Dogelog Player Language

Version 1.2.0, March 16, 2024

XLOG Technologies AG

Dogelog Prolog

Dogelog Player 1.2.0

Language Reference

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

Date: March 16, 2024
Version: 0.21

Warranty & Liability

Rights & License

Trademarks

Table of Contents

Change history

Jan Burse, June 11, 2021, 0.1:
- Initial version.

Jan Burse, June 14, 2021, 0.2:
- Added appendix with example listing.

Jan Burse, June 16, 2021, 0.3:
- Added an atomics section in transpiler chapter.

Jan Burse, June 21, 2021, 0.4:
- Added snippet example and canned chapter.

Jan Burse, June 24, 2021, 0.5:
- Added html section and foreign section.

Jan Burse, July 3, 2021, 0.6:
- New instruction singleton introduced.

Jan Burse, July 13, 2021, 0.7:
- New section unicode introduced.

Jan Burse, August 21, 2021, 0.8:
- Started Python version of Dogelog player.

Jan Burse, September 05, 2021, 0.9:
- Dogelog player uses host language for evaluable predicates.

Jan Burse, September 06, 2021, 0.10:
- Fast bridge for evaluable predicates.

Jan Burse, September 16, 2021, 0.11:
- New instruction parameter and continuation introduced.

Jan Burse, December 14, 2021, 0.12:
- More built-ins.

Jan Burse, December 30, 2021, 0.13:
- Top-level example removed.

Jan Burse, February 02, 2022, 0.14:
- Section “async” and section “promise” introduced.

Jan Burse, March 23, 2022, 0.15:
- Self-hosting cross compiler.

Jan Burse, June 29, 2022, 0.16:
- Literate programming and non-monotonic stages.

Jan Burse, August 24, 2022, 0.17:
- library(_) and foreign(_) file specifiers.

Jan Burse, September 23, 2022, 0.18:
- New console game example introduced.

Jan Burse, April 12, 2023, 0.19:
- AST compiler and annotated read and write.

Jan Burse, December 01, 2023, 0.20:
- Transpiler section moved to separate host document.

Jan Burse, January 29, 2024, 0.20:
- Section “appendix” removed.

1 Introduction

t.b.d.

- **Dogelog Examples:** We provide examples of using the Dogelog player out in the wild. Currently there is a website sandbox example.

- **Dogelog Novacore:** Dogelog Player is based on a novel core, which is derived from ISO core standard but has a much smaller set of predicates and data structures.

- **Dogelog Entry:** The target files are either result of transpilation or genuinely written. They are among the files that define the Dogelog player.

2 Dogelog Examples

We provide examples of using the Dogelog player out in the wild. Currently there are the following examples, most of them addressing the JavaScript target, but there is also an example addressing the Python target:

- **Console Game:** t.b.d.

- **Website Sandbox:** The current sandbox API from the Dogelog player aims at providing a toolbox to integrate a Dogelog sandbox into a website.

- **Browser Game:** The Dogelog player performs well enough to provide an interactive board game. We use consult/1 to load the Prolog text of the game.

- **Audio Sequencer:** Dogelog features a foreign function interface. In this example we demonstrate how to use it, to let Dogelog play Mary had a little Lamb.

2.1 Console Game

t.b.d.

- **Game Search:** The game search uses Prolog terms to represent the board. Negation as failure the helps to find best move suggestions.

- **Console Play:** The play responsibility is realized in the Prolog programming language in that a while loop is replaced by a tail recursive loop.

- **Example Uses:** t.b.d.

Game Search

The Game search uses Prolog terms to represent the board. Negation as failure the helps to find best move suggestions. We do a full game search, which means our game search does not have any parameter limiting the search depth. The game search requires that predicates such as move/3 and win/2 be defined.

The game search then uses backtracking to search a best move:

```
best(X, P, Y) :-
  move(X, P, Y),
  (win(Y, P) -> true;
    other(P, Q),
    \+ tie(Y, Q),
    \+ best(Y, Q, _)).
```

The above Prolog code corresponds to a min/max search. The predicate best/3 succeeds with a move where the opponent will move into a tie or where the current player will win. This corresponds to the values 0 and 1. The predicate fails if the opponent will win or the current player moves into a tie. This corresponds to the values -1 and 0.

The above Prolog code also implements a kind of alpha/beta pruning. The sub goal best/3 only needs to succeed once. The if-then-else means the parent calls best/3 only if it has not yet a direct winning move. To oppose an indirect winning move, only a single opponent winning move need to be found, the rest of the moves can be pruned.

Console Play

The game search is located in the Prolog text tictac.p. The Prolog text console.p is then responsible for an online interactive game play based on an ASCII console. In an imperative language the game play can be viewed as a while loop along the following lines, whereby we alternate between a user move and a computer move:

```
while not /* complete */ do
   /* user move */
   /* computer move */
end
```

We realize the same responsibility in the Prolog programming language and replace the while loop by a tail recursive loop. The realization is made declarative in that no side effect is used to store the board, which is also supported by the game search which represents a board as a Prolog term. We get the following sketch:

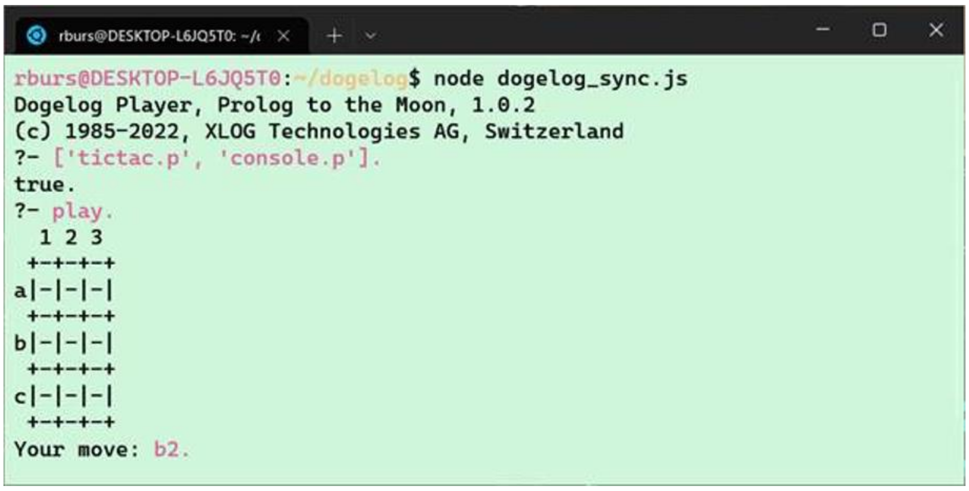
```
loop(S) :-
   \+ complete(S),
   user_move(S, S2),
   computer_move(S2, S3),
   loop(S3).
```

The Tic-Tac-Tow game is complete when either of the players has won, or when there is a draw. This might also happen after the user move and before the computer move, in which case the computer will not compute a move anymore, but show a message with the actual outcome of the game. For more details see the code listing in the appendix.

Example Uses

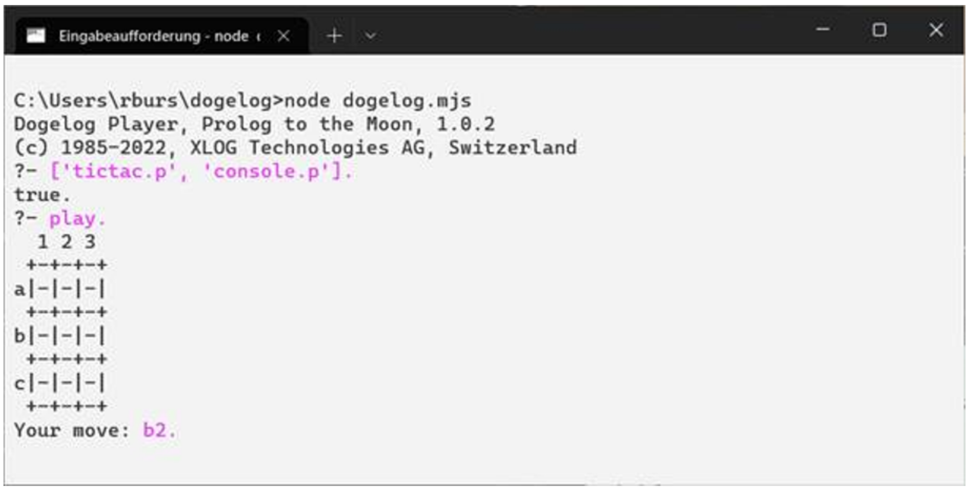
The Dogelog player runs not only in the browser. There are currently command line versions available for the JavaScript platform and the Python platform. These command line versions come with a colored top-level that can be easily started as soon as one has installed a JavaScript runtime respectively a Python runtime.

