Jekejeke Runtime Reference

Version 1.3.5, February 01th, 2019

XLOG Technologies GmbH

Jekejeke Prolog

Runtime Library 1.3.5

Language Reference

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

Date: February 01th, 2019
Version: 0.68

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, November 07th, 2009, 0.1:
- Initial Version.

Jan Burse, November 21st, 2009, 0.2:
- Added sections text syntax and structure theory.

Jan Burse, February 22nd, 2010, 0.3:
- Added sections consult theory and control theory.

Jan Burse, March 05th, 2010, 0.4:
- Fixed some compatibility issues.

Jan Burse, Mai 14th, 2010, 0.5:
- Enhancement of control theory.

Jan Burse, Mai 18th, 2010, 0.6:
- Better exception handling and more compatibility issues fixed.

Jan Burse, June 02nd, 2010, 0.7:
- Better exception handling and more compatibility issues fixed.

Jan Burse, July 02nd, 2010, 0.8:
- Code styling, grammar rules, arithmetic operations and Prolog flags.

Jan Burse, September 02nd, 2010, 0.9:
- Bitwise operations, set predicates and more compatibility issues fixed.

Jan Burse, October 02nd, 2010, 0.10:
- Predicate testing predicates and more compatibility issues fixed.

Jan Burse, November 08th, 2010, 0.11:
- Data types reference and decimal introduced.

Jan Burse, November 18th, 2010, 0.12:
- Number and string syntax enhanced.

Jan Burse, November 22nd, 2010, 0.13:
- String predicates and character input/output enhanced.

Jan Burse, November 24th, 2010, 0.14:
- Basic stream control introduced and input/output by stream arguments enhanced.

Jan Burse, December 02nd, 2010, 0.15:
- Advanced stream control and module transparent introduced.

Jan Burse, January 02nd, 2011, 0.16:
- Stream and consult theory enhanced.

Jan Burse, April 15th, 2011, 0.17:
- Unicode extension document, syntax enhanced and stream theory enhanced.

Jan Burse, April 25th, 2011, 0.18:
- Development theory moved into separate document and capability predicates.

Jan Burse, Mai 06th, 2011, 0.19:
- Clause expansion, flags & properties and source files section introduced.

Jan Burse, June 15th, 2011, 0.20:
- Multi-threading and byte I/O section introduced.

Jan Burse, August 11th, 2011, 0.21:
- Definite clause grammar, strings, variable names and Unicode handling updated.

Jan Burse, August 23th, 2011, 0.22:
- Clause indexing and optimization sections updated.

Jan Burse, September 17th, 2011, 0.23:
- Kernel predicate and signal handling section updated.

Jan Burse, October 06th, 2011, 0.24:
- References added, stream section and clause expansion section updated.

Jan Burse, November 13th, 2011, 0.25:
- Body conversion, higher order and fruits example section introduced.

Jan Burse, February 16th, 2012, 0.26:
- Message fixes, conversations integrated and few enhancements.

Jan Burse, June 4th, 2012, 0.27:
- Taxonomy example introduced and few enhancements.

Jan Burse, July 16th, 2012, 0.28:
- Money example introduced and few enhancements.

Jan Burse, August 22th, 2012, 0.29:
- Flag example introduced and few enhancements.

Jan Burse, September 06th, 2012, 0.30:
- Aggregate predicates introduced and few enhancements.

Jan Burse, October 30th, 2012, 0.31:
- Index section, call-site transfer section and few enhancements introduced.

Jan Burse, January 12th, 2013, 0.32:
- Random functions introduced and stability analysis removed.

Jan Burse, May 07th, 2013, 0.33:
- New sys_notrace flag hierarchy and new lambda operator introduced.

Jan Burse, June 05th, 2013, 0.34:
- Surrogate pairs introduced and quote flags made settable.

Jan Burse, July 23th, 2013, 0.35:
- Pseudo modules introduced.

Jan Burse, December 02nd, 2013, 0.36:
- Pseudo module visibility for foreign predicates introduced.

Jan Burse, February 03rd, 2014, 0.37:
- New decimal compression, build system and file options introduced.

Jan Burse, March 23rd, 2014, 0.38:
- Module system introduced.

Jan Burse, June 04th, 2014, 0.39:
- Syntax operator properties and module re-export introduced.

Jan Burse, July 14th, 2014, 0.40:
- Non-essential modules and examples moved to frequent document.

Jan Burse, August 05th, 2014, 0.41:
- Some fixes and new predicates for forward chaining.

Jan Burse, August 22nd, 2014, 0.42:
- Better printing and better foreign predicates.

Jan Burse, February 22nd, 2015, 0.43:
- Ubiquitous possibly qualified predicate indicator.

Jan Burse, May 08th, 2015, 0.44:
- Some predicates for the new locale module and message lists removed.

Jan Burse, May 28th, 2015, 0.45:
- Document title page introduced.

Jan Burse, July 11th, 2015, 0.46:
- New solution REPL documented.

Jan Burse, August 05th, 2015, 0.47:
- DCG references removed and total rework of clause/retract/assert zoo.

Jan Burse, September 02nd, 2015, 0.48:
- Improved foreign function interface and evaluable function handling.

Jan Burse, October 20th, 2015, 0.49:
- Improved resource bundle handling.

Jan Burse, December 25th, 2015, 0.50:
- New anonymous import directive.

Jan Burse, February 15th, 2016, 0.51:
- Reference type evaluation and (::)/2 associativity.

Jan Burse, March 06th, 2016, 0.52:
- The throw/1 predicate does now context filling.

Jan Burse, April 12th, 2016, 0.53:
- New miscellaneous definitions section and new indexes section.

Jan Burse, May 16th, 2016, 0.54:
- Some predicates moved from Jekejeke Minlog to here.

Jan Burse, June 23th, 2016, 0.55:
- Different error message for undefined in bridging.

Jan Burse, August 05th, 2016, 0.56:
- New module “distributed”.

Jan Burse, September 25th, 2016, 0.57:
- New evaluable expression meta-specifier.

Jan Burse, December 17th, 2016, 0.58:
- Some improvements in the decimals.

Jan Burse, April 23th, 2017, 0.59:
- Some improvements in the residues.

Jan Burse, July 18th, 2017, 0.60:
- Some improvements in the atoms.

Jan Burse, October 11th, 2017, 0.61:
- New variable names handling.

Jan Burse, February 01th, 2018, 0.62:
- New help utilities introduced.

Jan Burse, May 10th, 2018, 0.63:
- Simplification and changes from hierarchical knowledge bases.

Jan Burse, July 05th, 2018, 0.64:
- Tagged structures and zero argument syntax introduced.

Jan Burse, September 28th, 2018, 0.65:
- Improved module reflection.

Jan Burse, November 19th, 2018, 0.66:
- Better nursery for load balancing.

Jan Burse, January 04th, 2019, 0.67:
- Some improvements in the atoms and pseudo strings.

Jan Burse, February 01th, 2019, 0.68:
- The tty flags have been removed.

1 Introduction

This document gives a reference of the Jekejeke Prolog programming language. The language is motivated by mathematical logic but it is not fully declarative since it has still procedural elements, which destroy a simple semantic. Here and there we will compare our definitions with the Edinburgh Prolog standard [1] and the ISO Prolog standard [2].

- **Prolog Examples:** We show some examples of the use of the Jekejeke Prolog programming language. Readers who might be interested in getting a quick grip of the Jekejeke Prolog programming language and who have already a basic knowledge of Prolog might stick to this section only.

- **Prolog Conversations:** The Jekejeke Prolog runtime library provides character terminal based interactions. Among the interactions we find query answering and source consulting.

- **Prolog Syntax:** In this section we show what syntax the Jekejeke Prolog interpreter accepts and how the syntax relates to the mathematical concepts. The syntax covers multiple levels consisting of tokens, terms and texts. The syntax is dynamically extensible by operator definitions.

- **Prolog Theories:** The Jekejeke Prolog programming language comes with a standard set of predefined predicates. Predicates can be grouped into theories and we present them as such.

- **Appendix Example Listing:** The full source code of Prolog examples is given.

2 Prolog Examples

We show some examples of the use of the Jekejeke Prolog programming language. Readers who might be interested in getting a quick grip Jekejeke Prolog programming language and who have already a basic knowledge of Prolog might stick to this section only.

- **Animals Example [ISO]:** The Jekejeke Prolog programming language provides the usual logical operators. The given example shows a small expert system.

- **Primes Example [ISO]:** Arithmetic and lists are also part of the Jekejeke Prolog programming language. The given example computes the prime numbers up to a given upper bound by the Sieve of Eratosthenes.

- **Money Example [ISO]:** The Jekejeke Prolog programming language provides the usual backtracking. The given example shows the solving of a small letter puzzle.

- **Rabbits Example:** Jekejeke Prolog allows the grouping of predicates to modules. The given example shows mutual recursion among modules.

- **Parallel Example:** Jekejeke Prolog provides easy to use parallel generate and test. As an example we determine perfect numbers.

- **Pound Example:** Jekejeke Prolog allows object-oriented programming based on ISO modules, including inheritance of methods by sub classes.

2.1 Animals Example [ISO]

The Jekejeke Prolog programming language provides the usual logical operators. The given example shows a small expert system. The knowledge of the expert systems is represented as a Prolog text. The Prolog text will consists of facts about a particular animal and rules about animals in general. After reading the Prolog text by the Prolog system it will be possible to pose a query to the Prolog system and guess the type of an animal.

The Prolog text starts with the facts. To guess another animal these facts have to be changed and the Prolog text has to be re-read by the Prolog system. A Prolog fact consists of a predicate name followed by a number of arguments. For our example we will only make use of predicates with a single argument that is an atom, but in general predicate can have more arguments and their argument types can differ from an atom:

```
motion(walk).
Etc..
```

The Prolog text then continues with the rules. This order is not mandatory for a Prolog text since rules can have references to predicates either occurring before or after the rule itself. A Prolog rule makes use of the turnstile operator (:-)/2. This operator will separate the head of the rule from the body of the rule. The body itself will make use of the comma operator (,)/2. This operator will separate the goals in the body of the rule:

```
class(mamal) :- motion(walk), skin(fur).
Etc..

animal(rodent) :- class(mamal), diet(plant).
Etc..
```

The Prolog text is found in the support .zip under the file name ‘animals.p’. The Prolog text can be read-in for the first time or re-read a second time by the consult command. The consult command can be issued from the Prolog top-level by enclosing the file name in square brackets. If you don’t have your file name in the class path, you can fully qualify the file name and consult it this way. Otherwise it is enough to use the file name only:

```
?- ['animals.p'].
Yes
```

Upon successful consult the Prolog system will respond with a Yes. The Prolog system is now ready to respond to a query. We can now ask the Prolog system to guess the animal. The Prolog system will search through the Prolog rules and facts and try to find a variable binding so that the query is satisfied:

```
?- animal(X).
X = cat ;
No
```

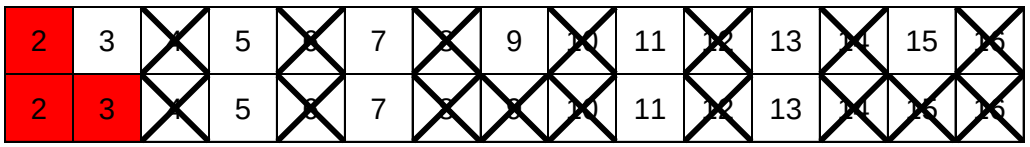
Typing the semicolon instruct the Prolog system searching for an additional solution, which would not be found in the present case.

2.2 Primes Example [ISO]

Arithmetic and lists are also part of the Jekejeke Prolog programming language. The given example computes the prime numbers up to a given upper bound by the Sieve of Eratosthenes. The graphical intuition behind it is as follows. The sieve starts with listing the integer numbers starting from 2 up to some upper bound m. Here is an example for m=16:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

During an iteration we pick the first uncrossed number and cross out the multiples of it. Each picked number will be a prime number. The process stops when no more prime numbers are left. Of course we can also already stop crossing out a little bit earlier, namely when we find a prime number p such that p*p>m:



From the above sieve we can read off that 2, 3, 5, 7, 11 and 13 are the only prime numbers between 2 and 16. We will now go on and implement this sieve in Prolog. We use Prolog lists to represent the sieve. The list will only contain the uncrossed numbers. The first predicate that we need is the predicate that builds the initial list:

```
integers(Low, High, [Low | Rest]) :-
      Low =< High, !,
      M is Low + 1,
      integers(M, High, Rest).
integers(_, _, []).
```

The first rule first checks whether we are still inside the lower and upper bound. If yes we return the lower bound as the first list element and recursively call the predicate with an increased lower bound. Otherwise we return an empty list. We can test it for m=16:

```
?- integers(2,16,X).
X = [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
```

As a next step we need a predicate that is able to cross out multiples of a given number. This predicate will input the sieve before crossing out multiples and the output the sieve after crossing out the sieve. To check multiplicity we use the arithmetic remainder function:

```
remove([], _, []).
remove([I | Is], P, Nis) :-
      I rem P =:= 0, !,
      remove(Is, P, Nis).
remove([I | Is], P, [I | Nis]) :-
      remove(Is, P, Nis).
```

The first rule states that nothing needs to be removed from the empty list. The second rule removes the first element of a list if it is a multiple of the given number and then continues recursively. Otherwise the third rule will keep the first element and continue recursively. We can test it also for m=16 and sift two times:

```
?- integers(2,16,X), remove(X,2,Y), remove(Y,3,Z).
X = [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16],
Y = [3, 5, 7, 9, 11, 13, 15],
Z = [5, 7, 11, 13]
```

We now need a predicate that does the iteration of the crossing out and the collection of the prime numbers, until we have reached the stopping condition. Again we will have input and output argument positions of the predicate. To check the stopping we use the arithmetic multiplication function:

```
sift([I | Is], High, [I | Is]) :-
    I * I > High, !.
sift([I | Is], High, [I | Ps]) :-
    remove(Is, I, New),
    sift(New, High, Ps).
```

The first rule expresses our stopping condition. When we have reached p*p>m we will return all the remaining numbers. The second rule will keep the first element, does one crossing out and continues recursively. We can test it for m=16:

```
?- integers(2,16,X), sift(X,16,Y).
X = [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16],
Y = [2, 3, 5, 7, 11, 13]
```

It is now simple to put the last pieces together. We can define a predicate primes which generate the initial list and then sift through it:

```
primes(High, R) :-
    integers(2, High, L),
    sift(L, High, R).
```

We are now done with our implementation and can invoke the predicate for bigger upper bounds than only m=16:

```
?- primes(16,X).
X = [2, 3, 5, 7, 11, 13]
?- primes(100,X).
X = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61,
67, 71, 73, 79, 83, 89, 97]
```

2.3 Money Example [ISO]

The Jekejeke Prolog programming language provides the usual backtracking. The given example shows the solving of a small letter puzzle. The problem consists of finding distinct digits for the letters below so that they fit the below addition:

```
  S E N D
+ M O R E
---------
M O N E Y
```

Many puzzles are formulated in a similar fashion, i.e. to find an exemplar from a collection that satisfies a certain condition. This leads to the programming pattern of generate and test. The pattern consists of defining goals G_1, .., G_n that enumerate the exemplars of the collection and of defining goals T_1, .., T_m that verify the condition. The problem is then solved by the following query:

```
G1, .., Gn, T1, .., Tm
```

The interpreter will move between these goals via backtracking until a solution is found. The interpreter can also be used to return multiple solutions. Since modern Prolog systems and modern machines do more than 2 mega logical inferences per second many small problems can be solved in time. We also apply this method to our letter puzzle.

So that we don't look too stupid we will enumerate intelligently. Simply enumerating all possible digit combinations would lead to 10^8 possibilities. What we will do, we will only enumerate permutations of the available digits which will lead to 8! possibilities. We do make use of the oneof/3 predicate which non-deterministically picks an element from a list. The generator for the permutations then reads as follows:

```
% assign(-List,+List)
assign([], _).
assign([X|Y], L) :- oneof(L, X, R), assign(Y, R).
```

The above predicate is then used to assign digits to the letters. More details about the generation phase can be found in the appendix. The test phase is now simply a code phrasing of the problem statement. We will look for a solution where M and S are non-zero. And we look for a solution where the addition holds. These conditions read as follows:

```
M =\= 0,
S =\= 0,
          1000*S + 100*E + 10*N + D +
          1000*M + 100*O + 10*R + E =:=
 10000*M + 1000*O + 100*N + 10*E + Y.
```

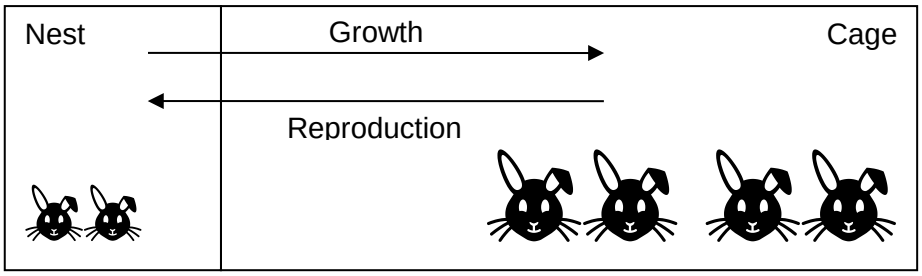
We can now run the Jekejeke Prolog interpreter to find the sole solution.

```
?- puzzle(X).
X = [9,5,6,7,1,0,8,2] ;
No
```

2.4 Rabbits Example

Jekejeke Prolog allows the grouping of predicates to modules. The given example shows mutual recursion among modules. We are going to work on a problem of mathematical biology. A certain man put a pair of rabbits in between walls. How many pairs of rabbits can be produced from that pair in a year if it is supposed that every month each pair begets a new pair which from the second month on becomes productive?

We will model the problem by two predicates adults/2 and babies/2. Each predicate has a first argument for mount count starting with zero and second argument for the corresponding size of the population counted in rabbit pairs. The dynamics of the two populations is depicted below. In each month the number of adults grows by the number of babies. On the other hand in each month there are new babies in the number of adults.



We will place the predicate adults/2 in a module named cage and the predicate babies/2 in a module named nest. In Jekejeke Prolog a module resides in a Prolog text. The Prolog text has to start with the module/2 directive. This directive states the name of the module and the exported predicates. The declaration reads as follows for the cage module:

```
:- module(cage, [adults/2]).
```

As a next step the Prolog text is allowed to import other modules. This is done with the use_module/1 directive. The directive has to mention the file name of the module that has to be imported. We will place the module cage in a file ‘cage.p’ and the module nest in a file ‘nest.p’. The directive for the import of the module nest into the module cage thus reads as follows. Importing a module makes the public predicates visible.

```
:- use_module('nest.p').
```

The first rule for the adults/2 predicate reads as follows. It states that for the first month there is one adult pair in the cage:

```
adults(0, 1) :- !.
```

Without visibility a module predicate has to be accessed by qualifying the module name via the colon (:)/2 operator. So for example to access the babies/2 predicate inside the module nest we would need to call nest:babies(M, Z). When the module nest has been imported it is enough to write babies(M, Z). We can take advantage of the visibility when formulating our second rule for the predicate adults/2. It states that the population of the adults grows by the number of babies:

```
adults(N, X) :-
   N > 0, M is N-1,
   adults(M, Y), babies(M, Z), X is Y+Z.
```

The Prolog text of the other module nest can be found in the appendix. As a next step we can import the module we want to work with into the top-level. This is also done by the use_module/1 directive. We will import the cage module, since we are interested in adult population. The Jekejeke Prolog interpreter will show on the console the time and size of the loading of the module. The Jekejeke Prolog interpreter will also show the loading of the directly or indirectly dependent modules:

```
?- use_module('cage.p').
% 'nest.p' consulted, 19 lines in 4 ms.
% 'cage.p' consulted, 18 lines in 8 ms.
Yes
```

We can now determine the adult population after a year:

```
?- adult(12, X).
X = 233
```

The example shows that the Jekejeke Prolog interpreter will not loop when loading recursive module imports. Technically the interpreter performs a traversal of a possibly cyclic import graph. The example also shows that no special forward declarations are necessary to executed predicates from recursive module imports. Technically the interpreter lazily binds predicates to atoms via small local caches at runtime.

The example can as well be run with the help of the graphical user interface of the Jekejeke Prolog interpreter on the Swing or on the Android platform. The “Load File …” respectively “Load …” menu item can be used to find the file for the use_module/1 directive. When a consult or ensure loaded is called from within a Prolog text the directive will resolve a relative file name based on the location of the current Prolog text. This explains how the module cage finds the module nest as long as they reside in the same directory.

2.5 Parallel Example

In this example, we demonstrate how Jekejeke Prolog can be used for parallel search. We will determine the perfect numbers between 1 and 10'000. Perfect numbers were already mentioned in Euclid's Elements and also considered by Euler. A perfect number is such that the proper divisors sum up the number itself. Here is how to decide 28:

```
1 || 28, 2 || 28, 4 || 28, 7 || 28, 14 || 28
1 + 2 + 4 + 7 + 14 = 28 ✓
```

A Prolog code to check for perfect numbers needs to be able to compute the sum of a set of numbers. Since Prolog does only have lists and not sets, we use a list without duplicates to represent a set. Summing a list can be coded as a simple recursive predicate sum_list/2 directly as follows:

```
sum_list([], 0).
sum_list([X|Y], R) :-
   sum_list(Y, H), R is X+H.

?- sum_list([1,2,4,7,14],X).
```

The above predicate could be rewritten into a predicate with an accumulator, which would then be tail recursive. To find the list of divisors we use the built-in predicates between/3 and findall/3. The largest proper divisor can only be the size of half of the number, so that we only need to search up to half of the number:

```
?- use_module(library(advanced/arith)).

?- findall(Z, (between(1,14,Z), 28 rem Z=:=0), L).
L = [1,2,4,7,14]
```

The idea is now a predicate perfect/2, which puts everything together. This predicate will first determine the half of the number, then collect the divisors and finally compute the sum and check whether it equals the given number. Our predicate should answer "yes" for 28 and for non-perfect numbers it should answer "no":

```
perfect(X) :- Y is X//2,
              findall(Z, (between(1,Y,Z), X rem Z=:=0), L),
              sum_list(L, X).

?- perfect(27).
No

?- perfect(28).
Yes
```

The first four perfect numbers were the only ones known to early Greek mathematics, the mathematician Nicomachus had noted 8128 as early as 100 AD. The above predicate is a brute force solution and does not use much number theory. It can already find the perfect numbers between 1 and 10'000 in a reasonable time.

```
?- between(1,10000,X), perfect(X).
X = 6 ;
X = 28 ;
X = 496 ;
X = 8128 ;
No

?- time((between(1,10000,X), perfect(X), fail; true)).
% Up 2,965 ms, GC 11 ms, Thread Cpu 2,938 ms
(Current 02/01/19 15:31:25)

?- time((between(1,20000,X), perfect(X), fail; true)).
% Up 11,818 ms, GC 94 ms, Thread Cpu 11,672 ms
(Current 02/01/19 15:43:35)
Yes
```

One way to speed up the search is to use more CPU power. We will now explain how Jekejeke Prolog can help in running a search on multiple CPUs. Usually setting up a parallel solution involves using the modules "thread", "pipe" and "group". The predicate balance/[1,2] from the module "distributed" abstracts away all these details.

