C#Prolog -- A Prolog interpreter written in C#

John Pool
Amersfoort
Netherlands
j.pool@ision.nl

C#Prolog -- A Prolog interpreter written in C#

Version 4.1 -- September 2015

Licensing scheme changed into LGPL.

Only a few bugfixes (bug in weekno/1/2) and two new predicates (bw_transform/3, combination/3).

Version 4.0 -- April 2014

Also for this release the main purpose was bugfixing. Quite a large number of bugs has been fixed. In addition, a couple of predicates have been added (the main reason being that I needed them for my own purposes…):

- permutation/2
- :- fail_if_undefined/2
- ip_address/1/2

Use the 'help' command for further details.

I also implemented a mechanism for conditional inclusion of code in source files, comparable to the conditional compilation mechanism in C#. See help('!if') for details.

John Pool
Amersfoort
Netherlands

Version 3.2.0 -- June 2013

As in the previous release, the main purpose of this release was fixing a number of bugs and shortcomings.

Furthermore, a JSON-parser has been added which transforms a JSON structure to a Prolog term. Enter *help(json_term)* for more info.

In the previous version, the number of simultaneous open database connections was limited to one. The restriction has been lifted.

Also, a predicate xml_transform/1/2/3 has been added for applying xsl style sheets.

In addition, it has been made a little easier to run a Prolog engine as an external program. See the new C# project PLX.CSPROJ that has been added to the solution.

New / modified predicates are:

- json_term/2/3
- xml_transform/1/2/3
- sql_connect/3
- sql_select/3
- sql_select2/3
- sql_disconnect/1
- sql_connection/3
- sql_command/2/3
- stacktrace/1
- regex_match/3/4
- regex_replace/4

Please use the help/1 predicate for more information on each of these.

Version 3.1.0 -- December 2012

This release is mainly intented for fixing a number of bugs and shortcomings. It contains little new funcionality. New / modified predicates are:

- console/1
- date_part/2
- json_format/2
- json_xml/2/3/4
- readeof/2
- regex_match/3/4
- regex_replace/4
- shell/0/1/2/3
- string_datetime/2/4/7

Please consult the help/1 predicate for more information on each of these.

Version 3.0.0 -- November 2011

The primary activity for this version was refactoring and bugfixing. In addition, a number of new predicates was added. Refactoring resulted in a speed improvement of some 20 percent. The biggest contribution in this respect came from overloading the `Unify()` method for the various Term types.

The following modifications are worthwile mentioning:

1. ListPatternTerm

A new term type *ListPatternTerm* has been added. A ListPatternTerm is a list that is opened by [! and closed by !]. It is used to specify a pattern (arrangement of list elements) that a regular list must conform to in order to get unified with it. If the list conforms, variables in the *ListPatternTerm* can be used for picking up specific terms or sublists. Here are some examples:

```
% get the last element of a list.
% '..' denotes a 'gap' consisting of an arbitrary number of elements.

1 ?- [1,2,3,4] = [! .., L !].

 L = 4

% A gap can also be specified as follows:
% {<minimum number of elements>, <maximum number of elements>}
% A gap can be prefixed with a variable, which will contain a list
% of the elements in the gap, e.g. X.. or X{1,6}

% Search in a list for sublist with a length between 2 and 5
% inclusive, that is bounded by the atom xxx at both sides.

2 ?- [! .., xxx, S{2,5}, xxx, ..!] = [a, xxx, 1,2,3,4, xxx, 5].

 S = [1, 2, 3, 4]

% Search in an name=value list for the value of
% a specific attribute

3 ?- [! .., color=C, ..!] = [length=12, height=78, color=red,
weight=100].

 C = red

% A pattern can also be used in a predicate clause head.
% Suppose we have the following predicate definition:

x([! .., p(X)|q(X), .. !], X).

% The | is used for separating alternative matches.
% Now we can call it like this:

4 ?- x( [1, 2, 3, p(9), 4], Z).

 Z = 9

% Max. 5 elements at the beginning, followed by a or b or c,
```

```
% followed by arbitrary elements.
% Y! means that the selected alternative must be unified with Y

5 ?- [! X{,5}, Y!a|b|c, .. !] = [1, 2, 4 ,b , 5,6].

 X = [1, 2, 4]
 Y = b

% A list of integers, containing N followed by precisely N
% elements, followed by N again.

6 ?- [!.., N, L{N}, N, ..!] = [1,7,5,4,8,7,6,5,4,9].

 N = 4
 L = [8, 7, 6, 5]

% Look for the first p followed by a term that is *not* equal to
% a or b or c. Assign the result to Z. Notice that the not-operator
% (~) applies to the whole or-expression.

7 ?- [! .., p, Z!~a|b|c, .. !] = [a, p, b, p, a, p, e, p, q, r].

 Z = e
```

Although this would be technically feasible, for now I have chosen not to make the *ListPatternTerm* backtrackable. It only returns the first match. You can, however, create a recursive predicate that in the recursion attempts to find the desired pattern in the as yet unexplored parts.

