Cloud Logic Programming for Integrating Language Technology Resources

Markus Forsberg and Torbjörn Lager

Språkbanken, University of Gothenburg, Sweden
Department of Philosophy, Linguistics and Theory of Science, University of Gothenburg, Sweden
markus.forsberg@svenska.gu.se, torbjorn.lager@ling.gu.se

Abstract

The main goal of the CLT Cloud project is to equip lexica, morphological processors, parsers and other software components developed within CLT (Centre of Language Technology) with so called web API:s, thus making them available on the Internet in the form of web services. We present a proof-of-concept implementation of the CLT Cloud server where we use the logic programming language Prolog for composing and aggregating existing web services into new web services in a way that encourages creative exploration and rapid prototyping of LT applications.

Keywords: web service, processing pipeline, declarative language

1. Introduction

The Centre for Language Technology[1] (CLT) in Gothenburg is an organization for collaboration between LT researchers at the Faculty of Arts and the IT Faculty at the University of Gothenburg and Chalmers University of Technology. The main goal of the centre is to strengthen the collaboration of LT research in Gothenburg, and as one way of reaching this goal, we are working on the integration of our tools and resources to synchronize the efforts within the centre.

However, our experience with tool integration has been one with technical hurdles – the tools have been written in a multitude of programming languages targeting different system platforms; to overcome these technical difficulties we are now exploring the use of web services as a way of integrating the CLT tools and resources, an effort we call *CLT Cloud*.

The benefits of web services in our setting is that they provide platform and programming language independence, and at the same time, an environment for distributed and parallel computation. In addition, a distributed environment supports the possibility to give developers access to the latest versions of the resources and tools.

The present paper is not concerned with the setting up of LT web services per se. Rather, it deals with the task of composing and aggregating *existing* web services into *new* web services, and with the problem of doing this in a declarative and flexible manner that encourages creative exploration and rapid prototyping of LT applications. Requirements such as declarativity, compositionality, security and a principled approach to the handling of ambiguity made us choose a declarative subset of Prolog for this purpose. In the rest of the article we motivate our choice further, and present a proof-of-concept implementation of CLT Cloud.

2. The CLT Cloud server

In Figure 1 we sketch the big picture of our CLT Cloud server running in the context of other servers and clients.
The machine in the middle runs our dedicated CLT Cloud server which provides a common access point to LT web services (lexica, taggers, parsers, etc.) being offered by other servers running inhouse or elsewhere. A server acting as a proxy has many technical advantages: the identities of machines behind the proxy are not known to clients, which not only provides security but also indirection (i.e., web services can be moved around). It furthermore enables us to perform load-balancing and caching, log and analyze traffic, and apply access policies such as restrictions on content size or execution time.

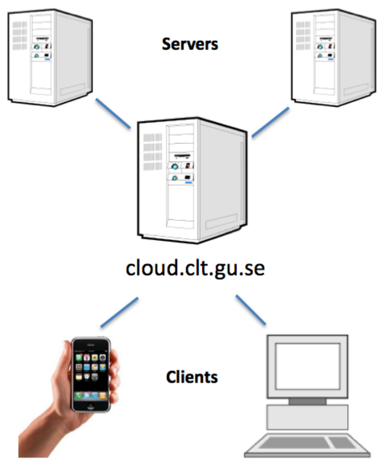


Figure 1: The big picture.

Our proof-of-concept CLT Cloud server is built upon ClioPatria[2], a well designed and mature platform implemented in SWI-Prolog (Wielemaker et al., 2011). ClioPatria provides us with a ready-to-run web server, an administration interface, support for authentication and authorization, and more.

3. An LT pipeline and logic programming

We want CLT Cloud to be more than just a collection of interoperable web services – we want the ability to combine them in a flexible and declarative manner, and at the same time, reduce the amount of network traffic required between CLT Cloud and clients.

