□ ◇ □

TEX in Teaching

Michael Moortgat, Richard Moot & Dick Oehrle*

ABSTRACT. A well-known slogan in language technology is 'parsing-as-deduction': syntax and meaning analysis of a text takes the form of a mathematical proof. Developers of language technology (and students of computational linguistics) want to visualize these mathematical objects in a variety of formats.

We discuss a language engineering environment for computational grammars. The kernel is a theorem prover, implemented in the logic-programming language Prolog. The kernel produces LATEX source code for its internal computations. The front-end displays these in a number of user-defined typeset formats. Local interaction with the kernel is via a tcl/tk GUI. Alternatively, one can call the kernel remotely from dynamic PDF documents, using the form features of Sebastian Rahtz' `hyperref` package.

This paper discusses some uses of the dynamic possibilities offered by Sebastian Rahtz' `hyperref` package in the context of a courseware project we have been engaged in. The project provides a grammar development environment for Type-Logical Grammar — one of the formalisms that are currently used in computational linguistics. Our paper is organized as follows. First, we offer the reader a glimpse of what type-logical grammars look like. In the next section, we discuss the TEX-based visualisation tools of the Grail workbench in its original unix habitat. Finally, we report on our current efforts to provide browser-based access to the Grail kernel via dynamic PDF documents.

Type-logical grammar

Type-logical (*TLG*) grammar is a logic-based computational formalism that grew out of the work of the mathematician Jim Lambek in the late Fifties. For readers with easy access to issues of the *American Mathematical Monthly* in the pre-TEX era, the seminal paper (Lambek 1958) is warmly recommended; (Moortgat 1997) gives an up-to-date survey of the field. The mathematically-inclined reader of these Proceedings will easily appreciate why it is such a pleasure to work with TLG.

As the name suggests, *TLG* has strong type-theoretic connections. One could think of it as a functional programming language with some special features to handle the

*We thank Willemijn Vermaat and Bernhard Fisseni for helpful discussions of TEXnicalities. For comments and suggestions, contact {moortgat,oehrle}@let.uu.nl.

peculiarities of natural (as opposed to programming) languages. In a functional language (say, Haskell), expressions are typed. There is some inventory of basic types (integers, booleans, ...); from types T, T' one can form functional types $T \to T'$. With these functional types, one can do two things. An expression/program of type $T \to T'$ can be used to compute an expression of type T' by *applying* it to an argument of the appropriate type T. Or a program of type $T \to T'$ can be obtained by *abstracting* over a variable of type T in an expression of type T'. Below we give a simple example: the construction of a square function out of a built-in times function. We present this as a logical derivation — the beautiful insight of Curry allows us to freely switch perspective between types and logical formulas, and between type computations and logical derivations in a constructive logic (Positive Intuitionistic Logic).

$$\dfrac{\dfrac{\dfrac{\mathsf{times} : \mathsf{Int} \to (\mathsf{Int} \to \mathsf{Int}) \quad x : \mathsf{Int}}{(\mathsf{times}\ x) : \mathsf{Int} \to \mathsf{Int}}\ (\mathit{Elim} \to) \qquad x : \mathsf{Int}}{(\mathsf{times}\ x\ x) : \mathsf{Int}}\ (\mathit{Elim} \to)}{\lambda x.(\mathsf{times}\ x\ x) : \mathsf{Int} \to \mathsf{Int}}\ (\mathit{Intro} \to)$$

How can we transfer these ideas to the field of natural language grammars? The basic types in this setting are for expressions one can think of as 'complete' in some intuitive sense — one could have a type *np* for names ('Donald Knuth', 'the author of *The Art of Computer Programming*', ...), common nouns *n* ('author', 'art', ...), sentences *s* ('Knuth wrote some books', 'TEX is necessary', ...). Now, where a phrase-structure grammar would have to add a plethora of non-terminals to handle incomplete expressions, in *TLG* we use functional (implicational) types for these. A determiner like 'the' is typed as a function from *n* expressions (like 'author') to *np* expressions; a verb phrase (like 'is necessary') as a function from *np* expressions into *s* expressions, and so on.

