

BinProlog 2006 version 11.x  
Professional Edition  
User Guide

**Paul Tarau**

BinNet Corp.  
e-mail: [binnetcorp@binnetcorp.com](mailto:binnetcorp@binnetcorp.com)  
WWW: <http://www.binnetcorp.com>

# 1 Installation

Copy the BinProlog executable for your architecture to something called **bp.exe** or **bp** somewhere in your path, then type 'bp'. BinProlog standalone executables are usually provided in subdirectory **bin** of the distribution.

For *multi-user* installations set the BP\_PATH variable to point to the BinProlog source directory, for instance **/bp\_dist/src**. This will allow users to load example programs in directory **/bp\_dist/progs** without providing path information and will also make include commands like `:-[library(lists)]`, referring to **/bp\_dist/library** accessible from any directory.

Normally the appropriate **bp.exe** or **bp** file (a self contained executable) is all you need to have BinProlog running. For PCs, just copy the bp.exe Windows executable file somewhere in your PATH.

To start BinProlog use

```
$ bp <command-line options> <wam-bytecode-file> or <prolog-file>
```

or simply

```
$ bp
```

Sizes of the blackboard, heap, stack, trail and code areas can be passed as command line parameters etc., as well as other options, as shown with `bp -x help` request:

With `bp -x` you can list the following:

## COMAND LINE OPTIONS:

```
-h ==> HEAP SIZE in Kbytes, default:1024K
-s ==> STACK SIZE in Kbytes, default:256K
-t ==> TRAIL SIZE in Kbytes, default:256K
-c ==> CODE SIZE in Kbytes, default:1024K
-b ==> BLACKBOARD SIZE in Kbytes, dynamic if 0, default:512K
-a ==> MAX. ATOMS, give exponent of 2, 2<value>, default=216=65536
-d ==> HASH DICT. entries, give exponent of 2, 2<value>, default 216=65536
-i ==> IOBUFFER, in bytes, default:131072bytes
-q ==> QUIETNESS level, default:2 (lower means more verbose)
-l ==> LOAD_METHOD:
    (1=mcompile, 2=scompile, 3=oconsult, 4=dconsult, 5=sconsult): 1
-r ==> call/update RATIO controlling dynamic recompilation:10
-p ==> PORT to run on as a daemon if >0, 1 shorthand for default_port, current d
default:0, -p10 forces detection of current IP address
```

## ARGUMENTS:

```
STARTUP FILE: *.bp, *.pl, *.wam or
GOAL: pred(args),
default: wam.bp
```

```
Prolog execution halted(2). CPU time = 0.241s
```

## ARGUMENTS:

```
STARTUP FILE: *.bp, *.pl, *.wam or
GOAL: pred(args),
default: wam.bp
```

Type

```
?- help(<word>).
```

and then use `info/1` with the matching predicate names to get a short description and possibly an example of use:

```
?- info(name/arity).
```

Type

```
?-info.
```

for a (long) description of builtins and examples.

## 2 Deprecated Predicates

BinProlog has undergone a number of simplifications and as a result, older, seldomly used functionality has been removed. Please take a look at the file `library/deprecated.pl` for the code of the removed predicates. By including the file in your older programs, most of the functionality of the removed predicates can be reactivated. The changes include replacement of the Assumption Grammar operations (now part of standard DCG processing) and mobile code+client/server networking (now replaced by robust Jinni compatible code which supports socket reuse for improved performance - see `help(rpc)`).

## 3 Obtaining BinProlog

The ORIGINAL DISTRIBUTION SITE for BinProlog<sup>1</sup> is:

```
http://www.binnetcorp.com/BinProlog
```

Please send comments and bug reports to [binnetcorp@binnetcorp.com](mailto:binnetcorp@binnetcorp.com).

## 4 Release Notes for This Version of BinProlog

Welcome to BinProlog Full Source Edition and BinProlog Professional Edition!

```
!!! after you work with them on Windows some makefiles might need conversion
to remove ^M character on Unix systems
```

```
!!! Please look for *.bat files which usually automate frequent maintenance
operations on XP, 2000, NT or Win9x PCs.
```

With Microsoft VCC based precompiled distributions for Windows XP/2000/NT/98/95,

---

<sup>1</sup>BinProlog Copyright © Paul Tarau 1992-98 and BinNet Corp. 1998-2006. All rights reserved

-a ready to run binary (bp.exe) is included in directory bin,  
as well as a redistributable runtime bpr.exe for Windows

-a packaging of BinProlog as a DLL and examples are available in  
directory BP\_DLL.

