Jekejeke Runtime Frequent

Version 1.3.4, December 29th, 2018

XLOG Technologies GmbH

Jekejeke Prolog

Runtime Library 1.3.4

Frequent Predicates

Author: XLOG Technologies GmbH
Jan Burse
Freischützgasse 14
8004 Zürich
Switzerland

Date: December 29th, 2018
Version: 0.28

Warranty & Liability

To the extent permitted by applicable law and unless explicitly otherwise agreed upon, XLOG Technologies GmbH makes no warranties regarding the provided information. XLOG Technologies GmbH assumes no liability that any problems might be solved with the information provided by XLOG Technologies GmbH.

Rights & License

Trademarks

Table of Contents

Change history

Jan Burse, June 22th, 2014, 0.1:

- Initial Version.

Jan Burse, August 17th, 2014, 0.2:

- The modules expand and simp introduced.

Jan Burse, October 10th, 2014, 0.3:

- Package stream and standard moved to here, modules random and shell introduced.

Jan Burse, January 1st, 2015, 0.4:

- The module lists improved and the text simp demodulified.

Jan Burse, Mai 6th, 2015, 0.5:

- Improved URI handling and new locale module.

Jan Burse, July 9th, 2015, 0.6:

- Some additions to stream/console and system/locale.

Jan Burse, August 5th, 2015, 0.7:

- DCG extensibility improved.

Jan Burse, September 2th, 2015, 0.8:

- New module surrogate and sequence.

Jan Burse, October 20th, 2015, 0.9:

- Improved resource handling and new proxy module.

Jan Burse, January 2nd, 2016, 0.10:

- Functional improvements to the proxy module.

Jan Burse, February 17th, 2016, 0.11:

- Non-functional improvements to the proxy module.

Jan Burse, March 8th, 2016, 0.12:

- Broader scope of clause references.

Jan Burse, April 12th, 2016, 0.13:

- New miscellaneous definitions section and new indexes section.

Jan Burse, May 15th, 2016, 0.14:

- New module text introduced.

Jan Burse, June 27th, 2016, 0.15:

- New modules maps and ordmaps introduced.

Jan Burse, August 6th, 2016, 0.16:

- New module “clean” introduced.

Jan Burse, October 28th, 2016, 0.17:

- String pattern matching framework introduced.

Jan Burse, December 15th, 2016, 0.18:

- Aggregates moved to Jekejeke Minlog.

Jan Burse, April 23th, 2017, 0.19:

- Some improvements in the residue module.

Jan Burse, July 18th, 2017, 0.20:

- Some improvements in the XML module.

Jan Burse, December 13th, 2017, 0.21:

- Complex types in the DOM parser and un-parser.

Jan Burse, May 21th, 2018, 0.23:

- Improve Java proxy classes API and new module score.

Jan Burse, October 22th, 2018, 0.25:

- Some improvements.

Jan Burse, November 19th, 2018, 0.26:

- New modules "socket" and "group" introduced.

Jan Burse, December 26th, 2018, 0.27:
- Some improvements.

Jan Burse, December 29th, 2018, 0.28:
- New modules "json" and "http".

1 Introduction

The document describes the collection of modules that is bundled with the Jekejeke Prolog runtime library and that provide frequently used predicates.

- **Frequent Examples:** Some examples that make use of Runtime frequent predicates.
- **Frequent Conversations:** Typical interpreter interactions for loading modules.
- **Frequent Syntax:** t.b.d.
- **Frequent Theories:** The theories that group the Runtime frequent predicates.
- **Appendix Example Listing:** The full source code of Prolog examples is given.

2 Frequent Examples

In the following we show some examples that make use of Runtime frequent predicates.

- **Flag Example:** The Jekejeke Prolog programming language provides higher order programming. The given example shows how to define a loop construct.

- **Palindrome Example [ISO]:** Grammar rules are also included in the Jekejeke Prolog programming language. The given example allows detecting and generating palindromes.

- **Fruits Example:** The higher order programming of Jekejeke Prolog is also available for grammar rules. The given example shows how to define a list construct.

2.1 Flag Example

The Jekejeke Prolog programming language provides higher order programming. The given example shows how to define a loop construct. The goal of this example is to return a star banner. In a first step we will only display the star banner on the console which is simpler than returning a star banner. On purpose we will first build a loop construct so that we can use FORTRAN like loops to solve our problem. The displayed star banner should have size 8x8 and look as follows:

```
xoxoxoxo
oxoxoxox
...
xoxoxoxo
oxoxoxox
```

To enumerate the x- and y-axis we can define a predicate between/3 which will enumerate integers in a given range. The detailed Prolog text of this predicate can be found in the appendix together with the rest of this example. We then use the between/3 predicate to define a loop construct as follows:

```
% for1(+Integer, +Integer, +Closure)
for1(Lo, Hi, Closure) :-
  between(Lo, Hi, Value),
  call(Closure, Value),
  fail.
for1(_,_,_).
```

The above loop construct uses the higher order programming predicate call/n to invoke the given closure. Since 2012 this predicate is part of the ISO Prolog standard via the corrigendum 2. The closure is supposed to be the loop body and the identification of the loop variable. The loop construct will then repeatedly invoke the loop body with integer values ranging from the given lower bound to the given upper bound, both inclusively. A solution to the flag display problem is quickly coded by using the system predicates write/1 and nl/0.

```
% flag
flag :-
  for1(1,8,X\
     for1(1,8,Y\
        (0 =:= (X+Y) mod 2 -> write(x); write(o))), nl)).
```

The above solution makes use of another higher order programming predicate besides call/n. We also find the abstraction operator (\)/2 to create a closure. This use of the abstraction operator (\)/2 is Jekejeke Prolog specific and not part of some ISO Prolog standard. One characteristic of the above solution is the peculiar use of backtracking. The image of the flag is generated via backtracking and will thus not persist.

Let's now turn to the question how we could return a star banner instead of only displaying it? We want a solution where 'x' and 'o' are assigned to some variable and then returned. But to make this happen we are not allowed to backtrack over the assignment step, since we would then loose the binding. We must therefore chain the closure invocations instead of backtracking over them. Here is our take for an alternative for loop:

```
% for2(+Integer, +Integer, +Closure)
for2(Lo, Hi, _) :- Lo > Hi, !.
for2(Lo, Hi, Closure) :-
```

```
   call(Closure, Lo),
   Lo2 is Lo+1,
   for2(Lo2, Hi, Closure).
```

The above predicate does not anymore make use of a predicate between/3. Instead it does do the bounds check by itself. There is now no more failure when we are outside of the bounds, instead there is some success and a no-op concerning the closure. When we are inside the bounds the closure is invoked and in the same time a recursion into the new loop predicate happens again. Any bindings that are done by the invocation of the closure will not be lost during the next iteration.

We can now go on and for example populate an array in each invocation of the closure via instantiation and will not lose this instantiation. For arrays we do not provide some special constructs but rely on the ISO Prolog standard defined predicates. We simply view arrays as compounds with irrelevant functors. An array with not instantiated elements can be created via the system predicate functor/3:

```
?- functor(X,'',8).
X = ''(_A,_B,_C,_D,_E,_F,_G,_H)
```

An element of an array can be accessed via the system predicate arg/3. Please note that the first argument of a compound and thus an array has the index value 1:

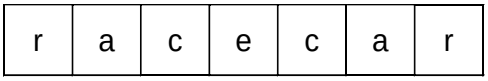
```
?- functor(X,'',8), arg(3,X,x), arg(4,X,o).
X = ''(_A,_B,x,o,_E,_F,_G,_H)
```

The full code of the flag solution that will return a flag as a 2-dimensional array can be found in the appendix. Running the code yields the following result:

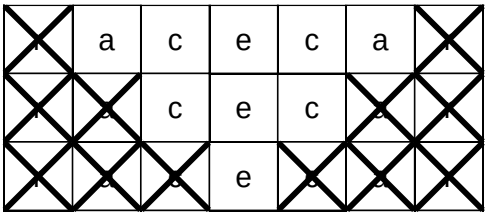
```
?- flag(X).
X = ''(''(x,o,x,o,x,o,x,o),
''(o,x,o,x,o,x,o,x),
''(x,o,x,o,x,o,x,o),
''(o,x,o,x,o,x,o,x),
''(x,o,x,o,x,o,x,o),
''(o,x,o,x,o,x,o,x),
''(x,o,x,o,x,o,x,o),
''(o,x,o,x,o,x,o,x))
```

2.2 Palindrome Example [ISO]

Grammar rules are also included in the Jekejeke Prolog programming language. The given example allows detecting and generating palindromes. A palindrome is a string that reads the same in the reverse. We can again invoke a graphical intuition based on a sieve to explain our algorithm. In a first step we list the characters of the word, here “racecar”:



The algorithm works outside in by crossing out uncrossed characters. A left outer character and a right outer character can be crossed out when they are the same. If all characters have been crossed out or only one character is left, we have detected a palindrome.



Prolog grammar rules by default work on lists of terminals. Our terminals here are the characters. We define one non-terminal palin that should be able to detect palindromes. The corresponding grammar rules are:

```
palin --> [_].
palin --> [Middle, Middle].
palin --> [Border], palin, [Border].
```

The first rule is one of our stopping conditions. We stop when no more characters are left. The second rule expresses that we stop when a single character is left, whereby we don’t care which character it is. The last rule expresses our crossing out step and the continuation of the palindrome check. Grammar rules can be invoked via the phrase/2 predicate:

```
?- phrase(palin, "racecar").
Yes
?- phrase(palin, "anna").
Yes
?- phrase(palin, "bert").
No
```

Grammar rules might also return attributes. We simply have to extend the non-terminal palin by further arguments. Our plan is to return the middle element, if it exists and the border elements whereby we do not duplicate them. The corresponding grammar rules are:

```
palin([], [Middle]) --> [Middle].
palin([Middle], []) --> [Middle, Middle].
palin([Border | List], Middle) --> [Border], palin(List, Middle), [Border].
```

In the first rule we encode the nonexistence of a middle element by an empty list. In the second rule we encode the middle element by a singleton list. The last rule collects the border elements. The grammar rules are again invoked via the phrase/2 predicate:

```
?- phrase(palin(X1, Y1), "racecar"), atom_codes(X2,X1), atom_codes(Y2,Y1).
X1 = [114, 97, 99],
Y1 = [101],
X2 = rac,
Y2 = e
?- phrase(palin(X1, Y1), "anna"), atom_codes(X2,X1), atom_codes(Y2,Y1).
X1 = [97, 110],
Y1 = [],
X2 = an,
Y2 = ''
?- phrase(palin(X1, Y1), "bert"), atom_codes(X2,X1), atom_codes(Y2,Y1).
No
```

But grammar rules can not only be used to detect strings or to extract information from strings. It is also possible to use grammar rules to generate strings. In many cases we it is enough to simply use the grammar attributes as input and the strings as output.

```
?- phrase(palin("ra", "d"), X1), atom_codes(X2,X1).
X1 = [114, 97, 100, 97, 114],
X2 = radar
```

The presented examples should work across a great range of Prolog systems. Although there is not yet a definite DCG standard most of the Prolog systems support DCGs.

2.3 Fruits Example

The higher order programming of Jekejeke Prolog is also available for grammar rules. The given example shows how to define a list construct. In the following we will explore how to define a custom grammar construct for repetition, which in turn will be used to define a list. Repetition is not found in definite clause grammar. Other grammar formalisms typically define a construct for repetition since it is a recurring pattern.

For instance the Extended Backus-Naur Form (EBNF) allows curly braces to denote repetitions. Our working example will be the grammar for a fruit list:

```
fruits :== fruit { "," fruit }. /* EBNF */
```

In definite clause grammars curly braces do not denote repetition. They are reserved for auxiliary conditions. We will pick the non-terminal repetition//1 and give it the phrase that will be repeated as an argument. The corresponding code looks as follows:

```
repetition(G) --> G, repetition(G).
repetition(_) --> [].
```

We can now remodel the EBNF grammar in DCG. We omit the fruit non-terminal. The details of it can be found in the appendix. The fruits non-terminal then reads as follows:

```
fruits --> fruit, repetition((",", fruit)).
```

Here are some example queries:

```
?- phrase(fruits,"appleorange").
No
?- phrase(fruits,"apple,orange,apple").
Yes
```

In a next step we will extend the DCG by attributes. The repetition construct should take as an argument a phrase G(X) with a parameter X and it should yield the list $[X_1, .., X_n]$ of parameter instances that occur during the repetition. Instead of G(X) we can only write call(G,X) in Prolog. Therefore we will formulate repetition//2 as follows:

```
repetition(G,[X|Y]) --> call(G,X), repetition(G,Y).
repetition(_,[]) --> [].
```

The fruits production starts with one fruit and the repeats zero, one or many fruits. We can access the first one fruit by fruit(X). The fruits in the repetition of (“,”, fruit(Z)) are identified by the parameter Z. We will use the binder Z\ to pass the parameter to the repetition construct. The new fruits non-terminal then reads as follows:

```
fruits([X|Y]) --> fruit(X), repetition(Z\(",", fruit(Z)),Y).
```

By using higher order we don’t lose the bi-directionality of DCG productions. We can use the new non-terminal to parse fruit texts:

```
?- phrase(fruits(L),"appleorange").
No
?- phrase(fruits(L),"apple,orange,apple").
L = [apple, orange, apple]
```

And we can use the new non-terminal to generate fruit texts:

```
?- phrase(fruits([apple, orange]),X), atom_codes(A,X).
A = 'apple,orange',
X = [97, 112, 112, 108, 101, 44, 111, 114, 97, 110, 103, 101]
```

The example without attributes should work across a great range of Prolog systems. Although there is not yet a definite DCG standard most of the Prolog systems support DCGs with goal parameters.

On the other hand the example with attributes might not immediately work in a Prolog system different from Jekejeke Prolog. Even if the Prolog system has higher order programming in the form of call/n and some variable binder, the higher order constructs might still fail inside a DCG grammar.

3 Frequent Conversations

The section shows typical interpreter interactions for loading modules. Jekejeke Prolog features an intelligent loader based on a dependency model. Further there is a module system in place which provides name spaces and protection.

- **Ensure Loaded:** Ensure loaded is recommended to load modules and Prolog texts from within a Prolog text or from within the top level.
- **Make:** The intelligent loader is capable to automatically update changed modules and Prolog texts. This allows the integration with development environments.
- **Unload File:** Loaded modules and Prolog texts can be disconnected and a kind of garbage collection will reclaim all unused modules and Prolog texts.
- **Compatibility Matrix:** We compare our approach to some existing Prolog systems that provide intelligent loaders and module systems.

3.1 Ensure Loaded

Ensure loaded is recommended to load modules and Prolog texts from within a Prolog text or from within the top level. This predicate will load a module or Prolog text only if necessary.

3.2 Make

The intelligent loader is capable to automatically update changed modules and Prolog texts. This allows the integration with development environments.

3.3 Unload File

Loaded modules and Prolog texts can be disconnected and a kind of garbage collection will reclaim all unused modules and Prolog texts.

3.4 Compatibility Matrix

We compare our approach to some existing Prolog systems that provide intelligent loaders and module systems:

Table 1: Compatibility Matrix for Interactions

Nr	*Description*	*System*
1	t.b.d.	
2	t.b.d.	
3	t.b.d.	
4	t.b.d.	

4 Frequent Syntax

We mention here syntax that is introduced by the modules. Some modules directly introduce new syntax operators. Some modules indirectly introduce new rule or goal formats by corresponding expansion rules.

- **Term Syntax:** Syntax operators introduced by the frequent predicates.
- **Text Syntax:** Rule or goal formats introduced by the frequent predicates.
- **Miscellaneous Definitions:** The interpreter keeps track of flags and properties.

4.1 Term Syntax

This syntax describes the grammar that forms terms from token sequences. We find the following topics:

- **Compatibility Matrix:** t.b.d.

Compatibility Matrix

t.b.d.

Table 2: Compatibility Matrix for the Token Syntax

Nr	*Description*	*System*
1	t.b.d.	
2	t.b.d.	
3	t.b.d.	
4	t.b.d.	

4.2 Text Syntax

This syntax describes the grammar that forms Prolog texts and queries from term sequences. We find the following topics:

- **Grammar Rules:** Grammar rules support parsing and un-parsing of phrases.
- **Compatibility Matrix:** t.b.d.

Grammar Rules

Rules based on the (-->)/2 functor are historically called definite clause grammars (DCG). The term definite refers to the absence of negation. But this is not the case anymore nowadays, since grammar rules might also include a grammatical negation. The head predicate of a grammar rule represents a non-terminal. The predicates in the body of a grammar rule normally refer to other non-terminals. The list and the set construct are used to escape this interpretation. The form of grammar rules can be summarized as follows:

```
clause'      --> gr_clause
               | head ":-" body
               | head.
gr_rule      --> gr_head "-->" gr_body.

gr_body      --> gr_goal [ "," gr_body ].
gr_goal      --> set
               | list
               | non-terminal | variable
               | gr_body "->" gr_body
               | gr_body ";" gr_body
               | "call(" non-terminal "," arguments ")"
               | "!" | "fail" | "\\+" gr_goal.

gr_head      --> non_terminal "," gr_body
               | non_terminal.

non-terminal --> callable.
```

Examples:

```
zero --> "0".                % is a grammar rule.
s(s(X,Y)) --> np(X), vp(Y).  % is a grammar rule.
```

The list construct allows referring to terminals, and the set construct allows referring to normal rule bodies. The terminals in a list construct need not only be character codes. They can be arbitrary terms, so as to allow higher level parsing and un-parsing. Again the head of a grammar rule has to be a callable. It is not possible to have grammar rules for numbers or variables. Double quoted strings are short hands for lists of character codes, and they will therefore be interpreted as character code terminals in the body of a grammar rule.