A typical command line JavaScript runtime is NodeJS. This JavaScript runtime uses the V8 JavaScript Engine from the Chrome browser. Similar like the browser JavaScript has evolved and is now available as common JavaScript and module JavaScript. Different versions of NodeJS might support these variants to different degrees.



```
rburs@DESKTOP-L6JQ5T0:~/dogelog$ node dogelog_sync.js
Dogelog Player, Prolog to the Moon, 1.0.2
(c) 1985-2022, XLOG Technologies AG, Switzerland
?- ['tictac.p', 'console.p'].
true.
?- play.
  1 2 3
 +-+-+-+
a|-|-|-|
 +-+-+-+
b|-|-|-|
 +-+-+-+
c|-|-|-|
 +-+-+-+
Your move: b2.
```

Picture 1: Console Game in NodeJS 12.x WSL2



```
C:\Users\rburs\dogelog>node dogelog.mjs
Dogelog Player, Prolog to the Moon, 1.0.2
(c) 1985-2022, XLOG Technologies AG, Switzerland
?- ['tictac.p', 'console.p'].
true.
?- play.
  1 2 3
 +-+-+-+
a|-|-|-|
 +-+-+-+
b|-|-|-|
 +-+-+-+
c|-|-|-|
 +-+-+-+
Your move: b2.
```

Picture 2: Console Game in NodeJS 18.x Windows 10

For more details on the Dogelog player variants and their support see chapter “cross” and the two sections “main” and “mainpy”. In the above we used dogelog_sync.js for an older NodeJS version and dogelog.mjs for a newer NodeJS version. What we didn’t show that the console game can be also run in Python.

2.2 Website Sandbox

The current sandbox API from the Dogelog player aims at providing a toolbox to integrate a Dogelog sandbox into a website. Such a sandbox has typically an input box, some button and some output box, and allows running Prolog code without a server roundtrip. We give here an example project that does exactly this.

- **HTML Page:** This is the main page of the sandbox example with the corresponding HTML elements and using the Dogelog player.

- **Other Assets:** A browser allows interactive multimedia content. We do not show how Prolog would do that, but we make also use of it to spice up the example.

- **Example Uses:** These are screenshots of using the sandbox example. We show the sunshine case, that the input can be process and a case with a Prolog error.

HTML Page

This is the main page of the sandbox example with the corresponding HTML elements and using the Dogelog player. To use the Dogelog player in our example, we first import those functions or variables we need. This is done by the following JavaScript statement:

```
    import {init, clear, set_output, consult, show} from "./lib/index.js";
```

The import will also load all the other JavaScript files that are subsequently imported by the File "index". We do not have to list everything in the main page. We now have to define an output callback and initialize the Dogelog interpreter once:

```
    init();

    function log(buf) {
        document.getElementById("demo").innerText += buf;
        document.getElementById("demo").innerHTML += "<br>";
    }
```

Finally there is a function main which gets called every time the button is pressed. This function will first reset the output box and also remove all user clauses from the Prolog database via the function clear(). It will then consult the text from the input box:

```
    function main() {
        document.getElementById("demo").innerHTML = "";
        clear();
        set_output(log);
        let text=document.getElementById("text").value;
        consult(text);
    }

    document.getElementById("launch").addEventListener("click", main);
```

The error handling is quite primitive. The function consult() doesn't need some special clean-up, it does all the clean-up itself. All we do is show the error term. The function show() will use the registered output callback and write the error term canonical.

Other Assets

A browser allows interactive multimedia content. We do not show how Prolog would do that, but we make also use of it to spice up the example. There is a static fact sheet page. Clicking on doge graphic in the main page, shows the fact sheet page and vice versa. The static fact sheet page shows the following text:

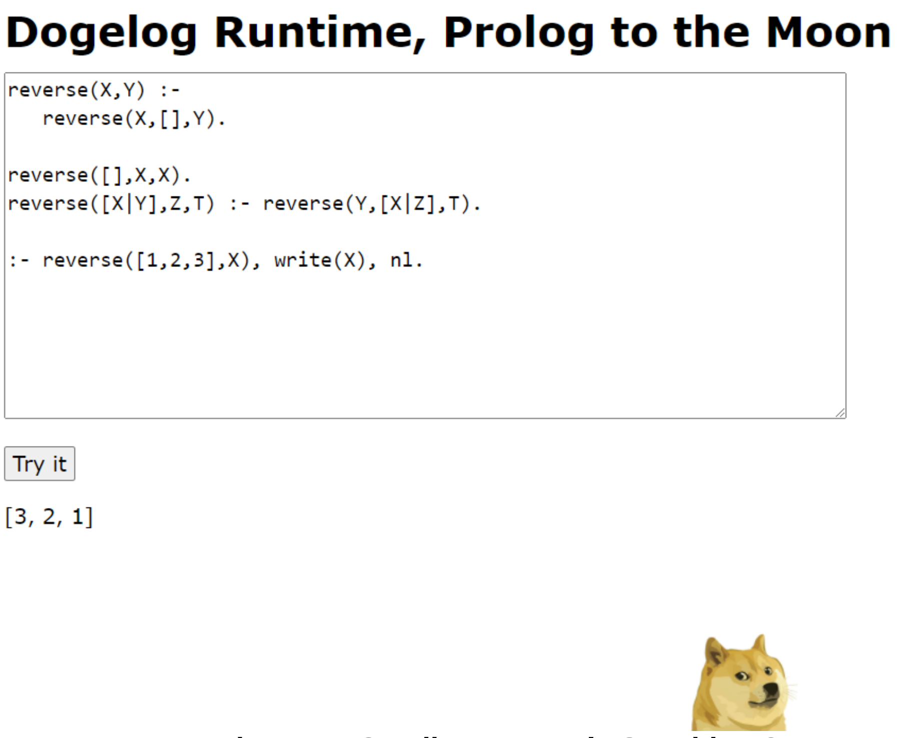
```
Q: When was Dogelog born?
A: Dogelog is still a puppy, born in 2021 after heated internet
discussions.

Q: How does Dogelog barf?
A: Dogelog speaks Prolog.

Q: Is Dogelog flesh and bones?
A: No, Dogelog is made with JavaScript.

Q: What pedigree is Dogelog?
A: Dogelog is a mongrel, half Jekejeke and half Albufeira.

Q: Does Dogelog have a dog chip?
A: No, but Dogelog has an URL www.dogelog.ch.

Q: Who is the master of Dogelog?
A: Currently Dogelog gets the most care by XLOG Technologies AG.
```

There is the large doge graphic. We dress it via CSS so that it sticks up from the bottom of the main page and the fact sheet page. In addition, there is a small moon graphic that we mention in the head of the pages, so that it gets associated with the website:

Picture 3: Sandbox Example Graphic Artefacts

Example Uses

The main page populates the input box with a working example. To show that the Dogelog player does really some work, we show a screenshot of a slightly modified input. Unfortunately, Dogelog does not yet display answer substitutions, nevertheless we can let Dogelog compute the reverse of a list and display it with the write/1 predicate:



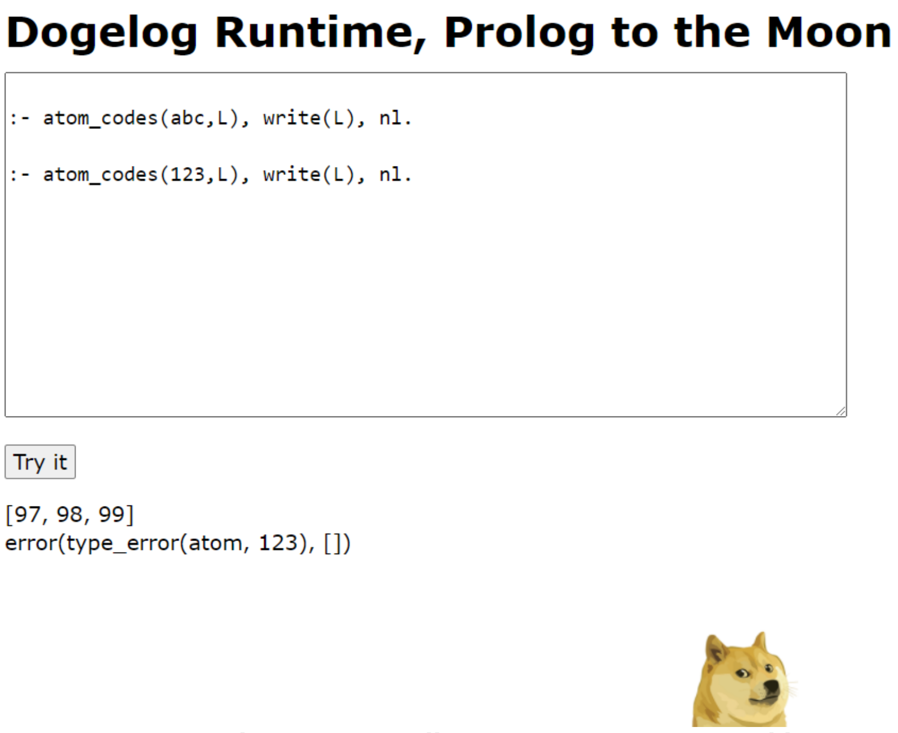
Dogelog Runtime, Prolog to the Moon

```
reverse(X,Y) :-
   reverse(X,[],Y).

reverse([],X,X).
reverse([X|Y],Z,T) :- reverse(Y,[X|Z],T).

:- reverse([1,2,3],X), write(X), nl.
```

```
[3, 2, 1]
```

