universitätbonn ROOTS

Chapter 7. Metadata, Metaprogramming and Reflection

– **Updated: June 20, 2012** –

Meta-Data

Meta-Programming

Computational Reflection

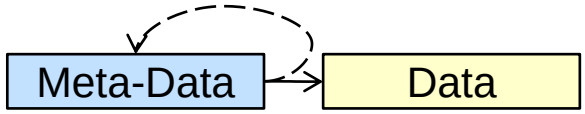
Meta-Interpreters

Meta-Data

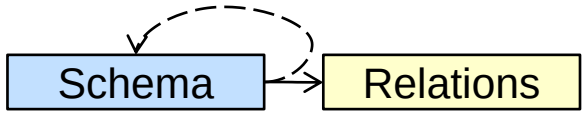
Scenario: From JTransformer to StarTransformer – Language independent factbase traversal

Meta-Data

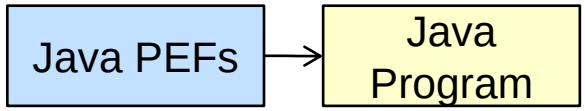
- Meta-Data
  - Data holding information about (other) data



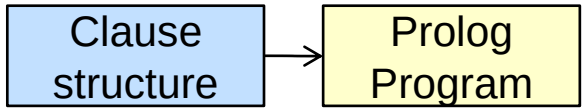
- Examples
  - Schema information of a database
    - ⇨ Describes the structure of relations / tables
    - ⇨ Must be self-descriptive if schema information is stored in relations too



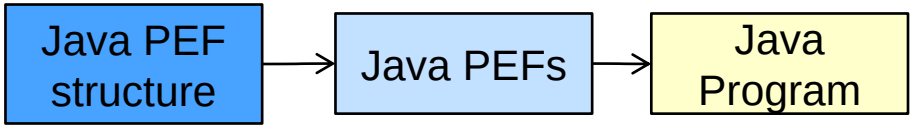
  - JTransformer PEFs
    - ⇨ Describe the structure of a Java program



  - Prolog
    - ⇨ Clauses that describe the other clauses

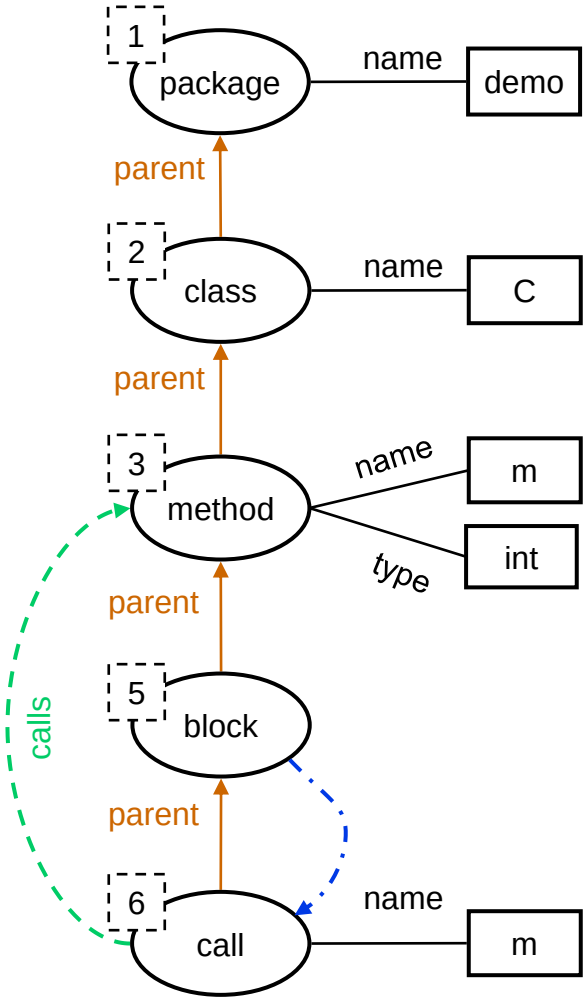


- Next: Meta-Data about PEFs
  - Clauses that describe the structure of PEFs
  - Actually Meta-meta-data



Scenario: Navigation in the AST

Legend: parent reference
child reference
other reference




```
package(1, 0, 'demo')
class(2, 1, 'C')
method(3, 2, 'm', int,[])
block(5, 3, [6])
call(6, 5, null, 3 )
```

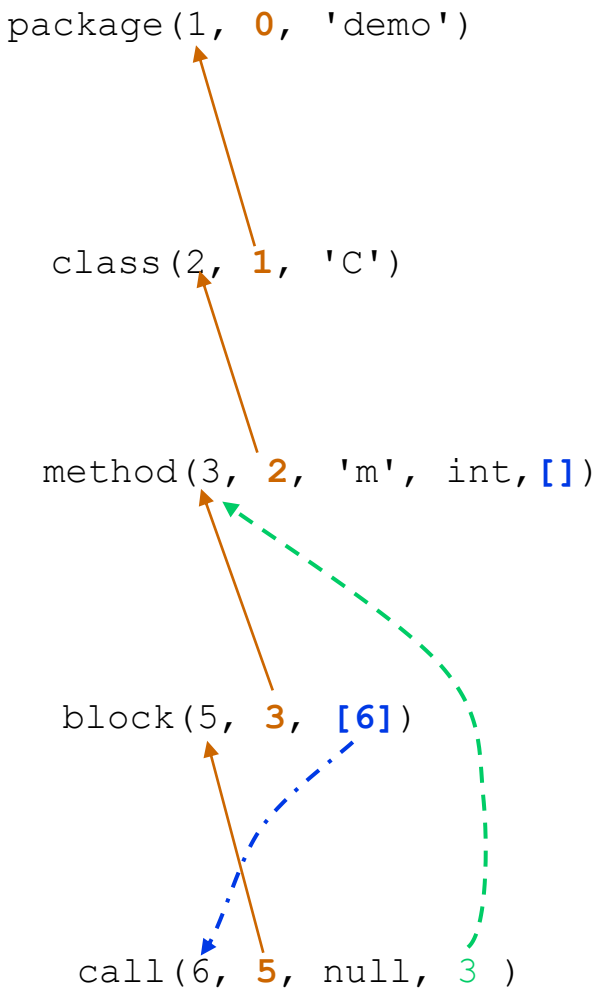
Scenario: Navigation in the AST

Legend: parent reference
child reference
other reference

- How to get from a block to the containing package?

```
getContainingClass(BlockId,PackageId) :-
   block(BlockId, MethodId, _),
   method(MethodId,ClassId, _, _, _),
   class(ClassId,PackageId, _).
```

- But what if the block is nested inside another statement?
  - Try all possible statements? ☹
- What if we do not know the exact path and the program element types to traverse?
- How to write a generic `getParent(Id, Parent)`?



