Jekejeke Runtime Compliance

Version 1.1.3, May 1st, 2016

XLOG Technologies GmbH

Jekejeke Prolog

Runtime Library 1.1.3

Compliance Results

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

Date: May 1st, 2016
Version: 0.23

Participants: None

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, October 2nd, 2010, 0.1:
- Initial version.

Jan Burse, November 14th, 2010, 0.2:
- Arithmetic re-evaluated and trigonometric operation test cases introduced.

Jan Burse, November 18th, 2010, 0.3:
- Atom and number syntax test cases updated.

Jan Burse, November 28th, 2010, 0.4:
- Character test cases added.

Jan Burse, December 2nd, 2010, 0.5:
- Steam control test cases added.

Jan Burse, January 2nd, 2011, 0.6:
- Omissions section updated and sections reordered.

Jan Burse, April 15th, 2011, 0.7:
- Omissions and discrepancies updated to version 0.8.8.

Jan Burse, Mai 15th, 2011, 0.8:
- Discrepancies updated to version 0.8.9.

Jan Burse, June 15th, 2011, 0.9:
- Byte I/O test cases introduced.

Jan Burse, August 15th, 2011, 0.10:
- String test cases enhanced and terminal based diagnosis application introduced.

Jan Burse, August 20th, 2011, 0.11:
- Draft technical corrigendum 2 test cases introduced and syntax omissions documented.

Jan Burse, September 23th, 2011, 0.12:
- New setup_call_cleanup/3 test cases introduced.

Jan Burse, November 4th, 2011, 0.13:
- Portable test helper introduced and results updated to version 0.9.2.

Jan Burse, April 5th, 2012, 0.14:
- Results updated to version 0.9.3.

Jan Burse, June 5th, 2012, 0.15:
- Results updated to version 0.9.4.

Jan Burse, October 6th, 2012, 0.16:
- New simplify glitch discrepancy documented.

Jan Burse, November 20th, 2012, 0.17:
- Results updated to version 0.9.7.

Jan Burse, December 13th, 2013, 0.18:
- Results updated to version 0.9.12.

Jan Burse, July 23rd, 2014, 0.19:
- Results updated to version 1.0.3.

Jan Burse, August 11th, 2014, 0.20:
- Test results moved to discrepancies section and test cases combined into bigger files.

Jan Burse, January 24th, 2016, 0.21:
- Test results adapted to new automatic bridging and tunnelling.

Jan Burse, March 4th, 2016, 0.22:
- Results updated to version 1.1.2.

Jan Burse, May 1st, 2016, 0.23:
- Document moved to development environment and shared modules used.