-header files in static libraries are available in directory lib - allowing  
to use the C-interface or generate C-code without need to recompile the sources

Just copy the executables bp.exe or bpr.exe somewhere on your path,  
and bp\_lib.dll and bpr\_lib.dll to the Windows or Winnt/System32 directory,  
or simply copy them to the local directory where you keep your application.

BinProlog's C-interface tools are in directory c\_inter.

Tools for generation of standalone executables, through compilation to C  
are available in directory pl2c.

BinProlog's Tcl/Tk interface is in directory TCL. Is ready to run, on  
all platforms (new client/server design), no special make actions are  
needed.

Finally, the directory doc, in distributions coming with full documentation,  
contains the documentation in PostScript and HTML form - note the new API  
description in file help.html, also generated when just typing "help" in  
BinProlog.

-----  
In case you have BinProlog Full Source Edition, the directory src  
contains makefiles for gcc, VCC 6.0 and .NET C/C++ batch files to rebuild the  
sources after you make changes. Just go in directory src and type  
make all or makeall.bat

Please read the documentation files provided separately and visit  
our ONLINE HELP SYSTEM at BinNet Corp.'s web site, for last minute  
updates, demos and information about new features and components.

On Cygnus' CYGWIN gcc, taking

```
make all
```

which also works on normal Unix systems not requiring unusual flags.

On Linux PCs type

```
make linux_mt
```

```
or
```

```
make linux
```

On Solaris sparcs type

```
make solaris
```

This will take care to generate the content of most of the directories.

Finally, the Full Source Edition also contains VCC 6.0 project files in deirectory winbp, for working conveniently with the sources on a Windows NT workstation.

Alternatively, work on sources is now also supported for Cygnus gcc as well as Linux and othe Unix platforms, through a complete set of updated makefiles.

-----  
Additional components, available on some platforms.

The directory csocks contains tools for building standalone C-based socket based client, server and a remote toplevel components based on BinProlog's modular and portable socket package.

Tools to build an ISO parser for BinProlog are in directory cparser.

## 5 Using BinProlog

### 5.1 Learning Prolog

If you are just starting, you might want to take a look at one of the following online tutorials. Most of their content applies to any Prolog around, and in particular to BinProlog which a fairly standard Prolog system.

Online Prolog Introductions/Tutorials:

<http://kti.ms.mff.cuni.cz/~bartak/prolog/learning.html>  
[http://www.cs.bham.ac.uk/~pjh/prolog\\_course/sem223.html](http://www.cs.bham.ac.uk/~pjh/prolog_course/sem223.html)  
<http://www.cs.sfu.ca/CC/SW/Prolog/Notes/toc.html>

The latest version of BinProlog is usable directly over the Internet (follow the *demo* and *query* links from <http://www.binnetcorp.com>).

### 5.2 Consulting/compiling files

BinProlog features a number of different compilation and consulting methods as well as dynamic recompilation of consulted (interpreted) code for fast execution.

The shorthand

```
?- [myFile].
```

defaults to the last used compilation method applied to `myFile.pro` `myGile.pl` or `myFile`. Among them, the default `mcompile/1` comiling to memory and `scompile/1` which uses temporary `*.wam` files and to quickly load files which have not been changed. A good way to work with BinProlog is to make a "project" `*.pro` file includeing its components, as in:

```
:-[myFile1].  
:-[myFile2].  
.....
```

The shorthand for `co(myFile)`,

```
?- ~myFile.
```

defaults to the last used *interpretation* method applied to `myFile.pro` `myFile.pl` or `myFile`. For online information on their precise behavior, do:

```
?-info(oconsult),info(consult),info(dconsult),info(sconsult).
```

`oconsult/1`:

```
reconsult variant, consults and overwrites old clauses
```

`consult/1`:

```
consults with possible duplication of clauses, allows  
later dynamic recompilation
```

`consult/2`:

```
consult(File,DB) consults File into DB
```

`dconsult/1`: `reconsult/1` variant, cleans up data areas,  
consults, allowing dynamic recompilation

`sconsult/1`:

```
reconsult/1 variant:
```

```
cleans up data areas, consults, makes all static
```

To control dynamic recompilation you can use `dynco/1` with `yes` and `no` as arguments or `db_ratio/1` to precisely specify the ratio between calls and updates to predicates which will trigger moving it from interpreted to compiled representation (default=10).

*BinProlog's compilation mode is intended to work with ONE toplevel project file including various components.* For interactive development use:

```
~file1.  
~file2.  
.....  
~fileN.
```

Note that, by default, older definitions from previous or the same file of predicates with same name/arity are quietly overwritten by default.

