# A PROLOG Tool for Slicing Source Code

Marbod Hopfner, Dietmar Seipel, and Joachim Baumeister

University of Würzburg, Institute for Computer Science
Am Hubland, D – 97074 Würzburg, Germany
{hopfner, seipel, baumeister}@informatik.uni-wuerzburg.de

**Abstract.** We describe a PROLOG tool for slicing source code. We assume that there exists an XML representation of the *parse tree* of the code. Then, we can perform an analysis of the extended *call graph* based on methods from the tool VISUR/RAR to determine the relevant predicates for the slice.
User–defined *policies* reflecting the different styles of programming of different users can be plugged into the system; they are formulated in a declarative way using the XML query language FNQUERY.
We have implemented VISUR/RAR and FNQUERY as part of the DISLOG developers toolkit DDK. So far we have applied slicing to extract several subsystems of the DDK.

*Keywords.* program extraction, comprehension, refactoring, visualization

## 1 Introduction

In a large software system it often becomes difficult to keep an overview of the entire system, even if it consists of classes, modules, and methods. For debugging a system, or for porting a subsystem to another language or dialect, it is important – but sometimes very difficult – to extract the relevant entities. A *slice* for a certain target method consists of all methods of the considered software system which are necessary to correctly run the subsystem in focus [9, 10].

We have implemented a PROLOG slicing tool for extracting source code for a certain functionality from a software system. Our approach is based on the extended *predicate dependency graph* (call graph for predicates) of the source code, which can be handled using the system VISUR/RAR [6], and on some further *policies* for handling specialities of the source language. We use an XML representation of the parse tree of the considered source code.

So far we have developed an extensible collection of extraction policies for the source language SWI PROLOG, cf. [11], which are specified declaratively as PROLOG predicates using the library FNQUERY for XML [5], but we could also extract methods from source code of other languages, provided that we have an XML representation of the parse tree and corresponding slicing policies for it.

We have in mind several reasons for slicing a software system:

– We can use slicing for *debugging* and for *extracting* certain functionality of a large software system to use it in a foreign application. One can obtain an overview of the relevant methods much more easily, and one does not need to read and understand unnecessary source code.
– Although in general it is very difficult to port PROLOG systems to other PROLOG dialects, porting becomes much easier if we can focus on slices of the systems, which could have only a small fraction of the size of the entire system.
– The integration of the slice into a foreign application becomes easier, since the probability that some methods and global variables of the application conflict with methods from the slice is reduced.

The rest of this paper is organized as follows: In Section 2 we give an overview of the slicing tool. In Section 3 we present a collection of basic policies for slicing PROLOG source code. User–defined slicing policies reflecting the programming styles of further users can be added using the language FNQUERY, which we also describe shortly. If slicing is done as a first step for porting a subsystem to another language or dialect of the same language, then it often is accompanied by *refactoring*; in Section 4 we mention some types of refactorings that could be useful in this context. Finally, in Section 5 we present two case studies applying the proposed slicing approach to the toolkit DDK,

## 2 An Overview of the Slicing Tool

In this section we give an overview of the slicing tool, and we list some problems that arise when we slice PROLOG source code. Moreover, we describe the XML representation of the source code and the XML query language called FNQUERY, on which the slicing tool is based.

### 2.1 Slicing PROLOG Source Code

Usually, we slice at the *granularity* of predicates, but we can also slice at the granularity of files. The result of the slicing process is the following:

– an archive, i.e., a directory with subdirectories containing the files with the selected predicates in the same structure as in the file hierarchy of the source system, and
– an index file containing some settings and consult statements for the files in the archive.

Consulting the index file from a foreign PROLOG application causes all necessary files from the archive to be properly consulted.

The following problems arise during the slicing of PROLOG source code: Firstly, it can be very complicated to determine the relevant predicates and files from the predicate dependency graph, if there are meta–call predicates calling other predicates. Secondly, we have to handle external libraries and dynamic loadings, and the consulting order

of the files in the slice needs to be the same as in the original source code. Thirdly, some generic predicates might be defined in many files, and not all of their clauses are relevant for the slice. Finally, the handling of predicate properties (such as dynamic or multifile) and of global variables is difficult. In Section 3 we will discuss how we have solved these problems based on the XML representation of the source code.

