Prolog use cases *other than* genealogy

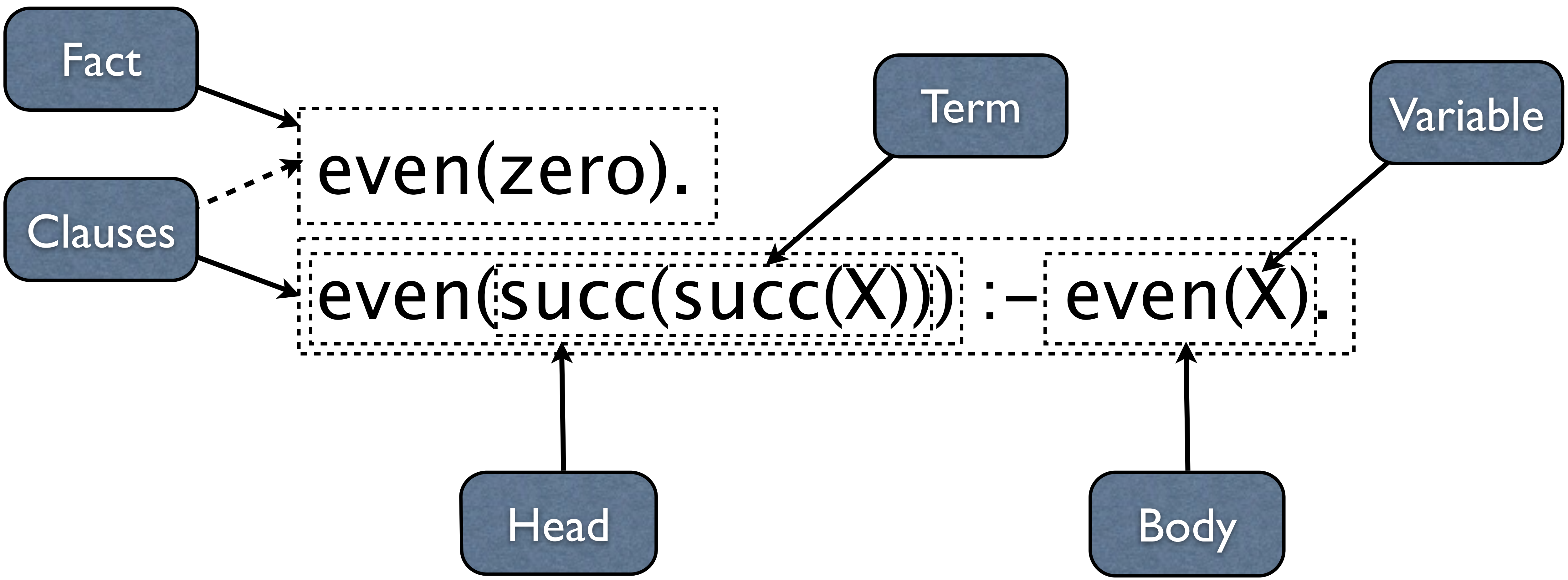
Ralf Lämmel
Software Languages Team, Koblenz

Slides hosted at https://github.com/rlaemmel/pltcourse

What’s Prolog?

- A language based on ***logic*** (say, definite Horn clauses).
- A full-blown ***declarative*** programming language.

Terminology



Demo

```
even(zero).

even(succ(succ(X))) :- even(X).
```

```
$ pwd
/home/pltcourse/src/prolog-samples
$ swipl -f peano.pro
?- even(X).
X = zero ;
X = succ(succ(zero)) ;
X = succ(succ(succ(succ(zero)))) ;
X = succ(succ(succ(succ(succ(succ(zero))))))
EOF: halt
```

Prerequisites

- Propositional logic
- Predicate logic
- Herbrand universe
- Unification
- SLD resolution

You may be able to program w/o knowing these things, but your understanding will be severely limited.

Prolog — why?

- Highly declarative.
- Highly operational.
- Highly scripted.
- Highly untyped.
- Highly typeable.
- Highly debuggable.
- **Highly under-appreciated.**
- ...

