

Prolog as a Meta-Language

Guy Wiener

Sayeret Lambda

July 14, 2010

Disclaimer

Prolog is:

- Well-unknown
- Domain-specific
- Inefficient

Outline

- 1 Introduction to Prolog
- 2 Functional Prolog
- 3 Prolog as an Interpreter
- 4 Prolog for Software Modeling
- 5 Prolog Libs and Tricks
- 6 Summary

Outline

- 1 Introduction to Prolog
- 2 Functional Prolog
- 3 Prolog as an Interpreter
- 4 Prolog for Software Modeling
- 5 Prolog Libs and Tricks
- 6 Summary

Outline

1 Introduction to Prolog

- Terms
- Unification
- Queries
- Rules
- Cutting
- Term Expansion
- Summary

Grounded Atomic Terms

Numbers 1, 2, 3.14...

Atoms foo, bar, 'Baz'

Characters \$a, \$b, \$c

Grounded Composite Terms

Functors `func(a, b, 5)`

Grounded Composite Terms

Functors `func(a, b, 5)`

Lists `[a, b, 5], [1, 2 | ...]`

Strings `"some text"`

Operators `3 + 4, foo @> goo`

Ungrounded Terms

Variables X, Y, Foo, G00, _, _A

Composites func(X, Y, 5), [1, 2 | Rest]

Outline

1 Introduction to Prolog

- Terms
- **Unification**
- Queries
- Rules
- Cutting
- Term Expansion
- Summary

Unifying Grounded Terms

- $1 = 1$, $\text{foo} = \text{foo}$
- $1 \neq 2$, $\text{foo} \neq \text{goo}$
- $\text{func}(a, b, 5) = \text{func}(a, b, 5)$
- $\text{func}(a, b, 5) \neq \text{func}(a, c, 5)$

Unifying Ungrounded Terms

- $X = 1, Y = \text{foo}(\text{bar}(5))$
- $\text{foo}(X, \text{goo}(4)) = \text{foo}(a, Y) \Rightarrow$
 $X = a, Y = \text{goo}(4)$
- $[1, 2, 3] = [X \mid \text{Rest}] \Rightarrow$
 $X = 1, \text{Rest} = [2, 3]$
- $\text{foo}(1, 2) \neq \text{foo}(X, X)$

The Reflection Operator

- $\text{func}(a, b, c) =.. [\text{func}, a, b, c]$
- $\text{func}(a, b, c) =.. [F \mid \text{Args}] \Rightarrow$
 $F = \text{func}, \text{Args} = [a, b, c]$
- $F =.. [\text{foo}, \text{bar}, X] \Rightarrow F = \text{foo}(\text{bar}, X)$

Arithmetic

- **X is** $2 + 3 \Rightarrow X = 5$
- **5 is** $X + 3 \Rightarrow$ **false**

Outline

1 Introduction to Prolog

- Terms
- Unification
- **Queries**
- Rules
- Cutting
- Term Expansion
- Summary

Programs and Queries

Program

```
extends(man, person).
extends(woman, person).
consists(couple, man).
consists(couple, woman).
```

Queries

<code>?- extends(man, X).</code>	<code>?- extends(X, person).</code>
<code>X = person</code>	<code>X = man ;</code>
	<code>X = woman</code>

And-Queries

Program

```
extends(man, person).  
extends(woman, person).  
consists(couple, man).  
consists(couple, woman).
```

Queries

```
?- consists(couple, X) , extends(X, Y).  
X = man, Y = person ;  
X = woman, Y = person
```

And-Queries

Program

```
extends(man, person).  
extends(woman, person).  
consists(couple, man).  
consists(couple, woman).
```

Queries

```
?- consists(couple, X), consists(couple, Y),  
   X \= Y.  
X = man, Y = woman
```

Or-Queries

Program

```
extends(man, person).  
extends(woman, person).  
consists(couple, man).  
consists(couple, woman).
```

Queries

```
?- extends(man, X) ; consists(X, man).  
X = person ;  
X = couple
```

Queries

Program

```
extends(man, person).  
extends(woman, person).  
consists(couple, man).  
consists(couple, woman).
```

Queries

```
?- (extends(man, X) ; consists(X, man)) ,  
   extends(woman, X).  
X = person
```

Operators are Functors Too

Program

```
:- op(xfx, 300, |>).
```

```
man |> person.
```

```
woman |> person.
```

