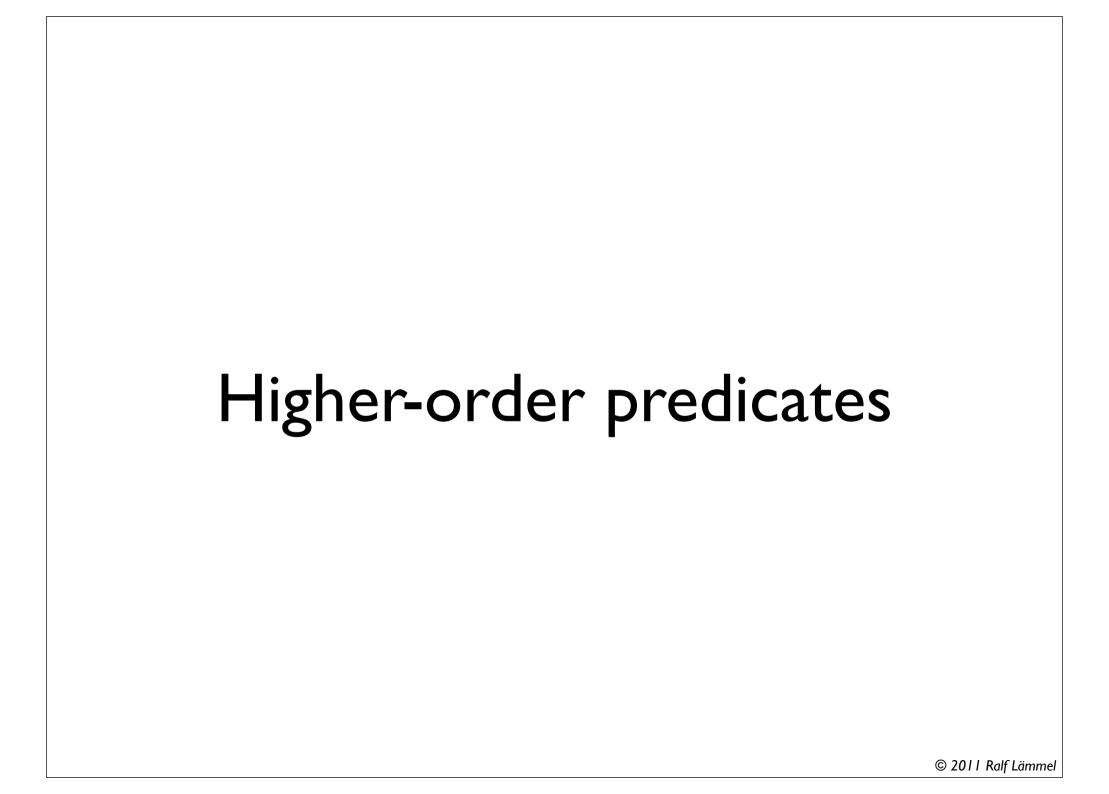
Prolog use cases other than genealogy (Part II/II)

Ralf Lämmel Software Languages Team, Koblenz

https://developers.svn.sourceforge.net/svnroot/developers/repository/prolog-crash-course/



Mediation between terms and goals

?- 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.



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

```
?- 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).
```

greaterThan42(X): -X > 42.

findall/3

There is also friends such as bagof/3 which we skip over 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?

Use 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

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*]

The simplest meta-interpreter

solve(Goal) :- call(Goal).

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).
```

A computed proof tree

```
eval(add(add(num(1),num(2)),num(3)),6) -
 (eval(add(num(1),num(2)),3) -
  (eval(num(I),I) -
    (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
```



Remember all this boilerplate?

http://101companies.org/index.php/101implementation:prolog