### 2.2 Source Code in Field Notation / XML

In the following we will briefly describe the used XML representation of PROLOG source code, the corresponding PROLOG structure, which we call field notation, and the XML query language FNQUERY; a more detailed introduction to the DDK library FNQUERY can be found in [5]. E.g., we can parse a PROLOG file `inc.pl` consisting of the single clause

```
increment(X, Y) :-
   Y is 1 + X.
```

into the following XML representation:

```
<file path="inc.pl" module="user">
  <rule operator=":-">
    <head>
      <atom module="user" predicate="increment" arity="2">
        <var name="X"/> <var name="Y"/>
      </atom>
    </head>
    <body>
      <atom module="user" predicate="is" arity="2">
        <var name="Y"/>
        <term functor="+">
          <term functor="1"/> <var name="X"/>
        </term>
      </atom>
    </body>
  </rule>
</file>
```

We represent an XML element as a PROLOG term structure `T:As:C`, which we call FN triple: it consists of the tag `T`, an association list `As` for the attribute/value pairs, and a list `C` containg the subelements. E.g., the PROLOG statement `Y is 1 + X` becomes the term structure `atom:As:C` shown below. Our parser from XML to field notation is based on the XML parser of SWI PROLOG, and we derive the field notation as a slightly more compact variant of the XML data structure of SWI PROLOG.

For obtaining the representation `Code` of a PROLOG program in field notation we use the predicate `program_file_to_xml/2` – without converting to an XML file first. From the FN triple `Code` we can select a body atom with the predicate symbol `is` using the infix predicate `:=/2` of FNQUERY:

```
?- program_file_to_xml('inc.pl', Code),
   Atom := Code^rule^body^atom::[@predicate=is].

Code = ...,
Atom =
   atom:[module:user, predicate:is, arity:2]:[
      var:[name:'Y']:[],
      term:[functor:+]:[
         term:[functor:1]:[], var:[name:'X']:[]]]

Yes
```

Every component `^T` of the used path expression selects a corresponding subelement with the tag `T`, and the condition `[@predicate=is]` checks that the value of the attribute `predicate` is equal to `is`.

## 3 Basic Policies for Slicing PROLOG Source Code

For slicing a PROLOG system w.r.t. to a target predicate we have to take into account the special aspects of PROLOG. The set of *potentially relevant files* includes files that are loaded using statements such as `consult`, `use_module`, or `ensure_loaded`. Within this set of potentially relevant files we search for the *transitive definition* of the target predicate. Here, we can use the well–known concept of the *predicate dependency graph*, cf. Figure 1, but we have to pay special attention to *meta–call predicates* [1, 2].

Assume, e.g., that we would like to extract the predicate `increment/2`, which is defined in the file `my_arithmetic.pl`:

```
:- use_module(arithmetic).
:- consult([portray_predicates, test_predicates]).

increment(X, Y) :-
   call(add(1, X, Y)),
   portray(increment(X, Y)).

test(my_arithmetic, increment) :-
   increment(3, 4),
   message(sucessful_test, increment(3, 4)).
```

Then, the following file `arithmetic.pl`, which exports the predicate `add/3`, is potentially relevant, and it can be found due to the `use_module`–statement:

```
:- module(arithmetic, [add/3]).

add(U, V, W) :-
   W is U + V.
```