The predicate balance/1 implements a non-deterministic distributed generate and test. The predicate balance/2 allows additionally specifying the number of desired threads. Using a modern microprocessor with a modern operating system, multiple threads will run truly parallel on multiple cores. The only bottleneck being the shared memory:

```
?- time((balance((between(1,10000,X), perfect(X))), fail; true)).
% Up 1,587 ms, GC 14 ms, Thread Cpu 0 ms (Current 02/01/19 15:40:45)
Yes

?- time((balance((between(1,20000,X), perfect(X))), fail; true)).
% Up 6,800 ms, GC 57 ms, Thread Cpu 0 ms (Current 02/01/19 15:45:34)
Yes
```

In the above query, we used the predicate balance/1 without specifying the number of desired threads. The predicate will then use the number of logical CPUs of the main board. When using the predicate balance/1 also some new overhead will be inducted by the setup and teardown of the threads, and the communication between threads.

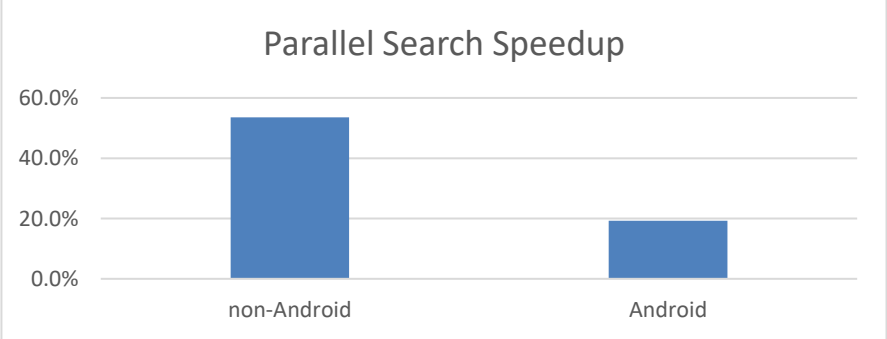


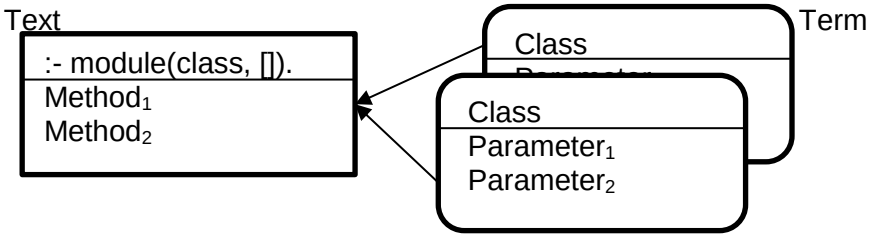
Figure: Measurements on a Lenovo Ideapad and on a Huawei Media Pad

The predicate balance/1 is both available on the non-Android and Android platform. On both platforms, the modules "thread" and "group" were implemented via the platform Java thread and thread groups. On the other hand, the module "pipe" uses some custom Queues implemented with the platform Java monitors.

2.6 Pound Example

In this example, we demonstrate object-oriented programming. Object-oriented programming has to be distinguished from object-oriented modelling. In object-oriented programming the classes of objects are directly represented as program text. We will make use of the ISO module standard and represent classes as Prolog module texts.

The anatomy of an instance object will be such that it will be simply a Prolog term. The functor of the Prolog term will indicate the class of the instance object and the arguments will be the actual parameters of the instance object. The methods of the class will be stored in a Prolog module text with the same name as the class of the object.



To invoke methods on instance objects Jekejeke Prolog then provides the operator (::)/2. This operator is bootstrapped from ordinary module operator (:)/2. It will first determine the class of the instance object, then add the instance object as a new first argument to the message and finally invoke the message in the determined class.

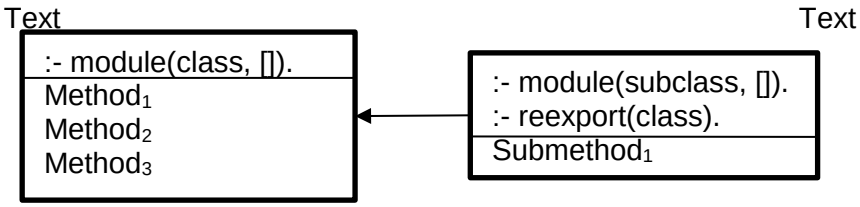
We now consider a dog pound. We assume that there is a class dog that determines the behaviour of dogs through its methods. Further, we assume that a dog instance has a name directly found in the first parameter. The method bark/1 is a dog command. The method barking/2 is used to retrieve the barking sound of a dog.

```
:- module(dog, [bark/1, barking/2]).

bark(Self) :-
   arg(1, Self, Name),
   Self::barking(Barking),
   write(Name), write(' says '),
   write(Barking), write('.'), nl.

barking(_, ruff).
```

Both methods are implemented so that they adhere to the convention of the Jekejeke Prolog operator (::)/2. Namely they have a first argument for the instance object itself. This convention was adopted from Python, which features the same manner of defining methods. We can now send the bark command to different dogs:

```
?- dog(susi)::bark.
susi says ruff.
Yes

?- dog(strolch)::bark.
strolch says ruff.
Yes
```

Before consulting the above Prolog text, make sure to have the path set via sys_add_path/1. Further place a package directive use_package/1 into the Prolog text and before top-level queries, so that short class names can be used. We now consider programming of sub classes. The ISO module standard reexport/1 directive serves as an inheritance indicator.



The Jekejeke Prolog (::)/2 is already polymorphic. The reexport/1 directive will make all the methods of the parent class of a subclass visible in the subclass itself. This allows for the additional benefit of code sharing in a parent class of a subclass if the code is common for multiple subclasses. We will implement a class basset with a different barking:

```
:- module(basset, [barking/2]).
:- reexport(dog).

:- override barking/2.
barking(_, woof).
```

In the above Prolog text, the directive override/1 is an extension by Jekejeke Prolog and is necessary to suppress the warning that a re-exported predicate is overridden. However, this exactly what we want to do in the present example, implement a different barking. We can send the bark command to a basset dog:

```
?- basset(lafayette)::bark.
lafayette says woof.
Yes
```

Classes defined as above retain all the advantages of the Jekejeke Prolog module system. They can be automatically reloaded by the make/0 command. Further, it is possible to debug classes as if they were ordinary Prolog texts. Since methods are Pythonesk predicates, it carries over to set spy points and break points without restrictions.

3 Prolog Conversations

The Jekejeke Prolog runtime library provides character terminal based interactions. Among the interactions we find query answering and source consulting.

- **Solution REPL:** We explain the Prolog query answer loop which consists of a query answering session between the Prolog system and the end-user.

- **Error Handling:** The Prolog query answer loop aborts the query upon syntax errors or unhandled execution errors.

- **Source Consulting:** Source consulting does not stop upon the first syntax or execution error.

- **Interrupt Handling:** The console allows manually interrupting the interpreter loop either during read or during execution.

- **Compatibility Matrix:** We compare our approach with the former DEC10 standard and the current ISO core standard.

3.1 Solution REPL

We explain the Prolog query answer loop, which consists of a query answering session between the Prolog system and the end-user. The interpreter announces that it can accept queries by the Prolog prompt (?-):

```
?-
```

The end-user can type in his query, terminate it by a period (.) and issue it with a carriage return. The syntax of a query is documented in the language reference manual in section 3.4. Let's turn to our first example query. The query will display the text "Hello World!" and new line in the console:

```
?- write('Hello World!'), nl.
```

When the interpreter receives a query it starts searching for solutions. Any output produced by the query during its execution is immediately displayed. When the query finally succeeds the interpreter displays the variable bindings. When there were no variables in the query the interpreter displays "Yes":

```
Hello World!
Yes
?-
```

The interpreter now checks whether the query left some choice points. When no choice points are left the interpreter immediately returns to the Prolog prompt (?-). Let's now turn to our second query. The second query will contain variables so that we will see how variable bindings will work:

```
?- X = 1; X = 2.
```

When the execution of the query succeeds for the first time the interpreter will show the variable bindings. Again the interpreter will check whether some choice points are left. Since in the case of this query some choice points are left, the interpreter will wait for further instructions from the end-user:

```
X = 1
```

The end-user can choose upon the following options:

```
EOF     = Exit the current session.
;       = Redo the query and possibly display more solutions.
        = Don't redo the query no more solutions will be displayed.
?       = Display this help text.
<Goal>. = Execute the <Goal> and prompt command again.
```

If the end-user chooses a goal, this goal will be executed and the end-user will be prompted again. The side effect of the goal can control the interpreter state or the goal can be used to query the interpreter state. Besides that the end-user can also use session predicates such as abort/0, break/0, etc.. or predicates he has defined on his own.

This is what happens when the end-user chooses the semicolon (;):

```
X = 1 ;
X = 2
```

The interpreter internally works with unnamed variables. Query variables are mapped to these unnamed variables. When displaying an answer the interpreter maps the unnamed variables back to the query variables. Anonymous variables which are an underscore only (“_”) are not shown to the end-user.

3.2 Error Handling

Syntactically incorrect queries are ignored by the Prolog query answer loop. The end-user might issue queries that are not syntactically correct. Here is an example:

```
?- X = .
```

The interpreter detects syntax errors and reports them to the end-user. The query text except for the initial comments is repeated and the error position is indicated. Also the syntax error message is displayed:

```
Error: Term missing.
X =
    ^
?-
```

Queries with syntax errors are not executed and the interpreter returns to its prompt. It can further happen that a query causes an unhandled execution error during its execution. The Prolog query answer loop aborts the query upon unhandled execution errors. Here is an example:

```
?- foobar.
```

When an unhandled execution error happens, the interpreter will stop searching and report it to the end-user. The execution error message and its context will be display. For affected goal originating in the query the contexts will simply consist of the predicate identification:

```
Error: Undefined predicate foobar / 0.
      foobar / 0
?-
```

When the affected goal originates from a source file a more elaborated context is displayed. In particular the context will also show the file name of the absolute source path together with a line number where the affected goal can be found. The context will also show the stack trace from the affected goal back to the top level:

```
?- t1.
Error: Undefined predicate foobar / 0.
      foobar / 0 in 'exception.p' at 5
      t3 / 0 in 'exception.p' at 3
      t2 / 0 in 'exception.p' at 1
      t1 / 0
?-
```

The stack frame elimination optimization will remove stack frames from the call chain. As a result the stack trace will be shortened. In the extreme only the affected goal together with the query answer loop goal will remain in the call chain:

```
?- s1.
Error: Undefined predicate foobar / 0.
      foobar / 0 in 'exception.p' at 12
      s1 / 0
```

```
?-
```

The Prolog text can handle execution errors via the constructs catch/3. When an execution error is fully handled, that is when the execution error is not re-thrown and no other execution error occurs, then the Prolog query answer loop will not display any execution error and the query execution will continue as defined by the handler. The Prolog text cannot handle system errors, since the construct catch/3 is not able to handle them.

System errors have a special handling. When a system error of type user abort percolates to the query answer loop the interpreter will return to the Prolog prompt (?-). In case of a system error of type user exit the interpreter will leave this query answer loop. For all other system errors the interpreter will leave all query answer loops. In all cases the display of the execution error message or its context is suppressed.

3.3 Source Consulting

Source consulting does not stop upon the first syntax or execution error. The end-user can instruct the interpreter to consult a Prolog file. This is easily done by issuing a query with the consult/1 predicate. The predicate receives a path. The clauses in the corresponding file are asserted and the directives are executed:

```
?- consult('exception.p').
```

When a syntax error or an execution error from a clause or directive is encountered the consult predicate does not abort. Syntax errors are displayed and the corresponding clause is not asserted respectively the corresponding directive is not executed:

```
Error: Term missing.
foobar :-
          ^
      'exception.p' at 1
```

The syntax error will be displayed together with its source location and the stack trace. Execution errors related to the assertion of clauses will also be shown with their source location and the stack trace:

```
Error: Can't modify system, built-in or static predicate write / 1.
      'exception.p' at 3
```

Not all problems related to the assertion of clauses are so fatal as to prevent the interpreter from adding the clause to the knowledge base. When the interpreter encounters such a minor problem it issues a warning instead of an error:

```
Warning: Singleton variable(s) [Y], use anonymous variable(s) (_).
      'exception.p' at 5
```

The special source name “user” is reserved for consulting from the console. The source name can be used without further specifications and it instructs the interpreter to read clauses and directives directly from the console window. The consult from console ends with an end-of-file (^D on Mac and Linux, ^Z on Windows):

```
?- consult(user).
member(X,[X|_]).
member(X,[_|Y]) :- member(X,Y).
:- member(X, "ABC"), write(X), nl, fail; true.
65
66
67

Yes
?-
```

The consult from the console does not abolish its previously consulted predicates. It can thus be used to add more and more predicates to the knowledge base. Predicates have to be manually reset by means of the system predicate abolish/1.

3.4 Interrupt Handling

The native console allows manually interrupting the interpreter loop either during read or during execution. The interrupt key (^C on Mac, Linux and Windows) will cause the invocation of an interrupt handler which will affect the interpreter.

For the runtime library we decided to assign the exit function to the interrupt key. When the interrupt key is pressed the interrupt handler will signal a system error of type user exit. When not otherwise trapped, this will cause an exit of the current session.

```
>
Jekejeke Prolog, Runtime Library 1.0.7
(c) 1985-2015, XLOG Technologies GmbH, Switzerland
?- repeat, fail.
^C
>
```

In the above example the interpreter hangs in an infinite computation from the repeat fail query. The interrupt key allows terminating this infinite computation.

3.5 Compatibility Matrix

We compare our approach with the former DEC10 standard and the current ISO core standard. The following compatibility issues persist for the interactions:

Table 1: Compatibility Matrix for Interactions

Nr	*Description*	*System*
1	Does not have syntax error description.	DEC10
2	Directives are not viewed as arbitrary goals.	ISO
3	The initialize directive allows arbitrary goals.	ISO
4	Consulting is not part of the standard.	ISO

4 Prolog Syntax

In this section we show what syntax the Jekejeke Prolog interpreter accepts and how the syntax is interpreted relates to the mathematical concepts. The syntax covers multiple levels consisting of tokens, terms and texts. The syntax is dynamically extensible by operator definitions. Each level can be characterized as follows:

- **Token Level:** The smallest unit above characters are tokens. The interpreter recognizes a number of different tokens.

- **Term Level:** The tokens of a line are then assembled into terms. The operator definitions control the assemblage of terms with infix or prefix operators.

- **Text Level:** On the text level we find the terms that are either used in a user session or that are consulted from a file.

- **Miscellaneous Definitions:** The interpreter keeps track of flags and properties.

4.1 Token Syntax

This syntax describes the grammar that forms tokens from character sequences. The syntax follows mainly the ISO core Prolog standard [6]. The Unicode extensions is based on our own research. We find the following topics:

- **Filler Syntax:** Fillers are for layout and comments.
- **String Syntax:** Strings use quotes to enclose characters.
- **Word Syntax:** Words are based on character classes.
- **Number Syntax:** Numbers are used to denote integers and floats.
- **Line Syntax:** Lines are sequence of tokens terminated by a period.
- **Unicode Extension:** Jekejeke Prolog supports the Unicode character set.
- **Compatibility Matrix:** ISO/DEC10 compatibility issues of this syntax.

Filler Syntax

In the following presentation we only assume an ASCII encoding. The token syntax here is thus defined based on the 7-bit characters from ASCII. There is a separate section that shows how the token syntax is extended to Unicode. The tokens can be interspersed with layout characters and comments. There are two kinds of comments, namely line comments and block comments. A line comment starts with "%" and ends with an end of line. A block comments starts with "/*" and continues until the closing "*/":

```
filler           --> { layout
                     | linecomment
                     | blockcomment }.
layout           --> "\0\" ... " ".
linecomment      --> "%" { char } eol.
eol              --> "\n".
blockcomment     --> "/*" { char } "*/".
```

Examples:

```
% hanoi: Solves the towers of hanoi problem    % is a line comment
/* hanoi:
     Solves the towers of hanoi problem */     % is a block comment
```

Besides the space character all ASCII control characters count as layout characters. The underlying streams allow the transmission of the null character (NUL) ('\0'). The null character does not indicate the end of a character stream or string. The encoding of a resource might use character sequences such as carriage return (CR) ('\r'), carriage return line feed (CRLF) ('\r\n') or line feed (LF) ('\n') to indicate the end of a line. These are made visible by a single line feed (LF) ('\n') through the character streams.

String Syntax

Strings are tokens that are started and ended by a quote character. Between the quote character practically arbitrary characters can be used. Line comments and block comments are not recognized inside strings. Among the layout characters only the space (“ “) can be part of a string. When we want to include the quote character in a string we simply repeat it twice. The single quote (‘), the double quote (“) and the back quote (`) can start and end a string:

```
str_single  --> "'" { "''" | "\\" esccont | " " | str_char } "'".
str_double  --> "\"" { "\"\"" | "\\" esccont | " " | str_char } "\"".
str_back    --> "`" { "``" | "\\" esccont | " " | str_char } "`".
str_char    --> char except layout.

esccont     --> control | meta | escape | eol.
control     --> "a" | "b" | "r" | "f" | "t" | "n" | "v".
meta        --> "\\" | "’" | "\"" | "`".
escape      --> oct_code | uni_code | hex_code.

oct_code    --> oct_digit { oct_digit } "\\".
oct_digit   --> "0" ... "7".
uni_code    --> "u" hex_digit hex_digit hex_digit hex_digit.
hex_code    --> "x" hex_digit { hex_digit } "\\".
hex_digit   --> digit | "A"  ... "F" | "a" ... "f".
```

Examples:

```
"Hello ""John""!"       % is a double quoted string
`Line 1\nLine 2`        % is a back quoted string
"very-long-\
code-list"              % is a double quoted string
'\xE54\'                % is a single quoted string
"\uD83D\uDE02"          % is a double quoted string
```

Strings can contain escape sequences that start with the backslash (\). After the backslash escape codes, control symbol or a meta-code can follow. The escapes codes allow octal codes, uni-codes and hexadecimal codes. Surrogate pairs are automatically combined into a single code point.

The octal code is simply a sequence of octal digits terminated by the backslash. The uni-codes start with the Unicode indicator ("u") and require exactly four hexadecimal digits. The hexadecimal code starts with a hexadecimal indicator (“x”) followed by a sequence of hexadecimal digits terminated by the backslash.

There are control symbols for the alert (‘\a’), the backspace (‘\b’), the carriage return (‘\r’), the form feed (‘\f’), the horizontal tab (‘\t’), the new line (‘\n’) and the vertical tab (‘\v’). Among the meta-codes we find the backslash (‘\\’), single quote (‘\’’), the double quote (‘”’) and the back quote (‘`’). When escaped they simply denote them.

Strings are not allowed to include an end of line. Instead the escape sequences for the line feed (‘\n’) should be used. A backslash followed by an end of line is used to continue a string on the next line. Strings are also not allowed to include layout characters. These have to be escaped as well.

Word Syntax

Words are tokens that are not enclosed by some characters. Character classes determine the border from one word to another. We have defined the classes of delimiters, alphabetical characters, decimal digits and graphic characters. Delimiters are characters that form a word by their own. Alphabetical characters and digits glue together to form words. Similarly graphic characters form a word when arranged in sequence:

```
word         --> delimiter
                 | alpha { alpha | digit }
                 | graphic { graphic }
                 | number
                 | reference.
delimiter    --> "(" | "{" | "[" | "]" | "}"
                 | ")" | "," | ";" | "!" | "|".

alpha        --> upperscore | lower.
upperscore   --> "_" | "A" ... "Z".
lower        --> "a" ... "z".
digit        --> "0" ... "9".
graphic      --> "\\" | "=" | "<" | ">" | "."
                 | ":" | "?" | "-" | "+"
                 | "*" | "/" | "#" | "@"
                 | "&" |"^" | "~" | "$".
```

Numbers and references share in common that they start with a digit. Their syntax is documented in more detail in the next section. Words are further classified into names, variables and the rest. The rest cannot be used as a name or a variable in a Prolog text. The accepted word syntax depends on the character set extension. By default Jekejeke Prolog provides a Unicode character set extension which is documented in a later section.

```
name         -->  delimiter except "(", "{", "[",
                    "]", "}", ")", ",", "|"
                 | lower { alpha | digit }
                 | graphic { graphic } except "."
                 | str_single.

variable     --> upperscore { alpha | digit }
                 | str_back.
```

Examples:

```
(                % A word, but not a name and not a variable.
!                % The name '!'.
:-               % The name ':-'.
foobar           % The name 'foobar'.
'arc sin'        % The name 'arc sin'.
X12              % The variable 'X12'.
`col 7`          % The variable 'col 7'.
```

Words that start with an upper case letter or with the underscore (“_”) are considered as variables. Back quoted strings also belong to the category of variables, they allow for arbitrary character sequences used as variables. Words that start with a lower case letter or a graphic character are considered as names. Finally single quoted strings also belong to the category of names, similarly they allow for arbitrary character sequences used as names.

Number Syntax

When the number word starts with a radix indicator then after the radix indicator an integer number in the corresponding radix is expected. There is the binary radix indicator (“0b”), the octal radix indicator (“0o”) and the hex radix indicator (“0x”). Currently only integers can be provided with a non-decimal radix. Floats and decimals do not currently have this feature.

```
number      --> integer | float | decimal.
reference   --> "0r" hex_digit { hex_digit }.
integer     --> "0b" bin_digit { bin_digit | "_" }
                | "0o" oct_digit { oct_digit | "_" }
                | "0x" hex_digit { hex_digit | "_" }
                | "0'" ( "'" | "\\" escape | " " | str_char )
                | mantissa.
float       --> mantissa fraction [ exponent ]
                | "0f" [ mantissa ] [ fraction ] [ exponent ].

decimal     --> "0d" [ mantissa ] [ fraction ] [ exponent ].
mantissa    --> digit { digit | "_" }.
fraction    --> "." digit { digit | "_" }.
exponent    --> ( "e" | "E" ) [ "-" | "+" ] digit { digit | "_" }.
bin_digit   --> "0" ... "1".
```

When the number word starts with a reference indicator (“0r”) then it is considered a reference. References can only be written to an output stream, but they cannot be read in from an input stream. If an attempt is made to read a reference from an input stream then a syntax error is thrown.

When the number word starts with a small float indicator (“0f”) then it is converted into a small float constant. The conversion might fail when the number word does not conform to the small float syntax or when the number is outside of the representation range.