Compatibility Matrix

t.b.d.

Table 3: Compatibility Matrix for the Token Syntax

Nr	*Description*	*System*
1	t.b.d.	
2	t.b.d.	
3	t.b.d.	
4	t.b.d.	

4.3 Miscellaneous Definitions

The interpreter also needs to keep track of flags and properties definitions. The following flags and properties are provided by the Runtime frequent predicates:

- **Prolog Flags:** The predefined Prolog flags.
- **Predicate Properties:** The predefined predicate properties.
- **Source Properties:** The predefined source properties.
- **Atom Properties:** The predefined atom properties.

Prolog Flags

Prolog flags can be accessed via the system predicates current_prolog_flag/2 and set_prolog_flag/2. The following Prolog flags are supported by the Runtime frequent predicates:

char_conversion:	See the term input / output section.
double_quotes:	See the term input / output section.
back_quotes:	See the term input / output section.
single_quotes:	See the term input / output section.
sys_cur_input:	See the stream control section.
sys_cur_output:	See the stream control section.
sys_cur_error:	See the stream control section.
sys_mask:	See the signal handling section.
sys_choices:	See the signal handling section.
sys_variables:	See the signal handling section.
sys_cpu_count:	See the module thread section.
sys_runtime_version:	See the module thread section.
sys_random:	See the module random section.

Predicate Properties

Predicate properties can be accessed via the system predicates predicate_property/2, set_predicate_property/2 and reset_predicate_property/2. The following predicate properties are supported by the Runtime frequent predicates:

automatic:	See the module score section.
sys_noexpand:	See the module expand section.
sys_nomacro:	See the module expand section.

Source Properties

Source properties can be accessed via the system predicates source_property/2, set_source_property/2 and reset_source_property/2. The following source properties are supported by the Runtime frequent predicates:

t.b.d.

5 Frequent Theories

We have split the frequent predicates into two packages. One package with basic predicates and one package with advanced predicates.

- **Standard Package:** This package contains predicates that are either mentioned in the ISO core standard or in some of its corrigenda.
- **Basic Package:** This package contains basic predicates that are partly already used by the Prolog interpreter itself.
- **Advanced Package:** This package contains advanced predicates that can be used in various Prolog applications.
- **Experiment Package:** This package contains experimental predicates that deliver some alternative or extended realizations of commonly found predicates.
- **Stream Package:** This theory is concerned with input/output and the selection of streams.
- **System Package:** This package contains predicates that allow the access to various operating system elements.
- **Miscellaneous Package:** This package contains miscellaneous predicates.

5.1 Standard Package [partially preloaded]

This package contains predicates that are either mentioned in the ISO core standard or in some of its corrigenda, or that are to be expected to be soon published by the ISO Prolog committee. The modules of the package are nameless and preloaded. They don't needed to be loaded and cannot be loaded by the end-user. The following modules are provided:

- **Module apply [preloaded]:** This module provides of apply predicates.
- **Module bags [preloaded]:** This module provides grouped solution lists.
- **Module expand [preloaded]:** This module provides term and goal expansion.
- **Module dcg:** This module provides definite clause grammars.
- **Module signal [preloaded]:** A secondary thread can control a primary thread.
- **Module sort [preloaded]:** This module provides predicates that can sort lists.
- **Compatibility Matrix:** t.b.d.

Module apply [preloaded]

We provide function application via the predicates call/n. This predicate takes as a first argument a term which plays the role of a closure, then extends it by the remaining n-1 arguments and calls the resulting goal. Since the closure need not necessarily be an atom but can also be a compound, it is possible to create closures that carry around data.

Example:

```
?- [user].
writeln(X) :- write(X), nl.
^D
?- call(writeln, hello).
hello
```

The call/n predicates also work for qualified closures. A qualified closure is similarly extended to an unqualified closure. The arguments will be added to the inner unqualified closure and the qualification will be preserved. So extending a:b(X) by Y results in a:b(X,Y). The predicates sys_modext_args/[3..9] allow to call the argument extension without further invoking the result.

Example:

```
?- X is call(sin,0.2).
X = 0.19866933079506122
?- Y = sin, X is call(Y,0.2).
Y = sin,
X = 0.19866933079506122
```

Thanks to bridging it is also possible to use the call/n predicates inside arithmetic expressions. The bridge will turn the call/n evaluable function into a call/n+1 predicate with the desired effect.

The following apply predicates are provided:

sys_modext_args(P, Y_1, .., Y_n, Q):
: The predicate adds the arguments Y_1, .., Y_n to the callable P and unifies the result with Q. The result Q will have the same call-site information and the same colon and double notation as the callable P. The predicate is currently defined for $1 \le n \le 7$.

call(P, Y_1, .., Y_n): [TC2 8.15.4]
: The goal call(p(X_1, .., X_m), Y_1, .., Y_n) succeeds whenever the goal p(X_1, .., X_m, Y_1, .., Y_n) succeeds. The predicate is currently defined for $1 \le n \le 7$.

Module bags [preloaded]

The following predicates do a grouping of the solutions and may succeed more than once. This grouping is based on the determination of the witnesses of a solution. The witness are the global variables of T^G where T is the template and G is the goal.

Examples:

```
p(a,y). p(a,x). p(b,x).

?- bagof(X,p(X,Y),L).
Y = y, L = [a] ;
Y = x, L = [a, b]
?- bagof(X,Y^p(X,Y),L).
L = [a, a, b]
```

The predicate bagof/3 will do a sorting of the witnesses but not of the resulting lists. The variation setof/3 will sort the witnesses and the resulting lists. Finally the variation sys_heapof/3 will neither sort the witnesses nor the resulting lists.

Examples:

```
p(a). p(b).
q(a). q(b). q(c).

?- forall(p(X), q(X)).
Yes
?- forall(q(X), p(X)).
No
```

The predicate copy_term/2 can be used to copy a term. The predicate findall/3 can be used to collect a resulting list without any grouping. The elements will be copied. The forall/2 performs generate and test and can be used for a bounded universal quantification.

The following bags predicates are provided:

bagof(T, X_1^…^X_n^G, L): [ISO 8.10.2]
: The predicate determines all the solutions to the matrix G, whereby collecting copies of the template T grouped by the witnesses in a list. The predicate then repeatedly succeeds by unifying the witnesses and when L unifies with the corresponding list.

setof(T, X_1^…^X_n^G, L): [ISO 8.10.2]
: The predicate determines the same lists as the predicate bagof/3. But before returning them the lists are sorted by means of the predicate sort/2.

sys_heapof(T, X_1^…^X_n^G, L):
: The predicate determines the same lists as the predicate bagof/3. But the lists are sorted by the witnesses instead of grouped by the witnesses.

copy_term(X, Y): [ISO 8.5.4]
: The predicate creates a copy of X and succeeds when the copy unifies with Y.

findall(T, G, L): [ISO 8.10.1]

findall(T, G, L, R):
: The predicate first finds all the solutions to the goal G, whereby collecting copies of the template T in a list. The predicate then succeeds when L unifies with the list.

forall(A,B): [N208 8.10.4]
: The predicate succeeds when there is no success of A such that B fails. Otherwise the predicate fails.

Module expand [preloaded]

Clauses and goals are automatically expanded for the Prolog session queries, for the Prolog text clauses and for the Prolog text directives. The effect can be seen by the following example. The first goal_expansion/2 fact defines a new expansion for the predicate writeln/2. The next rule for the predicate hello/0 already makes use of the expansion in the body:

Example:

```
?- [user].
goal_expansion(writeln(X), (write(X), nl)).
hello :- writeln('Hello World!').
^D
?- listing(hello/0).
hello :-
    write('Hello World!'), nl.
```

The clauses are expanded with the help of the system predicate expand_term/2, which in turn expands the goals of the bodies via the system predicate expand_goal/2. The two system predicates are customizable by the end-user via additional rules for the multi-file predicates term_expansion/2 and goal_expansion/2.

The predicate property sys_noexpand/0 allows excluding a meta-predicate from the goal or term traversal. Closures are currently not expanded. If the term or goal expansion steps into the colon notation (:)/2 or the double colon notation (::)/2 with a sufficiently instantiated first argument it will look up the meta-declarations for the qualified predicate name.

The result of the expansion can be no clause, a single clause or multiple clauses. No clause is indicated by unit/0 as a result. A single clause is simply returned by itself. Multiple clauses can be conjoined by the operator (/\)/2 and returned this way. Expansion is also performed along the existential quantifier (^)/2 second argument.

Example:

```
?- op(500,yfx,++).
Yes
?- [user].
rest_expansion(X++Y,sys_cond(Z,append(X,Y,Z))).
^D
?- Y = [1,2]++[3].
Y = [1,2,3]
```

It is also possible to define rest expansion via the predicate rest_expansion/2 and to invoke rest expansion via the predicate expand_rest/2. Rest expansion is applied to goal or term arguments that are not goals or terms. Rest expansion is driven by meta function declarations and can be block by the predicate property sys_nomacro.

Rest expansion might return a result of the form sys_cond(R, C) where C is the so-called side condition. In the context of rest arguments, the side conditions are merged via conjunction to give a new side condition. In the context of a goal G, the condition C is prepended as (C,G), in the context of a term T, the condition C is appended as (T:-C).

The following expansion results are recognized:

unit:
: An empty clause list.

C /\ D:
: A clause list of two clause lists C and D.

sys_cond(T, C):
: This result during rest expansion indicates rest T and a side condition C.

The following expansion predicates are provided:

term_expansion(C, D):
: This predicate can be used to define custom term expansion rules.

expand_term(C, D):
: The system predicate succeeds if the expansion of the term C unifies with D.

goal_expansion(C, D):
: This predicate can be used to define custom goal expansion rules.

expand_goal(C, D):
: The system predicate succeeds if the expansion of the goal C unifies with D.

rest_expansion(C, D):
: This predicate can be used to define custom rest expansion rules.

expand_rest(C, D):
: The system predicate succeeds if the expansion of the rest C unifies with D.

The following predicate properties for clause expansion are provided:

sys_noexpand:
: The property indicates that the meta-predicate should not be traversed in goal or term expansion. The property can be changed for user predicates.

sys_nomacro:
: The property indicates that the meta-function should not be traversed in rest expansion. The property can be changed for user predicates.

Module dcg

Definite clause grammars (DCG) are an extension of context free grammars [5]. DCGs allow arbitrary tree structures to be built in the course of parsing and they allow extra conditions dependent on auxiliary computations. A grammar rule can have one of the following two forms. The second form is known as a push-back grammar rule, since it will complete with re-installing R:

```
P    --> Q.   % DCG rule without push back
P, R --> Q.   % DCG rule with push back
```

The term expansion augments the head by two additional parameters that are to represent the sentence position before and after the non-terminal that is defined. A grammar head with predicate identifier p/n will be turned into a normal Prolog head with predicate identifier p/n+2. The new predicate identifier can be used in system predicates such as listing/1, spy/1, etc... The outcome of this first expansion is basically:

```
phrase(P, I, O) :- phrase(Q, I, O).
phrase(P, I, O) :- phrase(Q, I, H), phrase(R, O, H).
```

The term expansion will then go to work and tackle the head of the new Prolog rule, whereas the goal expansion will tackle the body. The goal expansion will introduce unifications (=)/2 here and then to keep the expansion steadfast. One requirement is that the two queries phrase(G, I, O) and (phrase(G, I, H), H = O) should return the same results. This allows for example for a consistent definition of phrase(G, I) as an expansion to phrase(G, I, []).

Example:

```
?- [user].
factor(X) --> "(", expr(X), ")".

Yes
?- listing(factor/3).
factor(X, [40|A], B) :- expr(X, A, [41|B]).
```

We see in the example that the translation does not make use of the connection predicate ‘C’/3 for terminals. Instead terminals are directly based on the list definition of ‘C’/3 and translated into corresponding list equations. If possible these equations are merged into the head or into the body goals of the grammar rule. This gives better performance but renders the grammar mechanism not anymore customizable via ‘C’/3.

The following grammar rule predicates are provided. The (grammar) marking indicates that this operator is only understood in the context of the push back or body position of the grammar operator (-->)/2:

phrase(A, I, O):
Succeeds when the list I starts with the phrase A giving the remainder O. Can be used for parsing when I is input and for un-parsing when I is output. The predicate is multi-file and can be extended by the end-user.

phrase(A, I):
Succeeds when the list I starts with the phrase A giving the empty remainder.

P (grammar):
The grammar non-terminal P succeeds whenever the callable P extended by the current input and output succeeds.

fail (grammar):
The grammar connective fails.

A, B (grammar):
The grammar connective succeeds whenever A and B succeed. The output of A is conjoined with the input of B.

A ; B (grammar):
The grammar connective succeeds whenever A or B succeeds. The goal arguments A and B are cut transparent.

A -> B (grammar):
The grammar connective succeeds when A succeeds and then whenever B succeeds. The goal argument B is cut transparent. The output of A is conjoined with the input of B. When used inside (;)/2 it is interpreted as if-then-else.

A *-> B (grammar):
The grammar connective succeeds whenever A succeeds and then whenever B succeeds. The goal argument B is cut transparent. The output of A is conjoined with the input of B. When used inside (;)/2 it is interpreted as if-then-else.

[A_1, …, A_n] (grammar):
The grammar connective succeeds when the terminals A_1, …, A_n can be consumed.

! (grammar):
The grammar connective removes pending choice and then succeeds once.

{A} (grammar):
The grammar connective succeeds whenever the goal argument A succeeds. The goal argument A is cut transparent and not grammar translated.

\+ A (grammar):
When the goal argument A succeeds, then the grammar connective fails. Otherwise the grammar connective succeeds. The second argument is left loose.

The following grammar rule operators are provided. The (grammar) marking indicates that this operator is only understood in the context of the head position of the grammar operator (-->)/2:

P (grammar):
The grammar non-terminal P is defined with the callable P extended by the current input and output.

H --> B:
The construct defines a grammar rule with grammar head H and grammar body B.

H, P --> B:
The construct defines a push back with grammar head H, push back P and grammar body B.

Module signal [preloaded]

The predicates call_cleanup/2 and setup_call_cleanup/3 install a choice point with a clean-up goal. During a cut the current bindings are visible to the clean-up goal. During an exception the bindings are undone before invoking the clean-up goal. The latter ternary predicate differs from the ISO proposal in that it accumulates errors and in that it throws an error when the clean-up goal fails. The former binary predicate is not part of the ISO proposal.

Example:

```
?- call_cleanup((X = 1; X = 2),(write(X), write(' '))).
X = 1 ;
2 X = 2
?- call_cleanup((X = 1; X = 2),(write(X), write(' '))), !.
1 X = 1
```

Situations might demand that a secondary thread controls a primary thread. The programming interface allows raising a soft signal in a primary Prolog thread from a secondary thread. The method for this purpose is setSignal(). The effect on the primary Prolog thread will be that the signal message is thrown as an error the first possible moment a call port is reached.

The primary Prolog thread might be in a blocking operation. Therefore the method setSignal() also interrupts the primary Prolog thread. The operations of the method setSignal() are disabled as long as the mask flag is set to false. The mask flag can be read off from the corresponding Prolog flag. It can be temporarily reset by the system predicate sys_atomic/1.

The following signal handling predicates are provided:

call_cleanup(B, C):
: The predicate succeeds whenever B succeeds. Additionally the clean-up C is called when B fails or deterministically succeeds. The clean-up C is also called when a cut or an exception happens inside B or in the continuation.

sys_atomic(A):
: The predicate succeeds whenever A succeeds. The goal A is invoked with the signal mask temporarily set to off.

setup_call_cleanup(A, B, C):
: The predicate succeeds when the setup A succeeds once and whenever B succeeds. Additionally the clean-up C is called when B fails or deterministically succeeds. The clean-up C is also called when a cut or an exception happens inside B or in the continuation. The setup A and the clean-up C are called with the signal mask temporarily set to off.

The following Prolog flags for signal interception are provided:

sys_mask:
: Legal values are on and off. The flag indicates whether the interpreter currently accepts signals. Default value is on. The value can be changed.

sys_choices:
: The current number of choice points. The value cannot be changed.

sys_variables:
: The current serial number. The value cannot be changed.

Module sort [preloaded]

The predicate sys_distinct/1 will remove duplicates from a list using a hash set in the current implementation, thus relying only on equality among the elements. On the other hand the predicate sort/2 will sort a list by using a tree in the current implementation and also requires comparison among the elements.

Examples:

```
?- sys_distinct([2,1,3,1], X).
X = [2,1,3]
?- sort([2,1,3,1], X).
X = [1,2,3]
```

The predicate sys_keygroup/2 will key group a list using a hash table in the current implementation, thus relying only on equality among the keys. On the other hand the predicate keysort/2 will key sort a list by using a tree in the current implementation, thus also requiring comparison among the keys.

Examples:

```
?- hash_code(f, R).
R = 102
?- term_hash(f(X), 1, 1000, R).
R = 102
```

The hash code that is the basis for the removal and grouping predicates can be queried by the predicates hash_code/2. The hash code is recursively computed along the structure of the given term. The hash code that forms the basis of our clause indexing can be queried by the predicates term_hash/[2,4].