Queries

```
?- X |> Y.
```

```
X = man, Y = person ;
```

```
X = woman, Y = person
```

Outline

1 Introduction to Prolog

- Terms
- Unification
- Queries
- **Rules**
- Cutting
- Term Expansion
- Summary

Rules

Program

```
% ...as before  
extends(boy, man).
```

```
subtype(X, Y) :- extends(X, Y).  
subtype(X, Y) :- extends(X, Z), subtype(Z, Y).
```

Queries

```
?- subtype(X, person).  
X = man ; X = woman ;  
X = boy
```

Rules

Program

```
extends(item, element).  
extends(bucket, element).  
consists(bucket, element).  
  
composite(Comp, Base) :-  
    subtype(Comp, Base),  
    consists(Comp, Base).
```

Query

```
?- composite(X, Y).  
X = bucket,  
Y = element.
```

Outline

1 Introduction to Prolog

- Terms
- Unification
- Queries
- Rules
- **Cutting**
- Term Expansion
- Summary

The Notorious Cut

The '!' operator

- Do not retry previous goals
- Do not try subsequent clauses

Example

```

type(X, v) :- var(X).           ?- type(a, T).
type(X, a) :- atom(X).         T = a ;
type(_, f).                     T = f

```

The Notorious Cut

The '!' operator

- Do not retry previous goals
- Do not try subsequent clauses

Example

```

type(X, v) :- var(X) !.      ?- type(a, T).
type(X, a) :- atom(X) !.    T = a ;
type(_, f).

```

Outline

1 Introduction to Prolog

- Terms
- Unification
- Queries
- Rules
- Cutting
- **Term Expansion**
- Summary

Term Expansion

Macro definition

```

term_expansion(recursive(R, P),
                [(Head :- CallP1),
                 (Head :- CallP2, CallR)]) :-
    Head      =.. [R, X, Y],
    CallP1    =.. [P, X, Y],
    CallP2    =.. [P, X, Z],
    CallR     =.. [R, Z, Y].

```

Term Expansion

Source

```
recursive(subtype,  
          extends).
```

Expanded

```
subtype(X, Y) :-  
  extends(X, Y).  
subtype(X, Y) :-  
  extends(X, Z),  
  subtype(Z, Y).
```

Outline

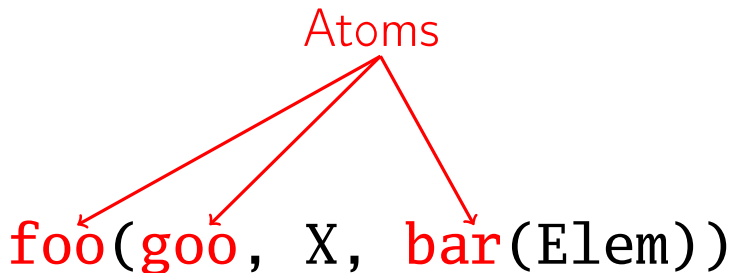
1 Introduction to Prolog

- Terms
- Unification
- Queries
- Rules
- Cutting
- Term Expansion
- **Summary**

Summary of Prolog Terminology

`foo(goo, X, bar(Elem))`

Summary of Prolog Terminology



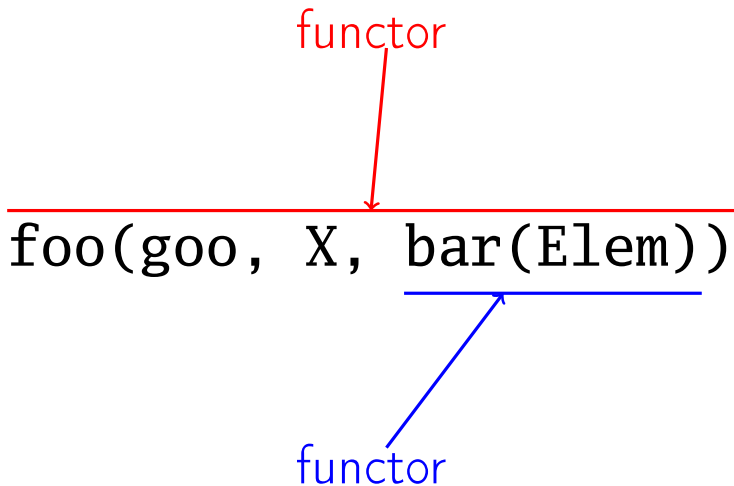
Summary of Prolog Terminology

foo(goo, X, bar(Elem))