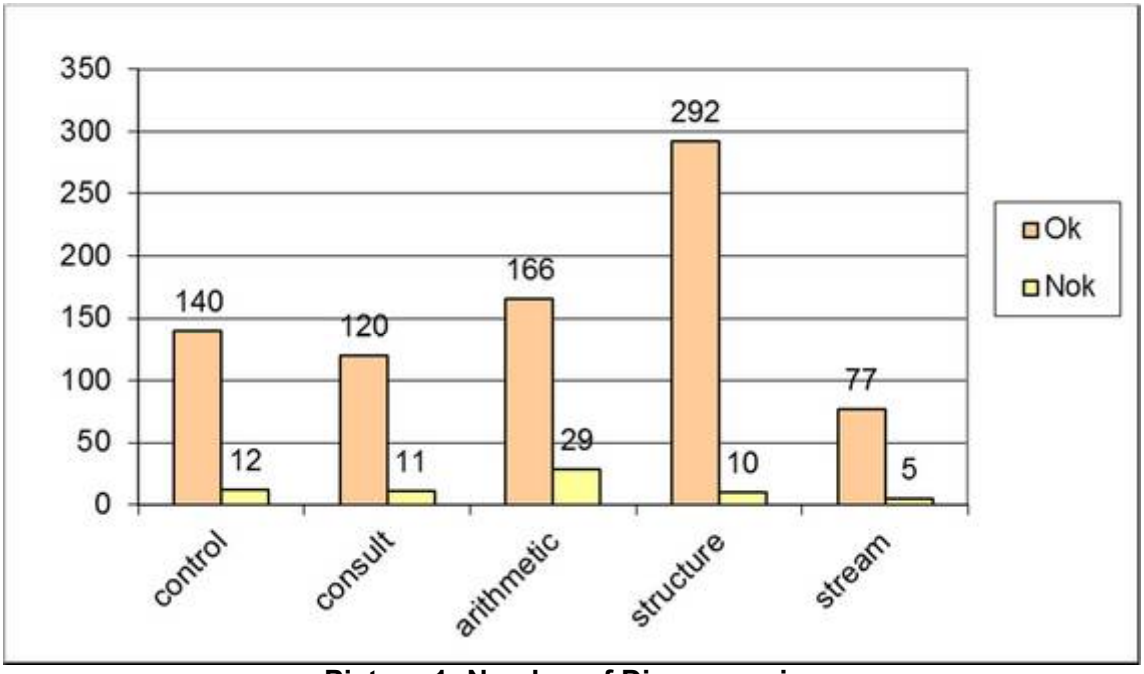
1 Introduction

This document describes some compliance results for the Jekejeke Prolog system.

- **ISO Core Discrepancies:** We will present and discuss the findings.
- **ISO Core Omissions:** We will list the unimplemented requirements.
- **Test Setup:** We explain our test scope, method and sources.
- **Appendix Diagnosis Listings:** We will list the diagnosis terminal application.
- **Appendix Test Case Listings:** We will list the test cases and their test results.

2 Discrepancies

We will present and discuss the findings. We gathered 862 test cases from the standard examples and through our own development. In the average around 7.8% of the test cases failed. Relatively seen the most test cases failed in the arithmetic theory with 14.9%. The fewest test cases failed in the structure theory with 3.3%. The results are very promising, since they show that we would not have a very long path ahead for full ISO compliance.



Picture 1: Number of Discrepancies

There were no test case failures which we could not explain. We have analysed the failures and come up with a set of findings. For each finding we have compared the Jekejeke Prolog behaviour with the behaviour mandated by the ISO standard. We have then classified the Jekejeke Prolog behaviour into the categories enhancement or limitation. In summary the following findings could be collected from our testing:

Table 1: Identified Findings

Theory	Title	Classification
Control	Predicate Visibility	Enhancement
	Error Message	Limitation
	Clean-up Safety	Enhancement
	Simplify Glitch	Limitation
Consult	Predicate Sealing	Enhancement
Arithmetic	Narrower Arithmetic	Limitation
	Broader Arithmetic	Enhancement
Structure	Array Access	Enhancement
Stream	Stream Property	Enhancement

The findings that were classified a limitation need further work by us. They will be probably fixed in an upcoming release of Jekejeke Prolog. The findings that were classified an enhancement will only be worked on, when we have introduce an ISO compatibility flag. This flag will then either allow the Jekejeke Prolog specific behaviour or it will revert to the ISO compliant behaviour. The ISO compliant behaviour will also need further work by us.

We will present our findings grouped according to our packages:

2.1 Package Control

The below table shows a breakdown of the test results for the control package:

Table 2: Control Discrepancies

Module	Cases	Ok	Nok	Comment
pred	7	6	1	Predicate Visibility
kernel	40	40	0	
logical	68	63	5	Simplify Glitch Error Message
signal	37	31	6	Clean-up Safety
Total	152	140	12	

The main findings for the control package were:

- Predicate Visibility
- Error Message
- Clean-up Safety
- Simplify Glitch

Predicate Visibility

Classification Enhancement

Discrepancy The ISO standard requires that the built-in current_predicate/1 finds all predicates, static or dynamic, with or without clauses that were defined by the user. Such a predicate might have some applications in a listing/0 predicate that should only show the user defined clauses.