[1]<http://clt.gu.se>

[2]<http://cliopatria.swi-prolog.org>

The solution we are currently employing is the use of Prolog as the processing pipeline language. Prolog has a number of traits that distinguishes it from most other programming languages:

- Declarativeness, inherited from its roots in logic
- Inference capabilities, inherited from its roots in theorem proving
- No separation between data and programs (reflexivity)
- The lazy a-tuple-at-a-time generation of solutions to queries

A Prolog query performing tokenization followed by POS tagging of a given text could be written as follows:

```
?- text(Text), tokens(Text,Tokens),
   tags(Tokens,Tags).
Text = 'Tom runs'
Tokens = ['Tom', runs]
Tags = [[token-'Tom', pos-'NNP'],
        [token-runs, pos-'VBZ']]
```

We can compare this to a Unix style pipeline chaining the tokenizer and tagger processes by their standard streams by means of an anonymous pipe, like so:

```
$ tokens < text.txt | tags > tags.txt
```

However, this comparision quickly breaks down:

- In comparison with the Unix pipline, the intermediate results in a Prolog pipeline are *terms* rather than streams and they are *explicit* and *named* rather than anonymous.
- Prolog is *nondeterministic* in that the variables in a query may be bound to different values on backtracking. From a formal point of view, a Unix pipeline can be seen as a kind of functional composition. Prolog is a relational nondeterministic language, and supports *relational* composition.
- Furthermore, (some of) the predicates in a Prolog pipeline may be *bi-directional*, so that *tokens(Text, Tokens)* can also be used to *detokenize* a list of tokens into a Prolog atom.

Via the `bagof/3`, `setof/3` and `findall/3` built-in predicates, Prolog also supports not only relational composition but also *aggregation* of all solutions to a query. For example, if a lexicon is stored in the form of `wordform/2` clauses, we may compute the number of words stored by running the following query.

```
?- findall(Word,wordform(Word,S),Words),
   length(Words,Length).
```

We believe that Prolog offers a suitable foundation for building a very flexible NLP computation pipeline. We intend to show that explicit and named variables holding intermediate results should be considered an advantage, that nondeterministic computing makes a lot of sense also in this context, and that the other traits that Prolog has can be very useful.

We are of course aware of the fact that Prolog has fallen out of fashion in NLP circles since its heyday in the 1980's. One likely reason for this (but not the only reason) is the dominance of statistical approaches to LT, and the fact that Prolog is fairly weak at number crunching. However, as the pendulum swings back towards more knowledge intensive approaches, the appreciation of logic programming languages may again increase. Fortunately, at least judging from our own colleagues, most members of the LT world still remember the Prolog they once learned, at least sufficiently well for our purpose. Besides, we must emphasize that we do not propose the use of Prolog for each and every LT programming task – quite the opposite!

Predicates such as `tokens/2` and `tags/2` *can* of course be implemented in Prolog, but tokenizers and part of speech taggers written in other programming languages are more readily available. Now, if such resources are also available, or can be made available, in the form of web services, it is often a good idea and very straightforward to access these from Prolog instead. The definition of `tags/2`, for example, may access a POS tagging web service over HTTP, written in any programming language. Thus, Prolog can be used as a *glue language* – a way to combine two or more HTTP web services into a new web service. This is indeed our main intended use of Prolog in this project, and we believe that it is a role in which Prolog shines.

In order to make a particular web service available as a predicate in the CLT Cloud, one often has to do a bit of Prolog programming. An effort must be made to ensure that predicates can be used in either direction (bi-directionality), errors must be handled gracefully, and one must decide whether to return an empty representation or simply fail (and thus force backtracing) when (say) a parsing process fails to come up with an analysis.

4. The CLT Cloud infrastructure

The web services in CLT cloud follows the REST (Representational State Transfer) architectural principles (Fielding, 2000), which entails making good use of all the major HTTP methods (GET, PUT, POST and DELETE), rather than building a protocol on top of HTTP.

On a first blush, Prolog does not seem to mesh well with the way the Web and its protocols work, in particular when it comes to GET. We have what is often referred to as an *impedance mismatch problem*: Prolog is relational in that a query may map to more than one result, but HTTP GET is essentially functional in that one query/request should map to exactly one result/response. Sometimes this can be solved by using an all-solutions predicate such as `findall/3`, but this only works for a finite number of solutions and only if they are not too many. Besides, we may prefer to generate the solutions one-by-one, sometimes because it is cheaper in terms of memory requirements (on both server and client), and sometimes because we want to decide, after having seen the first couple of solutions, whether we want to see more.