To adjust the type-logical approach to the natural language domain, we have to introduce two refinements. The syntax of our programming language example obeys the martial law of Polish prefix notation: functions are put before their arguments. Natural languages are not so disciplined: a determiner (in English) comes before the noun it combines with; a verb phrase follows its subject. Instead of one implication, *TLG* has two to capture these word-order distinctions: an expression of type T/T' is *prefixed* to its T'-type argument; an expression $T'\backslash T$ is *suffixed* to it. An example is given below. (The product $\circ$ is the explicit structure-building operation that goes with use of the slashes. It imposes a tree structure on the derived sentence.)

$$\dfrac{\mathsf{mathematicians} \vdash np \qquad \dfrac{\mathsf{like} \vdash (np\backslash s)/np \quad \mathsf{T_EX} \vdash np}{\mathsf{like} \circ \mathsf{T_EX} \vdash np\backslash s}\ [/E]}{\mathsf{mathematicians} \circ (\mathsf{like} \circ \mathsf{T_EX}) \vdash s}\ [\backslash E]$$

The second refinement has to do with the management of 'programming resources'. In our Haskell-style example, one can use resources as many times as one wants (or not use them at all). You see an illustration in the last step of the derivation, where two occurrences of $x : \mathsf{Int}$ are withdrawn simultaneously. In natural language, such a cavalier attitude towards occurrences would not be a good idea: a well-formed sentence is not likely to remain well-formed if you remove some words, or repeat some. (You will agree that ‘mathematicians like’ does not convey the message that mathematicians like mathematicians.) Our grammatical type-logic, in other words, insists that every resource is used exactly once. And in addition to resource-sensitivity, there may be certain structural manipulations that are allowable in one language as opposed to another. To control these, there is a module of non-logical axioms (so-called structural postulates) in addition to the logical rules for the slashes. The derivation below contains such a structural move: the inference labeled $P2$ which uses associativity to rebracket the antecedent tree.

$$\dfrac{\dfrac{\textsf{the}}{np/n} \qquad \dfrac{\dfrac{\textsf{book}}{n} \qquad \dfrac{\dfrac{\textsf{that}}{(n\backslash n)/(s/np)} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\textsf{knuth}}{np} \qquad \dfrac{\dfrac{\textsf{wrote}}{(np\backslash s)/np} \qquad [\textsf{p}_1 \vdash np]^1}{\textsf{wrote} \circ \textsf{p}_1 \vdash np\backslash s}\,[/E]}{\textsf{knuth} \circ (\textsf{wrote} \circ \textsf{p}_1) \vdash s}\,[\backslash E]}{(\textsf{knuth} \circ \textsf{wrote}) \circ \textsf{p}_1 \vdash s}\,[P2]}{\textsf{knuth} \circ \textsf{wrote} \vdash s/np}\,[/I]^1}{\textsf{that} \circ (\textsf{knuth} \circ \textsf{wrote}) \vdash n\backslash n}\,[/E]}{\textsf{book} \circ (\textsf{that} \circ (\textsf{knuth} \circ \textsf{wrote})) \vdash n}\,[\backslash E]}{\textsf{the} \circ (\textsf{book} \circ (\textsf{that} \circ (\textsf{knuth} \circ \textsf{wrote}))) \vdash np}\,[/E]$$

At this point, you are perfectly ready to write your first type-logical grammar! Assign types to the words in your lexicon, and decide whether any extra structural reasoning is required. The type-inference machine of *TLG* does the rest.