In our implementation the built-in is capable of enumerating and checking all possibly qualified predicates that are visible from the call-site. This allows for example checking the existence and accessibility of a predicate before it is called, accessed or modified.

In our implementation the visibility of a predicate depends on its reachability via import and export and on its visibility attributes. Non-qualified predicates are visible everywhere. In as far there is no difference between user and non-user defined predicates.

Integration / Elimination The ISO module standard defines a predicate current_visible/1. The intention of our built-in current_predicate/1 is to function similarly to this predicate. The predicate cannot enumerate private predicates or package local predicates when the call-site is a different package.

For an unrestricted enumeration and checking we provide a module in the development environment. The module is inspection/provable and the predicate current_provable/1 does the job.

Failed Test Cases ISO 8.8.2.4, ISO 2

Related Findings -

Error Message

Classification Limitation

Discrepancy The Jekejeke Prolog provides the same type errors as ISO Prolog. But Jekejeke Prolog only shows the sub term as a culprit when a type error occurs. But the ISO examples usually show the full term as a culprit. Further if a predicate has output parameters, we do not check their types. Instead we simply compute the output value and leave it to unification.

There are further problems with our error handling. The ISO core standard does not really mandate an order in checking the predicate arguments, but some of the examples imply a certain order. We should respect the same order in our test cases.

Also the ISO core standard provides a couple of internal data types that are derived from the usual Prolog data types. Among these we find the character which is an atom of length 1, and the byte which is an integer in the range 0 … 255. We do not yet correctly check these types, we first check for atom resp. integer, and then for the additional condition, yielding two different errors. The ISO core standard demands only one error.

Integration / Elimination One reason for our simpler error messages is implementation effort and execution performance. Sub term error messages are easier to generate on the fly. They also give better information than argument based error messages. Omitting checking given output values results in higher execution speed.

Concerning the predicate arguments, we should check the error handling in all our predicates, and align it in case the usual order is not respected.

Further the error checking for character and byte should be fixed, so that it does not split up into an atom resp. integer checking, and a further additional condition checking.

Failed Test Cases

ISO 7.8.3.4, ISO 6b
ISO 7.8.3.4, ISO 13
ISO 7.8.3.4, ISO 14
ISO 7.8.3.4, ISO 15
ISO 8.14.3.4, ISO 4
ISO 8.14.3.4, ISO 8
ISO 8.14.3.4, ISO 10
Corr.2 8.15.4.4, XLOG 3
Corr.2 8.15.4.4, XLOG 4
ISO 9.1.7, ISO 10
ISO 9.1.4, XLOG 5
ISO 9.1.7, ISO 5
ISO 9.1.7, ISO 20
ISO 9.1.7, ISO 28
ISO 9.1.7, ISO 9
Corr.2 9.1.3, XLOG 4
ISO 9.1.7, ISO 34
ISO 9.3.1.4, ISO 5
ISO 9.3.2.4, ISO 4
ISO 9.3.3.4, ISO 4
Corr.2 9.3.14.4, XLOG 3
Corr.2 9.3.12.4, XLOG 4
ISO 9.3.5.4, ISO 5
ISO 9.3.6.4, ISO 6
ISO 9.4.3.4, ISO 6
ISO 8.7.1.4, XLOG 1
ISO 8.7.1.4, XLOG 3
ISO 8.7.1.4, XLOG 4
ISO 8.7.1.4, XLOG 5
Corr.2 9.3.9, XLOG 1
ISO 8.5.3.4, ISO 9
ISO 8.5.1.4, ISO 16
ISO 8.5.1.4, ISO 17
ISO 8.5.2.4, ISO 11
ISO 8.16.1.4, ISO 7
Corr.2 8.4.4.4, XLOG 7
Corr.2 8.4.3.4, XLOG 6
ISO 8.13.3, ISO 4

Related Findings

Clean-up Safety

Classification Enhancement

Discrepancy The behaviour of the predicate setup_call_cleanup/3 is described in [10]. Our implementation is based on a dissection of this behaviour into the system predicates sys_atomic_call/1 and sys_on_cleanup/1. The implementation needs the newly introduced cutter mechanism.