Meta-Data: Specification of PET structure

- Sample PEF: `call(6, 5, null, 3 )`
- Sample PEF structure: `call`(id#, parent#, recv#, called#)
- Specified by:

language

PEF type (= AST node type)

```
ast_node_def(‚Java', call, [
    ast_arg( id,     mult(1,1,no ), id,  [call]   ),
    ast_arg( parent, mult(1,1,no ), id,  [block]  ),
    ast_arg( recv,   mult(0,1,no ), id,  [...]    ),
    ast_arg( called, mult(1,1,no ), id,  [method] )
]).
```

specification of PEF arguments

argument name

multiplicity

order (no = not ordered, ord = ordered)

kind of value (id = identity, attr = primitve)

legal syntactic type(s) of argument values (multiple types are allowed, e.g. a recv, can have many)

ast_node_def/3

- **ast_node_def**(?Lang, ?NodeType, ?ArgumentDescriptors) is nondet
  - Describes a syntax element of a given language
  - *Lang* is the language we want to describe (e.g. 'Java')
  - *NodeType* represents the AST node type (classT, callT, blockT,…)
  - *ArgumentDescriptors* describes the arguments of this particular node type
- Example: callT/7 in JTransformer
  - See http://sewiki.iai.uni-bonn.de/research/jtransformer/api/java/pefs/3.0/callt

```
ast_node_def('Java',callT,[
     ast_arg(id,       mult(1,1,no ),  id,    [callT]),
     ast_arg(parent,   mult(1,1,no ),  id,    [id]),
     ast_arg(encl,     mult(1,1,no ),  id,    [methodT, constructorT,
                                               classInitializerT, fieldT]),
     ast_arg(expr,     mult(0,1,no),   id,    [expressionType, typeRefT,
                                               nullType]),
     ast_arg(name,     mult(1,1,no ),  attr,  [atom]),
     ast_arg(args,     mult(0,*,ord ), id,    [expressionType]),
     ast_arg(ref,      mult(1,1,no ),  id,    [methodT, constructorT])
 ]).
```

Argument Descriptors (1)

- **ast_arg**(ArgName, Cardinality, IdOrAttribute, Types)
  - *ArgName* is the name of the argument (usually an atom)
  - *Cardinality* is a term of the form mult(From,To,OrderedOrNot)

Cardinality	Explanation
mult(0,*,no)	Any cardinality including 0, no order.
mult(0,*,ord)	Any cardinality including 0, the values are ordered. In JTransformer such argument are lists.
mult(1,2,no)	A cardinality range with a lower and upper bound. Not ordered.
mult(0,1,no)	An optional, single-valued argument - may be the atom 'null'. Clearly not ordered.

Argument Descriptors (2)

- **ast_arg**(ArgName, Cardinality, IdOrAttribute, Types)
  - *IdOrAttribute* is either
    - ⇨ **id** – Indicates that the value is the identity of an AST node.
    - ⇨ **attr** – Indicates that the value can be any legal Prolog term and is not to be interpreted as an id.
  - *Types* is a list of AST node types defined for this language. That may be
    - ⇨ any term in a second argument of an ast_node_def/3 fact for this language,
    - ⇨ the types 'typeTermType' and 'atom' may be used.
      - – The *'typeTermType'* indicates a term of the form type(class, id, int) or type(basic, typename, int).
    - ⇨ Each value of an AST node argument must be from one of these types.
    - ⇨ 'null', is legal if the cardinality includes 0

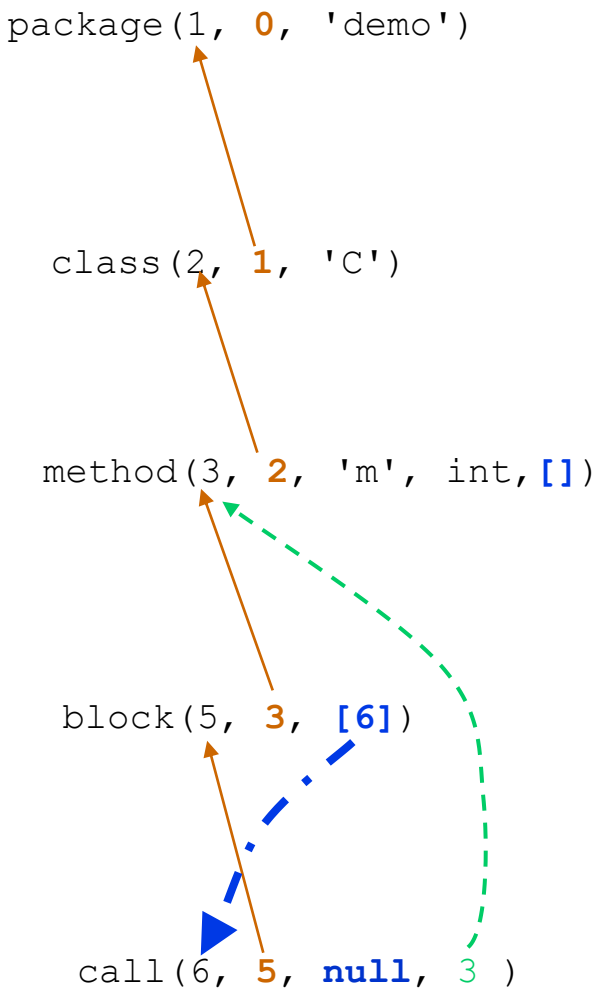
ast_subtree/2

Legend: parent reference
child reference
other reference

- **ast_sub_tree**(?L, ?ArgName) is nondet
  - Describes a child-reference
  - *L* represents the language that we specify (e.g. 'Java')
  - *ArgName* is an argument name that refers to a child node in any AST node of L.

```
ast_sub_tree('Java', body).
ast_sub_tree('Java', stms).
ast_sub_tree('Java', recv).
ast_sub_tree('Java', args).
...
```

