Demonstrating (Hybrid) Active Logic Documents and the Ciao Prolog Playground, and an Application to Verification Tutorials

[1,2]Daniela Ferreiro, [1,2]José F. Morales, [3] Salvador Abreu and [1,2]Manuel V. Hermenegildo
[1] Universidad Politécnica de Madrid (UPM) and [2]IMDEA Software Institute, Madrid, Spain
[3]NOVA LINCS / University of Évora, Portugal
{daniela.ferreiro,josef.morales,manuel.hermenegildo}@imdea.org; spa@uevora.pt

We propose a demonstration of the Active Logic Documents (ALDs) approach and the Ciao Playground, as well as a recent extension to ALDs to facilitate the integration of other tools into the system for creating Hybrid Active Logic Documents (HALD), and a concrete application of these technologies.

Active Logic Documents (ALD) [10, 11] are web pages which incorporate embedded Prolog engines. ALD's are generated using the LPdoc documentation generation system [7, 6], extended to be able to include embedded Ciao Playground cells in documents, so that interaction with Prolog is possible within the documents produced (Fig. 1). Documents can contain integrated environments that allow for the processing of code blocks, including compiling, parsing, and analyzing them. This facilitates the creation of web-based materials with editable and runnable educational resources such as activities, runnable examples, programming exercises, etc. With the ability to directly edit and evaluate code within the document, these interactive documents can act as oracles, providing valuable feedback and supporting self-evaluation mechanisms. In comparison with other tools [14, 2, 1], two fundamental aspects of the Active Logic Documents approach are that a) all the reactive parts run locally on the user's web browser without any dependency on a central server or a local Prolog installation, and that b) output is generated from a single, easy to use source that can be developed with any editor. We argue that the ALD approach has multiple advantages from the point of view of scalability, low maintenance cost, security, privacy, ease of packaging and distribution, etc. over other approaches. As an example,[1] the ALD source and output for a simple Prolog exercise[2] is shown in Fig. 4. Active Logic Documents are used for development of materials for teaching logic programming [8], embedding runnable code and exercises in slides [3], manuals, etc., and in other projects, such as, for example, in the development of a Programming course for young children (around 10 years old) within the Year Of Prolog initiatives.

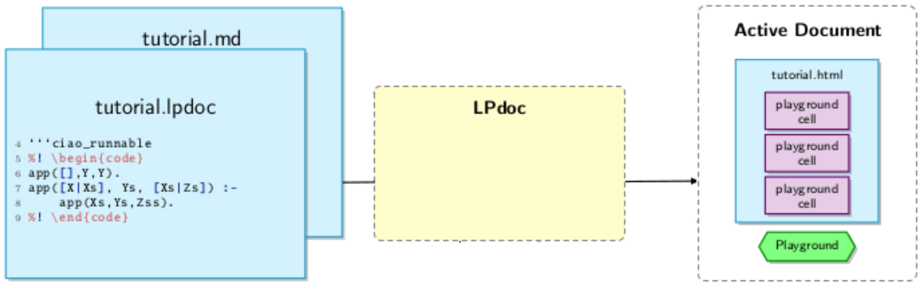
The Ciao Prolog Playground [3, 4] is a key component of our approach. In its standalone form,[4] it allows editing and running code locally on the user's web browser (See Fig. 2). To this end, the playground

[1]**Located for the reviewer's reference in an appendix, due to space limitations (i.e., not part of the submission).**

[2]http://ciao-lang.org/ciao/build/doc/ciao_playground.html/factorial_peano_iso.html

[3]E.g., Course material in Computational Logic: https://cliplab.org/~logalg

[4]https://ciao-lang.org/playground

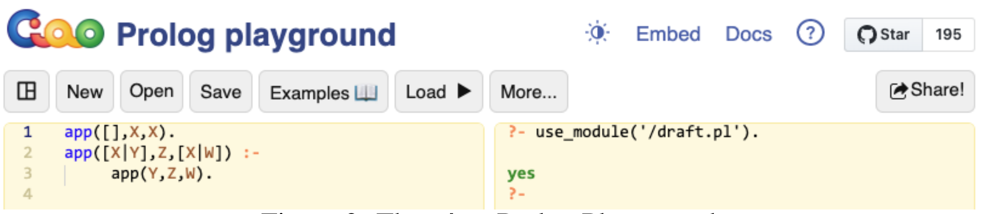


```
'''ciao_runnable
%! \begin{code}
app([],Y,Y).
app([X|Xs], Ys, [Xs|Zs]) :-
    app(Xs,Ys,Zss).
%! \end{code}
```

Figure 1: Generating Active Logic Documents.

