

# Logic Programming

Georg Moser

Department of Computer Science @ UIBK

Winter 2016



# Summary of Last Lecture

## Definitions

- a Prolog clause is called **iterative** if
  - 1 it has one recursive call, and
  - 2 zero or more calls to system predicates, **before** the recursive call
- a Prolog procedure is **iterative** if it contains only facts and iterative clauses

## Observation

- iterative programs are tail recursive
- sometimes tail recursion in general can be implemented as iteration which doesn't require a stack

## Example

```
built_in(+,2).  
user_def(fib,1).  
:- eval(fib(13),N), N=233.
```

# Outline of the Lecture

## Monotone Logic Programs

introduction, basic constructs, logic foundations, unification, semantics, database and recursive programming, termination, complexity

## Incomplete Data Structures and Constraints

incomplete data structures, definite clause grammars, constraint logic programming, answer set programming

## Full Prolog

semantics (revisted), cuts, correctness proofs, meta-logical predicates, nondeterministic programming, pragmatics, efficient programs, meta programming

# Outline of the Lecture

## Monotone Logic Programs

introduction, basic constructs, logic foundations, unification, semantics, database and recursive programming, termination, complexity

## Incomplete Data Structures and Constraints

incomplete data structures, definite clause grammars, constraint logic programming, answer set programming

## Full Prolog

semantics (revisted), cuts, correctness proofs, meta-logical predicates, nondeterministic programming, pragmatics, efficient programs, **meta programming**

# Meta-Programming and Meta-Interpreters

## Definition

- a **meta-program** treats other programs as data; it analyses, transforms, and simulates other programs

# Meta-Programming and Meta-Interpreters

## Definition

- a **meta-program** treats other programs as data; it analyses, transforms, and simulates other programs
- a **meta-interpreter** for a language is an interpreter for the language written in the language itself

# Meta-Programming and Meta-Interpreters

## Definition

- a **meta-program** treats other programs as data; it analyses, transforms, and simulates other programs
- a **meta-interpreter** for a language is an interpreter for the language written in the language itself
- for example, relation **solve**(**Goal**) is true, if *Goal* is true with respect to the program interpreted

# Meta-Programming and Meta-Interpreters

## Definition

- a **meta-program** treats other programs as data; it analyses, transforms, and simulates other programs
- a **meta-interpreter** for a language is an interpreter for the language written in the language itself
- for example, relation **solve**(Goal) is true, if *Goal* is true with respect to the program interpreted

## Example (simple meta-interpreter)

```
solve(true).
solve((A,B)) :- solve(A), solve(B).
solve(A) :- clause(A,B), solve(B).
```



# Meta-Program We Have Already Seen

## Example

```
accept(S) :-  
    initial(Q),  
    accept(Q,S).  
  
accept(Q, [X|Xs]) :-  
    delta(Q,X,Q1),  
    accept(Q1,Xs).  
  
accept(Q, []) :-  
    final(Q).  
  
initial(q0).  
final(q2).  
  
delta(q0,0,q0).  
delta(q0,0,q1).  
delta(q0,1,q0).  
delta(q1,1,q2).
```

# Meta-Program We Have Already Seen

## Example

```
accept(S) :-  
    initial(Q),  
    accept(Q,S).  
  
accept(Q, [X|Xs]) :-  
    delta(Q,X,Q1),  
    accept(Q1,Xs).  
  
accept(Q, []) :-  
    final(Q).  
  
initial(q0).  
final(q2).  
  
delta(q0,0,q0).  
delta(q0,0,q1).  
delta(q0,1,q0).  
delta(q1,1,q2).
```

# (Sort of) Meta-Program We'll See Soon

## Example

```

prove( and(A,B), UnExp, Lits, FreeV, VarLim) :- !,
    prove(A, [B|UnExp], Lits, FreeV, VarLim).
prove( or(A,B), UnExp, Lits, FreeV, VarLim) :- !,
    prove(A, UnExp, Lits, FreeV, VarLim),
    prove(B, UnExp, Lits, FreeV, VarLim).
prove( all(X,Fml), UnExp, Lits, FreeV, VarLim) :- !,
    \+ length(FreeV, VarLim),
    copy_term((X,Fml,FreeV),(X1,Fml1,FreeV)),
    append(UnExp, [all(X,Fml)], UnExp1),
    prove(Fml1, UnExp1, Lits, [X1|FreeV], VarLim).
prove( Lit, _UnExp, [L|Lits], _FreeV, _VarLim) :- !,
    (Lit = neg Neg; neg Lit = Neg) ->
        (unify_with_occurs_check(Neg, L);
         prove(Lit, [], Lits, _FreeV0, _VarLim0)).
prove( Lit, [Next|UnExp], Lits, FreeV, VarLim) :- !,
    prove(Next, UnExp, [Lit|Lits], FreeV, VarLim).