Variables



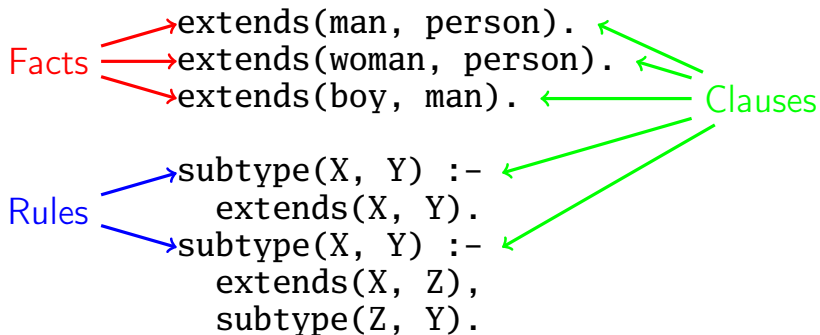
Summary of Prolog Terminology



Summary of Prolog Terminology

Head \longrightarrow subtype(X, Y) :- \longleftarrow Operator
 Body \longrightarrow extends(X, Z), \longleftarrow Goals
 subtype(Z, Y). \longleftarrow

Summary of Prolog Terminology



Summary of Prolog Terminology

extends/2 →


```

extends(man, person).
extends(woman, person).
extends(boy, man).
    
```

subtype/2 →


```

subtype(X, Y) :-
    extends(X, Y).
subtype(X, Y) :-
    extends(X, Z),
    subtype(Z, Y).
    
```

Predicates

Outline

- 1 Introduction to Prolog
- 2 Functional Prolog**
- 3 Prolog as an Interpreter
- 4 Prolog for Software Modeling
- 5 Prolog Libs and Tricks
- 6 Summary

Outline

- 2 Functional Prolog
 - Variables as Goals
 - Finding All Solutions
 - Failure as Negation

How Two Things are Related?

Program

```
related(R, X, Y) :-  
    P =.. [R, X, Y],    % P = R(X, Y)  
    P.                  % Call P
```

Query

```
?- related(extends, X, Y).  
X = man, Y = person ;  
X = woman, Y = person ;  
...
```

How Two Things are Related?

Program

```
related(R, X, Y) :-  
    call(R, X, Y).
```

Query

```
?- related(extends, X, Y).  
X = man, Y = person ;  
X = woman, Y = person ;  
...
```

How Two Things are Related?

Program

```
relation(extends).
relation(consists).
relation(subtype)

related(R, X, Y) :-
    relation(R),
    call(R, X, Y).
```

Query

```
?- related(R, boy, person).
R = subtype
```

How Two Things are Related?

Program

```
relation(extends).      related(R, X, Y) :-
relation(consists).    relation(R),
relation(subtype)      object(X),
object(person).        object(Y),
object(couple). % etc. call(R, X, Y).
```

Query

```
?- related(R, X, Y).
R = consists, X = couple, Y = man ;
...
```

Curried Call

Query

```
?- call(consists(couple), X).  
X = man ;  
X = woman
```

Curried Call

call appends the arguments to
the functor

Query

```
?- call(consists(couple), X).  
X = man ;  
X = woman
```

Outline

- 2 Functional Prolog
 - Variables as Goals
 - Finding All Solutions**
 - Failure as Negation

Finding All Solutions

Meta-Predicates

- `findall(Template, Goal, Bag)`
- `bagof(Template, Goal, Bag)`
- `setof(Template, Goal, Bag)`

`findall` example

```
?- findall(X, extends(X, person), Xs).  
Xs = [man, woman].
```

List Processing

List meta-predicates

- `maplist(Pred, List1, List2)` (in `lists` module)
- `include`, `exclude`, `partition` (in `apply` module)

`include` example

```
subtype_of(Y, X) :- subtype(X, Y).
```

```
?- include(subtype_of(person),  
          [item, man, boy, element], L).  
L = [man, boy].
```

Outline

- 2 Functional Prolog
 - Variables as Goals
 - Finding All Solutions
 - Failure as Negation

Failure as Negation

The 'fail' operator

$\backslash+$ R Succeed only if R fails

Program

```
unassignable(X, Y) :- \+ subtype_of(X, Y),
```

Query

```
?- unassignable(boy, bucket).  
true.  
?- unassignable(boy, person).  
false.
```

