

Teaching tools for logic-based grammar development

Michael Moortgat, Richard Moot and Dick Oehrle

Utrecht Institute of Linguistics OTS
Utrecht University
Trans 10, 3512 JK Utrecht
The Netherlands
E-mail: {moortgat,oehrle}@let.uu.nl

An earlier version of this paper appeared in the Proceedings of EuroT_EX 2001. We thank Willemijn Vermaat and Bernhard Fisseni for helpful discussions of T_EXnicities.

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, and their dynamic unfolding, in a variety of formats.

We discuss a language engineering environment for type-logical computational grammars. The kernel is a theorem prover, implemented in the logic-programming language Prolog. The kernel produces L^AT_EX source code for its internal computations. The front-end displays these in a number of user-defined typeset formats. We report on our work to make the kernel accessible over the web via dynamic PDF documents.

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 T_EX-based visualisation tools of the GRAIL workbench as it was originally developed for use on a unix platform. Finally, we report on our current efforts to provide browser-based access to the GRAIL kernel via dynamic PDF documents.

1. 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. The seminal paper [1] is still highly readable; the paper is available electronically for those who don't have easy access to issues of the *American Mathematical Monthly* in the pre- \TeX era. [3] gives an up-to-date survey of the field. The mathematically-inclined \TeX user 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 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 $T \rightarrow T'$. With these functional types, one can do two things. An expression/program of type $T \rightarrow T'$ 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 $T \rightarrow T'$ 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).

$$\frac{\frac{\text{times} : \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}) \quad x : \text{Int}}{(\text{times } x) : \text{Int} \rightarrow \text{Int}} \text{ (Elim } \rightarrow)}{\frac{(\text{times } x) : \text{Int} \quad x : \text{Int}}{(\text{times } x \ x) : \text{Int}} \text{ (Elim } \rightarrow)}{\lambda x.(\text{times } x \ x) : \text{Int} \rightarrow \text{Int}} \text{ (Intro } \rightarrow)}$$

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.)

$$\frac{\text{mathematicians} \vdash np \quad \frac{\text{like} \vdash (np \backslash s)/np \quad \text{T}\mathbf{E}\mathbf{X} \vdash np}{\text{like} \circ \text{T}\mathbf{E}\mathbf{X} \vdash np \backslash s} [/E]}{\text{mathematicians} \circ (\text{like} \circ \text{T}\mathbf{E}\mathbf{X}) \vdash s} [\backslash E]$$

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 $x : \text{Int}$ 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.

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.

2. 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 [4] for a short description of the system, which is available

$$\begin{array}{c}
\frac{\text{the}}{np/n} \quad \frac{\text{book}}{n} \quad \frac{\text{that}}{(n \setminus n) / (s / np)} \quad \frac{\frac{\text{knuth}}{np} \quad \frac{\frac{\text{wrote}}{(np \setminus s) / np} \quad [p_1 \vdash np]^1}{\text{wrote} \circ p_1 \vdash np \setminus s} [E]}{\text{knuth} \circ (\text{wrote} \circ p_1) \vdash s} [P2]}{\text{knuth} \circ \text{wrote} \vdash s / np} [I]^1}{\text{that} \circ (\text{knuth} \circ \text{wrote}) \vdash n \setminus n} [E]}{\text{book} \circ (\text{that} \circ (\text{knuth} \circ \text{wrote})) \vdash n} [E]}{\text{the} \circ (\text{book} \circ (\text{that} \circ (\text{knuth} \circ \text{wrote}))) \vdash np} [E]
\end{array}$$

Figure 1: Natural deduction derivation: logical and structural rules.

under the GNU General Public License agreement from `ftp://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,
- 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 prooftheoretic framework specially