```

## Example (meta-interpreter with proofs)

```
solve(true,true).  
solve((A,B),(ProofA,ProofB)) :-  
    solve(A,ProofA),  
    solve(B,ProofB).  
solve(A,(A :- Proof)) :-  
    clause(A,B),  
    solve(B,Proof).
```

## Example (meta-interpreter with proofs)

```

solve(true,true).
solve((A,B),(ProofA,ProofB)) :-
    solve(A,ProofA),
    solve(B,ProofB).
solve(A,(A :- Proof)) :-
    clause(A,B),
    solve(B,Proof).

```

## Example

```

father(andreas,boris).      female(doris).      male(andreas).
father(andreas,christian).  female(eva).      male(boris).
father(andreas,doris).     male(christian).
father(boris,eva).         mother(doris,franz). male(franz).
father(franz,georg).       mother(eva,georg). male(georg).
son(X,Y) :- father(Y,X), male(X).

```

## Example (A Meta-Interpreter with Proofs (cont'd))

```
:- solve(son(christian, andreas), Proof).
```

```
Proof  $\mapsto$  (son(christian, andreas) <--  
  (father(andreas, christian) <-- true,  
  male(christian) <-- true))
```

## Example (A Meta-Interpreter with Proofs (cont'd))

```
:- solve(son(christian, andreas), Proof).
```

```
Proof  $\mapsto$  (son(christian, andreas) <--  
  (father(andreas, christian) <-- true,  
  male(christian) <-- true))
```

## Example (Tracing Pure Prolog)

```
trace(Goal) :- trace(Goal, 0).
```

## Example (A Meta-Interpreter with Proofs (cont'd))

```
:- solve(son(christian,andreas),Proof).

Proof  $\mapsto$  (son(christian, andreas) <--
  (father(andreas, christian)<--true,
  male(christian)<--true))
```

## Example (Tracing Pure Prolog)

```
trace(Goal) :- trace(Goal,0).

trace(true,Depth).
trace((A,B),Depth) :-
  trace(A,Depth), trace(B,Depth).
trace(A,Depth) :-
  clause(A,B),
  display(A,Depth),
  Depth1 is Depth + 1,
  trace(B,Depth1).
```



## Example (A Meta-Interpreter with Proofs (cont'd))

```
:- solve(son(christian,andreas),Proof).

Proof  $\mapsto$  (son(christian, andreas) <--
  (father(andreas, christian)<--true,
  male(christian)<--true))
```

## Example (Tracing Pure Prolog)

```
trace(Goal) :- trace(Goal,0).

trace(true,Depth).
trace((A,B),Depth) :-
  trace(A,Depth), trace(B,Depth).
trace(A,Depth) :-
  clause(A,B),
  display(A,Depth),
  Depth1 is Depth + 1,
  trace(B,Depth1).

display(A,Depth) :- tab(Depth), write(A), nl.
```

# Example

<code>system(A is B).</code>	<code>system(read(X)).</code>	<code>system(integer(X)).</code>
<code>system(clause(A,B)).</code>	<code>system(A &lt; B).</code>	<code>system(A &gt;= B).</code>
<code>system(write(X)).</code>	<code>system(functor(T,F,N)).</code>	<code>system(system(X)).</code>

## Example

<code>system(A is B).</code>	<code>system(read(X)).</code>	<code>system(integer(X)).</code>
<code>system(clause(A,B)).</code>	<code>system(A &lt; B).</code>	<code>system(A &gt;= B).</code>
<code>system(write(X)).</code>	<code>system(functor(T,F,N)).</code>	<code>system(system(X)).</code>

## Example

```
trace(Goal) :- trace(Goal,0).
```

## Example

<code>system(A is B).</code>	<code>system(read(X)).</code>	<code>system(integer(X)).</code>
<code>system(clause(A,B)).</code>	<code>system(A &lt; B).</code>	<code>system(A &gt;= B).</code>
<code>system(write(X)).</code>	<code>system(functor(T,F,N)).</code>	<code>system(system(X)).</code>

## Example

```
trace(Goal) :- trace(Goal,0).  
trace(true,Depth) :- !.
```

## Example

<code>system(A is B).</code>	<code>system(read(X)).</code>	<code>system(integer(X)).</code>
<code>system(clause(A,B)).</code>	<code>system(A &lt; B).</code>	<code>system(A &gt;= B).</code>
<code>system(write(X)).</code>	<code>system(functor(T,F,N)).</code>	<code>system(system(X)).</code>