- The ast_sub_tree/2 declarations enables language-independent top down traversal of an AST



AST-Meta-Model

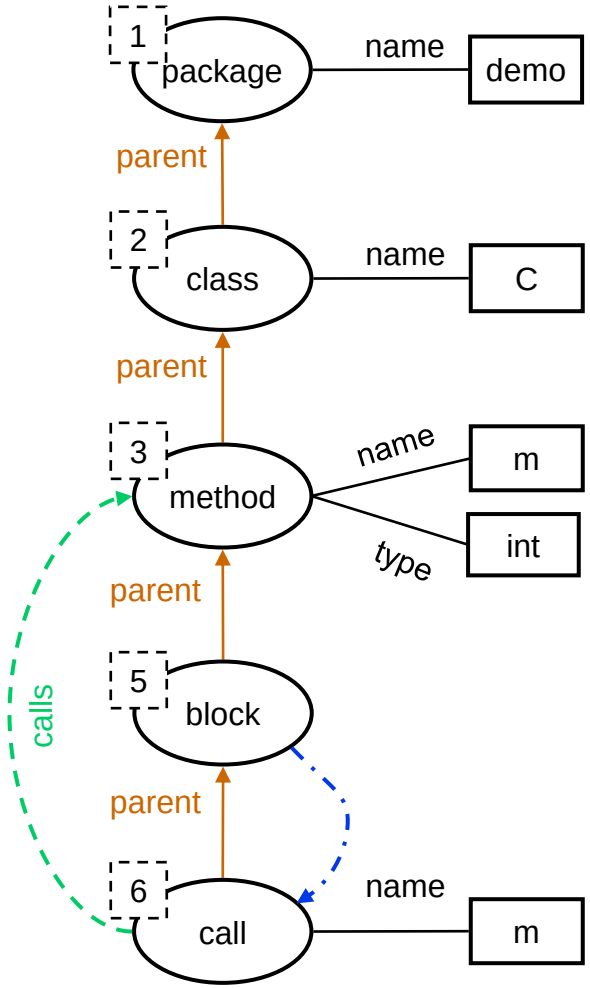
- The AST of a language is specified by:
  - ast_node_def /3 – AST nodes
  - ast_relation/3 – AST relations (extends, modifier, …)
- The navigation information is specified by
  - ast_sub_tree/2 – argument names of child references
  - ast_ref_tree /2 – argument names of other references
  - ast_argname_parent /2 – argument name of the parent reference
- These definitions are not hard-coded but can be provided incrementally for each new language to be supported by StarTransformer
  - They are „multifile predicates“
- For full description see

  https://sewiki.iai.uni-bonn.de/research/jtransformer/api/meta/meta-model/astspecification

Navigation in the AST
> Sample Meta-Model

Legend: parent reference
child reference
other reference

AST of a Java program



Its Prolog fact representation

```
package(1, 0, 'demo')

class(2, 1, 'C')

method(3, 2, 'm', int,[])

block(5, 3, [6])

call(6, 5, null, 3 )
```

Its meta-model

AST references (navigation info):

```
ast_argname_parent(,Java', parent).
ast_sub_tree(,Java', body).
ast_sub_tree(,Java', recv).
ast_ref_tree(,Java', called).
...
```

AST nodes:

```
ast_node_def(,Java', block, [
    ast_arg( id,        …, …, … ),
    ast_arg( parent,    …, …, …),
    ast_arg( body,      …, …, …)
]).
ast_node_def(,Java', call, [
    ast_arg( id,        …, …, …),
    ast_arg( parent,    …, …, …),
    ast_arg( recv,      …, …, …),
    ast_arg( called,    …, …, [method] )
]).
...
```

Meta-Programming

- Analysing and Manipulating Terms -

functor/3

arg/3

=../2

copy_term/2

term_variables/2

Meta-Programming

- Meta-Programming
  - Programming based on meta-data
- Meta-Progamming in Prolog
  - Data = Terms
  - Metadata = Terms
  - Meta-Programming = Programming based on analyzing and manipulating terms

functor/3

- functor(+Term, ?Functor, ?Arity)
- functor(?Term, +Functor, +Arity)
  - True if Term is a term with functor Functor and arity Arity.
  - If Term is a variable it is unified with a new term holding only variables.
  - If Term is an atom or number, Functor will be unified with Term and arity will be unified with the integer 0 (zero)
- Example: functor(+, -, -) – Analysing a term

```
?- functor(packageT(a,Y), Functor,Arity).
Functor = packageT,
Arity= 2 ;
```

- Example: functor(-, +, +) – Constructing a term template

```
?- functor(Template, packageT, 2).
Template = packageT(_G100,_G101)
```

- Example: functor(+, +, +) – Checking a term's structure

```
?- functor(packageT(a,Y), packageT, 2).
true.
```

functor/3: Practical use

- Example
  - Use the `ast_node_def/3` meta-data introduced before to find out the most general terms describing AST nodes of a language

```
ast_node_template(Language, NodeType, Arity, Template) :-
   ast_node_def(Language, NodeType, ArgList),  // as explained before
   length(ArgList,Arity),
   functor(Template,NodeType,Arity).
```

```
?- ast_node_template('Java', NodeType, Arity, Template).
...
NodeType= packageT,
Arity= 2 ;
Template= packageT(_G1042, _G1043) ;
...
```

  - Use the derived templates to find all current facts describing AST elements:

```
ast_element(Language, Element) :-
   ast_node_template(Language, _, _, Element),  // Element template
   call(Element).            // Real element = Instantiated template
```

arg/3

- arg(?ArgNumber, +Term, ?Value)
  - Value is unified with the ArgNumber-th argument of Term
  - If ArgNumber is free the predicate backtracks from left to right over all arguments
- Example: Getting the third subterm of the second subterm of the term `ast_arg(name, mult(1, 1, no), attr, [atom])`
  - With unification:

```
?- ArgumentDescr = ast_arg(name, mult(1, 1, no), attr, [atom]),
   ArgumentDescr = ast_arg(_, mult(_,_,Ordered), _, _).

ArgumentDescr = ast_arg(name, mult(1, 1, no), attr, [atom]),
Ordered = no .
```

