

Logic Programming

Manipulating Programs

Temur Kutsia

Research Institute for Symbolic Computation
Johannes Kepler University Linz, Austria
kutsia@risc.jku.at

1/24

Contents

Listing

Writing Prolog Interpreter in Prolog

The `consult` Predicate

2/24

Introduction

- ▶ Programs as data.
- ▶ Manipulating Prolog programs with other Prolog programs.
- ▶ Meta-Programming

3/24

`clause` Predicate

`clause(X, Y)`

- ▶ Built-in binary predicate, very important if one wishes to construct programs that examine or execute other programs.
- ▶ Satisfying `clause(X, Y)` causes `X` and `Y` to be matched with the head and body of an existing clause in the database.
- ▶ `X` must be instantiated so that the main predicate of the clause is known.

4/24

clause Predicate

Satisfying `clause(X, Y)`

- ▶ If there are no clauses that match `X`, the goal fails.
- ▶ If there is more than one clause that matches, Prolog returns the first one. The other matches will be chosen, one at a time, when Prolog backtracks.

5/24

clause Predicate. Examples

```
append([], X, X).  
append([A|B], C, [A|D]) :-  
    append(B, C, D).  
  
?- clause(append(L1, L2, L3), Y).  
  
L1 = []  
L2 = L3  
Y = true ;  
  
L1 = [_G463|_G464]  
L3 = [_G463|_G467]  
Y = append(_G464, L2, _G467) ;  
  
No
```

6/24

A Version of `listing` Predicate

```
list1(X)
```

- ▶ Satisfying the goal `list1(X)` will print out the clauses in the database whose head matches `X`.
- ▶ The definition of `list1(X)` will involve `clause` with `X` as the first argument.
- ▶ Therefore, `X` has to be sufficiently instantiated.

7/24

Definition of `list1`

```
list1(X) :-
    clause(X, Y),
    output_clause(X, Y),
    write(' '), nl,
    fail.
list1(_).

output_clause(X, true) :-
    !,
    write(X).
output_clause(X, Y) :-
    write((X:-Y)).
```

8/24

How Does `list1` Work?

- ▶ The first clause causes a search for a clause whose head matches `X`.
- ▶ If one found, it is printed and a failure is generated.
- ▶ Backtracking will reach the `clause` goal and find another clause, if there is one, and so on.
- ▶ When there is no more clause to be found, the `clause` goal will fail.
- ▶ At this point, the second clause for `list1` will be chosen, so the goal will succeed.
- ▶ As a “side effect”, all the appropriate clauses will have been printed out.

9/24

How Does `output_clause` Work?

- ▶ Specifies how the clauses will be printed.
- ▶ It looks for a special case of the body `true`. In this case it just prints the head.
- ▶ Otherwise, it writes out the head and the body, constructed with the functor `:-`.
- ▶ The “cut” in the first rule for `output_clause` says that the first rule is the only valid possibility if the body is `true`.
- ▶ The “cut” is essential because the example relies on backtracking.

10/24

Writing Prolog Interpreter in Prolog

Idea:

- ▶ Define what it is to run a Prolog program by something which is itself a Prolog program.

11/24

The `interpret` Predicate

Idea:

- ▶ `interpret(X)` succeeds as a goal exactly when `X` succeeds as a goal.
- ▶ `interpret` is similar to built-in predicate `call`, but is more restricted: It does not deal with cuts or built-in predicates

12/24

The `interpret` Predicate

```
interpret(true) :-  
    !.  
interpret((G1, G2)) :-  
    !,  
    interpret(G1),  
    interpret(G2).  
interpret(Goal) :-  
    clause(Goal, MoreGoals),  
    interpret(MoreGoals).
```

13/24

The `interpret` Predicate

- ▶ The first clause of `interpret` deals with the special case when the goal is true.
- ▶ The second clause deals with the case when a goal is a conjunction.
- ▶ The third clause covers a simple goal: The procedure is the following:
 1. Find a clause whose head matches the goal
 2. `interpret` the goals in the body of that clause.
- ▶ Limitations: The program will not cope with programs using built-in predicates, because such predicates do not have clauses in the usual sense.