2. Windows interface

An elementary Windows interface has been developed. To this end, the Prolog engine was decoupled from the user interface and turned into a CsProlog.dll library. This library is referenced in the console version (pld.exe) and in the Windows version (plw.exe).

The only purpose of the Windows interface is to show that it is possible to develop one! Nothing but the basic features are available, although obviously many enhancements are conceivable. It is mainly useful to see how you can link a CsProlog engine to some other application (the user interface in this case).

The Windows version can be found in the PLW map. Both executables have their own configuration file (pld.exe.config and plw.exe.config), with (at the moment) identical contents. I have not found a way yet to have a single config file only.

3. Batch mode

It is possible to call the Prolog engine (pld.exe) from within a batch file. When pld.exe is called with arguments, it is automatically assumed that Prolog must be started up in batch mode. The first argument specifies the query that should be carried out; the optional second argument specifies the number of times backtracking should be applied. The default is 0; if a * is specified, backtracking will take place as ofte as possible.

If an error occurs, the error message is displayed on the console, and the DOS ExitCode is set to 1.

Example:

```
call pld.exe "[test], run_test( 12)" *
```

This means that Prolog will consult TEST.PL, and then execute `run_test(12)`. Backtracking will be carried out as often as possible.

It is also possible to call plw.exe in batch mode. In that case, output is written to a log file in the subdirectory called BATCHLOGS of the directory in which the executable resides. The reason you might be wanting to use this is that certain Windows-mode operations will not run in console mode (e.g. writing to the clipboard).

3. Recognition of singleton variables

When a Prolog source file is read in, singleton variables (i.e. named variables occurring in the code once only) are now reported.

4. Improved XML handling

A couple of bugs have been removed from the XML reading process, and a slightly more readable and compact Prolog representation of an XML-structure has been chosen.

5. Online help

Online help has been added. Entering '`help.`' gives a survey of all available builtin predicates. Help on a specific predicate can be obtained by entering '`help( predicate name)`' or '`help( predicate name/arity)`'.

6. TRY/CATCH construct, in combination with throw/2/3

It is now possible to use a TRY/CATCH construct in a predicate. It has the following syntax:

```
TRY
(<terms>)
CATCH [<exception class>] [, M]
(<terms>)
CATCH [<exception class>] [, M]
(<terms>) ...
```

So, a TRY/CATCH statement can have more than one CATCH-clauses, each labeled with the name of an 'exception class' that can be given freely by the user and that corresponds to the exception class name in the `throw/2/3` predicate. When a throw is executed within a TRY-body, the list of CATCH-clauses is searched for one with the exception class specified in the throw, or for one without an exception class.
In throw, a message can be specified. This message will be bound to the optional message variable M that a CATCH-clause can have.

If the exception cannot be handled in the predicate in which the throw was done, the calling predicate is searched (provided the call was made in the body of a TRY).

Example:

```
tc_test :-
  TRY
  (
    nested
  )
  CATCH outer, T
  (
    writelnf( "Now in tc_test CATCH outer, error message is '{0}'", T),
    throw( foo, "No catch available")
  ).

nested :-
  TRY
  (
    writeln("Now in nested TRY"),
    throw( excp1, "excp1 thrown")
  )
  CATCH excp0, M
  (
    writelnf("Now in CATCH excp0, exception msg is '{0}'", M)
  )
  CATCH excp1, M
  (
    writelnf("Now in CATCH excp1, exception msg is '{0}'", M),
    throw( outer, M)
  )
  CATCH outer
  (
    writeln( "Now in nested CATCH outer")
  )
  CATCH M
  (
    writeln( "Now in nested CATCH without exception class")
  ).
```

If tc_test is executed, the following output will appear:

```
Now in nested TRY
Now in CATCH excp1, exception msg is 'excp1 thrown'
Now in tc_test CATCH outer, error message is 'excp1 thrown'

*** error: No CATCH found for throw( foo, "No catch available ")
```

6. Complex numbers

In the context of some other project, I built an expression evaluator that was able to handle complex numbers. I included the ideas I used there in the logic that is used by the is/2 predicate. The standard operators and a number of math functions now also work for complex numbers, so you can do things like `X is e^(i*pi)` and verify that this still evaluates to -1.

7. New predicates

The predicates below are new. Use the (also new) `help/0/1` predicate to obtain more information on each of these.