Submitted to:
ICLP 2023

© D. Ferreiro, J.F. Morales, S. Abreu, and M.V. Hermenegildo
This work is licensed under the
Creative Commons Attribution License.



```
app([],X,X).
app([X|Y],Z,[X|W]) :-
    app(Y,Z,W).
```

```
?- use_module('/draft.pl').

yes
?-
```

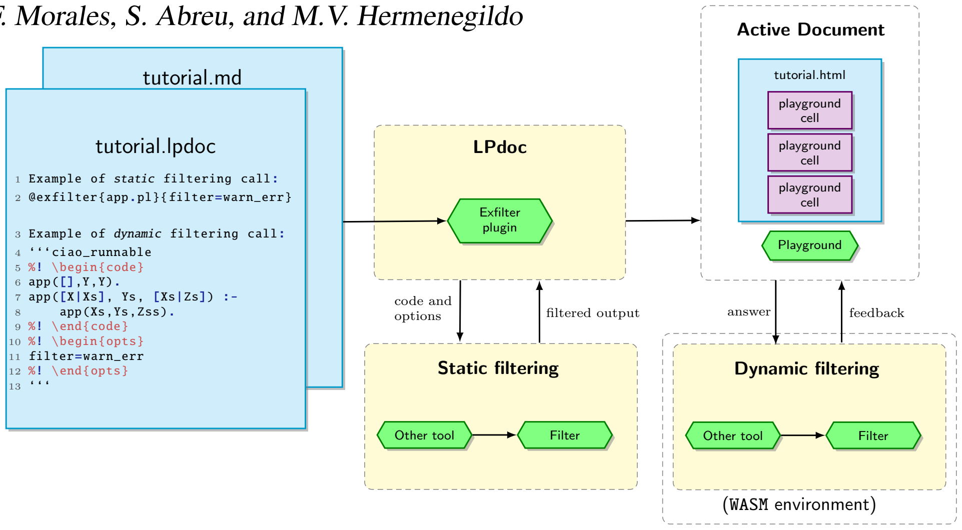
Figure 2: The `Ciao` Prolog Playground

uses modern Web technology (WebAssembly and Emscripten) to run an off-the-shelf Prolog engine and top level *directly in the browser*, with access to browser-side local resources, including a full interface with JavaScript that enables graphical input and output, interactivity, etc. This engine is contained in `Ciaowasm`, a `Ciao` Prolog bundle that compiles a variant of the standard `Ciao` engine with all necessary `Ciao` bundles and manuals. The main advantage of this general architecture over the widely used server-based approaches such as SWISH [14, 2] or Jupyter notebooks [1], is that it is easily reproducible and significantly alleviates maintenance effort and cost, as it essentially eliminates the server-side infrastructure. Previously, compilation of Prolog to JavaScript [9] enabled the use of Prolog to develop also the client side of web applications, running fully on the browser. This functionality is also provided by, e.g., Tau Prolog [13] and the tuProlog playground [5] which are recent Prolog interpreters written in JavaScript. However, the approach of compilation to WebAssembly and Emscripten allows the immediate reuse of existing, well-developed engines (which include many non trivial optimizations and features) and tooling.