14/24

The `consult` Predicate

- ▶ `consult` is provided as a built-in predicate in most systems.
- ▶ Interesting to see how it can be defined in Prolog.
- ▶ A simplified definition.

15/24

`retractall`

First we define `retractall`:

```
retractall(X) :-  
    retract(X),  
    fail.  
retractall(X) :-  
    retract((X:-Y)),  
    fail.  
retractall(_).
```

16/24

Program for consult

```
consult (File) :-
    retractall(done(_)),
    current_input (Old),
    open(File, read, Stream),
    repeat,
        read(Term),
    try(Term),
    close(Stream),
    set_input (Old),
    !.
```

17/24

Program for consult, Cont.

```
try(end_of_file) :- % End of file marker read
    !.
try((?- Goals)) :-
    !,
    call(Goals),
    !,
    fail.
try((:- Goals)) :- % ignore directives
    !.
try(Clause) :-
    head(Clause, Head),
    record_done(Head),
    assertz(Clause),
    fail.
```

18/24

Program for `consult`, Cont.

```
:- dynamic done/1.

record_done(Head) :-
    done(Head),
    !.
record_done(Head) :-
    functor(Head, Func, Arity),
    functor(Proc, Func, Arity),
    asserta(done(Proc)),
    retractall(Proc),
    !.

head((A :- B), A) :-
    !.
head(A, A).
```

19/24

Program for `consult`. Explanations.

- ▶ `current_input(Old)` and `set_input(Old)` ensure that the current input file stays the same after the consult.
- ▶ `try` is to cause an appropriate action to be taken for each term read from the input.
- ▶ `try` only succeeds when its argument is the end of file mark.
- ▶ Otherwise, a failure occurs after the appropriate action, and backtracking goes back to the `repeat` goal.
- ▶ The “cut” at the end of the `consult` definition cuts out the choice introduced by `repeat`.

20/24

Program for `consult`. Explanations.

The actions `try` performs:

- ▶ Question read:

```
try((?- Goals)) :- !, call(Goals), !, fail.
```

Attempt to satisfy the appropriate goal and fails.

- ▶ Directive read:

```
try((:- Goals)) :- !.
```

Ignore. (Different systems treat them differently.

SWI-Prolog makes no difference between questions and directives.)

21/24

Program for `consult`. Explanations.

```
try(Clause) :-  
    head(Clause, Head),  
    record_done(Head),  
    assertz(Clause),  
    fail.
```

- ▶ When the first clause for a given predicate appears in a file, all the clauses in the database for that predicate must be removed before the new one is added.
- ▶ Clauses must not be removed when later ones for that predicate appear, because then we will be removing clauses that have just been read in.
- ▶ How to determine whether a clause is the first one in the file for its predicate?

22/24

Program for `consult`. Explanations.

```
:- dynamic done/1.

record_done(Head) :- done(Head), !.
record_done(Head) :-
    functor(Head, Func, Arity),
    functor(Proc, Func, Arity),
    asserta(done(Proc)),
    retractall(Proc),
    !.
```

- ▶ Keep record of the predicates for which we have found clauses in the file.
- ▶ When the first clause of a predicate (e.g., of the binary predicate `foo`) is found, then
 - ▶ remove from the database the existing clauses for it, and
 - ▶ add the new clause in the database.
- ▶ In addition, the fact `done(foo(_, _))` is added.

23/24

Program for `consult`. Explanations.

```
:- dynamic done/1.

record_done(Head) :- done(Head), !.
record_done(Head) :-
    functor(Head, Func, Arity),
    functor(Proc, Func, Arity),
    asserta(done(Proc)),
    retractall(Proc),
    !.
```

- ▶ When a later clause for predicate `foo` is read from the file, we will be able to see that the old clauses have already been removed, and so we avoid removing new clauses.
- ▶ The test is `done(Head)` in the first clause, with `Head` instantiated by the head of the clause for predicate `foo`.

24/24