  - Note: Unification only works if one knows the functors and arities of a term and of all its traversed subterms
  - But what if we don't? ☹ What if they change? ☹

arg/3

- arg(?ArgNumber, +Term, ?Value)
  - Value is unified with the ArgNumber-th argument of Term
  - If ArgNumber is free the predicate backtracks from left to right over all arguments
- Example: Getting the third subterm of the second subterm of the term `ast_arg(name, mult(1, 1, no), attr, [atom])`
  - With arg/3:

```
?- ArgumentDescr = ast_arg(name, mult(1, 1, no), attr, [atom]),
   arg(2, ArgumentDescr, Multiplicity),
   arg(3, Multiplicity, Ordered).

ArgumentDescr = ast_arg(name, mult(1, 1, no), attr, [atom]),
Multiplicity = mult(1, 1, no),
Ordered = no .
```

  - Note: arg/3 makes programs insensitive to change of functors and arity ☺

arg/3: Practical use

- Example
  - Determine the `ArgumentName` associated to some `ArgumentNumber` of AST nodes of type `NodeType` in the logic-based representation of the language `Language`.

```
/**
 * ast_arg_nr_name(?Language, ?NodeType, ?ArgumentNumber, ?ArgumentName)
 *
 *  In nodes of type arg2 in the language arg1 the argument with
 *  number arg3 has the name arg4.
 */
ast_arg_nr_name(Language, NodeType, ArgumentNumber, ArgumentName) :-
   ast_node_def(Language, NodeType, ArgDescrList),      // backtracks
   nth1(ArgumentNumber, ArgDescrList, ArgumentDescr),   // backtracks
   arg(1, ArgumentDescr, ArgumentName).
```

```
?- ast_arg_nr_name('Java', packageT, ArgumentNumber, ArgumentName).
ArgumentNumber = 1,
ArgumentName = id
;
ArgumentNumber = 2,
ArgumentName = name
false.
```

```
?- ast_arg_nr_name('Java', classT, ArgNr, parent).
ArgNr = 2
true.
```

What if…

- … we need all arguments of a term?
  - We cannot use arg/3, which only gives us one argument at a time ☹
- … we do not know the number of arguments (= term arity)?
  - We cannot use unification, which assumes we know the arity ☹

➢ Use the „univ“ operator (next slide)

+Term =.. ?List
?Term =.. +List

- The first element of List is the functor of Term and the remaining elements are the arguments of Term.
  - This predicate is called 'Univ'.
- Examples

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

```
?- Term =.. [baz, foo(1)]
Term = baz(foo(1))
```

- Application
  - Use 'univ' when you need to intercept and manipulate the entire argument list of a goal (often for constructing a modified version of the goal)
  - Scenario: "Replace a goal by a renamed version with an additional argument added in front of the others."

```
modified_goal(Goal,Suffix,Self,NewGoal) :-
   Goal =.. [Pred | Args],                 // split goal
   atom_concat(Pred, Suffix, NewPred),     // predicate renaming
   NewGoal =.. [NewPred, Self | Args].     // assemble new goal
```

Dealing with Variables

- copy_term(+In, ?Out)
  - Out is a copy of In with renamed (fresh) variables
  - Can deal with infinite trees (cyclic terms)
  - Ground sub-terms are shared between In and Out

```
?- copy_term(f(a,X), Out).
Out = f(a,_G9467).
```

```
?- copy_term(f(a,X), X).
X = f(a,_G9443).
```

- term_variables(+Term, ?List)
  - List is a list of new variables, each unified with a unique variable of Term.
  - The variables in List are ordered in order of appearance traversing Term depth-first and left-to-right

```
?- term_variables(a(A, b(B, A), C), Vars).

A = _G367,
B = _G366,
C = _G371,
Vars = [_G367, _G366, _G371].
```

Putting it all Together

Combining the previous predicates one can achieve generic access to the parent of any program element in any language if just know its ID.

This is what we want:

```
get_parent(ID, ParentId) :-
    ast_node_for_id(ID, Term),
    ast_parent_for_term(Term, ParentId).
```

Remaining tasks:

- Task 1: determine the PEF for any ID

```
ast_node_for_id(ID, Term)
```

- Task 2: determine the parent argument value from the PEF

```
ast_parent_for_term(Term, ParentId)
```

Putting it all Together

```
get_parent(ID, ParentId) :-
    ast_node_for_id(ID, Term),
    ast_parent_for_term(Term, ParentId).
```

- Task 1: determine the PEF for any ID
  - Let's assume that node_type(+ID, ?Type) gives us the type of the AST node of whose identity is ID.
  - Then we can implement `ast_node_for_id(Id, Term)` as

```
ast_node_for_id(Id, Template) :-
    node_type(Id,NodeType),
    ast_node_template(Language, NodeType, _, Template),
    arg(1, Template, Id),
    call(Template).
```

- Task 2: Determine the parent argument value from the PEF

```
ast_parent_for_term(Language, Term, ParentId) :-
    functor(Term,NodeType,_),
    ast_argname_parent(Language, ParentName),
    ast_arg_nr_name(Language, NodeType, ArgNr, ParentName),
    arg(ArgNr, Term, ParentId).
```

- Think! How can you eliminate the hard coded assumption that the id is the first argument of a PEF? Tip: The solution is on this page!

