Chapter 5. Metadata, Metaprogramming and Reflection

July 2, 2010
Chapter 5: Metadata, Metaprogramming and Reflection

Meta-Data and Meta-Programming

Meta-Data
Scenario: From JTransformer to StarTransformer – Language independent PEF traversal

Meta-Programming
Basic prolog predicates and their use in our scenario
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 structure of other clauses

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

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

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

  \begin{align*}
  \text{ast_node_def('Java', \ call, [} \\
  & \text{ast_arg( id, \ mult(1,1,no ), \ id, \ [call] \ ),} \\
  & \text{ast_arg( parent, \ mult(1,1,no ), \ id, \ [block] \ ),} \\
  & \text{ast_arg( recv, \ mult(0,1,no ), \ id, \ [...] \ ),} \\
  & \text{ast_arg( called, \ mult(1,1,no ), \ id, \ [method] )} \\
  \text{])}. 
  \end{align*}

  \begin{itemize}
  \item \textbf{argument name} \texttt{id}, \texttt{parent}, \texttt{recv}, \texttt{called}.
  \item \textbf{multiplicity} \texttt{mult(1,1,no)}.
  \item \textbf{order} \texttt{no = not ordered, \ ord = ordered}.
  \item \textbf{kind of value} \texttt{id = identity, \ attr = primitive}.
  \item \textbf{legal syntactic type(s) of argument values} \texttt{[call], \ [block], \ [...]}. 
  \item \textbf{specification of PET arguments}.
  \end{itemize}
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

```prolog
ast_node_def(
    'Java', % The AST of a simplified 'Java' version ...
    call,   % ... contains nodes of type 'call' ...
    [      % ... that have 4 arguments:
        ast_arg(id, mult(1,1,no ), id, [id]),
        ast_arg(parent, mult(1,1,no ), id, [id]),
        ast_arg(receiver, mult(0,1,no ), id, [expressionType,nullType]),
        ast_arg(ref, mult(1,1,no ), id, [methodDefT])
    ]
).
```
Argument Descriptors (1)

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

<table>
<thead>
<tr>
<th>Cardinality</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{mult(0,*},no)</td>
<td>Any cardinality including 0, no order.</td>
</tr>
<tr>
<td>\texttt{mult(0,*},ord)</td>
<td>Any cardinality including 0, the values are ordered. In JTransformer such argument are lists.</td>
</tr>
<tr>
<td>\texttt{mult(1,2},no)</td>
<td>A cardinality range with a lower and upper bound. Not ordered.</td>
</tr>
<tr>
<td>\texttt{mult(0,1},no)</td>
<td>An optional, single-valued argument - may be the atom 'null'. Clearly not ordered.</td>
</tr>
</tbody>
</table>
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

- **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 $L$.

```prolog
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
  [http://sewiki.iai.uni-bonn.de/research/jtransformer/astspecification](http://sewiki.iai.uni-bonn.de/research/jtransformer/astspecification)
Navigation in the AST

> Sample Meta-Model

**AST of a Java program**

- **package**
  - name: demo
  - parent:

- **class**
  - name: C
  - parent:

- **method**
  - name: m
  - type: int
  - parent:

- **block**
  - parent:

- **call**
  - name: m
  - parent:

**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] )]).`

© 2009, 2010 Dr. G. Kniesel

Course „Advanced Logic Programming“ (ALP)
Chapter 5: Metadata, Metaprogramming and Reflection

From Meta-Data to 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-Programming 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(Langauge, 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 \texttt{ast_arg(name, mult(1, 1, no), attr, [atom])}
  - With arg/3:

  ```prolog
  ?- 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`.

```prolog
/**
 * ast_arg_nr_name(?Language, ?NodeType, ?ArgumentNumber, ?ArgumentName)
 */
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)
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.”

