Dogelog Player Frequent

Version 1.2.0, March 16, 2024

XLOG Technologies AG

Dogelog Prolog

Dogelog Player 1.2.0

Frequent Predicates

Author: XLOG Technologies AG
Jan Burse
Mittlere Mühlestrasse 2
8598 Bottighofen
Switzerland

Date: March 16, 2024
Version: 0.5

Warranty & Liability

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

Rights & License

Trademarks

Table of Contents

Change history

Jan Burse, August 26, 2022, 0.1:
- Initial version.

Jan Burse, October 02, 2022, 0.2:
- New library(compat) introduced.

Jan Burse, January 26, 2022, 0.3:
- New library(markup) introduced.

Jan Burse, May 08, 2023, 0.4:
- New library(react) introduced.

Jan Burse, December 02, 2023, 0.5:
- Native section moved to separate host document.

1 Introduction

The Dogelog Player does not depend on some essential libraries. Currently the Dogelog Player allows switching between a Dogelog Player specific view of libraries or a dialect specific view of libraries. The later can serve as emulators of other Prolog systems. In the following we deal with the former:

- **Dogelog Tutorials:** We show some use cases of the Dogelog Player specific libraries. We might also show a scenario where a Prolog system is emulated.

- **Dogelog Libraries:** In this section we list the Dogelog Player specific libraries which currently cover data structures, logic constructs and a testing utility.

- **Dogelog Literate:** The Dogelog Player can be also used as a runtime for Prolog notebooks, which is the topic of this section.

2 Dogelog Tutorials

We show some use cases of the Dogelog Player specific libraries. We might also show a scenario where a Prolog system is emulated.

- **Turtle Graphics:** So far, we have used Dogelog's output call back to display text. In this tutorial we use it to display line graphics via turtle graphics.
- **Database Notebook:** To exercise the literate programming webifyer utility we turned half of a Learn Prolog Now! section into a Dogelog notebook.
- **Moon Server:** We describe the implementation of a dynamical HTML page in Prolog that displays the moon phase for a given day.

2.1 Turtle Graphics

So far, we have used Dogelog players output call back to display text. In this tutorial we use it to display line graphics via turtle graphics. We recently added trigonometric functions and can demonstrate Turtle graphics. The precision of 64-bit floating point numbers should be enough for small examples.

- **Turtle Commands:** The turtle commands are found in the Prolog text "turtle.p". This Prolog text is loaded by the main HTML page.
- **SVG Port:** The SVG port is a deviation of the text port that we normally use to show Dogelog player substitution answers.
- **Example Uses:** As an example, we were drawing the Sierpinski triangle. Since the Dogelog player is automatically yielding, we get a little animation.

Turtle Commands

The turtle commands are found in the Prolog text "turtle.p". This Prolog text is loaded by the main HTML page. The turtle commands were written in Prolog itself. The turtle has a state represented as dynamic facts, consisting of the orientation and position of the turtle:

```
:- dynamic current_angle/1.

:- dynamic current_position/2.
```

The turtle commands are turn/1, move/1 and line/1. As a side effect they not only change the turtle state but they might also emit SVG commands. This is the case for the line/1 command which will draw a line on a SVG port:

```
line(D) :-
   retract(current_position(X1, Y1)),
   current_angle(A),
   X2 is X1+D*cos(A),
   Y2 is Y1+D*sin(A),
   assertz(current_position(X2, Y2)),
   line_svg(X1, Y1, X2, Y2).
```

We first planned to include some speed parameter when drawing the turtle. But the current version of Dogelog player yields automatically around 60 times per second which gives a kind of animation by itself, since this yielding is currently clamped at 4ms by the browser. In a future version of this tutorial example, we might do it differently.

SVG Port

The SVG port is a deviation of the text port that we normally use to show Dogelog player substitution answers. Instead that we use a HTML text node, we now use a SVG graphics node. SVG is an XML based graphics format that already exists for a while and has quite some browser support:

```
<!-- p style="..." id="demo"></p -->
<svg id="demo" width="500" height="400"> </svg>
```

To be able to emit SVG commands to the SVG port we have to change the output routine of the Dogelog player slightly. Since the output will be XML, we have to refrain from escaping it. Further it turned out that insertAdjacentHTML() is quite fast:

```
function out(buf) {
    // document.getElementById("demo").innerHTML += xml_escape(buf);
    document.getElementById("demo").insertAdjacentHTML("beforeend", buf);
}
```

For the rest of the HTML page, we have adopted the design of the audio sequencer tutorial example. This means we have an async main function which is also responsible for the automatic yielding of the Prolog engine.

Example Uses

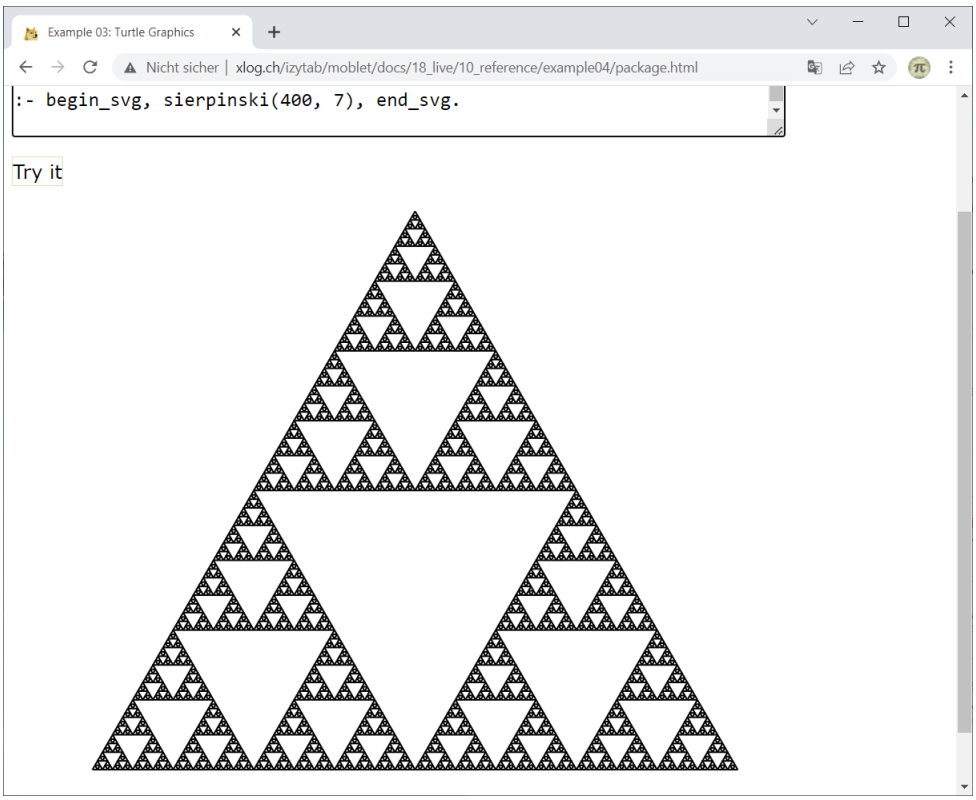
As an example, we were drawing the Sierpinski triangle. The fractal can be drawn with a certain depth L. For depth L=0 only an equilateral triangle is drawn with length D. This can be done with Prolog by means of the turtle commands. The Prolog code is seen here:

```
sierpinski(D, 0) :- !, line(D), turn(-pi*2/3), line(D),
   turn(-pi*2/3), line(D), turn(pi*4/3).
```