The following set predicates are provided:

sort(L, R): [TC2 8.4.3]
: The predicate sorts the list L and unifies the result with R.

sys_distinct(L, R):
: The predicate removes duplicates from the list L and unifies the result with R.

keysort(L, R): [TC2 8.4.4]
: The predicate key-sorts the pair list L and unifies the result with R.

sys_keygroup(L, R):
: The predicate key-groups the pair list L and unifies the result with R.

hash_code(T, H):
: The predicate succeeds when H unifies with the hash code of T. The term T need not be ground. The hash will be in the range from -2147483648 to 2147483647.

term_hash(T, H):

term_hash(T, D, R, H):
: The predicate succeeds when T is ground and when H unifies with the hash code of T. The predicate also succeeds when T is non-ground, the H argument is then simply ignored. The quinary predicate allows specifying a depth D and a modulus R. A negative depth D is interpreted as infinity.

locale_sort(L, R):

locale_sort(C, L, R):
: The predicate local sorts the list L and unifies the result with R. The ternary predicate allows specifying a locale C.

locale_keysort(L, R):
locale_keysort(C, L, R):
The predicate locale key-sorts the pair list L and unifies the result with R. The ternary predicate allows specifying a locale C.

Compatibility Matrix

t.b.d.

Table 4: Compatibility Matrix for the Standard Package

Nr	*Description*	*System*
1	Predicate indicator N//A is shortcut for N/A+2.	DCGD [3]
2	fail/0 is not a grammar construct.	DCGD
3	The call/1 grammar construct does not suspend.	DCGD
4	The phrase/3 is not automatically expanded.	DCGD
5	'|'/2 is a grammar construct.	DCGD

5.2 Basic Package [partially preloaded]

This package contains basic predicates that are partly already used by the Prolog interpreter itself. Many of the following modules provide persistent data structures. These are data structures that preserve their previous version if modified. The following modules are provided:

- **Module lists:** This module provides persistent lists.
- **Module random:** This module provides random number generators.
- **Module hyper:** This module provides hyperbolic functions.
- **Module proxy:** This module allows turning a Prolog text into a Java class.
- **Module array:** This module provides delegates for Java arrays.
- **Module utility [preloaded]:** This module provides help utilities.
- **Method score:** This module provides test predicates for Java objects.
- **Compatibility Matrix:** t.b.d.

Module lists

This module provides persistent lists. Prolog Lists are written as $[x_1,..,x_n]$ and are internally constructed by the pairing constructor [h|t] and the empty constructor []. The length of such a list is n and the i-th element is x_i. Most predicates are implemented such that they leave as few as possible choice points.

Example:

```
?- last([1,2,3], X).
X = 3
?- last([1,2,3], X, Y).
X = 3,
Y = [1,2]
```

The predicates append/3, reverse/2, member/2, select/3, last/2 and last/3 work directly with lists. The predicates length/2, nth0/3, nth0/4, nth1/3 and nth1/4 take also a length respective index into account. The predicates maplist/n and foldl/n have a closure argument to apply a predicate repeatedly to a number of arguments.

The following list predicates are provided:

append(L1, L2, L3):
The predicate succeeds whenever L3 unifies with the concatenation of L1 and L2.

reverse(L1, L2):
The predicate succeeds whenever L2 unifies with the reverse of L1. The current implementation does not terminate on redo for an input L2 and an output L1.

member(E, L):
The predicate succeeds for every member E of the list L.

select(E, L, R):
The predicate succeeds for every member E of the L with remainder list R.

last(L, E):
The predicate succeeds with E being the last element of the list L.

last(L, E, R):
The predicate succeeds with E being the last element of the list L and R being the remainder of the list.

length(L, N):
The predicate succeeds with N being the length of the list L.

nth0(I, L, E):
The predicate succeeds with E being the (I+1)-th element of the list L.

nth0(I, L, E, R):
The predicate succeeds with E being the (I+1)-th element of the list L and R being the remainder of the list.

nth1(I, L, E):
The predicate succeeds with E being the I-th element of the list L.

nth1(I, L, E, R):
The predicate succeeds with E being the I-th element of the list L and R being the remainder of the list.

maplist(C, L_1, ..., L_n):
The predicate succeeds in applying the closure C to the elements of L_1, ..., L_n. The predicate is currently defined for $1 \leq n \leq 4$.

foldl(C, L_1, ..., L_n, I, O):
The predicate succeeds in applying the closure C to the elements of L_1, ..., L_n and accumulating the result among I and O. The predicate is currently defined for $1 \leq n \leq 4$.

Module random

The evaluable functions random/0 and random/1 generate uniformly distributed members of the arithmetic domains. Each knowledge base has its own pre-allocated random number generator which can be accessed concurrently. Random number generator objects can be created with the predicates random_new/1 and random_new/2.

Examples:

```
random                  --> 0.6011883752343405
random(100)             --> 61
```

The result type of the evaluable function random/0 is always a Prolog float, which amounts to a Java double. The result type of the evaluable function random/1 reflects the type of the argument. The predicates random_next/2 and random_next/3 do the same, except that they take an additional random number generator object as a first parameter.

The predicate ticket_new/1 can be used to create a counter whch will be initialized to zero. The counter can then be incremented via the predicate counter_next/2 whereby the old value is returned. The later predicate is implemented with the help of an atomic integer.

The following random numbers evaluable functions are provided:

random(F):
The predicate succeeds for a continuous uniform random number F in the interval [0..1) from the knowledgebase random number generator.

random(M, N):
The predicate succeeds for a uniform random number N in the interval [0..M) for M>0 from the knowledgebase random number generator. The distribution is discrete when M is discrete and continuous otherwise.

The following random numbers predicates are provided:

random_new(R):
The predicate succeeds for a new random number generator R with a randomized seed.

random_new(S, R):
The predicate succeeds for a new random number generator R with seed S.

random_next(R, F):
The predicate succeeds for a continuous uniform random number F in the interval [0..1) from the random number generator R.

random_next(R, M, N):
The predicate succeeds for a uniform random number N in the interval [0..M) for M>0 from the random number generator R. The distribution is discrete when M is discrete and continuous otherwise.

counter_new(C):
The predicate succeeds for a new counter C.

counter_next(C, V):
The predicate succeeds for incrementing the counter C and unifying the old value V.

random_permutation(L, R):
: The predicate succeeds in R with a random permutation of L from the knowledgebase random number generator.

random_permutation(G, L, R):
: The predicate succeeds in R with a random permutation of L from the random number generator G.

The following random number Prolog flags are provided:

sys_random:
: Legal values are instances of the Java class java.util.Random. The flag is the random number generator for the knowledge base. The flag can be changed.

Module hyper

This module provides hyperbolic functions. The evaluable functions sinh/1, cosh/1 and tanh/1 compute the hyperbolic sinus, cosine and tangent respectively. The evaluable functions asinh/1, acosh/1 and atanh/1 compute their inverse.

The following hyper evaluable functions are provided:

sinh(X):
Returns the float representation of the hyperbolic sinus of X.
cosh(X):
Returns the float representation of the hyperbolic cosine of X.
tanh(X):
Returns the float representation of the hyperbolic tangent of X.
asinh(X):
Returns the float representation of the arcus hyperbolic sinus of X.
acosh(X):
Returns the float representation of the arcus hyperbolic cosine of X.
atanh(X):
Returns the float representation of the arcus hyperbolic tangent of X.

Module proxy

This module provides predicates to automatically turn a Prolog text into a Java class. The Java class will be a Java proxy class generated for a set of interfaces. The set of interfaces is collected from the re-exported auto loaded Java classes of the given Prolog text. The Java proxy class is generated when an instance of the Java proxy class is requested by the predicate sys_new_instance/2:

Example:

```
:- module(mycomparator, []).
:- reexport(foreign(java/util/'Comparator')).

:- public new/1.
new(X) :- sys_new_instance(mycomparator, X). % define the constructor

:- override compare/4.
:- public compare/4.
compare(_, X, Y, R) :- ... % define the method
```

The predicate and evaluable functions of the Prolog text will be used for the execution of the methods on the Java proxy instance. Only methods that belong to the set of interfaces can be invoked directly from Java on the Java proxy instances. If the set of interfaces contains the Java interface InterfaceSlots the proxy instances should be created with the predicate sys_new_instance/3 instead of the predicate sys_new_instance/2.

Example:

```
?- sys_subclass_of(java/util/'Comparator', mycomparator).
Yes

?- mycomparator:new(X), sys_instance_of(X, java/util/'Comparator').
X = 0r709d5f9e

?- mycomparator:new(X), X::compare(7,7,Y).
X = 0r43dd69,
Y = 0
```

The re-export chain of Prolog modules and auto loaded Java classes defines a module taxonomy. The module taxonomy can be tested by the predicate sys_assignable_from/2, which checks whether one module is derived from another module. Further Java Prolog proxy instances, instances directly created from within Java and Prolog callables can be tested with the predicate sys_instance_of/2.

The following proxy predicates are provided:

sys_new_instance(M, R):
: The predicate succeeds for a stateless instance R of the Java proxy class for the Prolog module M.

sys_new_instance(M, S, R):
: The predicate succeeds for a state-full instance R of size S of the Java proxy class for the Prolog module M.

sys_assignable_from(M, N):

The predicate succeeds when M is a subclass of N. N and M can be either class references or module names.

sys_instance_of(O, N):

The predicate succeeds when O is an instance of N. O can be a reference or callable. N can be either a class reference or a module name.

Module array

This module provides delegates that allow to access and modify Java arrays. Foreign predicates can be registered by one of the directives foreign_dimension/2, foreign_element/2 and foreign_update/2. Foreign evaluable functions can be registered by one of the directives foreign_length/2 or foreign_member/2.

Syntax:

```
directive --> "foreign_dimension(" indicator "," module ")"
            | "foreign_element(" indicator "," module ")"
            | "foreign_update(" indicator "," module ")"
            | "foreign_length(" indicator "," module ")"
            | "foreign_member(" indicator "," module ")".
```

Example:

```
:- foreign_dimension(new/2, int[]).
```

As a convenience we have defined the postfix operator [] and the path resolution understands this syntax to find Java array classes. When accessing or modifying array elements the delegates will see to it that the values are automatically normalized or de-normalized Prolog terms. The supported data types are the same as in the ordinary foreign function interface.

The following array predicates are provided:

foreign_dimension(I, C):
Succeeds with registering the predicate indicator I as a foreign array constructor for the array class C.

foreign_element(I, C):
Succeeds with registering the predicate indicator I as a foreign array element getter for the array class C.

foreign_update(I, C):
Succeeds with registering the predicate indicator I as a foreign array element setter for the array class C.

foreign_length(I, C):
Succeeds with registering the predicate indicator I as a foreign array length getter for the array class C.

foreign_member(I, C):
Succeeds with registering the predicate indicator I as a foreign array numeric element getter for the array class C.

Module utility [Preloaded]

This module provides help utilities. The predicate apropos/1 allows listing the advertised predicates. The lists are available independent whether a module is loaded or not. The lists are taken from the successfully activated and still registered capabilities.

Example:

```
?- apropos(time).
Indicator                Module
get_time/1               system/shell
get_time/2               system/shell
get_time_file/2          system/file
set_time_file/2          system/file
time_out/2               misc/time
time/1                   swing/stats
```

The following utility predicates are provided:

apropos(P):
The predicate succeeds in listing the public predicates on the terminal that are advertised by the loaded capabilities and that contain the given atom P in their name.

sys_apropos_table(T):
The predicate succeeds in T with the file name of a apropos table.

Module score

Foreign predicates can be automatically generated when a Java class is loaded. The module loader will proceed in that it will analyse the Java class and automatically collect all the Java methods and Java fields and generate foreign predicates for them. For Java methods that were overloaded the module loader will generate branching code.

Example:

```
?- system/automatic:generated('String':valueOf/2).
% String.class
:- package(foreign(java/lang)).
:- module('String', []).
..
:- public valueOf/2.
valueOf(A, B) :-
   sys_boolean(A), !,
   valueOf_var0(A, B).
valueOf(A, B) :-
   sys_char16(A), !,
   valueOf_var1(A, B).
..
```

The branching code uses a type check and then a cut. The type checks go beyond what the core standard defines as type checks. Since Java uses expression infered types and we have only value manifest types, we pursue the strategy that we look at the magnitude of a value. This explains for example the variety of test predicates such as sys_integer8/1 and sys_integer16_and_not_integer8/1 defined here.

The following predicates for the module score are provided:

sys_boolean(X):
The predicate succeeds when X is an atom from the set {true, false}.
sys_integer8(X):
The predicate succeeds when X is an integer and X is in the range - 2^7 to 2^7-1.
sys_char16(X):
The predicate succeeds when X is a character and X is in the range of 0 to 2^16-1.
sys_integer16(X):
The predicate succeeds when X is an integer and X is in the range - 2^15 to 2^15-1.
sys_integer32(X):
The predicate succeeds when X is an integer and X is in the range - 2^31 to 2^31-1.
sys_integer64(X):
The predicate succeeds when X is an integer and X is in the range - 2^63 to 2^63-1.
sys_integer32_or_float32(X):
The predicate succeeds when X is an 32-bit integer or a 32-bit float.
sys_integer64_or_float(X):
The predicate succeeds when X is an 64-bit integer or a float.
sys_integer16_and_not_integer8(X):
The predicate succeeds when X is an 16-bit integer but not an 8-bit integer.
sys_integer32_and_not_integer16(X):
The predicate succeeds when X is an 32-bit integer but not an 16-bit integer.
sys_integer64_and_not_integer32(X):
The predicate succeeds when X is an 64-bit integer but not an 32-bit integer.
sys_integer_and_not_integer64(X):
The predicate succeeds when X is an integer but not an 64-bit integer.
sys_atom_or_type_of(C, X):
The predicate succeeds when X is an atom or an instance of C.

sys_type_of(C, X):
The predicate succeeds when X is an instance of C.

The following predicate properties for method branching are provided:

sys_automatic:
The foreign function was automatically added by the Java class auto loader. The property can be missing. The property can be modified.

Compatibility Matrix

t.b.d.

Table 5: Compatibility Matrix for the Basic Package

Nr	*Description*	*System*
1	t.b.d.	
2	t.b.d.	
3	t.b.d.	
4	t.b.d.	

5.3 Advanced Package

This package contains advanced predicates that can be used in various Prolog applications. Many of the following modules provide persistent data structures. These are data structures that preserve their previous version if modified. The following modules are provided:

- **Module arith:** This module provides additional integer predicates.
- **Module sets:** This module provides persistent unordered sets.
- **Module ordsets:** This module provides persistent ordered sets.
- **Module sequence:** This module provides solution sequence shaping.
- **Module dict:** This module provides tagged structure access.
- **Module func:** This module functions on Prolog tagged structure.
- **Module json:** This module provides JSON object access.
- **Compatibility Matrix:** t.b.d.

Module arith

This module provides additional number predicates. The predicates between/3 and above/2 allow enumerating numbers in a given range by the unit step. For both predicates the type of the result is the type of the lower bound. The predicate above/2 doesn't have an upper bound and will return numbers forever.

Examples:

```
?- between(1, 3, X).
X = 1 ;
X = 2 ;
X = 3
?- between(1, 3, 4).
No
```

The predicates plus/3 and succ/2 allow solving primitive numeric addition equations. These predicates will not enumerate solutions, but they will work in different modes. The predicate plus/3 requires at least two instantiated arguments and the predicate succ/2 requires at least one instantiated argument.

The following arith predicates are provided:

between(L, H, X):
: The predicate succeeds in unit steps for every number X between the two numbers L and H.

above(L, X):
: The predicate succeeds in unit steps for every number X above the number L.

plus(A, B, C):
: The predicate succeeds for numbers A, B and C such that A+B =:= C. At least two arguments have to be instantiated.

succ(A, B):
: The predicate succeeds for numbers A and B such that A+1 =:= B. At least one arguments has to be instantiated.

Module sets

This module provides unordered sets. The unordered sets are represented by lists $[x_1, .., x_n]$. The lists need not to be ordered or duplicate free. But the provided operations do not necessarily preserve duplicates.

Examples:

```
?- union([2,3,4],[1,2,4,5],X).
X = [3,1,2,4,5]
?- union([1,2,4,5],[2,3,4],X).
X = [1,5,2,3,4]
```

The realization uses a membership check based on (==)/2. As a result the predicates are safe to be used with non-ground terms. On the other hand, since this comparison is not arithmetical, 1 and 1.0 are for example considered different.

The following unordered sets predicates are provided:

contains(E, S):
The predicate succeeds when the set S contains the element E.

remove(E, S, T):
The predicate succeeds when the set S contains the element E and T is the set without the element.

difference(S1, S2, S3):
The predicate succeeds when S3 unifies with the difference of S1 by S2.

intersection(S1, S2, S3):
The predicate succeeds when S3 unifies with the intersection of S1 and S2.

union(S1, S2, S3):
The predicate succeeds when S3 unifies with the union of S1 and S2.

subset(S1, S2):
The predicate succeeds when S1 is a subset of S2.

permutation(S1, S2):
The predicate succeeds when S1 is a permutation of S2.

Module ordsets

This module provides ordered sets. The ordered sets are represented by lists [x1, .., xn]. The lists must be ordered and duplicate free. If this precondition is violated the behaviour of the predicates is undefined.

Examples:

```
?- ord_union([2,3,4],[1,2,4,5],X).
X = [1,2,3,4,5]
?- ord_union([1,2,4,5],[2,3,4],X).
X = [1,2,3,4,5]
```

The realization uses a membership check based on (==)/2 and lexical ordering based on (@<)/2. As a result the predicates are safe to be used with non-ground terms. On the other hand, since this comparison is not arithmetical, 1 and 1.0 are for example considered different.

An unordered set can be converted into an ordered set by using the ISO predicate sort/2. Also there is no need for predicate permutation/2 here, since equality of ordered sets can be tested via the ISO predicate ==/2, provided the elements are sufficiently normalized.

The following ordered sets predicates are provided:

ord_contains(E, O):
The predicate succeeds when the set O contains the element E.

ord_difference(O1, O2, O3):
The predicate succeeds when O3 unifies with the difference of O1 by O2.

ord_intersection(O1, O2, O3):
The predicate succeeds when O3 unifies with the intersection of O1 and O2.

ord_union(O1, O2, O3):
The predicate succeeds when O3 unifies with the union of O1 and O2.

ord_subset(O1, O2):
The predicate succeeds when O1 is a subset of O2.