Hybrid Active Logic Documents (HALDs). This addition to the ALD architecture, illustrated in Fig. 3, allows for the integration into ALDs of output from other tools, that can be run either at document generation time (to produce *static* content) or during the user interactions with the generated documents (producing the content *dynamically*). The overall input is, as in Fig. 1, a set of `LPdoc` source files (e.g., in `.md` markdown format or `.lpdoc` documentation files) including code blocks, narrative text, etc. In the **static phase**, center of Fig. 3, `LPdoc` scans and composes all these files, processing the different elements and inserting embedded Prolog playground instances (editors, engine instances, query blocks, etc.) as necessary. In addition, in the case of a *hybrid* document, one or more auxiliary tool(s) are additionally run by `LPdoc` while generating the ADL so that selected output from such tools can be incorporated. We refer to this process of projection of tool output as **filtering**. There are a number of library filters available for this purpose, and this set can be extended by the user. When `LPdoc` finds filtering calls (again, center of Fig.3, and see also Fig. 5), it sends requests to the corresponding tool (including code fragments, command line options, queries at its top level, etc.), and then *filters* the output obtained and incorporates this projected output statically into the document being produced. This is very useful for example when writing lecture notes, tutorials, manuals, etc.: when including, e.g., the results to a query or exercise, instead of pasting them in manually, these results can be generated automatically from the tool being used or documented. This ensures that the contents and format of the outputs produced by the tool are kept automatically in sync with any changes in the tool, which is a tedious and error-prone task without automation. This includes also for example obtaining from the tool the results to compare to in student exercises.

The **dynamic phase** (right of Fig.3) occurs once the HTML pages produced by `LPdoc` are deployed, and users have loaded them into their browsers. When processing some user input in one of the interactive elements of the pages –e.g., when a student issues a solution to an exercise–, the embedded playground framework issues the intervening filtering calls *dynamically*. This process involves loading the program, calling the corresponding tool (via command line interface or interacting with its top level), obtaining the output from this execution, and processing it with the specified filter in order to select the relevant part before presenting it to the user. Filtering can be customized to accommodate various types of tasks, such as

[5]`https://pika-lab.gitlab.io/tuprolog/2p-kt-web`



~~~
Example of static filtering call:
@exfilter{app.pl}{filter=warn_err}

Example of dynamic filtering call:
```ciao_runnable
%! \begin{code}
app([],Y,Y).
app([X|Xs], Ys, [Xs|Zs]) :-
    app(Xs,Ys,Zss).
%! \end{code}
%! \begin{opts}
filter=warn_err
%! \end{opts}
```
~~~

Figure 3: Overall architecture of hybrid ALDs.

"fill in the blanks" style exercises or to facilitate understanding and remediation of testing and verification failures, by displaying warning or error messages. Thus, students can test their code rapidly and effectively and receive useful feedback, all within the browser. As mentioned before, the embedded Prolog is a full system with all necessary Ciao bundles, manuals, etc, so that not just filtering but also the tool called or any user applications can be loaded and called to run locally on the browser, provided they are written in Prolog, or also of course JavaScript, compiled to wasm. Alternatively, calls can also be issued to tools available at some server or installed locally. Overall, the framework greatly facilitates the whole process, saving much coding.

An example application of HALDs. We will also present a concrete application in the generation of hybrid, interactive, web-based documentation, tutorials, courses, etc., for a *program verification tool*, CiaoPP [5, 12]. The aim is assisting students in learning how to use an advanced verification tool like CiaoPP, as well as learning more about Prolog. The HALD approach is a great aid here for adding solutions to exercises, examples, etc. automatically, while keeping the content synchronized with the system, greatly reducing the tracking of what documentation needs to be changed when an update to the system is made. CiaoPP [5, 12] performs several program debugging, analysis, and source-to-source transformation tasks for Prolog programs, and also for other high- and low- level languages. The output produced by CiaoPP generally contains significant amounts of information, including transformations, static analysis information, results of assertion checking or testing, counterexamples, etc. These results are typically presented as a new version of the source file annotated with (additional) assertions. The full analysis results produced by CiaoPP can be quite large, and cover the whole file or program. When writing the documentation it is interesting to show only a small fraction of this information at a time: the particular part that helps to understand the topic or step being explained. The filters are used here to extract, e.g., particular properties of a concrete predicate, particular types of assertions, etc. Analysis information can easily be embedded in human-readable explanations. A set of tutorials is available at https://ciao-lang.org/ciao/build/doc/ciaopp_tutorials.html/ and we also refer again to Fig. 5 in the appendix.