For depth L>0 the routine sierpinski/2 will call itself 3 times. It will first call for the left bottom, then for the right bottom and finally for the middle top. Between these invocations we move the turtle without drawing:

```
sierpinski(D, L) :- D1 is D/2, L1 is L-1,
   sierpinski(D1, L1), move(D1), sierpinski(D1, L1),
   turn(-pi*2/3), move(D1), turn(pi*2/3), sierpinski(D1, L1),
   turn(pi*2/3), move(D1), turn(-pi*2/3).
```

Since the Dogelog player is automatically yielding, we get a little animation. In a future version we might even add some more animation, like a little turtle, but here the animation is only the result of successively adding line segments. Here is an example end-result:



Picture 1: Turtle Graphics with Sierpinski L=7

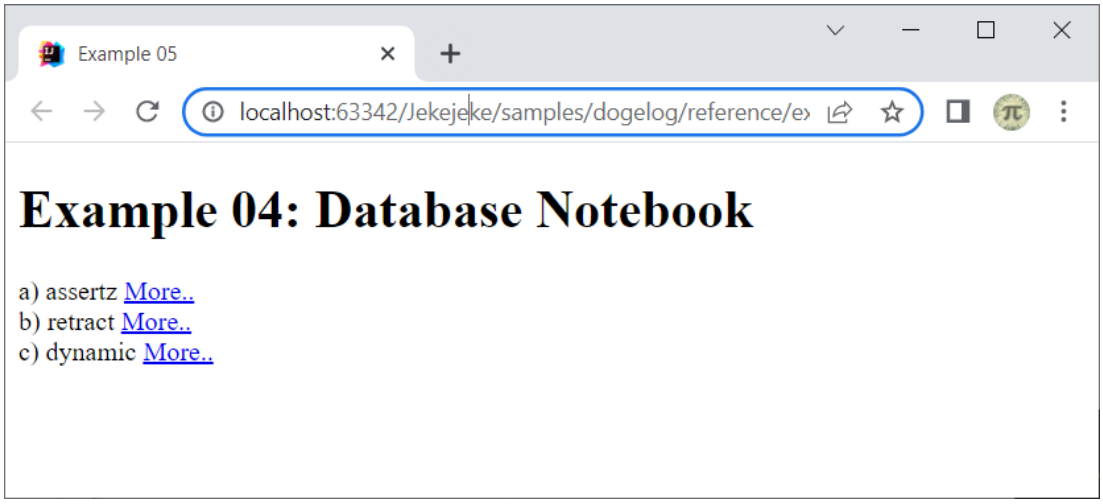
2.2 Database Notebook

To exercise the webifyer utility we turned half of a Learn Prolog Now! section into a notebook. We picked the section 11 .1 Database Manipulation from Learn Prolog Now! as our victim. Since the Dogelog player has rollback of monotonic and non-monotonic Prolog dynamic database changes, the example also highlights this feature.

- **Making Subsections:** We split up the single section HTML page of the online version into subsections.
- **Standard Output:** The Dogelog player notebook page can also handle standard output and will render it below the code cell.
- **Non-Monotonic Updates:** There is no restriction on authoring non-monotonic updates and the presented subsection shows a retract/1 in action.

Making Subsections

We split up the single section HTML page of the online version into subsections. We did so by a single Prolog text index.p with a Prolog HTML comment only that contains some HTML links. That's a little unusual literate programming use, but also in the scope of the webifyer utility to convert a Prolog text that doesn't contain a single code cell.

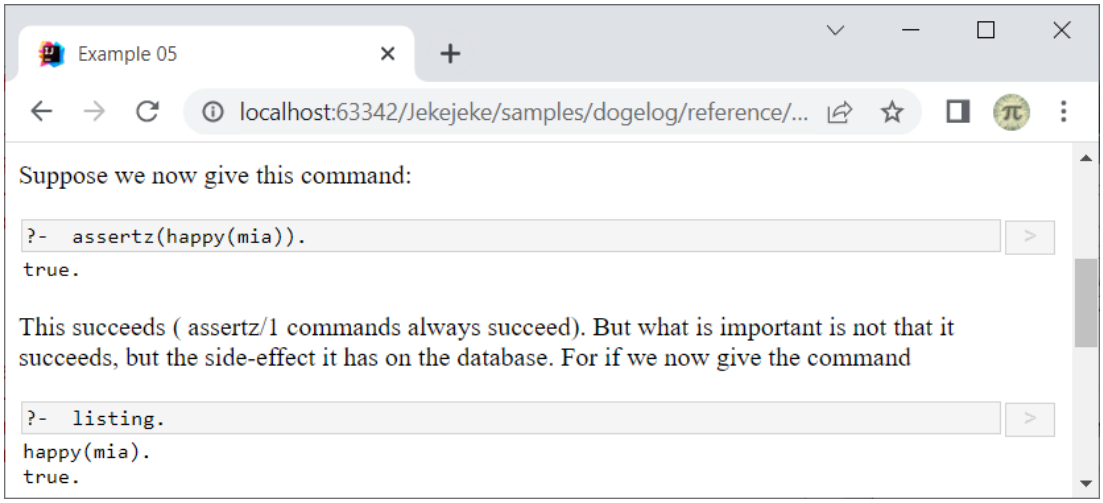


Picture 2: The Produced Sections of the Exercise

We did not convert the paragraphs talking about asserting clauses, asserta/1 and retractall/1 to make the conversion simpler. For consistency we here and then change the text a little. For example, Learn Prolog Now! Expects a Prolog system that returns "yes" for closed answer success, whereas many modern Prolog systems rather use "true" nowadays.

Standard Output

A further challenge was that assert/1 is not an ISO core standard. So, we had to replace the assert/1 by assertz/1. The below screenshot shows the first subsection of the exercise in action. Code cells are generated for facts, rules and queries. The usual output of a query are its answer substitutions which are then exhaustively displayed.



