compute a reverse of a list (using *naïve reverse*, O(n^2)).

```
reverse([], []).
reverse([X|L], _lX) :- reverse(L, _l), append(_l, [X], _lX).
```

LIPS: logical inferences per second. Number of steps per second on a naïve reverse. We can improve the above code to O(n) using an *accumulator*.

```
reva(L,A,R)    // R is the reverse(L) concatenated with A
reva([],A,A).
reva([X|L], A, R) :-   reva(L, [X|A], R).
```

Syntax of Prolog

Term	**Example**	**Description**
atom	abc	Starts with a lowercase letter and can have letters, digits, underscores. *atom* can start with and have any character if it’s surrounded by single quotes
number	3	
variable	_34, X, _	Stands for arbitrary value.
structure	functor(arg_1,…,arg_n)	Could be thought of like OCaml constructors.

Syntactic Sugar

Shortcut	**Full Expression**
`[]`	`‘[]’`
`[x,y]`	`‘.’(x, ‘.’(y, ’[]’))`
`3+5`	`‘+’(3,5)`
`3+5*7`	`‘+’(3,’*’(5,7))`

`is/2` is a function symbol with *arity* 2 (*arity* is the number of arguments to the data structure).

```
?- is(N, ‘+’(5, 7))
N = 12
?- N = 10, N is N + 1
no
```