A super-weapon
of a
computer scientist

Simple examples

hello.pro

```
main :-
  write('Hello, world!'),
  nl.
```

```
$ swipl
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 5.10.4)
Copyright (c) 1990-2011 University of Amsterdam, VU Amsterdam

?- ['hello.pro'].
% hello.pro compiled 0.00 sec, 992 bytes
true.

?- main.
Hello, world!
true.

?- halt.
$
```

... or use CTRL-D

auto.pro

```
main :-
  write('Hello, world!'),
  nl.

:- main, halt.
```

```
$ swipl -f auto.pro
Hello, world!
$
```

The genealogy use case

```
% Steve's adopted parents

sex(steve,male).
father(paul,steve).
mother(clara,steve).

% Steve's biological parents

father(abdul,steve).
mother(joanne,steve).

% Sister of Steve

sex(mona,female).
father(abdul,mona).
mother(joanne,mona).

% Steve's daughter back from his sterile period

sex(lisa,female).
father(steve,lisa).
mother(anne,lisa).

...
```

http://www.applegazette.com/feature/the-family-tree-of-steve-jobs/

Genealogy relations

```
grandfather(X,Y) :-
  father(X,Z),
  father(Z,Y).

sibling(X,Y) :-
  father(F,X),
  father(F,Y),
  mother(M,X),
  mother(M,Y),
  X \== Y.

sister(X,Y) :-
  sibling(X,Y),
  sex(X,female).
```

Prolog queries

% Do we know who Steve's grandfather is?

```
?- grandfather(X,steve).
false.
```

% Do we know who Reed's grandfather is?

```
?- grandfather(X,reed).
X = paul ;
X = abdul ;
false.
```

Genealogy relations cont'd

```
halfsister(X,Y) :-
  sex(X,female),
  father(FX,X),
  mother(MX,X),
  father(FY,Y),
  mother(MY,Y),
  overlap(FX,FY,MX,MY).

overlap(F,F,MX,MY) :- MX \== MY.
overlap(FX,FY,M,M) :- FX \== FY.
```

Use of “disjunction”

```
halfsister(X,Y) :-
  sex(X,female),
  father(FX,X),
  mother(MX,X),
  father(FY,Y),
  mother(MY,Y),
  ( FX == FY, MX \== MY; FX \== FY, MX == MY ).
```

List processing

```
?- member(X,[a,b,c]).
X = a ;
X = b ;
X = c.

?- append([1,2,3],[4,5,6],X).
X = [1, 2, 3, 4, 5, 6].
```

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

append([],L,L).
append([H|T],L,[H|R]) :- append(T,L,R).
```

Directed graphs

```
node(1).
node(2).
node(3).

edge(1,2).
edge(2,3).

connected(X,Y) :-
  edge(X,Y).

connected(X,Y) :-
  edge(X,Z),
  connected(Z,Y).
```

```
?- connected(1,2).
true

?- connected(1,3).
true

?- connected(2,1).
false
```

Implementing Peano axioms

```
add(zero,X,X).
add(succ(X),Y,succ(Z)) :- add(X,Y,Z).
```

```
?- add(succ(succ(zero)),succ(zero),X).
X = succ(succ(succ(zero))).
```

A simple expression interpreter

```
eval(num(N),N) :-
  number(N).

eval(add(E1,E2),N) :-
  eval(E1,N1),
  eval(E2,N2),
  N is N1 + N2.
```

```
?- eval(add(add(num(1),num(2)),num(3)),X).
X = 6.
```

Totaling salaries

http://101companies.org/wiki/Contribution:prologStarter

```
total(company(_,Ds),R) :-
    total(Ds,R).

total([],0).

total([H|T],R) :-
    total(H,R1),
    total(T,R2),
    R is R1 + R2.

total(dept(_,M,Units),R) :-
    total(M,R1),
    total(Units,R2),
    R is R1 + R2.

total(employee(_,_,S),S).
```

```
?- total(company(me,[dept(leadership,employee(ralf,b127,42),[])]),X).
X = 42.
```

Cutting salaries

http://101companies.org/wiki/Contribution:prologStarter

```
cut( company(N,Ds1),
     company(N,Ds2)) :-
  cut(Ds1,Ds2).

cut(N1,N2) :-
  number(N1), N2 is N1 / 2.

cut([],[]).
cut([H1|T1],[H2|T2]) :-
  cut(H1,H2), cut(T1,T2).

cut( dept(X,M1,Units1),
     dept(X,M2,Units2)) :-
  cut(M1,M2),
  cut(Units1,Units2).

cut( employee(X,Y,S1),
     employee(X,Y,S2)) :-
  cut(S1,S2).
```

```
?- cut(company(me,[dept(leadership,employee(ralf,b127,42),[])]),X).
X = company(me, [dept(leadership, employee(ralf, b127, 21), [])])
```

I/O

File I/O Edinburgh style

