Logic Programming II

09.11.2009

Christian M. Hansen
chrisha@ifi.uio.no

Department of Informatics – University of Oslo

Based on slides by Martin Giese and Arild B. Torjusen

INF 3110/4110 – 2009

Outline

- Repetition
- Consulting vs. Querying
- Lists in Prolog
- Different views of a Prolog program
- Arithmetic in Prolog
- Cut and negation
- Problems with Prolog

INF 3110/4110 – 2009

Facts and rules

- Remember: A declarative program admits two interpretations
  - Declarative: **What** is being computed.
  - Procedural: **How** the computation takes place
- A Prolog program consists of a sequence of *clauses*
- clauses are *facts* (H) or *rules* (H :- $A_1,\dots,A_k$)
  - person(anne, sofia, martin, 1960)
  - child(X,Y) :- person(X,Z,Y,U)
- Declaratively, the rule H:- A_1 , A_2 is read as:
  - "H is implied by the conjunction A_1 , A_2"
- Procedurally, the rule H:- A_1 , A_2 is interpreted as:
  - "To answer the query H, answer the conjunctive query A_1 , A_2"

INF 3110/4110 – 2009

Queries and unification

- We initiate a computation by posing a *query* (|?- $A_1,\dots,A_k$)
  | ?- child(paul,Parent)
- For queries without variables we will get a yes/no answer.
- For queries with variables the result is the substitutions for (assignment of) the variables which will make the query true.
- The process of matching a query with facts and rules is called *unification*. The result of the unification is a *substitution*.

INF 3110/4110 – 2009

Outline

- Repetition
- Consulting vs. Querying
- Lists in Prolog
- Different views of a Prolog program
- Arithmetic in Prolog
- Cut and negation
- Problems with Prolog

INF 3110/4110 – 2009

Consulting vs. Querying

- Prolog treats all expressions entered after | ?- as *queries*
  - Program:
    hungry(anne).
    hungry(sofia).
  - Typing | ?- hungry(martin). produces *no*
- In order to define new predicates, or redefine existing ones, enter *consulting* mode
  - | ?- consult(*file*). or | ?- [*file*]. consults *file*.pl
  - | ?- consult(user). or | ?- [user]. consults user to enter facts and rules directly
  - End user consulting mode with *Ctrl+D*

INF 3110/4110 – 2009

Outline

- Repetition
- Consulting vs. Querying
- Lists in Prolog
- Different views of a Prolog program
- Arithmetic in Prolog
- Cut and negation
- Problems with Prolog

INF 3110/4110 – 2009

Lists in Prolog

- Basic idea: same as in ML.
- Conceptually, a list is either:
  - nil, the empty list
  - cons(hd,tl), the list with head hd and a tail tl
- A list of prime numbers:
  cons(2,cons(3,cons(5,cons(7,nil))))
- BUT: use special syntax [] and [hd | tl]
  [2 | [3 | [5 | [7 | []]]]]

INF 3110/4110 – 2009

Prettier Syntax for Lists

- [] : the empty list
- [a,b,c] : a list with three elements, same as
  [a | [b | [c | []]]]
- [a,b|X] : another way of writing
- [a | [b | X]]

Unification: just like always...

- [a, b, c] ≡ [A | B] will be unified as
- A/a and B/[b, c]

INF 3110/4110 – 2009

Unification on lists

- [a,b,c] unifies with [Head | Tail]
  Result: Head=a and Tail=[b,c]
- [a] unifies with [H | T]
  Result: H=a and T=[ ]
- [a,b,c] unifies with [a | T]
  Result: T=[b,c]
- [a,b,c] does **not** unify with [b | T]
- [] does **not** unify with [H | T]
- [] unifies with []

INF 3110/4110 – 2009

Unification on lists: Example

- Assume the following fact: p([H | T], H, T).
- Query:

```
| ?- p([a,b,c], X, Y).

X=a
Y=[b,c]
yes
```

INF 3110/4110 – 2009

Unification on lists: Example

- Assume the following fact: p([H | T], H, T).
- Query:

```
| ?- p([a], X, Y).
X=a
Y=[]
yes

| ?- p([], X, Y).
no
```

INF 3110/4110 – 2009

Find an element in a list

- Check if the first element is the one we are searching for.
- If not, we look for the element in the rest of the list.
- Either we find X or the list becomes empty.

```
member(X, [X|Rest]).
member(X, [H | Tail]) :- member(X, Tail).
```

member(2,[1,2,3]) ? -> member(2,[2,3]) ? -> yes

INF 3110/4110 – 2009

Append two lists

- We will define a relation to concatenate two lists Xs and Ys into a third list Zs:

| ?- append([1, 2, 3], [4,5], Result). Should give
Result = [1,2,3,4,5].