Outline

- 1 Introduction to Prolog
- 2 Functional Prolog
- 3 Prolog as an Interpreter**
- 4 Prolog for Software Modeling
- 5 Prolog Libs and Tricks
- 6 Summary

Outline

3 Prolog as an Interpreter

- Programs as Inputs
- The Prolog Meta-Interpreter
- Forward Chaining
- OOP
- Other Interpreters

Programs as Inputs

Interpreter

```
subtype(X, Y) :-  
    extends(X, Y).  
subtype(X, Y) :-  
    extends(X, Z),  
    subtype(Z, Y).  
  
composite(C, B) :-  
    subtype(C, B),  
    consists(C, B).
```

Input

```
extends(man, person).  
extends(woman, person).  
extends(boy, man).  
consists(couple, man).  
consists(couple, woman).
```

Queries

```
?- subtype(X, Y).  
X = man, Y = person
```

Programs as Inputs

Interpreter

```
subtype(X, Y) :-  
    extends(X, Y).  
subtype(X, Y) :-  
    extends(X, Z),  
    subtype(Z, Y).  
  
composite(C, B) :-  
    subtype(C, B),  
    consists(C, B).
```

Input

```
extends(item, elem).  
extends(bucket, elem).  
consists(bucket, elem).
```

Queries

```
?- subtype(X, Y).  
X = item, Y = elem
```

Programs as Inputs

Interpreter

```
:- op(xfx, 300, |>).
:- op(xfx, 300, <>).
```

```
subtype(X, Y) :-
  X |> Y.
```

```
subtype(X, Y) :-
  X |> Z, Z <> Y.
```

```
composite(C, B) :-
  subtype(C, B),
  C <> B.
```

Input

```
item |> elem.
bucket |> elem.
bucket <> elem.
```

Queries

```
?- subtype(X, Y).
X = item, Y = elem
```

Outline

- 3 Prolog as an Interpreter
 - Programs as Inputs
 - The Prolog Meta-Interpreter
 - Forward Chaining
 - OOP
 - Other Interpreters

The Prolog Meta-Interpreter

Meta-Interpreter

```
meta((A, B)) :- !, meta(A), meta(B).  
meta(A) :- system(A), !, A.  
meta(A) :- (A :- B), meta(B).  
system(true).
```

The Prolog Meta-Interpreter

Meta-Interpreter

```
meta((A, B)) :- !, meta(A), meta(B).  
meta(A) :- system(A), !, A.  
meta(A) :- clause(A, B), meta(B).  
system(true).
```

Modifying the Meta-Interpreter

Asking your opinion

```
meta((A, B)) :- !, meta(A), meta(B).
meta(A) :- system(A), !, A.
meta(A) :- clause(A, B), meta(B).
meta(A) :-
    \+ clause(A,_), ground(A),
    print(A), print('?'),
    read(yes).
```

Outline

- 3 Prolog as an Interpreter
 - Programs as Inputs
 - The Prolog Meta-Interpreter
 - **Forward Chaining**
 - OOP
 - Other Interpreters

Forward Chaining

Example program using a ':-' operator

```
p :- q.  
r, s :- p.  
q :- t.
```

Example run

```
?- fire(r), fire(s).  
true  
?- t.  
true.
```

Forward Chaining Interpreter

Fire

```
:- op(1150, xfx, -:).

fire(X) :- clause(X, true), !.
fire(X) :-
    assert(X),
    (If -: Then),
    amember(X, If), % member in a commas list
    \+ (amember(Y, If), \+ clause(Y, true)),
    fire(Then),
    fail.
fire(_).
```

Outline

- 3 Prolog as an Interpreter
 - Programs as Inputs
 - The Prolog Meta-Interpreter
 - Forward Chaining
 - **OOP**
 - Other Interpreters

Prologer's OOP

Objects

```
object(rect(W, H),
      [(area(A) :- A is W*H),
       (show :- print(W * H))]).
object(square(X), []).
isa(square(X), rect(X, X)).
```

Sending messages

```
?- Rec1 = rect(4,5), send(Rec1, area(A)).
A = 20
```

Prolog OOP Interpreter

```
send(Obj, Msg) :-  
    get_methods(Obj, Methods),  
    process(Msg, Methods).  
  
get_methods(Obj, Methods) :-  
    object(Obj, Methods).  
get_methods(Obj, Methods) :- % Inheritance  
    isa(Obj, Super),  
    get_methods(Super, Methods).  
  
process(Msg, [Msg | _]). % Fact  
process(Msg, [(Msg :- Body) | _]) :- % Rule  
    call(Body).  
process(Msg, [_ | Rest]) :-  
    process(Msg, Rest).
```

Outline

- 3 Prolog as an Interpreter
 - Programs as Inputs
 - The Prolog Meta-Interpreter
 - Forward Chaining
 - OOP
 - Other Interpreters

Other Interpreters

- Breadth-first
- Best-first
- Uncertainty
- Constraint solving
- ...