During the implementation here and then we did not closely follow the ISO proposal. At the moment the following behavioural discrepancies are known:

- Clean-up pre-validation, not implemented.
- Clean-up exception accumulation, not part of ISO proposal.
- Failure of clean-up throws exception, not part of ISO proposal.

Integration / Elimination Some of the discrepancies have simple workarounds. If pre-validation is needed one could invoke setup_call_cleanup/3 as follows:

```
?- setup_call_cleanup((S,
    (var(C) -> sys_throw_error(instantiation_error);
     \+ callable(C) -> sys_throw_error(
             type_error(callable, C));
     true)), G, C).
```

If no exception accumulation is needed one could invoke the setup_call_cleanup/3 as follows:

```
throw_cause(cause(_,E)) :- !, throw_cause(E).
throw_cause(E) :- throw(E).

?- catch(setup_call_cleanup(S, G, C), E, throw_cause(E))).
```

If no exception on failure is needed one could invoke the setup_call_cleanup/3 as follows:

```
?- setup_call_cleanup(S,G, (C;true)).
```

Failed Test Cases

WG17 N215, ISO 3	WG17 N215, ISO 27
WG17 N215, ISO 25	WG17 N215, ISO 9
WG17 N215, ISO 26	WG17 N215, XLOG 12

Related Findings -

Simplify Glitch

Classification Limitation

Discrepancy Since release 0.9.4 of Jekejeke Prolog the interpreter features a simplification framework. Simplification is directly applied after the expansion of a goal or term. It can be customized by the end-user.

It now happens that certain rules can cause problems in the semantics of the if-then-else. This happens whenever the simplification generates a free standing (->)/2 term inside a (;)/2 context. The following innocently looking simplification rule already causes this problem:

```
A, true ~~> A
```

When the simplification acts on (! -> fail), true; true, it will turn the goal into ! -> fail; true. The former goal succeeds whereas the latter goal fails. The scope of the local cut inside the (->)/2 is broadened by the simplification, which causes the discrepancy in the semantics.

Integration / Elimination The simplification is vital to the Jekejeke Runtime and Jekejeke Minlog. It is internally used to simplify DCG grammar rules and to simplify forward chaining handlers. So we will need to keep this feature.

We did not yet identify all corners where such a glitch can happen. And for those cases where we already see the glitch, we do not yet have a definite concept to fix the simplification rules. For the already mentioned simplification rule a fix would be eventually:

```
A, true ~~> A      if    A \= (_ -> _)
```

But this also prevents simplification when the potential free standing (->)/2 is not inside a (;)/2 context. Thus reducing the number of simplified goals and therefor lowering the benefit of simplification.

Failed Test Cases ISO 7.8.8.4, ISO 9

Related Findings

2.2 Package Consult

The below table shows a breakdown of the test results for the consult package:

Table 3: Consult Discrepancies

Module	Cases	Ok	Nok	Comment
file	12	6	6	Error Message Predicate Sealing
data	45	42	3	Predicate Sealing
apply	18	16	2	Error Message
dcg	56	56	0	
Total	131	120	11	

The new main findings for the control package were:

- Predicate Sealing

Predicate Sealing

Classification Enhancement

Discrepancy The ISO standard requires the existence of a multi-file directive. But it leaves open how such a directive is implemented. To allow separate compilation we only allow predicate attribute transitions from undefined to defined, and we warn so that for multi-file predicates different Prolog text mentions repeat all the predicate attributes.

We also apply this approach to syntax operators. As a side effect the level or mode of a syntax operator cannot be changed when it has been set once without first abolishing it. The same holds for example for the comma operator (‘,’) as well, but the error message is different than what has recently been defined by the ISO standard.