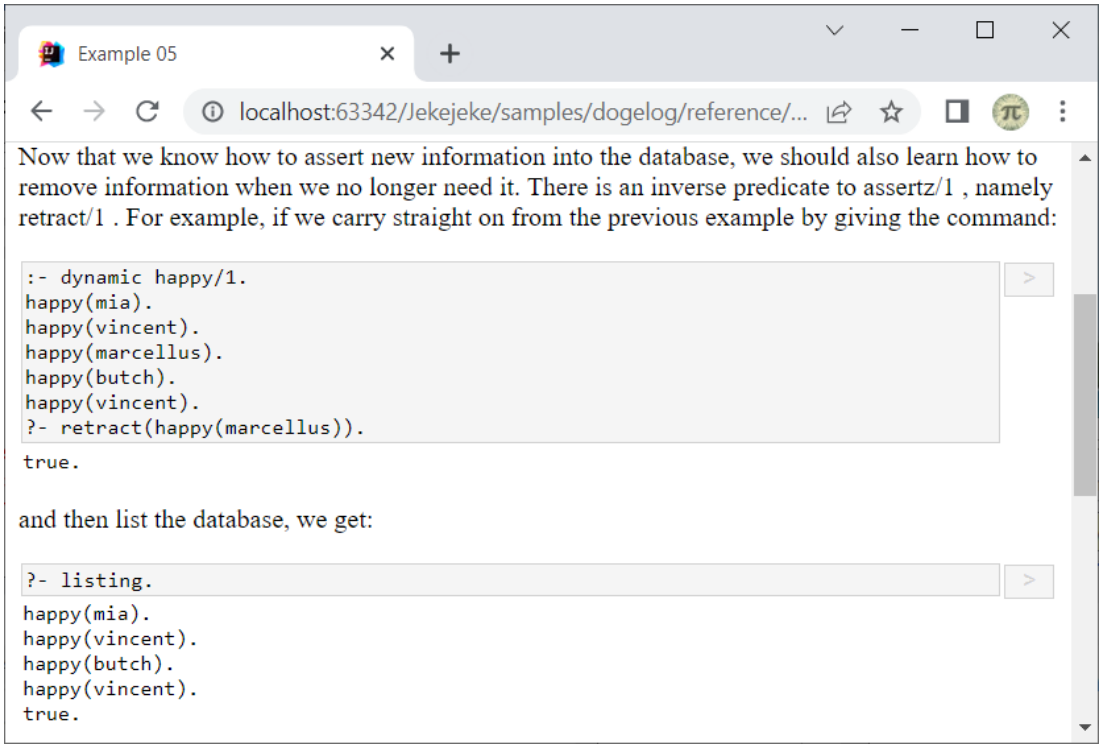
Picture 3: Database Notebook Page with Standard Output

The very beginning of the Learn Prolog Now! section referred to another output, namely the output generated by the built-in predicates listing/[0,1]. The Dogelog player notebook page can also handle standard output and will render it below the code cell. Further listing/[0,1] shows only the user predicates as required by the ISO core standard.

What routinely needed some changes was the presence of expected output in the Learn Prolog Now! text. We removed all expected output since it is anyway generated by the queries in the text. Here and then a query was missing and we added some “?- listing”.

Non-Monotonic Update

We were a little bit unprepared to the fact that the Learn Prolog Now! text uses a reference from the subsection on b) retract to the subsection on a) assert. Since our webifyer utility currently generates independent HTML pages, the state between the page a) and the page b) is not automatically carried over.



Picture 4: Database Notebook Page with Non-Monotonic Update

We helped ourselves in that we put the state into the first code cell. We were lucky that the accompanied Learn Prolog Now! text favours this move by an according phrase. So, we didn't need to make some changes on the Learn Prolog Now! text on this occasion.

There is no restriction on authoring non-monotonic updates and the above subsection shows a retract/1 in action. Dogelog notebook pages also allow tampering by the end-user, for example the argument "marcellus" could be replaced by "butch". The Dogelog notebook will correctly incorporate this change in its result cells.

2.3 Moon Server

We describe the implementation of a dynamical HTML page in Prolog that displays the moon phase seen from the northern hemisphere for a given day. The code demonstrates a HTTP server addition available for the platforms JavaScript, Python and Java.

- **Synodic Month:** We compute the moon phase from epoch time in milliseconds using the length of the synodic month.
- **Server Graphics:** We generate server side continuously varying graphics depending on the given floating point value of the moon phase.
- **HTML Page:** By means of the library(spin) we wrote a HTTP server that does serve two pages /style.css and /moon.cgi.
- **Example Uses:** The server can be run with either Dogelog Player for JavaScript, Python or Java. A browser will show a document with the sunlit part of the moon.

Synodic Month

The synodic month, or complete cycle of phases of the Moon as seen from Earth, averages 29.530588 mean solar days in length. The cycle was used by Meton (fl. 432 BC), an Athenian astronomer for a luni-solar calendar. We use epoch time in milliseconds as input to our calculations and calculate the moon phase as follows where 86400000 = 24*60*60*1000, i.e. the number of milliseconds in one day:

```
% phase(+Integer, -Float)
phase(T, N) :-
   D is T/86400000,
   P is (D - 6.5)/29.5305882,
   N is (P-truncate(P))*8.
```

A German description of the moon phase can be extracted by the following table and Prolog code:

```
% descr(+Integer, -Atom)
descr(1, R) :- !, R = 'Zunehmender Sichelmond'.
descr(2, R) :- !, R = 'Zunehmend Halbmond'.
descr(3, R) :- !, R = 'Zunehmend Dreiviertelmond'.
descr(4, R) :- !, R = 'Vollmond'.
descr(5, R) :- !, R = 'Abnehmend Dreiviertelmond'.
descr(6, R) :- !, R = 'Abnehmend Halbmond'.
descr(7, R) :- !, R = 'Abnehmend Sichelmond'.
descr(_, 'Neumond').
```

A Prolog query then gives us, where the wall statistics key gives us epoch time in milliseconds:

```
?- statistics(wall, T), phase(T, P),
    K is truncate(P+0.5), descr(K, D).
T = 1706027531173, P = 3.4556983476595633,
K = 3, D = 'Zunehmend Dreiviertelmond'.
```

We are now going to wrap the Prolog query into a HTML page reachable through a HTTP server. The HTTP server will do the computation and produce some HTML markup for a visualization.

Server Graphics

Our library(vector) provides a couple of SVG primitives. To show a moon and its sunlit portion, we use the SVG path primitive and an Arc path element.

```
% moon(+Stream, +Float)
moon(S, N) :-
   X is min(60,(N-2)*30), abs_flag(X, RX, SG),
   X2 is min(60,(6-N)*30), abs_flag(X2, RX2, SG2),
   svg_begin(S, [width(40),height(40)]),
   svg_rect(S, 0, 0, 200, 160, 'empy'),
   svg_circle(S, 100, 80, 60, 'dark'),
   svg_path(S, ['M',100,20,'A',RX,60,0,0,SG,100,140,
             'A',RX2,60,0,0,SG2,100,20,'Z'], 'light'),
   svg_end(S).

% abs_flag(+Float, -Float, -Integer)
abs_flag(X, RX, 1) :- X < 0, !, RX is -X.
abs_flag(X, X, 0).
```

Thanks to the new library(markup) the SVG primitives can be automatically pretty printed to an output stream. So that we can use a Prolog query to inspect the generate SVG markup:

```
?- current_output(S), dom_output_new(S, T),
   moon(T, 3.4556983476595633).
<svg style="width: 16.6667em; height: 13.3333em" viewBox="0 0 200 160">
   <rect x="0" y="0" width="200" height="160" class="empy"/>
   <circle cx="100" cy="80" r="60" class="dark"/>
   <path d="M 100 20 A 43.671 60 0 0 0 100 140 A 60 60 0 0 0 100 20 Z"
class="light"/>
</svg>
```

The SVG rendering is not discretized into 8 different pictures, instead the above code will generate continuously varying graphics depending on the given floating point value.