Outline

- 1 Introduction to Prolog
- 2 Functional Prolog
- 3 Prolog as an Interpreter
- 4 Prolog for Software Modeling**
- 5 Prolog Libs and Tricks
- 6 Summary

Outline

4 Prolog for Software Modeling

- OPM
- Erlang AST
- Synthesizing Code

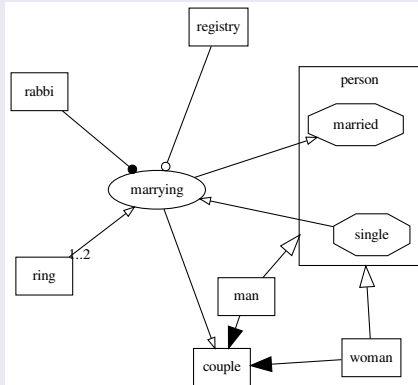
The Object-Process Methodology (OPM)

Dov Dori, 2002

Paragraph (OPL)

object person
states single, married.
object man, woman, couple.
object registry, rabbi, ring.
process marrying.
 man, woman **extends** person.
 couple **consists** man, woman.
 marrying **changes** person
from single **to** married.
 marrying **consumes** ring(1,2).
 marrying **yields** couple.

Diagram (OPD)



Representing OPM

OPM database

```
object(person).
object(man).
object(woman).
process(marrying).
state(person::single).
state(person::married).
extends(man, person).
extends(woman, person).
consists(couple, man).
consists(couple, woman).
consumes(marrying, ring).
produces(marrying, couple).
...
```

Querying

```
?- object(X).
X = person ;
X = man ;
X = woman;
...
```

```
?- extends(X, person).
X = man ;
X = woman
```

More Queries

Use cases

```
use_case(Proc) :-  
    process(Proc),  
    object(User),  
    handles(User, Proc),  
    \+ (process(Top),  
        consists(Top, Proc)).
```

Conversion functions

```
convert(Proc, From, To) :-  
    process(Proc),  
    consumes(Proc, From),  
    yields(Proc, To),  
    \+ requires(Proc, _),  
    \+ handles(_, Proc),  
    \+ (consumes(Proc, X),  
        X \= From),  
    \+ (yields(Proc, Y),  
        Y \= To).
```

Outline

4 Prolog for Software Modeling

- OPM
- Erlang AST
- Synthesizing Code

Erlang ASTs

Erlang

```
-module(pq).  
p(X) -> X + 1.  
q(X) -> X + 2.  
r(A) -> A + 3.
```

AST tuples

```
{function,4,p,1,  
  [{clause,4, [{var,4,'X'}],[],  
    [{op,5,'+',{var,5,'X'},{integer,5,1}}]}}
```

Erlang ASTs

Erlang

```
-module(pq).  
p(X) -> X + 1.  
q(X) -> X + 2.  
r(A) -> A + 3.
```

AST functors

```
function(4, 'p', 1,  
  [clause(4, [var(4, 'X')], [],  
    [op(5, '+', var(5, 'X'), integer(5, 1))])])
```

Outline

4 Prolog for Software Modeling

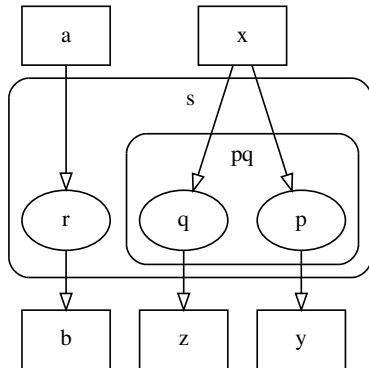
- OPM
- Erlang AST
- Synthesizing Code

A Data Flow Model

Paragraph

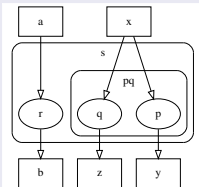
process p, q, r, pq, s.
object x, y, z, a, b.
 pq **consists** p, q.
 s **consists** pq, r.
 p **consumes** x.
 q **consumes** x.
 p **yields** y.
 q **yields** z.
 r **consumes** a.
 r **yields** b.