The **source code** of all components of our system is available at https://github.com/ciao-lang/, including the build/install instructions. When the Ciao playground is installed, all the bundles and documentation will be run and hosted on your localhost server. Additional examples and instructions can be found in the Ciao playground manual [3] [6]. Although we have developed and described our work for concreteness in the context of the Ciao system, we believe the proposed mechanisms are of general applicability and readily adaptable to other systems.

[6]https://ciao-lang.org/ciao/build/doc/ciao_playground.html/

References

[1] Anne Brecklinghaus & Philipp Koerner (2022): *A Jupyter Kernel for Prolog*. In: *Proc. 36th Workshop on (Constraint) Logic Lrogramming (WLP 2022)*, Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn.

[2] Peter Flach, Kacper Sokol & Jan Wielemaker (2023): *Simply Logical - The First Three Decades*. In David S. Warren, Veronica Dahl, Thomas Eiter, Manuel Hermenegildo, Robert Kowalski & Francesca Rossi, editors: *Prolog - The Next 50 Years*, *LNCS* 13900, Springer.

[3] G. Garcia-Pradales, J.F. Morales & M. V. Hermenegildo (2021): *The Ciao Playground*. Technical Report, Technical University of Madrid (UPM) and IMDEA Software Institute. Available at `http://ciao-lang.org/ciao/build/doc/ciao_playground.html/ciao_playground_manual.html`.

[4] G. Garcia-Pradales, J.F. Morales, M. V. Hermenegildo, J. Arias & M. Carro (2022): *An s(CASP) In-Browser Playground based on Ciao Prolog*. In: *ICLP'22 Workshop on Goal-directed Execution of Answer Set Programs*.

[5] M. Hermenegildo, G. Puebla, F. Bueno & P. Lopez Garcia (2005): *Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor)*. *Science of Computer Programming* 58(1–2), pp. 115–140.

[6] M. V. Hermenegildo (2000): *A Documentation Generator for (C)LP Systems*. In: *International Conference on Computational Logic, CL2000*, *LNAI* 1861, Springer-Verlag, pp. 1345–1361.

[7] M. V. Hermenegildo & J.F. Morales (2011): *The LPdoc Documentation Generator. Ref. Manual (V3.0)*. Technical Report, UPM. Available at `http://ciao-lang.org`.

[8] M.V. Hermenegildo, J.F. Morales & P. Lopez-Garcia (2023): *Some Thoughts on How to Teach Prolog*. In David S. Warren, Veronica Dahl, Thomas Eiter, Manuel Hermenegildo, Robert Kowalski & Francesca Rossi, editors: *Prolog - The Next 50 Years*, *LNCS* 13900, Springer. Available at `http://cliplab.org/papers/TeachingProlog-PrologBook.pdf`.

[9] J. F. Morales, R. Haemmerlé, M. Carro & M. V. Hermenegildo (2012): *Lightweight compilation of (C)LP to JavaScript*. *Theory and Practice of Logic Programming, 28th Int'l. Conference on Logic Programming (ICLP'12) Special Issue* 12(4-5), pp. 755–773.

[10] J.F. Morales, S. Abreu, D. Ferreiro & M.V. Hermenegildo (2022): *Teaching Prolog with Active Logic Documents*. Technical Report, Technical University of Madrid (UPM) and IMDEA Software Institute.