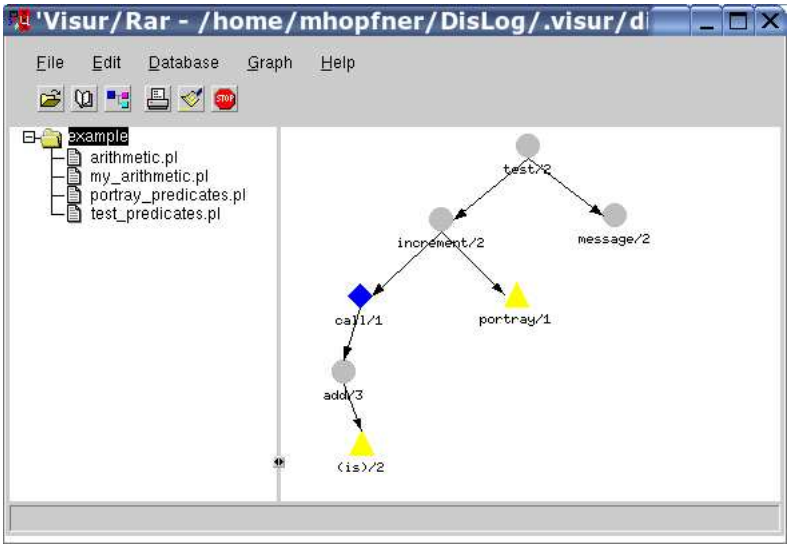


**Fig. 1.** Extended Predicate Dependency Graph in VISUR/RAR

But, at first sight it is not clear that `increment/2` transitively depends on `add`, which can only be inferred since we know that `call` is a meta–call predicate. Moreover, due to the `consult`–statements in `my_arithmetic.pl`, two further files are potentially relevant. Since `portray_predicates.pl` defines the pretty–printing predicate `portray/1` on which `increment/2` depends, this file needs to be in the slice. For ensuring the correctness of the produced slice, the test clause needs to be in the slice, i.e., we include the file `test_predicates.pl` containing the test suite, too.

### 3.1 The Potentially Relevant Files

The collection of potentially relevant files cannot be determined from the directory structure. All files that are loaded using `consult/1` from other potentially relevant files are also potentially relevant. But, moreover, the files must be consulted in the correct order. Sometimes, consulting a file calls a predicate that depends on other predicates. Then, it must be ensured that these predicates are available, before the depending predicates are called. We determine the order of the files using a call

```
code_to_sequence(Code, Sequence).
```

The predicate `code_to_sequence/2` has to be defined for each style of programming. For the DDK, it can be based on the file `dislog_units`.

All files that are consulted from other potentially relevant files are potentially relevant for the slice. E.g., external libraries are frequently loaded using `use_module/1`, and the further modules that they require are loaded using `ensure_loaded/1`.

### 3.2 Dependency Graphs with Meta–Call Predicates

The *transitive definition* of a predicate $p$ consists of all predicates $q$, which are a member of the definition of the predicate $p$ together with the transitive definition of these predicates $q$. In the previous example, the transitive definition of `increment/2` consists of the predicates `call/1`, `portray/1`, `add/3`, and `is/2`. The latter two predicates are reached, since the meta–call predicate `call/1` calls `add/3`.

*Meta–Call Predicates* We need to know which predicates are meta–call predicates and are able to call other predicates in one or more of their arguments. This is important in order to obtain the entire transitive definition of a predicate.

- In general it is impossible to automatically determine the predicates (and their arities) that are called from meta–call predicates. Then, such information has to be provided by the user by specifying the meta–call predicates in a configuration file. Otherwise, the slice will not be complete/correct. However, we can analyze many calls to meta–call predicates using some heuristics.
- For user–defined meta–call predicates, we additionally have to determine the arguments containing potential goals. In order to support the user in collecting the user–defined meta–call predicates, the system is able to automatically generate a list of possible meta–call predicates by searching for rules that call other already known meta–call predicates, such that one of the arguments in the call is connected to an argument of the head of the rule by a sequence of body atoms.
- For built–in meta–call predicates, we already know the arguments containing the goals that will be called, and in many cases we can infer their predicate symbol and arity. If the goal arguments are variables, then we track their bindings. E.g., if we can determine the predicate $p$ and arity $N$ of `Goal`, then we know that `checklist(Goal, Xs)` calls $p$ with the arity $N + 1$; similarly, `maplist/3` adds 2 to the arity of the called goal.

*Extended Dependency Graphs* The call `calls_pp(Code, P1/A1, P2/A2)` determines the predicates `P2/A2` that are called by `P1/A1` in the PROLOG code represented by the FN triple `Code`. If we implement our policies correctly as clauses for `calls_pp/3`, then the transitive definition of the target predicate can be computed as the transitive closure of `calls_pp/3`.