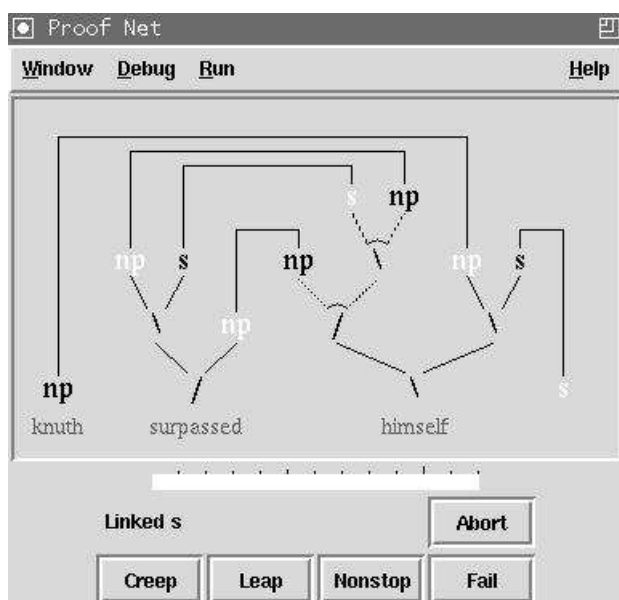


Figure 2: The proof net debugger window

designed for resource-sensitive deductive systems. Figure 2 shows a proof net for the derivation of the sentence ‘Knuth surpassed himself’. Lexical formulas are unfolded up to atomic literals. Literals are signed with an input or output polarity. A net is wellformed if there is a matching of literals with opposite polarities, and if some extra graphtheoretic conditions are met. The interested reader is referred to [5] for details.

When successful derivations have been found, GRAIL stores these proof objects in an internal representation format. An example is given in Figure 3. The internal format is not for human consumption, but it contains all the necessary information for the conversion of the proof objects to natural deductions in the form of L^AT_EX output. The internal representation of derivations may look forbidding; yet, the structure is basically simple. A proof object consists of a conclusion together with a list of proof objects which validate this conclusion. L^AT_EX output is produced by recursively traversing this structure.

A number of parameters guide the production of the L^AT_EX proofs. The output parameters include, for example, a choice to have proofs presented in the tree-like Prawitz output format, as shown in Figure 4, or in the list-like Fitch output format, as shown in Figure 5. 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.

An extract of the L^AT_EX source for Figure 4 is shown in Figure 6. The Prawitz derivations are typeset using the `proof.sty` package of [8]. The `\infer` command from this package takes an optional rule label, conclusion, and (&-separated) premise(s) as arguments. The subscripts and superscripts on rule labels and connectives remain empty in this example. They are for extra control information, which the user can enable or disable.

The reader will have noticed that core notion of ‘proof’ for type-logical grammatical derivations is inherently *dynamic*: a derivations is a sequence of inference steps, leading from axioms (lexical assumptions) to the desired conclusion. This naturally suggests a dynamic display format, with a stepwise unfolding of the proof object. The tools we use for dynamic display were developed by Bernhard Fisseni, as part of a student project in our computational linguistics program. The basis is an expanded version of `\infer` from `proof.sty`, taking advantage of the `\stepwise` family of commands from the `texpower` package of [2]. The kernel computes the sequencing order of derivational steps from the internal proof object. The choice for bottom-up or top-down unfolding is left to the user. In the first case, one assembles the desired end result starting from lexical assumptions; the second option decomposes the end result in its atomic (lexical) parts. For an illustration, we refer the reader

```

N: 1 ; Mean: $\iota$(^K.(write(knuth,K) & book(K))) ;
rule(dre([]),(the *[] (book *[] (that *[] (knuth *[] wrote)))) ,np,B(D(^E.H(E)(G))(C)),
[rule.lex,the,(np /[] n),B,[]],
rule(dle([]),(book *[] (that *[] (knuth *[] wrote))),n,D(^E.H(E)(G))(C),
[rule.lex,book,n,C,[]],
rule(dre([]),(that *[] (knuth *[] wrote)),(n \[] n),D(^E.H(E)(G)),
[rule.lex,that,((n \[] n) /[] (s /[] np)),D,[]],
rule(dri([],1),(knuth *[] wrote),(s /[] np),^E.H(E)(G),
[rule.P2,((knuth *[] wrote) *[] E),s,H(E)(G),
[rule.dle([]),(knuth *[] (wrote *[] E)),s,H(E)(G),
[rule.lex,knuth,np,G,[]],
rule(dre([]),(wrote *[] E),(np \[] s),H(E),
[rule.lex,wrote,((np \[] s) /[] np),H,[]],
rule(hyp(1),E,np,E,[]]]]]]]]]]])),
Con: [],Subst: [$\iota$,book,3-^I.^J.^K.(I(K) & J(K)),knuth,write], NV 8