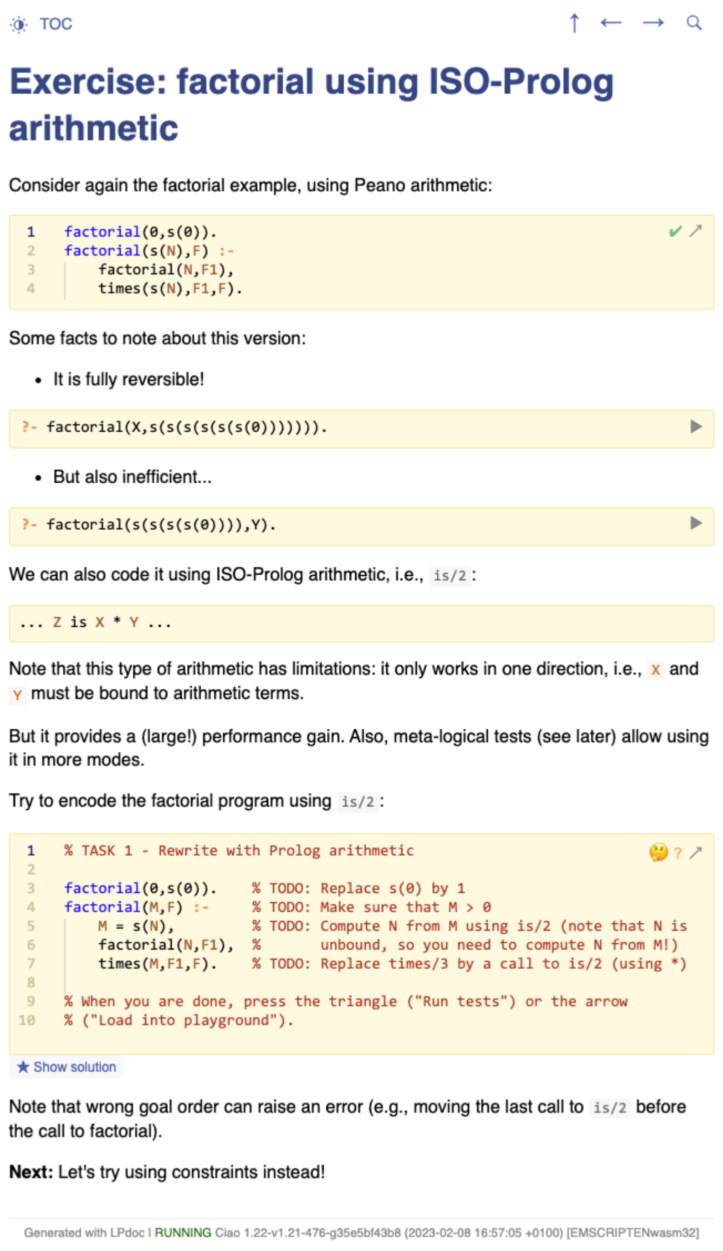
[11] J.F. Morales, Salvador Abreu, D. Ferreiro & M.V. Hermenegildo (2023): *Teaching Prolog with Active Logic Documents*. In David S. Warren, Veronica Dahl, Thomas Eiter, Manuel Hermenegildo, Robert Kowalski & Francesca Rossi, editors: *Prolog - The Next 50 Years*, *LNCS* 13900, Springer. Available at `http://cliplab.org/papers/ActiveLogicDocuments-PrologBook.pdf`.

[12] G. Puebla, F. Bueno & M. V. Hermenegildo (2000): *Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs*. In: *Logic-based Program Synthesis and Transformation (LOPSTR'99)*, *LNCS* 1817, Springer-Verlag, pp. 273–292, doi:10.1007/10720327_16.

[13] (2021): *τProlog — An open source Prolog interpreter in JavaScript*. `http://tau-prolog.org`. Last access: June 3, 2023.

[14] Jan Wielemaker, Fabrizio Riguzzi, Robert A. Kowalski, Torbjörn Lager, Fariba Sadri & Miguel Calejo (2019): *Using SWISH to Realize Interactive Web-based Tutorials for Logic-based Languages*. *Theory Pract. Log. Program.* 19(2), pp. 229–261, doi:10.1017/S1471068418000522. Available at `https://doi.org/10.1017/S1471068418000522`.

Appendix A – ALD example



Exercise: factorial using ISO-Prolog arithmetic

Consider again the factorial example, using Peano arithmetic:

```
factorial(0,s(0)).
factorial(s(N),F) :-
    factorial(N,F1),
    times(s(N),F1,F).
```

Some facts to note about this version:

- It is fully reversible!

```
?- factorial(X,s(s(s(s(s(s(0))))))).
```

- But also inefficient...

```
?- factorial(s(s(s(s(0)))),Y).
```

We can also code it using ISO-Prolog arithmetic, i.e., `is/2`:

```
... Z is X * Y ...
```

Note that this type of arithmetic has limitations: it only works in one direction, i.e., `X` and `Y` must be bound to arithmetic terms.