We choose instead to work with a *virtual index* to the solutions that a query has, without actually generating the solutions. Each solution in the sequence of n solutions to a query receives an integer index in the range 0::n-1. This makes a query for the ith solution of a goal functional and deterministic, and thus solves the impedance mismatch problem. To implement this efficiently we have to avoid regenerating solutions 0-i when asking for solution i+1. Fortunately, a technique for this which preserves the Prolog state (stack, choice points, variable bindings) between requests by creating a thread that is associated to the HTTP session, running the state-full computation there, and sending messages back and forth between the HTTP handlers and the session thread to communicate queries and results, has already been developed (Wielemaker et al., 2011).

Apart from sending queries to the server using GET, a client may PUT or POST Prolog clauses to the service to be stored in a scratch module associated with the current session and used in subsequent queries. This is typically the way to upload data (e.g. text) to the server for processing, but since the uploaded content may also be Prolog *rules*, it is also something that increases the power of the querying facilities. Clauses can also be DELETEd at any time.

We illustrate this by a small example. Suppose we want to part of speech tag the sentence "Tom runs". This is done by an update followed by a query, i.e., by using two HTTP requests. First we perform a PUT request like so:

```
PUT cloud.clt.gu.se
text('Tom runs').
```

This places the clause `text('Tom runs')` in the client's scratch module on the server and returns a simple JSON acknowledgment. Then we follow up with an GET request of the following form:

```
GET cloud.clt.gu.se?query=Q&cursor=C&limit=L
```

Here, `Q` is a Prolog query, `C` is and integer acting as a key to the zero-based virtual index of solutions to `Q`, and `L` is an integer specifying the maximum number of solutions to be returned. If `Q` is replaced by our simple example Prolog query, and if `C` is set to 0 and `L` to 1 (the defaults), then the JSON encoded response would be:

```
{"success": true,
 "message": "yes",
 "bindings": [
    {"Text": "Tom runs",
     "Tokens": ["Tom", "runs"]
     "Tags": [{"token":"Tom", "pos":"NNP"},
              {"token":"runs", "pos":"VBZ"}]
    }
   ]
}
```

There is another way to perform exactly the same task. Along with the `text/1` clause(s), we may send program clause(s) to the server, in this case for defining a new predicate `my_tags/1`:

```
PUT cloud.clt.gu.se
text('Tom runs').
my_tags(Tags) :-
      text(Text),
      tokens(Text, Tokens),
      tags(Tokens, Tags).
```

and then use the new predicate from the client as follows:

```
GET cloud.clt.gu.se?query=my_tags(Tags)
```

We can think of this as a way to do rule-based ad-hoc inferencing in the context of querying, or as a way to on the fly create new web services on the basis of existing ones.

In connection with the above very small and artificial example we would like to make the following points:

- Our approach scales very well. We have tried it with a `text/1` clause with an atom representing a text containing hundreds of thousands of words. For such tasks, and thanks to the backtracking behavior inherent in our approach, we are able to formulate a query which splits the text into sentences, picks one, tokenizes and part of speech tags it, and returns the tagged sentence to the client. On backtracking (forced by the client sending a new request with an incremented cursor), it picks the next sentence, tags and returns it to the client, and so on.

- Our simple example showed how to retrieve solutions one by one from the server. By setting the request parameter `limit` to an integer, for example 5, we may retrieve up to five solutions at a time. If five solutions exist, the value of the `bindings` property in the returned JSON will be a list with five members. Again, if we ask for the *next* five solutions, the first five solutions will not have to be recomputed.

- Except when debugging, a client usually has no need for intermediate results, and thus no need to see the values of all variables in a query. The convention here is to prefix the "uninteresting" variable names with an underscore character, which will instruct the server to not include the corresponding bindings in the JSON responses. This is also a convention implemented in some (but not all) Prolog commandline interpreters.

- For lack of space and for pedagogical reasons, our simple example did not show how we in practice use Prolog's module system to allow the same predicate (same name and arity) to be independently defined in multiple namespaces. In order to access the NLTK tokenizer, we must in fact write `nltk:tokens(Ws,Ts)` rather than just `tokens(Ws,Ts)`. Another (and much faster) tokenizer, available as a module in SWI-Prolog, can currently be accessed as `swipl:tokens(Ws,Ts)`.