## Example

```
trace(Goal) :- trace(Goal,0).  
trace(true,Depth) :- !.  
trace((A,B),Depth) :- !, trace(A,Depth), trace(B,Depth).
```

## Example

<code>system(A is B).</code>	<code>system(read(X)).</code>	<code>system(integer(X)).</code>
<code>system(clause(A,B)).</code>	<code>system(A &lt; B).</code>	<code>system(A &gt;= B).</code>
<code>system(write(X)).</code>	<code>system(functor(T,F,N)).</code>	<code>system(system(X)).</code>

## Example

```

trace(Goal) :- trace(Goal,0).

trace(true,Depth) :- !.
trace((A,B),Depth) :- !, trace(A,Depth), trace(B,Depth).
trace(A,Depth) :- system(A), A, !, display2(A,Depth), nl.
    
```

## Example

<code>system(A is B).</code>	<code>system(read(X)).</code>	<code>system(integer(X)).</code>
<code>system(clause(A,B)).</code>	<code>system(A &lt; B).</code>	<code>system(A &gt;= B).</code>
<code>system(write(X)).</code>	<code>system(functor(T,F,N)).</code>	<code>system(system(X)).</code>

## Example

```

trace(Goal) :- trace(Goal,0).

trace(true,Depth) :- !.
trace((A,B),Depth) :- !, trace(A,Depth), trace(B,Depth).
trace(A,Depth) :- system(A), A, !, display2(A,Depth), nl.
trace(A,Depth) :-
    clause(A,B), display(A,Depth), nl,
    Depth1 is Depth + 1, trace(B,Depth1).
    
```

## Example

<code>system(A is B).</code>	<code>system(read(X)).</code>	<code>system(integer(X)).</code>
<code>system(clause(A,B)).</code>	<code>system(A &lt; B).</code>	<code>system(A &gt;= B).</code>
<code>system(write(X)).</code>	<code>system(functor(T,F,N)).</code>	<code>system(system(X)).</code>

## Example

```

trace(Goal) :- trace(Goal,0).

trace(true,Depth) :- !.
trace((A,B),Depth) :- !, trace(A,Depth), trace(B,Depth).
trace(A,Depth) :- system(A), A, !, display2(A,Depth), nl.
trace(A,Depth) :-
    clause(A,B), display(A,Depth), nl,
    Depth1 is Depth + 1, trace(B,Depth1).
trace(A,Depth) :-
    \+ clause(A,B), display(A,Depth),
    tab(8), write(f),nl,fail.

```



## Example

<code>system(A is B).</code>	<code>system(read(X)).</code>	<code>system(integer(X)).</code>
<code>system(clause(A,B)).</code>	<code>system(A &lt; B).</code>	<code>system(A &gt;= B).</code>
<code>system(write(X)).</code>	<code>system(functor(T,F,N)).</code>	<code>system(system(X)).</code>

## Example

```

trace(Goal) :- trace(Goal,0).

trace(true,Depth) :- !.
trace((A,B),Depth) :- !, trace(A,Depth), trace(B,Depth).
trace(A,Depth) :- system(A), A, !, display2(A,Depth), nl.
trace(A,Depth) :-
    clause(A,B), display(A,Depth), nl,
    Depth1 is Depth + 1, trace(B,Depth1).
trace(A,Depth) :-
    \+ clause(A,B), display(A,Depth),
    tab(8), write(f),nl,fail.

display(A,Depth) :- Spacing is 3*Depth, tab(Spacing), write(A).
```

# Meta-Interpreters for Debugging

## Example (Control Execution)

```

solve(true, _D, no_overflow) :-
    !.
solve(_A, 0, overflow([])).
solve((A,B), D, Overflow) :-
    D > 0,
    solve(A, D, OverflowA),
    solve_conjunction(OverflowA, B, D, Overflow).
solve(A, D, no_overflow) :-
    D > 0,
    system(A), !, A.
solve(A, D, Overflow) :-
    D > 0,
    clause(A, B),
    D1 is D - 1,
    solve(B, D1, OverflowB),
    return_overflow(OverflowB, A, Overflow).
    
```

## Example (Control Execution (cont'd))

```
solve_conjunction(overflow(S),_B,_D,overflow(S)).
solve_conjunction(no_overflow,B,D,Overflow) :-
    solve(B,D,Overflow).
```

```
return_overflow(no_overflow,_A,no_overflow).
return_overflow(overflow(S),A,overflow([A|S])).
```

```
% isort(Xs,Ys) <— Ys is sorted Xs, using insertion sort
```