- Prolog program:

```
append([], Ys, Ys).
append([X | Xs], Ys, [X | Zs]) :- append(Xs, Ys, Zs).
```

INF 3110/4110 – 2009

Functions?

- There are no functions in Prolog, but relations
  - Functions are a particular case of relations
  - This allows using Prolog programs in multiple ways
- A function f: A -> B can be represented in Prolog as a relation relf(a,b)
  - relf(a,b) may be understod as f(a)=b
- So, in append(List1, List2, Result).
  - List1 and List2 may be seen as input parameters
  - Result is the output parameter
- Compare with ML:
  - ML: fun fst(x::xs) = x
  - Prolog: fst([X|Xs],X) .
    | ?- fst([1,2,3],X). X = 1 ? ;

INF 3110/4110 – 2009

Anonymous variables

- When we are not interested in the value of a certain parameter, we may use `_´
- Example: In the program

```
member(X, [X|Rest]).
member(X, [Head | Tail]) :- member(X, Tail).
```

we are not interested in the Head parameter
(nor in the Rest parameter).

- We can write it as follows:

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

INF 3110/4110 – 2009

Outline

- Repetition
- Consulting vs. Querying
- Lists in Prolog
- Different views of a Prolog program
- Arithmetic in Prolog
- Cut and negation
- Problems with Prolog

Multiple uses of a Prolog program (1)

- Some Prolog programs may be used both for testing and for computing
- Example: member(X, Xs) means X is a member of the list Xs

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


Multiple uses of a Prolog program (1)

- For testing:

```
| ?- member(wed, [mon, wed, fri]).
yes
```

- For computing:

```
| ?- member(X, [mon, wed, fri]).
X = mon ?
X = wed ?
X = fri ?
no
```


Multiple uses of a Prolog program (2)

- It's possible to use the same program to concatenate two lists and to split a list in all possible ways
- Example: append(Xs,Ys,Zs)
- To concatenate two lists:

```
| ?- append([first, second, third], [fourth,
  fifth], Zs).

Zs = [first, second, third, fourth, fifth].
```

Multiple uses of a Prolog program (2)

- To split a list in all possible ways:

| ?- append(Xs, Ys, [first, second, third, fourth, fifth]).

Xs = [] Ys = [first,second,third,fourth,fifth] ?

Xs = [first] Ys = [second,third,fourth,fifth] ?

Xs = [first,second] Ys = [third,fourth,fifth] ?

Xs = [first,second,third] Ys = [fourth,fifth] ?

Xs = [first,second,third,fourth] Ys = [fifth] ?

Xs = [first,second,third,fourth,fifth] Ys = [] ?

Outline

- Repetition
- Consulting vs. Querying
- Lists in Prolog
- Different views of a Prolog program
- Arithmetic in Prolog
- Cut and negation
- Problems with Prolog

Arithmetic in Prolog

- Prolog programs presented so far were *declarative*: they admitted a dual reading as a formula
  - Operations of arithmetic are functional, not relational
- Arithmetic compromises Prolog's declarativeness
  - Solved in constraint logic programming languages

Arithmetic operators

- Built-in data structures:
  - Integers: 1,2,3,... (+, -, *, //)
  - Floating points: 2.3, 3.4456, 5.4e-13,... (+, -, *, /)
- Infix vs prefix notation*
  - 45+35
  - '+'(45,35)
- It is possible to have user-defined operators with specified priority, associativity, etc

*We will see later how to evaluate expressions

Arithmetic comparison relations

- Prolog allows comparison of ground arithmetic expressions (*gae*, i.e. expressions without variables). *gae*s have *values*
- Built-in comparison relations: <, =<, =:= ("equal"), =\= ("different"), >= and >
- Queries
  - | ?- 6*3 =:= 9*2.
    yes
  - | ?- 8 > 5+3.
    no
  - | ?- 34>=X+4.
    uncaught exception: error(instantiation_error,(>=)/2
- Note difference between
  - = (unifiability relation) 1+1=2 *gives* no, X = 1 *gives* X = 1
  - == (syntactic equality) 1+1 == 2 *gives* no , X == x *gives* no
  - \== (syntactic inequality) 1+1\==2 *gives* yes.
  - =:= (value equality) 1+1 =:= 2 *gives* yes
  - =\= (value inequality) 1+1 =\= 2 *gives* no