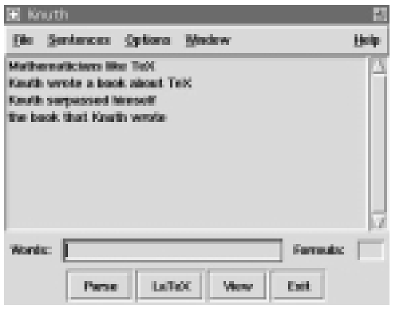
The Grail theorem prover

The Grail system, developed by the second author, is a general grammar development environment for designing and prototyping type-logical grammars. We refer the reader to (Moot 1998) for a short description of the system, which is available under the GNU General Public License agreement from `ftp.let.uu.nl/pub/users/moot`. The original Grail implementation presupposes a unix environment. It uses the following software components:

- SICStus Prolog: the programming language for the kernel;
- Tcl/Tk for the graphical user interface;
- a standard teTeX environment for the visualization/export of derivations.

In a Grail session, the user can design a grammar fragment, which in the *TLG* setting comes down to the following:

- assign formulas (and meaning programs) to words in the lexicon or edit formulas already in the lexicon,
- add or modify structural rewrite rules,



Mathematicians like TeX
Knuth wrote a book about TeX
Knuth surpassed himself
the book that Knuth wrote

FIGURE 1: The Grail main window

- ○ and finally, to run the theorem prover on sample expressions to see which expressions are grammatical in the specified grammar fragment by trying to find a derivation for them.

The theorem prover can operate either automatically or interactively. In interactive mode, the user decides which of several possible subproofs to try first, or to abandon subproofs which the user knows cannot succeed, even though the theorem prover might take a very long time to discover that. Another possibility is that the user is only interested in some of the proofs. The interactive debugger is based on proof net technology — a proof-theoretic framework specially designed for resource-sensitive deductive systems. In Figure 1, we give a screenshot of the main window of the GUI. Figure 2 shows a proof net for the derivation of the sentence 'Knuth surpassed himself'.

When successful derivations have been found, Grail can convert the internal representation of the proof objects to natural deductions in the form of LATEX output. We have already seen some examples in the previous section. Though the internal representation of derivations contains a lot of information, the structure is basically simple: a proof object consists of a conclusion together with a list of proof objects

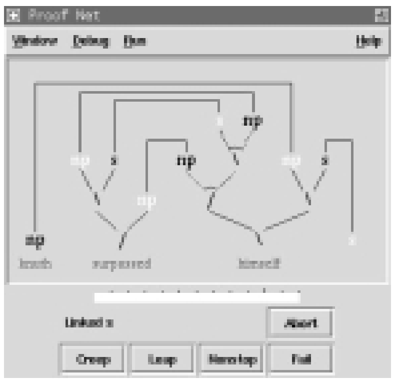


FIGURE 2: The proof net debugger window

$$\cfrac{\cfrac{\textsf{knuth}}{np} \qquad \cfrac{\cfrac{\textsf{surpassed}}{(np\backslash s)/np} \quad \cfrac{\textsf{himself}}{((np\backslash s)/np)\backslash(np\backslash s)}}{\textsf{surpassed} \circ \textsf{himself} \vdash np\backslash s}[\backslash E]}{\textsf{knuth} \circ (\textsf{surpassed} \circ \textsf{himself}) \vdash s}[\backslash E]$$

FIGURE 3: PRAWITZ STYLE NATURAL DEDUCTION OUTPUT

1.	$\textsf{knuth} : np - \mathbf{knuth}$	*Lex*
2.	$\textsf{surpassed} : (np\backslash s)/np - \mathbf{surpass}$	*Lex*
3.	$\textsf{himself} : ((np\backslash s)/np)\backslash(np\backslash s) - \lambda z_2.\lambda x_3.((z_2\ x_3)\ x_3)$	*Lex*
4.	$\textsf{surpassed} \circ \textsf{himself} : np\backslash s - \lambda x_3.((\mathbf{surpass}\ x_3)\ x_3)$	$\backslash E\ (2,3)$
5.	$\textsf{knuth} \circ (\textsf{surpassed} \circ \textsf{himself}) : s - ((\mathbf{surpass\ knuth})\ \mathbf{knuth})$	$\backslash E\ (1,4)$

1. $((\mathbf{surpass\ knuth})\ \mathbf{knuth})$

FIGURE 4: FITCH STYLE NATURAL DEDUCTION OUTPUT

which validate this conclusion and the LATEX output is produced by recursively traversing this structure.