Module sequence

This module is inspired by SQL query options such as TOP. Providing such a module was recently pioneered by SWI-Prolog. Currently predicates limit/2 and offset/2 are provided. The predicates solely work tuple oriented and it is possible to cascade these predicates:

Example:

```
?- limit(5, offset(3, between(1, 10, X))).
X = 4 ;
X = 5 ;
X = 6 ;
X = 7 ;
X = 8
```

The predicates are implemented with thread local annonymous state variables and setup_call_cleanup/3, they thus work without synchronization and are interrupt safe. Predicates for ORDER BY, DISTINCT and GROUP BY are currently not yet implemented.

The following sequence predicates are provided:

limit(C, G):
The predicate succeeds whenever the goal G succeeds but limits the number of solutions to C.

offset(C, G):
The predicate succeeds whenever the goal G succeeds except for the first C solutions which are suppressed.

Module dict

This module provides tagged structure access. Tagged structures are also known as Prolog dicts. They have their own syntax as either an empty dict Term0 {} or a non-empty dict Term0 { Term1 } which are short-hands for ordinary compounds. When this module is imported, the syntax will be enabled in the importing module.

Examples:

```
?- X = point{y:2, x:1}.
X = point{x:1,y:2}
?- point{y:2, x:1} = point{x:1, y:2}.
Yes
```

The keys inside tagged structures are not restricted to any Prolog term category. All that is required is that they are ground. With the introduction of function expansion in the Jekejeke Prolog runtime library, the tagged structures will be automatically pre-sorted during consult. This assures that they are equal even if they differ in their key order.

Examples:

```
?- P = point{x:1,y:2}, get_dict(y, P, Y).
Y = 2
?- P = point{x:1,y:2}, Tag{y:Y} :< P.
Tag = point,
Y = 2
```

The set of predicates for tagged structures is modelled after the corresponding SWI-Prolog library for Prolog dicts. We have adopted most of the instantiation checks and most of the type checks. The tagged structures can be also used in connection with the dot notation. This functionality is provided through the module "func".

The following dict predicates are provided:

is_dict(X):
The predicate succeeds when X is a tagged structure.

is_dict(X, T):
The predicate succeeds when X is a tagged structure and when T unifies with the tag of the tagged structure.

dict_pairs(X, T, L):
The predicate succeeds in X with the tagged structure that has tag T and key value pairs L.

get_dict(K, S, V):
The predicate succeeds with the value V of the key K in the tagged structure S.

S :< T:
The predicate succeeds when the tags of S and T unify and when the key value pairs of the tagged structure S appear in the tagged structure T.

S >:< T:
The predicate succeeds when the tags of S and T unify and when the values for the common keys of S and T unify.

del_dict(K, S, V, T):
The predicate succeeds in T with the deletion of the key K from the tagged structure S and in V with the old value.

select_dict(S, T, R):

The predicate succeeds when the tags of S and T unify and when R unifies with a fresh tag and the removal of the key value pairs of S from the tagged structure T.

put_dict(K, S, V, T):

The predicate succeeds in T with the replacement of the new value V for the key K by in the tagged structure S.

put_dict(S, T, R):

The predicate succeeds in R with the replacement of the key value pairs of S in the tagged structure T.

Module func

This module provides functions on tagged structures and otherwise Prolog terms. Like already with the tagged structures itself, no new Prolog term category is introduced and we stay complete in the data model of the ISO core standard. Further, the translation is such that head side conditions are added to the end of a Prolog clause.

Examples:

```
?- P = point{x:1,y:2}, X = P.x, Y = P.y.
X = 1, Y = 2
?- P = [1,2], V = P.K.
V = 1, K = 0 ;
V = 2, K = 1
```

After importing the module a dot notation by the operator ('.')/2 will be available to the importing module. The operator can be used to access tagged structure fields, JSON object fields and JSON array elements anywhere inside the head or the body of a Prolog clause. The operator will be replaced through the function expansion framework.

Examples:

```
?- D = {"x":1,"y":2}.dist().
D = 2.23606797749979
?- D = point{x:1,y:2}.offset(3,4).dist().
D = 7.211102550927978
```

The operator ('.')/2 can be also used to perform a couple of field operations. Among the field operations we find get(K), which will return the field value. For a non-existing field the field operation will not throw an exception and fail instead. Further field operations include put(D) and put(K,V) to set multiple respectively a single key values at once.

Finally the operator ('.')/2 can be also used to invoke arbitrary arity functions. To disambiguate between a field access and a zero argument function invocation the module a unit notation by the operator (())/1. Arbitrary arity function definitions can be done by the further operator (:=)/2. Our translation is Pythonesk, the self is placed in the first argument.

The following expansions for functions are provided:

D.F:
: This rest expansion replaces a dot notation D.F where F is a variable or atomic by a side condition to access the field F.

D.get(K):
D.put(E):
D.put(K, V):
: This rest expansion replaces a dot notation D.get(K), D.put(E) respectively D.put(K,V) by a side condition to perform a field operation get/3, put/3 respectively put/4.

D.F():
D.F(X_1, .., X_n):
: This rest expansion replaces a dot notation D.F respectively D.F(X_1, .., X_n) by a side condition to invoke the definition of F/0 respectively F/n.

D.F() := X:
D.F(X_1, .., X_n) := X:
: This term expansion replaces a dot notation D.F() respectively D.F(X_1, .., X_n) by a clause head with result X making up a definition for F/0 respectively F/n.

Module json

This module provides JSON object access. Compared to tagged structures the JSON objects do not have a tag and their key value pairs are not automatically sorted. In addition, it is more common for JSON objects to use strings instead of atoms as keys. Further, except for the strings syntax, JSON objects do not require some new syntax.

Examples:

```
?- set_prolog_flag(double_quotes, string).
Yes
?- X = {"y":2,"x":1}.
X = {"y":2,"x":1}
```

The set of predicates for JSON objects is again modelled after the corresponding SWI-Prolog library for Prolog dicts. The predicates thus resembles our predicates for tagged structures. The JSON objects can be also used in connection with the dot notation. Again, this functionality is provided through the module "func".

The following json predicates are provided:

is_json(X):
The predicate succeeds when X is a JSON object.

get_json(K, S, V):
The predicate succeeds with the value V of the key K in the JSON object S.

select_json(S, T, R):
The predicate succeeds when R unifies with the removal of the key value pairs of S from the JSON object T.

del_json(K, S, V, T):
The predicate succeeds in T with the deletion of the key K from the JSON object S and in V with the old value.

put_json(S, T, R):
The predicate succeeds in R with the replacement of the key value pairs of S in the JSON object T.

put_json(K, S, V, T):
The predicate succeeds in T with the replacement of the new value V for the key K by in the JSON object S.

Compatibility Matrix

The following compatibility issues persist for the advanced package:

Table 6: Compatibility Matrix for the Advanced Package

Nr	*Description*	*System*
1	Does compress Prolog dictionaries during read.	SWI7
2	Does un-compress Prolog dictionaries during write.	SWI7
3	Does not use Pythonesk function translation.	SWI7
4	Does not reuse the ISO core data model for dot notation.	SWI7
5	Does not reuse the ISO core data model for unit notation.	SWI7
6	t.b.d.	t.b.d.

5.4 Experiment Package [partially preloaded]

This package contains experimental predicates that deliver some alternative or extended realizations of commonly found predicates. The following modules are provided:

- **Module maps:** The module provides unordered maps.
- **Module simp [preloaded]:** Simplification during expansion.
- **Module abstract:** An alternate variant of lambda expressions.
- **Module ordmaps:** The module provides ordered maps.
- **Module ref:** Each clause has a unique reference.
- **Module surrogate:** Surrogate keys can be created and stored.
- **Compatibility Matrix:** t.b.d.

Module maps

This module provides unordered maps. The unordered maps are represented by lists of key value pairs [k_1-v_1, .., k_n-v_n]. The lists need not be key ordered or key duplicate free. During operations for duplicates the first key wins:

Examples:

```
?- get([2-a,1-b], 1, X).
X = b
?- put([2-a,1-b], 1, c, X).
X = [2-a,1-c]
```

The realization uses a membership check based on (==)/2. As a result the predicates are safe to be used with non-ground terms. On the other hand, since this comparison is not arithmetical, 1 and 1.0 are for example considered different.

The following maps predicates are provided:

get(M, K, V):
The predicate succeeds for the value V associated with the key K in the map M.

put(M, K, V, N):
The predicate succeeds for a map N where the value V is associated with the key K and the other key values are associated as in the map M.

remove(M, K, N):
The predicate succeeds for a map N where the key K has no value and the other key values are associated as in the map M.

Module simp [preloaded]

The body conversion only caters for wrapping variables into call/1. It is possible to implement further heuristics be explicitly calling expand_term/2 respectively expand_goal/2 during asserts or calls. By this module it is arranged that simplify_term/2 respectively simplify_goal/2 are called after the expansion. The predicates are customizable by the end-user via term_simplification/2 respectively goal_simplification/2.

Example:

```
?- [user].
goal_expansion((X is E), true) :- ground(E), X is E.
test(Y) :- X is 1+2, Y is 4*X.
^D
?- listing(test/1).
test(12).
```

There are situations where the compile-time heuristics have to be undone to make them transparent. For example when listing clauses or debugging goals. The predicates rebuild_term/2 respectively rebuild_goal/2 are responsible for undoing expansions and simplifications. The rebuilding uses the same flags as the expansion and as well customizable via term_rebuilding/2 respectively goal_rebuilding/2.

The following term simplification predicates are provided:

term_simplification(C, D):
This predicate can be used to define custom term simplification rules.
simplify_term(C, D):
The system predicate succeeds if the simplification of the term C unifies with D.
goal_simplification(C, D):
This predicate can be used to define custom goal simplification rules.
simplify_goal(C, D):
The system predicate succeeds if the simplification of the goal C unifies with D.
rest_simplification(C, D):
This predicate can be used to define custom rest simplification rules.
simplify_rest(C, D):
The system predicate succeeds if the simplification of the rest C unifies with D.
term_rebuilding(C, D):
This predicate can be used to define custom term rebuilding rules.
rebuild_term(C, D):
The system predicate succeeds if the rebuild of the term C unifies with D.
goal_rebuilding(C, D):
This predicate can be used to define custom goal rebuilding rules.
rebuild_goal(C, D):
The system predicate succeeds if the rebuild of the goal C unifies with D.
rest_rebuilding(C, D):
This predicate can be used to define custom rest rebuilding rules.
rebuild_rest(C, D):
The system predicate succeeds if the rebuild of the rest C unifies with D.

Module abstract

Jekejeke Prolog also provides a simple denotation for lambda abstraction. We use the operator (\)/2 to denote abstracted terms and the operator (^)/2 to denote local terms. We can describe the lambda abstraction via the following syntax:

```
abstraction      --> binder "\" body_with_local.
body_with_local  --> local "^" body_with_local.
                   | body.
binder           --> term.
local            --> term.
```

It is possible to abstract goals and closures. The result is a new closure with an incremented closure index. The binder can be an arbitrary term, which allows pattern matching. The local can be an arbitrary term as well, which allows combining multiple local variables. The global variables of a lambda abstraction are aliased along invocations.

Examples:

```
?- map(X\Y\(H is X+1, Y is H*H),[1,2,3],R).
No                          % Aliasing prevents success.
?- map(X\Y\H^(H is X+1, Y is H*H),[1,2,3],R).
R = [4,9,16]                % Now everything is fine.
```

When a lambda abstraction is invoked the binder is replaced by the argument. In normal lambda calculus the global variables of the argument can clash with further binders in the body. In our implementation it can also happen that binders, local variables and global variables can clash. Local variables can be used to prevent clashes by renaming variables:

Examples:

```
?- K=X\Y\ =(X), call(K,Y,B,R).
K = X\Y\ =(X),
B = R                       % Clash gives wrong result.
?- K=X\Y^Y\ =(X), call(K,Y,B,R).
K = X\Y^Y\ =(X),
R = Y                       % Now everything is fine.
```

The following abstract predicates are provided:

\(X, A, Y_1, .., Y_n):
The predicate is defined for $1 \leq n \leq 7$. The goal \(X, A, Y_1, .., Y_n) succeeds whenever the goal call(A[X/Y_1], Y_2, ..., Y_n) succeeds.

Module ordmaps

This module provides ordered maps. The ordered maps are represented by lists of key value pairs [k_1-v_1, .., k_n-v_n]. The lists need to be key ordered and key duplicate free. If this precondition is violated the behaviour of the predicates is undefined:

Examples:

```
?- ord_get([1-a,2-b], 2, X).
X = b
?- ord_put([1-a,2-b], 2, c, X).
X = [1-a,2-c]
```

The realization uses a membership check based on (==)/2 and lexical ordering based on (@<)/2. As a result the predicates are safe to be used with non-ground terms. On the other hand, since this comparison is not arithmetical, 1 and 1.0 are for example considered different.

An unordered map can be converted into an ordered map by using the ISO predicate keysort/2. Also there is no need for predicate permutation/2 here, since equality of ordered maps can be tested via the ISO predicate ==/2, provided the keys and values are sufficiently normalized.

The following ordered maps predicates are provided:

ord_get(M, K, V):
The predicate succeeds for the value V associated with K in the ordered map M.

ord_put(M, K, V, N):
The predicate succeeds for an ordered map N where the value V is associated with the key K and the other key values are associated as in the ordered map M.

ord_remove(M, K, N):
The predicate succeeds for an ordered map N where the key K has no value and the other key values are associated as in the ordered map M.

Module ref

The predicates assertable_ref/2 and assumable_ref/2 allow the compilation of a clause without any thread contention. The clause can be associated and de-associated with the head predicate via the predicates recorda_ref/1, recordz_ref/1 and erase_ref/1 whereby only one thread will win. A new instance of the original clause can be retrieved again by the predicate compiled_ref/2.

The predicate clause_ref/3 can be used to find a clause in the knowledge base. This predicate respects the logical view approach form the ISO core Prolog standard. The predicate will further filter and only return those clauses that are visible from the head predicate that is used in the search. The predicate additionally returns clauses that can be used with the other predicates here.

The predicates ref_property/2, set_ref_property/2 and reset_re_property/2 allow inspecting and modifying clause properties. Clause references might not only refer to clauses, they implement a more general base class. Clause references are for example used in the Jekejeke Minlog extension to refer to attributed variable slots.

The following clause reference predicates are provided:

assertable_ref(C, R):
: The predicate compiles the term C into a new clause reference R. An undefined or unimplemented head predicate will be turned into a dynamic predicate. Otherwise the head predicate must be dynamic or thread local.

assumable_ref(C, R):
: The predicate compiles the term C into a new clause reference R. An undefined or unimplemented head predicate will be turned into a thread local predicate. Otherwise the head must be dynamic or thread local.

recorda_ref(R):
: The predicate inserts the clause referenced by R at the top. The predicate fails when the clause has already been recorded.

recordz_ref(R):
: The predicate inserts the clause referenced by R at the bottom. The predicate fails when the clause has already been recorded.

erase_ref(R):
: The predicate removes the clause referenced by R. The predicate fails when the clause has already been erased.

compiled_ref(R, C):
: The predicate returns a copy of the term C that was compiled into the clause reference R.

clause_ref(H, B, R):
: The predicate succeeds with the user clauses that match H :- B and the clause reference R of the user clause. The head predicate must be dynamic or thread local.

clause_ref(C, R):
: The predicate succeeds with the user clauses that match C and the clause reference R of the user clause. The head predicate must be dynamic or thread local.

ref_property(R, P):
: The predicate succeeds for the properties P of the reference R.

set_ref_property(R, P):
: The predicate assigns the property P to the reference R.

reset_ref_property(R, P):
: The predicate de-assigns the property P from the reference R.

Module surrogate

The predicate sys_new_surrogate/1 helps realizing relational data models. It will return a new unique reference object. The reference cannot be used in lexical orderings. Nevertheless the reference object can be asserted inside facts and rules.

As an example application we deliver a programming interface to local state variables. The protocol includes the predicates new_local/2, get_local/2, set_local/2, free_local/2 and current_local/2. Since the implementation uses a thread local fact it is synchronization free.

The following surrogate predicates are provided:

new_local(N, V):
: Creates a new local variable N with value V.

get_local(N, V):
: Succeeds when V unifies with the value of the local variable N.

set_local(N, V):
: Sets the value of the local variable N to V.

free_local(N):
: Removes the local variable N.

current_local(N):
: Succeeds for every local variable N.

sys_new_surrogate(K):
: Creates a new surrogate key K.

Compatibility Matrix

t.b.d.

Table 7: Compatibility Matrix for the Standard Package

Nr	*Description*	*System*
1	t.b.d.	
2	t.b.d.	
3	t.b.d.	
4	t.b.d.	

5.5 Stream Package [partially preloaded]

This theory is concerned with input/output and the selection of streams. The modules of the package are nameless and preloaded. They don't needed to be loaded and cannot be loaded by the end-user. We find the following topics:

- **Module char [preloaded]:** Characters can be input / output from / to text streams.
- **Module byte [preloaded]:** Bytes can be input / output from / to binary streams.
- **Module term [preloaded]:** Terms can be input / output from / to text streams.
- **Module stream [preloaded]:** Predicates to select and control streams.
- **Module console:** Predicates to address the console.
- **Module xml:** This module provides generation and parsing of XML texts.
- **Compatibility Matrix:** ISO/DEC10 compatibility issues of the stream theory.