Example: ordered lists

```
ordered([]).
ordered([X]).
ordered([X,Y|Ys]) :- X =< Y, ordered([Y|Ys]).
```

- Queries
  - | ?- ordered([3,4,67,8]).
    no
  - | ?- ordered([3,4,67, 88]).
    yes
  - | ? - ordered([3,4,X,88]).
    {INSTANTIATION ERROR: 4=<_30 - arg 2}

Evaluation of arithmetic expressions

- We need to introduce a way to evaluate expressions
  - | ?- X=:=3+4. yields an error
  - | ?- X=3+4.
    X = 3+4
- Evaluation is done using "is"
  - | ?- X is 3+4.
    X = 7
  - "is" is a builtin predicate which has been defined as an operator for simpler syntax, we could also write:
    | ?- is(X,3+4).
    X = 7

Example: Factorial

```
factorial(0,1).
factorial(N,F) :-N>0, N1 is N-1,
                        factorial(N1,F1),
              F is N*F1.
```

- Queries
  - | ?- factorial(5,X).
    X = 120
    Yes

The following query gives an error however:

- | ?- factorial(X,120). "X>0" is not allowed!
  uncaught exception: error(instantiation_error,(>)/2)

Example: Length of lists

- An intuitive definition **but wrong**
  length([],0).
  length([_ | Ts], N+1) :- length(Ts,N).
- Query
  - | ?- length([3,5,56,7],X).
    X = 0+1+1+1+1
    Yes
- What's the problem?

Expressions are not automatically evaluated in Prolog!

INF 3110/4110 – 2009

Example: Length of lists

- A good definition
  length([],0).
  length([_ | Ts], N) :- length(Ts,M), N is M+1.
- Queries
  - | ?- length([3,5,56,7],X).
    X = 4
    Yes
  - | ?- length(X,5).
    X = [_,_,_,_,_]
    yes

INF 3110/4110 – 2009

length(X,5)

length([],0).
length([_ | Ts], N) :- length(Ts,M), N is M+1.

:- length(X,5)
:- length(Ts,M), 5 is M+1
1. :- 5 is 0+1   Ts/[], M/0   FAIL
2. :- length(Ts1,M1), M is M1+1, 5 is M+1   Ts/[_,Ts1]
2.1 :- M is 0+1, 5 is M+1   Ts1/[], M1/0, Ts/[_,Ts1]
2.1 :- 5 is 1+1   Ts1/[], M1/0, Ts/[_,Ts1], M/1   FAIL
2.2 :- length(Ts2,M2), M1 is M2+1, M is M1+1, 5 is M+1
Ts1/[], M1/0, Ts/[_,Ts1], Ts1/[_,Ts2]
...

INF 3110/4110 – 2009

Outline

- Repetition
- Consulting vs. Querying
- Lists in Prolog
- Different views of a Prolog program
- Arithmetic in Prolog
- Cut and negation
- Problems with Prolog

INF 3110/4110 – 2009

Cut - !

- *Cut* is a built in system predicate which affects the procedural behaviour of a program
- Its main function is to reduce the search space of Prolog computations by dynamically prunig the search tree
- Example:
  p($\mathbf{s}_1$) :- A_1
  …
  p($\mathbf{s}_i$) :- B, !, C
  …
  p($\mathbf{s}_k$) :- A_k
- We compute p(**t**) using the i-th clause, B succeeds, and ! is encountered:
  - All alternative ways of computing B are discarded
  - All computations of p(**t**) using the i-th to k-th clauses are discarded as backtrackable alternatives
- Cut gives more control to the programmer, but compromises the declarative reading of the Prolog programs and makes it difficult to see what will happen in the computation.

rsiblings example

- Recall the rsiblings rule.
```
rsiblings(X,Y) :-   child(X,Parent1),
                    child(Y,Parent1),
                    X \== Y,
                    child(X,Parent2),
                    child(Y,Parent2),
                    Parent1 \== Parent2.
```
- | ?- rsiblings(anne,X).
- X = paul ? ;
- X = paul ? ;
- no

rsiblings with cut

- With cut
```
rsiblings(X,Y) :- child(X,Parent1),
                  !,
                  child(Y,Parent1),
                  X \== Y,
                  child(X,Parent2),
                  child(Y,Parent2),
                  Parent1 \== Parent2.
```
```
| ?- rsiblings(anne,X).

X = paul ? ;

no
| ?- rsiblings(X,anne).

no
```


rsiblings with cut, next try…

```
rsiblings(X,Y) :- child(X,Parent1),
                  child(Y,Parent1),
                  X \== Y,
                  !,
                  child(X,Parent2),
                  child(Y,Parent2),
                  Parent1 \== Parent2.
| ?- rsiblings(anne,X).

X = paul

yes
| ?- rsiblings(X,anne).

X = paul

yes
```

But what if anne has more than one sibling?