E.g., for an atom `call(Goal)`, we try to determine the predicate and the arity of `Goal`. If the goal is created at runtime, then this is not always possible. But often, `Goal` is bound by statements such as `Goal = add(1,X,Y)` or `Goal =.. [P,1,X,Y]`, where `P` was assigned to `add` before. The following rule for the predicate `calls_pp/3` determines a rule `Rule` with the head predicate `P1/A1`, such that there exists a body atom of the form `call(Goal)`. In XML this atom is encoded as

```
<atom module="user" predicate="call" arity="1">
   <var name="Goal"/>
</atom>
```

`Rule` and `Goal` are selected from the FN triple `Code` using suitable path expressions in FNQUERY. Within `Rule` we search for the predicate `P2/A2` of `Goal` using the predicate `goal_to_predicate/4`:

```
calls_pp(Code, P1/A1, P2/A2) :-
   Rule := Code^rule::[
      ^head@predicate=P1, ^head@arity=A1],
   Goal := Rule
      ^body^atom::[
         @module=user, @predicate=call, @arity=1]
      ^var@name,
   goal_to_predicate(Rule, Goal, [], P2/A2).

goal_to_predicate(Rule, Goal, Goals, P/A) :-
   Atom := Rule^body^_^atom::[^var@name=Goal],
   [user, =, 2] := Atom@[module, predicate, arity],
   ( [P, A] := Atom^term@[functor, arity]
   ; G := Atom^var@name,
     not(member(G, Goals)),
     goal_to_predicate(Rule, G, [Goal|Goals], P/A) ).
```

It could happen that `P2/A2` becomes the predicate of `Goal` by a sequence of assignments. E.g., for the following rule for the predicate `P1/A1=increment/2`, the predicate `goal_to_predicate/4` determines `P2/A2=add/3` by subsequently looking at `Goal` and `G`, respectively:

```
increment(X, Y) :-
   G = add(1, X, Y),
   Goal = G,
   call(Goal).
```

The second call to `goal_to_predicate/2` determines the atom `Atom` representing the equality `G = add(1, X, Y)`, and then it selects the functor `add` and the arity `3` of the term argument of `Atom`.

### 3.3 Slicing of Individual Clauses

For some predicates the slice should only include individual clauses rather than all of the defining clauses. E.g., the defining clauses of generic predicates might be spread over many different files; examples from the DDK are `dislog_help_index/0`, the pretty–printer `portray/1`, and the test predicate `test/2`. In a slice we only need some of the clauses, and including all of them would be highly redundant.

We do not consider these generic predicates when we compute the set of potentially relevant files. But, when we include individual clauses into the slice, then we have to include the transitive definitions of all predicates that are called from them as well.

E.g., in the DDK many files contain some tests – which are usually located at the end of the files. For ensuring that the slice works correctly, the slice should include all clauses for `test/2` calling sliced predicates; in practice, most of these clauses will be contained in the potentially relevant files. In our example, for testing `increment/2`, we obviously need to include the definition of `message/2` in the slice.

### 3.4 Declarations of Predicate Properties

We have to detect the properties of all predicates in the slice, i.e., the corresponding declarations of the types `dynamic`, `multifile`, or `discontiguous`. Some of these declarations will be located in centralized files of the system, others are contained in separate files for each unit. Since these files might not be reached by the file dependency graph, the properties for these predicates have to be extracted using a special policy, such that they can be included into the slice.

```
predicate_to_property(Code, Pred/Arity, Property) :-
   Atom := Code^rule^body
      ^atom::[@predicate=Property],
   ( Term := Atom^term::[@functor='/']
   ; Term := Atom^_^term::[@functor='/'] ),
   Pred := Term-nth(1)^term@functor,
   Arity := Term-nth(2)^term@functor.
```

E.g., in the DDK the declaration `:- multifile test/2.` for the test predicate can be found in the file `test_predicates.pl`.

### 3.5 Global Variables

In the DDK we have a generic concept of global variables, which are implemented using assert and retract. The global variables are accessed using the calls

```
dislog_variable_get(+Variable, ?Value),
dislog_variable_set(+Variable, +Value).
```

It might happen that the value of a relevant global variable is set in a file – either at consultation time or at runtime – that is not contained in the slice. Thus, the following predicate determines – on backtracking – the assignments of all variables `Variable` that are set in a file `File` using a call `dislog_variable_set(Variable, Value)`:

```
file_to_assignment(Code, File, Variable:Value) :-
   Variable := Code^rule::[@file=File]^body
      ^atom::[@predicate=dislog_variable_get]
      -nth(1)^term@functor,
   Atom := Code^rule^body
      ^atom::[@predicate=dislog_variable_set],
   Variable := Atom-nth(1)^term@functor,
   Value := Atom-nth(2)^term@functor.
```

In this rule, the path expression `-nth(N)^term` selects the `N`–th subelement (for `N=1,2`) with the tag `term` of an FN triple.