A number of parameters guide the production of the LATEX proofs. The output parameters include, for example, a choice to have proofs presented in the tree-like Prawitz output format, as shown in Figure 3, or in the list-like Fitch output format, as shown in Figure 4. The Fitch list format is handy when the user chooses to include the meaning assembly in a derivation: tree format quickly exceeds the printed page format in these cases. The Prawitz derivations are typeset in LATEX using the `proof.sty` package of Tatsuta (1997), but, as the conversion to LATEX is quite modular, it would be possible to generate proofs in different formats, using for example the `prooftree.sty` of Taylor (1996) as an alternative.

An extract of the LATEX source for Figure 3 is shown in Figure 5. Automated generation of derivations is agreeable especially in the case of more complex examples, which can be frustrating to produce or edit manually.

```
\infer[\bo \bs E \bc^{}]
   {\textsf{knuth}\circ_{}(\textsf{surpassed}\circ_{}\textsf{himself})\vdash s}{
   \infer{np}{\textsf{knuth}}
&
   \infer[\bo \bs E \bc^{}]
      {\textsf{surpassed}\circ_{}\textsf{himself} \vdash np \bs_{}s}{
      \infer{(np \bs_{}s) /_{}np}{\textsf{surpassed}}
   &
      \infer{((np \bs_{}s) /_{}np) \bs_{}(np \bs_{}s)}{\textsf{himself}}
   }
}
```

FIGURE 5: THE LATEX SOURCE FOR FIGURE 3

Grail on the web

The Tcl/Tk-based graphical interface to Grail provides a pleasant working environment, especially for users unfamiliar with Prolog. But it is a complex platform, dependent on the interaction of a number of programs—Prolog, latex, Tcl/Tk. The World Wide Web provides an environment that in principle allows access to Grail's facilities for grammatical research, testing, and development to anyone with a graphical browser, and this is the objective of our current efforts. Moving Grail onto the web involves a natural series of stages, which we describe below.

Command line interaction

It would be difficult to manage via a browser the interaction of a remote user and the Tcl/Tk graphical interface. But it is not so difficult to manage this interaction directly via the SICStus Prolog command line prompt. In particular, if one wants to test whether a given expression is assignable a particular type relative to a particular fragment, it is enough to start Grail under SICStus, load the fragment in question—simply an additional sequence of Prolog clauses—then pass the expression and the goal formula to Grail. The results of the parse can be written out as a latex file and displayed in .dvi or .ps or .pdf format, as discussed above.

One commonly executes these steps sequentially, as shown below, suppressing unnecessary detail and extraneous messages. The command line instruction `% sicstus` initiates a session with SICStus Prolog and the command `consult('notcl2000.pl')` loads Grail without the Tcl/Tk interface. One can then load a fragment—here `consult('knuth.pl')` —and test whether the expression Knuth surpassed himself can be assigned the type s by entering the clause `tex([knuth,surpassed,himself],s)`.

