Teaching Prolog with Active Logic Documents *

Jose F. Morales[1,2], Salvador Abreu[3], Daniela Ferreiro[1,2], and Manuel V. Hermenegildo[1,2]

[1] Universidad Politécnica de Madrid (UPM)
[2] IMDEA Software Institute
[3] NOVA LINCS / University of Évora
{josef.morales,daniela.ferreiro,manuel.hermenegildo}@imdea.org
spa@uevora.pt

Abstract. Teaching materials for programming languages, and Prolog in particular, classically include textbooks, slides, notes, and exercise sheets, together with some Prolog programming environment. However, modern web technology offers many opportunities for embedding interactive components within such teaching materials. We report on our experiences in developing and applying our approach and the corresponding tools to facilitating this task, that we call Active Logic Documents (ALD). ALD offers both a very easy way to add click-to-run capabilities to any kind of teaching materials, independently of the tool used to generate them, as well as a tool-set for generating web-based materials with embedded examples and exercises. Both leverage on (components of) the Ciao Prolog Playground. Fundamental principles of our approach are that active parts run locally on the student's browser, with no need for a central infrastructure, and that output is generated from a single, easy to use source that can be developed with any editor. We argue that this has multiple advantages from the point of view of scalability, low maintenance cost, security, ease of packaging and distribution, etc. over other approaches.

Keywords: Active Logic Documents, Prolog Playgrounds, Teaching Prolog, Prolog, Ciao-Prolog, Logic Programming, Web, Literate Programming.

1 Introduction

Teaching programming languages traditionally relies on an array of dispersed materials, such as textbooks, class notes, slides, or exercise sheets, as well as some programming environment(s) for students to run programs. Teaching Prolog is of course no exception. More recently, web-based technology has been facilitating the combination or embedding of interactive components into such

* Partially funded by MICINN projects PID2019-108528RB-C21 *ProCode*, TED2021-132464B-I00 *PRODIGY*, and FJC2021-047102-I, by the Comunidad de Madrid program P2018/TCS-4339 *BLOQUES-CM*, by FCT under strategic project UIDB/-04516/2020 (NOVA LINCS) and by the Tezos foundation. The authors would also like to thank the anonymous reviewers for very useful feedback on previous drafts of this paper.

teaching materials. This, however, poses a number of challenges, since there are multiple possible approaches to this end, and new technologies are constantly appearing that offer different trade-offs and capabilities. In this paper we report on our experiences in developing and applying two approaches and the corresponding tools in order to facilitate this task, that we collectively call *Active Logic Documents* – ALD, and which we believe offer interesting advantages over other approaches.

Mixing text and code has long been a topic of research and development, largely stemming from Knuth's seminal Literate Programming [10] concept. However, packaging and distribution of hybrid text and code systems has traditionally been complicated by dependencies on specific working environments, such as, for instance, the need for a specific operating system or even a specific version thereof, the availability of specific support software, library dependencies, etc. Because of this, over the years, several efforts have been made to provide online learning platforms such as the Khan Academy [13] which also strives to present teaching materials in a game-like form, and the idea has more recently materialized in web-based platforms, as exemplified by Jupyter notebooks [4]. This modern web technology affords dynamic and multimedia components, which clearly make teaching materials more palatable. In the Prolog world, SWISH provides a web-based platform for producing notebook-like sites that has been used to create online courses and exercises for logic-based programming languages [15]. Flach et al. [3] offer a very interesting account of their efforts to create progressively more interactive versions of their book, including combinations with Jupyter notebooks and with SWISH. Independently, Brecklinghaus et al. [2] implement a Jupyter kernel for SICStus Prolog and SWI-Prolog.

All these systems, however, rely on a server-side platform. Although this is in principle convenient to the end user, server-centric architectures also have drawbacks, e.g.: they introduce a dependency on the server; maintaining a server-side infrastructure can represent a significant burden; the user content built on such a platform is tied to the availability and reachability of such platform; the approach may also affect other aspects, such as scalability or privacy; etc.