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

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

```
my_insert(X,[Y|Ys],[Y|Zs]) :-
    X >= Y,
    my_insert(X,[Y|Ys],Zs).
```

```
my_insert(X,[],[X]).
```

# Expert Systems in Prolog

## Expert Systems

expert systems typically consists of

- knowledge base
- inference engine

this separation is not suitable for a Prolog implementation

# Expert Systems in Prolog

## Expert Systems

expert systems typically consists of

- knowledge base
- inference engine

this separation is not suitable for a Prolog implementation

## Employ Meta-Interpreters

we implement the following features of expert systems using meta-interpreters:

- interaction with the user
- explanation facility
- uncertainty reasoning

## Toy Expert System

```
place_in_oven(Dish, top) :-  
    pastry(Dish), size(Dish, small).  
place_in_oven(Dish, middle) :-  
    pastry(Dish), size(Dish, big).  
place_in_oven(Dish, middle) :-  
    main_meal(Dish).  
place_in_oven(Dish, low) :-  
    slow_cooker(Dish).  
  
pastry(Dish) :- type(Dish, cake).  
pastry(Dish) :- type(Dish, bread).  
  
main_meal(Dish) :- type(Dish, meat).  
  
slow_cooker(Dish) :- type(Dish, milk_pudding).
```

## solve1/1

```

solve1(true) :-
    !.
solve1((A,B)) :-
    solve1(A), solve1(B).
solve1(A) :-
    A \= (_A1, _A2),
    clause(A,B), solve1(B).
solve1(A) :-
    askable(A), \+ known(A),
    ask(A, Answer),
    respond(Answer, A).

ask(A, Answer) :- display_query(A), read(Answer).

askable(type(_Dish, _Type)).
askable(size(_Dish, _Size)).

respond(yes, A) :- assert(A).
respond(no, A) :- assert(untrue(A)), fail.

```

## Interaction (in the Naive)

```
interact (Goal) :-  
    reset , solve1 (Goal).  
  
reset :- retractall (type ( _Dish , _Type ) ) ,  
    retractall (size ( _Sish , _Size ) ) ,  
    retractall (untrue ( _Fact ) ) .  
  
?- interact (place_in_oven (dish , X)) .  
type (dish , cake)? yes .  
size (dish , small)? no .  
type (dish , bread)? no .  
size (dish , big)? yes .  
X = middle
```



## Interaction (in the Naive)

```

interact(Goal) :-
    reset, solve1(Goal).

reset :- retractall(type(_Dish, _Type)),
         retractall(size(_Sish, _Size)),
         retractall(untrue(_Fact)).

?- interact(place_in_oven(dish, X)).
type(dish, cake)? yes.
size(dish, small)? no.
type(dish, bread)? no.
size(dish, big)? yes.
X = middle

```

### Question

what about explanations for questions?

## solve2/1

```

solve2(Goal) :- solve2(Goal, []).

solve2(true, _Rules) :-
    !.
solve2((A,B), Rules) :-
    solve2(A, Rules), solve2(B, Rules).
solve2(A, Rules) :-
    A \= (_A1, _A2),
    clause(A,B),
    solve2(B, [rule(A,B) | Rules]).
solve2(A, Rules) :-
    askable(A), \+ known(A),
    ask(A, Answer), respond(Answer, A, Rules).

respond(why, A, [Rule | Rules]) :-
    display_rule(Rule),
    ask(A, Answer),
    respond(Answer, A, Rules).

```

## Interaction with Explanations

```
interact_why(Goal) :- reset , solve2(Goal).
```

## Interaction with Explanations

```
interact_why(Goal) :- reset , solve2(Goal).
```

```
?- interact_why(place_in_oven(dish,X)).  
type(dish,cake)? yes.  
size(dish,small)? no.  
type(dish,bread)? no.  
size(dish,big)? why.  
if pastry(dish) and size(dish,big)  
then place_in_oven(dish,middle)  
size(dish,big)? yes.  
X = middle
```

## Interaction with Explanations

```
interact_why(Goal) :- reset , solve2(Goal).
```

```
?- interact_why(place_in_oven(dish,X)).  
type(dish,cake)? yes.  
size(dish,small)? no.  
type(dish,bread)? no.  
size(dish,big)? why.  
if pastry(dish) and size(dish,big)  
then place_in_oven(dish,middle)  
size(dish,big)? yes.  
X = middle
```

### Question

how to obtain general explanations

## interpret/1

```

interpret((Proof1, Proof2)) :-
    interpret(Proof1), interpret(Proof2).
interpret(Proof) :-
    fact(Proof, Fact),
    nl, write(Fact),
    writeln(' is a fact in the database ').
interpret(Proof) :-
    rule(Proof, Head, Body, Proof1),
    nl, write(Head),
    writeln(' is proved using the rule '),
    display_rule(rule(Head, Body)),
    interpret(Proof1).

extract_body((Proof1, Proof2), (Body1, Body2)) :-
    !, extract_body(Proof1, Body1),
    extract_body(Proof2, Body2).
extract_body((Goal <— _Proof), Goal).