• • •

```
% sicstus
SICStus 3.8.5 (sparc-solaris-5.7): Fri Oct 27 10:12:22 MET DST 2000
Licensed to let.uu.nl
| ?- consult('notcl2000.pl').
{consulting notcl2000.pl...}
========================
= Welcome to Grail 2.0 =
========================
{Warning: something appears to be wrong with the TclTk library!}
{You can still use Grail, but you will have limited functionality}

yes

| ?- consult('knuth.pl').
{consulting knuth.pl...}
{consulted knuth.pl in module user, 20 msec 6952 bytes}

yes
```

```
| ?- tex([knuth,surpassed,himself],s).

===
[knuth,surpassed,himself] => s
===
Lookup: 0

Max # links: 12
===

(FAILED). surpass(knuth,knuth)

(knuth *[] (((G \[] (E *[] (surpassed *[] G))) /[] E) *[] himself))-->>
IRREDUCIBLE

===

1. surpass(knuth,knuth)

(knuth *[] (((G \[] (G *[] (surpassed *[] E))) /[] E) *[] himself))-->>
(knuth *[] (surpassed *[] himself))

===

Telling LaTeX output directory eg.tex

1 solution found.
CPU Time used: 0.200

Telling LaTeX output directory eg.tex

true ? latex ready
```

• • •

The final comments indicate that Grail has written out the proof to the file `eg.tex` in a way that can be inserted directly in a LATEX document, as we have seen in Figure 5.

Shell interaction

SICStus provides facilities to combine all the steps of program initiation and input just illustrated into a single command line, using the built-in SICStus predicates. During a SICStus session, a call to the predicate `save_program` saves the state of the run of SICStus in a way that allows it to be restarted at exactly the same point.

For instance, if a SICStus program contains the clause

```
  make_savedstate:- save_program(wwwgrailstate, startup).
```

the run state will be saved as the executable *wwwgrailstate*, and upon reinitiation will attempt to prove the predicate `startup/0`. To restart SICStus in this way, one calls SICStus with the *-r* flag:

```
  % sicstus -r wwwgrailstate
```

Finally, there is an additional flag which allows one to pass a sequence of arguments to the re-initiated state, bound as a list of Prolog atoms to the special built-in constant *argv*, as shown below.

```
  % sicstus -r wwwgrailstate -a arg1 arg2 ...
```

Now, for our purposes, these arguments can provide information about a particular fragment to be loaded, a variety of choices about proof display, etc., and finally, the goal formula of the expression to be tested and the list of words making up the expression itself. What remains is to unpack the list *argv* inside and redeploy these individual arguments appropriately.

Here is an example in which the first argument following the *-a* flag selects the fragment *knuth.pl*, the second through the fifth arguments set switches governing the proof format, the sixth argument (*s*) sets the goal formula, and the remaining arguments specify the list of words of the expression to be tested.

```
% sicstus -r wwwgrailstate
-a knuth yes yes yes inactive nd s knuth surpassed himself
{restoring wwwgrailstate...}
{wwwgrailstate restored in 80 msec 513808 bytes}
{consulting knuth.pl...}
{consulted knuth.pl in module user, 20 msec 7064 bytes}


===
[knuth,surpassed,himself] => s
===

Lookup: 0

Max # links: 12
===

(FAILED). surpass(knuth,knuth)

(knuth *[] (((G \[] (E *[] (surpassed *[] G))) /[] E) *[] himself))-->>
IRREDUCIBLE

===

1. surpass(knuth,knuth)
```

```
(knuth *[] (((G \[] (G *[] (surpassed *[] E))) /[] E) *[] himself))-->>
(knuth *[] (surpassed *[] himself))

===

1 solution found.
CPU Time used: 0.200
```

• • •

Although we will not discuss here how the list of arguments bound to *argv* is treated internally to the SICStus code in *wwwgrailstate*, the reader may observe the similarity of the standard error message printed out immediately above and the standard error message encountered earlier in the course of our interactive command-line session with Grail.

Web interaction

From this point, it is straightforward to lift the interaction to the web. The arguments are encoded in an active form (or simply passed directly using the standard URL syntax for CGI programming). The form document can be prepared as an HTML document or as an active PDF document, using Sebastian Rahtz's hyperref package and Han The Thanh's pdfLATEX program. We will come back to details of this step momentarily, after looking briefly at how the server is set up to deal with such an interaction.