```
test :-
  see('eval.sample'),
  read(E),
  seen,
  eval(E,V),
  write(V),
  nl.
```

```
?- test.
6
true.
```

File I/O ISO style

```
test :-
  open('eval.sample',read,In),
  read(In,E),
  close(In),
  eval(E,V),
  write(V),
  nl.
```

```
?- test.
6
true.
```

I/O predicates

- see/1: open file for input, set it as current input
- seen/0: close current input, return to previous one
- read/1: read a term from the input
- tell/1: open file for output, set is as current output
- told/0: close current output, return to previous one
- write/1: write a term to the output
- nl/0: start a new line in the output
- format/2: formatted output
- open/3: open a stream for input or output
- close/1: close a stream
- write/2: write a term to a stream
- ...

Types

“Types are programs.”

```
expr(num(N)) :- number(N).
expr(add(E1,E2)) :- expr(E1), expr(E2).
```

```
?- expr(add(num(1),num(2))).
true.

?- expr(foo).
false.
```

Another example: finding the max leaf in a tree

```
tree(leaf(X)) :- integer(X).
tree(fork(T1,T2)) :- tree(T1), tree(T2).

max(leaf(X),X).
max(fork(T1,T2),X) :- max(T1,Y), max(T2,Z), X is max(Y,Z).
```

Another operator

```
?- max(fork(leaf(1),fork(leaf(42),leaf(88))),X).
X = 88.
```

Built-in type tests

- number/1
- integer/1
- atom/1
- is_list/1
- ...

```
?- number(1.1).
true.
?- number(foo).
false.
?- integer(1.1).
false.
?- integer(42).
true.
?- atom(42).
false.
?- atom(foo).
true.
?- is_list(foo).
false.
?- is_list([foo]).
true.
```

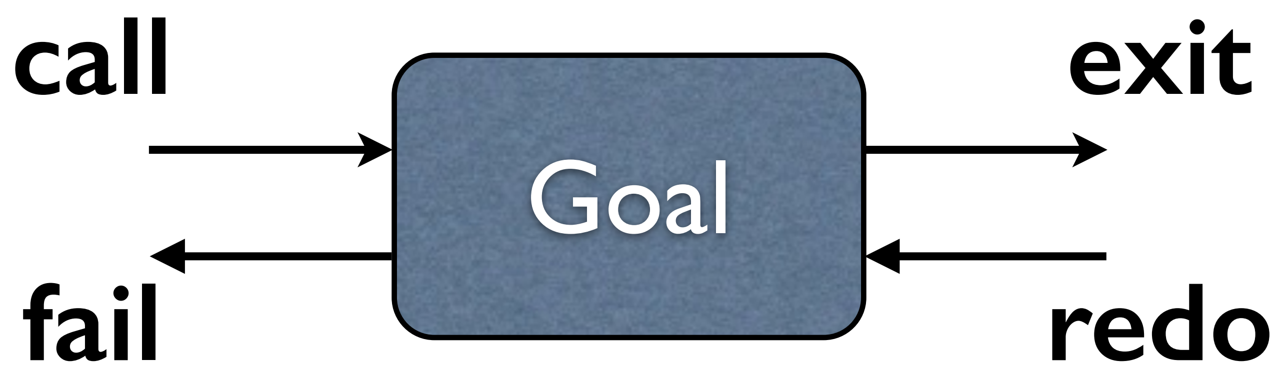
Debugging

Debugging with traces

```
?- trace, expr(add(num(1),num(2))).
   Call: (7) expr(add(num(1), num(2))) ? creep
   Call: (8) expr(num(1)) ? creep
   Call: (9) number(1) ? creep
   Exit: (9) number(1) ? creep
   Exit: (8) expr(num(1)) ? creep
   Call: (8) expr(num(2)) ? creep
   Call: (9) number(2) ? creep
   Exit: (9) number(2) ? creep
   Exit: (8) expr(num(2)) ? creep
   Exit: (7) expr(add(num(1), num(2))) ? creep
true.

[trace]  ?-
```

The *Box Model* of goal execution



- **call**: enter the goal when first attempting proof
- **exit**: leave the goal when completing proof
- **redo**: re-entering goal upon backtracking
- **fail**: ultimately finishing goal when without (further) proof

Debugging with traces

"skip" can be used to go from "call" to "exit" port right away.