Examples:

```
2009              % is a number and also an integer
0xFF              % is a number and also an integer.
0'a               % is a number and also an integer.
3.1415            % is a number and also a float
0d199.98          % is a number and also a decimal.
0rA276B3          % is a reference.
1e-12             % is not a number, float fraction missing.
```

When the number word starts with a decimal indicator (“0d”) then it is converted into a decimal constant. The conversion might fail when the number word does not conform to the decimal syntax or when the number is outside the representation range. The optional fraction and optional exponent influences the scale of the decimal number.

When the number word contains the period (“.”) then it is converted into a float constant, otherwise it is converted into an integer constant. The conversion might fail when the number word does not conform to the integer or float syntax or when the number is outside of the representation range.

Examples:

```
2_000_000               % is a number and also an integer
0xFFFF_FFFF             % is a number and also an integer.
3.14159_26535_89793     % is a number and also a float.
0b1__0                  % is not a number, two underscores in a row.
```

It is also allowed that the mantissa, the fraction or the exponent contain one or many underscores. This also holds if there is a radix, small float or decimal indicator. The syntax does not express the following additional constraints. It is not allowed that a number contains two underscores in a row. It is not allowed that a number contains an underscore at the end of the mantissa, the fraction or the exponent.

When the number word starts with a character indicator (“0’”) then it is decoded as a character constant. The value of a character constant is the integer character code of the character that immediately follows the character indicator. Character escaping as found in strings is also allowed after the character indicator.

Line Syntax

The detection of lines depends on the detection of tokens and fillers. A data line is a sequence of at least one token interspersed with fillers and terminated by a terminating period. A terminating period is a period that is not preceded by a graphic character and that is followed by a blank, a line comment or that is at the end of the text. Data lines are usually first detected before the term recognition process starts. This allows the interpreter to detect incomplete terms or superfluous tokens.

```
data      --> token { token } filler "." [ layout | linecomment ].
non-data  --> filler eof.
token     --> filler ( string | word ).
```

Examples:

```
abc.% ABC         % Is a data input.
./* .             % Is a data input.
/* good bye */    % Is a non-data input.
abc               % Is neither a data nor a non-data input.
.                 % Is neither a data nor a non-data input.
```

We might find a non-data line at the end of a Prolog text. Omitting the terminating period from a data line is considered an error. Similarly adding a terminating period to a non-data line is as well considered an error. Block comments are only detected when they are not preceded by a graphic character.

Unicode Extension

The ISO core standard allows extending the set of character codes beyond what is defined in its documentation. Jekejeke Prolog supports the Unicode character set as its processor extension. The particular Unicode version that is supported depends on the underlying Java virtual machine. The following definitions replace the corresponding definitions in the previous sections. We indicate a replaced non-terminal by appending a single quote (‘). The Unicode character types are denoted by all upper case.

What concerns fillers we did neither touch the line comments, the block comments nor the end of line character. But we extended the class of layout characters slightly. Including the FORMAT Unicode character type caters for covering the byte order mark (BOM) (‘\xFEFF\’). Further including the CONTROL Unicode character type caters for covering ASCII control characters. We have excluded the non-joiner (‘\x200C\’) and the joiner (‘\x200D\’) hints from layout, so that they can be later used in lower.

```
layout'     --> SPACE_SEPARATOR |
                LINE_SEPARATOR |
                PARAGRAPH_SEPARATOR |
                CONTROL |
                FORMAT except "\x200C\", "\x200D\".
```

Examples:

```
<BOM>:-          % The name ':-'.
```

The definition of strings has received a slight change. We introduced a new character class invalid. This character class includes the character types UNASSIGNED, PRIVATE_USE and SURROGATE. We do also consider the replacement character (‘\xFFFD\’) as an invalid Unicode character. This character usually indicates an invalid byte sequence which could not be converted back to a Unicode sequence during stream read.

```
invalid     --> UNASSIGNED |
                PRIVATE_USE |
                SURROGATE |
                "\xFFFD\".
```

Examples:

```
'\xFFFD\'         % The name '\xFFFD\', an invalid character.
'\xD800\'         % The name '\xD800\', a low surrogate.
```

When reading a term the string definition applies in its original form to the tokenization phase. During parsing strings undergo an additional validation step where strings with invalid characters are sorted out. If needed invalid characters can nevertheless be included in a string by using the backslash (\) to escape the code. Escaping can also be used to include single standing surrogates in strings.

With respect to the delimiters we added all Unicode punctuation character types that correspond to parenthesis or quotes. As a result these characters do not glue with other characters. Further important Prolog punctuations characters such as “,”, “;”, “;” and “|” are detected individually. The delimiter class also contains our invalid character class. An invalid characters delimiter is allowed during tokenization but sorted out during parsing.

```
delimiter'  --> START_PUNCTUATION |
```

```
                    END_PUNCTUATION |
                    INITIAL_QUOTE_PUNCTUATION |
                    FINAL_QUOTE_PUNCTUATION |
                    "," | ";" | "!" | "|" |
                    invalid.
```

Examples:

```
    «»              % The name '«' followed by the name '»'.
```

We will use the Unicode connecting punctuation character type to detect the underscore. The underscore detects the start of variables. We also added the Unicode uppercase character types and the Unicode title character types to the corresponding character class. This leaves fully intact the detection of ASCII variables. But it broadens what will be detected as Unicode variables. For example dashed low lines and certain digraphs now indicate also variables.

We filled the remaining class of lower letters with all remaining Unicode letter character types, all Unicode mark character types and all non-decimal digit number types. Among the mark character types we find for example the combining dieresis (UML) (‘\x308\’). Among the non-decimal digits number types we find roman numbers and fractions. We did not implement any composing or decomposing conversion of character sequences to other character sequences. As a result convertible character sequences are not recognized as identical.

```
    upperscore' --> UPPERCASE_LETTER |
                    TITLECASE_LETTER |
                    CONNECTOR_PUNCTUATION.
    lower'      --> LOWERCASE_LETTER |
                    MODIFIER_LETTER |
                    OTHER_LETTER |
                    NON_SPACING_MARK |
                    ENCLOSING_MARK |
                    COMBINING_SPACING_MARK |
                    LETTER_NUMBER |
                    OTHER_NUMBER |
                    "\x200C\" | "\x200D\".
```

Examples:

```
    ‿A              % The variable ‿A (Starts with dashed low line).
    Džep            % The variable Džep (Starts with digraph dzhe).
    Džep            % The variable Džep, different from first variable.
    a<UML>          % The name 'ä'.
    ä               % The name 'ä', different from first name.
    Ⅶ               % The name 'Ⅶ' (Roman seven).
    ⅓               % The name '⅓' (Fraction 1/3).
```

We added the non-decimal digit numbers to the lower letter class since we cannot offer some sensible number conversion for them. Therefore although they resemble numbers they will only be detected as names. When preceded by a decimal digit they need to be quoted. On the other hand the Unicode decimal digit number character type has a good support. We can use digits in various scripts and they are converted into numbers. To avoid the conversion one has to put the digits in quotes.

```
    digit'      --> DECIMAL_DIGIT_NUMBER.
```

Examples:

```
'2⅓'              % The name '2⅓'.
2⅓                % The number 2 and the name ⅓.
`٠`               % The name '٠' (Arabic zero).
٠                 % The number 0.
```

We have now covered almost all Prolog text character classes. What remains is the graphic character class. We assign to this character class all the Unicode character types that we did not yet assign. The dollar sign ($) is still covered by the Unicode currency symbol character type. Similarly it can be verified that all other ASCII graphic character classes are still covered. Also using Unicode character types broadens this character class. For example the euro currency symbol (€) is now also included as a graphic character.

Since graphic characters only glue with them selves and don't glue with delimiters, alphabetical characters and decimal digits, they can be still written directly adjacent to numbers and non-graphic names. Math symbols are now also part of the graphic character class, so care has to be taken when using these symbols adjacently with each other or in front of a period (.). When in doubt best is to use spaces between these symbols. Useful math symbols are for example the right arrow (→) ('\x2192\') or the bottom (⊥) ('\x22A5\').

```
graphic'    --> DASH_PUNCTUATION |
                OTHER_PUNCTUATION except ",", ";", "!", "'", "\"" |
                MATH_SYMBOL except "|" |
                CURRENCY_SYMBOL |
                MODIFIER_SYMBOL except "`" |
                OTHER_SYMBOL except "\xFFFD\".
```

Examples:

```
\=<>.:?-+*/#@&^~$ % The name '\=<>.:?-+*/#@&^~$'.
2€tax             % The number 2, the name '€' and the name 'tax'.
⊥→⊥.              % The name '⊥→⊥.'.
⊥ → ⊥             % The names '⊥', '→' and '⊥'.
```

When writing out Prolog terms spaces are automatically put around operators. Also names are quoted when necessary. Additionally the character codes inside quoted names are automatically escaped. The current rule is that all characters belonging to the layout character class except for the space (" ") itself are escaped. Escaping produces a control character, an octal code or a hex code. Hex coding is used for character codes above or equal 512.

Compatibility Matrix

The following compatibility issues persist on the token level:

Table 2: Compatibility Matrix for the Token Syntax

Nr	*Description*	*System*
1	Negative numbers are recognized as tokens.	DEC10 [1]
2	Integer numbers are bounded.	DEC10
3	Has (“%(“, “%)”) as synonym for the braces (“{“, “}”).	DEC10
4	Has (“,..”) as synonym for the vertical bar (“|”) in lists.	DEC10
5	Has (“|”) as synonym for semicolon (“;”) in goals.	DEC10
6	Floating point numbers are absent.	DEC10
7	Integer numbers can be given in a base between 2 and 9.	DEC10
8	Has no lower case option.	ISO [2]
9	Has mention of collation.	ISO
10	Does not have decimal syntax.	ISO
11	Does not have reference syntax.	ISO
12	Has prefix 0’ for an integer that is a character code.	ISO
13	Has char conversion.	ISO
14	Has additional characters in delimiter class.	UNID [4]
15	Has additional characters as quotes.	UNID

4.2 Term Syntax

This syntax describes the grammar that forms terms from token sequences. The syntax follows mainly the ISO core Prolog standard [6]. We also incorporate some syntax extensions as known from SWI-Prolog-7 [10]. We find the following topics:

- **Simple Syntax:** At the beginning there are variables and atomics.
- **Compound Syntax:** Compounds consist of a functor and arguments.
- **List Syntax:** Lists are a short hand for special compounds and constants.
- **Expression Syntax:** Expressions are a short hand for operator based compounds.
- **Special Syntax:** Convenience for array index and dictionary terms.
- **Compatibility Matrix:** ISO/DEC10 compatibility issues of this syntax.

Simple Syntax

At the very beginning of terms we have the unstructured variables and atomic elements. The variables have already been defined as a kind of words. Atomic elements are now atoms, numbers, negative numbers and references. The names, the empty sets and the empty lists are now found among the atoms. Among the structured terms we will have compounds, lists and expressions:

```
term(N)     --> variable
              | atomic
              | compound
              | list
              | expression(M) { M<=N }.

atomic      --> atom
              | number
              | "-" number
              | reference.

atom        --> name
              | "{" "}"
              | "[" "]".
```

Examples:

```
X                % is a variable and thus a term
foobar           % is a name and thus a term
3.1415           % is a number and thus a term
-3               % is a negative number and thus a term
- 3              % is an expression, corresponds to -(3)
```

The grammatical production rules for terms can be viewed as stratified into levels. This is used later for parsing operator expressions. The levels range from 0 to 1200. Reading a term usually starts with level 1200. To disambiguate from operator expressions a negative number is only recognized when the minus (“-“) directly precedes the first digit of the number.

Compound Syntax

The simplest form of structuring terms is the compound. The compound consists of a name followed by one or many arguments. The principal name is called the functor and the number of arguments is called the arity. To disambiguate from operator expressions a compound is only recognized when the name directly precedes the left parenthesis (“(“).

```
compound   --> "{" term(1200) "}"
             | atom "(" arguments ")"
             | atom "(" ")".

arguments  --> term(999) { "," term(999) }.
```

Examples:

```
f(a,b)            % is a compound
{}                % is the empty set, corresponds to '{}'
{a}               % is a compound, corresponds to '{}'(a)
-(a, b)           % is a compound, corresponds to a - b
- (a, b)          % is an expression, corresponds to - ,(a, b)
```

By means of the braces (“{}”) a form of set notation can be invoked. The empty set notation is just a short hand for the name ‘{}’. The set notation that surrounds a term is a compound of arity one with the functor ‘{}’. Since the empty set {} and the empty list [] do belong to the category of atoms they can be used as a functor as well.

List Syntax

A further form of structuring terms is the list. A list is a short hand for multiple compounds of arity two with the functor ‘.’. A closed list will end in the name ‘[]’. The vertical bar is used to denote open lists ending in the given term.

```
list        --> "[" arguments [ "|" term(999) ] "]"
              | string2.
```

Examples:

```
[]                % is the empty list, corresponds to '[]'
[a, b]            % is a list, corresponds to .(a, .(b, []))
[X | Y]           % is a list, corresponds to .(X, Y)
"ABC"             % is a list, corresponds to [65, 66, 67]
```

The double quoted string is a short hand for a list of character codes. The notation is especially useful in connection with definite clause grammars where it then directly denotes a sequence of character terminals.

Instead of a double quoted string also other strings can be used as a short hand, there are Prolog flags that control the default behaviour and there are Prolog options that control the behaviour of the read and write predicates.

Expression Syntax

The expressions that can be formed depend on the operators and their levels that have been defined as follows. The system can distinguish prefix, infix and postfix operators. The arguments of an operator have to be on a lower or equal level than the operator itself so that parsing is successful. The exact operator type further determines the acceptability and associativity of the operator.

```
expression(M) --> "(" term(1200) ")" { M=0 }
                | "(" ")"
                | op1(M,fx) term(M-1)
                | op1(M,fy) term(M)
                | term(M-1) op2(M,xfx) term(M-1)
                | term(M-1) op2(M,xfy) term(M)
                | term(M) op2(M,yfx) term(M-1)
                | term(M-1) op2(M,xf)
                | term(M) op2(M,yf).

op1(M,I)      --> atom(O) { current_op(M,I,O) }.

op2(M,I)      --> operator(O) { current_op(M,I,O) }.
```

Names, either quoted or unquoted, can be used as operators. Since the empty set {} and the empty list [] do belong to the category of atoms they can be used as an operator as well. Additionally the comma (“,”), the vertical bar (“|”) and the period (“.”) can also be used as an infix or postfix operator without quoting them. Whether a certain atom or operator can be used further depends on the built-in op/3 which might refuse certain definitions.

Examples:

```
- 5               % is a prefix expression, corresponds to -(5)
5 + 6             % is an infix expression, corresponds to +(5, 6)
5 days            % is a postfix expression, corresponds to days(5)
assertz((p :- q)) % is a compound
mode(+, -)        % is a compound
```

The parenthesis (“()”) serves to circumvent the level rule of the operators. Arguments from compounds and lists have the level 999. To use higher level operators as arguments, we therefore have to put them in parenthesis. The operator detection itself can be escaped when the operator directly precedes a stop character.

Special Syntax

We provide some syntax extensions in the spirit of SWI-Prolog 7 [10]. To allow a richer syntax we additionally allow closed lists, singleton sets and unit atoms as operators, thus providing additional parameters when forming a term. The resulting special syntax provides convenience for array subscripts, tagged structures and zero arguments.

```
operator      --> atom
                | "[" arguments "]"
                | "{" term(1200) "}"
                | "(" ")"
                | "|" | "," | ".".
```

A postfix list X_0 [X_1, .., X_n] expression is parsed as an array index term sys_index(X_0, X_1, .., X_n) when the operator sys_index is defined as postfix. The postfix empty set X_0 {} and singleton set X_0 {X_1} expressions are parsed as dictionary terms sys_struct(X_0) respectively sys_struct(X_0, X_1) when the operator sys_struct is defined as postfix.

Examples:

```
A[I, J]           % is an array index term
point{x:1, y:2}   % is an dictionary term
dist()            % is a zero argument term
```

A further special syntax allows zero argument terms. For this purpose, the compound syntax has been extended to accept an empty argument list. Further, the expression syntax has as well be accepted to accept a unit atom. An empty argument list is not mapped to a zero argument compound but to a unit atom postfix expression.

When writing out a term canonically operator definitions are ignored an expressions are written as compound. During non-canonical write, expressions are written back as expressions. Further parenthesis are put where the mix of operator level demands it. Finally, escaped operators are put in parenthesis when necessary.

Compatibility

The following compatibility issues persist on the term syntax level:

Table 3: Compatibility Matrix for the Term Syntax

Nr	*Description*	*System*
1	Has div operator (div).	DEC10
2	Modulo (mod) has different level.	DEC10
3	Has a module context operator (@).	ISO
4	Defines level of expression inside parenthesis as 1201.	ISO
5	Special list operator syntax comes as [] block operator.	SWI7
6	The block operator [] uses a different term building.	SWI7
7	Special set operator syntax comes as {} block operator.	SWI7
8	The block operator {} uses a different term building.	SWI7
9	Without space to the {} it's a native dictionary.	SWI7
10	The zero argument () uses a different term building .	SWI7

4.3 Text Syntax

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

- **Consult Syntax:** Consulted Prolog texts consist of directives and clauses.
- **Session Syntax:** The end-user interacts with the Prolog interpreter via a session.
- **Compatibility Matrix:** ISO/DEC10 compatibility issues of this syntax.

Consult Syntax

A theory text is a set of lines usually read in from a file. Each line is first converted by the term syntax and then interpreted according to the text syntax. The main purpose of a theory text is to provide an ordered collection of premises. Among the premises we find facts and rules. Rules consist of a head and of a body, separated by the implication operator (":-"). Facts consist only of a head. Facts can be considered to be rules where the body corresponds to then atom "true", which can then be omitted.

```
theory      --> { sentence "." }.
sentence    --> part [ "/\" sentence ] | "unit".
part        --> clause | directive.

clause      --> head ":-" body | head.
head        --> callable.

directive   --> ":-" body.
body        --> goal [ "," body ] | "true".
goal        --> callable | variable.

callable    --> atom | compound.
```

Examples:

```
in(X,[X|_]).                % is a fact.
in(X,[_|Y]) :- in(X,Y).     % is a rule.
:- op(800, xfx, in).        % is a directive.
```

Besides facts and rules a theory text can also include directives. Directives are detected by a degenerated implication operator (":-") which is used prefix position. Whenever a directive is encountered in a theory text, the interpreter is invoked to solve the goal against the current state of the interpreter. Solutions to directives are not printed and directives are only solved once. The most typical directive is the operator definition directive. But the application of directives is not limited to operator definitions.

To avoid common typing errors a number of sanity checks are performed on facts and rules. Among these checks we find the singleton variable check. This check assures that anonymous variables ("_") are used for variables that occur only once in a fact or rule. We also find the discontinued predicate check and the multi-file predicate check. These checks assure that in the normal case all the facts and rules for a predicate are defined in one row and that they only come from one source.

The head of a rule or fact needs to be a callable. That is it has to be either an atom or a compound. It is not possible to have rules or facts for numbers or variables. Internally the Prolog interpreter only supports natively conjunctions (,)/2 in the body. All other constructs in the body have to refer to predicates defined in the knowledge base. This includes logical predicates such as (,)/2, (;)/2, (->)/2 and (\+)/1 which are predefined defined by the system. If a predicate cannot be found an existence exception is thrown during execution.

It is possible to combine multiple clauses and directives into a single input line by means of the (/\)/2 operator or to indicate no clause and no directive by the atom "unit". This is useful for returning multiple clauses and directives by term expansion. The operator (/\)/2 has been chosen in remembrance of hypothetical reasoning but it works slightly different. Clauses and directives that are joined via (/\)/2 during consult give not raise to inter-clausal variables. The individual clauses are still automatically universally quantified.

Session Syntax

A session text is a set of lines usually interactively read in from the end-user. The interpreter prompts each line by the query operator (“?-“). Each line is first converted by the term syntax and then interpreted according to the session syntax. The main purpose of a session text is to answer queries against a logic program.

```
session     --> { query "." }.
query       --> "?-" body.
```

Examples:

```
?- X in [1,2,3].              % is a query.
?- X^(X in [[1]], Y in X).    % is a query.
```

The interpreter will display the bindings of the query variable names upon success of a goal. If there are no interesting query variable name bindings the interpreter will display “Yes”. The interpreter will display “No” upon failure of a goal. When the interpreter detects choice points it will request the end-user for redo. The end-user can do so by entering “;” and hitting return. The end-user can also terminate the search by directly hitting return.

Upon failure of a goal, if there are no choice points or if the end-user terminates the search, the interpreter returns to its prompt. The interpreter also returns to its prompt when the goal throws an exception. Before returning to the prompt the interpreter will first display the message and the stack trace of the exception. By entering queries that consult theory texts or update database predicates, the knowledgebase will be modified and thus further invocations of queries will return different solutions.

Compatibility

The following compatibility issues persist on the line level:

Table 4: Compatibility Matrix for the Text Syntax

Nr	*Description*	*System*
1	Has no singleton variable check.	DEC10
2	Has no discontinued predicate check.	DEC10
3	Has no multi file predicate check.	DEC10

4.4 Miscellaneous Definitions

The interpreter also needs to keep track of flags and properties definitions. The following flags and properties are provided by the Jekejeke Prolog runtime library:

- **Prolog Flags:** The predefined Prolog flags.
- **Predicate Properties:** The predefined predicate properties.
- **Source Properties:** The predefined source properties.
- **Operator Properties:** The predefined operator 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 Jekejeke Prolog runtime library:

sys_attached_to:	See the user session section.
sys_belongs_to:	See the user session section.
sys_break_level:	See the user session section.
sys_choice_point:	See the optimization techniques section
sys_body_variable:	See the optimization techniques section
sys_stack_frame:	See the optimization techniques section
sys_head_variable:	See the optimization techniques section.
sys_clause_index:	See the clause indexing section
sys_last_pred:	See the prolog texts section.
sys_timeout:	See the prolog texts section.
verbose:	See the prolog texts section.
sys_body_convert:	See the body conversion section.
sys_clause_expand:	See the meta predicates section.
bounded:	See the arithmetic domains section.
integer_rounding_function:	See the rounding operations section.
max_arity:	See the building and unification section.
max_code:	See the string predicates section.
base_url:	See the path resolution section.
sys_act_status:	See the capability plug-ins 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 Jekejeke Prolog runtime library:

sys_body:	See the body conversion section.
sys_rule:	See the body conversion section.
sys_notrace:	See the predicate definitions section.
sys_nobarrier:	See the logical predicates section.
built_in:	See the predicate definitions section.
static:	See the predicate definitions section.
full_name:	See the predicate definitions section.
sys_usage:	See the prolog texts section.
multifile:	See the prolog texts section.
sys_multifile:	See the prolog texts section.
discontiguous:	See the prolog texts section.
visible:	See the module system section.
sys_public:	See the module system section.
override:	See the module system section.
meta_predicate:	See the meta predicates section.
sys_meta_predicate:	See the meta predicates section.
meta_function:	See the meta predicates section.
sys_meta_function:	See the meta predicates section.
dynamic:	See the dynamic database section.
sys_dynamic:	See the dynamic database section.
thread_local:	See the dynamic database section.
sys_thread_local:	See the dynamic database section.
group_local:	See the dynamic database section.
sys_group_local:	See the dynamic database section.
virtual:	See the special predicates section.
sys_arithmetic:	See the special predicates 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 Jekejeke Prolog runtime library:

sys_notrace:	See the predicate definitions section.
sys_capability:	See the source files section.
last_modified:	See the source files section.
version_tag:	See the source files section.
expiration:	See the source files section.
date:	See the source files section.
max_age:	See the source files section.
short_name:	See the source files section.
sys_source_preload:	See the module system section.
sys_source_visible:	See the module system section.
sys_source_name:	See the module system section.
sys_timing:	See the module system section.
sys_link:	See the module system section.
package:	See the module system section.
use_package:	See the module system section.