```
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

```
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

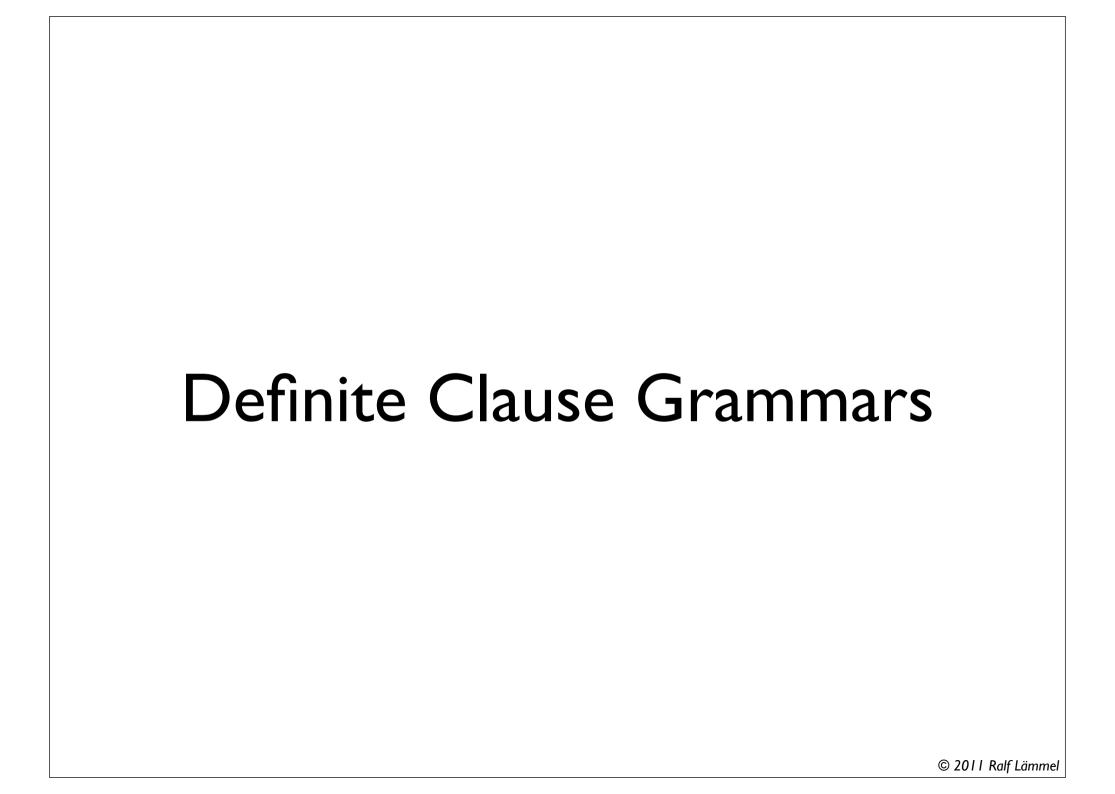
assertEdge((X,Y)) :assertz(edge(X,Y)).

- ?- 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.



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

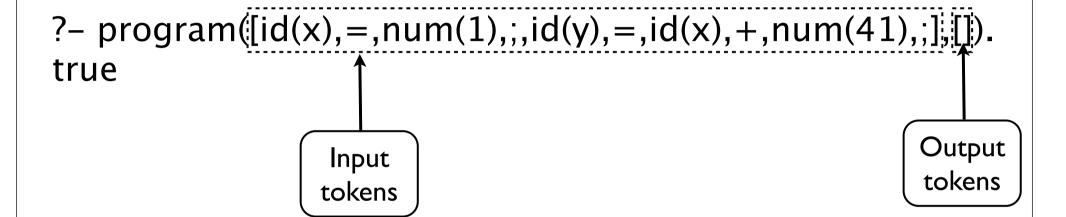
A simple EBNF for assign

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

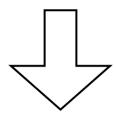
```
program --> expr, [;], rest.
rest --> [].
rest --> program.
expr --> [num(_)], add.
expr --> [id( )], add.
expr --> [id()], [=], expr.
add --> [].
add --> [+], expr.
```

Demo of parsing with DCG



Compilation of DCGs

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



The "accumulator" technique is used.

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

If we were to continue this crash course ...

- XML access
- Program refactoring on top of JDK
- Program analysis in reverse engineering
- Code generation (e.g., generate graphviz)

• ...

