

# Design of Programming Languages 91.531

## Problem Set 8

Due: Thursday 05-May-2005

For this problem set, use SWI Prolog, installed as `pl` on mercury. Or download your own copy from [www.swi-prolog.org](http://www.swi-prolog.org). Use the recommended sections in Periera & Shieber as well as Chapter 10 in the textbook as a reference.

### Problem 1. [25] Elementary Prolog

Program the following in Prolog. Do not use any built-in predicates unless instructed to do so.

**Part a.** [5] `final(?l, ?e)` is true if `e` is the last element of list `l`.

**Part b.** [10] `rev(?unreved, ?partreved, ?allreved)` is true if the reverse of the list `unreved` appended to the list `partreved` is the same as the list `allreved`. use `rev/3` to write `rev/2` that relates a list to its reverse.

**Part c.** [5] `flatten(+nested, -flat)` is true if `nested` is a list possibly containing elements that are lists nested to an arbitrary depth, and `flat` is a list with no sublists that contains the elements of all the nested lists in the order in which they print out.

Try running this backwards: you will notice that Prolog is “unfair” there are some patterns of nesting that you can never recreate from your flat list.

**Part d.** [5] `llength(+l, ?n)` is true if list `l` has length `n`. (You will have to use numbers as well as logic...)

### Problem 2. [10] Partially specified data structures in Prolog

Considering the following code:

```
lookup(Key,Data,bt(Key,Data,_LT,_RT)) :- !.
lookup(Key,Data,bt(Key0,_Data0,LT,_RT)) :- Key @< Key0, lookup(Key,Data,LT).
lookup(Key,Data,bt(Key0,_Data0,_LT,RT)) :- Key @> Key0, lookup(Key,Data,RT).
```

Given the compound goal

```
lookup(91.503,algorithms,BT), lookup(91.111,_,BT), lookup(91.531,dp1,BT),
lookup(91.502,foundations,BT), lookup(91.531,Name,BT).
```

What does `BT` look like after each sub-goal? Why do we not need leaves in this binary tree implementation?

### Problem 3. [10] Backtracking in Prolog

Chapter 10, Exercise 10 in the book. Note that `print` in Prolog does not automatically put on a line terminator as it does in `uProlog`, so your output will be different. Feel free to look up the `format` predicate if you want prettier output.

### Problem 4. [15] Applied Prolog

Chapter 10, Exercise 14 in the book.

Here is some support code for programming the uML evaluator in Prolog. Replace the lines that read “`print('LeftAsExercise'), !, fail.`”, and provide some test cases.

```

%%% problem14.pl
%%% This file implements evaluation for uML.
%%% The problem says nothing about type-checking uML input so
%%% we assume that ill-typed input will just fail to evaluate due
%%% to lack of evaluation rules.

%%% we use "true" and "false" internally for boolean values, but can print
%%% #t and #f using the SWIprolog builtin portray/1.

%portray(true) :- print('#t').
%portray(false) :- print('#f').

%%% environment manipulation, uses handy infix operator ->
%%% find (Var-,Env+,Val-) finds the value for a variable in an environment
%%% or fails if not bound.
%%% bind (Var+,Val+,OldEnv+,NewEnv-) adds a binding to an environment.
%%% bindlist(Vars+,Vals+,Oldenv+,Newenv-) adds multiple bindings.
%%%

find(X,[X -> Y|_],Y) :- !.
find(X,[_|L],Y) :- find(X,L,Y).

bind(X,Y,L,[X -> Y | L]).

bindlist([], [], Rho, Rho).
bindlist([HX|TX], [HV|TV], Rho, Rho2) :-
    bind(HX, HV, Rho, Rho3),
    bindlist(TX, TV, Rho3, Rho2).

%%% unzip for [(x1,e1), ..., (xn,en)]
%%%
unzip([], [], []).
unzip([(X,E)|More], [X|MoreXs], [E|MoreEs]) :- unzip(More,MoreXs,MoreEs).

%%% opposite used for some relational expressions.
%%%
opposite(lt, ge).
opposite(gt, le).
opposite(eq, ne).

%%% Binary operation for primitive operations.
%%% the boolean operations have values in {true, false}

binop(cons, X, Y, [X,Y]).