Operator Properties

Operator properties can be accessed via the system predicates oper_property/2, set_oper_property/2 and reset_oper_property/2. The following predicate properties are supported by the Jekejeke Prolog runtime library:

sys_usage:	See the prolog texts section.
visible:	See the module system section.
override:	See the module system section.
sys_alias:	See the syntax_operators section.
sys_portray:	See the syntax_operators section.

5 Runtime Packages

The Jekejeke Prolog programming language comes with a standard set of predefined predicates. Predicates can be grouped into theories and we present them as such:

- **Kernel Package:** The modules that need to be predefined by the interpreter itself.
- **Runtime Package:** The modules that are usually defined for the interpreter.
- **Arithmetic Package:** The modules concerned with the numbers of the interpreter.
- **Structure Package:** The modules concerned with the term model of the interpreter.
- **Reflect Package:** The modules concerned with accessing the interpreter.
- **Bootload Package:** The modules concerned with extending the interpreter.

5.1 Kernel Package

This theory groups the minimal set of predicates that need to be predefined by the interpreter itself.

- **Special Predicates:** The interface of special built-ins is currently not published.
- **Body Conversion:** Naked goals in bodies and rules are automatically wrapped.
- **Control Predicates:** The core predicates of the interpreter.
- **Optimization Techniques:** Optimization techniques speed up predicate execution.
- **Clause Indexing:** Clause indexing also speeds up predicate execution.
- **Module Statistics:** This module provides some execution statistics.
- **Compatibility Matrix:** ISO/DEC10 compatibility issues of the kernel theory.

Special Predicates

It is possible to register predicates and evaluable functions as special built-ins via the built-in special/3. A predicate indicator, the service class and the service number has to be specified. While registering the Prolog interpreter will automatically create an instance of the service class with the given service number parameter.

Syntax:

```
directive --> "special(" indicator "," module "," integer ")".
```

Example:

```
:- special(foo/1, 'FooAPI', 7).   % is a special predicate directive.
```

An evaluable function f/n is identified by the predicate property sys_arithmetic/0 and a predicate indicator f/n+1. To ease the end-user the Prolog system automatically implements for each predicate bridging to an evaluable function, and for each evaluable function tunnelling to a predicate.

Bridging, predicate as evaluable function:

```
X is p(Y1,..,Yn) :-
   Z1 is Y1, .., Zn is Yn, p(Z1, .., Zn, X), !.
_ is p(_,..,_):-
   throw(error(evaluation_error(partial_function),_)).
```

Tunnelling, evaluable function as predicate:

```
f(Y1, .., Yn, X) :-
   X is f(Y1,..,Yn).
```

During bridging the arguments are evaluated and then the corresponding predicate is called with an additional last argument for the result. If the corresponding predicate succeeds its choice points are removed. If the predicate fails or if the result is not a value, then an error is issued.

The bridging is further controlled by the virtual property of a predicate. If a predicate has this property the first argument Y_1 gets special treatment. During bridging this argument will not be evaluated. This is useful for predicates that pass as the first argument the receiver object. The directive virtual/1 can be used to set the virtual property of a predicate.

The following special predicate predicates are provided:

special(I, C, K):
: Succeeds with registering the predicate indicator I as a special builtin that calls an instance of the service class C with function index K.

virtual P, ..:
: The predicate sets the predicate P to virtual.

The following predicate properties for special predicates are provided:

virtual:
The property indicates that the predicate has the virtual property. The property can be missing. The property can be modified for clause defined predicates. The property cannot be modified for special or foreign predicates.

sys_arithmetic:
The service class implements the predicate via the internal evaluable functions API. The property can be missing. The property cannot be modified.

Body Conversion

Body conversion converts a goal of the form X where X is a variable, into a goal of the form call(X). And goals which are not of the form X where X is a callable are rejected by this conversion. Body conversion is in effect when clauses are asserted, either dynamically or statically. The effect can be seen by the following example. In the static rule for the predicate p/0 the variable X will be wrapped via a call/1:

Example:

```
?- [user].
p :- q(X), X.
^D
?- listing.
p :-
    q(X),
    call(X).
```

Body conversion is also in effect when goals are executed, either directly or deferred via meta-arguments. The body conversion can be switched off via the flag sys_body_convert. The flag only affects the body conversion for the Prolog session queries, for the Prolog text clauses and for the Prolog text directives. The dynamic clause assertions and the deferred meta-arguments are not affected by the flag, these places will still do body conversion.

The body conversion is table driven. The meta-predicate declarations and determine how arguments are traversed. The predicate properties sys_body/0 and sys_rule/0 will indicate that the meta-predicates should be traversed during body conversion respectively rule conversion. To facilitate the declaration the predicate sys_neutral_predicate/1 allows defining dictionary entries that are not yet completely defined.

The following body conversion predicates are provided:

:- A:
: The predicate cannot be executed and exists only to configure the body conversion table.

A :- B:
: The predicate cannot be executed and exists only to configure the body conversion table.

A, B: [ISO 7.8.5]
: The predicate succeeds whenever A and B succeed. Both goal arguments A and B are cut transparent.

call(A): [ÍSO 7.8.3]
: The predicate succeeds whenever A succeeds. The goal argument A is converted before calling.

The following predicate properties for body conversion are provided:

sys_body:
: The property indicates that the meta-predicate is traversed by body conversion. The property can be changed for user predicates.

sys_rule:
: The property indicates that the meta-predicate is traversed by rule conversion. The property can be changed for user predicates.

The following Prolog flags for body conversion are provided:

sys_body_conversion:
: The legal values are on and off. The flag indicates whether the body conversion is enabled for the knowledge base. The default value is on. The value can be changed.

Control Predicates

The backtracking control flow can be modified by the cut (!)/0. The cut will remove the choice points from the head to the cut, including a head choice point. Common programming patterns involving the cut are provided in the forms of the predicate once/1 and (\+)/1. Both predicates will remove any backtracking from the goal argument.

The interpreter has the capability to interrupt its normal flow by exception handling. An interruption happens when an exception is thrown or when a signal is raised. An exception can be an arbitrary Prolog term. Some exception terms are recognized by the interpreter so as to display a user-friendly stack trace. In particular we recognize:

error(Message, Context):
: The exception is an error.

warning(Message, Context):
: The exception is a warning.

cause(Primary, Secondary):
: The exception is a composite of a primary exception and a secondary exception.

The predicate throw/1 can be used to throw an exception. If the context is a variable the predicate will automatically instantiate the variable with the current stack trace. The predicate catch/1 can be used to catch a thrown exception. The predicate will not catch reserved exceptions. Currently system errors are the only reserved exceptions.

The following control predicates are provided:

fail: [ISO 7.8.2]
false: [TC2 8.15.5]
: The predicate fails.

true: [ISO 7.8.1]
otherwise:
: The predicate succeeds once.

!: [ISO 7.8.4]
: The predicate removes pending choice points between the first non-cut-transparent parent goal invocation and this goal and then succeeds once.

once(A): [ISO 8.15.2]
: The predicate succeeds once if A succeeds. Otherwise, the predicate fails.

\+ A: [ISO 8.15.1]
: When A succeeds, then the predicate fails. Otherwise, the predicate succeeds.

throw(E): [ISO 7.8.9]
: The predicate fills the stack trace if necessary and then raises the exception E.

sys_raise(E):
: The predicate raises the exception E.

catch(A, E, B): [ISO 7.8.9]
: The predicate succeeds whenever A succeeds. When an exception is thrown during the execution of A, this exception is non-reserved and this exception unifies with E then the predicate succeeds whenever B succeeds. Otherwise, the exception is re-thrown.

sys_trap(A, E, B):
: The predicate succeeds whenever A succeeds. When an exception is thrown during the execution of A and this exception unifies with E then the predicate succeeds whenever B succeeds. Otherwise, the exception is re-thrown.

The following predicate properties for control predicates are provided:

sys_notrace:
The property indicates that the predicate does not appear in an exception context. The property can be changed for user predicates.

The following source properties for control predicates are provided:

sys_notrace:
The property indicates that a call from the given definition scope does not appear in an exception context. The property can be changed for user sources.

Optimization Techniques

When a defined predicate is invoked the interpreter tries to find a matching clause and then invokes the body of the clause. Normally the interpreter creates a choice point. The choice point is used during backtracking to search for further matching clauses. Choice point elimination optimization refers to the ability of the interpreter to eliminate choice points when it seems clear that no further clauses will match.

From the ISO standard it is indicated to implement the cut via its own choice-point. So that upon back-tracking the cut choice-point will remove all previous choice-points in the scope of the current clause and then fail. Choice-point elimination also refers to the early elimination of choice-points by the cut. The cut then does then not need to create a choice-point on its own and it can deterministically succeed.

The omission of choice points is also possible for system and custom built-ins. A system built-in has full control of the interpreter environment and can voluntarily eliminate choice points. Inside the Prolog API there exists a protocol so that custom built-ins in the form of Java foreign predicates can also omit choice points. The Java method has simply not to set the retry flag upon succeeding and the invocation choice point will be discontinued.

Finally we have implemented optimization techniques that are based on an appropriate generation of an intermediate form for each clause. These techniques can attack the parent stack frame, the body or the head:

- **Stack frame elimination:** The conditions for the application of this optimization are as follows. The call has to be made from the last goal of a clause. And the call has to lead into a matching clause and it has been detected that no further clauses will match. The call chain will then be shortened.

- **Body variable elimination:** This optimization technique is capable of eliminating body variables. This is done by special reference count decrementing instructions inside the intermediate form. Also place holder creation is delayed into the body.

- **Head variable elimination:** This optimization technique can eliminate head variables. This is done by special unification instructions that can combine goal arguments directly. Also unifications are delayed into the body.

The following Prolog flags for the optimization techniques are provided:

sys_choice_point:
: Legal values are on and off. The flag indicates whether choice point elimination is enabled for the knowledge base. Default value is on. Value can be changed.

sys_body_variable:
: Legal values are on and off. The flag is per knowledge base and is inherited when predicates are created. Default value is on. The value can be changed.

sys_stack_frame:
: Legal values are on and off. The flag is per knowledge base and is inherited when predicates are created. Default value is on. The value can be changed.

sys_head_variable:
: Legal values are on and off. The flag is per knowledge base and is inherited when predicates are created. Default value is on. The value can be changed.

The following Predicate properties for the optimization techniques are provided:

sys_nobody:
: The property indicates that the clauses of the predicate should not have body variable elimination optimization in place. The value can be changed.

sys_nostack:
: The property indicates that the clauses of the predicate should not have stack frame elimination optimization in place. The value can be changed.

sys_nohead:
: The property indicates that the clauses of the predicate should not have head variable elimination optimization in place. The value can be changed.

Clause Indexing

The Jekejeke Prolog system is capable of dynamically creating indexes for multiple argument positions and that might span multiple arguments. Naïvely scanning the full clause set of a defined predicate would be inefficient. Therefore when a defined predicates is invoked an analysis of the arguments of the calling goal is performed:

1. **First Argument Indexing:** When the defined predicate is called with a non-variable first argument then the functor of this argument is used in a hash table lookup to find a smaller set of clauses for scanning.

2. **Non-First Argument Indexing:** If the first argument is a variable then the calling goal is searched for another argument which is non-variable and the functor of this argument is then picked for a hash table lookup.

3. **Multi-Argument Indexing:** The hash table does not only point to clause sets but also to further hash tables so that recursively after a found non-variable argument further non-variable arguments can be searched in the calling goal.

When clauses are consulted or asserted no indexes are maintained by default. Only after a defined predicate has been first invoked and the arguments have been analysed indexes are created. The index that is created is thus dependent on the call history of the predicate. Dynamic clause indexing is both available for static, dynamic and thread local predicates.

In general the resulting indexing structure need not be the same for different hash entries. When clauses are retracted and a clause set reaches zero size, then the corresponding hash entry is removed. The hash table automatically resizes to smaller sizes if necessary. When the hash table points to a trivial clause set of size one then search continues for an alternative indexing argument to further restrict the clause set.

When building the index we also consider constraints in the body of a clause. What we can currently detect are constraints of the form var(X) where X is a variable. If an argument of an indexed clause is a variable it takes a different route into the index depending whether it is guarded by such a constraint or not. When calling the predicate non variable arguments can take the non-guarded route.

The YAP Prolog system [7, section 6] does a similar dynamic clause indexing. But additionally the system considers constraints such as bindings in the body of each clause. Currently we do not incorporate bindings into our index. The Ciao Prolog system [8, section 88] allows declaring multi-argument indexes. In the declaration one can specify the key type. It is possible to index argument either on the functor or on the full term. Currently we do not support indexes based on full terms.

The following Prolog flags for the clause indexing are provided:

sys_clause_index:
: Legal values are on and off. The flag indicates whether clause indexing is enabled for the knowledge base. Default value is on. Value can be changed.

Module Statistics

This module provides some execution statistics. The predicate statistics/2 returns some key figures of the memory management and the runtime system, whereas the predicate statistics/0 displays the key figures on the standard output. The measurement of the time performance of a goal is facilitated by the predicate time/1.

Example:

```
?- statistics.
Max Memory             512,753,664 Bytes
Used Memory             68,568,872 Bytes
Free Memory            444,184,792 Bytes
Uptime                       5,293 Millis
GC Time                         12 Millis
Thread Cpu Time              1,000 Millis
Current Time            02/13/18 15:20:08
```

Since Jekejeke Prolog is a multi-threaded interpreter, we also provide statistics for the threads in JVM known to the interpreter. The predicate thread_statistics/3 returns some key figures concerning the given thread, whereas the predicate threads/0 displays the key figures for all known threads on the standard output.

Example:

```
?- threads.
Id      Alive Clauses
     21 Yes              0
     25 Yes      1,256,856
```

The JVM will notify the Prolog interpreter about low memory. On the Android platform this is a suicide notice, since foreground heap compaction is not yet available and it is therefore recommended to exit the Prolog interpreter. On both the Swing and the Android platforms the thread statistics are then used to decide which thread will be aborted.

The following stats predicates are provided:

statistics:

The predicate displays the current statistics key value pairs.

statistics(K, V):

The predicate succeeds for the values V of the keys K. The following keys are returned by the predicate:

max:	The maximum memory in bytes.
used:	The currently used memory in bytes.
free:	The currently allocated but unused memory in bytes.
uptime:	The time since start-up in milliseconds.
gctime:	The time spent for garbage collection in milliseconds.
time:	The time spent by this Jekejeke Prolog thread.
wall:	The current time in milliseconds.

time(A):

The predicate succeeds whenever the goal A succeeds. The predicate will measure the time for the execution of the goal A irrespective of whether the goal A succeeds or fails. Redoing the goal A is measured when the goal A has left some choice points.

thread_statistics(T, K, V):

The predicate succeeds for the values V of the keys K concerning the thread T. The following keys are returned by the predicate.

sys_thread_local_clauses:	The number of thread local clauses.

threads:

The predicate displays the current threads statistics key value pairs.

Compatibility Matrix

The following compatibility issues persist for the kernel theory:

Table 5: Compatibility Matrix for the Kernel Theory

Nr	*Description*	*System*
1	Does not have predicate once.	DEC10
2	Has predicate unknown to control undefined predicates.	DEC10
3	Does not have predicates catch and throw.	DEC10
4	Has public directive and incore predicate.	DEC10
5	Has interpreter and compiled code for same predicate	DEC10
6	Has mode declarations.	DEC10
7	Cannot compile (->)/2 predicate.	DEC10
8	Says folding of (;)/2 with cut not work, no sys_nobarrier.	DEC10
9	Says cut has no declarative reading, outdated view.	DEC10
10	No subtlety of sys_call/1 in ->/2 and \+/1 found.	DEC10
11	Leaves last call optimization open.	ISO
12	Leaves multi-argument indexing open.	ISO
13	Leaves memory management open.	ISO

5.2 Runtime Package

This theory groups the minimal set of predicates that are usually defined for the interpreter:

- **Meta Directives:** Functors can be marked as taking goal and clause arguments.
- **Qualified Names:** Predicates and evaluable functions can be qualified.
- **Logical Predicates:** Some logical extensions of the interpreter.
- **Work Distribution:** Some meta-predicates to distribute work.
- **Dynamic Databases:** Knowledge bases can be dynamically manipulated.
- **User Session:** Predicates can be interactively executed by submitting queries.
- **Compatibility Matrix:** ISO/DEC10 compatibility issues of the consult theory.

Meta Directives

The meta-predicate declaration takes as an argument a predicate name and a number of meta-argument specifiers. Analogously the meta-function directive takes as an argument a function name and a number of meta-argument specifiers. We can describe the arguments via the following grammar rules:

```
meta_directive    --> "meta_predicate" meta_signature
                    | "meta_function" meta_signature.

meta_signature    --> module ":" meta_signature
                    | name [ "(" meta_specifier
                               { "," meta_specifier } ")" ].

meta_specifier    --> integer
                    | "?"
                    | "::(" meta_specifier2 ")"
                    | "#(" meta_specifier3 ")"

meta_specifier2   --> integer
                    | "::(" meta_specifier2 ")".

meta_specifier3   --> integer.
```

Example:

```
:- meta_predicate count(0,?).
```

A positive integer n indicates a goal and a negative integer n indicates a clause. If the integer n, respectively –n-1 if n<0, is different from zero then the argument is a goal closure respectively clause closure. The question mark (?) indicates that the argument is neither a goal nor a clause. The (::)/1 wrapper indicates that the argument is also an object message.

It should be noted that Jekejeke Prolog executes meta-predicates even when meta-predicate directives are not present. The meta-predicate directives are needed for pretty printing and term expansion. Closures are not supported during term expansion. Term expansion for clauses can be switched on-off by the sys_clause_expand flag, and it is on by default.

The hash tag (#) indicates that the argument is an evaluable expression. Space, newline and indent insertion will be suppressed during pretty printing. Evaluable functions don't need special meta-predicate declarations. Meta-predicate declarations are assumed so that the whole evaluable expression is cross-referenced.

The following meta-predicate predicates are provided:

meta_predicate M, …:
The predicate sets the corresponding functor to meta-predicate declaration M.

meta_function M, …:
The predicate sets the corresponding functor to meta-function declaration M.

The following predicate properties for meta-predicates are provided:

meta_predicate M:
The property indicates that the functor has meta-predicate declaration M. The property is single valued.

sys_meta_predicate(S):
The property indicates that the predicate has been declared meta-predicate in source context S. The property is multi valued and can be missing. The property can be changed.

meta_function M:
The property indicates that the functor has meta-function declaration M. The property is single valued.

sys_meta_function(S):
The property indicates that the predicate has been declared meta-function in source context S. The property is multi valued and can be missing.

The following Prolog flags for meta-predicates are provided:

sys_clause_expand:
The legal values are on and off. The flag indicates whether the clause conversion is enabled for the knowledge base. The default value is off. The value can be changed.

Qualified Names

For qualified names a notation based on the colon (:) operator can be used when invoking predicates or evaluable functions. The module name itself can be structured by means of the slash (/)/2 operator and the set ({})/1 operator. This gives rise to a new primitive goal syntax which reads as follows:

```
goal         --> module ":" goal
               | receiver "::" goal
               | callable.

receiver     --> package "/" callable
               | reference
               | callable.
```

Under the hood qualified names are flattened to atoms with the help of an inline atom cache. Further the colon notation will also resolve module names based on the class loader of the call-site, the prefix list of the call-site and the prefix list of the system. A qualified predicate will be also searched in the re-export chain of the given module name.

Examples:

```
?- basic/lists:member(X, [1]).
X = 1
?- 'jekpro.frequent.basic.lists\bmember'(X, [1]).
X = 1
```

Finally there is also a double colon notation based on the (::)/2 operator that can be used to send message to a receiver. The receiver itself is prepended Python style to the callable before invoking it. For auto loaded Java classes the re-export chain contains the super class and implemented interfaces. If an unqualified predicate with the same name is defined, then this fall-back is called.

Examples:

```
?- 'System':err(X), X::println('abc').
X = 0r47733fca
?- current error(X), X::write('abc'), X::nl.
abc
X = 0r398aef8b
```

The predicates sys_callable/1, sys_var/1, sys_functor/3 and sys_univ/2 are the adaptations of callable/1, var/1, functor/3 and (=..)/2 in that these predicates respect the module colon (:)/2 and receiver double colon (::)/2 notation. A qualified functor may only contains the colon (:)/2 notation. The predicate sys_get_module/2 can be used to retrieve the class reference or module name of a receiver.

The following qualified names predicates are provided:

M:C:
: The predicate calls the callable C by qualifying the predicate name of C by the module name M. The call is performed in the same call-site as the colon notation.

R::C:
: The predicate calls the callable C by qualifying the predicate name of C by the module name of R and prepending R itself. The call is performed in the same call-site as the colon notation.

sys_callable(T):

Check whether T is a fully qualified callable.

sys_var(T):

Check whether T is a half qualified or not a qualified callable.

sys_functor(T, F, A):

The predicate unifies F with the possibly quantified functor of T and unifies A with the arity of T.

sys_univ(T, [F|L]):

The predicate unifies F with the possibly qualified functor of T and unifies L with the arguments of T.

sys_get_module(O, M):

The predicate succeeds in M with the class reference or module name of O.

sys_replace_site(Q, S, P):

The predicate succeeds for a new callable B which is a clone of the callable A with all the site properties of the callable Q.

The following qualified names evaluable functions are provided:

M:E:

The function evaluates the expression E by qualifying the function name of E by the module name M. The evaluation is performed in the same call-site as the colon notation.

R::E:

The function evaluates the expression E by qualifying the function name of E by the module name of R and prepending R itself. The evaluation is performed in the same call-site as the colon notation.

Logical Predicates

When a goal belonging to a defined predicate is invoked a new frame is created. When the interpreter encounters a cut (!) it will remove the choice points inside the frame. To instruct the interpreter that cuts can nevertheless propagate a defined predicate can be marked as cut transparent via the predicate property sys_nobarrier/1.