Picture 4: Sandbox Example Sunshine Case

In case there is some execution error, Dogelog will display an error term. The current version is not yet that good in spotting syntax error or continuing consult, but most of the usual ISO core standard errors concerning built-ins do work. This means the Dogelog player will not silently fail if there is some parameter type mismatch and similar:


Dogelog Runtime, Prolog to the Moon

```
:- atom_codes(abc,L), write(L), nl.

:- atom_codes(123,L), write(L), nl.
```

```
[97, 98, 99]
error(type_error(atom, 123), [])
```

Picture 5: Sandbox Example Error Handling

2.3 Browser Game

The Dogelog player performs well enough to provide an interactive board game. We use ensure_loaded/1 to load the Prolog text of the game.

- **Board Setup:** The board is a div element with a couple of child button elements. Foreign functions allow to access and modify the board from within Prolog.

- **Browser Play:** The game play mediates between the game search and the board. The game play also uses a status line to show the outcome of the game.

- **Example Uses:** The main HTML page does not anymore resemble the sandbox example, since the interactive game board has replaced the text area input.

Board Setup

The board is a div element with a couple of child button elements. Foreign functions allow to access and modify the board from within Prolog. Each of the 3x3 child buttons gets a different background image depending on a label. This is done via a CSS style sheet that is embedded in the HTML page. The CSS style sheet has the following rules:

```
        [aria-label="x"] {
            background-image: url('first.svg');
        }
        [aria-label="o"] {
            background-image: url('second.svg');
        }
```

We use scalable vector graphics (SVG) images for the button backgrounds, since the file format does not occupy much space and they allow adapting their size without losing quality of the graphics. The foreign functions will allow the Prolog code to access and modify the label. An example of such a foreign function is this setter:

```
    function set_button(id, val) {
        let help = document.getElementById(id);
        if (val === "-") {
            help.removeAttribute("aria-label");
            help.removeAttribute("disabled");
        } else {
            help.setAttribute("aria-label", val);
            help.setAttribute("disabled", "disabled");
        }
    }
```

The Prolog atom '-' serves as an indicator that one of the players has not yet marked a board cell. We do disable a board cell as soon as a player marked it. The Dogelog player offers a JavaScript call to register foreign functions. Besides the JavaScript function set_button(), we also register JavaScript functions get_button() and set_complete():

```
    register("get_button", 2, get_button, FFI_FUNC);
    register("set_button", 2, set_button, 0);
    register("set_complete", 0, set_complete, 0);
```

Browser Play

The game play mediates between the game search and the board. The game play also uses a status line to show the outcome of the game. The game play is initiated by the HTML page, which has GUI interaction logic. If the end-user hits an enabled button, the GUI interaction logic labels the button with "x" and triggers the game play as follows:

```
    let help = e.target;
    help.setAttribute('aria-label', 'x');
    help.setAttribute('disabled', 'disabled');
    try {
        perform("marked");
    } catch (error) {
        show(error);
    }
```

The game play first checks the board. If the game is not yet complete, it calls best/3, which succeeds non-deterministically with new boards. A more elaborate solution would then randomly pick a solution. For simplicity we only pick the first solution, which is done by placing a Prolog cut !/0. The game play then checks the board again.

```
looser(X) :- win(X, x), !, write('you win.'), nl, set_complete.
looser(X) :- tie(X, o), !, write('nobody won.'), nl, set_complete.
looser(X) :- best(X, o, Y), !, set_board(Y), winner(Y).
looser(_) :- write('I give up.'), nl, set_complete.
```

The HTML page also defines some output call back. We simply carried over the approach from the sandbox example and write into a DOM element. In the above, the status line is then populated by invoking the standard Prolog write/1 and nl/0. More advanced solutions might combine it with audio feedback or other visual cues.

Example Uses

The main HTML page does not anymore resemble the sandbox example, since the interactive game board has replaced the text area input. The game play experience needs a fast browser that can run Dogelog player and the Tic-Tac-Toe game search sufficiently fast. Interestingly the reach of Dogelog player extends to modern tablets.

We benchmarked a full game tree search and found:

Table 1: Platform Browser Benchmark Tic-Tac-Toe

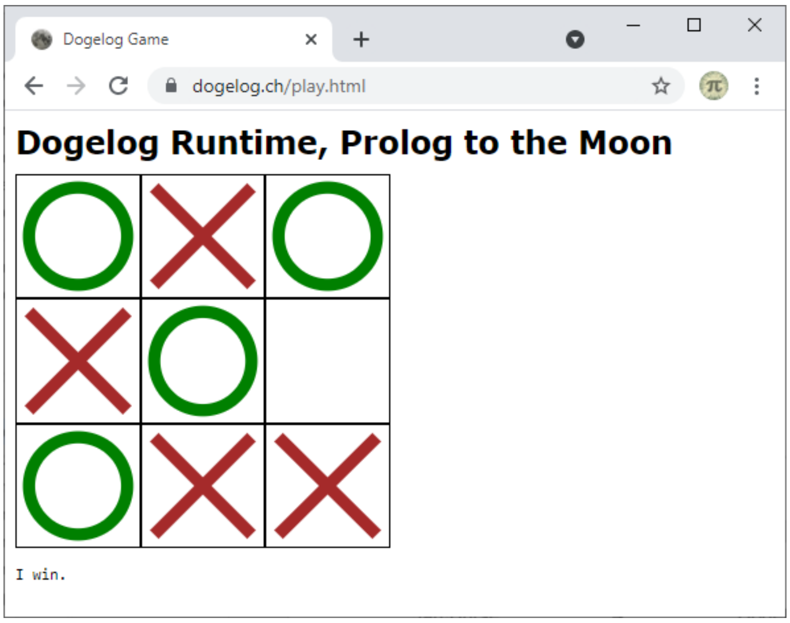
Platform, Browser	*M 0.9.2*	*C 0.9.2*
Windows 10, Chrome	312	276
Apple MacBook, Safari Prev	287	269
Android Tablet, Chrome	1'614	1'422
Apple iPad, Safari	N/A	312

We used laptops and devices not older than 5 years. We could not make the module version of Dogelog player execute on all platform and browser combination. The offending language construct was import(), which was not available on an iPad, and worked only on a MacBook after installing Safari technology preview.

We also measured canned Dogelog, which does not require modules. Interestingly canned Dogelog is faster, which might have to do that modules do have different JavaScript execution schemas. Most likely, the gap will go away in the future. Interestingly the little iPad was almost beating a laptop in speed.

Because there will be already a move when the game search kicks in, the game search will need 2-4 times less time than a full game search tree from the beginning. This means that on most platform and browser combinations the response time will be below 100ms, otherwise the end-user might get frustrated, annoyed or even angry.

Here is a screenshot where the computer wins:



```
I win.
```

Picture 6: Computer Wins the Game Board

2.4 Audio Sequencer

Dogelog features a foreign function interface. In this example, we demonstrate how to use it, to let Dogelog play Mary had a little Lamb. We also demonstrate the asynchronous API of the Dogelog player. When this API is used the Dogelog player can be forced to wait for Promise to be resolved, which we will use to time the tune.

- **Foreign Registration:** Foreign functions are useful for the end-user if he wants to quickly extend the Prolog system by JavaScript functionality.

- **Button Click:** JavaScript permits async functions as their event handler. We do so to play the tune, but we have to also intermittently disable the button.

- **Example Uses:** The HTML page has been derived from the sandbox example, pressing the Try it button will play the tune.