But it provides a (large!) performance gain. Also, meta-logical tests (see later) allow using it in more modes.

Try to encode the factorial program using `is/2`:

```
% TASK 1 - Rewrite with Prolog arithmetic

factorial(0,s(0)).    % TODO: Replace s(0) by 1
factorial(M,F) :-     % TODO: Make sure that M > 0
    M = s(N),         % TODO: Compute N from M using is/2 (note that N is
    factorial(N,F1),  %       unbound, so you need to compute N from M!)
    times(M,F1,F).    % TODO: Replace times/3 by a call to is/2 (using *)

% When you are done, press the triangle ("Run tests") or the arrow
% ("Load into playground").
```

Note that wrong goal order can raise an error (e.g., moving the last call to `is/2` before the call to factorial).

Next: Let's try using constraints instead!

Generated with LPdoc | RUNNING Ciao 1.22-v1.21-476-g35e5bf43b8 (2023-02-08 16:57:05 +0100) [EMSCRIPTENwasm32]

~~~~
\title Exercise: factorial using ISO-Prolog arithmetic

Consider again the factorial example, using Peano arithmetic:

```ciao_runnable
:- module(_, _, [assertions,library(bf/bfall)]).
%! \begin{focus}
factorial(0,s(0)).
factorial(s(N),F) :-
    factorial(N,F1),
    times(s(N),F1,F).
%! \end{focus}

nat_num(0).
nat_num(s(X)) :- nat_num(X).

times(0,Y,0) :- nat_num(Y).
times(s(X),Y,Z) :- plus(W,Y,Z), times(X,Y,W).

plus(0,Y,Y) :- nat_num(Y).
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).
```

Some facts to note about this version:
  - It is fully reversible!
```ciao_runnable
?- factorial(X,s(s(s(s(s(s(0))))))).
```
  - But also inefficient...
```ciao_runnable
?- factorial(s(s(s(s(0)))),Y).
```

We can also code it using ISO-Prolog arithmetic, i.e., `is/2`:
```ciao
 ... Z is X * Y ...
```
Note that this type of arithmetic has limitations: it only works in
one direction, i.e., `X` and `Y` must be bound to arithmetic terms.

But it provides a (large!) performance gain.  Also, meta-logical
tests (see later) allow using it in more modes.

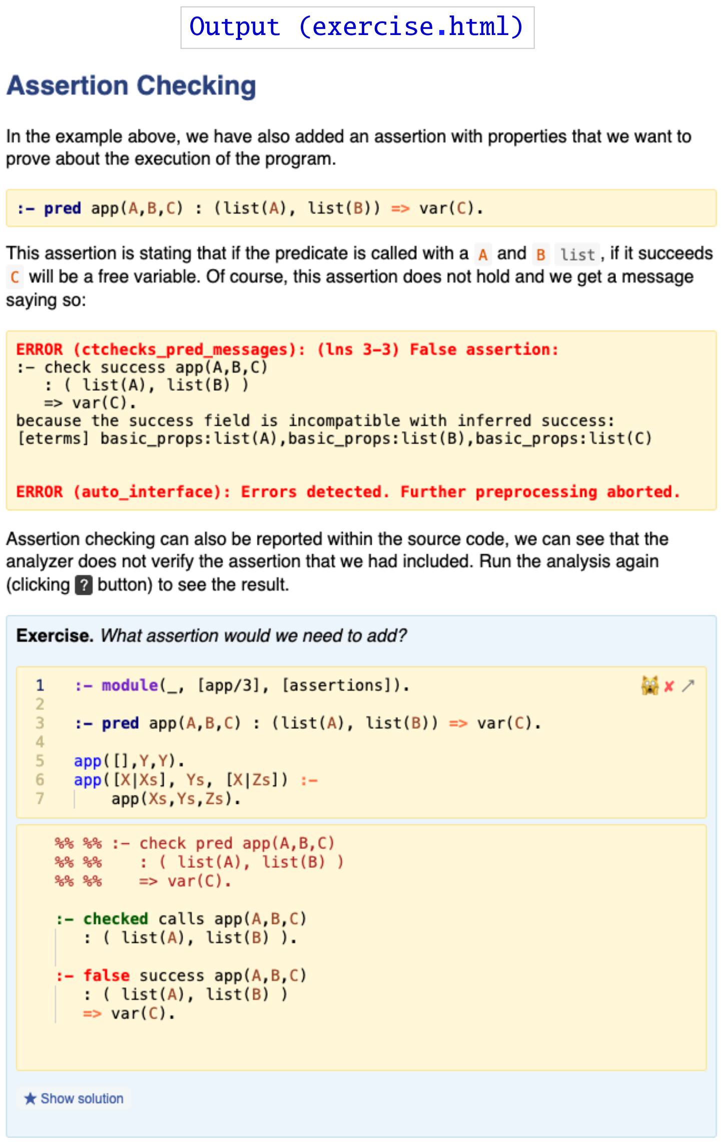
Try to encode the factorial program using `is/2`:
```ciao_runnable
:- module(_, _, [assertions]).
:- test factorial(5,  B) => (B = 120) + (not_fails, is_det).
:- test factorial(0,  0) + fails.
:- test factorial(-1, B) + fails.
%! \begin{hint}
% TASK 1 - Rewrite with Prolog arithmetic