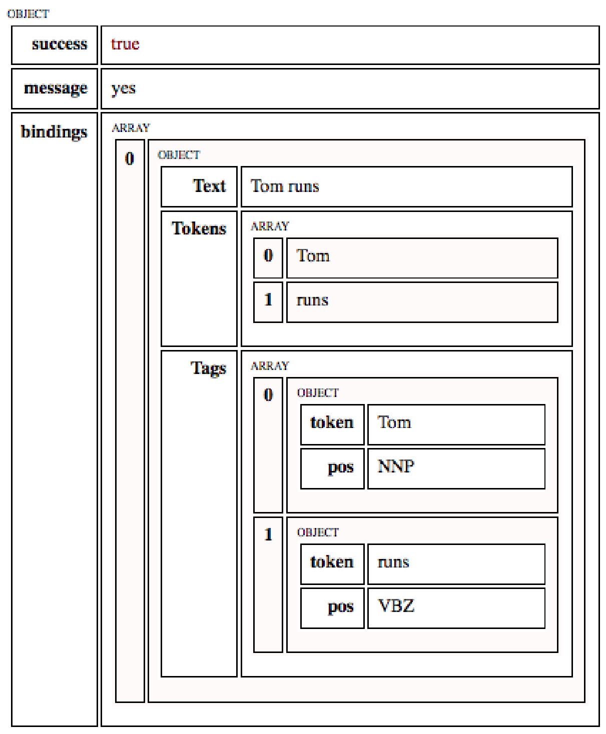
- Many CLT Cloud predicates takes an optional list of options that configure the processing in various ways. For example, `swipl:tokens(Ws,Ts, [downcase(true)])` converts all the characters of the input to lower case before tokenization is performed. Default is `downcase(false)`.

- Allowing clients to execute arbitrary Prolog programs and queries on the server does of course raise important security concerns. We handle this by carefully inspecting the programs and queries before allowing them to be executed, something which is relatively easy in a reflexive language such as Prolog.

5. The CLT Cloud API Explorer

The CLT Cloud API Explorer is a browser-based client written in HTML, CSS and JavaScript that makes it easy to explore the API:s and the predicates offered by the CLT Cloud.

JSON is easy to process by client programs, but is not very friendly on the inspecting human eye. When debugging an application, or when just exploring interesting queries, we choose to present JSON responses as tables rendered in HTML, as in Figure 2. We find the mapping between variable names in the query and the attribute names (`Text`, `Tokens`, and `Tags`) in the bindings section of the table to be clear and easy to follow.



OBJECT

success	true
message	yes
bindings	ARRAY

- **0** OBJECT
  - **Text**: Tom runs
  - **Tokens**: ARRAY
    - **0**: Tom
    - **1**: runs
  - **Tags**: ARRAY
    - **0** OBJECT
      - **token**: Tom
      - **pos**: NNP
    - **1** OBJECT
      - **token**: runs
      - **pos**: VBZ

Figure 2: JSON viewed as a table.

Apart from a JSON presentation area on the right, the API Explorer, depicted in it full in Figure 3, has an Update area containing a text area for entering clauses to be uploaded to the server by clicking the PUT or POST button, and a text field for entering templates for predicates to be deleted when clicking the DELETE button. The Query area contains a text field to hold the query, a GET button and fields and menus for specifying cursor and limit. For convenience, buttons marked First, Next and Previous for paging solutions are also provided.

6. Related work

In this paper we focused on the use of logic programming to create an abstraction on top of existing web services, and the only related work we are aware of is the work we are currently building upon. However, we are in a good company in the LT community with our work on web services and processing pipelines, e.g., WebLicht (Web Based Linguistic Chaining Tool) (Hinrichs et al., 2010), and IULA Web Services (Martínez et al., 2010).

7. Future work

We plan to incorporate all CLT software and resources in the CLT Cloud as web services in a unified web service directory with a common access point. This should not be a major task since it has already been done for the bulk of them. An important requirement is that these web services are actually the ones used by the local research group so that future developments become available.

We have already written a fair number of browser based demo client applications served by the CLT Cloud, but here we just include pictures of two of them – a DCG parsing tool (Figure 4), and a Wordnet browser (Figure 5). In the future, we plan to make the CLT Cloud available to our LT masters students and believe that this will result in many more applications using it.