```prolog
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**

\[
\text{?- copy_term}(f(a,X), \text{Out}).
\]
\[
\text{Out} = f(a,_G9467).
\]

\[
\text{?- copy_term}(f(a,X), X).
\]
\[
X = f(a,_G9443).
\]

- **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

\[
\text{?- term_variables}(a(A, b(B, A), C), Vars).
\]
\[
A = _G367,
B = _G366,
C = _G371,
Vars = [_G367, _G366, _G371].
\]
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:

\[
\text{ast_parent_for_id}(ID, ParentId) :- \\
\text{ast_node_for_id}(ID, Term), \\
\text{ast_parent_for_term}(Term, ParentId).
\]

Remaining tasks:

- Task 1: determine the PEF for any ID
  \[
  \text{ast_node_for_id}(ID, Term)
  \]
- Task 2: determine the parent argument value from the PEF
  \[
  \text{ast_parent_for_term}(Term, ParentId)
  \]
Putting it all Together

- **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

    ```prolog
    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

    ```prolog
    ast_parent(Language, Term, ParentId) :-
        functor(Term, NodeType, _),
        ast_argname_parent(Language, ParentName),
        ast_arg_nr_name(Language, NodeType, ArgNr, ParentName),
        arg(ArgNr, Term, ParentId).
    ```
Reflection: Metainterpreters
Reflection

- Interpreting program elements as data and data as program elements

By manipulating data (terms) we can manipulate program elements

- Terms can be interpreted and stored as clauses!

- Clauses can be accessed as terms

- Arguments are terms

- Compound terms
- Constants
- Variables

© 2009, 2010 Dr. G. Kniesel Course „Advanced Logic Programming“ (ALP)
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.
Chapter 5: Metadata, Metaprogramming and Reflection

Metaintepreters
Meta-interpreters

- Programs can also be considered as input data for other programs.
- Prolog programs are sequences of prolog terms, so prolog programs easily serve as input data.
- A prolog meta-interpreter uses program data as a basis for additional computations.
- We will discuss several prolog meta-interpreters that modify the computation 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: \texttt{clause/2} meta-predicate

- A program to interpret:

- A query to solve:

\[
\text{solve}(\texttt{true}).
\]
\[
\text{solve}(\texttt{(G,R)}) :- \text{solve}(\texttt{G}), \text{solve}(\texttt{R}).
\]
\[
\text{solve}(\texttt{G}) :- \text{clause}(\texttt{G}, \texttt{Body}), \text{solve}(\texttt{Body}).
\]
\[
\text{member}(X, [X|\_]).
\]
\[
\text{member}(X, [\_|R]) :- \text{member}(X, R).
\]
\[
? - \text{solve}(\texttt{member(E,[a,b,c])}).
\]
\[
E = a ;
E = b ;
E = c ;
\text{fail}.
\]
First successful derivation:

\[
\text{solve}(\text{true}).
\]

\[
\text{solve}(\text{G}, \text{R}) :\text{ solve}(\text{G}), \text{ solve} (\text{R}).
\]

\[
\text{solve}(\text{G}) :\text{ clause}(\text{G}, \text{Body}), \text{ solve} (\text{Body}).
\]

\[
\text{member}(\text{X}, [\text{X}|\_]).
\]

\[
\text{member}(\text{X}, [\_|\text{R}]) :\text{ member}(\text{X}, \text{R}).
\]

\[
?- \text{ solve}(\text{member}(\text{E}, [\text{a}, \text{b}, \text{c}])).
\]

\[
\begin{align*}
\text{E} = \text{a} ; \\
\text{E} = \text{b} ; \\
\text{E} = \text{c} ; \\
\text{fail}.
\end{align*}
\]
First successful derivation:

```prolog
solve(member(E,[a,b,c]))
```
First successful derivation:

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

Prolog: { G1←member(E,[a,b,c]) }

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

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

```prolog
member(X,[X|_]).
member(X,[_|R])← member(X,R).

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

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

Prolog:

```
\{ G1 \leftarrow member(E,[a,b,c]) \}
```

```
clause(member(E,[a,b,c]),Body1),solve(Body1)
```
```
clause/2 : 
\{ X2 \leftarrow E, R2 \leftarrow [b,c], Body1 \leftarrow member(X2,R2) \}
```

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

Prolog:

```
\{ G2 \leftarrow member(E,[b,c]) \}
```

```
clause(member(E,[b,c]),Body2),solve(Body2)
```
```
clause/2 : 
\{ X3 \leftarrow E, X3 \leftarrow b, E \leftarrow b, Body2 \leftarrow true \}
```

```
solve(true)
```

```
\ldots as before \ldots \quad \text{Report substitutions for variables of initial goal: } E \leftarrow b
```

```
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.
```
All Three Successful Derivations

The derivations shown below are for solving the Prolog subgoal `member(E,[a,b,c])`. Each derivation is represented by a tree structure, with `solve` and `clause` nodes indicating the action taken.

1. **First Derivation**
   - `solve(member(E,[a,b,c]))`
   - Prolog: `{ G1←member(E,[a,b,c]) }
   - Clause: `clause(member(E,[a,b,c]),Body1),solve(Body1)`
   - `clause/2:` `{ X1←E, X1←a, E←a, Body1←true }`

2. **Second Derivation**
   - `solve(true)`
   - Prolog: `{ }`

3. **Third Derivation**
   - `solve(member(E,[b,c]))`
   - Prolog: `{ G2←member(E,[b,c]) }
   - Clause: `clause(member(E,[b,c]),Body2),solve(Body2)`
   - `clause/2:` `{ X2←E, R2←[b,c], Body1←member(X2,R2) }`

Alternative derivations for the subgoal `clause(...,Body1)`: The other two derivations follow similar structures, with slight variations in the clauses and conditions to achieve the same goal of solving `member(E,[a,b,c])`. Each step involves applying Prolog's resolution principle to reduce the subgoals until `true` is reached.
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"), call the underlying Prolog interpreter to do the evaluation

- Try it out

```prolog
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
```

?− Goal = (X = 3, X < 5),
   solve(Goal).
   X = 3.
Enhancements: Incorporating disjunction

Enhance our metainterpreter

- Disjunction: Mimics conjunction

\[
\begin{align*}
\text{solve}(true). \\
\text{solve}(G,R) :- \\
\quad \text{solve}(G), \\
\quad \text{solve}(R). \\
\text{solve}(G) :- \\
\quad \text{clause}(G,Body), \\
\quad \text{solve}(Body). \\
\text{solve}(G) :- \\
\quad \text{predicate_property}(G,built_in), \\
\quad \text{call}(G). \quad \% \text{let Prolog do it}
\end{align*}
\]

- Try it out

\[
?- \text{Goal} = (X = 3 \ ; \ X = 4), \\
\quad \text{solve}(\text{Goal}). \\
X = 3 ; \\
X = 4 ; \\
\text{fail}.
\]

© 2009, 2010 Dr. G. Kniesel
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.

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

- 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

```prolog
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

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

- What happens
  - Derivation of clause 2 and 1 fails
  - Derivation continues at clause 3
A Simple Tracer: Building a Clause Tree

- Also generate a clause tree parameter while interpreting:

Example program

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

Try it

```prolog
?- 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 ...
```

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

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).

© 2009, 2010 Dr. G. Kniesel Course „Advanced Logic Programming“ (ALP)
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

```prolog
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)
    ).

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).
```

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,...
Iterative deepening

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

```
?- iterative_deepening(connect(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 $O(d)$, $d=\text{depth}$
    - time $O(b^d)$, $b=\text{average branching factor}$
  - 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 \(\rightarrow\) “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 access to immutable meta-data
- …

\[
\text{ast_arg_nr_name}(	ext{Language}, \text{NodeType}, \text{ArgumentNumber}, \text{ArgumentName}) :-
\text{ast_node_def}(	ext{Language}, \text{NodeType}, \text{ArgDescrList}), \quad // \text{metadata}
\text{nth1}(	ext{ArgumentNumber}, \text{ArgDescrList}, \text{ArgumentDescr}),
\text{arg}(1, \text{ArgumentDescr}, \text{ArgumentName}).
\]
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