Reflection

call/1

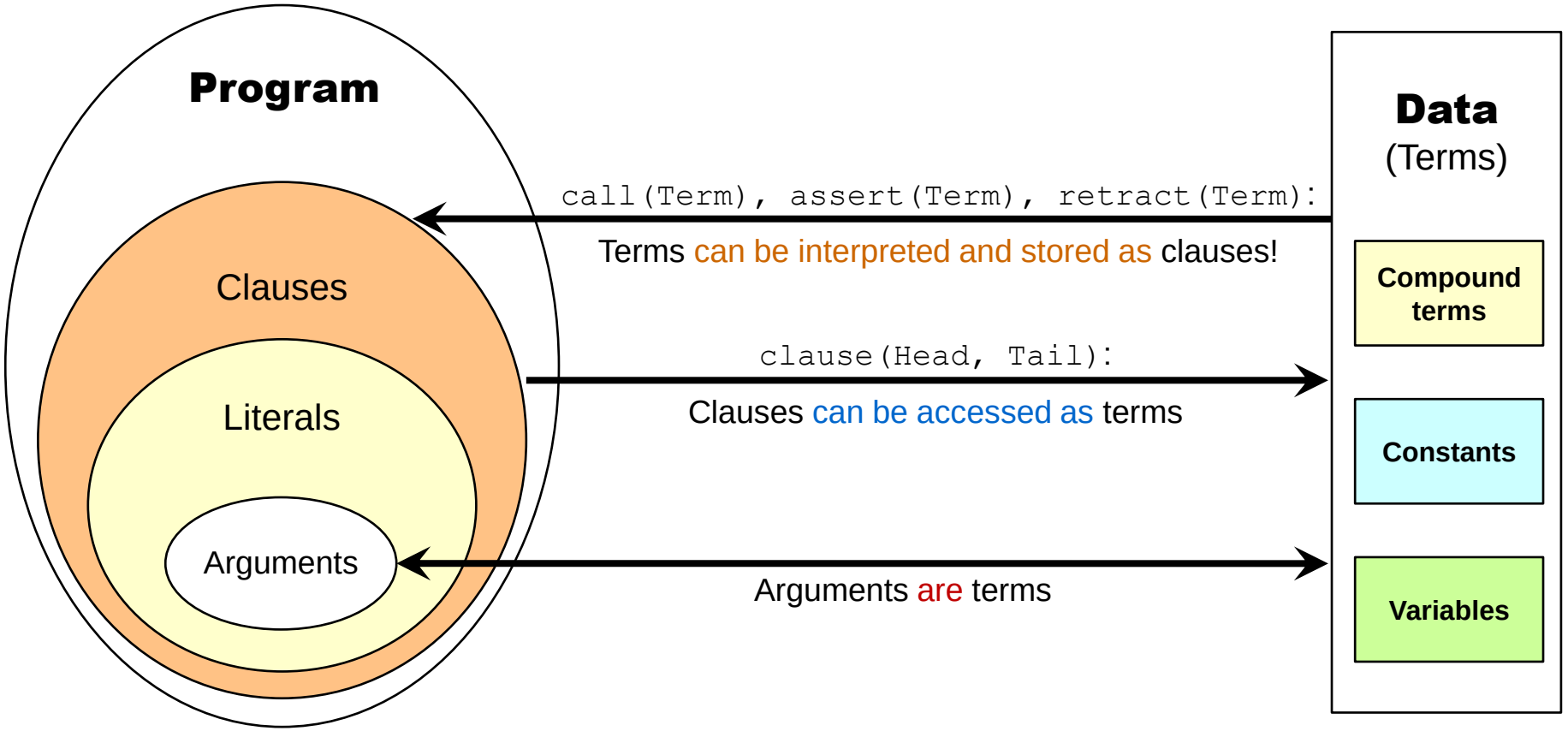
clause/2

assert/2

retract/1

Reflection

- Interpreting program elements as data and data as program elements



- By manipulating data (terms) we can manipulate program elements

Metaprogramming: Linking the world of clauses and terms

- call(Term)
  - Term is interpreted as a goal whose execution is started immediately
- clause(+Head,?Body)
  - Body is unified with the term that represents the body of the clause whose head unifies with Head
- assert(Head)
  - „Head“ is interpreted as a fact
  - ... that is added after the other clauses of the respective predicate
- assert(Head :- Body)
  - „Head :- Body“ is interpreted as the representation of a clause
  - ... that is added after the other clauses of the respective predicate
- retract(Head)
  - The first clause whose head unifies with Head is deleted.
  - Upon backtracking, the next such clause is deleted.
  - The predicate fails if there is no clause (left) whose head unifies.

Metaprogramming: A small task

- maplist(BinaryPred,List1,List2)
  - Predefined predicate that behaves like

For all i = 1..length of List1:
IF E1 is the i-th element of List1
AND E2 is the i-th element of List2
call BinaryPred(E1,E2)

- Challenge
  - Use „call(Head)“ and „Term =.. List“ to implement „maplist/3“

Meta-Predicates

Definition

- Meta-predicates are predicates that interprete some of their arguments as clauses (goals, facts or rules)

Examples

- See the previous 2 pages

Use

- Meta-predicates match, query, and manipulate other predicates.
- Through meta-predicates a Prolog program can analyze, interprete and transform other Prolog programs.
- It can even learn – by transforming itself in the course of its execution.

Metainterpreters

Meta-interpreters

- Programs can be input data for other programs.
  - Prolog programs are sequences of Prolog clauses, which can be accessed as Prolog terms (via the clause/2 predicate)

- A meta-interpreter uses programs as the data for its computations.
  - In the sequel we will discuss several Prolog meta-interpreters that reflect or modify the resolution of Prolog goals.

Basic Prolog Meta-Interpreter

- The Meta-Program: A Basic Prolog-Interpreter in Prolog
  - Treats goals as data: arguments
  - Treats clauses as data: clause/2 meta-predicate

```
solve(true).

solve((G,R)) :- solve(G),
                solve(R).

solve(G) :- clause(G,Body),
            solve(Body).
```

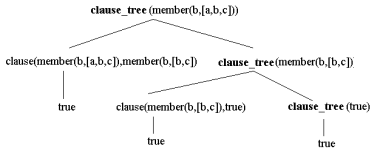
- A program to interprete:

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

- A query to solve:

```
?- solve( member(E,[a,b,c]) ).
E = a ;
E = b ;
E = c ;
fail.
```

Basic Prolog Meta-Interpreter



- First successful derivation:

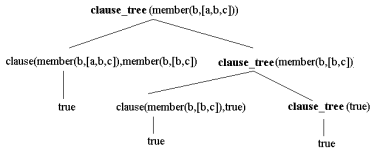
```
solve(member(E,[a,b,c]))
```

```
solve(true).

solve((G,R)) :- solve(G),
                solve(R).

solve(G) :- clause(G,Body),
            solve(Body).
```

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

```
?- solve( member(E,[a,b,c]) ).
E = a ;
E = b ;
E = c ;
fail.
```

Basic Prolog Meta-Interpreter



- First successful derivation:

solve(member(E,[a,b,c]))