factorial(0,s(0)).    % TODO: Replace s(0) by 1
factorial(M,F) :-     % TODO: Make sure that M > 0
    M = s(N),         % TODO: Compute N from M using is/2 (note that N is
    factorial(N,F1),  %       unbound, so you need to compute N from M!)
    times(M,F1,F).    % TODO: Replace times/3 by a call to is/2 (using *)

% When you are done, press the triangle ("Run tests") or the arrow
% ("Load into playground").
%! \end{hint}
%! \begin{solution}
factorial(0,1).
factorial(N,F) :-
    N > 0,
    N1 is N-1,
    factorial(N1,F1),
    F is F1*N.
%! \end{solution}
```

Note that wrong goal order can raise an error (e.g., moving the last
call to `is/2` before the call to factorial).

**Next:** Let's try using constraints instead!
~~~~

Figure 4: The full source and LPdoc output for the Active Logic Document for a simple factorial exercise.

Appendix B – HALD example



Output (exercise.html)

Assertion Checking

In the example above, we have also added an assertion with properties that we want to prove about the execution of the program.

```
:- pred app(A,B,C) : (list(A), list(B)) => var(C).
```

This assertion is stating that if the predicate is called with a `A` and `B` `list`, if it succeeds `C` will be a free variable. Of course, this assertion does not hold and we get a message saying so:

```
ERROR (ctchecks_pred_messages): (lns 3-3) False assertion:
:- check success app(A,B,C)
   : ( list(A), list(B) )
   => var(C).
because the success field is incompatible with inferred success:
[eterms] basic_props:list(A),basic_props:list(B),basic_props:list(C)

ERROR (auto_interface): Errors detected. Further preprocessing aborted.
```