```
?- expr(add(num(1),num(2))).
true.

?- trace, expr(add(num(1),num(2))).
   Call: (7) expr(add(num(1), num(2))) ? Options:
+:                  spy          -:                  no spy
/c|e|r|f|u|a goal:  find         .:                  repeat find
a:                  abort        A:                  alternatives
b:                  break        c (ret, space):     creep
[depth] d:          depth        e:                  exit
f:                  fail         [ndepth] g:         goals (backtrace)
h (?):              help         i:                  ignore
l:                  leap         L:                  listing
n:                  no debug     p:                  print
r:                  retry        s:                  skip
u:                  up           w:                  write
m:                    exception details
C:                  toggle show context
   Call: (7) expr(add(num(1), num(2))) ?
```

Breakpoints

```
?- spy(number/1).
% Spy point on number/1
true.

[debug]  ?- expr(add(num(1),num(2))).
 * Call: (8) number(1) ? creep
 * Exit: (8) number(1) ? creep
   Exit: (7) expr(num(1)) ? leap
 * Call: (8) number(2) ? leap
 * Exit: (8) number(2) ? leap
true.
```

Modes

Flexible modes

```
?- add(X,Y,Z).
X = zero,
Y = Z ;
X = succ(zero),
Z = succ(Y) ;
X = succ(succ(zero)),
Z = succ(succ(Y)) .

?- add(X,Y,succ(succ(zero))).
X = zero,
Y = succ(succ(zero)) ;
X = Y, Y = succ(zero) ;
X = succ(succ(zero)),
Y = zero ;
false.
```

Inflexible modes

```
?- X is 1 + 1.
X = 2.

?- 2 is 1 + 1.
true.

?- 2 is X + 1.
ERROR: is/2: Arguments are not sufficiently instantiated
```

Documentation of modes

- Modes
  - +: needs to be instantiated upon call
  - -: will be instantiated upon exit
  - ?: neither of the two above
- Application to example
  - add(?X,?Y,?Z)
  - is(-X,+Y)

Modes not sufficient here.
We need groundness.

Examples of modes in the list library

- member(?Elem, ?List)
- append(?List1, ?List2, ?List1AndList2)
- append(+ListOfLists, ?List)
- selectchk(+Elem, +List, -Rest)
- permutation(?Xs, ?Ys)
- subset(+SubSet, +Set)
- ...

Basic modularization

```
:- ['Company.pro'].
:- ['Total.pro'].
:- ['Cut.pro'].
:- ['Depth.pro'].

:-
   see('sampleCompany.trm'),
   read(C1),
   seen,
   isCompany(C1),
   total(C1,R1),
   format('total = ~w~n',[R1]),
   cut(C1,C2),
   total(C2,R2),
   format('cut = ~w~n',[R2]),
   depth(C1,R3),
   format('depth = ~w~n',[R3]).

 :- halt.
```

Loading all 101companies modules and running tests.

```
company(
  'meganalysis',
  [ dept(
      'research',
      employee('Craig','Redmond',123456),
      [ employee('Erik','Utrecht',12345),
        employee('Ralf','Koblenz',1234)
      ]
    ),
    dept(
      'dev',
      employee('Ray','Redmond',234567),
      [ dept(
          'dev1',
          employee('Klaus','Boston',23456),
          [ dept(
              'dev1.1',
              employee('Karl','Riga',2345),
              [ employee('Joe','Wifi City',2344)
              ]
            )
          ]
        )
      ]
    )
  ]
).
```

That's a term to be "read".

Basic modularization

% Basic form of input
:- consult('MyPrologFile.pro').

% Concise notation
:- ['MyPrologFile.pro'].

% Ensure import (avoid repeated import)
:- ensure_loaded('MyPrologFile.pro').

Related predicates

% Predicate may be defined in more than file.

```
:- multifile father/2.
```

% Clauses may appear discontiguously in file.

```
:- discontiguous father/2.
```

% Re-load all files (typically after edits).

```
:- make.
```

Some reflections on Prolog's declarative and operational features

Lists versus sets of answers

```
max(X,Y,X) :- X >= Y.
max(X,Y,Y) :- X =< Y.
```

```
?- max(42,88,X).
X = 88.

?- max(42,42,X).
X = 42 ;
X = 42.
```

A single answer is preferred.

Efficiency

```
max(X,Y,X) :- X >= Y.
max(X,Y,Y) :- X < Y.
```

```
?- max(42,88,X).
X = 88.