Module char [Preloaded]

Characters can be written to text streams or read from text streams. Text streams can be obtained by the to open/3 and open/4 system predicates documented in the stream control section. The standard output and/or the standard input might also point to text streams, but this is not guaranteed. Text streams are automatically flushed when the new line primitive is invoked. Additionally the text stream of the console window is automatically flushed when more than 1024 characters have been written.

The following character input/output predicates are provided:

nl: [ISO 8.12.3]
nl(T): [ISO 8.12.3]
: The predicate without arguments writes the system end of line sequence to the standard output and flushes it. The unary predicate takes an additional text stream sink as argument.

put_char(C): [ISO 8.12.3]
put_char(T, C): [ISO 8.12.3]
: The unary predicate writes the character C to the standard output. The binary predicate takes an additional text stream sink as argument.

put_code(C): [ISO 8.12.3]
put_code(T, C): [ISO 8.12.3]
: The unary predicate writes the code C to the standard output. The binary predicate takes an additional text stream sink as argument.

peek_char(C): [ISO 8.12.2]
peek_char(T, C): [ISO 8.12.2]
: The unary predicate reads a character from the standard input and puts it back. The predicate succeeds when C unifies with the peeked character or the atom end_of_file when the end of the stream has been reached. The binary predicate takes an additional text stream source as argument.

peek_code(C): [ISO 8.12.2]
peek_code(T, C): [ISO 8.12.2]
: The predicate reads a code from the standard input and puts it back. The predicate succeeds when C unifies with the read code or the integer -1 when the end of the stream has been reached. The binary predicate takes an additional text stream source as argument.

get_char(C): [ISO 8.12.1]
get_char(T, C): [ISO 8.12.1]
: The predicate reads a character from the standard input. The predicate succeeds when C unifies with the read character or the atom end_of_file when the end of the stream has been reached. The binary predicate takes an additional text stream source as argument.

get_code(C): [ISO 8.12.1]
get_code(T, C): [ISO 8.12.1]
: The predicate reads a code from the standard input. The predicate succeeds when C unifies with the read code or the integer -1 when the end of the stream has been reached. The binary predicate takes an additional text stream source as argument.

Module byte [Preloaded]

Bytes can be written to binary streams or read from binary streams. Binary streams can be obtained by the to open/3 and open/4 system predicates documented in the stream control section. The standard output and/or the standard input might also point to binary streams, but this is not guaranteed.

Text streams and binary streams share the notion of flushing and end of stream. Therefore the system predicates flush_output/[0,1] and at_end_of_stream/[0,1] apply to both text streams and binary streams.

The following byte input/output predicates are provided:

put_byte(B): [ISO 8.13.3]
put_byte(H, B): [ISO 8.13.3]
: The unary predicate writes the byte B to the standard output. The binary predicate takes an additional binary stream sink as argument.

get_byte(B): [ISO 8.13.1]
get_byte(H, B): [ISO 8.13.1]
: The predicate reads a byte from the standard input. The predicate succeeds when B unifies with the read byte or the integer -1 when the end of the stream has been reached. The binary predicate takes an additional binary stream source as argument.

peek_byte(B): [ISO 8.13.2]
peek_byte(H, B): [ISO 8.13.2]
: The predicate reads a byte from the standard input and puts it back. The predicate succeeds when B unifies with the read byte or the integer -1 when the end of the stream has been reached. The binary predicate takes an additional binary stream source as argument.

flush_output: [ISO 8.11.7]
flush_output(S): [ISO 8.11.7]
: The predicate without arguments flushes the standard output. The unary predicate takes an additional text or binary stream sink as argument.

at_end_of_stream: [ISO 8.11.8]
at_end_of_stream(S): [ISO 8.11.8]
: The predicate without arguments checks whether we are at the end of the standard input. The unary predicate takes an additional text or binary stream sink as argument.

read_block(L, B):
read_block(I, L, B):
: The predicate succeeds in a block B in reading maximally L bytes from I.

write_block(B):
write_block(O, B):
: The predicate succeeds in writing the byte block B to O.

Module term [Preloaded]

Terms can be written to text streams or read from text streams depending on the current operator definitions. It is possible to switch off operator usage by the write option ignore_ops/1. Further during writing variables and atoms are put into quotes when necessary. It is possible to switch on quoting by the write option quoted/1.

Finally terms of the form '$VAR'(<number>) are usually recognized and written out as <var>. It is possible to switch on the variable numbering by the write option numbervars/1. The Prolog system also supports '$STR'(<atom>) terms to represent strings and can read or write them as quoted strings.

Table 8: Predefined Write Predicates

Predicate	*numbervars*	*quoted*	*ignore_ops*
write	Yes	No	No
writeq	Yes	Yes	No
write_canonical	No	Yes	Yes

The spacing is determined by the context type option. The context type '?' minimizes the spacing. The other context types use spacing for the current compound and they also determine which meta-declaration should be looked up in case of a closure. Here are some examples whereby we assume a meta_predicate declaration solve(0):

Table 9: Context Dependent Spacing

Context	*Example 1*
?	solve((_A,_B)):-solve(_A),solve(_B)
0 or -1	solve((_A, _B)) :- solve(_A), solve(_B)

If the format option is newline then the spacing is enhanced by new lines and further spaces so that the output matches the Prolog coding guidelines as published in [9]. Further the priority option determines whether parentheses are needed around an operator expressions depending on the level of the operator.

When double quotes or back quotes are set to 'variable' and quote is true, then variable names are automatically set into the corresponding quotes when necessary. If neither double quotes nor back quotes are set to 'variable', then the predicates write_term/[2,3] throw an error if a variable name would need quotes.

The following term input/output predicates are provided:

write(E): [ISO 8.14.2]
write(T, E): [ISO 8.14.2]
: The unary predicate writes the term E to the standard output whereby numbering variables. The binary predicate takes an additional text stream sink as argument.

writeq(E): [ISO 8.14.2]
writeq(T, E): [ISO 8.14.2]
: The predicate writes the term E to the standard output whereby quoting atoms and variables if necessary. The binary predicate takes an additional text stream sink as argument.

write_canonical(E): [ISO 8.14.2]
write_canonical(T, E): [ISO 8.14.2]
: The predicate writes the term E to the standard output whereby quoting atoms and variables if necessary and ignoring operator declarations. The binary predicate takes an additional text stream sink as argument.

write_term(E, O): [ISO 8.14.2]
write_term(T, E, O): [ISO 8.14.2]

The predicate writes the term E to the standard output taking into account the write options O. The ternary predicate takes an additional text stream sink as argument. The following write options are available:

quoted(B): [ISO]	If B then functors are quoted if necessary.
numbervars(B): [ISO]	If B then compounds '$VAR'(I) are written as variables.
ignore_ops(B): [ISO]	If B then functors are not interpreted as operators.
variable_names(N):	N are the variable names.
context(C):	C is the meta-argument specifier.
format(F):	F is the write format.
priority(P):	P is the write priority.
double_quotes(U):	U is the parsing of double quotes.
back_quotes(U):	U is the parsing of back quotes.
single_quotes(U):	U is the parsing of single quotes.
annotation(A):	A is the annotation mode.
source(S):	S is the source the term belongs to.
part(P):	P is the part that should be written.

Legal context type values are '?' or an integer. The default context type value is '?'. Legal format values are 'false', 'newline', 'navigation' and 'true'. The default format value is 'false'. Legal values for the operator priority are integer values in the range 0 to 1200. The default operator priority is 1200.

Legal annotation values are 'false', 'makedot', 'filler' and 'true'. The default annotation value is 'false'. Legal single quotes, double quotes and back quotes values are 'error', 'chars' [ISO], 'codes' [ISO], 'variable', 'atom' [ISO] and 'string'. The default quote values are taken from the corresponding Prolog flags. Legal part values are 'false', 'comment', 'statement' and 'true'. The default part value is 'true'.

read(E): [ISO 8.14.1]
read(T, E): [ISO 8.14.1]

The unary predicate reads a sentence from the standard input and parses it into a Prolog term. The sentence consists of the tokens up to the first terminating period (“.”). When the sentence only contains filler without a terminating period (“.”) then the predicate succeeds when E unifies with the end_of_file atom. Otherwise the predicate succeeds when E unifies with the parsed term. The binary predicate takes an additional text stream source as argument.

read_term(E, O): [ISO 8.14.1]
read_term(T, E, O): [ISO 8.14.1]

The predicate reads the term E from the standard input taking into account the read options O. The ternary predicate takes an additional text stream source as argument. The following read options are available:

variables(L): [ISO]	L are the variables.
variable_names(N): [ISO]	N are the variable names.
singletons(S): [ISO]	S are the singleton names.
priority(P):	P is the read priority.
terminator(T):	T is the terminator.
double_quotes(U):	U is the parsing of double quotes.
back_quotes(U):	U is the parsing of back quotes.
single_quotes(U):	U is the parsing of single quotes.
annotation(A):	A is the annotation mode.
source(S):	S is the source the term belongs to.
line_no(N):	N is line number where the term starts.

Legal terminator values are 'period' [ISO], 'end_of_file' and 'none'. The default value is 'period'. Legal single quotes, double quotes and back quotes values are ‘error’, ‘chars’ [ISO], ‘codes’ [ISO], ‘variable’, 'atom' [ISO] and ‘string'. The default values are taken from the corresponding Prolog flags. Legal annotation values are ‘false’, ‘makedot’, ‘filler’ and ‘true’. The default annotation is ‘false’.

The following Prolog flags for term input/output are provided:

char_conversion: [ISO 7.11.2.1]
Legal values are ‘on’ [ISO] and ‘off’ [ISO]. The flag indicates whether unquoted tokens are character converted. The default value is ‘off’. Currently the value cannot be changed.

double_quotes: [ISO 7.11.2.5]
Legal values are ‘error’, ‘chars’ [ISO], ‘codes’ [ISO], ‘variable’, 'atom' [ISO] and ‘string'. The flag indicates how double quoted strings are parsed. The default value is ‘codes’.

back_quotes:
Legal values are ‘error’, ‘chars’, ‘codes’, ‘variable’, 'atom' and ‘string'. The flag indicates how back quoted strings are parsed. The default value is ‘error’.

single_quotes:
Legal values are ‘error’, ‘chars’, ‘codes’, ‘variable’, 'atom' and ‘string'. The flag indicates how back quoted strings are parsed. The default value is ‘atom’.

Module stream [Preloaded]

Streams can have a couple of properties. The if-modified-since date property can be supplied by the open options. It causes the open primitive to throw a not-modified exception when the source has not been modified since the given date. The last-modified and expiration date properties can be retrieved via the open options.

Text streams have further the character set encoding property. For remote sources with a mime type this property defaults to the character set property of the mime type. Otherwise the property defaults to UTF-8. Incoming CR LF sequences or CR characters are automatically compressed respectively translated to LF characters.

If a text stream belongs to the file schema and is opened for read with the byte order mark detection on, this detection will try to determine the default encoding. The detection can currently detect UTF-8, UTF-16LE and UTF-16BE. If a mark is detected the initial read position is placed after the mark.

The following stream control predicates are provided:

current_input(S): [ISO 8.11.1]
The predicate succeeds when S unifies with the standard input stream.
current_output(S): [ISO 8.11.2]
The predicate succeeds when S unifies with the standard output stream.
current_error(S):
The predicate succeeds when S unifies with the standard error stream.
set_input(S): [ISO 8.11.3]
The predicate sets the standard input stream to the source S.
set_output(S): [ISO 8.11.4]
The predicate sets the standard output stream to the sink S.
set_error(S):
The predicate sets the standard error stream to the sink S.
open(P, M, S): [ISO 8.11.5.4]
open(P, M, S, O): [ISO 8.11.5.4]
The ternary predicate succeeds when S unifies with the new stream associated with the path or socket P and the access mode M (read, write or append). The quaternary predicate additionally recognizes the following open options:

type(T): [ISO]	T is the type (text or binary), default value is text.
alias(A): [ISO]	A is the alias.
bom(B):	B is the byte order mark detection and generation flag.
use_caches(B):	B is the use caches flag for the connection.
encoding(C):	C is the character set encoding for the text stream.
buffer(S):	S is the buffer size, or 0 if no buffer is required.
if_modified_since(D):	D is the if-modified-since date for the connection.
If_none_match(V):	V is the if-none-match tag for the connection
reposition(B): [ISO]	B is the reposition (false or true), default value is false.
newline(S):	S is the newline character sequence to use in writing.

close(S): [ISO 8.11.6]
close(S, O): [ISO 8.11.6]
The unary predicate closes the closeable S. The binary predicate additionally recognizes the following close options.

force(F): [ISO]	F is the force flag, default is false.

stream_property(S, P): [ISO 8.11.8]

The predicate succeeds with all the properties of the stream S that unify with P. The following stream properties are supported:

mode(M): [ISO]	M is the mode (read, write or append).
type(T): [ISO]	T is the type (text or binary).
bom(B):	B is the byte order mark.
encoding(C):	C is the character set encoding of the text stream.
buffer(S):	S is the buffer size, or 0 if no buffer is present.
last_modified(D):	D is the last-modified date of the source.
version_tag(V):	V is the version tag of the source.
expiration(D):	D is the expiration date of the source.
line_no(N):	N is the line number of the text stream source.
reposition(B): [ISO]	B is the reposition (false or true)
position(P): [ISO]	P is the actual file position.
length(L):	L is the actual file length.
file_name(F): [ISO]	F is the absolute path of the stream.
input: [ISO]	The stream is an input stream.
output: [ISO]	The stream is an output stream.

An undefined encoding/1 property is returned as a zero length atom. An undefined buffer property is returned as the value 0. The last_modified/1 and the expiration/1 properties are given in milliseconds since January 1, 1970 GMT. An undefined last_modified/1 or expiration/1 property is returned as the value 0.

The version tag is an atom that starts and ends with double quotes (“). An undefined version tag is returned as a zero length atom. The line_no/1 of a source starts with 1. The position/1 and length/1 properties are measured in bytes, starting with 0 and are available when reposition(true) holds. An undefined file_name/1 is returned as a zero length atom.

set_stream_position(S, P): [ISO 8.11.9]

The predicate sets the file position of the stream S to P.

set_stream_length(S, L):

The predicate sets the file length of the stream S to L.

open_resource(P, S):

open_resource(P, S, O):

The predicate succeeds when S unifies with the new resource stream associated with the path P. The ternary predicate additionally recognizes the following open options.

The following Prolog flags for stream control are provided:

sys_cur_input:

The legal value is an input stream term. The stream is used for input by the input predicates. Default value is the current input stream. The value can be changed.

sys_cur_output:

The legal value is an output stream term. The stream is used for output by the output predicates. Default value is the current output stream. The value can be changed.

sys_cur_error:

The legal value is an output stream term. The stream is used for errors by the output predicates. Default value is the current error stream. The value can be changed.

Module console

This module supports access to the console. The first part consists of read utilities. Among the read utilities there are currently the predicates read_line/[1,2] and read_line_max/[2,3]. The former does an unbounded read the later does a bounded read.

The second part consist Quintus Prolog inspired formatted output predicates. Among the formatted output there are currently the predicates format/[2,3], print_message/[1,2], print_error/[1,2] and print_stack_trace/[1,2]. The formatting is based on the Java Formatter class.

Example:

```
?- format('res=%20d',[123123123]), nl.
res=           123123123
Yes
?- format('res=%25.4f',[123123123.123]), nl.
res=           123123123.1230
Yes
```

The third part consist Edinburgh Prolog inspired terminal input/output. Among the terminal input/output there are currently the predicates ttynl/0, ttywrite/1, ttywrite_term/2, ttyflush_output/0, ttyread_line/1 and ttyread_line_max/2.

The following console predicates are provided:

read_line(C):
read_line(T, C):
: The predicate succeeds in C in reading a line. The predicate fails upon an empty line and an end of file. The binary predicate allows specifying a text stream T.

read_line_max(L, C):
read_line_max(T, L, C):
: The predicate succeeds in C in reading a line with maximally L characters. The predicate fails upon an empty line and an end of file. The ternary predicate allows specifying a text stream T.

read_punch(C):
read_punch(T, C):
: The predicate succeeds in C in reading a punch. The predicate fails upon end of file. The punch must end in CR LF, otherwise an exception is thrown. The binary predicate allows specifying a binary stream T.

read_punch_max(L, C):
read_punch_max(T, L, C):
: The predicate succeeds in C in reading a punch with maximally L bytes. The predicate fails upon end of file. If less than L bytes different from CR are read, the punch must end in CR LF otherwise an exception is thrown. The ternary predicate allows specifying a binary stream T.

format(F, A):
format(T, F, A):
: The predicate formats the list of arguments A according to the format F using the current locale and writes it to the current output. The ternary predicate allows specifying a text stream T.

print_message(M):
print_message(T, M):
: The predicate formats the message term M according to the error properties of the knowledge base and the current locale, and writes it to the current output. The binary predicate allows specifying a text stream T.

print_error(E):
print_error(T, E):
The predicate formats the error term E without its context according to the error properties of the knowledge base and the current locale, and writes it to the current error. The binary predicate allows specifying a text stream T.
print_stack_trace(E):
print_stack_trace(T, E):
The predicate formats the error term E with its context according to the error properties of the knowledge base and the current locale, and writes it to the current error. The binary predicate allows specifying a text stream T.
ttynl:
The predicate writes the system end of line sequence to the terminal and flushes it.
ttywrite(E):
The predicate writes the term E to the terminal whereby numbering variables.
ttywrite_term(E, O):
The predicate writes the term E to the terminal taking into account the write options O.
ttyflush_output:
The predicate flushes the terminal.
ttyread_line(C):
The predicate succeeds in C in reading a line from the terminal. The predicate fails upon an empty line and an end of file.
ttyread_line_max(L, C):
The predicate succeeds in C in reading a line with maximally L characters from the terminal. The predicate fails upon an empty line and an end of file.