```

Figure 3: Internal representation for the derivation of Figure 1

$$\frac{\frac{\text{knuth}}{np} \quad \frac{\frac{\text{surpassed}}{(np \setminus s) / np} \quad \frac{\text{himself}}{((np \setminus s) / np) \setminus (np \setminus s)}}{\text{surpassed} \circ \text{himself} \vdash np \setminus s} [\setminus E]}{\text{knuth} \circ (\text{surpassed} \circ \text{himself}) \vdash s} [\setminus E]$$

Figure 4: Prawitz style natural deduction output

to the <http://www.ntg.nl/eurotex/Moortgat.pdfsection12slides> of our EuroTeX 2001 presentation.

3. GRAIL on the web

The original implementation of GRAIL was designed for class use in a unix computer lab environment. Our current efforts are aimed at developing web-based forms of interaction with the GRAIL computational kernel. This type of interaction makes it possible to design more flexible and personalized e-learning tools. Also, one can reach a broader group of users, because one no longer makes platform requirements.

The architecture we are using is based on two components: CGI scripts (calling the program state representing the GRAIL kernel and the typesetting output routines), and the dynamic features of Sebastian Rahtz' `hyperref` package. The interested reader can try out two aspects of the current set-up at <http://131.211.190.177/grail/>:¹

netpdffrag Submit grammar fragments for testing.

netgrail Produce derivations of test phrases on demand.

At the core of these two applications is a saved state of the program representing the GRAIL computational kernel. In SICStus Prolog, one obtains this saved state with the `save_program/2` command, which takes a file name (say, `grail.sav` for the GRAIL saved program state) and a start-up command as arguments. The start-up command is executed when the program state is restored with the command `sicstus -r grail.sav`; further arguments can be passed to the program state with the `-a` flag.

¹ This is a temporary address. Contact the first author in case of problems.

- | | | |
|----|---|----------------------|
| 1. | $\text{knuth} : np - \mathbf{knuth}$ | <i>Lex</i> |
| 2. | $\text{surpassed} : (np \setminus s) / np - \mathbf{surpass}$ | <i>Lex</i> |
| 3. | $\text{himself} : ((np \setminus s) / np) \setminus (np \setminus s) - \lambda z_2. \lambda x_3. ((z_2 \ x_3) \ x_3)$ | <i>Lex</i> |
| 4. | $\text{surpassed} \circ \text{himself} : np \setminus s - \lambda x_3. ((\mathbf{surpass} \ x_3) \ x_3)$ | $\setminus E (2, 3)$ |
| 5. | $\text{knuth} \circ (\text{surpassed} \circ \text{himself}) : s - ((\mathbf{surpass} \ \mathbf{knuth}) \ \mathbf{knuth})$ | $\setminus E (1, 4)$ |

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

Figure 5: Fitch style natural deduction output

```

...
\newcommand{\bs}{\backslash}
\newcommand{\bo}{[]}
\newcommand{\bc}{[]}
...
\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 6: The L^AT_EX source for Figure 4