?- max(42,42,X).
X = 42 ;
false.
```

Backtracking ultimately fails.

Operational reasoning

```
max(X,Y,X) :- X >= Y, !.
max(X,Y,Y) :- X < Y.
```

A green cut

```
?- max(42,88,X).
X = 88.

?- max(42,42,X).
X = 42.
```

No more superfluous backtracking

The logical meaning of the program is ***not*** changed by removing the cut.

Destroyed declarative semantics

```
max(X,Y,X) :- X >= Y, !.
max(X,Y,Y).
```

A red cut

```
?- max(42,88,X).
X = 88.

?- max(42,42,X).
X = 42.
```

No problem?

The logical meaning of the program is changed by removing the cut.

A red cut

```
max(X,Y,X) :- X >= Y, !.
max(X,Y,Y).
```

A green cut

```
max(X,Y,X) :- X >= Y, !.
max(X,Y,Y) :- X < Y.
```

```
?- max(88,42,42).
true.
```

```
?- max(88,42,42).
false.
```

Bottom line: don't use cut!

Structured cut

```
(If -> Then); _Else :- If, !, Then.
(If -> _Then); Else :- !, Else.
If -> Then :- If, !, Then.
```

```
max(X,Y,Z) :-
  X >= Y -> Z = X; X = Y.
```

Looks all good!

```
?- max(42,88,X).
X = 88.

?- max(42,42,X).
X = 42.

?- max(42,88,42).
false.
```

Graph example

```
connected(X,Y) :-
  edge(X,Y).

connected(X,Y) :-
  edge(X,Z),
  connected(Z,Y).
```

```
connected(X,Y) :-
  edge(X,Y) ->
    true;
    edge(X,Z),
    connected(Z,Y).
```

Free and bound variables

Terms with variables

- **var/1**: test a term to be a variable
- **ground/1**: test a term to be ground

```
?- var(42).
false.

?- var(X).
true.

?- X=42, var(X).
false.

?- var(foo(X)).
false.
```

```
?- ground(42).
true.

?- ground(X).
false.

?- X=42, ground(X).
X = 42.

?- ground(foo(X)).
false.
```

Use of non-ground terms

```
?- member(Y,[X,Z]).
Y = X ;
Y = Z.
```

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

```
?- varmember(Y,[X,Z]).
false
?- varmember(X,[X,Z]).
true
```

```
varmember(V,[H|_]) :- V==H.
varmember(V,[H|T]) :- V\==H, varmember(V,T).
```

Term de-/composition

Inspection of terms

- **functor/3**: observe functor symbol and arity
- **=../2**: take apart compound terms

```
?- functor(foo(bar),X,A).
X = foo,
A = 1.

?- foo(bar) =.. X.
X = [foo, bar].
```

Print terms with indentation

```
print_term(T) :-
  print_term(T,0).

print_term(T,N) :-
  spaces(N),
  ( var(T) ->
      format('~w~n',[T])
    ; T =.. [F|Ts],
      format('~w~n',[F]),
      M is N + 1,
      print_terms(Ts,M) ).

print_terms([],_).

print_terms([H|T],N) :-
  print_term(H,N),
  print_terms(T,N).

spaces(N) :-
  N > 0 -> write(' '), M is N - 1, spaces(M); true.
```

```
?-
print_term(add(num(1),a
dd(num(2),num(3)))).
add
 num
  1
 add
  num
   2
  num
   3
true.
```

Application to *Programming Language Theory*

Syntax of the trivial imperative language *assign*

```
program(Es) :- exprs(Es).

exprs([]).
exprs([E|Es]) :- expr(E), exprs(Es).

expr(N) :- number(N).
expr(E1+E2) :- expr(E1), expr(E2).
expr(V) :- atom(V).
expr(V=E) :- atom(V), expr(E).
```

```
?- program([x=1,y=x+41]).
true
```

Interpreter of *assign*

```
eval(Es,V) :- eval(Es,V,[],_).

eval([E],N,M1,M2) :-
  eval(E,N,M1,M2).
eval([E|Es],N,M1,M2) :-
  Es \== [], eval(E,_,M1,M0), eval(Es,N,M0,M2).
eval(N,N,M,M) :-
  number(N).
eval(E1+E2,N,M1,M2) :-
  eval(E1,N1,M1,M0), eval(E2,N2,M0,M2), N is N1+N2.
eval(V,N,M,M) :-
  atom(V), lookup(V,M,N).
eval(V=E,N,M1,M2) :-
  atom(V), eval(E,N,M1,M0), update(V,N,M0,M2).
```