To prevent the end-user from modifying non-user predicates we have devised the more general rule that non-multi-file predicates cannot be modified in multiple Prolog texts. Again the error message is different than what has recently been defined by the ISO standard.

Integration / Elimination We value the different error messages as non-severe. They express the different first principles that cause the same test cases to fail as the ISO standard requires.

On the other hand in our implementation the end-user is less free to modify attributes of syntax operators and predicates. Further in our implementation the end-user is forced to mark a predicate as multi-file if he wants to add clauses from within different Prolog texts.

Failed Test Cases ISO 8.14.3.4, ISO 4
Corr.2 6.3.4.3, ISO 3
ISO 8.9.1.4, ISO 7
ISO 8.9.2.4, ISO 7
ISO 8.9.4.4, ISO 5

Related Findings t.b.d.

2.3 Package Arithmetic

The below table shows a breakdown of the test results for the arithmetic package:

Table 4: Arithmetic Discrepancies

Module	Cases	Ok	Nok	Comment
basic	53	43	10	Narrower Arithmetic
round	32	28	4	
trigo	51	42	9	Broader Arithmetic
bitwise	21	20	1	
compare	38	33	5	
Total	195	166	29	

The new main findings for the arithmetic package were:

- Narrower Arithmetic
- Broader Arithmetic

Narrower Arithmetic

Classification Limitation

Discrepancy Jekejeke Prolog keeps some evaluable functions in a narrower scope. In particular the arithmetic function (^)/2 is only defined for a non-negative exponent and a non-float result. A broader range doesn't make sense since it is already covered by (**)/2 and a higher precision can be hardly archived.

Integration / Elimination For non-integer arguments or for a negative exponent, one can regress to the evaluable function (**)/2.

Failed Test Cases
Corr.2 9.3.10.4, ISO 2
Corr.2 9.3.10.4, ISO 7
Corr.2 9.3.10.4, ISO 9
Corr.2 9.3.10.4, XLOG 1
Corr.2 9.3.10.4, XLOG 2

Related Findings -

Broader Arithmetic

Classification Enhancement

Discrepancy Recently we have further extended the (^)/2 operator not only to apply to an integer basis but also to a decimal basis which will give a decimal result. A float basis is automatically promoted to a decimal.

Jekejeke Prolog defines some evaluable functions with a broader scope. In particular the arithmetic functions (mod)/2 and (rem)/2 are not only defined for integers, but also on floats. Nowadays this is in already found in many programming languages. For example Java has extended the mod operator % towards floats.

Integration / Elimination The enhancement of evaluable functions in that previously exceptional values are now defined is allowed by ISO (5.5.10). Only in the strict conforming mode (by the note referring to 5.1e) we would run into problems and would need to remove the additional functionality. We could stash the functionality to new evaluable functions (fmod)/2 and (frem)/2.

Failed Test Cases Corr.2 9.3.10.4, XLOG 3 ISO 9.1.7, ISO 35
Corr.2 9.3.10.4, XLOG 4:

Related Findings -

2.4 Package Structure

The below table shows a breakdown of the test results for the structure package:

Table 5: Structure Discrepancies

Module	Cases	Ok	Nok	Comment
type	55	55	0	
lexical	22	22	0	
term	90	85	5	Error Message Array Access
string	67	66	1	Error Message
set	68	64	4	Error Message
Total	302	292	10	

The new main findings for the structure package were:

- Array Access

Array Access

Classification Enhancement

Discrepancy The Jekejeke Prolog supports the built-ins functor/3 and arg/3. These predicates can be used to dynamically created and access compound. This is a simple substitute for arrays. The functor predicate already supports the view that atoms are compounds of arity zero. Unfortunately according to the ISO core standard, accessing atoms is not allowed.

Integration / Elimination If the ISO core standard requires this behaviour we can adapt our code easily. Unfortunately in our current implementation we do not yet have a compatibility switch. Here it would make sense to have such a switch.

The switch would also need to adapt the thrown exception. Consequently since our current arg/3 implementation also accepts atoms, we have changed the exception from type error compound to type error callable.