Since the CLT Cloud uses JSON rather than XML for interprocess communication we are developing Prolog tools for pointing to an exact location in a JSON object, and for transforming one JSON object into another. Such utilities correspond to XPath and XSLT in the XML world, but with the very important difference that the Prolog tools generate solutions a-tuple-at-a-time. Having access to a nondeterministic variant of XSLT will allow us to easily build adapters between for example partially incompatible tag sets.

Another obvious future task is to explore ways of performing query optimization. Optimization could be as simple as moving some software to the same server as the common access point, where the up-to-date requirement could be ensured by regular update from the software's versioning system, or more complex tasks such as optimization through a query rewrite system, something which is comparatively easy in a reflexive programming language that can treat programs as data.

8. Acknowledgements

The research presented here was supported by the University of Gothenburg through its support of the Centre for Language Technology (CLT).

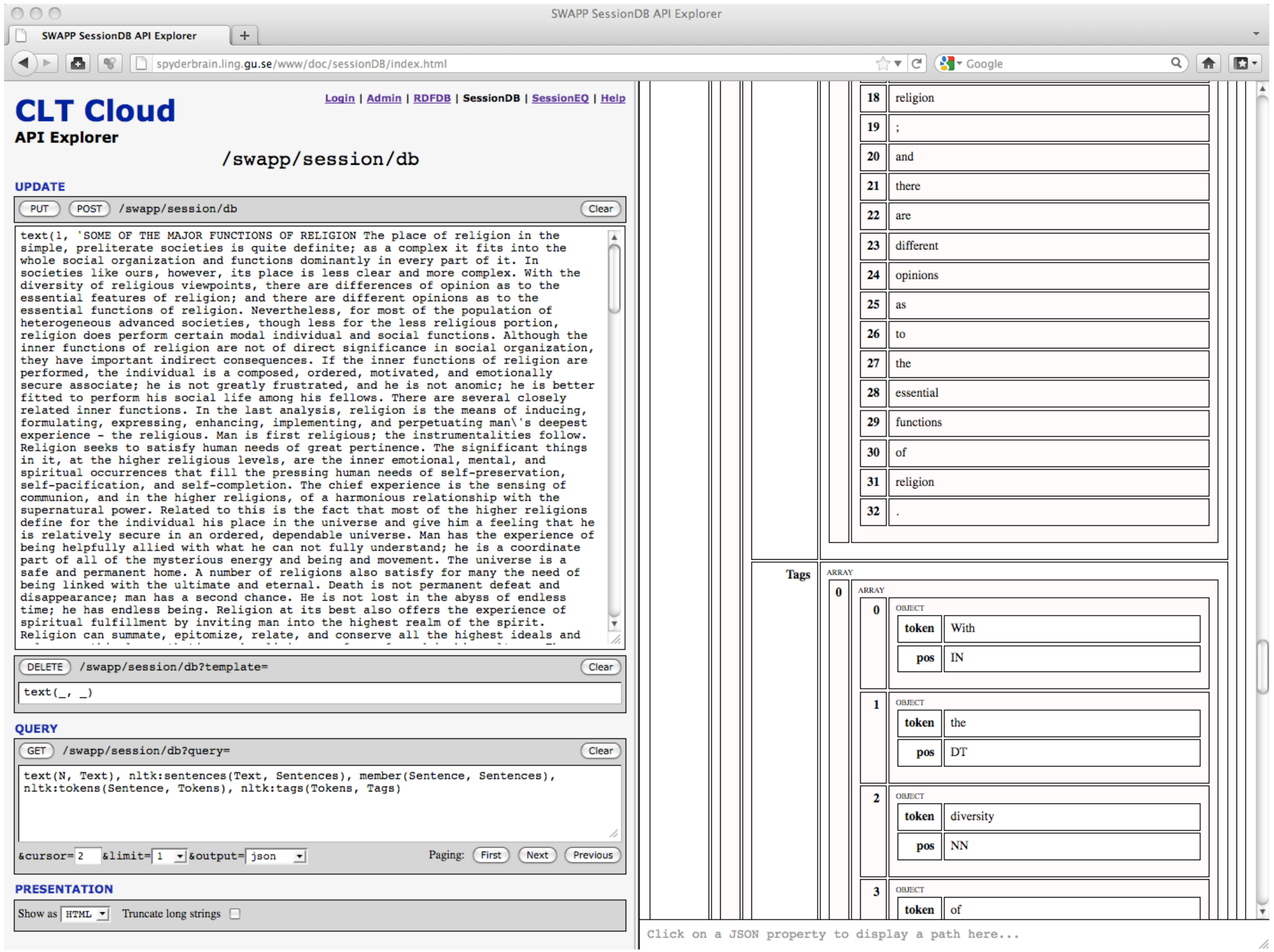
9. References

R.T. Fielding. 2000. *Architectural styles and the design of network-based software architectures*. Phd thesis, University of California, Irvine.

Erhard Hinrichs, Marie Hinrichs, and Thomas Zastrow. 2010. Weblicht: web-based lrt services for german. In *Proceedings of the ACL 2010 System Demonstrations*, ACLDemos '10, pages 25–29, Stroudsburg, PA, USA. Association for Computational Linguistics.

Héctor Martínez, Jorge Vivaldi, and Marta Villegas. 2010. Text handling as a web service for the iula processing pipeline. In *Web Services and Processing Pipelines in HLT: Tool Evaluation, LR Production and Validation*, pages 22–29, Paris. ELRA.

Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. 2011. SWI-Prolog. *Theory and Practice of Logic Programming: Special Issue on Prolog Systems*, 12:67–96.



SWAPP SessionDB API Explorer

spyderbrain.ling.gu.se/www/doc/sessionDB/index.html

CLT Cloud
API Explorer

Login | Admin | RDFDB | SessionDB | SessionEQ | Help

/swapp/session/db

UPDATE

PUT POST /swapp/session/db Clear

```
text(1, 'SOME OF THE MAJOR FUNCTIONS OF RELIGION The place of religion in the
simple, preliterate societies is quite definite; as a complex it fits into the
whole social organization and functions dominantly in every part of it. In
societies like ours, however, its place is less clear and more complex. With the
diversity of religious viewpoints, there are differences of opinion as to the
essential features of religion; and there are different opinions as to the