HTML Page

By means of the library(spin) we wrote a HTTP server that does serve two pages /style.css and /moon.cgi. The style sheet /style.css is a static file served from the file system, it is used to give the SVG elements some styling such as a fill colour:

```
dispatch('/style.css', _, _, Res) :-
   http_write_head(Res, 200,
        ['content-type'-'text/css; charset=utf-8']),
   http_text_new(Res, S),
   open('style.css', read, T),
   Etc..
```

The dynamic page /moon.cgi will present the description and present the sunlit portion in a page with forward and backward links. If the day is not given in the URL query parameters it will take the current day:

```
dispatch('/moon.cgi', List, _, Res) :-
   http_write_head(Res, 200,
       ['content-type'-'text/html; charset=utf-8']),
   http_text_new(Res, S),
   dom_output_new(S, T),
   (member(day - Day2, List) ->
      sys_time_atom('%Y-%m-%d', Time, Day2);
      statistics(wall, Time)),
   phase(Time, Phase),
Etc..
```

The learning curve is not extremely high, since there is no need to generate a Prolog term, that would describe the output. The developer can freely interleave output statements and computation statements. As he is most likely already used when dealing with a TTY.

Example Uses

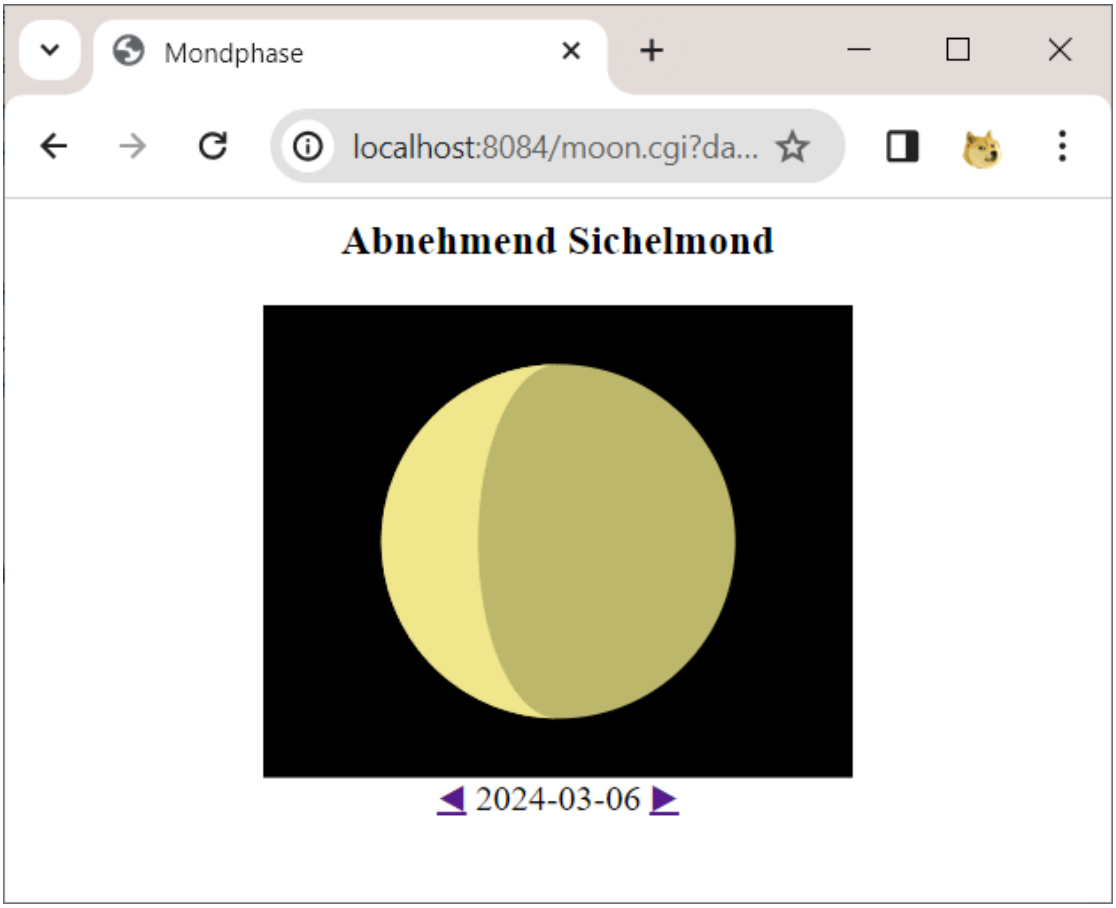
The server can be run with either JavaScript where it will use the library node:http, or Python where it will use the library http_server or Java where it will use the library jdk.httpserver. We chose port 8084 to run the HTTP server:

```
main :-
   http_server_new(S),
   http_server_on(S,'request', [P,Q], dispatch(P,Q)),
   http_server_listen(S, 8084).
```

After starting the server one can point the browser to the following URL:

```
http://localhost:8084/moon.cgi
```

The browser will show something along:



Picture 5: Hypertext Document shown by the Moon Server

3 Dogelog Libraries

In this section we list the Dogelog Player specific libraries which currently cover data structures, logic constructs and a testing utility. The libraries in “common” have been literally borrowed from the Dogelog Runtime, but do have less segmented names. The libraries in “tester” are a new take of what is found in the Dogelog Environment.

- **Folder "common":** This folder contains mainly libraries that cover data structures and logic constructs. Many of the libraries are common to other Prolog systems.

- **Folder “util”:** The folder contains mainly libraries that allow generating content in various formats. Some of the libraries are common to other Prolog systems.

- **Folder "tester":** This folder contains a testing utility. It’s the same testing utility that is used in the Dogelog Player compliance test suite.

- **Folder "misc":** The folder contains file system and host language access, as well as further content generation utilities.

3.1 Folder "common"

This folder contains mainly libraries that cover data structures and logic constructs. Many of the libraries are common to other Prolog systems.

- **File "compat":** This file provides compatibility predicates.
- **File "sequence":** This module is inspired by SQL query options such as TOP. Providing such a module was recently pioneered by SWI-Prolog.
- **File "aggregate":** This is reduced re-implementation of the library from formerly Jekejeke Prolog. Unlike the latter, this implantation is based on change_arg/3.
- **File "lists":** This file provides persistent lists.
- **File "sets":** This file provides persistent sets.

File "compat"

This file provides compatibility predicates and evaluable functions. The predicate number/3 assigns the compound '$VAR'/1 to variables, enumerating integers for the first argument of the compound. The predicate is not required by the ISO core standard, although the ISO core standard wants that the write predicates understand the compound.