Examples:

```
?- X = a; X = b.
X = a ;
X = b
?- X = a, !; X = b.
X = a
```

Some of the predefined logical predicates are cut transparent in all arguments. This includes the predicates (,)/2 and (;)/2. Others are only cut transparent in a few arguments. This includes the predicates (->)/2 and (*->)/2, and also the special forms in connection with the predicate (;)/2. Others, such as call/1, once/1 and (\+)/1, are not cut transparent at all.

The following logical predicates are provided:

A; B: [ISO 7.8.6]
: The predicate succeeds whenever A or B succeeds. Both goal arguments A and B are cut transparent.

A -> B: [ISO 7.8.7]
: The predicate succeeds when A succeeds and then whenever B succeeds. Only the goal argument B is cut transparent.

A *-> B:
: The predicate succeeds whenever A and B succeed. Only the goal argument B is cut transparent.

A -> B; C: [ISO 7.8.8]
: The predicate succeeds when A succeeds and then whenever B succeeds, or else whenever C succeeds. Only the goal arguments B and C are cut transparent.

A *-> B; C:
: The predicate succeeds whenever A and B succeed, or else if B didn't succeed whenever C succeeds. Only the goal arguments B and C are cut transparent.

repeat: [ISO 8.15.3]
: The predicate succeeds repeatedly.

The following predicate properties for logical predicates are provided:

sys_nobarrier:
: The property indicates that the predicate is defined and cut transparent. The property can be changed for defined user predicates.

Work Distribution

This module provides meta-predicates to distribute work over multiple threads. The simplest meta-predicate horde/[1,2] collects the results of the spawned threads and leaves the work distribution to the spawned threads itself. The other predicates distribute work items among the spawned threads, but do this only on a local scale.

Example:

```
?- balance(between(1,10,X), Y is X*X).
X = 1,
Y = 1 ;
X = 3,
Y = 9 ;
Etc..
```

The meta-predicates balance/[2,3] and setup_balance/[3,4] allow work distribution of a generate and test. The meta-predicates might change the order of the result set. If the meta-predicates are cancelled by a cut such as in a surrounding \+/1 or once/1 they will automatically cancel each spawned thread.

The meta-predicates balance/[2,3] and setup_balance/[3,4] assume a side effect free interaction between the generate and test. The only channels are the variables in the intersection of the generate and test and the instantiations are copied. Currently the copying doesn't support attribute variables.

The setup in the meta-predicates setup_balance/[3,4] is executed once per spawned thread. The setup is executed before the test and attribute variables can communicate between the setup and the test. The setup can for example be used to build a CLP(FD) model and the model will be available in the test for labelling.

The following work distribution predicates are provided:

horde(T):
horde(T, N):
: The predicate succeeds whenever T succeeds. The predicate spawns threads over the available processors running copies of T. The binary predicate allows specifying the number N of requested threads.

balance(G, T):
balance(G, T, N):
: The predicate succeeds whenever G, T succeeds. The predicate distributes the work generated by G over the available processors running copies of T. The ternary predicate allows specifying the number N of requested threads.

setup_balance(S, G, T):
setup_balance(S, G, T, N):
: The predicate succeeds whenever S, G, T succeeds. The predicate distributes the work generated by G over the available processors running copies of S, T. The ternary predicate allows specifying the number N of requested threads.

submit(C, N):
: The predicate succeeds in running a copy of the goal C in a new thread and unifies N with its new name.

cancel(N):
: The predicate succeeds in stopping the thread with the name N.

Dynamic Database

Static, dynamic and thread local predicates can be consulted. Dynamic and thread local predicates on the other hand can be also accessed and modified by the predicates here in. The predicate clause/2 matches clauses, the predicate retract/1 matches and removes clauses and the predicates asserta/1 and assertz/1 add a clause.

Examples:

```
?- assertz(p(a)).
Yes
?- p(X).
X = a
?- clause(p(X),Y).
X = a,
Y = true
```

The predicates clause/2, retract/1 and retractall/1 can only match clauses that are visible from the head predicate that is used in the search. The predicates asserta/1 and assertz/1 cannot redefine a predicate. Instead it must be marked multi-file. The clauses of a dynamic predicate are seen by all threads. A thread local predicate on the other hand has its own set of clauses for each thread.

Examples:

```
?- abolish(foo/1).
Yes
?- abolish(infix(=>)).
Yes
```

The predicate abolish/1 allows a predicate turning it non-existent again. The same predicate can be also used to remove operators by using indicators prefix/1, postfix/1 and infix/1. The predicate will attempt to remove own user clauses of the predicate, and only remove it if no system or foreign user clauses remain. If still some of the aforementioned clauses remain the predicate stays existent and non-empty.

The following dynamic database predicates are provided:

dynamic P, …: [ISO 7.4.2.1]
: The predicate sets the predicate P to dynamic.

thread_local P, …:
: The predicate sets the predicate P to thread local.

group_local P, …:
: The predicate sets the predicate P to group local.

clause(H, B): [ISO 8.8.1]
: The predicate succeeds with the user clauses that match H :- B. The head predicate must be dynamic or thread local.

retract(C): [ISO 8.9.3]
: The predicate succeeds and removes with the user clauses that match C. The head predicate must be dynamic or thread local.

retractall(H): [Corr.2 8.9.5]
: The predicate succeeds and removes the user clauses that match the head H. The head predicate must be dynamic or thread local.

asserta(C): [ISO 8.9.1]
The predicate inserts the clause C at the top. The head predicate must be dynamic or thread local.

assertz(C): [ISO 8.9.2]
The predicate inserts the clause C at the bottom. The head predicate must be dynamic or thread local.

abolish(P): [ISO 8.9.4]
The predicate removes the predicate, evaluable function or syntax operator.

The following predicate properties for dynamic databases are provided:

dynamic:
The property indicates that the defined predicate is shared dynamic. The property cannot be set.

sys_dynamic(S):
The property indicates that the predicate has been marked dynamic in source context S. The property is multi valued and can be missing.

thread_local:
The property indicates that the clauses are local to the interpreter. The property cannot be set.

sys_thread_local(S):
The property indicates that the predicate has been marked thread local in source context S. The property is multi valued and can be missing.

group_local:
The property indicates that the clauses are local to the knowledge base. The property cannot be set.

sys_group_local(S):
The property indicates that the predicate has been marked group local in source context S. The property is multi valued and can be missing.

User Session

The query answer loop of the Prolog interpreter repeatedly prompts a query and answers it by showing the variable bindings. The query answer loop can be entered recursively by the predicate break/0. The query answer loop can be terminated by issuing an end of file. The query answer loop runs in its own input/output stream pair.

The system predicates abort/1, exit/1 and close/1 throw some well-known system errors. The system predicate exit/1 allows terminating the query answer loop similarly like issuing an end of file. The system predicate abort/1 only terminates the current query but continues the loop. The system predicate close/1 recursively terminates all query answering loops.

The system predicate version/0 displays a version banner. Top-level answers are displayed with the operator (=)/2. For custom forms delivered by a printable hook the operator (is)/2 is displayed. For custom constraints delivered by an equation hook the corresponding operator is displayed. For printable and equation hooks see the module residue.

The following user session predicates are provided:

prolog:
break:
The predicate prompts and answers queries until an end of file is encountered.
abort:
The predicate throws a system error of type user abort.
exit:
The predicate throws a system error of type user exit.
close:
The predicate throws a system error of type user close.

The following Prolog flags for user sessions are provided:

sys_attached_to:
The legal value is a graphic interface component per interpreter. The type depends on the current platform and toolkit. The value can be changed.
sys_belongs_to:
The legal value is a graphic interface component per knowledge base. The type depends on the current platform and toolkit. The value can be changed.
sys_break_level:
The legal value is an integer. The value indicates the number of currently entered breaks. The value cannot be changed.

Compatibility Matrix

The following compatibility issues persist for the runtime theory:

Table 6: Compatibility Matrix for the Runtime Theory

Nr	*Description*	*System*
1	Does not have the predicate dynamic.	DEC10
2	Has minus sign for reconsult in files list consult.	DEC10
3	Has save and restore predicate for the interpreter state.	DEC10
4	Has logging of a session.	DEC10
5	Has current_atom and current_functor predicate.	DEC10
6	The abolish predicate has two arguments (name, arity).	DEC10
7	Has reinitialise predicate.	DEC10
8	Uses the predicate ‘C’ for terminals.	DEC10
9	Uses clause expansion for grammar rules.	DEC10
10	Mentions a predicate expand_term/2.	DEC10
11	Does not have discontinuous directive.	DEC10
12	Does not have multi-file directive	DEC10
13	The predicate consult is left undefined.	ISO
14	The concept of a session is left undefined.	ISO
15	Database predicates do not have options.	ISO
16	Leaves clause indexing open.	ISO
17	Disallows redefining the ‘,’ operator.	ISO
18	Simultaneous infix and postfix operator issues error.	ISO
19	Not defined that abolish/1 could remove built-ins.	ISO
20	Does not have a static directive.	ISO
21	Does not have a thread local predicates.	ISO
22	Does not have absolute_file_name//[2,3] predicates.	ISO

5.3 Arithmetic Package

This theory is concerned with the integers and floats of the Prolog interpreter. We find the following topics:

- **Arithmetic Domains:** The used number representation.
- **Elementary Operations:** The available elementary operations.
- **Rounding Operations:** The available rounding operations.
- **Bitwise Operations:** The available bitwise operations.
- **Trigonometric Operations:** The available trigonometric operations.
- **Arithmetic Comparisons:** The available number comparisons.
- **Compatibility Matrix:** ISO/DEC10 compatibility issues of the arithmetic theory.

Arithmetic Domains

The Jekejeke Prolog interpreter accepts numbers and internally represents them as integers, floats or decimals. The implementation represents floats and decimals as pairs of a mantissa and a scale. For floats the scale is binary, for decimals the scale is decimal:

float = mantissa * 2^{scale}
decimal = mantissa * 10^{scale}

Examples:

```
1.0 = 100.0E-2
0d0.01 = 0d1E-2
0d123.4 = 0d1.234E2
0d1.0 \= 0d100E-2
```

If an integral value is between -2^31 and 2^31-1 then the Java Integer is used instead of the Java BigInteger. Similarly if a decimal value has scale 0 and a mantissa between -2^63 and 2^63-1 then the Java Long is used instead of the Java BigDecimal. The decimal scale is restricted to 31 bits.

The float mantissa is bounded and approximated to the precision. The small respectively large float binary scale is restricted to 8 bits respectively 11 bits. A negative zero is mapped to a positive zero. A NaN or infinity, irrespective of its sign, is considered outside of the domain and leads to an error.

Example:

```
?- [user].
factorial(0, 1).
factorial(X, Y) :- X>0, H is X-1, factorial(H, J), Y is X*J.
^D
?- X is factorial(4).
X = 24
```

The predicate is/2 can be used to evaluate a term consisting of evaluable functions and number constants. Thanks to bridging an evaluable function can also be defined by ordinary Prolog clauses for the corresponding predicate. The evaluable function will only deliver the first result of the corresponding predicate.

The following built-in predicate is provided for arithmetic domains:

X is Y: [ISO 8.6.1]
The predicate succeeds when X unifies with the evaluation of Y.

X [Y_1, .., Y_n, Z]:
The predicate succeeds when Z unifies with the element of term with the subscripts Y_1, .., Y_n for $1 \leq n \leq 7$. The term X need not be homogenously shaped and the indexes start with one.

The following Prolog flags for arithmetic domains are provided:

bounded: [ISO 7.11.1.1]
Legal values are true [ISO] and false [ISO]. The flag indicates whether the integer domain is bounded. Default value is false. Currently the value cannot be changed.

Elementary Operations

When the arguments of the binary elementary operations do not have the same types then widening is applied to them before performing the operation. Widening is done towards the bigger domain of the two arguments. Widening from integer to float32 or float might fail with an exception, since the unbounded integers have a greater range than float32 or floats.

The ordering of the domains is as follows:

integer < float32 < float < decimal

The binary decimal operations return the preferred scale as defined in the java class BigDecimal. They thus differ from the usual float32 or float operations in that they work with unlimited precision. There is no division for decimals defined here, the division will convert to float and perform a float division.

The signature of the available binary and unary elementary operations is listed here:

+, -, *, ^:	integer x integer -> integer
/:	number x number -> float
+, -, *:	float32 x float32 -> float32
+, -, *:	float x float -> float
+, -, *:	decimal x decimal -> decimal
-, +, abs, sign:	integer -> integer
-, +, abs, sign:	float32 -> float32
-, +, abs, sign:	float -> float
-, +, abs, sign:	decimal -> decimal

Examples:

```
abs(-1)                 --> 1
abs(-1.0)               --> 1.0
abs(-0d1.00)            --> 0d1.00
9 + 1                   --> 10
0.99 + 0.01             --> 1.0
0d0.990 + 0d0.01        --> 0d1.000
5 * 2                   --> 10
5.0 * 2.0               --> 10.0
0d5.0 * 0d2.0           --> 0d10.00
5 / 2                   --> 2.5
5.0 / 2.0               --> 2.5
0d5.00 / 2              --> 2.5
3 ^ 27                  --> 7625597484987
```

The unary float32 respective float conversion is approximate for integer and decimal arguments and returns always float32 respective float. The unary decimal conversion is exact for integer, float32 and float arguments and returns always decimals.

The signature of the available unary conversion operations is listed here:

float32:	number -> float32
float:	number -> float
decimal:	number -> decimal

Examples:

```
decimal(0.1)            --> 0d0.1000000000 0000000555 1115123125
                                7827021181 5834045410 15625
```

Thanks to tunnelling an evaluable function can also be invoked by calling the corresponding predicate. When invoking the predicate the arguments are not evaluated, only type checked. The result of the evaluable function is returned in the last argument of the predicate.

Examples:

```
?- abs(-1, X).
X = 1
?- abs(- 1, X).
Error: Argument should be a number, found - 1.
      abs/2
```

The following arithmetic operations are recognized in evaluations:

- X: [ISO 9.1.7]
If X is a number then returns the sign inversion of X.
+ X: [TC2 9.1.3]
If X is a number then returns X unchanged.
abs(X): [ISO 9.1.7]
If X is a number then returns the absolute value of X.
sign(X): [ISO 9.1.4]
If X is a number then returns the sign of X.
float(X): [ISO 9.17]
If X is a number then returns the conversion of X to a float.
decimal(X):
If X is a number then returns the conversion of X to a decimal.
float32(X):
If X is a number then returns the conversion of X to a float32.
X + Y: [ISO 9.1.7]
If X and Y are both numbers then the function returns the addition of X and Y.
X - Y: [ISO 9.1.7]
If X and Y are both numbers then the function returns the subtraction of X by Y.
X * Y: [ISO 9.1.7]
If X and Y are both numbers then the function returns the multiplication of X and Y.
X / Y: [ISO 9.1.7]
If X and Y are both numbers then the function returns the division of X and Y.
X ^ Y: [TC2 9.3.10]
If X and Y are both integers then the function returns X raised to the power of Y.

Rounding Operations

We supply a number of binary and unary operations that deal with rounding. We find different rounding modes and we also find different forms of division. The result type of the evaluable function integer/1 is always an integer. On the other hand the result type of the evaluable functions truncate/1, floor/1, ceiling/1 and round/1 is the same as the argument type:

integer, truncate:	towards zero
floor:	towards lower infinity
ceiling:	towards upper infinity
round:	towards nearest neighbour or then away from zero

Examples:

```
floor(-3)         --> -3
floor(-3.14)      --> -4
floor(-0d3.1415)  --> -4
```

The division is based on a hypothetical $/_F$ operation. This operation is approximate for float arguments and exact for integer and decimal arguments. The result type of the evaluable functions (//)/2 and (div)/2 is always an integer. On the other hand the result type of the evaluable functions (rem)/2 and (mod)/2 is the same as the argument types:

X // Y = integer(X /F Y).
X div Y = integer(floor(X /F Y)).
X rem Y = X – (X // Y) * Y.
X mod Y = X – (X div Y) * Y.

Examples:

```
5 // 2            --> 2
5.0 // 2.0        --> 2
0d5.00 // 2       --> 2
(-5) // 2         --> -2
(-5) div 2        --> -3
5 rem 2           --> 1
5.0 rem 2.0       --> 1.0
0d5.00 rem 2      --> 0d1.00
(-5) rem 2        --> -1
(-5) mod 2        --> 1
```

If the arguments of the binary operations have different types the same widening as already defined for the basic operations is applied. This means the widening is done towards the bigger domain of the two arguments.

The following rounding operations are recognized in evaluations:

integer(X):
If X is a number the returns the integer of X.
truncate(X): [ISO 9.1.7]
If X is a number then returns the rounding of X towards zero.
floor(X): [ISO 9.1.7]
If X is a number then returns the rounding of X towards negative infinity.
ceiling(X): [ISO 9.1.7]
If X is a number then returns the rounding of X towards positive infinity.
round(X): [ISO 9.1.7]
If X is a number then returns the rounding of X towards the nearest integer. If the absolute fraction of the number is 0.5 then returns the rounding away from zero.

X // Y: [ISO 9.1.7]
If X and Y are both numbers then the function returns the truncated division of X by Y.
X rem Y: [ISO 9.1.7]
If X and Y are both numbers then the function returns the remainder of X by Y.
X div Y: [TC2 9.1.3]
If X and Y are both numbers then the function returns the floored division of X by Y.
X mod Y: [ISO 9.1.7]
If X and Y are both numbers then the function returns the modulus of X by Y.

The following Prolog flags for rounding operations are provided:

integer_rounding_function: [ISO 7.11.1.4]
Legal values are down [ISO] and toward_zero [ISO]. The flag indicates how the (//)/2 and rem/2 are performed. Default value is toward_zero. The value cannot be changed.

Bitwise Operations

We provide bitwise operations defined on integers. They are not defined for floats or decimals. The bitwise operations work for negative and positive integers including zero. They also work for unbounded integers. The bit pattern that corresponds to an integer can be viewed as infinitely extending to the left, for positive integers and zero by 0's and for negative integers by 1's. The bitwise operations then work on the corresponding binary digits:

Examples:

```
                -11   =       …10101
                  2   =         …010
(-11) \/ 2  -->   -9  =       …10111
(-11) /\ 2  -->    0  =           …0
(-11) >> 2  -->   -3  =         …101
(-11) << 2  -->  -44  =     …1010100
\ (-11)     -->   10  =       …01010
```

The displacements n in the shift operations are restricted to the range -2147483648 ≤ n ≤ 2147483647. When the displacement is outside this range an exception is thrown. A negative displacement shifts in the opposite direction.

The following bitwise operations are recognized in evaluations:

\ X: [ISO 9.4.5]
: If X is an integer then returns the bitwise complement of X.

X /\ Y: [ISO 9.4.3]
: If X and Y are both integers then the function returns the bitwise X and Y.

X \/Y: [ISO 9.4.4]
: If X and Y are both integers then the function returns the bitwise X or Y.

xor(X, Y): [TC2 9.4.6]
: If X and Y are both integers then the function returns the bitwise exclusive X or Y.

X << Y: [ISO 9.4.2]
: If X and Y are both integers then the function returns X shifted by Y places left.

X >> Y: [ISO 9.4.1]
: If X and Y are both integers then the function returns X shifted by Y places right.

Trigonometric Operations

The trigonometric operations are defined for integers, floats and decimals. If the argument is an integer it is widened to a float before computing the operation. On the other hand if the argument is a decimal it is narrowed to a float before computing the operation. Widening from decimal to float might fail with an exception, since the unbounded decimals have a greater range than floats.

If the argument of a trigonometric operation is outside of its domain then an undefined evaluation error is thrown. Further when the mathematical result exceeds the range of the float then a float overflow evaluation error is thrown. The throwing of an exception is preferred over returning a float with the meaning of not a number (NaN), negative infinite (-Inf) or positive infinite (+Inf).

The following trigonometric operations are recognized in evaluations:

sin(X): [ISO 9.3.2]
Returns the float representation of the sine of X, argument must be in radians.

cos(X): [ISO 9.3.3]
Returns the float representation of the cosine of X, argument must be in radians

tan(X): [TC2 9.3.14]
Returns the float representation of the tangent of X, argument must be in radians

asin(X): [TC2 9.3.11]
Returns the float representation of the arcus sine of X, the result is in radians.

acos(X): [TC2 9.3.12]
Returns the float representation of the arcus cosine of X, the result is in radians.

atan(X): [ISO 9.3.4]
Returns the float representation of the arcus tangent of X, the result is in radians.

X ** Y: [ISO 9.3.1]
Returns the float representation of X raised to the power of Y.

exp(X): [ISO 9.3.5]
Returns the float representation of Euler's number *e* raised to the power of X.

log(X): [ISO 9.3.6]
Returns the float representation of the natural logarithm of X.

sqrt(X): [ISO 9.3.7]
Returns the float representation of the square root of X.

pi: [TC2 9.3.15]
Returns the float representation of π.

e:
Returns the float representation of Euler's number *e*.

atan2(X,Y): [TC2 9.3.13]
Returns the float representation of the arc tangent of X and Y, the result is in radians.

Arithmetic Comparisons

The arithmetic comparisons are more flexible than the lexical comparisons. They are defined for integers, floats and decimals. For decimals comparison across scales is supported. The same widening as already defined for the basic operations applies as well:

Examples:

```
1 < 2
1.0 < 2.0
0d1.00 < 2
1 =:= 0d1.00
```

We also provide evaluable functions min/2 and max/2. These functions are based on the aforementioned arithmetic comparison. The type of the return value depends on the order of the arguments of these evaluable functions.

```
min, max: integer x integer -> integer
min, max: float x float -> float
min, max: decimal x decimal -> decimal
epsilon: float
```

The constant epsilon allows querying the smallest float number that when added to one will still result in a float number different from one without any rounding.

The following built-in predicates are provided for arithmetic comparison. The built-ins arithmetically evaluate their arguments before performing their tests:

X =:= Y: [ISO 8.7.1]
Succeeds when X arithmetically equals Y, otherwise fails.
X =\= Y: [ISO 8.7.1]
Succeeds when X does not arithmetically equal Y, otherwise fails.
X < Y: [ISO 8.7.1]
Succeeds when X is arithmetically less than Y, otherwise fails.
X =< Y: [ISO 8.7.1]
Succeeds when X is arithmetically less or equal to Y, otherwise fails.
X > Y: [ISO 8.7.1]
Succeeds when X is arithmetically greater than Y, otherwise fails.
X >= Y: [ISO 8.7.1]
Succeeds when X is arithmetically greater or equal to Y, otherwise fails.

The following evaluable functions are provided for arithmetic comparison:

min(X, Y): [TC2 9.3.9]
If X and Y are both numbers then the function returns the minimum of X and Y.
max(X, Y): [TC2 9.3.8]
If X and Y are both numbers then the function returns the maximum of X and Y.
epsilon: [N208 9.7.3]
Returns the ulp of one.