In the DDK, there also exist further similar concepts for global variables of subsystems, and we use the well–known predicate `gensym/2`, cf. [2].

## 4 Refactoring and Slicing

Refactoring methods [3] provide an effective tool in the context of slicing source code: Such methods modify source code without changing its external behavior. Refactoring methods have demonstrated their practical impact in numerous object–oriented software projects. They are strongly connected with appropriate test methods that are applied before and after performing the refactoring; thus, the preservation of the external behavior can be monitored.

Slicing is frequently accompanied or followed by refactoring. In order to adapt the slice to the software system in which it will be integrated, sometimes it is necessary to move or rename predicates, or to remove `consult`–statements. In the following, we briefly discuss some refactoring methods that are suitable in the context of slicing PROLOG code.

### 4.1 Predicate–Based Refactorings

- *Move Predicate:* A predicate is moved into a module. This refactoring may be useful for test predicates which are located in distributed modules. It is reasonable to apply this refactoring before performing the slicing procedure.
- *Rename Predicate:* This refactoring is usually performed after the slicing operation in order to adapt the slice to the target language, which will usually be a different PROLOG dialect. It is especially useful when slicing source code in order to allow for a simpler porting to another language dialect.
- *Extract Predicate:* Similar to the previous refactoring, this method is also useful, when we want to prepare the slice for the adaptation to another language dialect. Then, we extract *hostile* predicate blocks from the slice. In a subsequent step, the extracted blocks are replaced by suitable predicate calls from the target language.
- *Remove Unused Predicates:* If we slice at the granularity of files, then we may think of removing unused predicates from the files in the slice.

### 4.2 Module–Based Refactorings

Schrijvers et al. [8] also discuss useful refactorings based on the granularity of *modules*: merge modules, remove dead code intra–module, rename module, split module. In the context of slicing source code at the granularity of files these refactorings can be useful, if the sliced system does not need to allow for file–based updates.

## 5 Case Studies

We present two case studies applying the proposed slicing approach to the DISLOG Development Kit DDK, which we are developing using XPCE /SWI PROLOG [4]. The functionality ranges from (non–monotonic) reasoning in disjunctive deductive databases to various PROLOG applications, such as a PROLOG software engineering tool, and a tool for the management and the visualization of stock information.

Currently, the DDK consists of about 14.000 clauses; they are located in 440 files containing a total of about 100.000 LoC (lines of code). We have sliced out two subsystems, Minesweeper and FNQUERY. Slicing at the granularity of files has reduced the size of the system to subsystems of 16% (Minesweeper) and 4% (FNQUERY), respectively, of the original size, and slicing at the granularity of predicates has further reduced the size of the system to subsystems of 5% and 0.8%, respectively, of the original size.

The computation time is divided in a preprocessing time and the slicing time. The preprocessing takes about 0.6 msec per LoC on an Intel Pentium 1.1 GHz, 512 MB RAM. For the DDK the preprocessing took about 10 minutes. The preprocessing has to be done only once; its result can be used for each slicing operation, and for further operations like computing dependency graphs. The slicing operations took about 28.4 sec for Minesweeper and about 6.6 sec for FNQUERY, respectively.

Tests were also extracted and did sucessfully pass after slicing.

### 5.1 The Game Minesweeper

The slice for the game Minesweeper, which is a part of the DDK, should contain all files that are needed for a stand–alone version including the graphical user interface. Thus, we have extracted the transitive definition of the target predicate `create_board/0`, which creates the GUI of the game, cf. Figure 2.

In addition, we have included the transitive definition of all predicates called by the buttons `New Game`, `Show`, `Help`, `Help*`, and `Quit`, of the GUI. Clicking on these buttons calls a meta–call predicate, which calls another predicate, namely the predicates `new_game/0`, `show_board/0`, `clear_board/0`, and twice the help predicate `mine_sweeper_decision_support/1`, respectively. Since the syntax for meta–calls is often used in different ways, every user can write his own policies for obtaining the transitive definition of these calls, but one can also manually add predicates to the list of relevant predicates.