Prolog :
{ G1←member(E,[a,b,c]) }

clause(member(E,[a,b,c]),Body1),solve(Body1)

clause/2 :
{ X1←E, X1←a, E←a, Body1←true }

solve(true)

Prolog :
{ }

true

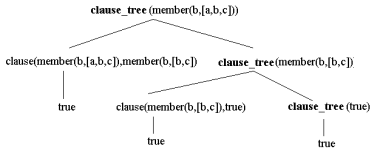
Prolog: Report substituions for variables of initial goal: E←a

```
solve(true).

solve((G,R)) :- solve(G),
                solve(R).

solve(G) :- clause(G,Body),
            solve(Body).
```

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

```
?- solve( member(E,[a,b,c]) ).
E = a ;
E = b ;
E = c ;
fail.
```

Basic Prolog Meta-Interpreter



- First successful derivation:

```
solve(member(E,[a,b,c]))
```

Prolog :
{ G1←member(E,[a,b,c]) }

```
clause(member(E,[a,b,c]),Body1),solve(Body1)
```

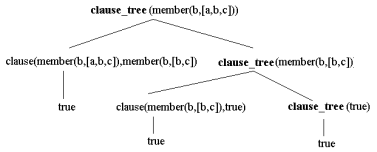
clause/2 :
{ X1←E, R1←[b,c], Body1←member(X2,R2) }

```
solve(true).

solve((G,R)) :- solve(G),
                solve(R).

solve(G) :- clause(G,Body),
            solve(Body).
```

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

```
?- solve( member(E,[a,b,c]) ).
E = a ;
E = b ;
E = c ;
fail.
```

Basic Prolog Meta-Interpreter



- Second successful derivation:

```
solve(true).

solve((G,R)) :- solve(G),
                solve(R).

solve(G) :- clause(G,Body),
            solve(Body).
```

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

```
?- solve( member(E,[a,b,c]) ).
E = a ;
E = b ;
E = c ;
fail.
```

solve(member(E,[a,b,c]))

Prolog :
{ G1←member(E,[a,b,c]) }

clause(member(E,[a,b,c]),Body1),solve(Body1)

clause/2 :
{ X2←E, R2←[b,c], Body1←member(X2,R2) }

solve(member(E,[b,c]))

Prolog :
{ G2←member(E,[b,c]) }

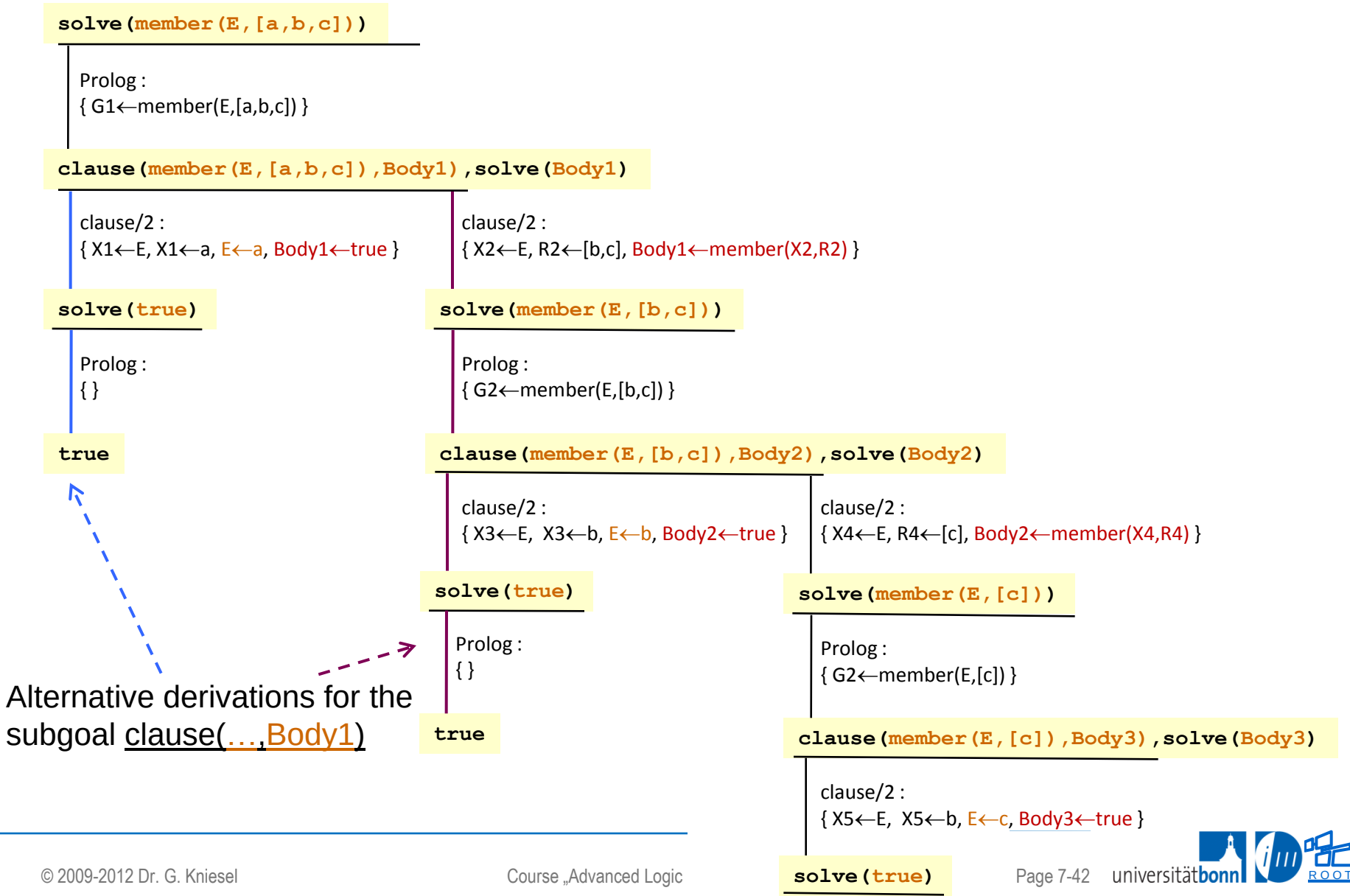
clause(member(E,[b,c]),Body2),solve(Body2)

clause/2 :
{ X3←E, X3←b, E←b, Body2←true }

solve(true)