In contrast, the fundamental principles of our ALD approach are that the reactive parts of the materials (the Prolog code written by the course developer or the student and all the related functionality) run locally on the student's web browser, with no need for a central infrastructure, and that the output is generated from a single, easy to use source that can be developed with any editor. We argue that this approach has multiple advantages from the point of view of scalability, low maintenance cost, security, independence from unconventional tools, etc. over other approaches. Our tools, described in the following sections, are meant to help course developers in at least two basic scenarios:

– Some course developers prefer to develop (or have already developed) their teaching materials with their own tools (such as, e.g., LaTeX, PowerPoint, Pages, Word, etc.), which have been stable for a long time, and may be reluctant to port these materials to a bespoke platform. For this case we

[4] `https://jupyter.org/`

Fig. 1. The Ciao Playground

offer a "click-to-run" methodology, based on the Ciao Prolog playground, which we describe in Section 2. This provides a very easy way to incorporate click-to-run capabilities in any kind of teaching materials, independently of the tool used to generate them or the generated format (pdf, html, etc.), and with no dependencies on any central server.

– For course developers that are willing to port their materials, we offer a tool (an extension of the LPdoc documenter) that greatly facilitates generating, using any conventional editor, web-based materials with embedded examples and exercises. These will run locally on the student's browser, again with no dependencies on any central server. We describe this part of our approach in Section 3.

2 Embedding Runnable Code in Documents via Browser-based "Click-to-Run"

A common method for adding interactivity to teaching materials is the *"click to run"* approach. Code blocks in such materials become *clickable* elements that load the code into a suitable environment for online execution. This functionality has been traditionally supported by server-side playgrounds or notebooks, where the code is run on a server and the examples need to be loaded and saved on that server. In contrast, our approach incorporates two aspects that depart from these classical methods: the first one is that, as mentioned before, code execution is performed fully on the browser; the second one is that examples are stored in the documents[5] themselves, with no need to previously upload them or have them stored in remote servers.

The main component providing such functionality in our approach is the Ciao Playground [6] [4,5] which allows editing and running code locally on the user's web browser (See Figure 1). To this end, the playground uses modern Web technology (WebAssembly and Emscripten, see Section 5) to run an off-the-shelf Prolog engine and top level *directly in the browser*, able to fully access browser-side local resources. The main advantage of this general architecture is that it is easily reproducible and significantly alleviates maintenance effort and cost, as it essentially eliminates the server-side infrastructure.

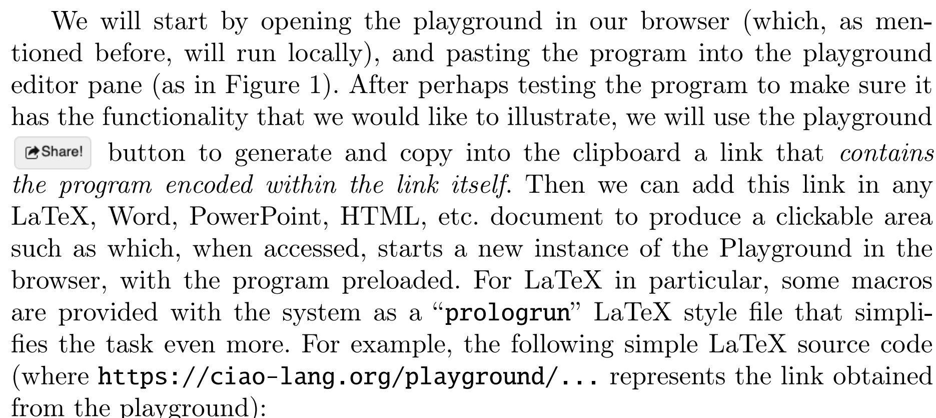
[5] By "document" we mean the actual document (in pdf, or XML, etc.) which has been produced by the course writer and which is being read by the student.

[6] `https://ciao-lang.org/playground`

In addition to the previously mentioned functionality, the playground provides an easy way to embed short code snippets (or links to larger source code) in web links themselves. These links can then be stored within documents and passed on as Prolog code to the playground, to be locally executed on the student's browser. This approach makes it very easy to include runnable code in manuals, tutorials, slides, exercises, etc., provided they are in a format that has support for HTML links, such as pdf files, and also Google Docs, Jupyter notebooks, Word, PowerPoint, LaTeX, Pages, Keynote, Org mode, or web site generators. Additionally, links can be easily shared by email or instant messaging applications.