Do not worry about performance. List your predicates at will, using `listing/1` or debug them with `trace/1`, modify them with `assert/retract`. Automagically, BinProlog will override heavily used predicates with compiled variants. This usually pushes performance to about half of optimized compiled code, while giving you the full flexibility of interpreted code.

### 5.3 The interactive toplevel shell

To see the command line options:

```
$ bp -x
```

To compile and load a file possibly having one of the suffixes 'pl' or 'pro', do:

```
?-[<file>].
```

To quickly consult a file for interpreted execution, having one of the suffixes 'pl' or 'pro', do:

```
?-co(<file>).
```

**Search path mechanism.** BinProlog searches for programs in the directories `.`, `./progs` and `./src` `./library`, relative to the `BP_PATH` environment variable which defaults silently to the current directory if undefined. The directory `./myprogs` is also searched for possible user programs. While compatible to previous configurations which were single user minded, this allows installing the BinProlog distribution for shared use in a place like `/usr/local/BinProlog`<sup>2</sup> In addition, the environment variable `PROLOG_PATH` provides a secondary search directory for including an individual user's Prolog library files.

Mode is interactive by default (for compatibility with other Prologs) but if you use a modern, windows based environment you may want to switch it off with:

```
?- interactive(no).
```

or turn it on again with

```
?- interactive(yes).
```

Operators are defined and retrieved with

```
:-op/3, current_op/3.
```

as in:

```
?-op(333,xfx,and).
```

allowing to enter:

```
?-write(joe and mary),nl.
```

instead of the usual `?-write(and(joe,mary))`.

## 5.4 Source code loading methods

The preferred use of BinProlog is through reloading a unique project file containing included files<sup>3</sup>.

A smart compile facility (`scompile/1`) implement a basic *make* facility: if the `*.wam` version of the file is newer it will be reloaded very quickly instead of recompiling the corresponding `*.pl` file.

---

<sup>2</sup>This is usually better to be left to a system person who also can ensure that users inherit the `BP_PATH` environment variable. An individual user can also put something like `setenv BP_PATH /usr/local/BinProlog` in his or her `.cshrc` or `.login` file, to access the shared BinProlog programs.

<sup>3</sup>This overcomes the limitation of previous versions of having only one top-level file.

**Including files.** For both compiled and consulted files use a generic include directive:

```
:-[file].
```

which automatically adjusts to your current *load method*. You can set your preferred loading method directly from the command line using option 1. Explicite use of `compile/1` or `reconsult/1` can be used to restrict your selection to a current compile method or consult method.

Use `co/0` and `ed/0` as a shorthand to reload/re-edit the last compiled/consulted file.

Use `~ file` or `co(file)` as a short hand to reconsult a file with your current consult method. This is useful if your load method is by default a version of compile but you want to reconsult a file so that you can debug/list clauses of verious predicates.

*Normally, after each compile BinProlog cleans up ALL its data areas. Not relying on hidden state of the database as most Prologs do is a very IMPORTANT element of reliable software developpment. In compiled BinProlog recompiling means a fresh start, so that the future of computation is only dependent on your code, not on what happens to be accidentally in your database. This FEATURE is unlikely to be changed in the future. .*

With full flexibility of interpreted code and its automatic on the fly compilation, this limitation does not affect by any means a user who prefers conventional Prolog semantics, which is the default for the consulting facility `co/1` or its shorthand `~`. There's however an ability to *push* your code into the BinProlog protected kernel at runtime by typing `pc`. This allows doing another `compile/1` without discarding the currently compiled file, while protecting the first one from accidental overriding of predicates. A better way to proceed with multiple files is to create a `*.pro` project file and include in it all the files needed for the project using the `:-[myFile]` directive. BinProlog's built-in `make` facility (using `scompile/1`) ensures quick compilation of a project by only recompiling files that have been edited.

## 6 Editing, `help/1`, `info/1`, `apropos/1`, `trace/1`, `spy/1`, `nosp/1`

To edit a file and then compile it use:

```
?- edit(<editor>,<file>).
```

To edit and recompile the currently compiled file using the `emacs` editor, attached to environment variables `EDITOR` or `VISUAL` type:

```
?- ed.
```

To edit and recompile the currently compiled file using the `edit` editor (under DOS) type:

```
?- edit.
```

To simply recompile the last file type:

```
?- co.
```

The debugger/tracer uses R.A. O'Keefe's public domain meta-interpreter. You can modify it in the file "extra.pl".