essential functions of religion. Nevertheless, for most of the population of
heterogeneous advanced societies, though less for the less religious portion,
religion does perform certain modal individual and social functions. Although the
inner functions of religion are not of direct significance in social organization,
they have important indirect consequences. If the inner functions of religion are
performed, the individual is a composed, ordered, motivated, and emotionally
secure associate; he is not greatly frustrated, and he is not anomic; he is better
fitted to perform his social life among his fellows. There are several closely
related inner functions. In the last analysis, religion is the means of inducing,
formulating, expressing, enhancing, implementing, and perpetuating man\'s deepest
experience - the religious. Man is first religious; the instrumentalities follow.
Religion seeks to satisfy human needs of great pertinence. The significant things
in it, at the higher religious levels, are the inner emotional, mental, and
spiritual occurrences that fill the pressing human needs of self-preservation,
self-pacification, and self-completion. The chief experience is the sensing of
communion, and in the higher religions, of a harmonious relationship with the
supernatural power. Related to this is the fact that most of the higher religions
define for the individual his place in the universe and give him a feeling that he
is relatively secure in an ordered, dependable universe. Man has the experience of
being helpfully allied with what he can not fully understand; he is a coordinate
part of all of the mysterious energy and being and movement. The universe is a
safe and permanent home. A number of religions also satisfy for many the need of
being linked with the ultimate and eternal. Death is not permanent defeat and
disappearance; man has a second chance. He is not lost in the abyss of endless
time; he has endless being. Religion at its best also offers the experience of
spiritual fulfillment by inviting man into the highest realm of the spirit.
Religion can summate, epitomize, relate, and conserve all the highest ideals and
```

DELETE /swapp/session/db?template= Clear

```
text(_, _)
```

QUERY

GET /swapp/session/db?query= Clear

```
text(N, Text), nltk:sentences(Text, Sentences), member(Sentence, Sentences),
nltk:tokens(Sentence, Tokens), nltk:tags(Tokens, Tags)
```

&cursor= 2 &limit= 1 &output= json Paging: First Next Previous

PRESENTATION

Show as HTML Truncate long strings

18	religion
19	;
20	and
21	there
22	are
23	different
24	opinions
25	as
26	to
27	the
28	essential
29	functions
30	of
31	religion
32	.