For example, assume that we would like to include in the teaching materials being developed the classic append program:

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



We will start by opening the playground in our browser (which, as mentioned before, will run locally), and pasting the program into the playground editor pane (as in Figure 1). After perhaps testing the program to make sure it has the functionality that we would like to illustrate, we will use the playground Share! button to generate and copy into the clipboard a link that *contains the program encoded within the link itself.* Then we can add this link in any LaTeX, Word, PowerPoint, HTML, etc. document to produce a clickable area such as which, when accessed, starts a new instance of the Playground in the browser, with the program preloaded. For LaTeX in particular, some macros are provided with the system as a "`prologrun`" LaTeX style file that simplifies the task even more. For example, the following simple LaTeX source code (where `https://ciao-lang.org/playground/...` represents the link obtained from the playground):

```
1 \codelink{https://ciao-lang.org/playground/...}
2 \begin{prologrun}
3 app([],X,X).
4 app([X|Y],Z,[X|W]) :- app(Y,Z,W).
5 \end{prologrun}
```

is rendered as follows (including the "`run`" button):

```
1 app([],X,X).                              run ▶
2 app([X|Y],Z,[X|W]) :-
3       app(Y,Z,W).
```

The Playground is essentially a fully-fledged Prolog environment which includes much other functionality such as running tests, generating documentation, verifying program assertions, or specializing code, some of which will become instrumental in the following steps. In addition, specialized instances of the Playground can be easily created, an example of which is the s(CASP) play-

ground [5] [7]. More information on the implementation of the Ciao WebAssembly back end and the Playground architecture can be found in Section 5 and in [5] (and for the s(CASP) system in [1]).

3 Active Logic Documents

While click-to-run functionality is convenient and highly portable, we have also developed a more comprehensive tool (as an extension of the LPdoc documenter) that greatly facilitates the generation of web-based materials with *embedded* examples and exercises, using any conventional editor. These full-fledged *Active Logic Documents* are web pages with embedded Prolog programs, all sharing a common environment. The examples run on the pages themselves, in an embedded version of the playground, without the need for a separate playground tab.

Creating Documents with Editable and Runnable Code using LPdoc The basis of our approach is LPdoc [7,8], which pioneered automatic program documentation in the context of Logic Programming and (C)LP.[8] Its main application is the generation of reference manuals directly from the actual code (including any assertions used to formally describe predicates), as well as from comments in the `.pl` source files or dedicated `.lpdoc` documentation files. However, LPdoc is often also used to generate other kinds of documents, such as tutorials, and also web sites and other kinds of on-line linked documents. Like many other tools, such as LaTeX, or the Web itself, LPdoc uses a human-oriented *documentation format* [9] for *typesetting* and does not impose the use of a particular WYSIWYG editor.[10] In particular, LPdoc supports writing rich-text documents in *markdown* syntax, with standard features like the inclusion of *verbatim* text and code blocks, syntax highlighting, and more, which allows for the inclusion of code segments in the midst of fairly flexible structured text, with hyperlinks. The use of documentation generation systems to write whole reference manuals, books, and teaching materials has become quite widespread in the past years.

To realize the ADL approach, the key step was to enhance LPdoc with the possibility of embedding Prolog environments, based on the Ciao Playground, which opens up a wide degree of possibilities for interaction. With this step, documents with *embedded* editable and runnable examples can be generated easily using LPdoc. The source that the developer of the course, tutorial, etc. works with is one or more LPdoc source files, in, e.g., markdown format. LPdoc

[7] `https://ciao-lang.org/playground/scasp.html`

[8] Written in Prolog of course!

[9] Editors like MS-Word use non-human oriented document formats: bloated with metadata, often binary encoded and undocumented, almost impossible to modify and maintain without the original tools, and really hard to integrate with code-oriented version control systems.

[10] However, note that once the *markup* language is stable and well defined, it is perfectly possible to implement rich WYSIWIG front-ends that can save documents in this format. See for example Lyx, TeXmacs, etc. or rich-editors for Markdown.

processes these files and generates html pages in which the code fragments in the source are automatically rendered as editable panes that can be run in place in an embedded playground (as well as loaded into the standalone playground as before). The generated pages can then be published easily as any other web pages, e.g., in a course web site, in a personal `public_html` area, etc. Everything needed, including the runnable examples, queries, etc., is contained in the pages themselves. When students visit these pages with their browser, all executions occur locally in their browser.