DCG-expansion is supported by the public domain file `dcg.pl`.

To debug a file type:

```
?- reconsult(FileName).
```

and then

```
?- trace(Goal).
```

For interactivity, both the toplevel and the debugger depend on

```
?-interactive(yes).
```

or

```
?-interactive(no).
```

My personal preference is using `interactive(no)` within a scrollable window. However, as traditionally all Prologs hassle the user after each answer BinProlog will do the same by default.

*If you forget the name of some builtin, `apropos/1` (or `help/1`) will give you some (flexible up to one misspelled or missing letter) matches with their arities, while `info/1` will give you online information on builtins.*

BinProlog allows debugging of dynamically recompiled code. Load your file with `reconsult` or one of its variants (`dconsult`, `oconsult`). You can benefit from efficient execution of code you do not want to look into which gets compiled on the fly during the debugging session, while being able to use `trace/1` and `listing/1` to see your predicates.

BinProlog also features some new tracing options, i.e:

```
ENTER ==> call without tracing
l ==> listing
q,a ==> abort
p ==> toplevel Prolog query
t ==> succeed, but do not call this goal
f ==> fail, and do not call this goal
k ==> keep goal for further inspection
s ==> show saved goals instances
h ==> help
; ==> continue (default)
```

The following terminal session shows an example of debugging session:

```
?-reconsult(allperms).
consulting(../progs/allperms.pl)
consulted(../progs/allperms.pl)
time(consulting = 50,quick_compiling = 0,static_space = 0)
yes
?- interactive(no).
yes
?- trace(g0(3)).
Call: g0(3)
!!! clause: g0/1
Call: nats(1,3,_x2770)
```

```

!!! clause: nats/3
Call: 1 < 3
  !!! compiled((<)/2)
Exit: 1 < 3
Call: _x3157 is 1+1
  !!! compiled((is)/2)
Exit: 2 is 1+1
Call: nats(2,3,_x3159)
  !!! clause: nats/3
  Call: 2 < 3
    !!! compiled((<)/2)
  Exit: 2 < 3
  Call: _x4115 is 2+1
    !!! compiled((is)/2)
  Exit: 3 is 2+1
  Call: nats(3,3,_x4117)
    !!! clause: nats/3
  CUT
  Exit: nats(3,3,[3])
  Exit: nats(2,3,[2,3])
Exit: nats(1,3,[1,2,3])
Call: perm([1,2,3],_x2773)
  !!! clause: perm/2
  Call: perm([2,3],_x5527)
    !!! clause: perm/2
  Call: perm([3],_x5905)
    !!! clause: perm/2
  Call: perm([],_x6283)
    !!! clause: perm/2
  Exit: perm([],[])
  Call: insert(3,[],_x5905)
    !!! clause: insert/3
  Exit: insert(3,[],[3])
  Exit: perm([3],[3])
  Call: insert(2,[3],_x5527)
    !!! clause: insert/3
  Exit: insert(2,[3],[2,3])
  Exit: perm([2,3],[2,3])
  Call: insert(1,[2,3],_x2773)
    !!! clause: insert/3
  Exit: insert(1,[2,3],[1,2,3])
Exit: perm([1,2,3],[1,2,3])
Call: fail
  !!! compiled(fail/0)
Fail: fail
Redo: perm([1,2,3],[1,2,3])
  Redo: insert(1,[2,3],[1,2,3])
    Call: insert(1,[3],_x7793)
      !!! clause: insert/3
    Exit: insert(1,[3],[1,3])
  Exit: insert(1,[2,3],[2,1,3])
Exit: perm([1,2,3],[2,1,3])
Call: fail
  !!! compiled(fail/0)