Foreign Registration

Foreign functions are useful for the end-user if he wants to quickly extend the Prolog system by JavaScript functionality. Since we want to let Dogelog play Mary had a little Lamb, we need at least have some Prolog predicate that can play a music note. We have the music notes as audio files and reference them inside the HTML via audio nodes:

```
[...]
<audio id="D" src="notes/D.mp3"></audio>
<audio id="Dis" src="notes/Dis.mp3"></audio>
<audio id="E" src="notes/E.mp3"></audio>
<audio id="F" src="notes/F.mp3"></audio>
[...]
```

We then register a JavaScript routine, which currently does not return any value. This routine looks up then corresponding audio node and then plays the corresponding audio through the DOM method play(). The setter currentTime is used to seek the audio node to the given time and rewind it to zero:

```
function note(key) {
    const audio = document.getElementById(key);
    audio.currentTime = 0
    audio.play();
}

register("note", 1, note, 0);
```

We experienced that the above doesn’t work on all platforms and in all browsers. The reason is possibly that in newer browsers the DOM method play() returns a promise and we do not yet correctly handle this promise. We will update the example when we have found a more portable solution.

Button Click

JavaScript permits async functions as their event handler. We do so to play the tune, but we have to also intermittently disable the button. To have an async function in JavaScript one can use the async keyword. The main method is now declared as follows; we also use the convention to change its name to append _async to the function name:

```
async function main_async() {
    document.getElementById("launch").disabled = true;
    document.getElementById("demo").innerHTML = "";
    clear();
    let text=document.getElementById("text").value;
    await consult_async(text);
    document.getElementById("launch").disabled = false;
}
```

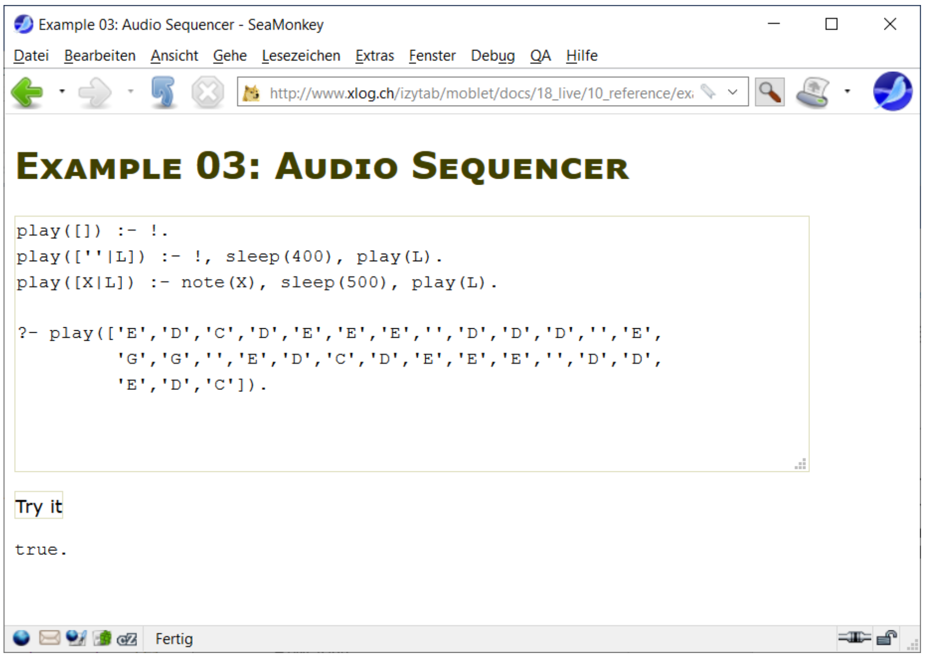
Inside the async main function there are some novel statements. We do access the launch button and set it to disabled at the beginning of the async main function. This is to avoid that the launch button is fired multiple times. We then await the Dogelog player concult_async() call and only when we pass this point, we enable the launch button again.

```
document.getElementById("launch")
        .addEventListener("click", main_async);
```

Interestingly programmatically the registration of the click event listener doesn’t change much. Except that we now register the new async main function, there is nothing in the syntax that would hint a registration of an async event listener.

Example Uses

The HTML page has been derived from the sandbox example, pressing the Try it button will play the tune. We simply added the a registrations of note/1 to the example. We also populated the initial text area with a play/1 predicate that can play a sequence. There is then a query in the Prolog text that passes a list with Mary had a little Lamb notes.


Example 03: Audio Sequencer - SeaMonkey
Datei Bearbeiten Ansicht Gehe Lesezeichen Extras Fenster Debug QA Hilfe
http://www.xlog.ch/izytab/moblet/docs/18_live/10_reference/ex
Example 03: Audio Sequencer

```
play([]) :- !.
play([''|L]) :- !, sleep(400), play(L).
play([X|L]) :- note(X), sleep(500), play(L).

?- play(['E','D','C','D','E','E','E','','D','D','D','','E',
         'G','G','','E','D','C','D','E','E','E','','D','D',
         'E','D','C']).
```

```
true.
```

Picture 7: Audio Sequencer with Default Music

The realization of the play/1 predicate makes use of the Dogelog sleep/1 command. This command will yield the Prolog engine for the given time. The Prolog engine will also automatically yield 60 times per second. The example seems to work in browsers such as Chrome and Edge, but for example Safari causes troubles in that it dims the audio.

3 Dogelog Novacore

Dogelog Player is based on a novel core, which is derived from ISO core standard but has a much smaller set of predicates and data structures. The set of predicates here only contains those predicates that are needed to allow a Prolog system with a Prolog text loader. Via this Prolog text loader further predicates can be loaded via native and non-native libraries.

- **File "store":** This file models data structures such as the Prolog knowledge base.

- **File "machine":** This file contains algorithms such as the Prolog engine.

- **File "special":** The file main goal is to provide natively implemented ISO core standard built-ins needed for the Dogelog player.

- **File "eval":** Prolog is not known for number crunching, nevertheless many Prolog systems provide a number type and corresponding operations and predicates.

- **File "runtime":** The file mainly defines predicates for reading and writing of Prolog terms.

3.1 File "store"

This file models data structures such as the Prolog knowledge base.

- **Section "include":** It is also possible to include Prolog clauses from a text file, resulting in predicates which provide further optimizations such as body indexing.

- **Section "paths":** Like many Prolog systems we provide addressing of Prolog text libraries and native libraries written in the host programming language.

- **Section "database":** Currently we do compile Prolog clauses that are dynamically asserted, which incurs a little cost, but provides faster retrieval.

Section "include"

It is also possible to include Prolog clauses from a text file, resulting in predicates which provide further optimizations such as body indexing. To do so we provide ISO core standard predicates such as include/1 and ensure_loaded/1. Diverting from the ISO core standard, a Prolog text might also contain unattended queries indicated by the (?-)/1 operator.

Prolog texts are subject to some common checks. Clauses or directives are examined by the singleton check, which gives a warning when the input text had a named variable appearing only once. A further warning respective an error aborting the clause or directive is given by the style checks concerning the discontiguous and multifile constraints.

Unlike other Prolog systems, we only provide shallow term expansion. Shallow term expansion can be defined via the multi-file predicate term_conversion/2, but there is no automatic call of goal_expansion/2 for any subgoals in the directive or clause provided as a term. Tooling can invoke expand_term/2 to apply shallow term expansion.

The following include predicates are provided:

multifile I: [ISO 7.4.2.2]
    The predicate sets the predicate I to multifile.
discontiguous I: [ISO 7.4.2.3]
    The predicate sets the predicate I to discontiguous.
include(P): [ISO 7.4.2.7]
    The predicate succeeds. As a side effect, the path P is included.
ensure_loaded(P): [ISO 7.4.2.8]
    The predicate succeeds. As a side effect, the path P is ensure loaded.
[P_1, .., P_n]:
    The predicate succeeds. As a side effect, the paths P_1, .., P_n are ensure loaded.
make:
    The predicate ensures that all used sources are loaded.
?- Q:
    The directive succeeds whenever Q succeeds. Additional it writes the answer substitutions to the standard output.
P, R --> Q:
    The fact is transformed into the clause for the non-terminal P, the push back R and the DCG body Q.
P --> Q:
    The fact is transformed into the clause for the non-terminal P and the DCG body Q.
term_conversion(C, D):
    This predicate can be used to define custom term conversion rules.
expand_term(C, D):
    The predicate succeeds in D with the expansion of the term C.
current_source(S):
    The predicate succeeds in S with the current source paths.
source_property(S, P):
    The predicate succeeds in P with the properties of the source path S. The following properties are supported:

    sys_link(T): The path loaded the source path T.