The following compat predicates are provided:

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

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>.

sort(L, R): [TC2 8.4.3]
The predicate succeeds in R with the unstable sorted list L.

keysort(L, R): [TC2 8.4.4]
The predicate succeeds in R with the stable key sorted list L.

unify_with_occurs_check(S, T): [ISO 8.2.2]
The built-in succeeds when the Prolog terms S and T unify with occurs check, otherwise the built-in fails.

subsumes_term(X, Y): [ISO 8.2.4]
The built-in succeeds if X subsumes Y without keeping the bindings.

subsumes(X, Y):
The built-in succeeds if X subsumes Y.

File "sequence"

This is reduced re-implementation of the corresponding library from formerly Jekejeke Prolog. Unlike the latter, which makes use of a Java Pivot data structure, this implantation is based on change_arg/3. This module is inspired by SQL query options such as TOP. Providing such a module was recently pioneered by SWI-Prolog.

Currently the predicates limit/2, offset/2 and call_nth/2 are provided. The current implementation of limit/2 and offset/2 is based on call_nth/2. The predicates limit/2, offset/2 and call_nth/2 solely work tuple oriented and it is possible to cascade these predicates as the example below shows:

Example:

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

The following sequence predicates are provided:

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

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

call_nth(G, C):
The predicate succeeds whenever G succeeds and unifies C with the numbering of the successes.

distinct(G):
The predicate succeeds lazily with only the first solutions of G.

File "aggregate"

This is reduced re-implementation of the corresponding library from formerly Jekejeke Prolog. Unlike the latter, which makes use of a Java Pivot data structure, this implantation is based on findall/3 respectively bagof/3. The later in turn is based on keysort/2, thus avoiding the Java Revolve data structure as well.

Example:

```
p(5).
p(2).
p(3).

?- aggregate_all((sum(X),count),p(X),R).
R = (10, 3).
```

The following aggregate predicates are provided:

aggregate_all(A, G, S):

The predicate aggregates A for the solutions of G and unifies the result with S. Works like findall/3, without witness grouping. The following aggregates are supported:

count: Returns the solution count.
sum(X): Returns the sum of X.
mul(X): Returns the product of X.
min(X): Returns the minimum of X.
max(X): Returns the maximum of X.
bag(X): Returns the list of X.
set(X): Returns the sorted list of X.
(F,G): Returns the pair of the aggregate functions F and G.

aggregate(A, G, S):

The predicate aggregates A for the solutions of G and unifies the result with S. Works like bagof/3, with witness grouping.

bagof(T, X1^…^Xn^G, L): [ISO 8.10.2]

The predicate determines all the solutions to the goal G, whereby collecting copies of the template T and the witness. The predicate then repeatedly succeeds for the witness and the list of associated templates.

setof(T, X1^…^Xn^G, L): [ISO 8.10.3]

The predicate determines all the solutions to the goal G, whereby collecting copies of the template T and the witness. The predicate then repeatedly succeeds for the witness and the set of associated templates.

File "lists"

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

Examples:

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

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

The predicates append/3, reverse/2, member/2, select/3, last/2 and last/3 work directly with lists. The predicates length/2, nth0/3, nth0/4, nth1/3 and nth1/4 take also a length respective index into account.

The following list predicates are provided:

memberchk(E, S):
The predicate succeeds once when the list S contains the element E.

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

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

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

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

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

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

File "sets"

This module provides persistent sets. The predicates subtract/3, intersection/3, union/3 and symdiff/3 provide basic set operations. To work correctly they require ground lists.

Examples:

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

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

Further ground list-based predicates are subset/2, disjoint/2 and equal/2. The implementations for the ground list-based predicates might differ from other implementations since they do not simply fail or loop if the leading argument is not a ground list. Instead, they throw either an instantiation error or type error to provide more safety.

The following set predicates are provided:

subtract(S, T, R):
The predicate succeeds when R unifies with the subtract of S by T.
intersection(S, T, R):
The predicate succeeds when R unifies with the intersection of S and T.
union(S, T, R):
The predicate succeeds when R unifies with the union of S and T.
symdiff(S, T, R):
The predicate succeeds when R unifies with the symmetric subtract of S and T.
subset(S, T):
The predicate succeeds when S is a subset of T.
disjoint(S, T):
The predicate succeeds when S is disjoint to T.
equal(S, T):
The predicate succeeds when S is equal to T.

3.2 Folder "util”

The folder contains mainly libraries that allow generating content in various formats. Some of the libraries are common to other Prolog systems.

- **File "charsio":** This Prolog text provides temporary input/output redirection.
- **File "format":** This Prolog text provides formatting of terms and numbers.
- **File "random":** This Prolog text provides a random number generator.
- **File "files":** This Prolog text provides file system access.
- **File "spin":** This Prolog text provides a HTTP server.

File "charsio"

This Prolog text provides temporary input/output redirection. The predicate with_text_to/2 redirects the output for the given goal and retrieves the stream content for each success. The predicate with_text_from/2 redirects the input for the given goal. Both predicates work with atoms for their return result respective actual argument.

As a convenience the predicate term_atom/[2,3] allows writing a Prolog term into a new atom. The binary predicate will use writeq/1, the ternary predicate will use write_term/2 and if quoting isn't desired, one needs to opt-out. This module also provides a couple of simple utilities to deal with the generation of XML texts.

The following charsio predicates are provided:

with_text_to(A, G):
: The predicate succeeds whenever G succeeds and unifies A with its text output.

with_text_from(A, G):
: The predicate succeeds whenever G succeeds and provides A as its text input.

term_atom(T, A):

term_atom(T, A, O):
: The build-in succeeds in writing the term T into a new atom A. The ternary predicate accepts write options O.

open_input_atom_stream(A, S):
: The built-in succeeds in S with a new input stream for the atom A.

open_output_atom_stream(S):
: The built-in succeeds in S with a new output stream.

close_output_atom_stream(S, A):
: The built-in succeeds in A with the content of the output stream S.

xml_escape(T, E):
: The predicate succeeds when E unifies with the text escape of T.

percent_encode(T, E):
: The predicate succeeds when E unifies with the percent encode of T.

File "format"

This Prolog text provides formatting of terms and numbers. The formatter format/[2,3] will delegate formatting of evaluable expressions via ~Ne, ~Nf and ~Ng to atom_number/4 and via ~d and ~Nr to atom_integer/3. Formatting of terms via ~k, ~q and ~w calls the corresponding write predicates and directly writes into the given stream or current output.