```

```

Fail: fail
Redo: perm([1,2,3],[2,1,3])
Redo: insert(1,[2,3],[2,1,3])
Redo: insert(1,[3],[1,3])
Call: insert(1,[],_x8180)
!!! clause: insert/3
Exit: insert(1,[],[1])
Exit: insert(1,[3],[3,1])
Exit: insert(1,[2,3],[2,3,1])
Exit: perm([1,2,3],[2,3,1])
Call: fail
!!! compiled(fail/0)
Fail: fail
Redo: perm([1,2,3],[2,3,1])
Redo: insert(1,[2,3],[2,3,1])
Redo: insert(1,[3],[3,1])
Redo: insert(1,[],[1])
Fail: insert(1,[],_x8180)
Fail: insert(1,[3],_x7793)
Fail: insert(1,[2,3],_x2773)
Redo: perm([2,3],[2,3])
Redo: insert(2,[3],[2,3])
Call: insert(2,[],_x7404)
!!! clause: insert/3
Exit: insert(2,[],[2])
Exit: insert(2,[3],[3,2])
Exit: perm([2,3],[3,2])
Call: insert(1,[3,2],_x2773)
!!! clause: insert/3
Exit: insert(1,[3,2],[1,3,2])
Exit: perm([1,2,3],[1,3,2])
Call: fail
!!! compiled(fail/0)
Fail: fail
Redo: perm([1,2,3],[1,3,2])
Redo: insert(1,[3,2],[1,3,2])
Call: insert(1,[2],_x8155)
!!! clause: insert/3
Exit: insert(1,[2],[1,2])
Exit: insert(1,[3,2],[3,1,2])
Exit: perm([1,2,3],[3,1,2])
Call: fail
!!! compiled(fail/0)
Fail: fail
Redo: perm([1,2,3],[3,1,2])
Redo: insert(1,[3,2],[3,1,2])
Redo: insert(1,[2],[1,2])
Call: insert(1,[],_x8542)
!!! clause: insert/3
Exit: insert(1,[],[1])
Exit: insert(1,[2],[2,1])
Exit: insert(1,[3,2],[3,2,1])
Exit: perm([1,2,3],[3,2,1])
Call: fail

```

```

!!! compiled(fail/0)
Fail: fail
Redo: perm([1,2,3],[3,2,1])
Redo: insert(1,[3,2],[3,2,1])
Redo: insert(1,[2],[2,1])
Redo: insert(1,[],[1])
Fail: insert(1,[],_x8542)
Fail: insert(1,[2],_x8155)
Fail: insert(1,[3,2],_x2773)
Redo: perm([2,3],[3,2])
Redo: insert(2,[3],[3,2])
Redo: insert(2,[],[2])
Fail: insert(2,[],_x7404)
Fail: insert(2,[3],_x5527)
Redo: perm([3],[3])
Redo: insert(3,[],[3])
Fail: insert(3,[],_x5905)
Redo: perm([],[])
Fail: perm([],_x6283)
Fail: perm([3],_x5905)
Fail: perm([2,3],_x5527)
Fail: perm([1,2,3],_x2773)
Redo: nats(1,3,[1,2,3])
Redo: nats(2,3,[2,3])
Redo: nats(3,3,[3])
Fail: nats(3,3,_x4117)
Redo: 3 is 2+1
Fail: _x4115 is 2+1
Redo: 2 < 3
Fail: 2 < 3
Fail: nats(2,3,_x3159)
Redo: 2 is 1+1
Fail: _x3157 is 1+1
Redo: 1 < 3
Fail: 1 < 3
Fail: nats(1,3,_x2770)
Exit: g0(3)

?- interactive(yes).

?- trace(insert(1,[2,3],Res)).
Call: insert(1,[2,3],_x2407) <ENTER=call, ;=trace, h=help>: 1

% dynamic: insert/3:
insert(A,B,[A|B]).
insert(B,[A|C],[A|D]) :-
    insert(B,C,D).

!!! clause: insert/3
Exit: insert(1,[2,3],[1,2,3])
Res=[1,2,3];

Redo: insert(1,[2,3],[1,2,3])
Call: insert(1,[3],_x3149) <ENTER=call, ;=trace, h=help>: ;

```

```

!!! clause: insert/3
Exit: insert(1,[3],[1,3])
Exit: insert(1,[2,3],[2,1,3])
Res=[2,1,3]q

```

?-

Starting with version 3.08 spy/1 and nospy/1 allow to watch entry and exit from compiled predicates. Note that they should be in the file to be compiled, before any use of the predicate to be spied on as in:

```

% FILE: jbond.pl
:-spy a/1.
:-spy c/1.

```

```

b(X):-a(X),c(X).

```

```

a(1).
a(2).

```

```

c(2).
c(3).

```

This gives the following interaction:

```

?-[jbond].
.....
?- b(X).

Call: a(_2158) <enter=call, other=trace>: ;
!!! compiled(a/1)
Exit: a(1)
Call: c(1) <enter=call, other=trace>: ;
!!! compiled(c/1)
Fail: c(1)
Redo: a(1)
Exit: a(2)
Call: c(2) <enter=call, other=trace>: ;
!!! compiled(c/1)
Exit: c(2)
X=2;

Redo: c(2)
Fail: c(2)
Redo: a(2)
Fail: a(_2158)

```

no

Although these are very basic debugging facilities you can enhance them at your will and with some discipline in programming they may be all you really need. Anyway, future of debugging is definitely not by tracing. One thing is to have stronger static checking. In

dynamic debugging the way go is to have a database of trace-events and then query it with high level tools. We plan to add some non-tracing database-oriented debugging facilities in the future.