```
?- eval([x=1,y=x+41],N).
N = 42
```

List-processing convenience

```
lookup(V,[(V,N)|_],N).
lookup(V,[(W,_)|R],N) :- V \== W, lookup(V,R,N).

update(V,N,[],[(V,N)]).
update(V,N,[(V,_)|R],[(V,N)|R]).
update(V,N,[(W,M)|R],[(W,M)|S]) :- V \== W, update(V,N,R,S).
```

Exercises
(in increasing order of difficulty)

Basic list processing

Define a predicate many/3 such that many(+X,+N,-L) creates a list L of length N where all elements are equal to X.

Basic file processing

Write a program that reads two numbers (terms) from a file, computes the sum, and writes the result to another file.

Basic tree processing

Define in–order traversal on an appropriate term representation for binary trees with numbers at the nodes such that the list of all numbers at the nodes is returned.

Syntax evolution

Consider again the syntax for the simple imperative programming language *assign*, as it was defined and interpreted earlier:

[x=1,y=x+41]

Revise the predicates program/1 and eval/2 (and friends) so that a more uniform syntax is used instead:

```
[assign(x,num(1)),
 assign(y,add(var(x),num(41)))]
```

Syntax evolution (variation)

Hard!

Consider again the syntax for the simple imperative programming language *assign*, as it was defined and interpreted earlier. Rather than using atoms for the program variables, use instead Prolog variables.

Higher-order predicates

Mediation between terms and goals

```
?- true.
true.

?- X=true, X.
X = true.

?- X=true, call(X).
X = true.
```

Applying predicates with apply/2

```
?- F=write,G=..[F,hello],G,nl.
hello
F = write,
G = write(hello).

?- call(write,hello),nl.
hello
true.

?- apply(write,[hello]),nl.
hello
true.
```

apply/2

```
apply(G1,L) :-
  G1 =.. [P|Args1],
  append(Args1,L,Args2),
  G2 =.. [P|Args2],
  G2.
```

List-processing combinators

Mapping over a list

```
?- map(increment,[1,2,3],R).
R = [2, 3, 4]
```

```
map(_,[],[]).
map(P,[H1|T1],[H2|T2]) :-
  apply(P,[H1,H2]),
  map(P,T1,T2).

increment(N1,N2) :- number(N1), N2 is N1 + 1.
```

In (SWI-)Prolog, there are predicates maplist/2+ just like that.

Filtering a list

```
greaterThan42(X) :- X > 42.
```

```
?- filter(greaterThan42,[40,41,42,43,44],R).
R = [43, 44]
```

```
filter(_,[],[]).
filter(P,[H|T],R) :-
  ( apply(P,[H]) -> R = [H|RR]; R = RR),
  filter(P,T,RR).
```

findall/3

There is also friends such as bagof/3, which we skip here.

Goals with multiple solutions

```
?- member(X,[40,41,42,43,44]), X > 42.
X = 43 ;
X = 44.
```

How to get access to the list of solutions programmatically?

Remember filter/3

```
?- filter(greaterThan42,[40,41,42,43,44],R).
R = [43, 44]
```

This is not a general approach in that we would need to define a new predicate each time we face a different goal with multiple solutions.

Use findall/3

```
?- findall(
|     X,
|     (
|       member(X,[40,41,42,43,44]),
|       X > 42
|     ),
|     L).
L = [43, 44]
```

Meta-interpreters

"Because it is possible to directly access program code in Prolog, it is easy to write interpreter of Prolog in Prolog. Such interpreter is called a **meta-interpreter**. Meta-interpreters are usually used to add some extra features to Prolog, e.g., to change build-in negation as failure to constructive negation." [*Barták98*]

http://kti.mff.cuni.cz/~bartak/prolog/meta_interpret.html

The simplest meta-interpreter

```
solve(Goal) :- call(Goal).
```

http://kti.mff.cuni.cz/~bartak/prolog/meta_interpret.html

Even simpler ...