Diagram



Synthesized Sequential Code

Model



Source Code

```
-module(pq).
p(X) -> X + 1.
q(X) -> X + 2.
r(A) -> A + 3.
```

Sequential Code

```
-module(pq1).
-record(pq_ret, {y,z}).
-record(s_ret, {b,y,z}).
% p, q, and r
pq(X) ->
    Y = p(X),
    Z = q(X),
    #pq_ret{y=Y, z=Z}.
s(A, X) ->
    #pq_ret{y=Y, z=Z} = pq(X),
    B = r(A),
    #s_ret{b=B, y=Y, z=Z}.
```

Synthesized Parallel Code

```
-module(pq1).  
-record(pq_ret, {y, z}).  
-record(s_ret, {b, y, z}).  
p(X) -> X + 1.  
p1(X) ->  
    Y = p(X),  
    pq ! {y, Y}.  
q(X) -> X + 2.  
q1(X) ->  
    Z = q(X),  
    pq ! {z, Z}.  
r(A) -> A + 3.  
r1(A) ->  
    B = r(A),  
    s ! {b, B}.
```

Synthesized Parallel Code

```
pq(X) ->
  register(q, spawn(fun() -> q1(X) end)),
  register(p, spawn(fun() -> p1(X) end)),
  receive {y,Y} -> ok end,
  receive {z,Z} -> ok end,
  #pq_ret{y = Y,z = Z}.

pq1(X) ->
  #pq_ret{y = Y,z = Z} = pq(X),
  s ! {y,Y}, s ! {z,Z}.

s(A, X) ->
  register(s, self()),
  register(r, spawn(fun() -> r1(A) end)),
  register(pq, spawn(fun() -> pq1(X) end)),
  receive {y,Y} -> ok end,
  receive {z,Z} -> ok end,
  receive {b,B} -> ok end,
  #s_ret{y = Y,z = Z,b = B}.
```

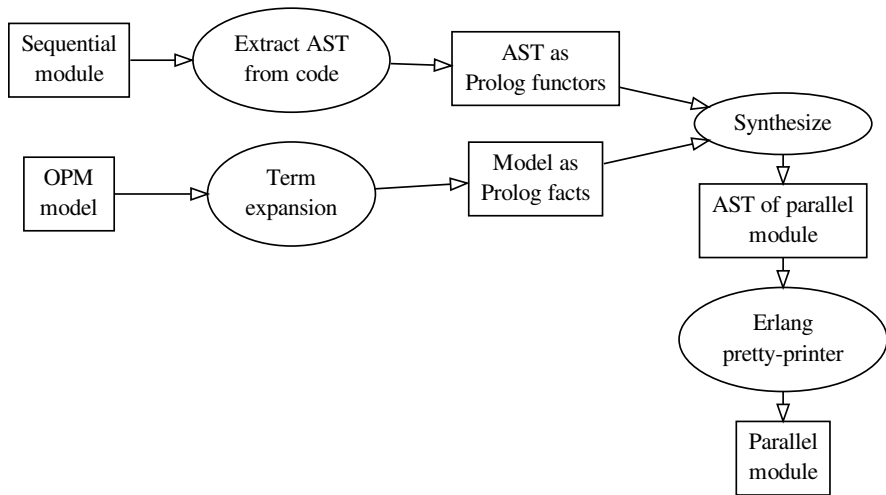
Example Rules

```
yields_snd0(P, X,  
            op('!', atom(P),  
              tuple([atom(X), V]))) :-  
  var_of(X, V).
```

```
yields_snd(From, To, X, S) :-  
  consists(P, From),  
  consists(P, To),  
  yields_snd0(To, X, S).
```

```
yields_snd(From, To, X, S) :-  
  consists(P, From),  
  \+ consists(P, To),  
  yields_snd0(P, X, S).
```

Workflow



Outline

- 1 Introduction to Prolog
- 2 Functional Prolog
- 3 Prolog as an Interpreter
- 4 Prolog for Software Modeling
- 5 Prolog Libs and Tricks**
- 6 Summary