The consult file `index.pl` for the slice – a fragment is shown below – declares the predicate properties, sets all necessary global variables and consults the relevant files:

```
:- dynamic ...
:- multifile test/2, ...

:- dislog_variable_set(
       smodels_in, 'results/smodels_in').
```

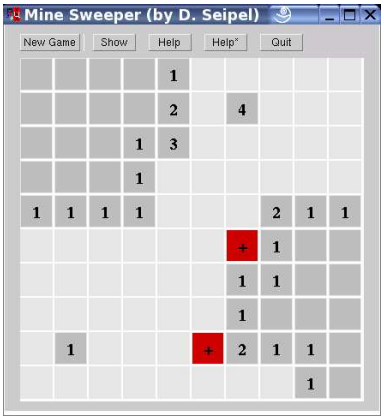


**Fig. 2.** Minesweeper

```
:- consult([
      'library/loops.pl',
      'library/ordsets.pl', ...
      'sources/basic_algebra/basics/operators',
      'sources/basic_algebra/basics/meta_predicates',
      ...]).

test(minesweeper, create_board) :-
   create_board.
```

### 5.2 The Field Notation and FNQUERY

In the second case study we have created a slice for the target predicate `:=/2` of FNQUERY, which is defined in the DDK module `xml/field_notation`. This slice consists only of about 800 LoC, which are extracted from 21 files of the DDK. 19 of these files are from regular DDK modules, which are located in the directory `sources`, and two of the files are library files from the directory `library`. The slice is derived from

- 8, i.e. 50%, of the files of the module `xml/field_notation`,
- 7, i.e. 25%, of the files of the module `basic_algebra/basics`,
- 3 files from the unit `development`, e.g., the file `development/refactoring/extract_method`,
- one file from the module `basic_algebra/utilities`, and
- the library files `lists_sicstus.pl` and `loops.pl` from the separate directory `library`.

## 6 Final Remarks

The slicing tool has been built based on two modules of the toolkit DDK: the system VISUR/RAR for visualizing and reasoning about source code, and the XML query and transformation language FNQUERY. It can be used for extracting slices from large software packages, provided that we have an XML representation for the considered language. So far we have applied it for extracting several subsystems of the DDK.

The tool can be extended by plugging in further user–defined slicing policies reflecting different styles of programming. Since the source code is represented in XML, slicing policies can be specified declaratively using the XML query language FNQUERY, which is fully interleaved with PROLOG.

We have used XML representations of PROLOG or JAVA source code for other purposes as well: for comprehending and visualising software with VISUR/RAR [6], for refactoring PROLOG programs and knowledge bases [5, 7], and for clone detection in PROLOG and JAVA source code.

## References


1. *I. Bratko:* PROLOG– Programming for Artificial Intelligence, 3rd Edition, Addison–Wesley, 2001.
2. *W.F. Clocksin, C.S. Mellish:* Programming in PROLOG, 3nd Edition, Springer, 1987.
3. *M. Fowler:* Refactoring – Improving the Design of Existing Code, Addison–Wesley, 1999.
4. *D. Seipel:* DISLOG– A Disjunctive Deductive Database Prototype, Proc. 12th Workshop on Logic Programming WLP 1997.
5. *D. Seipel:* Processing XML–Documents in PROLOG, Proc. 17th Workshop on Logic Programming WLP 2002.
6. *D. Seipel, M. Hopfner, B. Heumesser:* Analyzing and Visualizing Prolog Programs based on XML Representations. Proc. International Workshop on Logic Programming Environments WLPE 2003.
7. *D. Seipel, J. Baumeister, M. Hopfner:* Declarative Querying and Visualizing Knowledge Bases in XML, Proc. 15th International Conference on Applications of Declarative Programming and Knowledge Management INAP 2004, pp. 140-151.
8. *A. Serebrenik, B. Demoen:* Refactoring Logic Programs, Proc. Intl. Conference on Logic Programming ICLP 2003 (Poster Session).
9. *M. Weiser:* Programmers use Slices when Debugging, Communications of the ACM, vol. 25, 1982, pp. 446–452.
10. *M. Weiser:* Program Slicing, IEEE Transactions on Software Engineering, vol. 10 (4), 1984, pp. 352–357.
11. *J. Wielemaker:* SWI–PROLOG 5.0 Reference Manual, http://www.swi-prolog.org/
    *J. Wielemaker, A. Anjewierden:* Programming in XPCE/PROLOG
    http://www.swi-prolog.org/