Example:

```
?- format('abc ~4f def~n',[2*pi]).
Abc 6.2832 def
```

The following format predicates are provided:

tab(N):
tab(S, N):
The predicate succeeds. As side effect N spaces are written to the current output. The binary predicate allows specifying the output stream.
format(T, L):
format(S, T, L):
The built-in succeeds in outputting the list L formatted according to the template T. The ternary predicate allows specifying an output stream. The following format controls are supported:

~a: Formatted in atom xml escape format.
~c: Formatted in atom percent encode format.
~d: Formatted in integer decimal number format.
~Ne: Formatted in float exponential number format, N defaults to 6.
~Nf: Formatted in float fixed number format, N defaults to 6.
~Ng: Formatted in float general number format, N defaults to 6.
~k: Formatted with write_canonical/2 predicate.
~n: Emit the new line character.
~q: Formatted with writeq/2 predicate.
~Nr: Formatted in integer radix number format, N defaults to 8.
~w: Formatted with write/2 predicate.
~~: Emit the character ~.

format_atom(T, L, A):
The build-in succeeds in writing the list L formatted according to the template T into a new atom A.
atom_number(A, S, N, F):
The built-in succeeds in A with the number F formatted according to the format specifier S and the number of digits N. The following format specifiers are supported:

e: Formatted in exponential number format.
F: Formatted in fixed number format.
G: Formatted in fixed number format.

File "random"

This Prolog text provides a random number generator. There is currently only an evaluable function random/0 which returns a uniform random floating-point value. The evaluable function uses the corresponding host language random number generator.

The evaluable functions msb/1 and lsb/2, and the predicates testbit/2 and divmod/4 are not required by the ISO core standard. Implementing these functions and predicates by means of the bitwise and number theoretic functions is not feasible. Native implementations with access to the integer representation perform better by an order of magnitude.

The following random evaluable functions are provided:

msb(X):
If X is an integer, then the function returns the most significant bit.

lsb(X):
If X is an integer, then the function returns the least significant bit.

The following random predicates are provided:

random(A):
The predicate succeeds in A with a uniform random 64-bit floating point value in the interval [0..1).

testbit(X, Y):
The predicate succeeds when X /\ (1 << Y) =\= 0.

divmod(X, Y, Z, T):
If X and Y are both integers then the predicate succeeds in Z with the division of X by Y, and in T with the modulo of X by Y.

File "files"

This Prolog text provides file system access. The natively implemented predicates are directory_files/2, file_exists/1, make_directory/1 and delete_file/1. The predicates are modelled after GNU Prolog, which means that the predicate file_exists/1 succeeds for regular, directory and other file entries.

The other predicates are bootstrapped from the native predicates and from core predicates. Since Dogelog Player only supports text streams, there ore no predicates that can access or modify the content of binary streams. Nevertheless, predicates such as copy_time/2 that deal with meta information work also for binary streams.

The following files predicates are provided:

directory_files(F, L):
The predicate succeeds in L with the entries of the directory F. Barks if path F doesn't exist or if path F doesn't point to directory.

file_exists(F):
The predicate succeeds if the file F exists, otherwise fails.

make_directory(F):
The predicate succeeds. As a side effect a directory F is created. Barks if the parent of F doesn't exist or if F already exists.

delete_file(F):
The predicate succeeds. As a side effect the file F is deleted. Barks if the file F doesn't exist or if it is a directory.

copy_binary(F, G):
The predicate succeeds. As a side effect the file F is copied to the file G, without rollback upon error. An already existing file G is silently overwritten.

directory_member(F, N):
The predicate succeeds in N with the files of the directory F. Barks if path F doesn't exist or if path F doesn't point to directory.

ensure_directory(F):
The predicate succeeds. As a side effect it ensures a directory F.

copy_text(A, B):
copy_text(A, B, O):
The predicate succeeds. As side effect it copies the file A into the file B. An already existing file B is silently overwritten. The ternary predicate allows specifying copy text options. The following copy text options are supported:

append(B): B is the text append flag.

copy_time(A, B):
copy_time(A, B, O):
The predicate succeeds. As side effect it copies the last modified date from file A to the last modified date of file B. The ternary predicate allows specifying copy time options. The following copy time options are supported:

update(B): B is the time update flag.

enum_lines(S, A):
The predicate succeeds in A with the subsequent lines from the stream S.

File "spin”

This Prolog text provides a HTTP client in the form of open/4 predicate. The predicates http_server_new/1, http_server_on/4 and http_server_listen/2 provide a HTTP server configuration and start-up. The End-users can realize their own dispatch mechanism by means of the predicates http_current_method/2, http_current_path/2 and url_search_params/3.

Currently we only support text content and no binary content yet. The end-user can deliver content via the predicates write_head/3 and http_text_new/2. Calling flush_output/1 on the text writer is permitted and will complete a chunk and send it to request client. To indicate the end of a response it is mandatory to call close/1 on the text writer.

The following spin predicates are provided:

open(P, M, S, O): [ISO 8.11.5.4]

The built-in succeeds in S with a new stream for the path P and the mode M and open options O. The available open options are as follows. Depending on path, open mode and target platform not all open options might be supported.

type(T): The type T of the stream S, ‘binary’ or ‘text’.
method(M): Override the request method by M.
headers(P): Add the request headers P.
body(B): Send the request body B.
body(B, O): Send the request body B with write options O.

http_server_new(S):

The predicate succeeds in S with a new http server.

http_server_on(S, T, L, G):

The predicate succeeds. As a side effect it adds a type T event handler with formal parameter list L and callback goal G to the HTTP server S. Currently supported event type on all targets:

request: Parameter list [P, Q] where P is request and Q is response.

http_server_listen(S, P):

The predicate succeeds. As a side effect the server S starts listening on port P.

http_current_method(S, M):

The predicate succeeds in M with the method of the HTTP request S.

http_current_path(S, P):

The predicate succeeds in P with the path of the HTTP request S.

http_input_new(S, R):

http_input_new(S, R, O):

The predicate succeeds in R with a new text reader for the HTTP request S. The ternary predicate allows specifying text reader options.

url_search_params(U, P, L)

The predicate succeeds in P with the non-query part of U and in L with the query part of U decoded into a key-value list.

http_write_head(S, C, H):

The predicate succeeds. As a side effect it writes the status code C and the headers map H to the HTTP response S.

http_output_new(S, W):

http_output_new(S, W, O):

The predicate succeeds in W with a new text writer for the HTTP response S. The ternary predicate allows specifying text writer options.

3.3 Folder "tester”

This folder contains a testing utility. It’s the same testing utility that is used in the Dogelog Player compliance test suite.

- **File "runner":** This module allows executing test cases.
- **File "diagnose":** This module allows the online display of test results.
- **File "report":** This module allows the online display of test results.
- **File "indexer":** We provide a utility to build a predicate index.
- **File "beautify":** We provide a utility to pretty print a Prolog text.

File "runner"

This module allows executing test cases. The test runner can be invoked via the predicate runner_batch/1. The test runner executes the test cases and summarizes the results. The test suite under consideration needs to be supplied to the runner by consulting Prolog texts that contain facts and rules for the data model of the runner.

