Logik für Informatiker: PROLOG
Part 5: Extensions to Prolog Kernel

Andreas Karwath

&

Wolfram Burgard

(original slides by Peter Flach)

- up to now:
  „*pure Prolog*“, i.e. the problem to be solved was sufficiently specified by Horn clauses and solved by a resolution theorem prover

- now: extensions to Horn logic, loss of the purely declarative problem description
  (Horn clause: clause with at most one positive literal;
  definite clause: clause with *exactly* one positive literal)

- integration of „built-in“ predicates
  sometimes global side effects

- goal: as much declarative problem specification as possible; procedural aspects only if they cannot be avoided

Extensions of the Prolog kernel

- *built-in* predicates for input and output work as follows:

  - *input predicate*: proof is interrupted and user is asked for input, input is unified with the argument of the input, proof is continued

  - *output predicate*: argument shown on display, proof is continued

- No *backtracking*!
  => take into account procedural behavior

- *built-in* predicates for reading resp. writing Prolog terms:

```
read/1
write/1
```

- *built-in* predicates for reading resp. writing characters:

```
get0/1
put/1
nl/0
tab/1
```


Input/output

- *true*: always successful

  equivalent:

```
fact(a,b,c).
fact(a,b,c) :- true.
```

- *fail*: always fails

  Use of *fail*: compute all solutions for a given goal:

```
?- append(X,Y,[a,b,c]), write(X),tab(1), write(Y),
nl, fail.

[][a,b,c]
[a] [b,c]
[a,b] [c]
[a,b,c] []
no
```


Success und failure

Arithmetic expressions

- For computing numeric values without unification, Prolog provides arithmetic operators:

  `+, -, *, /, //, mod, /\, \/, ...`

  and the evaluation operator `is`.

- `X is Y` is true if `Y` is an arithmetic expression, where all variables are instantiated to numbers and `X` is unified with the result of evaluating `Y`.

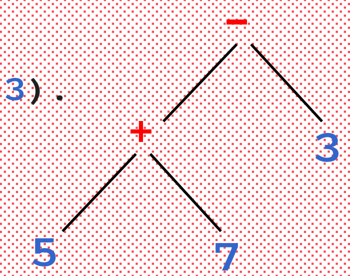
- `X < Y` holds if the evaluation of `X` gives a value smaller than the evaluation of `Y`.

- Analogously, `X =< Y`, `X > Y`, `X >= Y`, `X =:= Y`, `X =\= Y`

- The arguments have to be fully instantiated with arithmetic expressions (so that they can be evaluated).

```
?-X is 5+7-3.
  X = 9

?-9 is 5+7-3.
  Yes

?-9 is X+7-3.
  Error in arithmetic expression

?-X is 5*3+7/2.
  X = 18.5
```

```
?-display(5+7-3).
-(+(5,7),3)
```




```
?-X = 5+7-3.
  X = 5+7-3

?-9 = 5+7-3.
  No

?-9 = X+7-3.
  No

?-X = Y+7-3.
  X = _947+7-3
  Y = _947
```


Prolog arithmetic vs.unification


```
/*
   factorial(N,F) :- F is the integer N
   factorial.

*/

factorial(0,1).

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

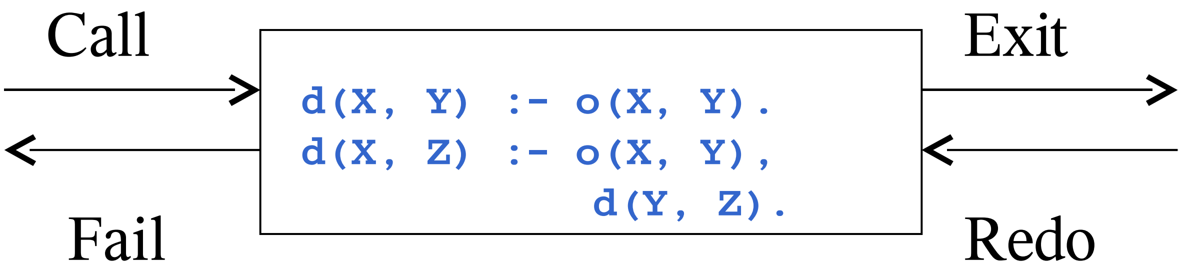

Example: arithmetic expressions

```
/*

   area(Chain,Area) :-
   Area is the area of the polygon enclosed by the
   list of points Chain, where the coordinates of each
   point are represented by a pair (X,Y) of integers.