Module xml

This module provides a couple of simple utilities to deal with the generation and parsing of XML texts. The predicate text_escape/2 can be used to escape and un-escape texts. The predicate text_escape/2 will escape the characters '"<>&' and the character 0xA0. It is suitable for attribute values in double quotes and for texts.

Examples:

```
?- text_escape('<abc>', X).
X = '&lt;abc&gt;'
?- text_escape(X, '&lt;abc&gt;').
X = '<abc>'
```

The predicate base64_block/2 can be used to base64 encode and decode a byte block. This code allows representing 8-bit bytes as ASCII characters. When generating base64 code the predicate will produce 10 blocks of 4 characters effectively encoding 30 bytes. While decoding the terminating characters = indicate the number of fill bytes.

The following xml predicates are provided:

text_escape(T, E):
: If T is a variable then the predicate succeeds when T unifies with the text un-escape of E. Otherwise the predicate succeeds when E unifies with the text escape of T.

hex_block(T, E):
: If T is a variable then the predicate succeeds when T unifies with the hex encode of E. Otherwise the predicate succeeds when E unifies with the hex decode of T.

base64_block(T, E):
: If T is a variable then the predicate succeeds when T unifies with the based64 encode of E. Otherwise the predicate succeeds when E unifies with the base64 decode of T.

Compatibility Matrix

The following compatibility issues persist for the stream theory:

Table 10: Compatibility Matrix for the Stream Theory

Nr	*Description*	*System*
1	Has see/tell style stream handling	DEC10
2	Has rename/delete predicate.	DEC10
3	Has nofileerrors/fileerrors directive.	DEC10
4	The predicate display bypasses the standard output.	DEC10
5	Has predicate print which can be customized by portray.	DEC10
6	Has standard aliases user_input and user_output.	ISO
7	Close of current input/output will default them to standard.	ISO
8	Does not have sys_print_exception/1.	ISO
9	Does not have variable_names/1 option in write_term.	ISO
10	Does not have back_quotes flag.	ISO
11	The char_conversion flag is changeable.	ISO

5.6 System Package

This package contains predicates that allow the access to various operating system elements. The following modules are provided:

- **Module file:** This module provides some basic file operations.
- **Module uri:** This module provides generation and parsing of URIs.
- **Module group:** The Prolog system supports multiple thread groups.
- **Module thread:** The Prolog system supports multiple concurrent interpreters.
- **Module shell:** This module provides access to the shell that started the interpreter.
- **Module locale:** This module provides the lookup and retrieval of properties.
- **Module domain:** This module provides access to domain names.
- **Compatibility Matrix:** t.b.d.

Module file

This module provides some basic file operations. The implementation is backed by some Java code that delivers some primitives to access files. The Prolog text invokes absolute_file_name/[2,3] before accessing these primitives. The result is that file name resolution and encoding is applied for these primitives and that they accept URIs.

Example:

```
?- directory_file('file:/C:/Program+Files/Java/', X).
X = 'jdk1.6.0_45' ;
X = 'jdk1.7.0_67' ;
X = 'jdk1.8.0_20' ;
Etc..
```

Currently only URIs of protocol “file:” are supported. But future implementations might support further protocols such as “jar:” or service based protocols such as Google Drive, DropBox or others. The realization might depend on the platform, so that the Swing variant and Android might support different protocols in the future.

The following file predicates are provided:

make_name(B, E, N):
: If B or E is a variable then the predicate succeeds when B and E unify with the base name and the extension respectively of the file name N. Otherwise the predicates succeeds when N unifies with the constructed name.

make_path(D, N, P):
: If D or N is a variable then the predicate succeeds when D and N unify with the directory and the name respectively of the path P. Otherwise the predicates succeeds when P unifies with the constructed path.

create_file(F):
: Succeeds when the file F could be created.

delete_file(F):
: Succeeds when the file or directory F could be deleted.

rename_file(F, G):
: Succeeds when the file or directory F could be renamed to the file or directory G.

exists_file(F):
: Succeeds when the file F exists and when it isn't a directory.

exists_directory(F):
: Succeeds when the file F exists and when it is a directory.

make_directory(F):
: The predicate succeeds when the directory F could be created.

directory_file(F, N):
: Succeeds whenever N unifies with an entry if the directory F.

get_time_file(F, T):

Succeeds when T unifies with the last modified date of the file or directory F. T is measured in milliseconds since the epoch.

set_time_file(F, T):

Succeeds when the last modified date of the file or directory F could be set to T. T is measured in milliseconds since the epoch but might be rounded by the system.

is_relative_path(P):

The predicate succeeds when the path P is a relative path.

follow_path(B, R, A):

If R is a variable then the predicate succeeds when R unifies with the relative or absolute path that leads from B to A. Otherwise the predicate succeeds when A unifies with the path that results from B by following R.

canonical_path(P, C):

The predicate succeeds when C unifies with the canonical path of P.

Module uri

This module provides a couple of simple utilities to deal with the generation and parsing of uniform resource identifiers (URIs). The predicates make_query/4, make_spec/4 and make_uri/4 allow constructing and deconstructing queries, specs and URIs. The predicates work bidirectional without loss of data.

Example:

```
?- make_query(X, Y, Z, 'a%3Db=c%26d&e').
X = 'a=b',
Y = 'c&d',
Z = e
```

The predicates make_query/4, make_spec/4 and make_uri/4 do a minimal encoding. For the parameter name and the parameter value the characters '#%=&\' will be encoded. For the hash the characters '%\' will be encoded. For the spec the characters '?#%\' will be encoded. If used in the other direction the predicates will perform decoding of the corresponding components.

An URI is relative when it neither contains a scheme nor an authority, and if the path is relative. The predicate is_relative_uri/1 checks whether an URI is relative. The predicate follow_uri/3 can be used to resolve and relativize URIs. Contrary to the Java URL class, it is agnostic to the scheme of the URIs and will allow any scheme. In these predicates the path component is handled by the corresponding routines from system/file.

Example:

```
?- follow_uri('file:/foo/bar/baz?jack#jill',
   X, 'file:/foo/tip/tap?fix#fox').
X = '../tip/tap?fix#fox'
```

The predicate canonical_uri/2 can be used to canonize URIs. For the file protocol the path component is handled by the corresponding routine from the module file. For other protocols the routine uses puny code from the module domain and accesses the server for redirects.

The predicate uri_encode/2 can be used to encode and decode URIs. The predicate uri_encode/2 will percent encode characters above 0x7F. As a result the URI will only contain ASCII. If used in the other direction the predicate will first decode and then minimal encode again.

The following URI predicates are provided:

make_query(N, V, R, Q):
: If N, V or R is a variable then the predicate succeeds when N, V and R unify with the first parameter name, the first parameter value and the rest query of the query Q. Otherwise the predicates succeeds when Q unifies with the constructed query.

make_spec(E, A, P, S):
: If E, A or P is a variable then the predicate succeeds when E, A and P unify with the scheme, authority and path respectively of the spec S. Otherwise the predicate succeeds when S unifies with the constructed spec.

make_uri(S, Q, H, U):
: If S, Q or H is a variable then the predicate succeeds when S, Q and H unify with the spec, query and hash respectively of the URI U. Otherwise the predicate succeeds when U unifies with the constructed URI.

is_relative_uri(U):

The predicate succeeds when the URI U is a relative URI.

follow_uri(B, R, A):

If R is a variable then the predicate succeeds when R unifies with the relative or absolute URI that leads from B to A. Otherwise the predicate succeeds when A unifies with the URI that results from B by following R.

canonical_uri(U, C):

The predicate succeeds when C unifies with canonical URI of U.

uri_encode(T, E):

If T is a variable then the predicate succeeds when T unifies with the URI decode of E. Otherwise the predicate succeeds when E unifies with the URI encode of T.

Module group

A Prolog thread group is simply a Java thread group. A Prolog thread group might contain Prolog threads and otherwise threads. The predicate group_new/1 creates a new thread group. The predicate group_thread/2 allows retrieving the oldest member. The predicate current_group_flag /3 allows inspecting thread group properties.

Example:

```
?- threads.
Thread       State           Group
Thread-2     WAITING         main
Thread-3     RUNNABLE        Group-1
Thread-4     WAITING         Group-1
Yes
```

The predicate current_thread/1 succeeds for the Prolog threads currently known to the base knowledge base. The predicate threads/0 lists the same threads on the standard output. The Prolog threads are shown with their state and their group. Currently the predicates also list threads across different sub knowledge bases.

The following group predicates are provided:

group_new(G):
The predicate succeeds for a new thread group G.

thread_new(G, C, T):
The predicate succeeds for a new thread T on the copy of the goal C inside the thread group G.

group_thread(G, T):
The predicate succeeds in T with the oldest thread of the thread group G if there is any. Otherwise the predicate fails.

current_thread(G, T):
The predicate succeeds in T with the threads of the thread group G.

current_group(G, H):
The predicate succeeds in H with the groups of the thread group G.

current_group_flag(G, K, V):
The predicate succeeds for the values V of the keys K concerning the group G. The following keys are returned by the predicate.

sys_group_name:	The name of the group.
sys_group_group:	The group of the group.

current_thread(T):
The predicate succeeds in T with the managed threads.

threads:
The predicate lists the managed threads.

Module thread

A Prolog thread is simply a Java thread that executes a Prolog call-in. The call-in can be created by the predicate thread_new/2 and the goal will be copied. The thread can then be started by the predicate thread_start/1. A thread need not be explicitly destroyed, it will automatically be reclaimed by the Java GC when not anymore used.

Examples:

```
?- thread_new((between(0,10,X),write(X),write(' '),fail;
     nl), I), thread_start(I).
I = 0r5ab10801
0 1 2 3 4 5 6 7 8 9 10
```

A new thread will share the knowledgebase and the display input/output of the creating thread. On the other hand a new thread will have its own thread local predicates. A thread can be aborted by the predicate thread_abort/2 and thread_down/[2,3]. A thread can be killed by the predicate thread_kill/1. The later predicate should only be used in emergency situation, since the receiving Prolog call-in will not be able to properly clean-up.

The predicates thread_join/1 and thread_combine/[1,2] allow waiting for the termination of a thread. The predicates will block, fail or timeout when the thread is alive. Every thread can be joined and joining does not retrieve an exit code and/or an exit Prolog term. The predicate current_thread_flag/3 allows inspecting thread properties.

The following thread predicates are provided:

thread_sleep(M):
The predicate suspends the current thread for M milliseconds.

thread_current(T):
The predicate succeeds for the current thread T.

thread_new(C, T):
The predicate succeeds for a new thread T on the copy of the goal C.

thread_start(T):
The predicate succeeds for starting the thread T.

thread_abort(T, M):
The predicate succeeds for signalling the error message M to the thread T.

thread_down(T, M):
The predicate succeeds for signalling the error message M to the thread T. Otherwise the predicate fails.

thread_down(T, M, W):
The predicate succeeds for signalling the error message M to the thread T in the timeout W. Otherwise the predicate fails.

thread_kill(T):
The predicate succeeds for killing the thread T.

thread_join(T):
The predicate succeeds when the thread T has terminated.

thread_combine(T):
The predicate succeeds when the thread T has terminated. Otherwise the predicate fails.

thread_combine(T, W):
The predicate succeeds when the thread T has terminated in the timeout W. Otherwise the predicate fails.

current_thread_flag(T, F, V):
The predicate succeeds for the value V of the flag F for the thread T.
set_thread_flag(T, F, V):
The predicate sets the flag F to the value V for the thread T.

The following thread thread flags are provided:

sys_thread_name:
The name of the thread. The flag cannot be changed.
sys_thread_status:
The status of the thread. The flag cannot be changed.
sys_thread_group:
The group of the thread. The flag cannot be changed.

The following thread Prolog flags are provided:

sys_cpu_count:
The flag returns the number of logical cores of the current process. The flag cannot be changed.
sys_runtime_version:
The flag returns the version of the Java language runtime of the current process. The flag cannot be changed.

Module shell

The module provides access to the shell that started the interpreter. The predicate getenv/2 allows accessing and enumerating shell environment variables. The access is always case insensitive.

Examples:

```
?- get_time(D), format('%tc\n', [D]).
Mon Aug 22 17:07:24 CEST 2016

?- statistics(wall, T), get_time(T, D), format('%tc\n', [D]).
Mon Aug 22 17:07:39 CEST 2016
```

The predicates get_time/1 and get_time/2 can be used to retrieve a time object that is suitable for format/2 and friends. The predicate get_time/2 has an integer time stamp constructor parameter.

The following shell predicates are provided:

getenv(N, V):
The predicate succeeds for the value V of the environment variable named N.

get_time(S):
The predicate succeeds with a current time object S. The time object is suitable for format/2 and friends.

get_time(T, S):
The predicate succeeds with a time object S for the time T in milliseconds since January 1, 1970, 00:00:00 GMT. The time object is suitable for format/2 and friends.

Module locale

A properties bundle consists of multiple properties files that vary in the file name by an injection of a locale code before the file extension. Our convention is that each bundle must contain a properties file without injection. This file acts as a root for the bundle and as a fall-back for locales that are not found:

Examples:

```
code.properties:    The root and fall-back.
code_de.properties: The German member of the bundle.
```

The predicates sys_get_lang/2 and sys_get_lang/3 allow retrieving a locale properties file of a resource bundle. These predicates make use of the predicate absolute_resource_name/2 to resolve the root so that the same base name without an extension can be used for both Prolog text and resource bundles. The resource bundle itself has to be loaded in advanced via the predicate sys_load_resoruce/1.

Examples:

```
:- sys_load_resource('code').
test(Y) :- sys_get_lang('code',X), get_property(X,'foo',Y).
?- test(X).
X = bar
```

The retrieval of a locale properties file is relatively fast, since we cache locale properties files on a per resource bundle basis. But it still not yet as fast and flexible as predicate invocation, since we have not yet implemented a call-site cache and an auto loader for resource bundles. The predicates get_property/3 and get_property/4 allow retrieving a property value from a locale properties file.

The predicate atom_format/[3,4] allows formatting a list of arguments based on a template and a locale. The predicates message_make/[3,4] and error_make/[3,4] allow formatting a term based on properties file and a locale. The predicates get_error_properties/[1,2] and get_description_properties/[2,3] allow retrieving knowledgebase respective capability defined properties files.

The following locale predicates are provided:

sys_get_lang(S, P):
sys_get_lang(S, L, P):
: The predicate unifies P with the properties from the bundle S for the current default locale. The ternary version of the predicate allows specifying the locale L. The resource bundle S has to be loaded in advance via sys_load_resource/1.

get_property(P, K, V):
get_property(P, K, D, V):
: The predicate unifies V with the value for the key K from the properties P. The quaternary version of the predicate allows specifying a default value D.

atom_format(F, A, S):
atom_format(L, F, A, S):
: The predicate formats the arguments A from the format F and unifies the result with S. The quaternary predicate allows specifying a locale L.

message_make(P, M, S):
message_make(L, P, M, S):
: The predicate formats the message term M from the properties P and unifies the result with S. The quaternary predicate allows specifying a locale L.

error_make(P, E, S):
error_make(L, P, E, S):
The predicate formats the error term E without its context from the properties P and unifies the result with S. The quaternary predicate allows specifying a locale L.
get_error_properties(P):
get_error_properties(L, P):
The predicate unifies P with the error properties of the knowledge base. The binary predicate allows specifying a locale L. The error resource bundles have to be loaded in advance via sys_load_resource/1.
get_description_properties(C, P):
get_description_properties(L, C, P):
The predicate unifies P with the description properties of the given capability C. The ternary predicate allows specifying a locale L.

The following locale Prolog flags are provided:

sys_locale:
Legal values are atoms as return by the Java method Locale.toString(). The flag indicates the current default locale. The value can be changed by the end-user.

Module domain

This module provides a couple of simple utilities to deal with the access to internationalized domain names (IDNs). The predicate make_domain/3 allows constructing and deconstructing user and host. The predicate works bidirectional without loss of data.

Example:

```
?- make_domain('foo','λ.com',X).
X = 'foo@λ.com'
```

The predicate host_lookup/2 can be used to perform a forward or reverse lookup of a host name. The predicate will fail if the host name is not known. The predicate ping_host/1 can be used to check the reachability of a host name. The Java internet libraries do not automatically a name preparation. Neither do our Prolog predicates presented so far.

Example:

```
?- uri_puny('http://zürich.ch/robots.txt', X).
X = 'http://xn--zrich-kva.ch/robots.txt'
```

Name preparation is for example required for host names. Domain name servers only work with ASCII represented host names and the recommended encoding of Unicode towards ASCII for host names is puny code. Such an encoding can be invoked by the predicate uri_puny/2 provided in this module.

The following domain predicates are provided:

make_domain(U, H, D):
: If U and H are variables, then the predicate succeeds when U and H unify with the user and the host of the domain D. Otherwise the predicates succeeds when D unifies with the constructed domain.

host_lookup(U, C):
: If U is a variable then the predicate succeeds when U unifies with reverse lookup of C. Otherwise the predicate succeeds when C unifies with the forward lookup of U.

ping_host(H):
: The predicate succeeds when the host H is reachable.

uri_puny(S, P):
: If S is a variable then the predicate succeeds when S unifies with the puny decode of P. Otherwise the predicate succeeds when P unifies with the puny encode of S.

sha1_hash(B, H):
: The predicate succeeds in H with the SHA-1 hash block of the block B.

Compatibility Matrix

t.b.d.

Table 11: Compatibility Matrix for the Token Syntax

Nr	*Description*	*System*
1	t.b.d.	
2	t.b.d.	
3	t.b.d.	
4	t.b.d.	