… as before … **Report substitutions for variables of initial goal: E←b**

All Three Successful Derivations



Enhancements: Evaluation of built-ins

Enhance our metainterpreter

- Add evaluation of built-in predicates:

If the predicate for the goal G is predefined ("built-in"), delegate its evaluation to the Prolog interpreter

```
solve(true).
solve((G,R)) :-
   solve(G),
   solve(R).
solve(G) :-
   clause(G,Body),
   solve(Body).
solve(G) :-
   predicate_property(G,built_in),
   call(G).      % let Prolog do it
```

- Try it

```
?- Goal = (X = 3, X <  5),
   solve(Goal).
X = 3.
```

Enhancements: Incorporating disjunction

Conjunction

Disjunction

Enhance our metainterpreter

- Disjunction:

  Mimics conjunction

```
solve(true).
solve((G,R)) :-
   solve(G),
   solve(R).
solve((G;R)) :-
   solve(G);
   solve(R).
solve(G) :-
   clause(G,Body),
   solve(Body).
solve(G) :-
   predicate_property(G,built_in),
   call(G).      % let Prolog do it
```

- Try it

```
?- Goal = (X = 3 ; X = 4),
   solve(Goal).
X = 3 ;
X = 4 ;
fail.
```

Non-Terminating Derivations

- The evaluation strategy of Prolog is incomplete.
  - Because of non-terminating derivations, Prolog sometimes only derives a subset of the logical consequences.

```
p :- q.    % 1
q :- p.    % 2
p :- r.    % 3
r.         % 4
```

- Example
  - `r`, `p`, and `q` are logical consequences of this program:
  - However, Prolog's evaluation strategy cannot derive them. It loops indefinitely:

```
?- p.
   |---   1st clause
   q
   |---   2nd clause
   p
  ...      etc.
```

- Note
  - Theoretical limitation: There is no static loop-detecting algorithm that would succeed in detecting all loops. If there were, one would have solved the halting problem.
  - For the sake of efficiency, Prolog does not try any detection of derivation loops.

Derivation with Loop Checking

- Try to detect some loops
  - remember derivation path in additional parameter
  - abort cyclic derivations

```
solve(true,Path).
solve((G,R),Path) :-
   solve(G,Path),
   solve(R,Path).
solve(G,Path) :-
   not( loop(G,Path) ),
   clause(G,Body),
   solve(Body,[G|Path]).

loop(G,[First|_]) :- G == First.
loop(G,[_|Rest] ) :- loop(G,Rest).
```

- Try it out

```
?- solve(p,[]).
true.
```

```
p :- q.    % 1
q :- p.    % 2
p :- r.    % 3
r.         % 4
```

- What happens
  - Derivation of clause 2 and 1 fails
  - Derivation continues at clause 3

```
?- p.
   |--- 1st clause
   q
   |--- 2nd clause
   p
   |
   fail (loop detected)
```

```
p.
|--- 3rd clause
r
|--- 4th clause
true
```

A Simple Tracer: Building a Clause Tree

- Also generate a clause tree parameter while interpreting:

```
solve(true,_,true) .
solve((G,R),Trail,(TG,TR)) :-
   solve(G,Trail,TG),
   solve(R,Trail,TR).
solve(G,_,prolog(G)) :-
   predicate_property(G,built_in),
   call(G).
solve(G,Trail,tree(G,T)) :-
   not( loop(G,Trail) ),
   clause(G,Body),
   solve(Body,[G|Trail],T).
```

- Example program

```
p(X) :- q(X), r(Y), X < Y.
q(3).
r(2).
r(5).
r(10).
```

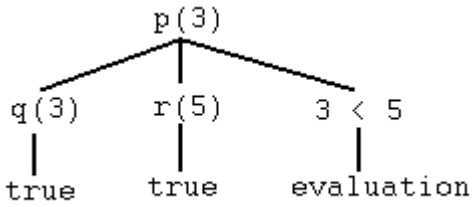
- Try it

```
?- solve(p(X),[],Tree)
Tree = tree(p(3),( tree(q(3),true),
                   tree(r(5),true),
                   prolog(3 < 5)
                 )
            ),
X = 3 ;
… continued on right-hand-side …
```

```
Tree = tree(p(3),( tree(q(3),true),
                   tree(r(10),true),
                   prolog(3 < 10)
                 )
            ),
X = 3 ;
fail.
```

A Simple Tracer: Printing the Clause Tree

```
trace(G):- solve(G,[],T),
           nl,
           draw_tree(T,5).

draw_tree(tree(Root,Branches),Tab):-
   !,
   draw_node(Root,Tab),
   Tab5 is Tab + 5,
   draw_tree(Branches,Tab5).
draw_tree((B,Bs),Tab) :-
   !,
   draw_tree(B,Tab),
   draw_tree(Bs,Tab).
draw_tree(Node,Tab) :-
   draw_node(Node,Tab).

draw_node(Node,Tab) :-
   tab(Tab),
   format('|-- ~w~n', [Node]).
```

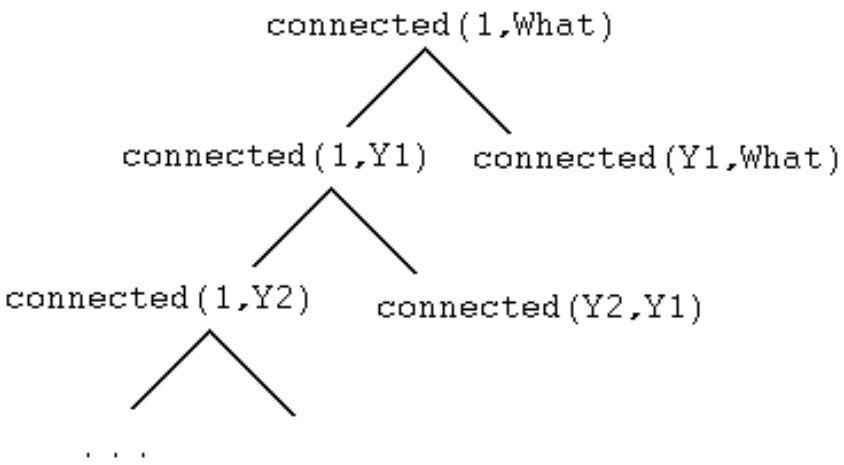



```
?- trace(p(X)).
   |-- p(3)
         |-- q(3)
               |-- true
         |-- r(5)
               |-- true
         |-- prolog(3 <  5)
X = 3 ;
   |-- p(3)
         |-- q(3)
               |-- true
         |-- r(10)
               |-- true
         |-- prolog(3 <  10)
X = 3 ;
fail.
```