Section "paths"

To allow for the circumstances of a browser, we do not provide a search path of multiple locations and/or multiple file extensions. Rather the mechanism is such that only one resource, the Prolog flag system_url, is probed. Dogelog player can also load native libraries written in the host programming language of the target platform:

library(N): (file spec)
The file spec addresses the Prolog text library N.

foreign(N): (file spec)
The file spec addresses the native library N.

Plain file names in include/1 or ensure_loaded/1 are resolved against the parent Prolog text. Otherwise, plain file names are resolved against the Prolog flag base_url. The resolution result can be inspected via the predicate absolute_file_name/2, although it is not necessary to call this predicate before using file systems operations.

The following file path predicates are provided:

absolute_file_name(F, G):
The predicate succeeds in G with the absolute file name of F. If F is already an absolute file name, then F is returned unchanged, otherwise the F is resolved against the Prolog flag base_url.

is_absolute_file_name(F):
The predicate succeeds if F is an absolute file name. A file name is considered absolute if it starts with a file separator or if it contains a protocol separator before the first file separator or in itself.

file_directory_name(F, G):
The predicate succeeds in G with the directory name of the file name F. The trailing file separator is included in the directory name.

file_base_name(F, G):
The predicate succeeds in G with the base name of the file name F. File names with trailing file separator return empty base name.

Section "database"

Besides input/output, the dynamic database of a Prolog system provides further means to have side effects by Prolog code. Currently we do compile Prolog clauses that are dynamically asserted, which incurs a little cost, but provides faster retrieval.

The following database predicates are provided:

dynamic I: [ISO 7.4.2.1]
The predicate succeeds. As a side effect, the predicate I Is touched.
asserta(C): [ISO 8.9.1]
The predicate succeeds. As a side effect, the clause C is inserted at the top.
assertz(C): [ISO 8.9.2]
The predicate succeeds. As a side effect, the clause C is inserted at the bottom.
clause(H, B): [ISO 8.8.1]
The predicate succeeds with the clauses that unify H :- B.
retract(C): [ISO 8.9.3]
The predicate succeeds with the clauses that unify C. As a side effect, the clause is removed.
retractall(H): [TC2 8.9.5]
The predicate succeeds. As a side effect, the clauses that unify the head H are removed.
abolish(I): [ISO 8.9.4]
The predicate succeeds. As a side effect, the predicate I is destroyed.
current_predicate(I): [ISO 8.8.2]
The predicate succeeds in I with current predicate indicators.
predicate_property(I, P):
The predicate succeeds in P with the properties of the predicate indicator I. The following properties are supported:

static: The predicate is static.
dynamic: The predicate is dynamic.
sys_multifile(S): The predicate is marked multifile in the source path S.
sys_usage(S): The predicate is defined in the source path S.
sys_discontiguous(S): The predicate is marked discontinuous in the source path S.

3.2 File "machine"

This file contains algorithms such as the Prolog engine.

- **Section “control”:** The Dogelog player provides already correct handling of a conjunction and disjunction of goals in a Prolog clause body or in a Prolog query.

- **Section "signal":** Situations might demand that a secondary thread controls a primary thread. The programming interface allows raising a soft signal.

- **Section "timer":** With the introducing of the async API the Prolog interpreter is able to process promises when in async mode.

- **Section "operators":** For syntax operators we use Prolog facts.

- **Section "statistics":** The Dogelog player allows querying and updating various parameters of its own environment or the host language environment.

Section "control"

The Dogelog player provides already correct handling of a conjunction and disjunction of goals in a Prolog clause body or in a Prolog query. This also extends to the cut. The conjunction and disjunction do not have a direct built-in since they are translated into internals. In particular the cut is mapped to '$CUT'/1 with an additional argument.

Example:

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

The interpreter has the capability to interrupt its control 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(_, _)
warning(_, _)
cause(_, _)
```

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 only catch non-urgent exceptions and match the head of a chained exception.

The following control built-ins are provided:

true: [ISO 7.8.1]
 The predicate succeeds.
A; B: [ISO 7.8.6]
 The predicate succeeds whenever A or B succeed.
A -> B: [ISO 7.8.7]
 The predicate succeeds when A succeeds and then whenever B succeed.
A, B: [ISO 7.8.5]
 The predicate succeeds whenever A and B succeed.
!: [ISO 7.8.4]
 The predicate removes choice points created in the current clause.
fail: [ISO 7.8.2]
 The built-in fails.
catch(G, E, F): [ISO 7.8.9]
 The built-in succeeds whenever G succeeds. If there was a non-urgent exception that unifies with E, the built-in further succeeds whenever F succeeds.
throw(E): [ISO 7.8.9]
 The predicate possibly fills the stack trace and then raises the exception B.

Section "signal"

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