5.7 Miscellaneous Package

This package contains miscellaneous predicates. The following modules are provided:

- **Module text:** This module provides character classifications.
- **Module residue:** This module allows custom displayable constraints.
- **Module lock:** This module provides synchronization primitives.
- **Module pipe:** This module provides communication primitives.
- **Module time:** This module provides timing primitives.
- **Module socket:** This module provides server and client sockets.
- **Module http:** This module provides a simple HTTP server.
- **Compatibility Matrix:** t.b.d.

Module text

The predicates code_type/2 and char_type/2 can be used to classify a character. The character classifier is configurable and plug-able. To use a custom character classifier the predicates code_type/3 and char_type/3 can be used.

Examples:

```
?- char_type(a, X).
X = lower
?- char_type('A', X).
X = upper
```

The default character classifier uses the Prolog ISO core standard classification for the ASCII range plus the Jekejeke Prolog specific extension for Unicode that extends the range to the Unicode range. Codes outside of the range are classified as invalid.

The predicates code_lower/2 and code_upper/2 can be used for case conversion of code points. The predicates downcase_atom/2 and upcase_atom/2 can be used for case conversion of atoms.

Examples:

```
?- pattern_match('foobarfoo','bar').
No
?- pattern_match('foobarfoo','*bar*').
Yes
```

We also provide predicates for pattern matching. The predicate pattern_match/2 takes an atom and matches it against a pattern. The predicate pattern_replace/4 takes a further pattern and produces a new atom. The pattern language provided by us is inspired by the former NEBIS library system.

The following text predicates are provided:

char_type(C, N):
char_type(H, C, N):

The predicate succeeds for the name N of the Prolog classification of the character C. The ternary predicate allows specifying a custom classifier H. The following classification names are currently supported:

white:	The character is a white space.
control:	The character is a control character.
inval:	The character is invalid.
solo:	The character is a solo character.
score:	The character is an underscore.
upper:	The character is an upper case letter.
lower:	The character is a lower case letter.
other:	The character is some other alpha numerical character.
digit:	The character is a decimal digit.
graph:	The character is a graphic character.

code_type(C, N):
code_type(H, C, N):

The predicate succeeds for the name N of the Prolog classification of the code point C. The ternary predicate allows specifying a custom classifier H.

code_lower(C, D):
The predicate succeeds in D for the lower case of C.
code_upper(C, D):
The predicate succeeds in D for the upper case of C.
downcase_atom(A, B): [Prolog Commons Atom Utilities]
The predicate succeeds in B for the lower case of A.
upcase_atom(A, B): [Prolog Commons Atom Utilities]
The predicate succeeds in B for the upper case of A.
pattern_match(S, P):
pattern_match(S, P, O):
The predicate succeeds when the atom S matches the shell pattern P. The ternary predicate allows specifying match options O. The following match options are currently supported:

boundary(B):	B is the pattern boundary condition.
ignore_case(I):	I is the ignore case flag.
style(S):	S is the style.

The legal values for the pattern boundary condition are whole, part and word. The default value is whole. The legal values for the ignore case flag are true and false. The default value is false. The legal values for the style are create and parse. The default value is create.

pattern_replace(S, P, R, T):
pattern_replace(S, P, R, T, O):
The predicate succeeds when the atom S matches the shell pattern P, and when replacing the matched pattern by R yields the atom T. The quinary predicate allows specifying match and replaces options O.
last_pattern_replace(S, P, R, T):
last_pattern_replace(S, P, R, T, O):
These predicates work similar to the predicates replace/4 and replace/5 except that they search backwards.

Module residue

By default the top-level shows the current unification equations. An extension can show arbitrary constraints. It can do so by defining further clauses for the multi-file predicates sys_current_eq/2 and sys_unwrap_eq/3.

The constraints that are related directly or indirectly to a term can be retrieved by the predicate sys_term_eq_list/2. As a further convenience the predicate call_residue/2 allows calling a goal and retrieving the related constraints for each success.

Terms that are directly instantiated to a variable can be customized by the multi-file predicate sys_printable_value/2 and queried by the predicate printable/2. The former predicate should fail if there is no custom form and the later predicate will then return the original.

The following residue predicates are provided:

sys_current_eq(V, H):
: The predicate succeeds for each equation H with variables wrapped that listens on the variable V. Constraint solvers should extend this multi-file predicate.

sys_unwrap_eq(H, I, O):
: The predicate converts equation H with variables wrapped into equations I with variables unwrapped. The list uses the end O. Constraint solvers should extend this multi-file predicate..

sys_term_eq_list(T, L):
: The predicate unifies L with the list of constraints that depend directly or indirectly on the variables of G.

call_residue(G, L):
: The predicate succeeds whenever the goal G succeeds. The predicate unifies L with the list of constraints that depend directly or indirectly on the variables of G.

printable(F, G):
: The predicate succeeds in G with a custom form of F.

sys_printable_value(F, G):
: The predicate succeeds in G with a custom form of F. The predicate should be extended for custom forms.

Module lock

A mutex is a binary semaphore. A mutex can be created by the predicates mutex_new/1 and unslotted_new/2. A mutex need not be explicitly destroyed, it will automatically be reclaimed by the Java GC when not anymore used. To balance acquires and releases of the same semaphore the use of setup_call_cleanup/3 is recommended. Threads waiting for a semaphore can be interrupted.

Example:

```
?- mutex_new(M), lock_acquire(M), lock_release(M).
M = 0r3f10bc2a
```

The predicates lock_acquire/1 and lock_attempt/[1,2] allow incrementing a semaphore by one. These predicates will block, fail or timeout when the semaphore has already reached its maximum by other threads. The predicate lock_release/1 allows decrementing the semaphore by one, provided it is not already zero. The slotted versions check that the owner doesn't change, but currently do not allow re-entrancy.

A read write pair can be created by the predicates lock_new/1 and nonescalable_new/1. In the non-escalable version the non-binary read semaphore can be retrieved by the predicate get_read/2 and it can be incremented provided the write semaphore is zero. The binary write semaphore can be retrieved by the predicate get_write/2 and it can be incremented provided the read semaphore is zero.

For the escalated version of the read write pair it is also allowed that the same thread holds a read and a write lock from a read write pair. This can for example be used to upgrade or downgrade a read write pair by using unbalanced locking. For example if a thread already holds a write lock, it can acquire the read lock and then release the write lock. The result is that the write lock was changed into a read lock.

The following lock predicates are provided:

mutex_new(M):
The predicate succeeds for a new slotted mutex M.

unslotted_new(M):
The predicate succeeds for a new unslotted mutex M.

lock_aquire(L):
The predicate succeeds after locking the lock L.

lock_attempt(L):
The predicate succeeds after locking the lock L. Otherwise the predicate fails.

lock_attempt(L, T):
The predicate succeeds after locking the lock L in the timeout T. Otherwise the predicate fails.

lock_release(L):
The predicate succeeds after unlocking the lock L.

lock_new(P):
The predicate succeeds for a new slotted and escalable read write pair P.

nonescalable_new(P):
The predicate succeeds for a new unslotted and non-escalable read write pair P.

get_read(P, R):
The predicate succeeds for the read lock R of the read write pair P.

get_write(P, W):
The predicate succeeds for the write lock W of the read write pair P.

Module pipe

Pipes allow exchanging messages. Messages are Prolog terms and are copied. An unbounded queue can be created by the predicate pipe_new/1. A bounded queue can be created by the predicate pipe_new/2. Pipes need not be explicitly destroyed, they will automatically be reclaimed by the Java GC when not anymore used. Threads waiting for a pipe can be interrupted.

Example:

```
?- queue new(1, Q), pipe put(Q, p(X)), pipe take(Q, R).
Q = 0ra2a372,
R = p(_A)
```

The predicates pipe_put/2 and pipe_offer/[2,3] allow sending a message to a bounded queue. The predicates will block, fail or timeout when the bounded queue is full. The predicate pipe_put/2 can also be used for unbounded queues and will never block. The predicates pipe_take/3 and pipe_poll/[2,3] allow getting a message from a pipe. The predicates will block, fail or timeout when the pipe is empty.

The following pipe predicates are provided:

pipe_new(Q):
The predicate succeeds for a new unbounded queue Q.

pipe_new(M, Q):
The predicate succeeds for a new bounded queue Q with maximum size M.

pipe_put(P, O):
The predicate succeeds for sending a copy of the term O to the pipe P.

pipe_offer(P, O):
The predicate succeeds for sending a copy of the term O to the bounded queue P. Otherwise the predicate fails.

pipe_offer(P, O, T):
The predicate succeeds for sending a copy of the term O to the bounded queue P in the timeout T. Otherwise the predicate fails.

pipe_take(P, O):
The predicate succeeds for getting a term O form the pipe P.

pipe_poll(P, O):
The predicate succeeds for getting a term O form the pipe P. Otherwise the predicate fails.

pipe_poll(P, T, O):
The predicate succeeds for getting a term O form the pipe P in the timeout T. Otherwise the predicate fails.

Module time

Alarm queues allow scheduling items. Items are Prolog terms and are copied. An alarm queue can be created by the predicate alarm_new/1. An alarm queue need not be explicitly destroyed, it will automatically be reclaimed by the Java GC when not anymore used. Threads waiting for an alarm queue can be interrupted.

Example:

```
?- time_out((repeat, write('Hello World!'), nl,
       thread_sleep(1000), fail), 3000).
Hello World!
Hello World!
Hello World!
Error: Execution aborted since time limit exceeded.
      thread_sleep/1
      time_out/2
```

An item can be scheduled with the predicate alarm_schedue/4 giving a delay in milliseconds. The predicate alarm_next/2 allows getting an item from a queue. The predicate will block for the earliest item. The predicate alarm_cancel/2 will remove an item from the queue.

The predicate time_out/2 uses a predefined alarm queue which is served by a predefined thread. The predicate executes the given goal once in the given timeout. When the timeout is reached before the goal completes an exception is thrown.

The following time predicates are provided:

alarm_new(A):
The predicate succeeds for a new alarm queue A.

alarm_schedule(A, O, T, E):
The predicate succeeds for a new alarm entry E that schedules a copy of the term O on the alarm queue A with a delay of T.

alarm_next(A, O):
The predicate succeeds for the next term O on the alarm queue A. The predicate blocks for the earliest item.

alarm_cancel(A, E):
The predicate succeeds for cancelling the alarm entry E from the alarm queue A.

time_out(G, T):
The predicate succeeds when G succeeds in the timeout T. The predicate fails when G fails in the timeout T. Otherwise the predicate throws the message system_error(timelimit_exceeded).

Module socket

This module provides TCP/IP sockets. A socket provides a duplex binary stream. The read and write stream can be obtained by the ordinary ISO core standard open/[3,4] predicates. The open options also apply to sockets so that a binary stream can be easily viewed as a text stream in various encodings.

Example:

```
?- client_new('pot.ty', C), open(C, write, S),
   write_term(S, 'Hello World!'), nl(S), close(C).
```

A server socket can be created with the predicates server_new/2. The predicate server_accept/2 delivers a session socket. A client socket can be created with the predicates client_new/3. Server, session and client sockets can be closed with the ISO core standard close/[1,2] predicates.

The predicate websock_new/2 allows promoting a socket to a web socket. The input and output streams will consume and generate web socket frames, but can be used as ordinary ISO core standard streams. During writing a final frame is generated when the predicate flush_output/[1,2] is used.

The following socket predicates are provided:

server_new(P, S):
The predicate succeeds in S with a new server socket for port P.
server_port(S, P):
The predicate succeeds in P with the port of the server socket S.
server_accept(S, H):
The predicate succeeds in H with a new session socket from server socket S.
client_new(H, P, C):
The predicate succeeds in C with a new client socket for host H and port P.
websock_new(S, W):
The predicate succeeds in a web socket W for the socket S.

Module http

This module provides a HTTP server based on Pythonesk dispatch of a server object. The class of the server object need only implement a predicate dispatch/4 with the Pythonesk convention that the receiver appears in the first argument. The server can be started by providing the server object that will be responsible for handling HTTP requests:

```
?- run_http(<object>, <port>), fail; true.
```

The server currently implements a minimal subset of the HTTP/1.1 protocol restricted to GET method. The server will read the request line and the header lines. The server is able to generate error messages in the case the request is erroneous or in case the server object cannot handle the request. The following HTTP/1.1 errors have been realized:

- **400 Bad Request:** Request could not be parsed.
- **404 Not Found:** Server object did not succeeds.
- **501 Not Implemented:** Request method not supported.

The predicate http_parameter/3 can be used by the server object to access URI query parameters. The predicate response_text/1, response_binary/1 and html_escape/1 can be used to generate dynamic content by the server object. The predicates handle_text/2 and handle_binary/2 can be used by the server object to deliver static content.

The following HTTP server predicates are provided:

run_http(O, P):
The predicate runs a web server with object O at port P.

http_parameter(R, N, V):
The predicate succeeds in V with the value of the parameter named N from the request R.

http_header(R, N, V):
The predicate succeeds in V with the value of the header named N from the request R.

response_text(O):
Send an OK response to the text output stream.

handle_text(F, O):
The predicate sends the HTML resource F to the socket O.

response_binary(O):
Send an OK response to the binary output stream.

handle_binary(F, O):
The predicate sends the binary resource F to the output socket O.

html_escape(O, T):
The predicate sends the text T escaped to the output stream O.

response_redirect(L, O):
Send a redirect response to location L to the text output stream O.

response_upgrade(R, O):
Send an upgrade response from request R to the binary output stream O.

The following abstract HTTP server predicates are provided:

initialized(O, S):
The predicate is called when the server S is initialized for object O.
destroyed(O, S):
The predicate is called when the server S is destroyed for object O.
dispatch(O, P, R, S):
The predicate succeeds in dispatching the request for object O, with path P, with request R and the session S.
upgrade(O, P, R, S):
The predicate succeeds in upgrading the request for object O, with path P, with request R and the session S.

Compatibility Matrix

t.b.d.

Table 12: Compatibility Matrix for the Token Syntax

Nr	*Description*	*System*
1	t.b.d.	
2	t.b.d.	
3	t.b.d.	
4	t.b.d.	

6 Appendix Example Listings

The full source code of the Prolog texts for the language examples is given. The following source code has been included:

6.1 Flag Example

For the flag example there are the following sources:

- **flag.p:** The Prolog text for the loop without bindings.
- **flag2.p:** The Prolog text for the loop with bindings.

Prolog Text flag

```
/**
 * Prolog code for the closure without bindings example.
 *
 * Copyright 2012, XLOG Technologies GmbH, Switzerland
 * Jekejeke Prolog 1.0 (a fast and small prolog interpreter)
 */

% between(+Integer, +Integer, -Integer)
between(Lo, Hi ,_) :- Lo > Hi, !, fail.
between(Lo, _, Lo).
between(Lo, Hi, X) :- Lo2 is Lo+1, between(Lo2, Hi, X).

% for1(+Integer, +Integer, +Closure)
for1(Lo, Hi, Closure) :-
  between(Lo, Hi, Value),
  call(Closure, Value),
  fail.
for1(_,_,_).

% flag
flag :-
  for1(1,8,X\
     (for1(1,8,Y\
        (0 =:= (X+Y) mod 2 -> write(x); write(o))), nl)).
```

Prolog Text flag2

```
/**
 * Prolog code for the closure with bindings example.
 *
 * Copyright 2012, XLOG Technologies GmbH, Switzerland
 * Jekejeke Prolog 1.0 (a fast and small prolog interpreter)
 */

% for2(+Integer, +Integer, +Closure)
for2(Lo, Hi, _) :- Lo > Hi, !.
for2(Lo, Hi, Closure) :-
   call(Closure, Lo),
   Lo2 is Lo+1,
```

```
   for2(Lo2, Hi, Closure).

% flag
flag(R) :-
  functor(R,'',8),
  for2(1,8,X\S^
     (arg(X,R,S),
      functor(S,'',8),
      for2(1,8,Y\T^
        (arg(Y,S,T),
         (0 =:= (X+Y) mod 2 -> T=x; T=o))))).
```

6.2 Palindrome Example [ISO]

For the palindrome example there are the following sources:

- **palin.p**: The Prolog text of the grammar without attributes.
- **palin2.p:** The Prolog text of the grammar with attributes.

Prolog Text palin

```
/**
 * Prolog code for the palindrome without attributes example.
 *
 * Copyright 2010, XLOG Technologies GmbH, Switzerland
 * Jekejeke Prolog 0.8.3 (a fast and small prolog interpreter)
 */

palin --> [_].
palin --> [Middle, Middle].
palin --> [Border], palin, [Border].
```

Prolog Text palin2

```
/**
 * Prolog code for the palindrome with attributes example.
 *
 * Copyright 2010, XLOG Technologies GmbH, Switzerland
 * Jekejeke Prolog 0.8.3 (a fast and small prolog interpreter)
 */

palin([], [Middle]) --> [Middle].
palin([Middle], []) --> [Middle, Middle].
palin([Border | List], Middle) --> [Border], palin(List, Middle), [Border].
```

6.3 Fruits Example

For the fruits example there are the following sources:

- **fruit.p**: The Prolog text of the grammar without attributes.
- **fruit2.p:** The Prolog text of the grammar with attributes.