Interaction facilities for Self-assessment Especially in the context of a self-taught Logic Programming course, the embedded playground approach allows for very rich interactions. That is, code can be evaluated and edited directly in the same document. This enables direct support for self-evaluation and testing mechanisms. For example, code examples allow automated "semantic" evaluation of user input, e.g., by running code tests on the solution provided by the student. Document-level dependencies between examples, topics, and sections, allow "gamification" (e.g., evaluating your progress, obtaining *points* and trophies, hiding/showing tasks, un-blocking levels, etc.) of the learning activities, ensuring that the reader can acquire the necessary confidence on basic topics before going on to more advanced concepts.

Moreover, the Prolog top-level loop which underlies the Playground can interpret terms which result from solutions to goals in more ways than just printing them out. Similarly to Prolog's `display/1` predicate, some terms may be interpreted as giving rise to graphical or other user-interface components.

4 A Simple Example: Coding Factorial

We now illustrate through a concrete, worked-out example, the process of creating documents with editable and runnable examples using LPdoc. We will develop an exercise where we present the student with a simple task: given a factorial program which uses Peano arithmetic, to rewrite it using Prolog's `is/2`. We will show piecemeal how to put together the source for this example. We will first show the part of the output that we want LPdoc to produce and then the source that produces that particular output. The full source and output can be found in Figure 3 in the appendix, and in the Ciao playground manual [4]. [11]

We start the exercise with a title and recalling the code for factorial using Peano arithmetic:

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

Exercise: factorial using ISO-Prolog arithmetic

Consider again the factorial example, using Peano arithmetic:

```
1  factorial(0,s(0)).
2  factorial(s(N),F) :-
3      factorial(N,F1),
4      times(s(N),F1,F).
```

This first part of the output is generated by the following code:

```
1  \title Exercise: factorial using ISO-Prolog arithmetic
2
3  Consider again the factorial example, using Peano arithmetic:
4  ```ciao_runnable
5  :- module(_, _, [assertions,library(bf/bfall)]).
6  %
7  factorial(0,s(0)).
8  factorial(s(N),F) :-
9      factorial(N,F1),
10     times(s(N),F1,F).
11 %
12
13 nat_num(0).
14 nat_num(s(X)) :- nat_num(X).
15
16 times(0,Y,0) :- nat_num(Y).
17 times(s(X),Y,Z) :- plus(W,Y,Z), times(X,Y,W).
18
19 plus(0,Y,Y) :- nat_num(Y).
20 plus(s(X),Y,s(Z)) :- plus(X,Y,Z).
21 ```
```

We first note that, in addition to text in markdown format, code between `‘‘‘ciao_runnable` and `‘‘‘` produces a panel in the output containing the code, which is editable and runnable. The code can be in modules and/or in 'user' files. We also note that it is possible to specify that only some parts of the code should appear in the output, by placing those parts between ***begin focus*** and ***end focus*** directives. This makes it possible to hide boilerplate lines (such as, e.g., module declarations, imports, auxiliary code, etc.) when they are not useful for the discussion. In this case we have hidden the auxiliary predicates that we assume have already been seen by the student in another lesson.

The arrow in the code pane allows loading the code in the playground, but we can also run the code in place within the document. One way to do this is to add one or more queries:

Some facts to note about this version:

- It is fully reversible!

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

This can be easily achieved with the following markdown with embedded Prolog code:

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

In the resulting panel, the query may be edited and pressing on the triangle executes it in place:

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

X = s(s(s(0))) ?
```

Next Stop

Regarding scoping, there is essentially one Ciao Prolog top level per page: all programs in the page are loaded into this Ciao Prolog top level and all queries in the same page are executed in that top level, against all the code (possibly separate modules) that has been loaded into the top level up to that point. Code can be (re)loaded anytime by clicking on the green tick mark in the top left of the pane; this facility could be used, for example, to reset the state of the program.

After perhaps mentioning that the Peano approach is elegant but inefficient, we could propose an actual exercise, which is to rewrite the code using Prolog's `is/2` (or constraints!):

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").
```