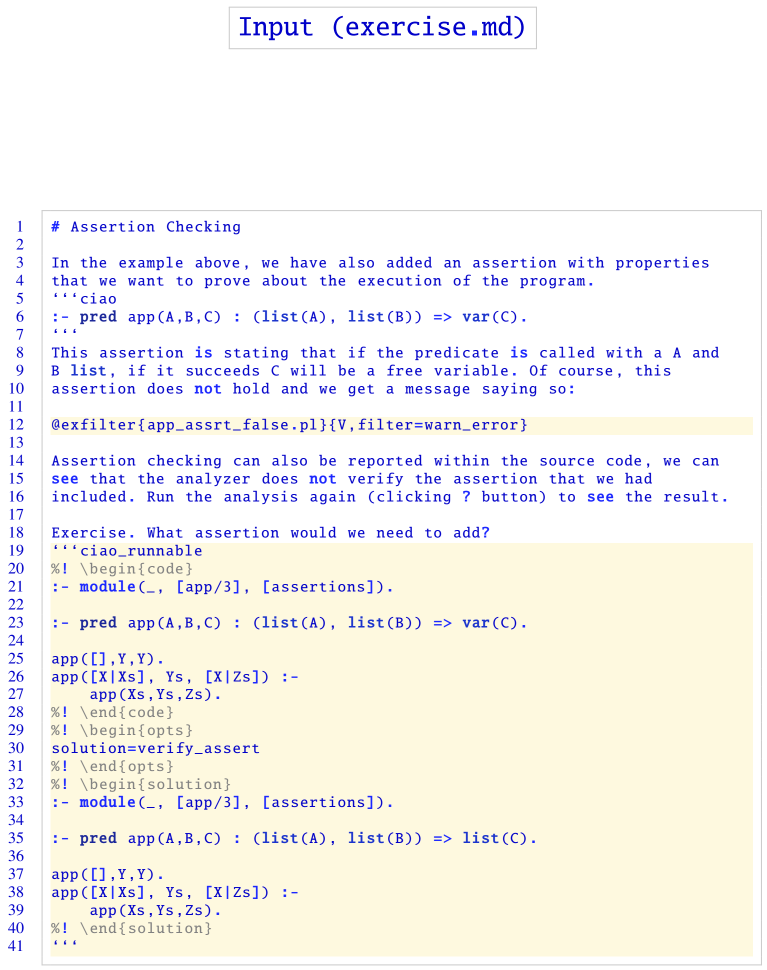
Assertion checking can also be reported within the source code, we can see that the analyzer does not verify the assertion that we had included. Run the analysis again (clicking ? button) to see the result.

Exercise. *What assertion would we need to add?*

```
:- module(_, [app/3], [assertions]).

:- pred app(A,B,C) : (list(A), list(B)) => var(C).

app([],Y,Y).
app([X|Xs], Ys, [X|Zs]) :-
    app(Xs,Ys,Zs).
```

```
%% %% :- check pred app(A,B,C)
%% %%    : ( list(A), list(B) )
%% %%    => var(C).

:- checked calls app(A,B,C)
   : ( list(A), list(B) ).

:- false success app(A,B,C)
   : ( list(A), list(B) )
   => var(C).
```

★ Show solution



Input (exercise.md)

~~~~
# Assertion Checking

In the example above, we have also added an assertion with properties
that we want to prove about the execution of the program.
```ciao
:- pred app(A,B,C) : (list(A), list(B)) => var(C).
```
This assertion is stating that if the predicate is called with a A and
B list, if it succeeds C will be a free variable. Of course, this
assertion does not hold and we get a message saying so:

@exfilter{app_assrt_false.pl}{V,filter=warn_error}

Assertion checking can also be reported within the source code, we can
see that the analyzer does not verify the assertion that we had
included. Run the analysis again (clicking ? button) to see the result.

Exercise. What assertion would we need to add?
```ciao_runnable
%! \begin{code}
:- module(_, [app/3], [assertions]).

:- pred app(A,B,C) : (list(A), list(B)) => var(C).

app([],Y,Y).
app([X|Xs], Ys, [X|Zs]) :-
    app(Xs,Ys,Zs).
%! \end{code}
%! \begin{opts}
solution=verify_assert
%! \end{opts}
%! \begin{solution}
:- module(_, [app/3], [assertions]).

:- pred app(A,B,C) : (list(A), list(B)) => list(C).

app([],Y,Y).
app([X|Xs], Ys, [X|Zs]) :-
    app(Xs,Ys,Zs).
%! \end{solution}
```
~~~~

Figure 5: The full source and LPdoc output for a Hybrid Active Logic Document containing a simple assertion checking exercise. Note the use of a filtering command in the source to call CiaoPP during the static phase and select certain outputs that are incorporated in the page (the box with the error messages). Also, an interactive exercise is embedded and filtering is used again (this time dynamically) to present a selected part of the analysis output for the program entered by the student and to compare it to the expected results.