```
error(system_error(_),_)
```

The secondary thread is free to signal whatever message it likes. Among the errors the Prolog system recognizes urgent errors as above. Urgent errors are automatically passed down in a catch/3. An example of such an urgent error is the error issued by the predicate abort/1. Urgent errors can still be catched by the predicate sys_trap/3.

The error terms are displayed using the multilingual strings facility. Whereby strings acting as formatting templates are used when the error term has further parameters. Multilingual strings can be retrieved by the predicates get_string/[2,3]. Multilingual strings can be defined by extending the predicate string/3. Locales are assumed underscore separated.

The following signal predicates are provided:

must(A):
: The predicate succeeds once if A succeeds. Otherwise, the predicate throws an error.

chain(A, B):
: The predicate succeeds whenever A and B succeed. If A throws an exception, then B is called once and an exception is chained.

abort:
: The predicate throws a system error of type user abort.

once_cleanup(G, C):
setup_once_cleanup(S, G, C):
: The predicate succeeds once if G succeeds. The clean-up C is called when G fails, succeeds or throws an exception. The ternary predicate permits an initial shielded call of a setup S.

shield(G):
: The predicate succeeds whenever the goal G succeeds. The goal is executed without auto-yield.

unshield(G):
: The predicate succeeds whenever the goal G succeeds. The goal is executed with auto-yield.

get_string(K, V):
get_string(K, L, V):
: The predicate succeeds in V with the value for the key K. The ternary predicate allows overriding the current locale by L.

strings(K, L, V):
: The predicate succeeds in the key K, the value V and the locale L. The predicate can be extended by libraries and applications.

Section "timer"

With the introducing of the async API the Prolog interpreter is able to process promises when in async mode. So far sleep/1 is the first predicate that relinquishes control. Further opportunities for the future will be input/output predicates. Last but not least the frequent garbage collection can be used to relinquish control on a regular basis.

The following async predicates are provided:

sleep(T):
The predicate suspends execution for T milliseconds.

call_later(G, T):
The built-in schedules the goal G to be executed after T milliseconds.

create_task(G):
The built-in schedules the goal G to be executed.

time_out(G, T):
The predicate succeeds once if G succeeds. If the goal has not terminated after T milliseconds a time limit system error is signalled to the goal.

Section "operators"

To lookup syntax operators we use ordinary Prolog facts. The predicate current_op/3 queries the current system and user syntax operators. The predicate op/3 defines a new operator or updates an existing operator.

The following syntax operators are predefined:

Table 2: Predefined Syntax Operators

Level	*Mode*	*Operators*
1200	fx	[:-, ?-, -->]
1200	xfx	[:-]
1150	fx	[dynamic, discontiguous, multifile]
1100	xfy	[;]
1050	xfy	[->]
1000	xfy	[',']
900	fy	[\+]
700	xfx	[is, =, =.., <, =<, =\=, >=, >, =:=, @<, @=<, \==, @>=, @>, ==]]
600	xfy	[:]
500	yfx	[+, -, /\, \/]
400	yfx	[*, /, //, rem, xor, >>, <<, div, mod]
200	xfx	[**]
200	xfy	[^]
200	fy	[-, \]

The following syntax operator predicates are provided:

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

op(L, M, O): [ISO 8.14.3]
The predicate succeeds. As a side effect, a new operator O with mode M and level L is asserted.

Section "statistics"

The Dogelog player allows querying and updating various parameters of its own environment or the host language environment. The predicate current_prolog_flag/2 allows reading a couple of Prolog flags. There is no predicate yet to change a Prolog flag. The convenience time/1 will call a given goal and show differential statistics on the standard output.

The following statistics predicates are provided:

current_prolog_flag(K, V): [ISO 8.17.2]

The predicate succeeds for the value V of the flag K. The following flags are available:

bounded: The integer domain type.
max_code: The maximum character code.
dialect: The current language dialect.
single_quotes: The interpretation of single quoted tokens.
double_quotes: The interpretation of double quoted tokens.
back_quotes: The interpretation of back quoted tokens.
version: The release integer.
version_data: The compound with Prolog system version components.
iso: The iso compatibility mode.
max_arity: The maximum arity of a compound.
base_url: The current working directory.
stage: The current stage.
system_url: The system library path.
emulator_url: The current emulator library path.
async_mode: The current async mode.
allow_yield: The current yield permission.
foreign_ext: The native libraries file name extension.
sys_locale: The current locale.
host_info: The compound with host language version components.

sys_time_atom(F, T, A):

If A is a variable, the built-in succeeds in A with the millisecond time T formatted by the pattern F. Otherwise the built-in succeeds in T with the millisecond time parsed from A by the pattern F. The following time field specifiers are supported:

%Y: Year as a 4-digit decimal number.
%m: Month as a 2-digit decimal number.
%d: Day as a 2-digit decimal number.
%H: Hour as a 2-digit decimal number.
%M: Minute as a 2-digit decimal number.
%S: Second as a 2-digit decimal number.

statistics(K, V):

The predicate succeeds for the value V of the flag K. The following flags are available:

time: The elapsed real time.
wall: The wall clock time.
gctime: The garbage collection time.
calls: The call count.

time(A):

The predicate succeeds whenever the goal A succeeds. As a side effect, timing of the call or redo is shown.

3.3 File “special”

This file captures host language code that was genuinely written. Its main goal is to provide natively implemented ISO core standard built-ins that help to realize the Prolog processor. In contrast to the file “machine”, which does not expose some Prolog callable built-ins, but provides host language routines like Prolog unification and execution.

- **Section "util":** The predefined predicates here mainly defined some convenience related to control flow.

- **Section “univ”:** Prolog terms are at the heart of a Prolog system. The Dogelog player is not different. Internally it uses host language objects to represent them.

- **Section “type”:** Not all Prolog terms in Dogelog require new host language objects. Some Prolog terms can be mapped to host language primitives.

- **Section “atom”:** We make the content of a host language string, which is also our datatype of a Prolog atom visible as a list of Unicode points.

- **Section "prologue":** The predefined predicates here mainly defined some convenience related to list processing.

Section "util"

In this section, we list common predefined Prolog predicates that are bootstrapped in Prolog itself with or without the help of predefined built-ins. The predefined predicates here mainly defined some convenience related to control flow. It is already possible to perform meta-calls that are allowed to contain conjunction, disjunction and the cut.

The following util predicates are provided:

repeat: [ISO 8.15.3]
  The predicate succeeds repeatedly indefinitely.
call(G): [ÍSO 7.8.3]
  The predicate succeeds whenever the goal G succeeds.
once(A): [ISO 8.15.2]
  The predicate succeeds once if A succeeds.
\+ A: [ISO 8.15.1]
  The predicate succeeds when A fails.

Section "univ"

Prolog terms are at the heart of a Prolog system. The Dogelog player is not different. Internally it uses host language objects that represent Prolog terms. Visible for the user are the usual ISO core standard built-ins dealing with Prolog terms. The number of arguments of a Prolog term is only limited by the possible size of a host language array.

The following built-ins are provided:

ground(T): [TC2 8.3.10]
The built-in succceeds if T is ground.

nonground(T, V):
The built-in succeeds if T is non-ground and V is the first variable.

term_variables(T, L): [TC2 8.5.5]
The built-in succeeds in L with the variables of T.

term_singletons(T, L):
The built-in succeeds in L with the singleton variables of T.

S = T: [ISO 8.2.1]
The built-in succeeds when the Prolog terms S and T unify, otherwise the built-in fails.

S \= T: [ISO 8.2.3]
The built-in succeeds when the Prolog terms S and T do not unify, otherwise the built-in fails.

copy_term(S, T): [ISO 8.5.4]
The built-in succeeds in T with a copy of S.

T =.. [F|L]: [ISO 8.5.3]
If T is a variable, the built-in succeeds in T with the Prolog term from the functor F and arguments L. Otherwise the built-in succeeds in F and L with the functor and arguments of the Prolog term T.

functor(T, F, A): [ISO 8.5.1]
If T is a variable, the built-in succeeds in T with a new Prolog term from the functor F and the arity A. Otherwise the built-in succeeds in F and L with the functor and arguments of the Prolog term T.

arg(K, X, Y): [ISO 8.5.2]
The predicate succeeds in Y with the K-th argument of X.

change_arg(K, X, Y):
The predicate succeeds. As a side-effect the K-th argument of X is set to Y.

Section "type"

Not all Prolog terms in Dogelog require new host language objects. Some Prolog terms can be mapped to host language primitives. For example, a Prolog number might have a complex mapping depending on the size. Subsequently host language routines are needed to provide the built-ins that check the ISO core standard types.

The following type built-ins are provided:

callable(C): [TC2 8.3.9]
The built-in succeeds if C is a Prolog compound, atom or reference. Otherwise, it fails.
var(V): [ISO 8.3.1]
The built-in succeeds if V is a Prolog variable. Otherwise, it fails.
nonvar(V): [ISO 8.3.7]
The built-in succeeds if V is not a Prolog variable. Otherwise, it fails.
compound(C): [ISO 8.3.6]
The built-in succeeds if V is a Prolog compound. Otherwise, it fails.
atomic(A): [ISO 8.3.5]
The built-in succeeds if A is a Prolog atom, number or reference. Otherwise, it fails.
atom(A): [ISO 8.3.2]
The built-in succeeds if A is a Prolog atom. Otherwise, it fails.
number(A): [ISO 8.3.8]
The built-in succeeds if A is a Prolog number. Otherwise, it fails.
integer(A): [ISO 8.3.3]
The built-in succeeds if A is a Prolog integer. Otherwise, it fails.
float(A): [ISO 8.3.4]
The built-in succeeds if A is a Prolog float. Otherwise, it fails.
reference(A):
The built-in succeeds if A is a Prolog reference. Otherwise, it fails.
acyclic_term(T): [TC2 8.3.11]
The predicate succeeds when the Prolog term T is an acyclic term, i.e. contains no cycles.

Section "atom"