Compatibility Matrix

The following compatibility issues persist for the arithmetic theory:

Table 7: Compatibility Matrix for the Arithmetic Theory

Nr	*Description*	*System*
1	Has only integer operations.	DEC10
2	Has 18-bit masking short hands (! $).	DEC10
3	Has ASCII value operator (.).	DEC10
4	Does not have decimals.	ISO
5	Does not require widening.	ISO
6	Does not require narrowing.	ISO

5.4 Structure Package

This theory is concerned with the underlying term model of the Prolog interpreter. We find the following topics:

- **Type Testing:** Testing the type of terms.
- **Term Variables:** Determining the variables of terms.
- **Lexical Comparison:** Lexically comparing terms.
- **Building Unification:** Constructing and deconstructing terms.
- **String Predicates:** Some predicates that deal with chars and codes.
- **Compatibility Matrix:** ISO/DEC10 compatibility issues of the structure theory.

Type Testing

Type testing allows checking terms for their type without an attempt to instantiate. These predicates are therefore meta-logical. The basic data types of the ISO Prolog core standard are variable, atom, integer, float and compound. The Jekejeke Prolog system adds to these data types the data types reference and decimal.

Examples:

```
?- callable(p(X,Y)).
Yes
?- callable(1).
No
```

We find elementary test predicates such as var/1, atom/1, integer/1, float/1 and compound/1. For the Jekejeke Prolog specific data types we find the test predicates reference/1, decimal/1 float32/1 and float64/1. We find also test predicates that group different data types together such as nonvar/1, atomic/1, number/1 and callable/1.

The following type testing predicates are provided:

integer(X): [ISO 8.3.3]
The predicate succeeds when X is an integer.
float(X): [ISO 8.3.4]
The predicate succeeds when X is a float.
atom(X): [ISO 8.3.2]
The predicate succeeds when X is an atom.
compound(X): [ISO 8.3.6]
The predicate succeeds when X is a compound.
reference(X):
The predicate succeeds when X is a reference.
decimal(X):
The predicate succeeds when X is a decimal.
number(X): [ISO 8.3.8]
The predicate succeeds when X is a number, i.e. an integer, a float or a decimal.
callable(X): [TC2 8.3.9]
The predicate succeeds when X is callable, i.e. an atom or a compound.
atomic(X): [ISO 8.3.5]
The predicate succeeds when X is a constant, i.e. an atom, a number or a reference.
var(X): [ISO 8.3.1]
The predicate succeeds when X is a variable.
nonvar(X): [ISO 8.3.7]
The predicate succeeds when X is not a variable, i.e. atomic or compound.
float32(X):
The predicate succeeds when X is a float32.
float64(X):
The predicate succeeds when X is a float64.

Term Variables

The test predicate ground/1 checks whether the given term is ground. This means that no un-instantiated variable occurs in the term. The predicate term_variables/2 allows collecting the un-instantiated variables that occur in a term. The predicate will thus return an empty list if the term was ground. Finally the predicate sys_term_singletons/2 collects the un-instantiated variables that only occur once. They are a subset of all the variables that occur in the term.

Examples:

```
?- sys_goal_kernel(X^p(X,Y),K).
K = p(X,Y)
?- sys_goal_globals(X^p(X,Y),L).
L = [Y]
```

Further there are predicates to deal with existential quantifiers. The existential quantifier is represented by the (^)/2 operator. In a goal X_1^..^X_n^K we call K the kernel of the goal and the variables K subtracted by the variables of X_1,..,X_n the global variables of the goal. The predicates sys_goal_kernel/2 and sys_goal_globals/2 cater for the determination of the kernel and the global variables of a goal.

An alternative to using the '$VAR'(<number>) construct is dynamically creating a variable names map. This has the advantage that the construct itself can be written out. The predicate sys_number_variables/4 helps in creating a variable names map. The resulting variable names map can be used with the predicates write_term/[2,3].

The variable names map from the current top-level query can be retrieved via the predicate sys_get_variable_names/1. The predicate will skip non-variable and duplicate entries. In the case of duplicates the entry with a lower dereferencing count is preferred. The result is intended to be used with the predicates write_term/[2,3].

The following term variable predicates are provided:

term_variables(X, L): [TC2 8.5.5]
term_variables(X, L, R):
: The predicate succeeds when L unifies with the variables of X. The variant with a third argument produces a difference list.

sys_term_singletons(X, L):
: The predicate succeeds when L unifies with the variables of X that occur only once.

sys_goal_kernel(G, K):
: The predicate succeeds when K unifies with the kernel of the goal G.

sys_goal_globals(G, L):
: The predicate succeeds when L unifies with the global variables of the goal G.

numbervars(X, N, M):
: The predicate instantiates the un-instantiated variables of the term X with compounds of the form '$VAR'(<index>). The <index> starts with N. The predicate succeeds when M unifies with the next available <index>.

sys_number_variables(V, N, S, M):
: The predicate succeeds with variable names M resulting from giving names to the variables in V, respecting the variable names N and the unnamed singletons S.

ground(X): [TC2 8.3.10]
: The predicate succeeds when X is a ground term, i.e. contains no variables.

sys_get_variable_names(L):

The predicate succeeds in L with the current variable names from the top-level query excluding non-variables and duplicates.

acyclic_term(X): [TC2 8.3.11]

The predicate succeeds when X is an acyclic term, i.e. contains no cycles.

Lexical Comparison

Lexical comparison allows comparing terms without an attempt to instantiate the terms. These predicates are therefore meta-logical. The predicates (==)/2 and (\==)/2 perform an equality test. The predicates (@<)/2, (@=<)/2, (@>)/2 and (@>=)/2 use a linear ordering based on a lexical comparison. The lexical comparison first looks on the basic type of the involved terms. The ordering of the basic types is as follows:

variable < decimal < float < integer < reference < atom < compound

Variables are ordered according to their internal instantiation numbering. Integers, floats and decimals are arithmetically ordered. But there is no mixing of integers, floats and decimals. Atoms are ordered according to their internal character representation. For compounds first the arity is compared, then the functor is compared and finally the arguments from left to right. The predicate compare/3 returns <, = or >.

Examples:

```
?- 1 @< 2.0.
No
?- compare(O, 1, 2.0).
O = >
```

Reference types can always be used in equality tests. Whether a reference type can be compared depends on whether it implements the Java Comparable interface. The predicates locale_compare/[3,4] allow a locale comparison. In locale comparison the atoms and functors are ordered according to a locale specific Java collator. Locale comparison for reference types is not yet supported.

The following built-in predicates are provided:

X == Y: [ISO 8.4.1]
The predicate succeeds when X is lexically equal to Y.

X \== Y: [ISO 8.4.1]
The predicate succeeds when X is not lexically equal to Y.

X @< Y: [ISO 8.4.1]
The predicate succeeds when X is lexically before Y.

X @=< Y: [ISO 8.4.1]
The predicate succeeds when X is lexically before or equal to Y.

X @> Y: [ISO 8.4.1]
The predicate succeeds when X is lexically after Y.

X @>= Y: [ISO 8.4.1]
The predicate succeeds when X is lexically after or equal to Y.

compare(O, X, Y): [TC2 8.4.2]
The predicate succeeds when O unifies with the result of comparing X to Y. The result is one of the following atoms <, = or >.

locale_compare(O, X, Y):
locale_compare(C, O, X, Y):
The predicate succeeds when O unifies with the result of locale comparing X to Y. The result is one of the following atoms <, = or >. The quaternary predicate allows specifying a locale C.

Building Unification

Term building allows the access and construction of arbitrary terms. Since some of the predicates require arguments to be instantiated these predicates are not logically complete. Nevertheless most of the predicates are flexible enough so that they can be called in both directions. Namely they can be called with the modes (+, -), (-, +) and (+, +), whereby + indicates an instantiated argument.

For performance reasons the interpreter performs unification without occurs check. This can result in cyclic structures which are not logically sound in the usual Herbrand model interpretation. The cyclic structures might result in infinitely looping programs or in a looping during term output. For programs that need a logically sound unification a special predicate is provided which does only instantiate variables when the check fails.

The following term building and unification predicates are provided:

X =.. Y: [ISO 8.5.3]
: If X is atomic then the predicate succeeds when Y unifies with [X]. If X is the compound $F(A_1, .., A_n)$ then the predicate succeeds when Y unifies with $[F, A_1, \ldots, A_n]$. If Y is [C] and C is atomic then the predicate succeeds when X unifies with C. If Y is $[F, A_1, \ldots, A_n]$ and F is an atom then the predicate succeeds when X unifies with $F(A_1, .., A_n)$.

functor(X, N, A): [ISO 8.5.1]
: If X is atomic then the predicate succeeds when N unifies with X and A unifies with 0. If X is the compound $F(A_1, .., A_n)$ then the predicate succeeds when N unifies with F and A unifies with n. If N is atomic and A is 0 then the predicate succeeds when Y unifies with N. If N is an atom, A is an integer $n \geq 1$ and $A_1, \ldots, A_n$ are fresh arguments then the predicate succeeds when Y unifies with $N(A_1, .., A_n)$.

arg(K, X, Y): [ISO 8.5.2]
: If K is a positive integer in the range of an arity and X is a callable $f(A_1, .., A_n)$ then the predicate succeeds when $1 \leq k \leq n$ and A_k unifies with Y.

set_arg(K, X, Y, Z):
: If K is a positive integer in the range of an arity and X is a callable $f(A_1, .., A_n)$ then the predicate succeeds when $1 \leq k \leq n$ and Z unifies with $f(A_1, .., A_{k-1}, Y, A_{k+1}, .., A_n)$.

X = Y: [ISO 8.2.1]
: The predicate succeeds when X and Y unify, no occurs check is performed.

unify_with_occurs_check(X, Y): [ISO 8.2.2]
: The predicate succeeds when X and Y unify, occurs check is performed.

X \= Y: [ISO 8.2.3]
: The predicate succeeds when X and Y do not unify, no occurs check is performed.

The following Prolog flags for term building and unification are provided:

max_arity: [ISO 7.11.2.3]
: The legal value is an integer. The value gives the maximum number of arguments in a compound. Default value is 2147483647. The value cannot be changed.

String Predicates

Characters and codes are not genuine data types in Jekejeke Prolog. Characters are simply atoms of code length one and codes are integer values in the range 0 to 0x10FFFF. Atoms are internally represented as sequences of 16-bit words. Codes in the range above 16-bit are represented as surrogate pairs. It is permitted to have single standing surrogates in an atom. It is also permitted to have surrogates in escape sequences.

Examples:

```
?- char_code('\xDBFF\\xDFFF\',X).
X = 1114111
?- char_code(X,1114111).
X = '\x10FFFF\'
?- char_code('\x10FFFF\',X).
X = 1114111
```

Besides the ISO core standard inspired atom related predicates we also provide Prolog commons inspired atom related predicates. The predicate atom_split/3 allows concatenating and splitting atom lists to and from atoms. The predicate atom_number/2 allows converting between atoms and numbers.

Examples:

```
?- atom_split(X, '_', [a,b,c]).
X = a_b_c
?- atom_split(a_b_c, '_', X).
X = [a,b,c]
```

The arguments of the below string predicates do not work with 16-bit word units. The arguments are measured in code units. This results in a certain performance penalty. For example the length of an atom is not anymore a one shot operation, but instead the whole atom has to be scanned to compute the length. Similar conversions apply to offsets.

The following string predicates are provided:

atom_length(X, Y): [ISO 8.16.1]
: The predicate succeeds when Y is the length of the atom X.

atom_concat(X, Y, Z): [ISO 8.16.2]
: The predicate succeeds whenever the atom Z is the concatenation of the atom X and the atom Y.

sub_atom(X, Y, Z, U):
sub_atom(X, Y, Z, T, U): [ISO 8.16.3]
: The predicate succeeds whenever the atom U is the sub atom of the atom X starting at position Y with length Z and ending T characters before.

atom_chars(X, Y): [ISO 8.16.4]
: If X is a variable and Y is a character list then the predicate succeeds when X unifies with the corresponding atom. Otherwise if X is an atom then the predicate succeeds when Y unifies with the corresponding character list.

atom_codes(X, Y): [ISO 8.16.5]
: If X is a variable and Y is a code list then the predicate succeeds when X unifies with the corresponding atom. Otherwise if X is an atom then the predicate succeeds when Y unifies with the corresponding code list.

char_code(X, Y): [ISO 8.16.6]
: If X is a variable and if Y is a code then the predicate succeeds when X unifies with the corresponding character. Otherwise if X is a character then the predicate succeeds when Y unifies with the corresponding code.

number_chars(X, Y): [ISO 8.16.7]

If X is a variable and Y is a character list then the predicate succeeds when X unifies with the corresponding number. Otherwise if X is a number then the predicate succeeds when Y unifies with the corresponding character list.

number_codes(X, Y): [ISO 8.16.8]

If X is a variable and Y is a code list then the predicate succeeds when X unifies with the corresponding number. Otherwise if X is a number then the predicate suceeds when Y unifies with the corresponding code list.

last_atom_concat(X, Y, Z):

The predicate succeeds whenever the atom Z is the concatenation of the atom X and the atom Y. Works like the ISO predicate atom_concat/3 except that non-deterministic results are enumerated in reverse order.

last_sub_atom(X, Z, T, U):

last_sub_atom(X, Y, Z, T, U):

The predicate succeeds whenever the atom U is the sub atom of the atom X starting at position Y with length Z and ending T characters before. Works like the ISO predicate sub_atom/5 except that non-deterministic results are enumerated in reverse order.

atom_split(L, S, R): [Prolog Commons Atom Utilities]

If R is a variable the predicate succeeds when R unifies with the concatenation of each atom from the non-empty list L separated by the atom S. Otherwise the predicate splits the atom R into a list L of atoms that are separated by the atom S.

atom_number(A, N): [Prolog Commons Atom Utilities]

If A is a variable, then the predicate succeeds in A with the number unparsing of N. Otherwise the predicate succeeds in N with the number parsing of A.

atom_block(A, B):

If A is a variable, then the predicate succeeds in A with the atom for the block B. Otherwise the predicate succeeds in B with the atom for the block B.

term_atom(T, A):

term_atom(T, A, O):

The predicate succeeds when A is the serialization of the term T. The ternary predicate accepts read respectively write options O.

The following Prolog flags for string predicates are provided:

max_code:

The legal value is an integer. The value gives the maximum value of a character code. Default value is 0x10FFFF. The value cannot be changed.

Compatibility Matrix

The following compatibility issues persist for the structure theory:

Table 8: Compatibility Matrix for the Structure Theory

Nr	*Description*	*System*
1	Does not have the predicates float and number.	DEC10
2	Does not have the predicate compound.	DEC10
3	Has predicate name.	DEC10
4	Does not have predicates atom_codes and number_codes.	DEC10
5	Does not have predicate unify_with_occurs_check.	DEC10
6	Does not have predicate copy_term.	DEC10
7	Has length predicate.	DEC10
8	Has no predicate findall/3.	DEC10
9	Has mention of null atom in lexical comparison.	ISO
10	Has mention of collating sequence in lexical comparison.	ISO
11	Does not have max_code flag.	ISO
12	Allows blanks at the beginning of numbers.	ISO
13	Has classification and case conversion built-ins.	UNID
14	Has normalization and boundary built-ins.	UNID

5.5 Reflect Package

This theory is concerned with accessing the Prolog system. The activity inside the Prolog system is distributed over multiple threads. A primary thread can interrupt a secondary thread via signals. Predicates can be marked as thread local.

- **Predicate Definitions:** Predicates can be user defined.
- **Syntax Operators:** Syntax operators can be user defined.
- **Source Files:** The Prolog system maintains of table of source files.
- **Foreign Predicates:** Predicates can be defined by Java methods.
- **Compatibility Matrix:** ISO/DEC10 compatibility issues of the system theory.

Predicate Definitions

A knowledge base consists of zero, one or more predicates. Each predicate is either a built-in or a defined predicate. Built-ins are implemented via Java methods whereas defined predicates are implemented via associated clauses. A defined predicate can have zero, one or more associated clauses. Each defined predicate is either static, dynamic or thread local. A predicate is identified by a predicate indicator:

```
indicator --> module ":" indicator
            | atom "/" integer.

module    --> package "/" atom
            | "{" array "}"
            | reference
            | atom.

array     --> package "/" atom.
            | "{" array "}"
            | atom.

package   --> package "/" atom.
            | atom.
```

The name of a predicate is qualified when it starts with a module name separated by the colon (:) operator. Unqualified predicate names are extended by the module name of the Prolog text if the Prolog text has been elevated to a module, or by the module names of corresponding public or package local predicates found in dependent modules.

Examples:

```
call/1                % is a predicate indicator
(=)/2                 % is a predicate indicator
basic/lists:member/2  % is a predicate indicator
```

A non-dynamic predicate without clauses can be declared via the directive static/1. The context of a clause is determined from the predicate name atom of the clause head.

The predicate current_predicate/1 succeeds for a predicate that is visible in the current context. The different visibility parameters are documented in the module system section. Properties of a predicate can be accessed and modified by the predicates predicate_property/2, set_predicate_property/2 and reset_predicate_property/3.

The following predicate definition predicates are provided:

static P, …:
The predicate sets the predicate P to static.
current_predicate(P): [ISO 8.8.2]
The predicate succeeds for the visible predicates P.
predicate_property(P, Q):
The predicate succeeds for the properties Q of the predicate P.
set_predicate_property(P, Q):
The predicate assigns the property Q to the predicate P.
reset_predicate_property(P, Q):
The predicate de-assigns the property Q from the predicate P.

sys_neutral_predicate(I):
: If no predicate has yet been defined for the predicate indicator I, defines a corresponding neutral predicate.

sys_make_indicator(F, A, I):
: The predicate succeeds when I is the indicator for the possibly quantified name F and the arity A.

The following predicate properties for predicate definitions are provided:

built_in:
: The property indicates that the predicate is a built-in. The property cannot be changed.

static:
: The property indicates that the predicate is static. The property cannot be changed.

full_name(N):
: The property indicates that the predicate has the flattened name N. The property cannot be changed.

Syntax Operators

A knowledge base keeps a list of zero, one or more syntax operators. The syntax operators define how Prolog terms are read and written. The interpreter provides the classical access to operators by the predicate op/3 and current_op/3. These predicates take respectively deliver an operator level, an operator mode and an operator name. If an operator has associativity, it is possible to use the operator multiple times in the same expression without parenthesis. The following operator icons are supported:

Table 9: Operator Modes

Icon	*Type*	*Associativity*
fx	prefix	No
fy	prefix	Yes
xf	postfix	No
yf	postfix	Yes
xfx	infix	No
yfx	infix	Left
xfy	infix	Right

Example:

```
?- [user].
:- op(200, xfy, ++).
append(nil, X, X).
append(X++Y, Z, X++T) :- append(Y, Z, T).
^D

?- append(X, Y, a++b++c++nil).
X = nil,
Y = a++b++c++nil ;
X = a++nil,
Y = b++c++nil
```

In the example above we have defined an infix operator (++)/2 with right associativity. Jekejeke Prolog provides further properties of individual operators. The access of the properties is based on an operator indicator which is one of the terms prefix(O), postfix(O) or infix(O) where O is the operator name. The user operator indicators can be enumerated via the predicate current_oper/1. The operator properties can be accessed and modified via the predicates oper_property/2, set_oper_property/2 and reset_oper_property/2.

```
oper    --> "prefix(" name ")"
          | "postfix(" name ")"
          | "infix(" name ")".

name    --> module ":" name
          | atom.
```

A first set of operator properties deals with the visibility of the operator. These are the properties system/0, full_name/1 and private/0. Pretty printing is done by controlling the indentation of operators and the spaces around an operator. Pretty printing is only in effect for terms aka clauses and goals. Arguments are printed in minimizing the number of spaces. Pretty printing is inferred from the meta-predicate declaration and the operator level.

The following syntax operator predicates are provided:

op(L, M, N): [ISO 8.14.3]
op(L, M, [N_1, …, N_n]): [ISO 8.14.3]
For L <> 0 the predicate (re-)defines the operator N with mode M and level L. For L = 0 the predicate undefines the operator N with mode M.

sys_neutral_oper(I):
If no syntax operator has yet been defined for the syntax operator indicator I, defines a corresponding neutral syntax operator.

current_op(L, M, O): [ISO 8.14.4]
The predicate succeeds for every defined operator O with mode M and level L.

current_oper(I):
The predicate succeeds for each user operator I.

oper_property(I, P):
The predicate succeeds for each property P of each user operator I. The following operator properties are supported:

full_name(I):	I is the qualified operator indicator.
nspl:	The operator has no space left pretty printing.
nspr:	The operator has no space right pretty printing.
level(L):	The operator precende is L.
mode(M):	The operator icon is M.

set_oper_property(I, P):
The predicate assigns the property P to the operator I.

reset_oper_property(I, P):
The predicate de-assigns the property P from the operator I.

The following syntax operator properties are provided:

sys_alias(A):
The syntax operator property indicates that the operator should be replaced by A during parsing. The property is single valued or can be missing. The property can be changed.

sys_portray(P):
The syntax operator property indicates that the operator should be replaced by P during un-parsing. The property is single valued or can be missing. The property can be changed.

Source Files

The Prolog system maintains a table of consulted source files in each knowledge base. The entries in this table have a longer life cycle than the entries in the stream alias table. When a named stream is closed, the alias is removed from the alias table and thus the stream properties are not anymore accessible via the alias. On the other hand when a source file is closed after consult, the entry in the source file table is not removed.

For a verbose consult, ensure loaded or unload the source also keeps a property with the timing of the operation. The timing is suspended when a source consults or ensure loads another source, so that the property only shows the time spent in the given source. Further a source records the other sources that this source has been consulted or ensure loaded. This way a graph of the sources and their dependency is built.

Each source has an individual lock. This lock serializes the predicate lookup, the operator lookup, the consult and the purge of sources by multiple threads. If multiple threads access sources that depend on each other this locking might cause a deadlock, since we do not the release the lock between a source and its imports. Deadlocks are detected by a timeout which can be controlled by the sys_timeout Prolog flag.

The source dependency graph is allowed to have cycles. During consult or ensure loaded a thread only processes each source once, so that there is no danger of duplicate imports or an infinite import loop. The dependency graph is also used by a mark and sweep algorithm to unload unused sources. The user and system source does play an important role here, it determines the root for the marking of the sources.

The following source file predicates are provided.

current_source(S):
: The predicate succeeds for the user absolute source paths S.

source_property(S, P):
: The predicate succeeds for the properties P of the source path S.

set_source_property(S, P):
: The predicate assigns the property P to the source path S.

reset_source_property(S, P):
: The predicate de-assigns the property P from the source path S.

sys_current_resource(P):
: Succeeds for every member P of the list of error resources.

current_module(M):
: The predicate succeeds for the modules M.

sys_context_property(C, Q):
: The predicate succeeds for the context property Q of the callable C.

sys_set_context_property(B, Q, A):
: The predicate succeeds for a new callable B which is a clone of the callable A with the context property Q.