```
solve(Goal) :- Goal.
```

The “vanilla” meta-interpreter

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

A meta-interpreter with proof construction

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

http://kti.mff.cuni.cz/~bartak/prolog/meta_interpret.html

A computed proof tree

```
eval(add(add(num(1),num(2)),num(3)),6) -
  (eval(add(num(1),num(2)),3) -
    (eval(num(1),1) -
      (number(1)-built_in),
     eval(num(2),2) -
      (number(2)-built_in),
     (3 is 1+2)-built_in),
   eval(num(3),3) -
     (number(3)-built_in),
   (6 is 3+3)-built_in
  )
```

Traversal combinators

Remember all the boilerplate?

http://101companies.org/wiki/Contribution:prologStarter

```
total(company(_,Ds),R) :-
    total(Ds,R).

total([],0).

total([H|T],R) :-
    total(H,R1),
    total(T,R2),
    R is R1 + R2.

total(dept(_,M,Units),R) :-
    total(M,R1),
    total(Units,R2),
    R is R1 + R2.

total(employee(_,_,S),S).
```

```
?- total(company(me,[dept(leadership,employee(ralf,b127,42),[])]),X).
X = 42.
```

Use a traversal scheme

http://101companies.org/wiki/Contribution:prologSyb

```
total(X,R) :-
  collect(getSalary,X,L),
  sum(L,R).

getSalary(employee(_,_,S),S).
```

collect/3

```
collect(P,X,L) :-
  apply(P,[X,Y]) ->
    L = [Y];
    X =.. [_|Xs],
    maplist(collect(P),Xs,Yss),
    append(Yss,L).
```

Traversal schemes exist for both queries and transformations.

```
cut(X,Y) :-
  stoptd(updateSalary,X,Y).

updateSalary(
  employee(N,A,S1),
  employee(N,A,S2)) :-
    S2 is S1 / 2.
```

stoptd/3

```
stoptd(P,X,Y) :-
  apply(P,[X,Y]) ->
    true;
    X =.. [F|Xs],
    maplist(stoptd(P),Xs,Ys),
    Y =.. [F|Ys].
```

Data = programs

Assertion of facts

```
assertEdge((X,Y)) :-
  assertz(edge(X,Y)).
```

```
?- maplist(assertEdge, [(1,2),(2,3)]).

?- listing(edge/2).
:- dynamic edge/2.

edge(1, 2).
edge(2, 3).
```

Database predicates

- *dynamic :PredicateIndicator*: indicates that a predicate can be manipulated (use with goal clause).
- *abolish(:PredicateIndicator)*: removes all clauses of a predicate.
- *retract(+Term)*: retracts first unifying fact or clause in the database.
- *compile_predicates(:ListOfNameArity)*: compiles a list of specified dynamic predicates.

Definite Clause Grammars

Different representations for the simple imperative language *assign*

Term representation using Prolog's built-ins

```
[x=1,y=x+41]
```

Term representation using "fresh" functors

```
[assign(x,num(1)), assign(y,add(var(x),num(41)))]
```

```
[id(x),=,num(1),;,id(y),=,id(x),+,num(41),;]
```

List of *tokens* to be *parsed* into terms

A simple EBNF for *assign*

```
program = (expr ';')+

expr =  num
     |  id
     |  expr '+' expr
     |  id '=' expr
```

Definite Clause Grammars (DCGs) are embedded into Prolog to directly enable parsing. We need to eliminate left recursion (when using the standard semantics).

A DCG for *assign*

```
program --> expr, [;], rest.

rest --> [].
rest --> program.

expr --> [num(_)].
expr --> [id(_)].
expr --> expr, [+], expr.
expr --> [id(_)], [=], expr.
```

Definite Clause Grammars (DCGs) are embedded into Prolog to directly enable parsing. We need to eliminate left recursion (when using the standard semantics).

An operational DCG for *assign*

```
program --> expr, [;], rest.

rest --> [].
rest --> program.

expr --> [num(_)], add.
expr --> [id(_)], add.
expr --> [id(_)], [=], expr.

add --> [].
add --> [+], expr.
```

Demo of parsing with DCG

```
?- program([id(x),=,num(1),;,id(y),=,id(x),+,num(41),;],[]).
true
```

Input tokens

Output tokens

Compilation of DCGs

```
add --> [].
add --> [+], expr.
```

⇓

```
add(A, A).
add([+|A], B) :-
   expr(A, B).
```

The "accumulator" technique is used.

End of crash course; if we were to continue this course ...

- XML, RDF, JSON access
- Relational algebra and DB access
- Program refactoring on top of JDK
- Program analysis in reverse engineering
- Code generation (e.g., generate graphviz)
- ...