Chapter 7. Metadata, Metaprogramming and Reflection

Updated: July 6, 2016

- Deleted / hided irrelevant slides, added new ones

Meta-Data
Meta-Programming
Computational Reflection
Meta-Interpreters

Not relevant for exam
Chapter 7: Metadata, Metaprogramming and Reflection

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, [1])
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: `call(6, 5, null, 3 )`
- Sample PEF structure: `call(id#, parent#, recv#, called#)`
- Specified by:

```
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] )
]).
```

- **Language**: PEF type (= AST node type)
- **PEF type**
  - `call`
- **Arguments**
  - `id`
  - `parent`
  - `recv`
  - `called`

- **Argument Name**
- **Multiplicity**
  - `mult(1,1,no)`
- **Order**
  - `no` = not ordered
  - `ord` = ordered
- **Kind of Value**
  - `id` = identity
  - `attr` = primitive
- **Legal Syntactic Type(s)**
  - `call`, `block`, `method`
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](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)`

<table>
<thead>
<tr>
<th>Cardinality</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mult(0,*,no)</code></td>
<td>Any cardinality including 0, no order.</td>
</tr>
<tr>
<td><code>mult(0,*,ord)</code></td>
<td>Any cardinality including 0, the values are ordered. In JTransformer such argument are lists.</td>
</tr>
<tr>
<td><code>mult(1,2,no)</code></td>
<td>A cardinality range with a lower and upper bound. Not ordered.</td>
</tr>
<tr>
<td><code>mult(0,1,no)</code></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 \( L \).

  ```
  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

Legend:
- **parent reference**
- **child reference**
- **other reference**
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

**AST of a Java program**

1. package
   - name: demo
   - parent

2. class
   - name: C
   - parent

3. method
   - name: m
   - type: int
   - parent

5. block
   - parent

6. call
   - name: m

**Legend:**
- parent reference
- child reference
- other reference

**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] ) ]).

...
How to use the meta-data?

- Query it
  - Let a program find out about the structure of Java PEFs

- Make your program independent of the Java PEF structure
  - No need to know that a class is represented by the predicate `classT(Id, Parent, Name, Members)`
  - Get the element knowing just its identity: `get(Id, Term) ← will unify Term to classT(Id, ..., ..., ...) if Id is the identity of a class`
  - Encapsulate the access to a term into accessor methods, e.g. generic `get_name(Term, Name)`

- For the above we need to be able to analyse the structure of a term
  - E.g. access just the 3rd argument (knowing that it is the name that we want)
  - Metaprogramming
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**
  ```prolog
  \?- functor(packageT(a,Y), Functor, Arity).  
  Functor = packageT,  
  Arity= 2 ;  
  ```

- **Example: functor(-, +, +) – Constructing a term template**
  ```prolog
  \?- functor(Template, packageT, 2).  
  Template = packageT(_G100, _G101)  
  ```

- **Example: functor(+, +, +) – Checking a term’s structure**
  ```prolog
  \?- 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(Lang, NodeType, Arity, Template) :-
  ast_node_def(Lang, 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 
  \texttt{ast_arg} (name, \texttt{mult(1, 1, no)}, attr, [atom])
  - With unification:

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

- 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:

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

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

?- ast_arg_nr_name('Java', packageT, ArgumentNumber, ArgumentName).
ArgumentNumber = 1,
ArgumentName = id

?- 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)
\[ +\text{Term} = \ldots \ ?\text{List} \]
\[ ?\text{Term} = \ldots +\text{List} \]

- 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

  \[
  \begin{align*}
  \text{?-} & \quad \text{foo}(\text{hello}, \, X) = \ldots \, \text{List}. \\
  \text{List} & = [\text{foo}, \, \text{hello}, \, X]
  \end{align*}
  \]

- 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.”

  \[
  \text{modified\_goal}(\text{Goal}, \text{Suffix}, \text{Self}, \text{NewGoal}) :\neg
  \quad \begin{align*}
  \text{Goal} & = \ldots \ [\text{Pred} \mid \text{Args}], \quad \% \text{split goal} \\
  \text{atom\_concat}(\text{Pred}, \text{Suffix}, \text{NewPred}), & \quad \% \text{predicate renaming} \\
  \text{NewGoal} & = \ldots \ [\text{NewPred}, \text{Self} \mid \text{Args}]. \quad \% \text{assemble new goal}
  \end{align*}
  \]
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:

\[
\text{get\_parent}(ID, \text{ParentId}) \leftarrow \\
\text{ast\_node\_for\_id}(ID, \text{Term}), \\
\text{ast\_parent\_for\_term}(\text{Term}, \text{ParentId}).
\]