The following source properties for source files are provided:

sys_capability(P):
: The property indicates that the source has the parent capability P. The property cannot be changed.

last_modified(L):
The property indicates that the source was last modified at date L. The date is given in milliseconds since January 1, 1970 GMT. An undefined date is indicated by the value 0. The property cannot be changed.

version_tag(V):
The property indicates that the source has the version tag V. The version tag is an atom which starts and ends with double quotes (“). An undefined version tag is indicated by the empty string. The property cannot be changed.

expiration(E):
The property indicates that the source will expire at the date E. The date is given in milliseconds since January 1, 1970 GMT. An undefined date is indicated by the value 0. The property cannot be changed.

date(E):
The property indicates that the source was requested at the date E. The date is given in milliseconds since January 1, 1970 GMT. An undefined date is indicated by the value 0. The property cannot be changed.

max_age(E):
The property indicates that the max age of the source in seconds. An undefined max age is indicated by the value -1. The property cannot be changed.

short_name(S):
The property indicates that the source has the short name S. The property cannot be changed.

Foreign Predicates

Foreign predicates can be defined by Java methods, constructors and fields. Foreign predicates will have retrieved the actual goal arguments by the interpreter and automatically passed to the associated Java method, constructor or field. More details about foreign predicates can be found in the Jekejeke Prolog Programming Interface documentation.

Foreign predicates can be registered by one of the directives foreign/3, foreign_constructor/3 foreign_setter/3 and foreign_getter/3. Foreign evaluable functions can be registered by one of the directives foreign_function/3 or foreign_constant/3. The directives take as arguments a predicate specification, a declaring class specification and a method, constructor or field specification. We can describe the arguments via the following syntax:

```
directive --> "foreign(" indicator "," module "," signature ")"
            | "foreign_constructor(" indicator "," module "," signature ")"
            | "foreign_setter(" indicator "," module "," atom ")"
            | "foreign_getter(" indicator "," module "," atom ")"
            | "foreign_fun(" indicator "," module "," signature ")"
            | "foreign_const(" indicator "," module "," atom ")".

signature --> atom [ "(" module { "," module } ")" ].
```

Example:

```
:- foreign(hello/0, 'OutHello', hello('Interpreter')).
                          % is a foreign predicate directive.
```

Not all declared classes or parameter types have to be fully qualified. The below class names can be directly used without specifying the package name. All formal parameters not of class Term cause a range check and/or conversion of the actual argument. A formal parameter of type BigDecimal or BigInteger causes a widening, whereas a formal parameter of type Integer, Float or Long causes a range check.

The supported primitive datatypes are handled analogously. The Java method, constructor or field might also have one of the above classes or primitive types as a return type. By returning a non-null object the Java method, constructor or field can indicate success and the interpreter will unify the object with the last argument of the corresponding predicate. By returning a null the Java method or field can indicate a failure.

A Java method might also have a boolean or a void return type. The return type boolean can indicate success or failure without returning an object. The return type void always indicates success without returning an object. For non-static methods or fields an additional argument for the receiving object is added to the front of the foreign predicate.

A formal parameter of type Interpreter is needed for foreign predicate that change variable bindings. A formal parameter of type CallOut is needed for non-deterministic foreign predicates. The API of the CallOut allows fine control of the creation of choice points, of the choice point data, of clean-up handling and barrier handling. For more information one should consult the programming interface documentation.

Table 10: Parameter Type Mapping

Java Type	*Prolog Type*
java.lang.String	atom
java.lang.CharSequence	atom or reference
java.lang.Boolean, boolean	atom from the set {true, false}
java.lang.Byte, byte	integer between - 2^7 and 2^7-1
java.lang.Char, char	char between 0 and 2^16-1
java.lang.Short, short	integer between - 2^15 and 2^15-1
java.lang.Integer, int	integer between - 2^31 and 2^31-1
java.lang.Long, long	integer between - 2^63 and 2^63-1
java.math.BigInteger	integer
java.lang.Float, float	float32
java.lang.Double, double	float
java.math.BigDecimal	decimal
java.lang.Number	number
jekpro.tools.term.AbstractTerm	term
java.lang.Object	term
non-arrays	reference
arrays	reference
jekpro.tools.call.Interpreter	The current interpreter
jekpro.tools.call.Callout	The current call-out

The interpreter allows InterpreterException and InterpreterMessage exceptions thrown by the Java method. For InterpreterMessage the Prolog stack trace is determined and a corresponding InterpreterException is thrown. A couple of Java exceptions are recognized and wrapped into Prolog errors before throwing. The family of interrupt exceptions is mapped to the signal currently stored in the interpreter and the signal is cleared.

Table 11: Exception Type Mapping

Java Class	*Jekejeke Prolog Error*
jekpro.tools.call.InterpreterException	As is
jekpro.tools.call.InterpreterMessage	As is plus stack trace
jekpro.tools.term.RuntimeWrap	Mapping of the <cause>
java.net.SocketTimeoutException	resource_error(socket_timeout)
java.io.InterruptedIOException	<signal>
java.nio.channels.FileLockInterruptionException	<signal>
java.nio.channels.ClosedByInterruptException	<signal>
java.io.UnsupportedEncodingException	existence_error(encoding, <msg>)
java.net.MalformedURLException	syntax_error(malformed_url)
java.util.zip.ZipException	resource_error(corrupt_archive)
java.nio.charset.CharacterCodingException	syntax_error(malformed_path)
java.io.FileNotFoundException	existence_error(source_sink, <msg>)
java.net.UnknownHostException	existence_error(host, <msg>)
java.net.SocketException	existence_error(port, <msg>)
java.io.IOException	resource_error(io_exception)
java.lang.InterruptedException	<signal>
java.lang.ArithmeticException	evaluation_error(<msg>)
java.lang.RuntimeException	system_error(<msg>)
java.lang.Error	As is
java.lang.Exception	representation_error(<msg>)
Otherwise	New java.lang.Error

Prolog variables and Prolog compounds are always passed as the Java classes TermVar and TermCompound. A formal parameter or result by the Java class Object does not wrap

atomics and these are passed unwrapped. On the other hand a formal parameter or result by the Java class Term, does wrap atomics and these are passed wrapped. The distinction is important for atoms, since atoms also carry call-site information which can only be preserved by representing them through the Java class TermAtomic.

The following foreign predicate predicates are provided:

foreign(I, C, M):
: Succeeds with registering the predicate indicator I as a foreign predicate that calls the method M of the class C.

foreign_constructor(I, C, M):
: Succeeds with registering the predicate indicator I as a foreign predicate that calls the constructor M of the class C.

foreign_getter(I, C, M):
: Succeeds with registering the predicate indicator I as a foreign predicate that gets the field M of the class C.

foreign_setter(I, C, M):
: Succeeds with registering the predicate indicator I as a foreign predicate that sets the field M of the class C.

foreign_fun(I, C, M):
: Succeeds with registering the predicate indicator I as a foreign evaluable function that calls the method M of the class C.

foreign_const(I, C, M):
: Succeeds with registering the predicate indicator I as a foreign evaluable function that gets the field M of the class C.

Compatibility Matrix

The following compatibility issues persist for the system theory:

Table 12: Compatibility Matrix for the System Theory

Nr	*Description*	*System*
1	N/A	N/A

5.6 Bootload Package

This theory is concerned with extending the Prolog system. One way to extend the Prolog system is by adding foreign predicates. Another way to extend the Prolog system is by adding a capability.

- **Interpreter State:** Some interpreter state can be queried and updated.
- **Capability Plug-Ins:** Capabilities bundle a set of predicates.
- **Prolog Texts:** Knowledge bases can be consulted from text sources.
- **Path Resolution:** Paths can be resolved against the class paths or active libraries.
- **Module System:** Predicates can be protected from external access.
- **Compatibility Matrix:** ISO/DEC10 compatibility issues of the administration theory.

Interpreter State

Some interpreter state can be queried and updated. The predicate current_prolog_flag/2 allows accessing an interpreter attribute. The predicate set_prolog_flag/2 allows updating an interpreter attribute. The predicates halt/[0,1] allow exiting the current process.

The predicates begin_module/1 and end_module/0 can be to open respectively close a local module. For a consulted file the predicate begin_module/1 will also do first a clear of the local module, and the predicate end_module/0 will do a style check of the local module.

The following interpreter state predicates are provided:

current_prolog_flag(F, V): [ISO 8.17.2]
: The predicate succeeds for the value V of the flag F.

set_prolog_flag(F, V): [ISO 8.17.1]
: The predicate sets the flag F to the value V.

halt: [ISO 8.17.3]

halt(N): [ISO 8.17.4]
: The predicate without arguments terminates the interpreter with exit value zero. The unary predicate terminates the interpreter with exit value N.

welcome:

version:
: The predicate displays a version banner.

begin_module(N):
: The predicate begins a new typein module N.

end_module:
: The predicate ends the current typein module.

Capability Plug-Ins

Capabilities bundle a set of predicate definitions. Capabilities are also able to bundle special built-ins and various properties. Further capabilities define a product description. The programming interface of capabilities is currently not in the public domain. But capabilities itself might be placed on the market by parties that will have access to this programming interface.

Capabilities that need activation can only be added with a valid activation. Activations can be performed over the internet via the predicate sys_activate_capability/2. If internet access is not available an install ID can be retrieved via sys_calc_install_id/2. This install ID can be sent to the supplier by E-mail, surface mail, phone etc.. who will in turn return a license text. This license text can then be deposited via the predicate sys_reg_license_text/2.

A capability can then be added via the predicate sys_init_capability/1. Capabilities can also be removed at any time via the predicate sys_fini_capability/1. It is recommended to remove unused capabilities, since a capability migh have spawn a thread which is shutdown by the removal. The currently tracked capabilities can be retrieved via the predicate sys_current_capability/1.

The license for a capability might expire or be tempered with. To validate it the predicate sys_check_license/1 can be called. Calling the predicate only makes sense if the application cannot wait for when the system automatically validates the license. The predicate sys_check_licenses/0 allows updating the knowledge base status. The predicate assumes that the licenses of the enlisted capabilities have already been validated.

The following capability plug-in predicates are provided:

sys_activate_capability(C, H):
: The predicate activates the capability C by the license key H.

sys_calc_install_id(C, I):
: The predicate calculates the install ID for the capability C and unifies it with I.

sys_reg_license_text(C, T):
: The predicate registers the license text T for the capability C in the user preferences.

sys_reged_license_text(C, T):
: The predicate returns the registeres license text in T for the capability C in the user preferences.

sys_init_capability(C):
sys_init_capability(C, O):
: The unary predicate initializes and enlists the capability C. The binary predicate additionally recognizes the following init options:

    prompt(B): B is prompt (true or false), default value is false.

sys_fini_capability(C):
: The predicate removes the capability C.

sys_current_capability(C):
: The predicate succeeds with the currently initialized capabilities C.

sys_capability_property(C, P):

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

family_descr(D):	D is the localized family description.
product_descr(D):	D is the localized product description.
language_descr(D):	D is the localized language description.
platform_descr(D):	D is the localized platform description.
needs_act(B):	B is the need for activation (true or false).
act_status(S):	S is the activation status.
license_descr(D):	D is the localized license description.
expiration_date(T):	T is the expiration date.
bundle_dir(P):	P is the path of the bundle storage.
image_icon(I):	I is the image icon.
big_image_icon(I):	I is the big image icon.
help_docs(H):	H is the list of localized document titles and their URLs.
shop_url(U):	U is the URL of the license provider.
sys_notrace(B):	B is the no trace flag (true or false).

The properties act_status/1, license_descr/1 and expiration_data/1 are only available when the capability has been successful enlisted. The expiration_date/1 is given in milliseconds since January 1, 1970 GMT. An undefined expiration_date/1 is returned as the value 0.

sys_check_license(C):

The predicate updates the license status of the capability C. Throws an exception if license status is not OK.

sys_check_licenses:

The predicate updates the knowledge base status. Throws an exception if the knowledge base status is not OK.

The following Prolog flags for capability plug-ins are provided:

sys_act_status:

The value is a license error type. The value indicates the activation status of the predefined and enlisted capabilities. The value cannot be changed.

Prolog Texts

Prolog texts have the syntax of a consult text, see section 4.3. When consulted the contained facts and rules are asserted. The encountered head predicates of the facts and rules are declared static by default. Further any directives in the theory text are executed in the order they appear. These directives might additionally declare foreign predicates. Upon consulting again a source, the related declared predicates are first abolished.

The consult performs various style checks. The facts and rules are checked for singleton variables. Singleton variables need to be input in the form of anonymous variables (_). The facts and rules for the same head predicate need to form one block. The directive discontiguous/1 allows exempting a predicate from this style check. The special file name user can be used to consult from the standard input.

Normally the facts and rules for the same head predicate come from one source only. The directive multifile/1 allows exempting a predicate from this style check. Multi-file predicates behave differently during re-consult. If a declared predicate spans multiple sources only the clauses belonging to the re-consulted source are retracted. The defined predicate will only be abolished when it does not belong to any source anymore.

The following theory files predicates are provided:

ensure_loaded(R): [ISO 7.4.2.8]
: The predicate ensures that the relative source path R is loaded. If the current time is after the expiration of the source then it will connect to the source. If the source was not modified since its last modified then it will consult the source.

consult(R):
: First retract the old facts and rules of the relative source path R. Then assert the new facts and rules from the relative source path R. During assert also process the directives from the relative source path R. Before assert the scope is temporarily changed to the relative source path R.

unload_file(R):
: Detach the source identified by the relative source path R.

[S_1, ..., S_m]:
: The predicate processes the path specifications S_1, ..., S_m. The following path specifications are recognized:

+R:	Ensure load the relative path R.
-R:	Detach the relative path R.
R:	Consult the relative path R.

make:
: The predicate ensures that all used sources are loaded.

rebuild:
: The predicate consults all used sources.

include(R): [ISO 7.4.2.7]
: Doesn't retract the old facts and rules of the relative source path R. Asserts the new facts and rules from the relative source path R. Processes the directives from the relative source path R. Doesn't change the scope to the relative source path R.

discontiguous I, …: [ISO 7.4.2.3]
: The predicate sets the predicate indicator I to discontinuous.

multifile I, …: [ISO 7.4.2.2]
: The predicate sets the predicate indicator I to multi-file.

listing:
The predicate lists the user clauses of the user syntax operators, evaluable functions and predicates. Only non-automatic evaluable functions and predicates are listed.

listing(I):
The predicate lists the user clauses of the user syntax operators, evaluable functions and predicates that match the pattern I. Only non-automatic evaluable functions and predicates are listed.

The following Prolog flags for theory files are provided:

sys_last_pred:
The value is a predicate indicator or null. The value indicates the head predicate of the recently consulted clause. The value can be changed.

sys_timeout:
The value is a positive 64-bit integer. The value determines the wait before source locks timeout during loading. The value is measured in milliseconds. The value can be changed.

verbose:
The value can be one of the atoms off, summary, details or on. The value indicates the verbosity level for the loading and unloading of files. The summary level shows a count of the loaded and unloaded files. The details level shows each loaded or unloaded file names. The on level shows both, details followed by summary.

The following predicate properties for theory files are provided:

sys_usage(S):
The property indicates that the defined predicate has definitions in the source context S. The property is multi valued or can be missing. The property cannot directly be changed.

discontiguous(S):
The property indicates that the predicate has been marked discontinuous in the source context S. The property is multi valued and can be missing.

multifile:
The property indicates that predicate is multi-file.

sys_multifile(S):
The property indicates that the predicate has been marked multi-file in source context S. The property is multi valued and can be missing. The property can be changed.

The following syntax operator properties for theory files are provided:

sys_usage(S):
The syntax operator property indicates that the defined operator has definitions in the source context S. The property is single valued or can be missing. The property cannot directly be changed.

The following theory files operators are provided:

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

Path Resolution

A relative path that is not wrapped into a compound is resolved against the current base in write or append mode. The current base is the Prolog flag “base_url”. Otherwise in read mode, if the call-site is not user it is resolved against the path of the call-site itself. In both cases the suffixes of the sys_add_file_extension/1 command are used:

Example:

```
?- absolute_file_name('my folder/my file', X).
Error: File 'my folder/my file' not found.
      absolute_file_name/2

?- set_prolog_flag(base_url, '/C:/Users/Jan Burse/Desktop/').
Yes

?- sys_add_file_extension(text('.dcg', 'text/prolog')).
Yes

?- absolute_file_name('my folder/my file', X).
X = 'file:/C:/Users/Jan Burse/Desktop/my folder/my file.dcg'
```

Paths should not use a system specific directory separator but always use the forward slash (/). For convenience paths have an automatic prefixing of a schema. Paths starting with a double slash (//) are prefixed by the “http” schema. Paths starting with a single slash (/) are prefixed by the “file” schema. Drive letters are not considered schema.

If the path is wrapped into a compound and if the functor of the compound is either library/1, foreign/1 or verbatim/1 then the path is looked up in the class path. The class path can be updated and queried by the predicates sys_add_path/1 and sys_current_path/1. In these cases the prefixes of the package/1 and use_package/1 command are also used.

Write or append access resolution:

```
  <path>                    resolve <path> in base.
```

Read access resolution:

```
  library(<path>)           lookup resource <path> in class path.
  foreign(<path>)           lookup class <path> in class path.
  verbatim(<path>)          like library(<path>) or take as is.
  <path>                    resolve <path> in scope or base.
```

The predicates absolute_file_name/[2,3] and absolute_resource_name/1 provide file name resolving. The predicate absolute_file_name/2 works bi-directionally. For a given already resolved path it will make a best effort attempt to reconstruct either a compound form foreign/1, library/1 or verbatim/1 or a relative path.

The following path resolution predicates are provided:

sys_add_path(R):
The predicate succeeds in adding the relative path R to the current class loader.

sys_current_path(A):
The predicate succeeds in A with the currently added absolute paths A along the class loaders.

sys_add_file_extension(E):
The predicate succeeds in adding the file extension database entry E to the current knowledge base. The following database entries are recognized:

text(<name suffix>, <mime type>):	Library file.
binary(<name suffix >, <mime-type>):	Foreign file.
resource(<name suffix >, <mime-type>):	Resource file.

sys_remove_file_extension(E):
The predicate succeeds in removing file extension database entry with the name suffix E from the current knowledge base.

sys_current_file_extension(E):
The predicate succeeds in E with the currently added file extension database entries along the knowledge bases.

absolute_file_name(R, A):
absolute_file_name(R, A, O):
The binary predicate succeeds when the read path R resolves to an absolute path and this absolute path unifies with S. The ternary predicate additionally recognizes the following path option:

access(M):	M is the access (read, write or append).

The binary predicate can also be used in a backward manner.

absolute_resource_name(R, A):
The binary predicate succeeds when the read path R resolves to an absolute resource path A.

The following Prolog flags for path resolution are provided:

base_url:
The legal value is an atom. The value is the base URL of the current knowledge base. A missing base URL is indicated by the atom ‘’. The value can be changed.

Module System

A Prolog text member can be either public, package local or private. Private members can only be seen from within the same Prolog text. Package local members will be seen from Prolog texts that share the same package name. And public members are visible everywhere. The default visibility for a member is package local inside Prolog texts elevated to modules and public for ordinary Prolog texts.

Examples:

```
?- member(X,[1,2,3]).
X = 1 ;
X = 2 ;
X = 3

?- member2([2,3],X,1).
Error: Undefined, private or package local predicate member2/3.
```

Normal Prolog texts can be turned into modules via the predicate module/2. For a nameless module the name user can be used. A further convenience is the predicate use_module/1 which does an ensure loaded of the given file specification. Instead of the predicate use_module/1 also the predicate reexport/1 can be used. The later predicate will make the corresponding imported members visible to qualified invocations and client imports.

Examples:

```
?- absolute_file_name(library(basic/lists), Y),
   source_property(Y, package(X)).
X = library(jekpro/frequent/basic)

?- absolute_file_name(library(basic/lists), Y),
   source_property(Y, sys_source_name(X)).
X = lists
```

The path resolution uses prefixes from the current source and the system sources along the knowledge bases. The prefixes for the current source can be set via the predicates package/1 and use_package/1. The prefixes for the system sources can be set via the predicate set_source_property/2. The prefixes can be queried via the source_property/2 predicate.

The following module system predicates are provided:

package(E):
: The predicate adds the prefix P to the list of prefixes of the current source. Currently library/1 and foreign/1 prefixes are supported. The prefix is also used as a prefix to the module/2 directive.

use_package(P):
: The predicate adds the prefix P to the list of prefixes of the current source. Currently library/1 and foreign/1 prefixes are supported. The prefix is not used as a prefix to the module/2 directive.

module(N, L):
: The predicate is a convenience for a combination of setting the module name to N, setting the source to package and setting the members L to public.

use_module(R):
: The predicate imports the read path R with making its predicates and syntax operators visible.

reexport(R):
The predicate imports the read path R with making its predicates and syntax operators visible. The predicates and syntax operators along the reexport chain become also visible.

sys_auto_load(R):
The predicate imports the read path R without making its predicates and syntax operators visible.

sys_load_resource(R):
The predicate imports the read path R trying resolve it to a resource bundle instead of a Prolog text.

sys_add_resource(R):
Add the read path R to the list of error resources.

private P, ..:
The predicate sets the member P to private.

public P, ..:
The predicate sets the member P to public.

override I, …:
The predicate sets the predicate indicator I to override.

The following predicate properties for the module system are provided:

visible(V):
The property indicates that the predicate or evaluable function has visibility V. The argument V can have the values “public” and “private”. The property is single valued and can be missing.

sys_public(S):
The property indicates that the predicate or evaluable function has been marked public in source context S. The property is multi valued and can be missing. The property can be changed.

sys_private(S):
The property indicates that the predicate or evaluable function has been marked private in source context S. The property is multi valued and can be missing.

override(S):
The property indicates that the predicate has been marked override in source context S. The property is multi valued and can be missing. The property can be changed.

The following operator properties for the module system are provided:

visible(V):
The property indicates that the syntax operator has visibility V. The argument V can have the values “public” and “private”. The property is single valued and can be missing. The property can be changed.

override:
The property indicates that the override warning for this operator is suppressed. The property is single valued and can be missing. The property can be changed.