★ Show solution

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

Here the pane is again editable and contains the original (Peano) code adorned with comments, all of which act as hints or instructions on how to proceed. Of course, this description could also be somewhere else, e.g., in the surrounding text. Clicking on the yellow face will perform the evaluation, in this case running some (hidden) *unit tests* [11], on the code in order to give feedback to the student. Other evaluation methods (e.g., running a program analysis or a mere syntactic comparison) can also be useful. It is also possible for the student to give up and ask for the solution, in which case the proposed solution will be shown and can be executed.
All this functionality can be generated using the following code:

```
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.

%
%

factorial(0,s(0)).    %
factorial(M,F) :-     %
    M = s(N),         %
    factorial(N,F1),  %
    times(M,F1,F).    %

%
%
%

%
factorial(0,1).
factorial(N,F) :-
    N > 0,
    N1 is N-1,
    factorial(N1,F1),
    F is F1*N.
%
‘‘‘

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

The included unit tests are the ones that will be run to test the student's code (a small subset has been included for brevity). The segment within hint directives behaves similarly to the focus segments but represents a hint or instructions, and will be replaced by the solution, should it be asked for. The solution, if provided, is marked with the corresponding directives.

The appendix provides a complete example of a class exercise based on the code fragments above, showing the full source and the full output. The resulting, working Active Logic Document can be found, as mentioned before, as an example in the Ciao playground manual [4]. [12]

5 The Technical Approach

From a technical point of view the Ciao playground requires devising a means for running Prolog code directly in the browser.

Our first attempt at this was the Ciao Prolog JavaScript compiler backend [12], that enabled the use of Prolog and, in general, (constraint) logic programming to develop not just the server side, but also the client side of web applications, running fully on the browser. Tau Prolog [14] and the tuProlog playground [13] are recent Prolog interpreters written in JavaScript which also make

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

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

it easy to run Prolog in a web page, serverless. While these JavaScript-based approaches are attractive, they also have drawbacks. *Compilation* to JavaScript was a good option at the time, since it was a client (i.e., browser)-based solution and the resulting speed made it useful for many applications. However, performance does suffer with respect to native implementations (see [12]). This is even more pronounced in the case of the Prolog *interpreters* written in JavaScript mentioned above. It is precisely this performance impact that has led to the development of the WebAssembly virtual machine [6] [14], which is currently supported by all major browsers.

WebAssembly and the supporting compilation toolchains, such as Emscripten [16], enable programs written in languages supported by LLVM to be compiled to a form which can be executed entirely in the browser, i.e., without any server-side intervention at runtime, all with very reasonable efficiency. This is the approach used by the Ciao playground in order to be able to run Prolog code in the browser. The playground uses the standard Ciao engine, compiled to WebAssembly using the Emscripten C compiler and the Ciao library for C, which offers functions for term creation, type conversions, term checks, queries, and calls. The result is that in the playground Prolog code runs with performance that is competitive with native Prolog implementations. Additionally, the Ciao environment is comprised of several independent bundles (collections of modules) which can be compiled independently and demand-loaded from WebAssembly. The WebAssembly port of Ciao Prolog thus supports most of the system's software tools, such as LPdoc, CiaoPP (including the testing framework), etc., all of which are written in Prolog.

6 Conclusions and Outlook

We have described the Active Logic Documents (ALD) approach and toolset, that we have developed and been applying for embedding interactive Prolog components within teaching materials. ALD offers on one hand, support for easily adding click-to-run capabilities to any kind of teaching materials, independently of the tool used to generate them, and on the other hand a tool for generating web-based materials with embedded examples and exercises, based on the LPdoc documenter and the embedded version of the playground. We have also justified the fundamental principles of our approach which are that active parts run locally on the student's browser, with no need for a central infrastructure, and that the whole active document (tutorial, manual, exercise, etc.) is generated from a single, easy to use source that can be written and modified with any editor. We argue that this approach has multiple advantages from the point of view of scalability, maintenance cost, security, ease of packaging and distribution, etc.

Our tools evolved as a side-effect of the development of our own materials over the years for teaching logic programming[15], embedding runnable code and

[14] `https://webassembly.org/`

[15] See also [9] in this same book.

Task

You have found the following description of the picture, which unfortunatelly is wrong. Can you fix it?
Read carefully what each sentence (facts) means and see it if describes what you see in the picture.