We make the content of a host language string, which is also our datatype of a Prolog atom visible as a list of Unicode points. The string encoding of the host language is made transparent. The built-ins do not expose the string encoding. This translates also to the Prolog characters codes and atoms used by streams in the next section.

The following atom built-ins are provided:

atom_codes(A, L): [ISO 8.16.5]
: If A is a variable, the built-in succeeds in A with the for the Prolog list L. Otherwise the built-in succeeds in L with the Prolog list from the atom A.

char_code(C, N): [ISO 8.16.6]
: If C is a variable, the built-in succeeds in C with the character for the code N. Otherwise the built-in succeeds in N with the code from character C.

atom_number(A, N):
: If A is a variable, then the built-in succeeds in A with the atom for the Prolog number N. Otherwise the built-in succeeds in N with the Prolog number from the atom A.

atom_integer(A, R, N):
: If A is a variable, then the built-in succeeds in A with the atom for the Prolog integer N in radix R. Otherwise the built-in succeeds in N with the Prolog number from the atom A in radix R.

atom_reference(A, R):
: The built-in succeeds in A with the atom for the Prolog reference R.

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

atom_concat(X, Y, Z): [ISO 8.16.2]
: The built-in succeeds when Z is the concatenation of X and Y.

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.

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.

atom_split(A, D, L):
: The built-in succeeds when L is the split of the atom A by the delimiter D.

atom_arg(K, X, Y):
: The predicate succeeds in Y with the zero-based K-th code point of X.

Section "prologue"

In this section, we list common predefined Prolog predicates that are bootstrapped in Prolog itself with or without the help of predefined built-ins. The predefined predicates here mainly defined some convenience related to list processing.

The following lists predicates are provided:

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

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

number_codes(A, B): [ISO 8.16.8]
If A is a variable, then the predicate succeeds in A with the number from the Prolog list B. Otherwise the predicate succeeds in B with the Prolog list for the number A.

findall(T, G, L): [ISO 8.10.1]
The predicate succeeds in L with all T such that G succeeds.

list_to_set(L, S):
The predicate succeeds in S with the deduplication of L.

reverse(L, R):
The predicate succeeds in R with the reverse of L. Currently only implemented for mode reverse(+, -).

append(L, R, S):
The predicate succeeds whenever S unifies with the concatenation of L and R.

between(L, H, X):
The predicate succeeds for every integer X between L and H. Currently only implemented for mode between(+, +, -).

length(L, N):
The predicate succeeds with N being the length of the list L. Currently only implemented for mode length(+, -).

3.4 File "eval”

Prolog is not known for number crunching, nevertheless many Prolog systems provide a number type and corresponding operations and predicates. Besides floating-point numbers, the Dogelog player also supports arbitrary long integers.

- **Section "arithmetic":** Traversing arithmetic expressions in Prolog itself is usually not efficient. Nevertheless, this currently the approach for the Dogelog player.
- **Section "magnitude":** We support predicates that compare numbers by their magnitude, looking only at the value and not at the type.
- **Section "math":** We provide trigonometric and exponential evaluable functions from the ISO core standard and its corrigenda.
- **Section "bitwise":** We support bitwise evaluable functions as found in the ISO core standard and offer them for both smallint and bigint.
- **Section "syntactic":** Uninterpreted function symbols and constants lead to the Herbrand domain, which can be equipped with equality and comparison.

Section "arithmetic"

Providing efficient arithmetic is notoriously difficult for a Prolog system. The reason is that the ISO core standard defines arithmetic expressions. Traversing these in Prolog itself is usually not efficient. Nevertheless, this currently the approach for the Dogelog player.

The following arithmetic evaluable functions are provided:

-(A, B): [ISO 9.1.7]
The predicate succeeds in B with the negation of A.
+(A, B, C): [ISO 9.1.7]
The predicate succeeds in C with the sum of A and B.
-(A, B, C): [ISO 9.1.7]
The predicate succeeds in C with A subtracted by B.
***(A, B, C): [ISO 9.1.7]**
The predicate succeeds in C with the product of A and B.
/(A, B, C): [ISO 9.1.7]
The predicate succeeds in C with A float divided by B.
//(A, B, C): [ISO 9.1.7]
The predicate succeeds in C with A truncate divided by B.
rem(A, B, C): [ISO 9.1.7]
The predicate succeeds in C with A remainder B.
float(A, B): [ISO 9.17]
The predicate succeeds in B with the approximated A.
^(A, B, C): [TC2 9.3.10]
The predicate succeeds in C with A int power by B.
div(A, B, C): [TC2 9.1.3]
The predicate succeeds in C with A floor divided by B.
mod(A, B, C): [ISO 9.1.7]
The predicate succeeds in C with A modulus B.

The following arithmetic predicates are provided:

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

Section "magnitude"

We support predicates that compare numbers by their magnitude, looking only at the value and not at the type. This is in contrast to Prolog unification (=)/2 or Prolog syntactic equality (==)/2 which both take the value and the type into account, whereas (=:=)/2 does convert integer arguments to float before comparison, if at least one argument is float.

The following magnitude evaluable functions are provided:

abs(A, B): [ISO 9.1.7]
The predicate succeeds in B with the absolute value of A.
sign(A, B): [ISO 9.1.4]
The predicate succeeds in B with the sign of A.
min(A, B, C): [TC2 9.3.9]
The predicate succeeds in C with the minimum of A and B.
max(A, B, C): [TC2 9.3.8]
The predicate succeeds in C with the maximum of A and B.
truncate(A, B): [ISO 9.1.7]
The predicate succeeds in B with the truncate of A.
floor(A, B): [ISO 9.1.7]
The predicate succeeds in B with the floor of A.
ceiling(A, B): [ISO 9.1.7]
The predicate succeeds in B with the ceiling of A.
round(A, B): [ISO 9.1.7]
The predicate succeeds in B with the rounding of A.

The following magnitude predicates are provided:

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

Section "math"

The basic evaluable functions take integer and float arguments. The math evaluable functions take float arguments and automatically convert integer arguments to float. We provide math evaluable functions from the ISO core standard and its corrigendas.

The following math evaluable functions are provided:

sin(A, B): [ISO 9.3.2]
The predicate succeeds in B with the sine of A.
cos(A, B): [ISO 9.3.3]
The predicate succeeds in B with the cosine of A.
tan(A, B): [TC2 9.3.14]
The predicate succeeds in B with the tangent of A.
asin(A, B): [TC2 9.3.11]
The predicate succeeds in B with the arcus sine of A.
acos(A, B): [TC2 9.3.12]
The predicate succeeds in B with the arcus cosine of A.
atan(A, B): [ISO 9.3.4]
The predicate succeeds in B with the arcus tangent of A.
pi(A): [TC2 9.3.15]
The predicate succeeds in B with the bitwise not of A.
****(A, B, C): [ISO 9.3.1]**
The predicate succeeds in C with A float power by B.
exp(A, B): [ISO 9.3.5]
The predicate succeeds in B with e power by A.
log(A, B): [ISO 9.3.6]
The predicate succeeds in B with the natural logarithm of A.
sqrt(A, B): [ISO 9.3.7]
The predicate succeeds in B with the square root of A.
e(A): [N208 9.7.2]
The predicate succeeds in A with the Euler number.
epsilon(A): [N208 9.7.3]
The predicate succeeds in A with the machine epsilon.
atan2(A, B, C): [TC2 9.3.13]
The predicate succeeds in C with the arc tangent of A and B.

Section "bitwise"

The basic evaluable functions take integer and float arguments. The bitwise evaluable functions only take integer arguments. We support bitwise evaluable functions as found in the ISO core standard and offer them for both smallint and bigint.

The following bitwise evaluable functions are provided:

\(A, B): [ISO 9.4.5]
The predicate succeeds in B with the bitwise not of A.
/\(A, B, C): [ISO 9.4.3]
The predicate succeeds in C with the bitwise and of A and B.
\/(A, B, C): [ISO 9.4.4]
The predicate succeeds in C with the bitwise or of A and B.
xor(A, B, C): [TC2 9.4.6]
The predicate succeeds in C with the bitwise xor of A and B.
>>(A, B, C): [ISO 9.4.1]
The predicate succeeds in C with A shift right by B.
<<(A, B, C): [ISO 9.4.2]
The predicate succeeds in C with A shift left by B.

Section "syntactic"

Uninterpreted function symbols and constants lead to the Herbrand domain, which can be equipped with equality and comparison. We provide the usual ISO core standard predicates with a few restrictions. The Dogelog player does currently not yet support variable ordering or reference ordering, so that comparison throws an exception in these cases.

The following syntactic predicates are provided:

X == Y: [ISO 8.4.1]
    The built-in succeeds when X and Y are syntactically equivalent, otherwise fails.
X \== Y: [ISO 8.4.1]
    The built-in succeeds when X and Y are not syntactically equivalent, otherwise fails.
X @< Y: [ISO 8.4.1]
    The predicate succeeds when X is syntactically less than Y, otherwise fails.
X @>= Y: [ISO 8.4.1]
    The predicate succeeds when X is syntactically greater or equal to Y, otherwise fails.
X @> Y: [ISO 8.7.1]
    The predicate succeeds when X is syntactically greater than Y, otherwise fails.
X @=< Y: [ISO 8.7.1]
    The predicate succeeds when X is syntactically less or equal to Y, otherwise fails.
compare(C, X, Y): [TC2 8.4.2]
    The built-in succeeds in C with the syntactic comparison of X and Y.

3.5 File "runtime"

This file provides Unicode ready reading and writing of Prolog terms. Since reading and writing involves comparing and manipulating Unicode code points, the file also defines Prolog predicates to categories Unicode code points. For this purpose, the Python version and the JavaScript version contain a replica of the Java Unicode database.

- **Section "session":** t.b.d.

- **Section “stream”:** The Dogelog player currently does provide a stream abstraction. It also keeps a global state, for the current input and output streams.

- **Section “code”:** Communication over text streams is mainly done via codes or atoms. Directly mapping to the host language integers and strings.

- **Section "term":** Writing these routines in Prolog itself spares us from contrived host language code.

Section "session"

Unless inside a browser, Dogelog Player can be invoked from the command line. If a Prolog text <text> is supplied the Prolog system goes into scripting mode. In scripting mode, the Prolog system consults the Prolog text and then exits the Prolog system. The consulted Prolog text has access to the arguments <arg1> .. <argn> via the Prolog flag argv.