The following source properties for the module system are provided:

sys_source_preload:
The property indicates that the source is preload. The property is single valued and can be missing. The property can be changed for user sources.

sys_source_visible(V):
The property indicates that the source has default visibility V for their members. The argument V can have the values “public” and “private”. The property is single valued and can be missing. The property can be changed for user sources.

sys_source_name(N):
The property indicates that the source has the module package and name N. The property is single valued and can be missing. The property can be changed for user sources.

sys_timing(T):
The property indicates the time spent for consult, ensure loaded or unload in milliseconds. The property cannot be changed.

sys_link(S, M):
The property indicates that this source is linked to the source S with visibility and export mode M. The mode M can take the values use_module, reexport, sys_auto_load, sys_load_resource and sys_parent_module. The property is multi-value and can be missing. The property cannot be changed.

package(P):
The property indicates that this source has prefix P and that the name of the source has also this prefix. The prefix P can be a compound with functor library/1 or foreign/1. The argument of the compound should be a package. The property is single-valued and can be missing. The property can be changed.

use_package(P):
The property indicates that this source has prefix P, but it doesn't put forward a prefix for the source name. The prefix P can be a compound with functor library/1 or foreign/1. The argument of the compound should be a package. The property is multi-value and can be missing. The property can be changed.

Compatibility Matrix

The following compatibility issues persist for the administration theory:

Table 13: Compatibility Matrix for the Administration Theory

Nr	*Description*	*System*
1	Modes suggest multiple implementation methods.	DEC10
2	No API documentation therefore language binding unclear.	DEC10
3	Templates suggest multiple implementation methods.	ISO
4	Template allows out and in-out, besides in parameters.	ISO
5	No API documentation therefore language binding unclear.	ISO

6 Appendix Example Listings

The below examples can be also browsed on GitHub:

http://github.com/jburse/jekejeke-samples/tree/master/jekrun/reference

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

6.1 Animals Example [ISO]

For the animals example there are the following sources:

- **animals.p:** The Prolog text.

Prolog Text animals

```
/**
 * Prolog code for the logic example.
 *
 * Copyright 2012, XLOG Technologies GmbH, Switzerland
 * Jekejeke Prolog 0.9.4 (a fast and small prolog interpreter)
 */

motion(walk).
skin(fur).
diet(meat).

class(mamal) :- motion(walk), skin(fur).
class(fish) :- motion(swim), skin(scale).
class(bird) :- motion(fly), skin(feather).

animal(rodent) :- class(mamal), diet(plant).
animal(cat) :- class(mamal), diet(meat).
animal(salmon) :- class(fish), diet(meat).
animal(eagle) :- class(bird), diet(meat).
```

6.2 Primes Example [ISO]

For the primes example there are the following sources:

- **primes.p:** The Prolog text.

Prolog Text primes

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

/**
 * Generate a list of integers between Low and High.
 */
integers(Low, High, [Low | Rest]) :-
      Low =< High, !,
      M is Low + 1,
      integers(M, High, Rest).
integers(_, _, []).

/**
 * Remove all multiples of the given prime from a list.
 */
remove([], _, []).
remove([I | Is], P, Nis) :-
      I rem P =:= 0, !,
      remove(Is, P, Nis).
remove([I | Is], P, [I | Nis]) :-
      remove(Is, P, Nis).

/**
 * Detect primes and remove multiples.
 */
sift([I | Is], High, [I | Is]) :-
    I * I > High, !.
sift([I | Is], High, [I | Ps]) :-
      remove(Is, I, New),
      sift(New, High, Ps).

/**
 * First create the numbers and then sift.
 */
primes(High, R) :-
    integers(2, High, L),
    sift(L, High, R).
```

6.3 Money Example [ISO]

For the money example there are the following sources:

- **money.p**: The Prolog text to solve the letter puzzle.

Prolog Text money

```
/**
 * Prolog code for the backtracking example.
 *
 * Puzzle originally published July 1924 issue of
 * Strand Magazine by Henry Dudeney
 *
 * Copyright 2012, XLOG Technologies GmbH, Switzerland
 * Jekejeke Prolog 0.9.4 (a fast and small prolog interpreter)
 */

% oneof(+List,-Elem,-List)
oneof([X|Y], X, Y).
oneof([X|Y], Z, [X|T]) :- oneof(Y, Z, T).

% assign(-List,+List)
assign([],  ).
assign([X|Y], L) :- oneof(L, X, R), assign(Y, R).

% puzzle(-List)
puzzle(X) :-
   X = [S,E,N,D,M,O,R,Y],
   assign(X, [0,1,2,3,4,5,6,7,8,9]),
   M =\= 0,
   S =\= 0,
             1000*S + 100*E + 10*N + D +
             1000*M + 100*O + 10*R + E =:=
   10000*M + 1000*O + 100*N + 10*E + Y.
```

6.4 Rabbits Example

For the rabbits example there are the following sources:

- **cage.p**: The Prolog text for the cage.
- **nest.p**: The Prolog text for the nest.

Prolog Text cage

```
/**
 * Prolog code for the module mutual recursion example.
 *
 * Growth of an idealized rabbit population, according
 * to Liber Abaci from 1202 by Leonardo of Pisa, known
 * as Fibonacci.
 *
 * Copyright 2014, XLOG Technologies GmbH, Switzerland
 * Jekejeke Prolog 1.0.1 (a fast and small prolog interpreter)
 */

:- module(cage, [adults/2]).
:- use_module(nest).

adults(0, 1) :- !.
adults(N, X) :-
   N > 0, M is N-1,
   adults(M, Y), babies(M, Z), X is Y+Z.
```

Prolog Text nest

```
/**
 * Prolog code for the module mutual recursion example.
 *
 * Growth of an idealized rabbit population, according
 * to Liber Abaci from 1202 by Leonardo of Pisa, known
 * as Fibonacci.
 *
 * Copyright 2014, XLOG Technologies GmbH, Switzerland
 * Jekejeke Prolog 1.0.1 (a fast and small prolog interpreter)
 */

:- module(nest, [babies/2]).
:- use_module(cage).

babies(0, 0) :- !.
babies(N, X) :-
   N > 0, M is N-1,
   adults(M, X).
```

6.5 Prefect Example

For the perfect example there are the following sources:

- **perfect.p**: The Prolog text for the prefect number search.

Prolog Text perfect

```
/**
 * Prolog code for the parallel search example.
 *
 * Perfect numbers were deemed to have important numerological
 * properties by the ancients, and were extensively studied
 * by the Greeks, including Euclid.
 * http://mathworld.wolfram.com/PerfectNumber.html
 *
 * Copyright 2019, XLOG Technologies GmbH, Switzerland
 * Jekejeke Prolog 1.3.5 (a fast and small prolog interpreter)
 */

:- use_module(library(advanced/arith)).

perfect(X) :-
   Y is X//2,
   findall(Z, (between(1,Y,Z), X rem Z=:=0), L),
   sum_list(L, X).

sum_list([], 0).
sum_list([X|Y], R) :-
   sum_list(Y, H),
   R is X+H.

%%% single CPU
% ?- time((between(1,20000,X), perfect(X), fail; true)).

%%% multi CPU
% ?- use_module(library(runtime/distributed)).

% ?- time((balance((between(1,20000,X), perfect(X))), fail; true)).
```

6.6 Pound Example

For the perfect example there are the following sources:

- **basset.p**: The Prolog text for the basset class.
- **dog.p**: The Prolog text for the dog class.

Prolog Text basset

```
/**
 * Prolog code for the object oriented programming example.
 *
 * Copyright 2019, XLOG Technologies GmbH, Switzerland
 * Jekejeke Prolog 1.3.5 (a fast and small prolog interpreter)
 */

% ?- sys_add_path('<path>').

:- package(library(example06)).

:- module(basset, [barking/2]).
:- reexport(dog).

:- override barking/2.
barking(_, woof).

% ?- use_package(library(example06)).

% ?- basset(lafayette)::bark.
% lafayette says woof.
% Yes
```

Prolog Text dog

```
/**
 * Prolog code for the object oriented programming example.
 *
 * Copyright 2019, XLOG Technologies GmbH, Switzerland
 * Jekejeke Prolog 1.3.5 (a fast and small prolog interpreter)
 */

% ?- sys_add_path('<path>').

:- package(library(example06)).

:- module(dog, [bark/1, barking/2]).

bark(Self) :-
   arg(1, Self, Name),
   Self::barking(Barking),
   write(Name), write(' says '), write(Barking), write('.'), nl.

barking(_, ruff).

% ?- use_package(library(example06)).

% ?- dog(susi)::bark.
% susi says ruff
% Yes
```

```
% ?- dog(strolch)::bark.
% strolch says ruff
% Yes
```

Acknowledgements

We are grateful to Bart Demoen, KU Leuven for participation in discussion on ISO compatibility and body conversion of Jekejeke Prolog 0.9.1.

We would like to thank Ulrich Neumerkel, Technische Universität Wien, for testing the syntax and the console of Jekejeke Prolog 0.9.1 and handing us around 50 findings.

Further thanks go to Jos De Roo, Agfa Healthcare for providing us a critical test case for the indexing and for meta-arguments of Jekejeke Prolog 0.9.7.

Indexes

Public Predicates

Predicate	Module
* /3	arithmetic/elem
** /3	arithmetic/trigo
*-> /2	runtime/logic
(+)/2	arithmetic/elem
(+)/3	arithmetic/elem
,/2	runtime/logic
(-)/2	arithmetic/elem
(-)/3	arithmetic/elem
-> /2	runtime/logic
. /2	bootload/load
/ /3	arithmetic/elem
// /3	arithmetic/round
/\ /3	arithmetic/bits
: /2	runtime/quali
: /3	runtime/quali
:: /2	runtime/quali
:: /3	runtime/quali
;/2	runtime/logic
< /2	arithmetic/compare
<< /3	arithmetic/bits
= /2	structure/univ
=.. /2	structure/univ
=:= /2	arithmetic/compare
=< /2	arithmetic/compare
== /2	structure/lexical
=\= /2	arithmetic/compare
> /2	arithmetic/compare
>= /2	arithmetic/compare
>> /3	arithmetic/bits
@< /2	structure/lexical
@=< /2	structure/lexical
@> /2	structure/lexical
@>= /2	structure/lexical
[]/0	bootload/load
[]/3	arithmetic/eval
[]/4	arithmetic/eval
[]/5	arithmetic/eval
[]/6	arithmetic/eval
[]/7	arithmetic/eval
[]/8	arithmetic/eval
[]/9	arithmetic/eval
(\)/2	arithmetic/bits
(\+)/1	runtime/logic
\/ /3	arithmetic/bits
\= /2	structure/univ
\== /2	structure/lexical
^ /3	arithmetic/elem
abolish/1	runtime/dynamic
abort/0	runtime/session
abs/2	arithmetic/elem
absolute_file_name/2	bootload/path

absolute_file_name/3	bootload/path
absolute_resource_name/2	bootload/path
acos/2	arithmetic/trigo
acyclic_term/1	structure/vars
arg/3	structure/univ
asin/2	arithmetic/trigo
asserta/1	runtime/dynamic
assertz/1	runtime/dynamic
atan/2	arithmetic/trigo
atan2/3	arithmetic/trigo
atom/1	structure/type
atom_chars/2	structure/atom
atom_codes/2	structure/atom
atom_concat/3	structure/atom
atom_length/2	structure/atom
atom_list_concat/3	structure/atom
atom_or_instance_of/2	reflect/foreign
atom_property/2	reflect/pred
atomic/1	structure/type
balance/1	runtime/distributed
balance/2	runtime/distributed
boolean/1	reflect/foreign
break/0	runtime/session
call/1	runtime/logic
callable/1	structure/type
ceiling/2	arithmetic/round
char16/1	reflect/foreign
char_code/2	structure/atom
clause/2	runtime/dynamic
close/0	runtime/session
compare/3	structure/lexical
compound/1	structure/type
consult/1	bootload/load
cos/2	arithmetic/trigo
current_op/3	reflect/oper
current_oper/1	reflect/oper
current_predicate/1	reflect/pred
current_prolog_flag/2	bootload/engine
current_resource/1	reflect/source
current_source/1	reflect/source
decimal/1	structure/type
decimal/2	arithmetic/elem
(discontiguous)/1	bootload/load
div/3	arithmetic/round
(dynamic)/1	runtime/dynamic
e/1	arithmetic/trigo
ensure_loaded/1	bootload/load
exit/0	runtime/session
exp/2	arithmetic/trigo
float/1	structure/type
float/2	arithmetic/elem
float32/1	structure/type
float32/2	arithmetic/elem
float64/1	structure/type
floor/2	arithmetic/round
foreign/3	reflect/foreign

foreign_const/3	reflect/foreign
foreign_constructor/3	reflect/foreign
foreign_fun/3	reflect/foreign
foreign_getter/3	reflect/foreign
foreign_setter/3	reflect/foreign
halt/0	bootload/engine
halt/1	bootload/engine
horde/1	runtime/distributed
horde/2	runtime/distributed
include/1	bootload/load
instance_of/2	reflect/foreign
integer/1	structure/type
integer/2	arithmetic/round
integer16/1	reflect/foreign
integer16_and_not_integer8/1	reflect/foreign
integer32/1	reflect/foreign
integer32_and_not_integer16/1	reflect/foreign
integer64/1	reflect/foreign
integer64_and_not_integer32/1	reflect/foreign
integer64_or_float/1	reflect/foreign
integer64_or_float32/1	reflect/foreign
integer8/1	reflect/foreign
integer_and_not_integer64/1	reflect/foreign
is/2	arithmetic/eval
last_atom_concat/3	structure/atom
last_sub_atom/4	structure/atom
last_sub_atom/5	structure/atom
listing/0	bootload/load
listing/1	bootload/load
locale_compare/3	structure/lexical
locale_compare/4	structure/lexical
log/2	arithmetic/trigo
make/0	bootload/load
max/3	arithmetic/compare
(meta_function)/1	runtime/meta
(meta_predicate)/1	runtime/meta
min/3	arithmetic/compare
mod/3	arithmetic/round
module/2	bootload/module
(multifile)/1	bootload/load
number/1	structure/type
number_chars/2	structure/atom
number_codes/2	structure/atom
numbervars/3	structure/vars
once/1	runtime/logic
op/3	reflect/oper
oper_property/2	reflect/oper
(override)/1	bootload/module
package/1	bootload/module
pi/1	arithmetic/trigo
predicate_property/2	reflect/pred
(private)/1	bootload/module
prolog/0	runtime/session
(public)/1	bootload/module
rebuild/0	bootload/load
reexport/1	bootload/module

reference/1	structure/type
rem/3	arithmetic/round
repeat/0	runtime/logic
reset_atom_property/3	reflect/pred
reset_oper_property/2	reflect/oper
retract/1	runtime/dynamic
retractall/1	runtime/dynamic
round/2	arithmetic/round
set_arg/4	structure/univ
set_atom_property/3	reflect/pred
set_prolog_flag/2	bootload/engine
setup_balance/1	runtime/distributed
setup_balance/2	runtime/distributed
sign/2	arithmetic/elem
sin/2	arithmetic/trigo
source_property/2	reflect/source
sqrt/2	arithmetic/trigo
(static)/1	reflect/pred
sub_atom/4	structure/atom
sub_atom/5	structure/atom
sys_activate_capability/2	bootload/toolkit
sys_add_path/1	bootload/path
sys_add_resource/1	bootload/module
sys_auto_load/1	bootload/module
sys_calc_install_id/2	bootload/toolkit
sys_callable/1	runtime/quali
sys_capability_property/2	bootload/toolkit
sys_check_license/1	bootload/toolkit
sys_check_licenses/0	bootload/toolkit
sys_current_capability/1	bootload/toolkit
sys_current_path/1	bootload/path
sys_current_provable/1	bootload/load
sys_current_source_site/2	reflect/source
sys_current_syntax/1	bootload/load
sys_declaration_indicator/2	reflect/foreign
sys_declaration_indicator/2	reflect/pred
sys_declaration_indicator/2	bootload/load
sys_declaration_indicator/2	bootload/module
sys_declaration_indicator/2	runtime/meta
sys_declaration_indicator/2	runtime/dynamic
sys_finit_capability/1	bootload/toolkit
sys_functor/3	runtime/quali
sys_get_variable_names/1	structure/vars
sys_goal_globals/2	structure/vars
sys_goal_kernel/2	structure/vars
sys_init_capability/1	bootload/toolkit
sys_init_capability/2	bootload/toolkit
sys_load_resource/1	bootload/module
sys_make_indicator/3	reflect/pred
sys_make_oper/3	reflect/oper
sys_module_site/2	bootload/path
sys_number_variables/4	structure/vars
sys_provable_property_chk/3	bootload/load
sys_reg_license_text/2	bootload/toolkit
sys_show_base/1	bootload/load
sys_show_provable_source/2	bootload/load

sys_show_syntax/1	bootload/load
sys_show_vars/0	runtime/session
sys_syntax_property_chk/3	bootload/load
sys_term_singletons/2	structure/vars
sys_univ/2	runtime/quali
sys_var/1	runtime/quali
tan/2	arithmetic/trigo
term_variables/2	structure/vars
term_variables/3	structure/vars
(thread_local)/1	runtime/dynamic
truncate/2	arithmetic/round
unify_with_occurs_check/2	structure/univ
unload_file/1	bootload/load
use_file_extension/1	bootload/module
use_module/1	bootload/module
use_package/1	bootload/module
version/0	runtime/session
welcome/0	runtime/session
xor/3	arithmetic/bits

Package Local Predicates

Predicate	**Module**
sys_load_file/2	bootload/load
sys_register_file/1	bootload/load

Non-Private Meta-Predicates

Predicate	Exp	Body	Rule	Module
0*->0	yes	yes	no	runtime/logic
0,0	yes	yes	no	runtime/logic
0->0	yes	yes	no	runtime/logic
? :0	yes	no	no	runtime/quali
0;0	yes	yes	no	runtime/logic
\+0	yes	no	no	runtime/logic
asserta(-1)	yes	no	no	runtime/dynamic
assertz(-1)	yes	no	no	runtime/dynamic
balance(0)	yes	no	no	runtime/distributed
balance(0,?)	yes	no	no	runtime/distributed
call(0)	yes	no	no	runtime/logic
clause(-1,0)	no	no	no	runtime/dynamic
horde(0)	yes	no	no	runtime/distributed
horde(0,?)	yes	no	no	runtime/distributed
once(0)	yes	no	no	runtime/logic
retract(-1)	no	no	no	runtime/dynamic
retractall(-1)	no	no	no	runtime/dynamic
setup_balance(0)	yes	no	no	runtime/distributed
setup_balance(0,?)	yes	no	no	runtime/distributed

Non-Private Closure-Predicates

Predicate	**Module**

:(?,1,?)	runtime/quali
? :: ::(0)	runtime/quali
::(?,::(1),?)	runtime/quali
#(1)< #(1)	arithmetic/compare
#(1)=:= #(1)	arithmetic/compare
#(1)=< #(1)	arithmetic/compare
#(1)=\= #(1)	arithmetic/compare
#(1)> #(1)	arithmetic/compare
#(1)>= #(1)	arithmetic/compare
?is#(1)	arithmetic/eval

Non-Private Syntax Operators

Level	Mode	Operator	Module
1200	fx	?-	runtime/logic
1150	fx	thread_local	runtime/dynamic
1150	fy	override	bootload/load
1150	fy	public	bootload/module
1150	fx	meta_predicate	runtime/meta
1150	fx	meta_function	runtime/meta
1150	fx	dynamic	runtime/dynamic
1150	fy	discontiguous	bootload/load
1150	fy	multifile	bootload/load
1150	fy	private	bootload/module
1150	fx	static	reflect/pred
1105	xfy	|	runtime/logic
1100	xfy	;	runtime/logic
1050	xfy	->	runtime/logic
1050	xfy	*->	runtime/logic
900	fy	\+	runtime/logic
700	xfx	@>	structure/lexical
700	xfx	=	structure/univ
700	xfx	=..	structure/univ
700	xfx	=:=	arithmetic/compare
700	xfx	<	arithmetic/compare
700	xfx	==	structure/lexical
700	xfx	=<	arithmetic/compare
700	xfx	@=<	structure/lexical
700	xfx	\=	structure/univ
700	xfx	@<	structure/lexical
700	xfx	=\=	arithmetic/compare
700	xfx	\==	structure/lexical
700	xfx	@>=	structure/lexical
700	xfx	is	arithmetic/eval
700	xfx	>=	arithmetic/compare
700	xfx	>	arithmetic/compare
600	xfy	:	runtime/quali
600	xfy	::	runtime/quali
500	yfx	+	arithmetic/elem
500	yfx	/\	arithmetic/bits
500	yfx	-	arithmetic/elem
500	yfx	\/	arithmetic/bits
400	yfx	*	arithmetic/elem
400	yfx	//	arithmetic/round
400	yfx	xor	arithmetic/bits

400	yfx	>>	arithmetic/bits
400	yfx	div	arithmetic/round
400	yfx	<<	arithmetic/bits
400	yfx	rem	arithmetic/round
400	yfx	mod	arithmetic/round
200	fy	+	arithmetic/elem
200	fy	\	arithmetic/bits
200	fy	-	arithmetic/elem
200	xfy	^	arithmetic/elem
200	xfx	**	arithmetic/trigo
100	yf	[]	arithmetic/eval

Pictures

Es konnten keine Einträge für ein Abbildungsverzeichnis gefunden werden.

Tables

Acronyms

DEC10	[1]
ISO	[2]
	[3]
UNID	[4]
TC2	[6]

References

[1] Bowen, D.L. et al. (1982): DECsystem-10 PROLOG USER'S MANUAL, Department of Artificial Intelligence University of Edinburgh, 10 November 1982
http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/prolog/doc/intro/prolog.doc

[2] ISO (1995): Prolog, Part 1: General Core, International Standard ISO/IEC 13211-1, First Edition, 1995-06-01

[3]

[4] Richard O'Keefe (2011): An Elementary Prolog Library, Draft 8, November 19, 2010
http://www.complang.tuwien.ac.at/ulrich/iso-prolog/pllib-2010-11-19.htm

[5] Pereira, F.C.N. and Warren, D.H.D. (1980): Definite Clause Grammars for Language Analysis – A Survey of the Formalism and a Comparison with Augmented Transition Networks, North-Holland Publishing Company, Artificial Intelligence, 13, 231 – 278
http://cgi.di.uoa.gr/~takis/pereira-warren.pdf

[6] ISO (2012): Prolog, Part 1: General Core, International Standard ISO/IEC 13211-1, Technical Corrigendum 2, 2012-02-15

[7] Costa, V.S., Rocha, R. and Damas, L. (2011): The YAP Prolog System, Theory and Practice of Logic Programming, 2011
http://arxiv.org/abs/1102.3896

[8] Bueno F. et al. (2006): The Ciao Prolog System, Reference Manual, 2006
http://www.ciaohome.org/Legacy/ciao.pdf

[9] Covington, M.A., Bagnara, R., O'Keefe, R.A., Wielemaker, J. and Price, S. (2011): Coding guidelines for Prolog, 2011
http://arxiv.org/abs/0911.2899

[10] Wielemaker, J. (2014): SWI-Prolog Version 7 Extensions, Jan Wielemaker, Web and media Group, VU University Amsterdam, The Netherlands, 2014
http://www.swi-prolog.org/download/publications/swi7.pdf