```

% knuth.pl: example fragment
% =====
% Structural rules: Name # In ---> Out.
% =====
'P2' # (A*B)*C ---> A*(B*C).
% =====
% Macros: Abbreviation := Type.
% =====
iv := np\s.
tv := iv/np.
refl := tv\iv.
% =====
% Lexicon: Word :: Type (:: Meaning)
% =====
knuth :: np.
mathematicians :: np.
tex :: np.
surpassed :: tv.
himself :: refl.
the :: np/n.
book :: n.
that :: (n\n)/(s/np).
wrote :: tv.
like :: tv.
% =====
% Test examples: Example ==> GoalType.
% =====
"Mathematicians like TeX." ==> s.
"the book that Knuth wrote" ==> np.
"Knuth surpassed himself." ==> s.

```

Figure 7: Fragment input format.

In the case of the `netpdffrag` script, the user communicates with the server via a form (html or pdf). The form asks for a URL where the server can find a user-defined grammar fragment, i.e. a set of lexical type declarations, structural rules, abbreviatory macros (optional), and test phrases, all in a simple ascii line format. The fragment is fetched through the Perl LWP module, checked for correctness ('untainted') and passed to the saved GRAIL program state. The start-up goal in this case is to produce L^AT_EX source code of the submitted fragment. A call to `pdflatex` produces a PDF document, which is sent back to the user.

The fragment L^AT_EX source code exploits the dynamic features of the `hyperref` package in its treatment of test phrases. These are typeset with the `\href{URL}{TEXT}` command. The URL argument points to the `netgrail` CGI script. On clicking a test phrase in a grammar fragment, this script submits the phrase (together with the working fragment and a number of display options) to the GRAIL program state, which produces a derivation, if the test phrase is indeed derivable, or an informative error message. The derivation is typeset in the way indicated earlier in this paper, and sent back to the user in the form of a PDF document. Figure 8 provides further details. Notice that the `\parsescript` command can be used independently of the fragment typesetting routines, for example, when an author proposes a grammatical analysis and wants the derivations for example sentences to be available on demand.

4. Further developments

The tools described here are being used in the Linguistics program and the CKI (Cognitive Science and AI) program of the undergraduate curriculum at Utrecht University. They have also been used in graduate and postgraduate teaching at the Utrecht Institute of Linguistics (OTS) and at a number of European Summer Schools in Logic, Language and Information (ESSLLI). The set-up as described here is a snapshot of work in progress. The following features are currently being developed. First, the code of the GRAIL kernel is being refactored by Gert-Jan Verhoog and Xander Schrijen (both CKI), to optimize modularity and thus facilitate anticipated alternative interaction formats. Secondly, the system architecture discussed here is not specific to type-logical grammar development, but can be used for other logic-based grammar formalisms. Willemijn Vermaat (OTS) has designed web-based teaching tools for so-called Minimalist Grammars, using essentially the same software components as those described in this paper. The computational kernel, in this case, is a general parser/theorem prover for grammars in the minimalist format, developed by Ed Stabler.

```

\hyperbaseurl{http://.../cgi-bin/grail/} % base URL: server CGI directory
\newcommand{\hyperfrag}{http://.../fragments/knuth.pl} % client fragment URL
\newcommand{\parescript}[3]{\href{netgrail?% the CGI script
  url=\hyperfrag% the working fragment
  &struct=yes&sem=no&lexsem=yes&unary=inactive&mode=nd% display options
  &goal=#1% goal formula
  &test=#2% test phrase, as submitted to the script
  }{#3}}% test phrase for typesetting
...

\begin{enumerate}
\item \parescript{s}{mathematicians+like+tex}{Mathematicians like TeX.} $\vDash s$
\item \parescript{s}{knuth+surpassed+himself}{Knuth surpassed himself.} $\vDash s$
\item \parescript{s}{the+book+that+knuth+wrote}{the book that Knuth wrote}
$\vDash np$
\end{enumerate}

```

Figure 8: Generating derivations on demand.

Bibliography

- [1] Lambek, J. 1958, The mathematics of sentence structure. *American Mathematical Monthly*, **65**:154–170. Electronically available at <http://www.jstor.org>.
- [2] Lehmke, S. 2001, The T_EXPower bundle. Currently available in a pre-alpha release from <http://ls1-www.cs.uni-dortmund.de/~lehmke/tepower/>.
- [3] Moortgat, M. 2002, Categorical grammar and formal semantics. *Encyclopedia of Cognitive Science*, Nature Publishing Group, Macmillan (due November 2002, preliminary version available from <http://preprints.phil.uu.nl/aips/>).
- [4] 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. <ftp://ftp.let.uu.nl/pub/users/moot/uitp.ps.gz>
- [5] Moot, R. 2002, Proof Nets for Linguistic Analysis. PhD Thesis. Utrecht Institute of Linguistics OTS. Utrecht University.
- [6] Radhakrishnan, C.V. 1999, ‘Pdfscreen.sty’, <http://www.ctan.org/tex-archive/macros/latex/contrib/supported/pdfscreen/>.
- [7] Rahtz, S. 2000, ‘Hyperref.sty’, <http://www.ctan.org/tex-archive/macros/latex/contrib/supported/hyperref/>.
- [8] Tatsuta, M. 1997, ‘Proof.sty’, <http://www.ctan.org/tex-archive/macros/latex/contrib/other/proof/proof.sty>.