Remaining tasks:

- Task 1: determine the PEF for any ID

\[
\text{ast\_node\_for\_id}(ID, \text{Term})
\]

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

\[
\text{ast\_parent\_for\_term}(\text{Term}, \text{ParentId})
\]
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, Term) :-
  node_type(Id, NodeType),
  ast_node_template(Language, NodeType, _, Term),
  arg(1, Term, Id),
  call(Term).
```

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

```prolog
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!
Chapter 7: Metadata, Metaprogramming and Reflection

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

**Program**

- Clauses
- Literals
- Arguments

**Data**

- Compound terms
- Constants
- Variables

**Call, Assert, Retract**

\[ \text{call(Term), assert(Term), retract(Term):} \]

Terms can be interpreted and stored as clauses!

**Clause**

\[ \text{clause(Head, Tail):} \]

Clauses can be accessed as terms

**Arguments**

Arguments are terms
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.
Meta-Predicates

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

**Examples**
- See the previous pages

**Use**
- Meta-predicates match, query, and manipulate other predicates.
- Through meta-predicates a Prolog program can analyze, interpret and transform other Prolog programs.
- It can even learn – by transforming itself in the course of its execution.
Defining Metapredicates: An Example

maplist(BinPred, List1, List2)

◆ call BinPred on each pair of elements from List1 and List2 that have the same position: For all \( i = 1 \ldots \text{length of List1} \) : BinPred(List1[i], List2[i]) ;

◆ This can be implemented on the basis of =../2 and call/2 as

```prolog
maplist(_, [], []).  
maplist(Pred, [Head| Tail], [MHead| MTail]) :-  
    Cmd =.. [Pred, Head, MHead],  
    call(Cmd),  
    maplist(Pred, Tail, MTail).
```

This can be used like this:

```prolog
add1(X, X1) :- X1 is X + 1.

% Add 1 to each element of first list to get the second one:
?- maplist(add1, [1, 2, 3], ML).
ML = [2, 3, 4]
```
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

- A program to interpret:

- A query to solve:

```prolog
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.
```
First successful derivation:

```prolog
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.
```
First successful derivation:

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

Prolog:
\[
\{ G1 \leftarrow \text{member}(E, [a, b, c]) \}
\]

\[
\text{clause}(\text{member}(E, [a, b, c]), \text{Body1}), \text{solve}(\text{Body1})
\]

Prolog:
\[
\{ X1 \leftarrow E, X1 \leftarrow \text{a}, E \leftarrow \text{a}, \text{Body1} \leftarrow \text{true} \}
\]

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

Prolog:
\[
\}
\]

true

Prolog: Report substitutions for variables of initial goal: E\leftarrow a

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

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

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

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

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

?- solve( \text{member}(E, [a, b, c]) ).
E = a ;
E = b ;
E = c ;
fail.
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:
{ X2←E, R2←[b], Body1←member(X2,R2) }

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

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

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

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

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

```prolog
solve(true).
solve(G,R) :- solve(G), solve(R).
solve(G) :- clause(G,Body), solve(Body).
```
Second successful derivation:

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

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

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

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

```prolog
solve(true)
```

Report substitutions for variables of initial goal: $E \leftarrow b$

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

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

Prolog:
\{ G1 \leftarrow \text{member}(E, [a, b, c]) \}

\[ \text{clause}(\text{member}(E, [a, b, c]), \text{Body1}), \text{solve}(\text{Body1}) \]

\[ \text{clause/2}: \{ X1 \leftarrow E, X1 \leftarrow a, E \leftarrow a, \text{Body1} \leftarrow \text{true} \} \]

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

Prolog:
\{ \}

\[ \text{true} \]

Alternative derivations for the subgoal \[ \text{clause(..., \text{Body1})} \]
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.

```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
```

- Try it

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

Enhance our metainterpreter

- Disjunction:
  Mimics conjunction

- Try it

```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 = 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.

- 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

- Try it out

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

```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).

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

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

?- fail.
fail (loop detected)
```

```
p.  % 3rd clause
    p.  % 4th clause

p.        % 1st clause
q.        % 2nd clause
p.        % 3rd clause
```
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