## 6.1 Debugging dynamically compiled code

Let's suppose we load a file, let's say `progs/cal.pl` (a calendar benchmark) in interpreted mode with:

```
?- ~cal.
```

After running the benchmark (which, by the way gets such a big speed-up through dynamic recompilation that it actually executes faster than the benchmark witness empty loop) we would like to spy on a predicate, with:

```
?-listing(day_of_week).
?-spy(day_of_week/4).
?-go(1). % toplevel goal
```

Spy works as if no compilation has taken place! The trick is that BinProlog can switch back to the interpreted version quite easily as it has both in its working memory. You can at any time use something like `make_static/1` or `make_dynamic/1` to explicitly chose the desired execution format of a consulted predicate. This is a major improvement on debugging programs, over previous versions of BinProlog.

```
?- go(1).
Call: day_of_week(1992,12,1,_x2474) <ENTER=call, ;=trace, h=help>: ;
  !!! clause: day_of_week/4
Call: cal_key(12,_x2947,_x2948) <ENTER=call, ;=trace, h=help>: ;
  !!! clause: cal_key/3
Exit: cal_key(12,4,0)
.....
.....
```

## 7 Source-level stateless modules

```
(module)/1
current_module/1
is_module/1      - checks/generates an existing module-name
module_call/2, ':'/2 - calls a predicate hidden in a module
module_name/3    - adds the name of a module to a symbol
module_predicate/3 - adds the name of a module to a goal
modules/1       - gives the list of existing modules
```

The following example:

```
:-module m1.
:-public d/1.

a(1).
a(2).
a(3).
```

```

a(4).

d(X):-a(X).

:-module m2.

:-public b/1.

b(X):-c(X).

c(2).
c(3).
c(4).
c(5).
c(6).

:-module m3.

:-public test/1.

test(X):-b(X),d(X).

:-module user.

go:-modules(Ms),write(Ms),nl,fail.
go:-test(X),write(X),nl,fail.

```

Executing goal 'go' will generate the following output:

```

[user/0,m1/0,m2/0,m3/0]
2
3
4

```

Starting with version 3.30, predicates in the BinProlog system itself which are not intended to be used by applications, are hidden in the module `prolog` but can be accessed by calling them with

```
'prolog:my_predicate'(...)
```

Explicit naming of the module where the hidden predicate is defined should be used when `call/1`, `findall/3` etc. uses a hidden predicate, even if it is in the module itself.

This draconian constraint is motivated by simplicity of BinProlog's stateless purely source-level module system. Basically predicates in a module have their names prefixed as in

```
my_current_module:my_predicate
```

in a preprocessing step, except if they are declared `public` or are known to the system as being so (i.e. in the case of builtins).

This basic concept of modules (essentially the same as what can be achieved with `extern` and `static` declarations in C) covers only compiled code, and is mostly intended to ensure

multiple name spaces with a very simple semantics and no additional space or time overhead. On the other hand use of linear and intuitionistic implication is suggested for dynamic modular and hypothetical reasoning constructs.

*Meta-predicate* declarations are not supported at this time (mostly because they are at least as cumbersome as just putting the right name extension in argument positions which require it, but they might be added in the future if a significant number of users will ask to have them.

Note that builtins and predicates defined in a special module `user` are always public. A public predicate keeps its name unchanged in the global name space while hidden predicates have their names prefixed by the name of the module in their definitions and in all their statically obvious (first-order) uses.

Alternatively, `module/2` allows to define a module and its public predicates with one declaration as in:

```
:-module(beings,[cat/4,dog/4,chicken/2,fish/0,maple/1,octopus/255]).
```

## 8 Interoperation with Jinni

BinProlog interoperates as a client or server with Jinni . Communication over sockets is very fast, around 1000 exchanges per second.

The BinProlog side API is compatible with Jinni's server and client classes and supports socket reuse:

- `rpc_server(Port,Password)`: runs Jinni compatible server with socket reuse
- `rpc_server`: runs Jinni compatible server with socket reuse on default port
- `rpc(Query)`: calls server on current local reusable socket]
- `rpc(Answer,Goal,Result)`: calls server on local reusable socket and gets back Result as the(Answer) or no

Here is a Jinni client talking to a BinProlog server:

BinProlog Server Window:

```
?-rpc_server.
```

Jinni Client Window::

```
?-new(client,C),C:ask(println(hello)),C:disconnect.  
hello
```

Here is a BinProlog client talking to a Jinni server:

Jinni Server Window:

```
?- new(server,S),S:serve.  
hello
```

BinProlog Client Window:

```
?- new_client(C),ask(C,println(hello)),stop_service(C).
```

## 9 Threads

Windows and Linux versions of BinProlog also support multithreaded execution. Type `help(thread)`, then use `info/1` for more information on thread related operations.