- `workingdir/0/1`
- `get_counter/2`

- crossref/1
- sendmail/3/4/5
- readln/1
- readatom/1
- readatoms/1
- help/0/1
- crossref/1
- string_term/2
- succ/2
- fileexists/1
- profile/0
- noprofile/0
- throw/2/3
- showprofile/0/1
- clearprofile/0
- set_counter/2
- inc_counter/1
- dec_counter/1
- string_term/2
- writelnf/1/2
- errorlevel/1
- term_pattern/2/4/5
- stringstyle/1
- make_help_resx/0
- clipboard/1
- getenvvar/2
- setenvvar/2
- sql_command/2

- Extra builtin functions (enter '`help( is).`' for an overview)

8. How to create a new predicate

The easiest way to do this is by studying an existing predicate that is more or less similar in number, type and mode (in/out/inout) to your predicate. Anyway, you will have to do the following at least:

1. Create an entry in BOOTSTRAP.CS. The format of such an entry is `<predicate name>( <list of aruments>) :== <BI enum entry>.`
2. Add a value `BI.<BI enum entry>` to the BI enum in BUILTINS.CS.
3. Create the code for your predicate in the `switch`-statement in BUILTINS.CS.

Backtrackable predicates

These are all implemented according to the same pattern: a predicate with an extra first State argument is introduced. This argument maintains the state between two successive backtracking calls. It is initialized at the first call, and reset to an unbound variable after the last successful call. `UserClassTerm` (in DERIVEDTERMS.CS) can be used for creating a State term with an arbitrary class type content. In my experience, an enumerator is eminently suited for this task, since in fact it can be considered a finite state engine that yields one value at a time and saves state between calls. See BOOTSTRAP.CS for examples

9. How to create a new built-in function that can be evaluated by `is/2`

The code for such a function must be added to the `Apply`-method in TERMARITHMETIC.CS.

Please report to me any bug you may find. Any improvements or useful enhancements will certainly be included in some next version.

John Pool -- December 1, 2011

Version 2.0.0 -- December 2010

Version 1 was my first significant project in C#, in which a number of dubious design decisions were made. Therefore, I decided to redesign large parts of the code. Also, there was quite a number of 'loose ends' that needed to be fixed.

Some functional changes:

- A simplified way of interaction with a Prolog engine (query specification and retrieval of the answers; cf. main.cs)

- The introduction of predicates for accessing database tables. These were also present in the previous version, but not in a very satisfactory way. In the new verion, the result sets of arbitrary SELECT statements or Stored Procedured can be accessed as if they were predicate clauses stored in the Prolog knowledge base. Other SQL-statements can be executed as well. Any database can be accessed for which a (C#) SqlProvider is available (so Excelsheets and MS-Access db-files as well). See readme.txt in the SQL subdirectory.

- The *read* predicate now functions correctly. Edinburgh-style file-I/O has been implemented.

- A config file in which various properties can be set (CsProlog.exe.config), *config_setting/2* (for reading only).

- Unicode is now fully supported.

- Escape codes in strings are resolved.

- A feature has been implemented that allows retaining the command history over sessions. The last N commands are saved, where N can be set in the config file.

- The 'wrap/1/2' directive: syntactic sugar with which it is possible to define 'grouping tokens'. Easiest to explain by an example:

  ```
  1 ?- wrap( '[:', ':]').
  % or: wrap('[:'), which will automatically generate the ':]'.
  ```

  Now you can define terms such as:

  ```
  2 ?- X = [: 1, 2, 3, 4 :].
  ```

- A wrap can be considered a functor that is split in two parts '[:...:]'(1, 2, 3, 4).

- The is/2 predicate evaluates the + and * operators with lists as operands, e.g.
  `X is [1,2] + [a,b]` yields `X = [(1,2), (2,b)]`
  `X is [1,2] * [a,b]` yields `X = [(1,a), (1,b), (1,c), (2,a), (2,b), (2,c)]`
  It is fairly easy to enhance/modify this and to add specific operations that you consider useful.

Some technical changes:

- The most important change was the introduction of a class hierarchy for Prolog terms. Each different type of term (Atom, Compound, and furthere specalisations of those such as Number, String, List, etc.) now have their own subclass derived from BaseTerm. This has led to a significant improve in code maintainability, and it also makes it much easier to introduce more exotic subtypes if necessary.

- Another significant change is the way parsing is carried out. Prolog parsing is not easy, primarily as a consequence of the op/3 predicate with which new prefix, postfix and infix operators can be introduced at runtime. This precludes the use of parser and lexer generators such as Yacc. In the previous version, this problem was not adequately solved, causing incorrect parses in a number of situations. Important guidelines for the implementation were found in the article by Koen de Bosschere that is included in the docu-directory.

- Conversion to a Visual Studio 2008 solution (DotNet 3.5 framework). There is no reason why the solution could not be migrated to VS2010, apart from the fact that not every user may have access to this yet.

- The use of *generics* wherever possible. This also lead to improved readability and understandability.