The library shares the following data model with the test suite:

```
runner_folder(Folder, Descr).
runner_file(Folder, File, Descr).
runner_pred(Fun, Arity, Folder, File, Descr).
runner_case(Fun, Arity, Folder, File, Descr) :- Body.
```

The library shares the following data model with the test harness:

```
measure_time(Dialect, Time)
```

The test steps and the test validation points need to be implemented in the body of the predicate test_case/5. The Ok count reflects the number of bodies that succeed once. The Nok count reflects the number of bodies that fail or that throw an error. The body is assumed to terminate, the test runner doesn't impose some timeout currently.

The library shares the following data model with the further tooling:

```
result_summary(Tag, Data).
result_suite(Folder, Tag, Data).
result_tests(Folder, File, Tag, Data).
result_pred(Fun, Arity, Folder, File, Tag, Data).
result(Fun, Arity, Folder, File, Descr, Tag, Data).
```

The following diagnose predicates are provided:

runner_batch(T):
The predicate executes the currently loaded test cases, collects and summarizes the success and failure results under the tag T.

measure_batch(T):
The predicate executes the currently loaded test cases, collects and summarizes the time measurement results under the tag T.

diff_batch(L, T):
The predicate creates test results with tag T, that have pre-computed the difference indicator of the test results for the tag list L.

dump_batch(F, T):
The predicate writes the test results under the tag T to the file F. An already existing file F is silently overwritten.

File "diagnose"

This module allows the online display of test results. Beforehand the runner needs to be used to produce the test results. The predicate diagnose_online/1 will then provide an interactive command line prompt driven viewer for the test results. The viewer proceeds in providing a drill down along the following grouping levels:

```
+--- Folders
       +--- Files
              +--- Predicates
                     +--- Cases
```

The data shown for a select node of the grouping tree is multi-column. The shown columns are determined by the provided tag list. There will be a column for each tag in the tag list. The column can either show an Ok and Nok pair. Or the column might also show other data depending on the tooling, currently we do support a difference asterisk (*).

The following diagnose predicates are provided:

diagnose_online(L):
The predicate starts a terminal based drill down of the test results for the tag list L.

File "report"

This module allows batch generation of result pages. Through dump_batch/2 the runner provides the creating of result files. Multiple result files can be loaded into memory by means of the ordinary ensure_load/1 predicate. An integrated viewer is already available through diagnose_online/2. This viewer provides a command line style interface.

```
legend_table(Legend).
legend_column(Tag, Legend).
```

Alternatively, to generate a HTML report the predicate report_batch/2 can be invoked. It takes a directory and a list of tags. If the end-user desires, he can also beforehand call diff_batch/2 and produce a further column for the HTML report. Column legends can be provided through legend_column/2 facts.

The following report predicates are provided:

report_batch(D, L, O):
The predicate succeeds in writing HTML summary and results pages for the tag list L into the directory D. The parameter O is the options list. The report generator silently overwrites already existing HTML pages. The following options are accepted:

tests(A): A is the location of the test cases.
title(A): A is the title of the report.
date(A): A is the date of the report.

summary_batch(D, L, O):
The predicate succeeds in writing HTML summary only pages for the tag list L into the directory D. The parameter O is the options list. The report generator silently overwrites already existing HTML pages. Same options as in report_batch/3.

File "indexer"

We provide a utility to build a predicate index. The predicate build_file/2 allows running through a Prolog text and collecting the defined predicates. Unlike the old Jekejeke Prolog utilities, the files are only visited allowing the generation of cross indexes. Predicate indicators that start with “sys_”, “ir_”, “os_”, “kb_” or “dg_” are not collected.

Once collected, the predicate indicators can be output to various file formats. The predicate output_tsv/1 allows writing a tab separate values file. The predicate output_html/1 allows writing a HTML report.

The following indexer predicates are provided:

build_file(A, T):
: The predicate succeeds in building a predicate index for the Prolog text A with tag T.

output_tsv(A):
: The predicate succeeds in writing the index to the tab separated values file A. An already existing file A is silently overwritten.

output_html(A):
: The predicate succeeds in writing the index to the HTML file A. An already existing file A is silently overwritten.

File "beautify"

We provide a utility to pretty print a Prolog text. The pretty printing runs through the given file copying comments before a predicate verbatim. The predicate clauses are then copied as if they were output by listing/[0,1], except that variable names are preserved as well. Unlike the old Jekejeke Prolog utilities, the files are not loaded only visited.

The following beautify predicates are provided:

fancy_file(A, B):
: The predicate succeeds. As side effect it colorizes the Prolog text A into the file B. An already existing file B is silently overwritten.

beautify_file(A, B):

beautify_file(A, B, O):
: The predicate succeeds. As side effect it beautifies the Prolog text A into the file B. An already existing file B is silently overwritten. The ternary predicate accepts beautify options O. The following options are recognized:

  suite(B): B is the test case anchor flag.

3.4 Folder "misc”

The folder contains file system and host language access, as well as further content generation utilities.

- **File "json":** This module provides affine JSON to Prolog mapping.
- **File "markup":** This file provides predicates to generate markup.
- **File "react”:** This file provides predicates to handle events.
- **File "vector":** This file provides means the generate and interact with scalable vector graphics.

File "json"

This file provides JSON to Prolog mapping. We map lists to Prolog lists and objects to Prolog set notation. We resolve the string conflict in that we map strings to Prolog atoms, but use the compounds @(null), @(false) and @(true) for JSON null, false and true. The predicates write_json/[1,2] and read_json/[1,2] offer input output.

As a convenience we provide the predicate json_atom/2 to convert a JSON term into an atom. Further there are the predicate json_object_current/3, json_object_set/4 and json_object_remove/3 to access and modify a JSON term. Modification is non-destructive and creates a new JSON term.

The following JSON predicates are provided:

write_json(T):
write_json(S, T):
: The predicate succeeds. As a side effect the JSON term T is written. The binary predicate allows specifying an output stream S.

read_json(E):
read_json(S, E):
: The predicate succeeds in E with the JSON term or end_of_file. As a side effect, the input position is advanced. The binary predicate allows specifying an input stream S.

json_atom(T, A):
: The predicate succeeds in A with the atom for the JSON term T.

json_object_current(T, K, V):
: The predicate succeeds in V with the value for the key K in the JSON term T.

json_object_set(T, K, V, S):
: The predicate succeeds in S with the JSON term after replacing the value for the key K by V in the JSON term T.

json_object_remove(T, K, S):
: The predicate succeeds in S with the JSON term after removing the value for the key K in the JSON term T.

File "markup"

This file provides predicates to generate markup. This is done through filtered streams that automatically escape plain text. We provide filtered text output streams that either point to a browser DOM element or that that piggyback on an arbitrary given stream. The escaping can be bypassed by the predicates tag/[1,2] and tag_format/[2,3].