Prolog Text fruit

```
/**
 * Prolog code for the fruits without attributes example.
 *
 * Copyright 2011, XLOG Technologies GmbH, Switzerland
 * Jekejeke Prolog 0.9.2 (a fast and small prolog interpreter)
 */

repetition(G) --> G, repetition(G).
repetition(_) --> [].

fruit --> "apple".
fruit --> "orange".
fruit --> "pear".

fruits --> fruit, repetition((",", fruit)).
```

Prolog Text fruit2

```
/**
 * Prolog code for the fruits with attributes example.
 *
 * Copyright 2011, XLOG Technologies GmbH, Switzerland
 * Jekejeke Prolog 0.9.2 (a fast and small prolog interpreter)
 */

repetition(G,[X|Y]) --> call(G,X), repetition(G,Y).
repetition(_,[]) --> [].

fruit(apple) --> "apple".
fruit(orange) --> "orange".
fruit(pear) --> "pear".

fruits([X|Y]) --> fruit(X), repetition(Z\(",", fruit(Z)),Y).
```

Acknowledgements

Many thanks go to Jan Wielemaker who was open for discussions and also confirmed that our thread clean already exists in the form of Jeff Roses trick.

Further thanks go to Robert Klemme, Andreas Leitgeb and Eric Sosman who shared my general struggles with escalable read write locks.

Indexes

Public Predicates

Predicate	Module
!/2	standard/dcg
*-> /4	standard/dcg
,/4	standard/dcg
--> /2	standard/dcg
-> /4	standard/dcg
. /4	standard/dcg
/\ /2	standard/expand
: /4	standard/apply
: /5	standard/apply
: /6	standard/apply
: /7	standard/apply
: /8	standard/apply
: /9	standard/apply
:: /4	standard/apply
:: /5	standard/apply
:: /6	standard/apply
:: /7	standard/apply
:: /8	standard/apply
:: /9	standard/apply
;/4	standard/dcg
[]/2	standard/dcg
(\)/3	experiment/abstract
(\)/4	experiment/abstract
(\)/5	experiment/abstract
(\)/6	experiment/abstract
(\)/7	experiment/abstract
(\)/8	experiment/abstract
(\)/9	experiment/abstract
(\+)/3	standard/dcg
^ /2	standard/expand
^ /4	experiment/abstract
^ /5	experiment/abstract
^ /6	experiment/abstract
^ /7	experiment/abstract
^ /8	experiment/abstract
^ /9	experiment/abstract
above/2	advanced/arith
acosh/2	basic/hyper
alarm_cancel/2	misc/time
alarm_new/1	misc/time

alarm_next/2	misc/time
alarm_schedule/4	misc/time
append/3	basic/lists
asinh/2	basic/hyper
assertable_ref/2	experiment/ref
assumable_ref/2	experiment/ref
at_end_of_stream/0	stream/byte
at_end_of_stream/1	stream/byte
atanh/2	basic/hyper
atom_format/3	system/locale
atom_format/4	system/locale
atom_lower/2	misc/text
atom_upper/2	misc/text
bagof/3	standard/bags
between/3	advanced/arith
call/2	standard/apply
call/3	standard/apply
call/4	standard/apply
call/5	standard/apply
call/6	standard/apply
call/7	standard/apply
call/8	standard/apply
call_cleanup/2	standard/signal
call_residue/2	misc/residue
canonical_path/2	system/file
canonical_uri/2	system/uri
char_type/2	misc/text
char_type/3	misc/text
clause_ref/3	experiment/ref
close/1	stream/stream
close/2	stream/stream
code_lower/2	misc/text
code_type/2	misc/text
code_type/3	misc/text
code_upper/2	misc/text
compiled_ref/2	experiment/ref
contains/2	advanced/sets
copy_term/2	standard/bags
cosh/2	basic/hyper
counter_new/1	basic/random
counter_next/2	basic/random
current_error/1	stream/stream
current_input/1	stream/stream
current_local/1	experiment/surrogate
current_output/1	stream/stream
delete_file/1	system/file
difference/3	advanced/sets
directory_file/2	system/file
erase_ref/1	experiment/ref
error_make/3	system/locale
error_make/4	system/locale
exists_directory/1	system/file
exists_file/1	system/file
expand_goal/2	standard/expand
expand_term/2	standard/expand
fail/2	standard/dcg

findall/3	standard/bags
flush_output/0	stream/byte
flush_output/1	stream/byte
follow_path/3	system/file
follow_uri/3	system/uri
foreign_dimension/2	basic/array
foreign_element/2	basic/array
foreign_length/2	basic/array
foreign_member/2	basic/array
foreign_update/2	basic/array
format/2	stream/console
format/3	stream/console
free_local/1	experiment/surrogate
get/3	experiment/maps
get_byte/1	stream/byte
get_byte/2	stream/byte
get_char/1	stream/char
get_char/2	stream/char
get_code/1	stream/char
get_code/2	stream/char
get_description_properties/2	system/locale
get_description_properties/3	system/locale
get_error_properties/1	system/locale
get_error_properties/2	system/locale
get_local/2	experiment/surrogate
get_property/3	system/locale
get_property/4	system/locale
get_read/2	misc/lock
get_time/1	system/shell
get_time/2	system/shell
get_time_file/2	system/file
get_write/2	misc/lock
getenv/2	system/shell
goal_expansion/2	standard/expand
goal_expansion/2	standard/dcg
hash_code/2	standard/sort
intersection/3	advanced/sets
is_relative_path/1	system/file
is_relative_uri/1	system/uri
keysort/2	standard/sort
last/2	basic/lists
last/3	basic/lists
last_pattern_replace/4	misc/text
last_pattern_replace/5	misc/text
length/2	basic/lists
limit/2	advanced/sequence
locale_keysort/2	standard/sort
locale_keysort/3	standard/sort
locale_sort/2	standard/sort
locale_sort/3	standard/sort
lock_acquire/1	misc/lock
lock_attempt/1	misc/lock
lock_attempt/2	misc/lock
lock_new/1	misc/lock
lock_release/1	misc/lock
make_directory/1	system/file

make_name/3	system/file
make_path/3	system/file
make_query/4	system/uri
make_spec/4	system/uri
make_uri/4	system/uri
member/2	basic/lists
message_make/3	system/locale
message_make/4	system/locale
mutex_new/1	misc/lock
new_local/2	experiment/surrogate
nl/0	stream/char
nl/1	stream/char
nonescalable_new/1	misc/lock
nth0/3	basic/lists
nth0/4	basic/lists
nth1/3	basic/lists
nth1/4	basic/lists
offset/2	advanced/sequence
open/3	stream/stream
open/4	stream/stream
ord_contains/2	advanced/ordsets
ord_difference/3	advanced/ordsets
ord_get/3	experiment/ordmaps
ord_intersection/3	advanced/ordsets
ord_put/4	experiment/ordmaps
ord_remove/3	experiment/ordmaps
ord_subset/2	advanced/ordsets
ord_union/3	advanced/ordsets
pattern_match/2	misc/text
pattern_match/3	misc/text
pattern_replace/4	misc/text
pattern_replace/5	misc/text
peek_byte/1	stream/byte
peek_byte/2	stream/byte
peek_char/1	stream/char
peek_char/2	stream/char
peek_code/1	stream/char
peek_code/2	stream/char
permutation/2	advanced/sets
phrase/2	standard/dcg
phrase/3	standard/dcg
user:phrase/3	experiment/tecto
phrase_abnormal/1	standard/dcg
user:phrase_abnormal/1	experiment/tecto
phrase_expansion/4	standard/dcg
user:phrase_expansion/4	experiment/tecto
pipe_new/1	misc/pipe
pipe_new/2	misc/pipe
pipe_offer/2	misc/pipe
pipe_offer/3	misc/pipe
pipe_poll/2	misc/pipe
pipe_poll/3	misc/pipe
pipe_put/2	misc/pipe
pipe_take/2	misc/pipe
plus/3	advanced/arith
print_error/1	stream/console

print_error/2	stream/console
print_message/1	stream/console
print_message/2	stream/console
print_stack_trace/1	stream/console
print_stack_trace/2	stream/console
put/4	experiment/maps
put_byte/1	stream/byte
put_byte/2	stream/byte
put_char/1	stream/char
put_char/2	stream/char
put_code/1	stream/char
put_code/2	stream/char
random/1	basic/random
random/2	basic/random
random_new/1	basic/random
random_new/2	basic/random
random_next/2	basic/random
random_next/3	basic/random
read/1	stream/term
read/2	stream/term
read_line/1	stream/console
read_line/2	stream/console
read_term/2	stream/term
read_term/3	stream/term
recorda_ref/1	experiment/ref
recordz_ref/1	experiment/ref
remove/3	advanced/sets
remove/3	experiment/maps
rename_file/2	system/file
reverse/2	basic/lists
select/3	basic/lists
set_error/1	stream/stream
set_input/1	stream/stream
set_local/2	experiment/surrogate
set_output/1	stream/stream
set_stream_length/2	stream/stream
set_stream_position/2	stream/stream
set_time_file/2	system/file
setof/3	standard/bags
setup_call_cleanup/3	standard/signal
sinh/2	basic/hyper
sort/2	standard/sort
stream_property/2	stream/stream
subset/2	advanced/sets
sys_atomic/1	standard/signal
sys_clean_thread/1	misc/clean
sys_clean_threads/2	misc/clean
sys_cleanup/1	standard/signal
sys_current_eq/2	misc/residue
sys_distinct/2	standard/sort
sys_get_lang/2	system/locale
sys_get_lang/3	system/locale
sys_goal_rebuilding/2	experiment/simp
sys_goal_simplification/2	experiment/simp
sys_heapof/3	standard/bags
sys_instance/1	basic/proxy

sys_instance/2	basic/proxy
sys_instance_of/2	basic/proxy
sys_instance_size/2	basic/proxy
sys_instance_size/3	basic/proxy
sys_keygroup/2	standard/sort
sys_modext_args/3	standard/apply
sys_modext_args/4	standard/apply
sys_modext_args/5	standard/apply
sys_modext_args/6	standard/apply
sys_modext_args/7	standard/apply
sys_modext_args/8	standard/apply
sys_modext_args/9	standard/apply
sys_new_surrogate/1	experiment/surrogate
sys_phrase/3	standard/dcg
sys_phrase_delay/1	standard/dcg
user:sys_phrase_delay/1	experiment/tecto
sys_phrase_expansion/4	standard/dcg
user:sys_phrase_expansion/4	experiment/tecto
sys_portray_eq/2	misc/residue
sys_rebuild_goal/2	experiment/simp
sys_rebuild_goal_arg/3	experiment/simp
sys_rebuild_term/2	experiment/simp
sys_receiver_class/2	basic/proxy
sys_simplify_goal/2	experiment/simp
sys_simplify_term/2	experiment/simp
sys_subclass_of/2	basic/proxy
sys_term_eq_list/2	misc/residue
sys_term_hash/3	standard/sort
sys_term_object/3	basic/proxy
sys_term_rebuilding/2	experiment/simp
sys_term_simplification/2	experiment/simp
sys_unwrap_eq/2	misc/residue
tanh/2	basic/hyper
term_expansion/2	standard/expand
term_expansion/2	standard/dcg
term_hash/2	standard/sort
term_hash/4	standard/sort
text_escape/2	system/xml
thread_abort/2	system/thread
thread_combine/1	system/thread
thread_combine/2	system/thread
thread_current/1	system/thread
thread_down/2	system/thread
thread_down/3	system/thread
thread_join/1	system/thread
thread_kill/1	system/thread
thread_new/2	system/thread
thread_sleep/1	system/thread
thread_start/1	system/thread
time_out/2	misc/time
ttyflush_output/0	stream/console
ttynl/0	stream/console
ttyread_line/1	stream/console
ttywrite/1	stream/console
ttywrite_term/2	stream/console
union/3	advanced/sets

unit/0	standard/expand
unslotted_new/1	misc/lock
uri_encode/2	system/uri
write/1	stream/term
write/2	stream/term
write_canonical/1	stream/term
write_canonical/2	stream/term
write_term/2	stream/term
write_term/3	stream/term
writeq/1	stream/term
writeq/2	stream/term
{}/3	standard/dcg

Package Local Predicates

Predicate	Module
sys_flush_output/1	stream/byte
sys_get_alias/2	stream/stream
sys_nl/1	stream/char
sys_write/2	stream/term
sys_write_term/3	stream/term

Non-Private Meta-Predicates

Predicate	Exp	Body	Rule	Module
0/\0	yes	no	yes	standard/expand
\(?,0,?)	yes	no	no	experiment/abstract
? ^0	yes	yes	no	standard/expand
assertable_ref(-1,?)	yes	no	no	experiment/ref
assumable_ref(-1,?)	yes	no	no	experiment/ref
bagof(?,0,?)	yes	no	no	standard/bags
call_cleanup(0,0)	yes	no	no	standard/signal
call_residue(0,?)	yes	no	no	misc/residue
clause_ref(-1,0,?)	no	no	no	experiment/ref
compiled_ref(?,-1)	no	no	no	experiment/ref
expand_goal(0,0)	no	no	no	standard/expand
expand_term(-1,-1)	no	no	no	standard/expand
findall(?,0,?)	yes	no	no	standard/bags
goal_expansion(0,0)	no	no	no	standard/expand
goal_expansion(0,0)	no	no	no	standard/dcg
limit(?,0)	yes	no	no	advanced/sequence
offset(?,0)	yes	no	no	advanced/sequence
setof(?,0,?)	yes	no	no	standard/bags
setup_call_cleanup(0,0,0)	yes	no	no	standard/signal
sys_atomic(0)	yes	no	no	standard/signal
sys_clean_thread(0)	yes	no	no	misc/clean
sys_clean_threads(0,?)	yes	no	no	misc/clean
sys_cleanup(0)	yes	no	no	standard/signal
sys_current_eq(?,0)	yes	no	no	misc/residue
sys_goal_rebuilding(0,0)	no	no	no	experiment/simp
sys_goal_simplification(0,0)	no	no	no	experiment/simp
sys_heapof(?,0,?)	yes	no	no	standard/bags
sys_portray_eq(0,0)	yes	no	no	misc/residue

sys_rebuild_goal(0,0)	no	no	no	experiment/simp
sys_rebuild_term(-1,-1)	no	no	no	experiment/simp
sys_simplify_goal(0,0)	no	no	no	experiment/simp
sys_simplify_term(-1,-1)	no	no	no	experiment/simp
sys_term_rebuilding(-1,-1)	no	no	no	experiment/simp
sys_term_simplification(-1,-1)	no	no	no	experiment/simp
sys_unwrap_eq(0,0)	yes	no	no	misc/residue
term_expansion(-1,-1)	no	no	no	standard/expand
term_expansion(-1,-1)	no	no	no	standard/dcg
thread_new(0,?)	yes	no	no	system/thread
time_out(0,?)	yes	no	no	misc/time
{}(0,?,?)	yes	no	no	standard/dcg

Non-Private Closure-Predicates

Predicate	Module
*->(2,2,?,?)	standard/dcg
,(2,2,?,?)	standard/dcg
2--> -3	standard/dcg
->(2,2,?,?)	standard/dcg
.(2,2,?,?)	standard/dcg
:(?,2,?,?)	standard/apply
:(?,3,?,?,?)	standard/apply
:(?,4,?,?,?,?)	standard/apply
:(?,5,?,?,?,?,?)	standard/apply
:(?,6,?,?,?,?,?,?)	standard/apply
:(?,7,?,?,?,?,?,?,?)	standard/apply
::(?,::(2),?,?)	standard/apply
::(?,::(3),?,?,?)	standard/apply
::(?,::(4),?,?,?,?)	standard/apply
::(?,::(5),?,?,?,?,?)	standard/apply
::(?,::(6),?,?,?,?,?,?)	standard/apply
::(?,::(7),?,?,?,?,?,?,?)	standard/apply
;(2,2,?,?)	standard/dcg
\(?,1,?,?)	experiment/abstract
\(?,2,?,?,?)	experiment/abstract
\(?,3,?,?,?,?)	experiment/abstract
\(?,4,?,?,?,?,?)	experiment/abstract
\(?,5,?,?,?,?,?,?)	experiment/abstract
\(?,6,?,?,?,?,?,?,?)	experiment/abstract
\+(2,?,?)	standard/dcg
^(?,2,?,?)	experiment/abstract
^(?,3,?,?,?)	experiment/abstract
^(?,4,?,?,?,?)	experiment/abstract
^(?,5,?,?,?,?,?)	experiment/abstract
^(?,6,?,?,?,?,?,?)	experiment/abstract
^(?,7,?,?,?,?,?,?,?)	experiment/abstract
call(1,?)	standard/apply
call(2,?,?)	standard/apply
call(3,?,?,?)	standard/apply
call(4,?,?,?,?)	standard/apply
call(5,?,?,?,?,?)	standard/apply
call(6,?,?,?,?,?,?)	standard/apply
call(7,?,?,?,?,?,?,?)	standard/apply
phrase(2,?)	standard/dcg

phrase(2,?,?)	standard/dcg
user:phrase(2,?,?)	experiment/tecto
phrase_expansion(2,?,?,0)	standard/dcg
user:phrase_expansion(2,?,?,0)	experiment/tecto
sys_phrase(2,?,?)	standard/dcg
sys_phrase_expansion(2,?,?,0)	standard/dcg
user:sys_phrase_expansion(2,?,?,0)	experiment/tecto

Non-Private Syntax Operators

Level	Mode	Operator	Module
1200	xfx	-->	standard/dcg
200	xfy	\	experiment/abstract

Pictures

Es konnten keine Einträge für ein Abbildungsverzeichnis gefunden werden.

Tables

Acronyms

DCGD [1]

References

[1] ISO/IEC DTR 13211{3:2006}, Definite clause grammar rules, Jonathan Hodgson jpehodgson@verizon.net, July 30, 2015
http://www.complang.tuwien.ac.at/ulrich/iso-prolog/dcgs/dcgsdraft-2015-07-30.pdf

[2] Szeredi, P. (1999): Contributions To Or-Parallel Logic Programming, PhD Thesis, Péter Szeredi, Techical University of Budapest, December 1997
http://www.cs.bme.hu/~szeredi/docs/SzerediPhd.pdf