Cut destroys declarativity

Cut makes it possible to control program execution
-> Added efficiency.

On the other hand:

- Programs become hard to understand.
- Need to document in which ways predicates can be called.
- Compromises the original intension of the language.

INF 3110/4110 – 2009

Negation as failure

- Negation can be defined by cut.

```
not(X) :- X, ! , fail .
not(_) .
```

- The built-in negation operator is \+

```
| ?- \+ person(haakon,sonja,harald,1973) .
yes
```

- The query \+ A succeeds if and only if the query A fails.
- Corresponds to our "normal" notion of negation if the negated query always terminates and is ground. Consider negation of non-ground term X=1:

```
\+ (X=1)
no
```

INF 3110/4110 – 2009

IO in Prolog

- Various predicates for input/output.
  - print(f(a)) prints out a term.
  - display('Hello World') prints a string.

```
print_list([]) :- print(nothing).
print_list([X]):- write('only '), print(X).
print_list([X|Ys]) :- print(X), print_list_help(Ys).

print_list_help([]).
print_list_help([X|Xs]) :- write(' and '),print(X),
    print_list_help(Xs).
```

INF 3110/4110 – 2009

Outline

- Repetition
- Consulting vs. Querying
- Lists in Prolog
- Different views of a Prolog program
- Arithmetic in Prolog
- Cut and negation
- Problems with Prolog

INF 3110/4110 – 2009

Problems with Prolog

- No types
- No (standardized) module system
- Non-declarative arithmetic
- Need to use cut
- Cut makes automated optimization hard
- IO disaggrees with backtracking

INF 3110/4110 – 2009

Problem with IO

- IO and backtracking breaks program semantics
- Example:

io_problem :- print(one), fail. io_problem :- print(two).	prints	onetwo
io_problem :- fail, print(one). io_problem :- print(two).	prints	two

- The programs are semantically identical...
  - and (,) is commutative
- ...and should produce the same output: two

INF 3110/4110 – 2009

Further reading

- Mitchell's book – Chapter 15
- See also the tutorial by J. Power: [removed?] http://www.cs.may.ie/~jpower/Courses/PROLOG/
- Learn Prolog Now! – www.learnprolognow.org
- Even further reading: Sterling and E. Shapiro: *The art of Prolog*, 1994. MIT Press Series.

INF 3110/4110 – 2009

Mitchell's chap 15 – an overview.

15.1 History of logic programming

15.2 Brief overview of the logic programming paradigm

15.3 Equations solved by unification of atomic actions.
The formal basis for unification and the unification algorithm.

15.4 Clauses as parts of procedure declarations – Deals with Clauses = Rules and Facts and how they are computed.
1 Simple Clauses - The point is to make a relationship between logic programming and imperative programming.
2 Computation process
3 Clauses

15.5 Prolog's approach to programming
More about how computations take place. Multiple uses of prolog programs (testing vs. computing). Several examples.

15.6 Arithmetic in prolog

15.7 Control, ambivalent syntax and meta-variables.

15.8 Assessement of prolog.

15. 9 Bibliography

15.10 Summary

INF 3110/4110 – 2009

Prolog

"There is no question that Prolog is essentially a theorem prover à la Robinson. Our contribution was to transform that theorem prover into a programming language"

Colmerauer & Roussel (1996)

INF 3110/4110 – 2009

Appendix

- Mercury

INF 3110/4110 – 2009

The Mercury Language

- Logic PL developed at Univ. Of Melbourne
- First release 1995
- Compiled
- Strict type (and `mode') system
- Module system
- No cut
- Clean integration of IO
- Includes functional features
- A `pure' language

INF 3110/4110 – 2009

The Module System

- hello.m

```
:- module hello.

:- interface.
:- import_module io.
:- pred main(io::di, io::uo) is det.

:- implementation.
main(IOState_in, IOState_out) :-
io.write_string("Hello World\n", IOState_in, IOState_out).
```

INF 3110/4110 – 2009

The Module system (cont.)