Lincoln Stein's Perl module CGI.pm makes it especially simple to grab these arguments, check them for correctness ('untainting'), and pass them to the saved SICStus state. And the parse can be written out in latex, then passed back to the user's browser as a .pdf file (using pdfLATEX).

Fragment display

A disadvantage of the above setup is that the end user must have a reasonable idea of the capabilities of each fragment: what is its lexicon? what kinds of grammatical questions does it address? As an aid to the user, it would be helpful to display the fragment in a pleasant form, providing all the information the user needs. Grail already has facilities to spell out the properties of fragments in latex—including postulates, lexicon, and stored examples. One can collect these as a static display, which can be stored on a web server and accessed as a PDF document. A better idea is to utilize the capacities built into the hyperref package, so that the document becomes an active PDF document. In particular, example expressions of the fragment now take the form of active links: a click of the mouse triggers the whole series of events described above, sending appropriate form data to the Perl script described above, and returning the results of the parse attempt as a PDF document. Additionally, the active form can contain text fields in which the user can enter arbitrarily constructed examples (compatible with the fragment), rather than simply selecting among the examples listed in the fragment specification itself. To see the display of the fragment *knuth.pl*, visit http://grail.let.uu.nl/knuth.pdf.

Static libraries of fragments

Once it is possible to display a single fragment which allows active interaction with the GRAIL environment over the web, it is immediately possible to provide access to a library of fragments, each displayed as a PDF document prepared using the hyperref package and pdfLATEX. The interested reader can visit the fragments section of http://grail.let.uu.nl/tour.pdf for an example.

Dynamic libraries

Static libraries have their limitations. Most obviously, they depend on a webmaster to install fragments and make their displays available. For broader teaching, development, and research, it is preferable for web-based users to construct their own fragments and access displays of them, all in a way that allows the fragments to be tested and improved. Through the LWP.pm module, Perl makes it relatively simple to fetch remote files from hosts across the web. In brief, through interaction with text-fields in a web document, a remote GRAIL user can specify the URL of a fragment, specify a test expression and a test goal, and submit the test remotely. Resulting proof displays of the kind already seen are returned over the web. Alternatively, the user may request that a remote fragment be transformed into a an active LATEX document (as just discussed), which can be further used for teaching, development, and research.

FUTURE WORK

More complex forms of interaction involving fragment revision and editing directly via dynamic PDF documents would be desirable and are possible in principle. Also, the concept of a derivation itself is a dynamic notion: it would be nice to have an option to unfold derivations step by step in the typeset PDF document. We are currently experimenting with Stephan Lehmke's texpower bundle, which offers the required kind of functionality.

REFERENCES

Lambek, J. 1958, The mathematics of sentence structure. American Mathematical Monthly, **65**:154–170.

Lehmke, S. 2001, The TEXPower bundle. Currently available in a pre-alpha release from `http://ls1-www.cs.uni-dortmund.de/~lehmke/texpower/`.

Moortgat, M. 1997, Categorial type logics. Chapter 2, *Handbook of Logic and Language.* Elsevier/MIT Press, pp. 93–177.

Moot, R. 1998, Grail: an automated proof assistant for categorial grammar logics, *in* R. Backhouse, ed., 'Proceedings of the 1998 User Interfaces for Theorem Provers Conference', pp. 120–129.

Radhakrishnan, C.V. 1999, 'Pdfscreen.sty', Comprehensive TEX Archive Network. `macros/latex/contrib/supported/pdfscreen/`.

Rahtz, S. 2000, 'Hyperref.sty', Comprehensive TEX Archive Network. `macros/latex/contrib/supported/hyperref/`.

Tatsuta, M. 1997, 'Proof.sty', Comprehensive TEX Archive Network. `macros/latex/contrib/other/proof/proof.sty`.

Taylor, P. 1996, 'Prooftree.sty', Comprehensive TEX Archive Network. `macros/generic/proofs/taylor/prooftree.sty`.