- Removal of static instances as far as possible and necessary. This will open the possibility of having simmultaneous instances of the Prolog engine running at the same time, without the risk of overwriting each other's static instances.

- The *:-persistent* directive was dropped, I considered it of too limited use and too complicated.

Please feel free to modify the code, and please report to me any bug you may find. Useful enhancements will certainly be included in some next version.

John Pool -- December 24, 2010

Version 1.0.0 -- February 2007

The C#Prolog interpreter has been rewritten in C# from scratch, although some of the basic mechanisms are inspired by XProlog developed (in Java) by Jean Vaucher, who at his turn enhanced the WProlog program developed by Michael Winikoff. In comparison to XProlog it has the following new features:

- A more user-friendly (but still quite basic) DOS-box toplevel user-interface, with the possibility of referring to and editing of commands given earlier (using !, !!, !<n>, and editing via !<n>/<oldstring>/<newstring>/);

- The possibility of defining new operators using :-op(Precedence, Type, Name);

- Enhanced arithmetic (many of the standard C# builtin functions are available);

- Builtin types DateTime and TimeSpan;

- Predicates for manipulating XML (a la PLXML; cf. Google). These predicates use the superb C# capabilities for handling XML;

- (Simple) preprocessor directives #define, #undefine #if, #else, #ifnot, #endif;

- assert/retract not only for facts, but also for rules;

- Builtin DCG processing;

- :-*discontiguous(…)* directive, to indicate that predicate clauses may be scattered over the source file. Multi-file predicates are not allowed, however;

- :-*persistent(…)* directive, to indicate that the predicate clauses are to be retrieved from / inserted into a database table (or stored procedure). This only applies to fully instantiated fact (head-only) predicates. The use of such predicates is completely transparent: the only difference is that asserted values are still there at the next session. This is also the method that I used for storing huge fact databases. The RDBMS that I use is Firebird (for DotNet, freeware), which is very reliable, very fast, and has many features;

- First-argument indexing;

- An option to set a limit to the execution time of a query;

- Enhanced debugging and tracing options. Debugging output can be captured into an XML-file;

- Regular expressions (limited, but easily extendible using C#'s capabilities).

This version appears to be functioning correctly, and is fairly fast. I have tried to run CHAT-80, which works alright, though in some cases no answer is returned. Because of the complexity of the CHAT-80 code I have not been able to trace whether this is because of some defect in the interpreter or a bug in the CHAT-80 code.

This does not mean to say that there is nothing left to do. Some known problem areas to which I did not get around yet, are:

- The *read*-predicate may needs some debugging. I have not tested this predicate extensively. The same goes for 'seeing' and 'telling'. The consult-predicate works fine;

- The *setof*-predicate is not implemented according to the official definition, in that the existential operator (^) does not work. *setof* simply returns all solutions (sorted) in a single list;

- A graphical user-interface would be nice;

- C#Prolog does not completely support Unicode. In principle, this should not be to difficult, given the C# Unicode facilities;

- *Atom/1* is not implemented fully correctly. It works fine for 'regular' atoms, but e.g. in case of atom(++) it should return true, but if + is defined as a prefix operator ++ is regarded as a compound term +(+);

- Quite some of the standard ISO-predicates are lacking. Usually their implementation will not be difficult, but I simply have not had the time yet to do this;

- It would be nice to have a compile facility (WAM?);

For the parser I have used a grammatical description of the Prolog language which I fed into a proprietary parser generator (which is not publically avalailable) that turned it into C# code. If necessary (wich will not be very often, since the grammar of the Prolog language is not likely to change very much) I can do this again of course. The grammar from which the parser code was generated has been included, just to get a taste of how it works, but is is obviously of no practical use without the parser generator itself.

Here is a brief description of the files making up C#Prolog. Only the main features of each files are mentioned:

_csc.bat, csc_options.inc	Support files for the above commands
C#Prolog.cs	Top level; contains main()
Builtins.cs	Contains all predefined predicates
Engine.cs	Contains the Execute() method, and the implementation of all predefined predicates
IO.cs	All IO-related stuff. Must be adapted when a graphical interface is developed
Operator.cs	Operator details
Persistent.cs	Implementation of presistent predicates (interface with Firebird)
PL.cs, PL.grm	Parser, generated by parser generator, from PS.grm. Any modifications here will be wiped out by a next generation!
PredDescr.cs	Predicate descriptor: all there is to know about a predicate
PredStorage.cs	Datastructures and methods for storing the 'program': the predefined and consulted predicates
SimpleDOMParser.cs	XML-related stuff
Term.cs	Datastructures and methods for creating and handling Terms
TermNodeList.cs	Datastructures and methods for creating and handling

	linked lists of Terms
TermSet.cs	Sets of Terms
Utils.cs	Miscellaneous stuff