Failed Test Cases ISO 8.5.2.4, ISO 10

Related Findings -

2.5 Package Stream

The below table shows a breakdown of the test results for the stream package:

Table 6: Stream Discrepancies

Module	Cases	Ok	Nok	Comment
char	27	27	0	
byte	15	12	3	Error Message
read	31	30	1	
open	9	8	1	Stream Property
Total	82	77	5	

The new main findings for the stream package were:

- Stream Property

Stream Property

Classification Enhancement

Discrepancy The ISO core standard defines a predicate stream_property/2 that can enumerate all current streams. In Jekejeke Prolog it is only possible to inspect an existing stream or aliases, but not to enumerate streams.

Integration / Elimination Jekejeke Prolog provides streams to resources addressed by an URL. Such streams could be created in high number by multi-threaded applications. For efficiency and security reasons they should not be enumerable.

Stream references can be stored in clauses. Therefore if desired Prolog applications can create their own dynamic predicates to manage their streams and make their streams enumerable.

Failed Test Cases ISO 8.11.8, XLOG 1

Related Findings -

3 Omissions

We will list the unimplemented requirements. We did not yet follow the complete ISO core standard syntax. And not all predicates and evaluable functions from the ISO core standard have been implemented yet in the Jekejeke Prolog system. Therefore we find the following omissions:

3.1 Syntax Omissions

The following syntax is currently not followed in Jekejeke Prolog 1.1.2:

Table 7: Syntax Omissions

Syntax	Source	Examples
Empty set forbidden operator.	ISO Draft Corrigendum 2.	TC2 6.3.4.3
Empty list forbidden operator.	ISO Draft Corrigendum 2.	TC2 6.3.4.3

3.2 Predicate Omissions

The following predicates are currently not found in Jekejeke Prolog 0.9.7:

Table 8: Predicate Omissions

Predicate	Source	Examples
current_char_conversion/2	ISO Core Standard.	ISO 8.14.6
char_conversion/2	ISO Core Standard.	ISO 8.14.5
initialization/1	ISO Core Standard.	- not found -
subsumes_term/2	ISO Draft Corrigendum 2.	TC2 8.2.4.4
acyclic_term/1	ISO Draft Corrigendum 2.	TC2 8.3.11.4
retractall/1	ISO Draft Corrigendum 2.	TC2 8.9.5.4

3.3 Evaluable Function Omissions

The following evaluable functions are currently not found in Jekejeke Prolog 1.1.2:

Table 9: Evaluable Function Omissions

Evaluable Function	Source	Examples
float_integer_part/1	ISO Core Standard	- not found -
float_fractional_part/1	ISO Core Standard	- not found -

4 Test Setup

In this section we explain our test setup:

- **Test Scope:** In this section we explain what we will test. The aim of our testing is to show compliance with the ISO standard. Since the ISO standard does not cover all parts of Prolog systems, we will also not test all subsystems of our Prolog system.

- **Test Method:** We will present our test method. It is based on creating a number of test cases for each predicate in each theory. A test runner will then execute the test cases and summarize the results.

- **Test Sources:** We will give credit to the sources of our test cases. The Prolog ISO standard movement consists of some documents that have already been adopted by the ISO. Further there exists a working group that is involved in the definition of eventual supplements.

- **Test Harness:** The test harness is written in Prolog itself. It consists of a test runner and a test result browser, as well as a report generator.

- **Test Cases:** The test cases are written in Prolog itself. They consist of positive and negative test cases according to the sources.

4.1 Test Scope

In this section we explain what we will test. The aim of our testing is to show compliance with the ISO standard. Since the ISO standard does not cover all parts of Prolog systems, we will also not test all subsystems of our Prolog system. We will only test the following sub systems of Jekejeke Prolog 1.1.1:

- The Jekejeke Prolog language (Runtime Library) [4].