Outline

5 Prolog Libs and Tricks

- DCG
- Constraints
- Meta-Prolog

Definitive Clause Grammar (DCG)

Grammar Rules

```
integer(I) -->  
    digit(D0),  
    digits(D),  
    { number_chars(I, [D0|D]) }.
```

```
digits([D|T]) -->  
    digit(D), !,  
    digits(T).  
digits([]) --> [].
```

```
digit(D) -->  
    [ D ],  
    { code_type(D, digit) }.
```

Definitive Clause Grammar (DCG)

Grammar Rules

```

integer(I) -->
  digit(D0),
  digits(D),
  { number_chars(I, [D0|D]) }.

digits([D|T]) -->
  digit(D), !,
  digits(T).

digits([]) --> [].

digit(D) -->
  [ D ],
  { code_type(D, digit) }.

```

Call grammar rules

Regular Prolog

Match in list

Definitive Clause Grammar (DCG)

Phrasing

```
?- phrase(integer(X), "42_times", Rest).
```

```
X = 42
```

```
Rest = "_times"
```

```
?- phrase(integer(42), S).
```

```
S = "42"
```

Outline

5 Prolog Libs and Tricks

- DCG
- **Constraints**
- Meta-Prolog

Constraints Solving over Finite Domains

Problem

```

  S E N D
+ M O R E
-----
M O N E Y

```

Solver

```

:- use_module(library(clpfd)).

puzzle([S,E,N,D] + [M,O,R,E] =
        [M,O,N,E,Y]) :-
    Vars = [S,E,N,D,M,O,R,Y],
    Vars ins 0..9,
    all_different(Vars),
    S*1000 + E*100 + N*10 + D +
    M*1000 + O*100 + R*10 + E #=
    M*10000 + O*1000 + N*100 + E*10 + Y,
    M #\= 0, S #\= 0.

```

Variables Labeling

Solving the puzzle

```
?- puzzle(As+Bs=Cs), label(As).  
As = [9, 5, 6, 7],  
Bs = [1, 0, 8, 5],  
Cs = [1, 0, 6, 5, 2]
```

Labeling options

- Minimum / maximum of an expression
- By order
- Other strategies

Outline

5 Prolog Libs and Tricks

- DCG
- Constraints
- Meta-Prolog

Reflective Predicates

Caller/Callee relations

```
uses(Caller, Callee) :-  
    current_predicate(Caller, Head),  
    \+ predicate_property(Head, built_in),  
    clause(Head, Body),  
    amember(Goal, Body),  
    \+ var(Goal),  
    functor(Goal, Callee, _).
```

Reflective Predicates

Sample Program

```
bar(X) :- foo(X).  
bar(X) :- goo(1), bar(X).
```

Usage

```
?- uses(bar, X).  
X = foo ;  
X = goo ;  
X = bar
```

```
?- uses(X, foo).  
X = bar  
  
?- recursive(X).  
X = bar
```

Manipulative Predicates

Poor man's AOP

```
wrap(Head, Before, After) :-  
    clause(Head, Body),  
    retract((Head :- Body)),  
    Head =.. [Name | Args],  
    atom_concat(Name, 1, Name1),  
    Head1 =.. [Name1 | Args],  
    assert((Head1 :- Body)),  
    assert((Head :- Before, Head1, After)),  
    fail.  
wrap(_, _, _).
```

Manipulative Predicates

Predicate

```
:- dynamic test/1.  
test(A) :- atom(A), !, print(A).  
test(X) :- N is X + 1, print(N).
```

Wrapping

```
?- test(5).  
6  
?- wrap(test(X), print(X), print(a)).  
?- test(5).  
56a
```

Outline


- 1 Introduction to Prolog
- 2 Functional Prolog
- 3 Prolog as an Interpreter
- 4 Prolog for Software Modeling
- 5 Prolog Libs and Tricks
- 6 Summary**

Claimer

Prolog can:

- Help you write DSLs and interpreters
- Add AI capabilities to your domain
- Be a hacker's delight!

Bibliography

-  Leon Sterling, Ehud Shapiro
The Art of Prolog
-  Richard O'Keefe
The Craft of Prolog
-  Yoav Shoham
Artificial Intelligence Techniques in Prolog
-  Ivan Bratko
Prolog Programming for Artificial Intelligence
-  Dov Dori
Object-Process Methodology

thank :- you, !.