```

```

binop(append,X, Y, Z) :- append(X,Y,Z).
binop(and, true, true, true) :- !. /* do not backtrack to try false! */
binop(and, _, _, false).
binop(or, false, false, false) :- !. /* do not backtrack to try true! */
binop(or, _, _, true).
binop(plus, X, Y, V) :- V is X + Y.
binop(minus, X, Y, V) :- V is X - Y.
binop(times, X, Y, V) :- V is X * Y.
binop(div, X, Y, V) :- V is X / Y.
binop(lt, X, Y, true) :- X < Y.
binop(gt, X, Y, true) :- X > Y.
binop(eq, X, Y, true) :- X == Y.
binop(le, X, Y, true) :- X =< Y.
binop(ge, X, Y, true) :- X >= Y.
binop(ne, X, Y, true) :- X =\= Y.
binop(C, X, Y, false) :-
    opposite(C, C1),
    binop(C1, X, Y, true).

is_binop(X) :- member(X, [cons, append, and, or, plus, minus, times, div,
    lt, gt, eq, le, ge, ne]).

unop(car, [X,_], X).
unop(cdr, [_ ,Y], Y).
unop(list1, X, [X]).
unop(length, X, N) :- length(X,N).
unop(not, true, false).
unop(not, false, true).

is_unop(X) :- member(X, [car, cdr, list1, length, not]).

%%% evallist used by eval

evallist([], _Rho, []) :- !. /* succeeded or failed: do not retry */
evallist([HE|TE], Rho, [HV|TV]) :- eval(HE, Rho, HV), evallist(TE, Rho, TV).

%%% eval(+Expression,+Environment,-Value)
%%%
eval(print(E), Rho, V) :- eval(E, Rho, V), print(V).
%
% evaluate literals
% pair and nil make prolog lists.
%
eval(literal(true), _Rho, true) :- !.

```

```

eval(literal(false), _Rho, false) :- !.
eval(literal(nil), _Rho, []) :- !.
eval(literal(pair(H,T)), _Rho, .(LH,LT)) :-
    eval(literal(H), _Rho, LH),
    eval(literal(T), _Rho, LT).
eval(literal(X), _Rho, X) :- atomic(X).
%
eval(var(X), Rho, V) :- print('LeftAsExercise'), !, fail.
%
eval(if(E1, E2, E3), Rho, V) :- print('LeftAsExercise'), !, fail.
%
eval(lambda(XS, E), Rho, V) :- print('LeftAsExercise'), !, fail.
%
eval(begin(ES), Rho, V) :- print('LeftAsExercise'), !, fail.
%
eval(apply(E, ES), Rho, V) :-
    eval(E, Rho, Op),
    is_binop(Op),
    evallist(ES, Rho, [X,Y]),
    binop(Op, X, Y, V).          /* binary primitive operations */
eval(apply(E, ES), Rho, V) :-
    eval(E, Rho, Op),
    is_unop(Op),
    evallist(ES, Rho, [X]),
    unop(Op, X, V). /* unary primitive operations */
eval(apply(E, ES), Rho, V) :- print('LeftAsExercise'), !, fail.
%
eval(let(Bs,Body), Rho, V) :- print('LeftAsExercise'), !, fail.
%
eval(letstar(Bs,Body), Rho, V) :- print('LeftAsExercise'), !, fail.
%
eval(letrec(Bs,Body), Rho, V) :- print('LeftAsExercise'), !, fail.
%
% The names of all primops are self-evaluating
%
eval(X, _, X) :- is_binop(X).
eval(X, _, X) :- is_unop(X).

```

### Problem 5. [40] Prolog in Prolog

Recall the three clause metacircular interpreter presented in class:

```

prove(true).
prove(Goal) :-
    clause(( Goal :- Body )),

```

```

    prove(Body) .
prove((Body1, Body2)) :-
    prove(Body1),
    prove(Body2) .

```

**Part a. [8]** Metacircular interpreter with Disjunctive goals

Many Prolog systems allow disjunction in predicate definitions. The semicolon operator is used for disjunction, as in the following alternate definition of the `anc/2` predicate:

```

anc(X, Y) :-
    parent(X, Y);
    (parent(X, Z), anc(Z, Y)).

```

The semicolon operator adds no power to the language, as it could be defined as follows:

```

G1 ; G2 :- G1.
G1 ; G2 :- G2.

```

Extend the three clause metacircular interpreter presented in class so that it handles goals of the form `G1;G2`. Demonstrate that the interpreter handles clauses with disjunctions such as the example above.