In fact, for most programs just using

```
bg(Goal)
```

```
and
```

```
synchronize(Statement)
```

are all you need to get started. Take a look at the StockMarket Web-based demo at

```
http://www.binnetcorp.com
```

for a more interesting example using multiple threads. BinNet's *online demo* page contains a naive reverse benchmark allowing the user to measure BinProlog's performance with various thread/list length/iterations by filling in a Web form.

On a Pentium 4 2.4 GHz, depending on the parameters, BinProlog achieves between 120 and 130 million inferences/second (MegaLIPS).

## 10 Calling BinProlog as a DLL

The library `bp_lib.dll` (available to users with BinProlog Source or BinProlog Professional License) is called with parameters similar to BinProlog's command line.

When a C string parameter parameter like:

```
"call((consult(hello),go,halt))"
```

is passed to the function `bp_main()` exported by the DLL, BinProlog consults the file `hello.pro`, starts with the goal `go` and finally halts.

For more complex interaction with DLLs, we refer to BinProlog's C interface described in [3]. More complex interaction patterns may require some adaptation of the BinProlog sources, i.e. exporting some other functions and calling back through new BinProlog builtins. A Visual C/C++ project file is provided to simplify working with the sources as well as command line makefiles (see directory `BP_DLL`).

## 11 Exceptions

BinProlog support ISO Prolog exception handling. Throwing an exception specified with `throw/1` has the result that the control leaves the current point and the stack is searched until a matching `catch/3` is found (a default one waits at toplevel for uncought exceptions). `Catch(Goal,ExceptionPattern,Action)`, executes `Goal` and if it catches an exception matching `ExceptionPattern`, it executes `Action`, like in the following example:

```
?- catch(
    (for(I,1,100),I=10,throw(boo(I)),println(too_late)),
    X,
    println(got(X))
).
```

```
got(boo(10))
I=_x2316,
X=boo(10)
```

yes

## 12 Using the Redistributable BinProlog Runtime Executable

Starting with version 7.20, owners of BinProlog Professional Edition can distribute their applications in binary format, together with the BinProlog Runtime Executable (bpr.exe on Windows, bor or bpr.*platform* on Unix).

To build a binary application do the following. Create a file, for instance **myfile.pl**, possibly including various modules in it with

```
:-[file1].
...
:-[fileN].
```

Compile the myfile.pl file to a corresponding **myfile.wam** file, using **fcompile(myfile)** or **scompile(myfile)**. The later contains a builtin *make* facility, for upgrading only the modules which have changed, while also loading the file into the development environment at the end.

To execute the resulting **myfile.wam** binary application, type:

```
bpr <myfile>.wam
```

The BinProlog Runtime Executable (bpr.exe or bpr) has all the abilities of BinProlog, except the ability to compile applications to files or memory. However, it is able to consult and interpret Prolog data files, by asserting their content into the dynamic database. Executing complex code with the interpreter is, however, much slower than compiling, please make sure that the code you deliver has been previously compiled to the \*.wam form, which is executed by the runtime system at full speed.

## 13 Example programs

The directory `progs` contains a few BinProlog benchmarks and applications.

```
allperms.pl: permutation benchmarks with findall
bestof.pl:  an implementation of bestof/3

bfmeta.pl:  breadth-first metainterpreter
backprop.pl: float intensive neural net learning by back-propagation
cal.pl:     calendar: computes the last 10000 fools-days
chat.pl:    CHAT parser
choice.pl:  Choice-intensive ECRC benchmark
cbrev.pl:   nrev with ^/2 as a constructor instead of ./2
cube.pl:    E. Tick's benchmark program
fibonacci.pl: naive Fibonacci
```

```

ffibo.pl:    naive Fibonacci with floats
mfibo.pl:   Fibonacci with memoing
dfibo.pl:   Fibonacci with Delphi memoing
hello.pl:   example program to create stand-alone Unix application
knight.pl:  knight tour to cover a chess-board (uses the bboard)
lknight.pl: knight tour to cover a chess-board (uses the lists)
ltak.pl:    tak program with lemmas
lfibo.pl:   fibo program with lemmas
lq8.pl :    8 queens using global logical variables
maplist.pl: fast findall based maplist predicate
brev.pl:    naive reverse benchmark
nrev30.pl:  small nrev benchmark to reconsult for the meta-interpreter
or.pl:      or-parallel simulator for binary programs (VT100)
other_bm*:  benchmark suite to compare Sicstus, Quintus and BinProlog
puzzle.pl:  king-prince-queen puzzle
q8.pl:      fast N-queens
qrev.pl:    quick nrev using builtin det_append/3
subset.pl:  findall+subset
tetris.pl:  tetris player (VT100)