```
1  above(triangle,square).
2  above(circle,hexagon).
3  right_of(hexagon,square).
4  right_of(circle,triangle).
```

Fig. 2. Adding gameplay functionality in a course for children. This task is accompanied with introductory text (not shown here) that carefully explains that `above(X,Y)` must be read as *`X` is above `Y`*, etc. Rather than introducing *infix* operators at this very early stage, the course begins with trivial formalization tasks to get familiar with syntax and abstraction.

exercises in tutorials [16], slides [17], manuals, etc., and they are currently being used 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 latter effort has implied the inclusion of additional useful features in the toolset, such as a "gameplay" which progressively discloses more advanced parts of the course while striving to keep the interaction interesting and challenging (see Fig. 2). The JavaScript interface provided by the tools and the access to Web technology enable endless possibilities for richer Web-based interaction (e.g., SVG visualization of facts), media rich interactions, touch/click inputs, audio, graphics or videos, etc.

[16] E.g., Interactive CiaoPP tutorials `https://ciao-lang.org/ciao/build/doc/ciaopp_tutorials.html/`

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

Appendix

TOC

Exercise: factorial using ISO-Prolog arithmetic

Consider again the factorial example, using Peano arithmetic:

```
1  factorial(0,s(0)).
2  factorial(s(N),F) :-
3      factorial(N,F1),
4      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` :

```
1  % TASK 1 - Rewrite with Prolog arithmetic
2
3  factorial(0,s(0)).    % TODO: Replace s(0) by 1
4  factorial(M,F) :-     % TODO: Make sure that M > 0
5      M = s(N),         % TODO: Compute N from M using is/2 (note that N is
6      factorial(N,F1),  %       unbound, so you need to compute N from M!)
7      times(M,F1,F).    % TODO: Replace times/3 by a call to is/2 (using *)
8
9  % When you are done, press the triangle ("Run tests") or the arrow
10 % ("Load into playground").
```

★ Show 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!

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

```
1  \title Exercise: factorial using ISO-Prolog arithmetic
2
3  Consider again the factorial example, using Peano arithmetic:
4
5  ```ciao\_runnable
6  :- module(_, _, [assertions,library(bf/bfall)]).
7  %
8  factorial(0,s(0)).
9  factorial(s(N),F) :-
10     factorial(N,F1),
11     times(s(N),F1,F).
12 %
13
14 nat_num(0).
15 nat_num(s(X)) :- nat_num(X).
16
17 times(0,Y,0) :- nat_num(Y).
18 times(s(X),Y,Z) :- plus(W,Y,Z), times(X,Y,W).
19
20 plus(0,Y,Y) :- nat_num(Y).
21 plus(s(X),Y,s(Z)) :- plus(X,Y,Z).
22 ```
23
24 Some facts to note about this version:
25   - It is fully reversible!
26 ```ciao_runnable
27 ?- factorial(X,s(s(s(s(s(s(0))))))).
28 ```
29   - But also inefficient...
30 ```ciao_runnable
31 ?- factorial(s(s(s(s(0)))),Y).
32 ```
33
34 We can also code it using ISO-Prolog arithmetic, i.e., `is/2`:
35 ```ciao
36  ... Z is X * Y ...
37 ```
38 Note that this type of arithmetic has limitations: it only works in
39 one direction, i.e., `X` and `Y` must be bound to arithmetic terms.
40
41 But it provides a (large!) performance gain.  Also, meta-logical
42 tests (see later) allow using it in more modes.
43
44 Try to encode the factorial program using `is/2`:
45 ```ciao_runnable
46 :- module(_, _, [assertions]).
47 :- test factorial(5, B) => (B = 120) + (not_fails, is_det).
48 :- test factorial(0, 0) + fails.
49 :- test factorial(-1, B) + fails.
50 %
51 %
52
53 factorial(0,s(0)).    %
54 factorial(M,F) :-     %
55     M = s(N),         %
56     factorial(N,F1),  %
57     times(M,F1,F).    %
58
59 %
60 %
61 %
62 %
63 factorial(0,1).
64 factorial(N,F) :-
65     N > 0,
66     N1 is N-1,
67     factorial(N1,F1),
68     F is F1*N.
69 %
70 ```
71
72 Note that wrong goal order can raise an error (e.g., moving the last
73 call to `is/2` before the call to factorial).
74
75 **Next:** Let's try using constraints instead!
```

Fig. 3. The full source and LPdoc output for the Active Logic Document for the simple factorial exercise.