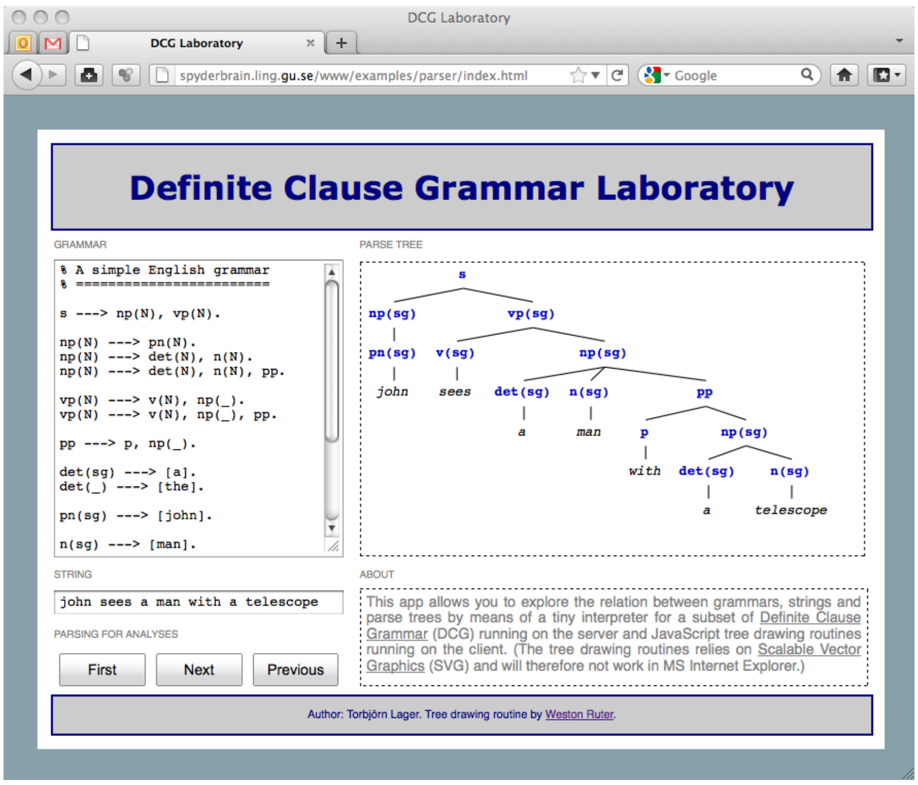
Tags ARRAY

0 ARRAY

0	OBJECT	token	With
		pos	IN
1	OBJECT	token	the
		pos	DT
2	OBJECT	token	diversity
		pos	NN
3	OBJECT	token	of

Click on a JSON property to display a path here...

Figure 3: The CLT Cloud API Explorer.


DCG Laboratory

spyderbrain.ling.gu.se/www/examples/parser/index.html

Definite Clause Grammar Laboratory

GRAMMAR

```
% A simple English grammar
% ========================

s ---> np(N), vp(N).

np(N) ---> pn(N).
np(N) ---> det(N), n(N).
np(N) ---> det(N), n(N), pp.

vp(N) ---> v(N), np(_).
vp(N) ---> v(N), np(_), pp.

pp ---> p, np(_).

det(sg) ---> [a].
det(_) ---> [the].

pn(sg) ---> [john].

n(sg) ---> [man].
```

PARSE TREE

```
s
├── np(sg)
│   └── pn(sg)
│       └── john
└── vp(sg)
    ├── v(sg)
    │   └── sees
    └── np(sg)
        ├── det(sg)
        │   └── a
        ├── n(sg)
        │   └── man
        └── pp
            ├── p
            │   └── with
            └── np(sg)
                ├── det(sg)
                │   └── a
                └── n(sg)
                    └── telescope
```