```

## how/1

```
how(Goal) :- solve(Goal,Proof), interpret(Proof).
```

```
?- interact(place_in_oven(dish,X)).  
% required for type and size of dish
```

```
?- how(place_in_oven(dish,top)).
```

```
place_in_oven(dish,top) is proved using the rule  
if pastry(dish) and size(dish,small)  
then place_in_oven(dish,top)
```

```
pastry(dish) is proved using the rule  
if type(dish,bread)  
then pastry(dish)
```

```
type(dish,bread) is a fact in the database
```

```
size(dish,small) is a fact in the database
```

## Shortcomings with Explanation

- the explanation is exhaustive
- Prolog computation is mirrored



## Shortcomings with Explanation

- the explanation is exhaustive  
not intelligible for a knowledge base with 100 rules
- Prolog computation is mirrored

## Shortcomings with Explanation

- the explanation is exhaustive  
not intelligible for a knowledge base with 100 rules
- restrict explanation to one level:  
`pastry(dish) can be further explained`
- Prolog computation is mirrored

## Shortcomings with Explanation

- the explanation is exhaustive  
not intelligible for a knowledge base with 100 rules
- restrict explanation to one level:

`pastry(dish) can be further explained`

- Prolog computation is mirrored
- take expert knowledge into account:

```
interpret((Goal <— Proof)) :-  
    classification(Goal),  
    write(Goal),  
    writeln(' is a classification example').
```

## Shortcomings with Explanation

- the explanation is exhaustive  
not intelligible for a knowledge base with 100 rules
- restrict explanation to one level:

```
pastry(dish) can be further explained
```

- Prolog computation is mirrored
- take expert knowledge into account:

```
interpret((Goal <— Proof)) :-  
    classification(Goal),  
    write(Goal),  
    writeln(' is a classification example').
```

- in general make use of filtered explanations

## Shortcomings with Explanation

- the explanation is exhaustive  
not intelligible for a knowledge base with 100 rules
- restrict explanation to one level:  
`pastry(dish) can be further explained`

- Prolog computation is mirrored
- take expert knowledge into account:

```
interpret((Goal <— Proof)) :-
    classification(Goal),
    write(Goal),
    writeln(' is a classification example').
```

- in general make use of filtered explanations

## Exercise

Modify the implementation of `how/1` such that the partial answer proposed is generated

## Definition

the **certainty** of a goal is computed as follows

$$\text{cert}(G) = \begin{cases} \min\{\text{cert}(A), \text{cert}(B)\} & G = (A, B) \\ \max\{\text{cert}(B) \cdot \textit{Factor} \mid \text{exists } \langle A : -B, \textit{Factor} \rangle\} & G = A \end{cases}$$

## Definition

the **certainty** of a goal is computed as follows

$$\text{cert}(G) = \begin{cases} \min\{\text{cert}(A), \text{cert}(B)\} & G = (A, B) \\ \max\{\text{cert}(B) \cdot \textit{Factor} \mid \text{exists } \langle A : -B, \textit{Factor} \rangle\} & G = A \end{cases}$$

## Definition (clauses with certification factor)

```

clause_cf(place_in_oven(Dish, top),
           (pastry(Dish), size(Dish, small)), 0.7).
clause_cf(place_in_oven(Dish, middle),
           (pastry(Dish), size(Dish, big)), 1).
clause_cf(place_in_oven(Dish, middle),
           main_meal(Dish), 1).
clause_cf(place_in_oven(Dish, low),
           slow_cooker(Dish), 0.5).

% otherwise
clause_cf(Head, Body, 1) :- clause(Head, Body).

```

## solve3/1

```

solve3(true,1) :-
    !
solve3((A,B),C) :-
    !,
    solve3(A,C1),
    solve3(B,C2),
    minimum(C1,C2,C).
solve3(A,C) :-
    clause_cf(A,B,C1),
    solve3(B,C2),
    C is C1 * C2.

?- interact(place_in_oven(dish,X)).
% required for type and size of dish

?- solve3(place_in_oven(dish,top),C).
C = 0.7

```



Thank You for Your Attention!