References

1. Arias, J., Carro, M., Salazar, E., Marple, K., Gupta, G.: Constraint Answer Set Programming without Grounding. Theory and Practice of Logic Programming **18**(3-4), 337–354 (2018). https://doi.org/10.1017/S1471068418000285 2
2. Brecklinghaus, A., Koerner, P.: A Jupyter kernel for Prolog. In: Proc. 36th Workshop on (Constraint) Logic Lrogramming (WLP 2022). Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn (September 2022) 1
3. Flach, P., Sokol, K., Wielemaker, J.: Simply Logical - The First Three Decades. In: Warren, D.S., Dahl, V., Eiter, T., Hermenegildo, M., Kowalski, R., Rossi, F. (eds.) Prolog - The Next 50 Years. No. 13900 in LNCS, Springer (July 2023) 1
4. Garcia-Pradales, G., Morales, J., Hermenegildo, M.V.: The Ciao Playground. Tech. rep., Technical University of Madrid (UPM) and IMDEA Software Institute (2021), http://ciao-lang.org/ciao/build/doc/ciao_playground.html/ciao_playground_manual.html 2, 4, 4
5. Garcia-Pradales, G., Morales, J., Hermenegildo, M.V., Arias, J., Carro, M.: An s(CASP) In-Browser Playground based on Ciao Prolog. In: ICLP'22 Workshop on Goal-directed Execution of Answer Set Programs (August 2022) 2, 2
6. Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A., Bastien, J.F.: Bringing the web up to speed with webassembly. In: Cohen, A., Vechev, M.T. (eds.) Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017. pp. 185–200. ACM (2017). https://doi.org/10.1145/3062341.3062363, https://doi.org/10.1145/3062341.3062363 5
7. Hermenegildo, M.V.: A Documentation Generator for (C)LP Systems. In: International Conference on Computational Logic, CL2000. pp. 1345–1361. No. 1861 in LNAI, Springer-Verlag (July 2000) 3
8. Hermenegildo, M.V., Morales, J.: The LPdoc Documentation Generator. Ref. Manual (v3.0). Tech. rep., UPM (July 2011), available at http://ciao-lang.org 3
9. Hermenegildo, M., Morales, J.: Some Thoughts on How to Teach Prolog. In: Warren, D.S., Dahl, V., Eiter, T., Hermenegildo, M., Kowalski, R., Rossi, F. (eds.) Prolog - The Next 50 Years. No. 13900 in LNCS, Springer (July 2023) 15
10. Knuth, D.: Literate programming. Computer Journal **27**, 97–111 (1984) 1
11. Mera, E., Lopez-Garcia, P., Hermenegildo, M.V.: Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework. In: 25th Int'l. Conference on Logic Programming (ICLP'09). LNCS, vol. 5649, pp. 281–295. Springer-Verlag (July 2009) 4
12. Morales, J.F., Haemmerlé, R., Carro, M., Hermenegildo, M.V.: 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), 755–773 (2012) 5
13. Morrison, B.B., DiSalvo, B.J.: Khan academy gamifies computer science. In: Dougherty, J.D., Nagel, K., Decker, A., Eiselt, K. (eds.) The 45th ACM Technical Symposium on Computer Science Education, SIGCSE 2014, Atlanta, GA, USA, March 5-8, 2014. pp. 39–44. ACM (2014). https://doi.org/10.1145/2538862.2538946, https://doi.org/10.1145/2538862.2538946 1
14. τProlog — an open source Prolog interpreter in javascript. http://tau-prolog.org (2021), last access: April 28, 2023 5
15. Wielemaker, J., Riguzzi, F., Kowalski, R.A., Lager, T., Sadri, F., Calejo, M.: Using SWISH to realize interactive web-based tutorials for logic-based languages. Theory

Pract. Log. Program. **19**(2), 229–261 (2019). https://doi.org/10.1017/S1471068418000522, https://doi.org/10.1017/S1471068418000522 1

16. Zakai, A.: Emscripten: an llvm-to-javascript compiler. In: Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications. pp. 301–312. SPLASH '11, ACM, New York, NY, USA (2011). https://doi.org/10.1145/2048147.2048224, http://doi.acm.org/10.1145/2048147.2048224 5