The following sub systems are not tested:

- The Jekejeke Prolog console [5].
- The Jekejeke Prolog programming interface [6].
- The Jekejeke Prolog language (Development Environment).

The Jekejeke Prolog language consists of some basic concepts, of the syntax of Prolog texts and queries and of Prolog text that define various artefacts. Among the defined artefacts we find predicate, evaluable functions, exceptions, flags and properties. We will only test on the level of predicates and evaluable functions.

For each tested predicate or evaluable function a number of test cases are defined. To reach our goal of showing compliance with the ISO standard, we only pick those predicates and evaluable functions which are also covered by the ISO standard. This means that we have only to consider some of our Prolog texts. The picked predicates and evaluable functions have been grouped into the following packages:

- Package Control
- Package Consult
- Package Arithmetic
- Package Structure
- Package Stream

4.2 Test Method

Our test method is a mix of test utilities that are bundled with the Jekejeke Prolog development environment plus some test helper predicates already available in the Jekejeke Prolog runtime library. The test utilities used are as follows. For further information on these modules the interested reads should consult the corresponding API documentation of the Jekejeke Prolog development environment:

- **Module testing/runner:** This module allows executing test cases.
- **Module testing/diagnose:** This module allows online display of test results.
- **Module testing/report:** This module allows batch reporting of test results.

The chosen test method will increase the number of test cases compared to the number of examples given in the test case sources. The reason is that the examples often phrase multiple validations points. And we suggest using a separate test_case/4 clause for each validation point. Let's make an example. Take one of the ISO examples for clause/2. We find the following description in the ISO source:

```
:- dynamic(insect/1).
:- assertz(insect(ant)).
:- assertz(insect(bee)).

clause(insect(I), T).
      Succeeds, unifying I with ant, and T with true.
      On re-execution,
      succeeds, unifying I with bee, and T with true.
```

A succeed or fail on the first call translates very directly into a test case. We simply need to place the tested predicate invocation as a step plus the required result comparison as a validation point into one test_case/4 clause. For success or failure after the first call we apply the findall/3 built-in. This will allow us to advance the found solutions in the tested predicate invocation. It can then be again followed by the required result comparison as a validation point. In our current example this will give our second test_case/4 clause:

```
test_case(clause, 2, consult, 1) :- clause(insect(I), T), !,
      I==ant, T==true.
test_case(clause, 2, consult, 2) :- findall(I-T,
      clause(insect(I), T), [_,I-T|_]), I==bee, T==true.
```

The use of the findall/3 built-in in separate test cases means that the tested predicate is invoked again. So our approach is not the most efficient. On the other hand the approach has the advantage that it can disclose individual failures of validation points separately.

4.3 Test Sources

We will give credit to the sources of our test cases. The Prolog ISO standard movement consists of some documents that have already been adopted by the ISO. Further there exists a working group that is involved in the definition of eventual supplements. The documents currently adopted by ISO contain more or less a systematic set of example uses of the predicates and evaluable functions. These examples form the basis for our test cases.

The Jekejeke Prolog system does not yet implement all of the documents adopted by the ISO. For example we do not yet implement the part 2 of the Prolog ISO standard that deals with modules. On the other hand we have already implemented some documents that have not been adopted by the ISO standard. Among these we find the definite clause grammars draft and the draft technical corrigendum 2.

In summary the documents that have been respected in our test cases are:

- Prolog, Part 1: General Core, ISO/IEC 13211-1 [1]
- Prolog, Part 1: General Core, Technical Corrigendum 1, ISO/IEC 13211-1 [2]
- Prolog, Part 1: General Core, Draft Technical Corrigendum 2, ISO/IEC 13211-1 [8]
- Prolog, Part 1: General Core, Draft Proposal for setup_call_cleanup/3 [10]

The documents that have not yet been respected in our test cases are:

- Prolog, Part 2: Modules, ISO/IEC DTR 13211 2:2000 [7]
- Prolog, Part 3: Definite Clause Grammar Rules, Draft, ISO/IEC DTR 13211 3:2006 [3]

As [9] has already observed there could be test cases that could prevent a test suite from properly functioning. These are test cases that either run infinitely or that produce an exception that is not wrapped by the Jekejeke Prolog system and thus considered to be fatal. In the ISO core standard we find a set of test cases that matches this category. These are test cases that involve cyclic terms. They are marked by the outcome undefined. We have excluded all of these test cases.