The following markup predicates are provided:

tag(M):
tag(W, A):
: The predicate emits the start tag, end tag or single tag M. The binary predicate allows specifying a DOM writer W.

tag_format(T, L):
tag_format(W, T, L):
: The predicate emits the start tag, end tag or single tag that results from formatting the template T with the arguments L. The ternary predicate allows specifying a DOM writer W.

dom_output_new(W):
dom_output_new(S, W):
: The predicate succeeds in W with a new writer to the cursor. The binary predicate allows specifying an underlying stream S.

dom_error_new(W):
dom_error_new(S, W):
: The predicate succeeds in W with a new writer to the cursor. The binary predicate allows specifying an underlying stream S.

dom_cell_current(C):
: The predicate succeeds in C with the cursor.

dom_cell_set(C):
: The predicate changes the cursor to C.

File "react”

This file provides predicates to handle events. An event handler can be established with the predicate bind/3 for the DOM cell at the cursor. This event handler will be called in callback mode, means it will be called in the task context when the event handler was created and it will be called without auto-yield or the possibility to use async.

We do not provide some type specific built-in predicate to access or modify the various objects involved in a DOM and its event model. One can use the general internal representation predicates ir_object_current/2 and ir_object_set/2 to access and modify object fields. For number values it is recommended to use ir_float_current/2 and ir_float_set/2.

The following react predicates are provided:

clear:
The predicate succeeds. As a side effect the current cursor is cleared.
goto(I):
The predicate changes the cursor to the element with id I.
bind(T, P, G):
bind(C, T, P, G):
The predicate succeeds. As a side effect it adds a type T stackless event handler with formal event parameter P and callback goal G to the cursor. The quaternary predicate allows specifiying an element.
bind_capture(T, P, G):
bind_capture(C, T, P, G):
The predicate succeeds. As a side effect it adds a type T stackless event handler with formal event parameter P and callback goal G to the cursor. The quaternary predicate allows specifiying an element.
dom_prevent_default(E):
The predicate prevents default of the event E.
dom_stop_propagation(E):
The predicate stops propagation of the event E.
ir_float_current(O, K, V):
The predicate succeeds in V with the float value of the key K in the object O.
ir_float_set(O, K, V):
The predicate sets the float value of the key K in the object O to V.

File "vector"

This file provides means the generate and interact with scalable vector graphics. The output predicates require a DOM writer as found in library(misc/markup), either explicit as a parameter or implicit as current output. The SVG output area will have 500x400 SVG coordinate dimension and a twelfth of it in relative font size as browser dimension.

The following vector predicates are provided:

svg_begin(L):
svg_begin(S, L):
The predicate succeeds. As a side effect a new SVG output area with options in L. The binary predicate allows specifying a DOM writer. The following begin options are recognized

height(H): The relative height in percent, defaults to 100%.
width(H): The relative width in percent, defaults to 100%.

svg_rect(X, Y, W, H, C):
svg_rect(S, X, Y, W, H, C):
The predicate succeeds. As a side effect a rectangle element at (X,Y) with dimension (W,H) and style C is added to the SVG output area. The septenary predicate allows specifying a DOM writer.
svg_line(X1, Y1, X2, Y2, C):
svg_line(S, X1, Y1, X2, Y2, C):
The predicate succeeds. As a side effect a line element from (X1,Y1) to (X2,Y2) with style C is added to the SVG output area. The septenary predicate allows specifying a DOM writer.
svg_text(X, Y, T, C):
svg_text(S, X, Y, T, C):
The predicate succeeds. As a side effect a text element at (X,Y) with content T and style C is added to the SVG output area. The pentamery predicate allows specifying a DOM writer.
svg_circle(X, Y, R, C):
svg_circle(S, X, Y, R, C):
The predicate succeeds. As a side effect a circle element (X,Y) with radius R and style C is added to the SVG output area. The pentamery predicate allows specifying a DOM writer.
svg_path(L, C):
svg_path(S, L, C):
The predicate succeeds. As a side effect a path element with shape L and style C is added to the SVG output area. The ternary predicate allows specifying a DOM writer.
svg_image(X, Y, W, H, U):
svg_image(S, X, Y, W, H, U):
The predicate succeeds. As a side effect an image element at (X,Y) with width W, height H and image URL U s added to the SVG output area. The sixternary predicate allows * specifying a DOM writer.

svg_end:
svg_end(S):
The predicate succeeds. As a side effect the SVG output area is closed. The unary predicate allows specifying a DOM writer.
svg_view_inverse(E, I):
The predicate succeeds in I with the inverse transform of the element E.
svg_apply_transform(I, CX, CY, SX, SY):
The predicate succeeds in the viewport coordinates (SX,SY) with the inverse transform I applied to the client coordinates (CX,CY).

4 Dogelog Literate

The Dogelog Player can be also used as a runtime for Prolog notebooks, which is the topic of this section.

- **Folder "press":** Literate programming was coined by Donald Knuth and we see it as the combination of comments, program code and test cases.

4.1 Folder "press"

Literate programming was coined by Donald Knuth and we see it as the combination of comments, program code and test cases. It has found a continuation in computational notebooks popular in data science. We support literate programming for the Prolog programming language whereas the Dogelog player takes care of code execution.

- **Section "literate":** We provide commands to convert Prolog texts into HTML Pages with text cells, code cells and output cells.

Section "literate"

We provide commands to convert Prolog texts into HTML Pages with text cells, code cells and output cells. The text cells are derived from the comments in the Prolog text. The code cells are derived from the facts, rules and queries in the Prolog text, and an output cell is automatically added to every code cell.

The literate programming provides the following commands:

webify_libs(D):
The predicate succeeds in adding libs to the directory D. The webifyer silently overwrites existing libs in the directory D.

webify_page(A, B):
webify_page(A, B, O):
The predicate succeeds in a HTML page B for the Prolog text A. The webifyer silently overwrites an already existing HTML page B. The ternary predicate allows specifying webify options. The following webify options are supported:

async_mode(M): M indicates whether async consult should be generated.

Acknowledgements

We are thankful to a discussant on SWI-Prolog discourse for challenging us over Prolog deterministic code execution versus imperative code execution.

Indexes

Pictures

Tables

Es konnten keine Einträge für ein Abbildungsverzeichnis gefunden werden.

Acronyms

ISO [1]

References

[1] ISO (1995): Prolog, Part 1: General Core, International Standard ISO/IEC 13211-1, First Edition, 1995-06-01
http://www.iso.org/standard/21413.html

[2] Clocksin, W. (1983): A portable Prolog compiler, Logic Programming Workshop, Albufeira Portugal, January 1983
http://www.softwarepreservation.org/projects/prolog/lisbon/lpw83/p74-Bowen.pdf

[3] Carlson, M. et al. (1988): Garbage collection for Prolog based on WAM. Communications of the ACM 31, 6, 719–740, June 1988
http://dl.acm.org/doi/10.1145/62959.62968

[4] Wirfs-Brock, A. (2020): JavaScript: the first 20 years, Proc. ACM Program. Lang., Vol. 4, No. HOPL, Article 77. Publication date: June 2020.
http://dl.acm.org/doi/10.1145/3386327

[5] JavaScript (2020): ECMAScript® 2020 Language Specification, 11th-Edition, Ecma International, June 2020
http://262.ecma-international.org/11.0/