```
dogelog <text> <arg1> .. <argn>
```

If no Prolog <text> is supplied the Prolog system enters a Prolog top-level that consists of a read eval print loop (REPL). The Prolog system will read queries and show answer substitutions until it encounters a stream end of file (EOF) or the atom “end_of_file” in its input. The REPL will use some ASCII coloring to distinguish input/output.

The following session predicates are provided:

current_output(S): [ISO 8.11.2]
  The built-in succeeds in S with the current output.
current_error(S):
  The built-in succeeds in S with the current error.
current_input(S): [ISO 8.11.1]
  The built-in succeeds in S with the current input.
set_output(S): [ISO 8.11.4]
  The built-in succeeds. As a side effect the current output is set to S.
set_error(S):
  The built-in succeeds. As a side effect the current error is set to S.
set_input(S): [ISO 8.11.3]
  The built-in succeeds. As a side effect it sets the current input to S.
initialization(G):
  The predicate succeeds whenever G succeeds.
listing:
listing(I):
  The predicate lists the user clauses of the user predicates. The unary predicate allows specifying a predicate indicator.

Section "stream"

The Dogelog player currently does provide a stream abstraction. It also keeps a global state, for the current input and output streams. By now, we only support text streams and no byte streams. The output abstraction is a bounded buffer that is flushed when it reaches a hard-wired limit or when the flush_output/2 built-in is explicitly called.

As a further restriction we currently do not support stream aliases. There is currently only a predicate open/3, the stream result being a reference data type. It is up to the application to manage its streams. The Prolog system will not keep a table of some sort and therefore simplify garbage collection by the underlying host programming language.

The following stream built-ins are provided:

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

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

read: Open the path for reading.
write: Open the path for writing, truncate when it already exists.
append: Open the path for writing, seek to end when it already exists.

close(S): [ISO 8.11.6]

The built-in succeeds. As a side effect, the stream S is closed.

stream_property(S, P): [ISO 8.11.8]

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

line_no(L): The current line number L for a read stream.

file_property(S, P):

The predicate succeeds with the properties of the file S that unify with P. The following file properties are support:

type(T): The type T of the file, possible values “regular”, “directory” and “other”.
real_path(Q): The real path Q for the file.
last_modified(L): The last modified date in milliseconds of the file.

set_file_property(S, P):

The predicate assigns the property P to the file F. Currently supports the file property last_modified/1 for file system paths.

Section "code"

Communication over text streams is mainly done via codes or atoms. We therefore provide the ISO core standard predicate put_code/2 to send a code point and the ISO core standard predicates get_code/2 and peek_code/2 to receive a code point. Currently the default is that the stream is UTF-8 coded and code points are then Unicode.

As a convenience we also provide the predicates put_atom/[1,2] and get_atom/[2,3]. These predicates allow sending and receiving multiple code points at once in the form of an atom. Common stop code points are 0'\n to read lines and -1 to read entire files. The codes and atoms directly mapping to the host language integers and strings.

The following code predicates are provided:

nl: [ISO 8.12.3]
nl(S): [ISO 8.12.3]
The predicate succeeds. As a side effect, a newline is written. The unary predicate allows specifying an output stream S.

flush_output: [ISO 8.11.7]
flush_output(S): [ISO 8.11.7]
The predicate succeeds. As a side effect, the current output is flushed. The unary predicate allows specifying an output stream S.

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

put_atom(S):
put_atom(S, A):
The built-in succeeds. As a side effect, it adds the atom A to the output stream. The binary predicate allows specifying an output stream S.

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

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

get_atom(C, A):
get_atom(S, C, A):
The built-in succeeds in A with the atom from the input stream up to the code point C. The ternary predicate allows specifying an input stream S.

code_type(C, T):
The predicate succeeds in T with the code type of C.

code_numeric(C, V):
The predicate succeeds in V with the Unicode code numeric value of C, in case it is integer and between 0 and 35. Otherwise, the predicate succeeds in V With -1.

current_lastcode(S, C):
The built-in succeeds in C with the last code point of the output stream S.

set_lastcode(S, C):
The built-in succeeds. As a side effect, the last code point of the stream S is set to C.

Section "term"

Term input/output occupies the greatest part of the loader file, since we have written these routines in Prolog itself. This approach spares us from producing contrived host language code. The Dogelog player provides the general term input/output predicates read_term/[2,3] and write_term/[2,3], together with its specializations as defined in the ISO core standard.

The numbervars/1 write option does nothing, instead the annotation/1 option was introduced to support the library(beautify). Whereas numbervars/1 write option was supposed to handle ‘$VAR’/1 compounds, the annotation/1 read and write option allows ‘$STR’/1, ‘$CHR’/1 and ‘$RDX’/2 to read and write the corresponding tokens verbatim.

The following input/output predicates are provided:

write(T): [ISO 8.14.2]
write(S, T): [ISO 8.14.2]
The predicate succeeds. As a side effect, the term T is written. The binary predicate allows specifying an output stream S.

writeq(T): [ISO 8.14.2]
writeq(S, T): [ISO 8.14.2]
The predicate succeeds. As a side effect, the term T is written with quoted strings. The binary predicate allows specifying an output stream S.

write_canonical(T): [ISO 8.14.2]
write_canonical(S, T): [ISO 8.14.2]
The predicate succeeds. As a side effect, the term T is written with quoted strings and ignored operators. The binary predicate allows specifying an output stream S.

write_term(T, O): [ISO 8.14.2]
write_term(S, T, O): [ISO 8.14.2]
The predicate succeeds. As a side effect, the term T is written with options O. The ternary predicate allows specifying an output stream S. The available options are:

variable_names(N): The variable names N.
quoted(B): The quoted strings flag B.
ignore_ops(B): The ignore operators flag B.
priority(L): The operator priority L.
numbervars(B): The number vars flag B.
annotation(B): The annotation flag B.

read(E): [ISO 8.14.1]
read(S, E): [ISO 8.14.1]
The predicate succeeds in E with the read term or end_of_file. As a side effect, the input position is advanced. The binary predicate allows specifying an input stream.

read_term(E, O): [ISO 8.14.1]
read_term(S, E, O): [ISO 8.14.1]
The predicate succeeds in E with the read term or end_of_file and in O with the options. As a side effect, the input position is advanced. The ternary predicate allows specifying an input stream. The available options are:

variable_names(N): The variable names N.
singletons(A): The singleton variable names A.
priority(L): The operator priority L.
annotation(B): The annotation flag B.

4 Dogelog Entry

The Dogelog player supports a variety of host languages. This is possible since for each host language a large part of the Dogelog player is cross-compiled. In the following when we refer to a file, we refer to the file name without extension, but different incarnations will exist depending on host language. We use the following extensions:

- **.mjs Extension:** For the JavaScript version of the Dogelog player. It supports both JavaScript in the Browser and JavaScript from node.js.

- **.py Extension:** For the Python version of the Dogelog player. Since we use match/case in the implementation this requires at least Python 3.12.

- **.java Extension:** For the Java version of the Dogelog player. Since we use match/case in the implementation this requires at least JDK 21.

The below host language files are either result of cross compilation or genuinely written. They are among the files that define the Dogelog player and what a client needs to load. In uncanned mode the main entry point is the file index, which is supposed to have a rather stable public API. Whereas the other files might internally change:

- **File "index":** This file index is the main entry point of the Dogelog player. It is supposed to provide a rather stable application API.

4.1 File "index"

This file index is the main entry point of the Dogelog player. It is supposed to provide a rather stable public API. Currently the API provides both unattended and attended queries. We also allow the configuration of streams and calling the host language.

- **Section "api":** The basic API provides both unattended and attended queries, suitable for batch or online processing.

- **Section "foreign":** The Dogelog player facilitates defining Prolog predicates that call the host language or vice versa.

Section "api"

The basic API provides both unattended and attended queries, suitable for batch or online processing. The function init() need to be only called once. The user database is organized in stages and partitions, which have the values -1 and “system” during start-up. The init() function will set the current stage to 0 and the current partition to “user”.

The function consult_async() will consult a Prolog text whereas the function toplevel_async() will provide a Prolog top-level. The functions will run as tasks on top of the stack of the current Prolog engine and they will not create their own stack. These tasks can be aborted by posting a signal via the function post().

The following api calls are provided:

init(): (host language)
Initializes the Prolog system.
consult_async(T): (async host language)
Include the given text into the current stage.
toplevel_async(): (async host language)
Query the input stream into the current stage.
post(M): (host language)
Post the message M to the Prolog interpreter.
show(T): (host language)
Writes the Prolog term to the error stream.

Section "foreign"

The Dogelog player facilitates defining Prolog predicates that call the host language or vice versa. The host language call register() adds a host language function by the given predicate indicator. After registration, the host language function can be called from within Prolog. To call Prolog from the host language the call perform() is offered.

The following host language calls are provided:

FFI_FUNC: (host language)
    Option that says the host language function has a return value.
register(F,N,J,K): (host language)
    Add a host language function J with options K by the predicate indicator F/N to the knowledge base. The registration is staged, so clear() will remove.
perform(G): (host language)
    Run the Prolog term G once. The Prolog term must succeed. The goal is run with auto-yield disabled and promises are not accepted.
perform_async(G): (async host language)
    Run the Prolog term G once. The Prolog term must succeed. The goal is run with auto-yield enabled and promises are accepted.

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

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/