Iterative deepening

- Consider following „bad“ program:

```
connected(X,Y) :- connected(X,Z),
                  connected(Z,Y).
connected(1,2).
connected(2,3).
connected(3,4).
connected(4,5).
```



- Try to compute `connected(1,2)`
  - problem with *left recursion*
  - putting the rule after the facts would prevent the problem for this goal
- Force `connected(1,What)` to backtrack to try to find all solutions
  - looping for goal that is not identical but “equivalent” to a previous goal
  - the **same clause** is repeatedly used
  - the rule does not know how many links might be between '1' and 'What',
    - ⇨ this rule is (all by itself) trying to allow for there being 1 link, 2 links, 3 links, ...
    - ⇨ sometimes referred to as *Prediction loop*

Iterative deepening

- … is a method for avoiding this kind of infinite descent.
  - Search to a certain depth, then deeper, then deeper, …
  - Still depth-first search

```
iterative_deepening(G,D) :- solve(G,0,D).
iterative_deepening(G,D) :-
     write('limit='), write(D), write('(Hit Enter to Continue.)'),
     get(CharCode),
     ( CharCode == 10
        -> D1 is D + 5,
           iterative_deepening(G,D1)
     ).
```

2 additional variables
- current depth of goal
- current depth limit

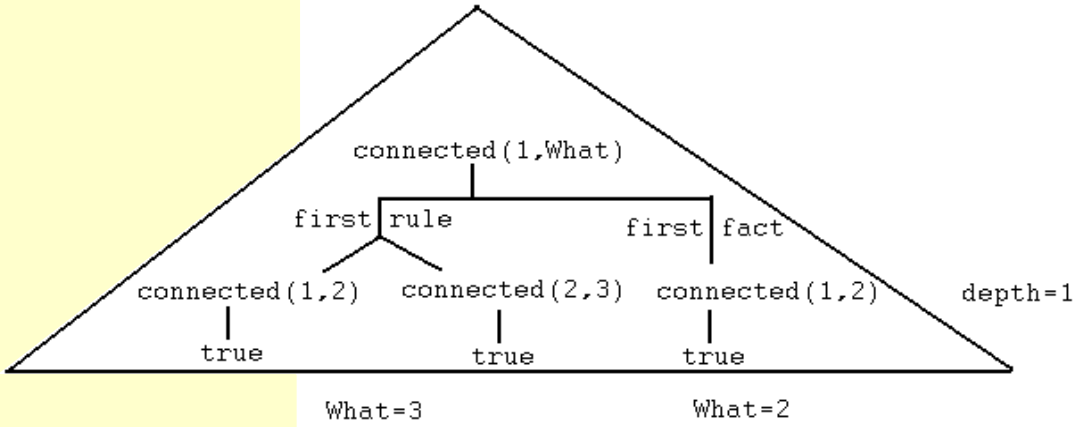
iterative_deepening(G,D)
- G can contain variables
- depth of 1st stage is D
- depth of 2nd stage is D+5,…

```
solve(true,_,_)       :- !.
solve(_,D,Limit)      :- D > Limit, !, fail.  %% reached depth limit
solve((A,B),D,Limit)  :- !,
                         solve(A,D,Limit),
                         solve(B,D,Limit).
solve(A,D,Limit)      :- clause(A,B),
                         D1 is D+1,
                         solve(B,D1,Limit).
```

Iterative deepening

- Use iterative deepening for `connect(1,What)`

```
?- iterative_deepening(connected(1,What),1).
What=3 ;
What=2 ;
Limit=1(Hit Enter to Continue)
What=5 ;
What=5 ;
What=5 ;
What=4 ;
What=5 ;
What=5 ;
What=4 ;
What=3 ;
What=2 ;
Limit=6(Hit Enter to Continue.)
What=5                              %% stop
true.
```



- Solutions that are near the current depth limit come first
- Then Prolog proceeds to discover shallower solutions

Iterative deepening

- Theoretically, iterative deepening has a kind of optimal behavior for "blind" search:
  - Iterative deepening will find any possible solution
  - In a stage deep enough to include the clause tree justifying that solution
    - ⇨ space O(d), d=depth
    - ⇨ time O(b**d), b= average branching factor
  - Other kinds of complete search, such as breadth-first (or iterative broadening), have to search through a larger expected number of nodes.

Partial Evaluation

Idea:

- Precompile Prolog clauses by evaluating what can already be evaluated at compile time
- Defer the parts that cannot be evaluated → “residuals”
- Assert the residuals as new, renamed clauses
- Call the new predicates instead of the original ones

Applications

- Eliminate clauses that do not match a query
- Precompute acces to immutable meta-data

```
ast_arg_nr_name(Language, NodeType, ArgumentNumber, ArgumentName) :-
   ast_node_def(Language, NodeType, ArgDescrList),     // static metadata
   nth1(ArgumentNumber, ArgDescrList, ArgumentDescr),
   arg(1, ArgumentDescr, ArgumentName).
```

Summary: Metaprogramming

- Meta-Data
  - Clauses can easily represent meta-information
    - ⇨ structure
    - ⇨ constraints
    - ⇨ rules
    - ⇨ …
  - Meta-information makes implicit knowledge explicit

- Meta-Programs
  - Use meta-data to capture an entire family of related application scenarios
  - Term manipulation predicates enable meta-programming
    - ⇨ functor/3
    - ⇨ arg/3
    - ⇨ =../2
    - ⇨ …

Summary: Reflection

- Reflection is meta-programming that
  - Uses the Prolog program as meta-data about itself
  - Manipulates the Prolog program
- Meta-Interpreters
  - Allow easy implementation of various operational semantics
  - Quick language prototyping
  - Aspect-Orientation, the Prolog way