**Part b. [12=5+7]** Metacircular interpreter with If

Conditionals can be encoded directly in Prolog using the `-> ... ;` operator, rather than with cuts. For example, the implicit conditional in the definition of `merge/3` can be restated explicitly:

```

merge(A, [], A).
merge([], B, B).
merge([A|RestAs], [B|RestBs], [AorB|Merged]) :-
    A < B
    -> (merge(RestAs, [B|RestBs], Merged), AorB = A)
    ; (merge([A|RestAs], RestBs, Merged), AorB = B)).

```

Note the use of the `=/2` predicate to give the value of `AorB` in the head of the clause.

In general, a goal of the form  $G_1 \rightarrow G_2; G_3$  is interpreted such that if the goal  $G_1$  succeeds then the entire goal succeeds if  $G_2$  does; otherwise if  $G_1$  fails, the entire goal succeeds if  $G_3$  does.

**i.** Show that the `->` notation can be implemented in Prolog itself, as as done for `;` above.

Note that this notation is already defined in SWI Prolog. you will need to use

```

:- module_transparent (->)/2.+
:- module_transparent (;)/2.

```

To tell SWI Prolog to let you override its definition.

**ii.** Extend the three clause metacircular interpreter presented in class so that it handles goals of the form  $G_1 \rightarrow G_2; G_3$ . Demonstrate that the interpreter handles clauses with conditionals such as the example above.

**Note:** It is considered poor Prolog programming style to make extensive use of these constructs. They should be used sparingly, and only in those cases where there is compelling reason (usually having to do with readability). The examples above are not especially in this form.

## Proposed metacircular interpreter with cut

In the next problem we consider the problem of writing an interpreter for pure Prolog with the impure cut operator. For purposes of this problem, we assume that clauses have at most one cut in them. There are two cases to consider:

1. If there is no cut in the clause, we can interpret it as before.
2. If there is a cut in the clause, we must interpret the part before the cut, then cut away any further choices of clause clauses, and then interpret the part after the cut.

Notice that we use a standard device for writing metacircular interpreters: “absorption”. We augment the object language with cut by augmenting the meta-language with cut (so the language of the interpreter is no longer pure Prolog). In essence, we absorb the execution of cut by using the cut of the underlying execution.

We need a predicate, call it `cut_split(Body, Before, After)`, which takes a `Body` of a clause and finds the parts `Before` and `After` the cut (if there is one). The predicate fails if there is no cut.

```
%%% cut_split(+Body, -Before, -After)
%%% =====
%%%
%%%   Body is a comma-separated list of terms, and Before and After
%%%   are comma-separated sublists before and after the single ! in
%%%   the Body.  If no ! exists, the predicate fails.

cut_split(!, true, true).
cut_split(!, After), true, After) :-
    !.
cut_split((Lit, Rest), (Lit, Before), After) :-
    cut_split(Rest, Before, After).

%%% prove(+Goals)
%%% =====
%%%
%%%   Holds if Goals is provable according to Prolog search
%%%   strategy.  A single cut per clause is allowed.

prove(true).
prove(Goal) :-
    clause(( Goal :- Body )),
    cut_split(Body, Before, After),
    prove(Before),
    !,                               % absorb interpreter cut using Prolog cut
    prove(After).
prove(Goal) :-
```

```

    clause(( Goal :- Body )),
    prove(Body).
prove(( Goal, Rest )) :-
    prove(Goal),
    prove(Rest).

```

(Note: the metacircular interpreter as handed out contained some typos. One, an extra space between `clause` and `(` created an error message and was easily fixed. The other, missing parens in the last clause of `prove`, would not produce an error message and would be harder to find and fix, but was irrelevant to the example code.)

This metacircular interpreter is almost faithful in that it exhibits almost identical behavior as Prolog itself does, but not quite. The following program shows the interpreter's infidelity:

```

p(a) :- true.
p(X) :- q(X), !.
q(b) :- true.

```

**Part c.** [10] Demonstrate that the interpreter is not faithful to Prolog on the above program. What accounts for this anomaly?

**Part d.** [10] Fix the metacircular interpreter so it is faithful on this and similar examples. (Hint: The Prolog constructs above may be useful).

**Problem 6.** [Extra credit to be determined] Exercise 10.20 in the book.

I have not had time to try this one. It seems that one could learn a lot about the cut operation from it, but the time taken to understand and modify the guts of uProlog might be excessive. Let me know if you want to try this since I will have to fill in some holes in uProlog to get an implementation where you can even start working on the problem.