```

## 14 Appendix

### 14.1 Default Operator Definition

Note the operators when used as non-operator atoms *need to be in paranthesises*, i.e. you should say `write([this,(module), is, (not), in, the,(public),domain])` instead of `write([this,module, is, not, in, the,public,domain])`.

BinProlog's default operator definitions (see file `oper.pl`) are the following:

```

:-op(1000,xfy,'').
:-op(1100,xfy,(';')).

:-op(1200,xfx,('-->')).
:-op(1200,xfx,(':-')).
:-op(1200,fx,(':-')).
:-op(700,xfx,'is').
:-op(700,xfx,'=').

:-op(1050,xfx,(@@)).

:-op(500,yfx,'-').
:-op(200,fy,'-').

:-op(500,yfx,'+').
:-op(200,fy,'+').

:-op(400,yfx,'/').
:-op(400,yfx,'*').
:-op(400,fx,'*').
:-op(400,yfx,(mod)).
:-op(200,yfx,(**)).
:-op(200,xfy,('^')).

```

```

:-op(300,fy,(~)).
:-op(650,xfy,'.').
:-op(660,xfy,'++').

:-op(700,xfx,'>=').
:-op(700,xfx,'>').
:-op(700,xfx,'=<').
:-op(700,xfx,<).
:-op(700,xfx,(=\=)).
:-op(700,xfx,(=:)).

:-op(400,yfx,>>).
:-op(400,yfx,<<).
:-op(400,yfx,(/)).

:-op(200,yfx,(\)).
:-op(200,yfx,(/\)).
:-op(200,yfx,(\)).
:-op(200,fx,(\)).

:-op(700,xfx,(@>=)).
:-op(700,xfx,(@=<)).
:-op(700,xfx,(@>)).
:-op(700,xfx,(@<)).

:-op(700,xfx,(\=)).
:-op(700,xfx,(=)).
:-op(700,xfx,(=.)).
:-op(700,xfx,(\=)).

:-op(900,fy,(not)).
:-op(900,fy,(\+)).
:-op(900,fx,(spy)).
:-op(900,fx,(nospy)).

:-op(950,fx,(##)).

:-op(950,xfy,(=>)).
:-op(950,xfx,<=).

:-op(1050,xfy,(->)).

:-op(1150,fx,(dynamic)).
:-op(1150,fx,(public)).
:-op(1150,fx,(module)).
:-op(1150,fx,(mode)).

:-op(1150,fx,(multifile)).
:-op(1150,fx,(discontiguous)).

:-op(1200,xfx,(:-)).

:-op(50,yfx,(:)).

```

`:-op(100,fx,(@)).`

`:-op(50,fx,(^)).`

`:-op(500,fx,(#>)).`

`:-op(500,fx,(#<)).`

`:-op(500,fx,(#:)).`

`:-op(500,fx,(#+)).`

`:-op(500,fx,(#*)).`

`:-op(500,fx,(#=)).`

`:-op(500,fx,(#-)).`

`:-op(500,fx,(#?)).`

## 15 Further readings

Related BinProlog documentation is available at: [4, 2, 3, 1].

## References

- [1] P. Tarau. BinProlog 7.0 Professional Edition: Predicate Cross-Reference Guide . Technical report, BinNet Corp., 1998. Available from <http://www.binnetcorp.com/BinProlog>.
- [2] P. Tarau. BinProlog 9.x Professional Edition: Advanced BinProlog Programming and Extensions Guide. Technical report, BinNet Corp., 2002. Available from <http://www.binnetcorp.com/BinProlog>.
- [3] P. Tarau. BinProlog 9.x Professional Edition: BinProlog Interfaces Guide. Technical report, BinNet Corp., 2002. Available from <http://www.binnetcorp.com/BinProlog>.
- [4] P. Tarau. BinProlog 9.x Professional Edition: User Guide. Technical report, BinNet Corp., 2002. Available from <http://www.binnetcorp.com/BinProlog>.