STRING

john sees a man with a telescope

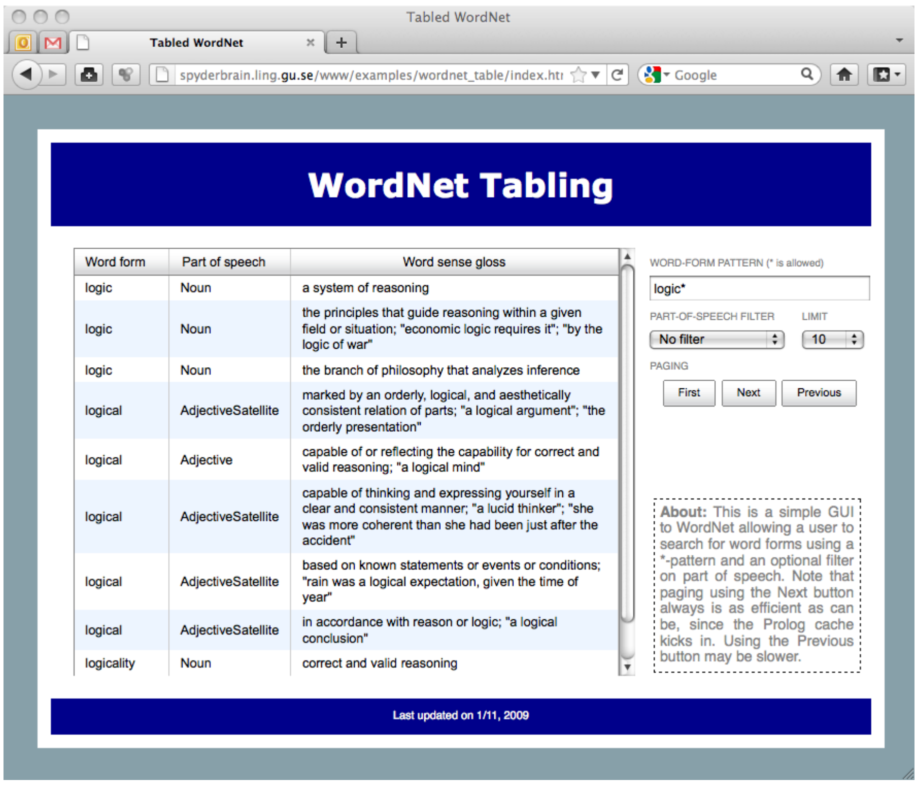
PARSING FOR ANALYSES

ABOUT

This app allows you to explore the relation between grammars, strings and parse trees by means of a tiny interpreter for a subset of Definite Clause Grammar (DCG) running on the server and JavaScript tree drawing routines running on the client. (The tree drawing routines relies on Scalable Vector Graphics (SVG) and will therefore not work in MS Internet Explorer.)

Author: Torbjörn Lager. Tree drawing routine by Weston Ruter.

Figure 4: DCG Lab – served by the CLT Cloud.


Tabled WordNet

spyderbrain.ling.gu.se/www/examples/wordnet_table/index.htı

WordNet Tabling

Word form	Part of speech	Word sense gloss
logic	Noun	a system of reasoning
logic	Noun	the principles that guide reasoning within a given field or situation; "economic logic requires it"; "by the logic of war"
logic	Noun	the branch of philosophy that analyzes inference
logical	AdjectiveSatellite	marked by an orderly, logical, and aesthetically consistent relation of parts; "a logical argument"; "the orderly presentation"
logical	Adjective	capable of or reflecting the capability for correct and valid reasoning; "a logical mind"
logical	AdjectiveSatellite	capable of thinking and expressing yourself in a clear and consistent manner; "a lucid thinker"; "she was more coherent than she had been just after the accident"
logical	AdjectiveSatellite	based on known statements or events or conditions; "rain was a logical expectation, given the time of year"
logical	AdjectiveSatellite	in accordance with reason or logic; "a logical conclusion"
logicality	Noun	correct and valid reasoning

WORD-FORM PATTERN (* is allowed): logic*

PART-OF-SPEECH FILTER: No filter

LIMIT: 10

PAGING

About: This is a simple GUI to WordNet allowing a user to search for word forms using a *-pattern and an optional filter on part of speech. Note that paging using the Next button always is as efficient as can be, since the Prolog cache kicks in. Using the Previous button may be slower.

Last updated on 1/11, 2009

Figure 5: Wordnet browser – served by the CLT Cloud.