Here is an example of an unused test case with cyclic terms:

```
/* X = Y, ISO 8.2.1.4 */

% test(=, 2, structure, _) :- f(X,Y,X,1) = f(a(X),a(Y),Y,2).
```

A further problem we observed was a certain redundancy in the test cases. The ISO standard often contains a test case with named variables and a test case with anonymous variables which are only variants of each other. We run the test suite with the singleton variables check on, so that either of the variants is not accepted. We have only included one variant since we did not see a benefit by going to some lengths to implement both variants as checks.

Here is an example of an unused test case with singleton variables:

```
/* X \= Y, ISO 8.2.3.4 */

% Redundant: test(\=, 2, structure, _) :- \+ X \= Y.
test(\=, 2, structure, 3) :- \+ _ \= _.
```

Some arithmetic examples in the ISO core standard are marked as implementation dependent. We have included them under the assumption that the 2-complement is used and the division (/)/2 uses rounding towards zero. Further we did not include ISO examples that made use of the Prolog flags max_integer or min_integer, since we did not see any use for them in the context of our unbounded integer arithmetic.

4.4 Test Harness

The test harness is written in Prolog itself. It consists of a test runner and a test result browser, as well as a report generator. This document does not contain the source code of the harness programs. A version of the source code of the harness programs can be found on the following web site:

www.jekejeke.ch/idatab/doclet/blog/en/docs/05_run/07_compliance/harness/package.jsp

Further the source code is also bundled in the suprun.zip when downloading the Jekejeke Prolog runtime library from the web site.

4.5 Test Cases

The test cases are written in Prolog itself. They consist of positive and negative test cases according to the sources. This document does not contain the source code of the test cases. A version of the source code of the test cases can be found on the following web site:

www.jekejeke.ch/idatab/doclet/blog/en/docs/05_run/07_compliance/package.jsp

Further the source code is also bundled in the suprun.zip when downloading the Jekejeke Prolog runtime library from the web site.

Pictures

Tables

References

[1] ISO (1995): Prolog, Part 1: General Core, International Standard ISO/IEC 13211-1, First Edition, 1995-06-01

[2] ISO (2007): Prolog, Part 1: General Core, Technical Corrigendum 1, International Standard ISO/IEC 13211-1:1995, 2007-11-15

[3] Moura, P. ed. (2010): Prolog, Part 3: Definite Clause Grammar Rules, Draft, ISO/IEC DTR 13211 3:2006, April 1, 2010

[4] Language Reference, Jekejeke Prolog 0.8.5, XLOG Technologies GmbH, Switzerland, October 2nd, 2010

[5] Console Manual, Jekejeke Prolog 0.8.5, XLOG Technologies GmbH, Switzerland, October 2nd, 2010

[6] Programming Interface, Jekejeke Prolog 0.8.5, XLOG Technologies GmbH, Switzerland, October 2nd, 2010

[7] ISO (2000): Prolog, Part 2: Modules, International Standard ISO/IEC 13211-2, First Edition, 2000-06-01

[8] ISO (2012): Prolog, Part 1: General Core, International Standard ISO/IEC 13211-1, Technical Corrigendum 2, 2012-02-15

[9] Szabo, P. and Szeredi, P.: Improving the ISO Prolog Standard by Analyzing Compliance Test Results, Twenty Second International Conference on Logic Programming, Seattle, August 17 - 20, 2006

[10] Neumerkel, U. ed. (2009): post-N215, Draft Proposal for setup_call_cleanup/3, http://www.complang.tuwien.ac.at/ulrich/iso-prolog/cleanup