- Can have private/public predicates, types
- Can compile modules separately
- Can refer to names with module prefix:
  io.write_string
  is predicate write_string in module io

INF 3110/4110 – 2009

The Type system

- Type system similar to ML
- Built-in types int, float, string, etc
- User-defined types
  - :- type weekday ---> mon;tue;wed;thu;fri;sat;sun.
  - :- type intOrString ---> anInt(int);aString(string).
- Parameterized (polymorphic) types
  - [1,2,3] is of type list(int)
  - {"a",12} is of type {string,int}
  - :- type maybe(T) ---> nothing ; just(T).
- Function types (Lambda terms)

INF 3110/4110 – 2009

The Type system (cont)

Need to declare types of predicates:

- :- pred append(list(T), list(T), list(T)).
- :- pred length(list(T), int).

Compiler checks that predicates are used with correct types.

INF 3110/4110 – 2009

The mode system

In Prolog, predicates can be used in different ways.

- :- append([1,2],[3,4,5],Zs).
- :- append(Xs,Ys,[1,2,3,4,5]).

In Mercury, declare this with modes:

- :- mode append(in,in,out) is det.
- :- mode append(out,out,in) is multi.

→ Predicates can be declared with multiple modes.

INF 3110/4110 – 2009

The mode system (cont)

Predicate has only one mode: Shorthand

- :- pred append(list(T)::in,list(T)::in,list(T)::out).
- :- pred length(list(T)::in,int::out).

Compiler checks that predicates are only used according to declared modes.

Implementation can be shared among modes or not.

INF 3110/4110 – 2009

IO

- hello.m

```
:- module hello.

:- interface.
:- import_module io.
:- pred main(io::di, io::uo) is det.

:- implementation.
main(IOState_in, IOState_out) :-
    io.write_string("Hello World\n", IOState_in, IOState_out).
```

INF 3110/4110 – 2009

IO (cont.)

Special modes for IO:

- di: destructive input:
  - Destroys input
  - The reference is therefore worthless afterwards.
- uo: unique output:
  - Guarantee: only this reference to the output.
  - Therefore be used for destructive input.

INF 3110/4110 – 2009

IO (cont.)

To do more output:

```
:- pred io.write_string(string::in, io::di, io::uo) is det.
:- pred io.write_int(int::in, io::di, io::uo) is det.
:- pred io.nl(io::di, io::uo) is det.

main(IO0, IO3) :-
    io.write_string("The meaning of life is ", IO0, IO1),
    io.write_int(42, IO1, IO2),
    io.nl(IO2, IO3).
```

INF 3110/4110 – 2009

Functions

If a function is a function, why encode it as predicate?

:- pred fib(int::in, int::out) is det.

```
fib(N, X) :-
    ( if N =< 2
      then X = 1
       else fib(N-1, A), fib(N-2, B), X = A + B
    ).
```

INF 3110/4110 – 2009

fib as a function

:- func fib(int) = int.

```
fib(N) = X :-
    ( if N =< 2
        then X = 1
        else X = fib(N-1) + fib(N-2)
    ).
```

or:

```
fib(N) = ( if N =< 2 then 1 else fib(N-1) + fib(N-2) ).
```

INF 3110/4110 – 2009

A note on functions

- In Prolog, 1+1 is just a term. To evaluate, use X is 1+1
- In Mercury 1+1 is evaluated, since + is declared as a function
- Programming with terms still possible
- Not all symbols are equal.
  - Possible cause for confusion
  - Usually easier to use

INF 3110/4110 – 2009

What about the cut?

Why use cut?

- For if-then-else;
  p(X) :- c(X), !, if-part(X).
  p(X) :- else-part(X).
- In Mercury, use if then else construct:
  p(X) :- **if** c(X) **then** if-part(X) **else** else-part(X).
- To reduce search space
- In Mercury, use modes and determinism

INF 3110/4110 – 2009

Mercury, Conclusion

- Mercury is a modern language, incorporating many ideas of PL design that did not exist when Prolog was invented.
- Has many aspects and details that make it harder to learn. (types, modes, determinism, terms vs. functions, higher order, modules, etc.)
- Has a cleaner, more `logical' semantics than Prolog.

More Logic PLs

- Higher-order logic programming, Lambda-Prolog
  - Like Prolog, but lambda terms instead of first order
  - Higher-order unification
  - *Not* a functional language!
- Constraint Logic Programming languages
  - Prolog just gathers instantiations for variables.
  - Instead, gather constraints that need to be satisfied.

    E.g. X > 3, X < 6, X \== 5

    System infers instantiation X=4