*/

area([Tuple],0).

area([(X1,Y1),(X2,Y2)|XYs],Area) :-area([(X2,Y2)|
  XYs],Area1),
  Area is (X1*Y2-Y1*X2)/2 + Area1.
```


Example: arithmetic expressions

Control Flow and Debugging

- Model of control flow during the execution of a Prolog program: box model
- All clauses of a predicate with the same name and arity are considered as one procedure
- The *debugger* provides information at the beginning and the end of a procedure call.



Control flow and debugging

- Box model accounts for non-determinism of Prolog in that a procedure can return multiple solutions by *backtracking*
- If procedure is called:
  - control goes via *Call* entrance to procedure
  - if proof is found: control via *Exit* to calling context
  - in case of back-tracking, the control goes back to the procedure via *Redo*
  - if no proof is found: exit via *Fail*
- Such boxes can be nested
  Each new (e.g., recursive) call „creates“ a new box

Control flow and debugging

```
?-trace.
yes.
?-  student_of(S,peter).
    1  1  Call: student_of(_65,peter) ?
    2  2  Call: follows(_65,_343) ?
    2  2  Exit: follows(paul,computer_science) ?
    3  2  Call: teaches(peter,computer_science) ?
    3  2  Exit: teaches(peter,computer_science) ?
    1  1  Exit: student_of(paul,peter) ?
S = paul ?
;
```

Unique invocation identifier, recursion depth

The procedure box control flow model: example


```
  1  1  Redo: student_of(paul,peter) ?
  3  2  Redo: teaches(peter,computer_science) ?
  3  2  Fail: teaches(peter,computer_science) ?
  2  2  Redo: follows(paul,computer_science) ?
  2  2  Exit: follows(paul,expert_systems) ?
  3  2  Call: teaches(peter,expert_systems) ?
  3  2  Fail: teaches(peter,expert_systems) ?
  2  2  Redo: follows(paul,expert_systems) ?
  2  2  Exit: follows(maria,ai_techniques) ?
  3  2  Call: teaches(peter,ai_techniques) ?
  3  2  Exit: teaches(peter,ai_techniques) ?
  1  1  Exit: student_of(maria,peter) ?
S = maria ?
;
```

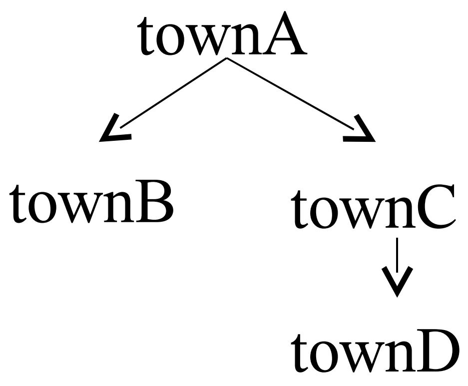

The procedure box control flow model: example


```
    1   1   Redo: student_of(maria,peter) ?
    3   2   Redo: teaches(peter,ai_techniques) ?
    3   2   Fail: teaches(peter,ai_techniques) ?
    2   2   Redo: follows(maria,ai_techniques) ?
    2   2   Fail: follows(_65,_343) ?
    1   1   Fail: student_of(_65,peter) ?
no
```


The procedure box control flow model: example

Program:

```
route(townA, townB).
route(townA, townC).
route(townC, townD).

travel(Destination, Destination).
travel(Source, Destination) :-
     route(Source, NextTown),
     travel(NextTown, Destination).
```



Exercise: control flow and debugging

`debug`
starts the *debugger*. At the next *spy-point* the user is asked for a command.

`trace`
starts the *debugger*. At the next procedure box the user is asked for a command.

`leash(+Mode)`
sets the *leashing mode* to *Mode*. The initial value is [*call,exit,redo,fail,exception*] (*full leashing*).

`nodebug, notrace`
stops the debugger.

`debugging`
outputs the current state of the *debugger*.

Predicates for debugging

Spy-Points allow to stop the program, whenever a specified predicate is used. Possible to set new spy points during debugging.

`spy Spec`

sets spy points for predicates using a generalized predicate spec *Spec*

Example:

```
| ?- spy my_predicate.
| ?- spy [my_first_predicate, my_second_predicate].
| ?- spy [user:p, m:q/(2-3)].
| ?- spy m:[p/1, q/1].
```


Spy points

`nospy Spec`

removes certain *spy points*

`nospyall`

removes all *spy points*

`spypoint_condition(:Goal,?Port,+Test)`

sets a conditional *spy point* for the predicate of *Goal*

Spy points