Proceedings of VEXTAL'99 - Venezia, San Servolo V.I.U. - 22-24 November 1999

Grail: an Interactive Parser for Categorial Grammars

Richard Moot
Utrecht Institute of Linguistics-OTS
3512 JK Utrecht - The Netherlands
E-mail: Richard.Moot@let.uu.nl
Fax: +3130-2536000 Tel.: +3130-2536057

1 Introduction

Since early work by Lambek [Lambek 58], linguists and logicians have been trying to specify grammars as logical theories (see e.g. [Morrill 94] or [Moortgat 97] for a good overview). An advantage of this approach over more traditional approaches to linguistics is that we can prove our grammars have abstract well-behavedness properties like soundness, completeness and consistency.

Grail is a tool which allows you to specify such logical theories and functions as a parser for the resulting grammar.

I will first give a brief introduction to categorial grammar and then show how proof search for this logic is implemented in Grail. The parser is divided in two parts: we use *proof nets* to indicate, roughly, the functor-argument structure and *term rewriting* to implement the structural constraints imposed by the language we are describing. On both levels, parsing is fully interactive. When there are multiple ways of continuing the computation, the user can select which one to try first. Benefits of this are that the user can guide the computation to a point in which he is interested, which is useful when debugging a grammar and sidesteps the computational complexity of the parser when it operates fully automated.

Grail is currently used at the University of Utrecht as a research tool and as courseware for introductory to advanced level courses in computational linguistics. It is implemented in a combination of SICStus Prolog and Tcl/Tk.

2 Categorial Grammar

Before we start with a description of the parser itself, it is useful to give at least a short introduction to modern, multimodal categorial grammars as used in [Moortgat 97].

Definition 1 *Over a finite set of atomic formulas $\mathcal{A}$ and for all modes i out of some fixed set, we define the set of* formulas *as follows*

$$\mathcal{F} ::= \mathcal{A} \mid \mathcal{F}/_i\mathcal{F} \mid \mathcal{F}\backslash_i\mathcal{F}$$

Intuitively, a formula of the form $A/_iB$ (resp. $B\backslash_iA$) looks to the right (resp. left) for a formula of type B to yield a formula of type A. The full logic supported by Grail contains three additional connectives, $\bullet_i$, $\Diamond_i$ and $\Box_i^{\downarrow}$. For the purpose of the current discussion, however, we will limit our discussion to formulas as defined above.

It is important to note that we use a *multimodal* system and that a formula $A/_aB$ may have different behaviour than, for example, a formula $A/_cB$. In example 3 we will see how different structural rules can apply depending on which mode we use.

As the linguistic structures we will talk about are *trees*, we need to give some definition as to what kinds of trees we allow our linguistic expressions to have.

Definition 2 *Over a countably infinite set $\mathcal{V}$ of structural variables we define the set of* structure trees *as follows*

$$\mathcal{S} ::= \mathcal{V} \mid \mathcal{S} \circ_i \mathcal{S}$$

We will write $\mathcal{S} \vdash \mathcal{F}$ to indicate that the structure $\mathcal{S}$ is of type $\mathcal{F}$. A *lexicon* assigns some initial structural variables to formulas.

Example 1 *The following is a minimal lexicon*

zaphod	$\vdash np$
marvin	$\vdash np$
snores	$\vdash np\backslash_a s$
likes	$\vdash (np\backslash_a s)/_a np$
anything	$\vdash (s/_a np)\backslash_a s$

Noun phrases, like 'zaphod' and 'marvin' are assigned an atomic formula np. An intransitive verb, like 'snores', is something which yields a sentence s when it finds such an np to its left, whereas the transitive verb 'likes' yields an intransitive verb when it finds an np to its right. The quantifier 'anything' is not a simple np but gets, in the tradition of Montague semantics, a higher order formula: it yields an s whenever it finds to its left an s missing an np.

The natural deduction rules tell us how to combine these lexical assignments to form larger expressions in accordance with the meaning of the connectives. Each connective has an elimination and an introduction rule.

$$\frac{X \vdash A/_iB \quad Y \vdash B}{X \circ_i Y \vdash A}[/E] \qquad \frac{\begin{array}{c}[\mathrm{p}_n \vdash B]^n \\ \vdots \\ X \circ_i \mathrm{p}_n \vdash A\end{array}}{X : A/_iB}[/I]^n$$

$$\frac{Y \vdash B \quad X \vdash B\backslash_i A}{Y \circ_i X \vdash A}[\backslash E] \qquad \frac{\begin{array}{c}[\mathrm{p}_n \vdash B]^n \\ \vdots \\ \mathrm{p}_n \circ_i X \vdash A\end{array}}{X \vdash B\backslash_i A}[\backslash I]^n$$

The elimination rule for $\backslash$ tells us, that whenever we have some structure X of type $B\backslash_i A$ and a structure Y of type B the combination of these two structures $Y \circ_i X$ is of type A.

The introduction rule tells us we can cancel a single assumption p of type B, provided that this assumption is on the right branch of the current structure tree. We put square brackets around the formula and the structural variable p to indicate it has been canceled. A natural deduction proof in our system is a tree consisting of applications of the rules above, where all non-canceled leaves are elements of the lexicon.

Example 2 *We can show 'zaphod snores' is a well-formed sentence according to our grammar by the following derivation*

$$\frac{\text{zaphod} \vdash np \quad \text{snores} \vdash np\backslash_a s}{\text{zaphod} \circ_a \text{snores} \vdash s}[\backslash E]$$

In addition to the logical rules, which are assumed to be linguistic universals, we can have a number of *structural rules*, which can vary from language to language. As their name suggests, structural rules operate on structure trees only and are of the following general form

$$\frac{Z[X] \vdash A}{Z[Y] \vdash A}[SR]$$

where the notation $Z[X]$ signifies a structure tree Z with a distinguished subtree occurrence X. We restrict X and Y to be structure trees with variables $X_1, \ldots, X_n$ (which range over structure trees) as leaves, with the condition that each variable in the conversion is mentioned exactly once in X and exactly once in Y. In other words, copying and deletion of structure is not allowed.

Example 3 *Typical examples of valid structural rules are the following*

$$\frac{Z[(X_1 \circ_a X_2) \circ_a X_3] \vdash A}{Z[X_1 \circ_a (X_2 \circ_a X_3)] \vdash A}[\mathit{Ass1}]$$

$$\frac{Z[X_1 \circ_a (X_2 \circ_a X_3)] \vdash A}{Z[(X_1 \circ_a X_2) \circ_a X_3] \vdash A}[\mathit{Ass2}]$$

$$\frac{Z[X_2 \circ_c X_1] \vdash A}{Z[X_1 \circ_c X_2] \vdash A}[\mathit{Com}]$$

which tell us that mode a is associative and mode c is commutative.

Example 4 *As an illustration, suppose we have a lexicon as given in example 1. Then we can, with the help of the structural rule [Ass2] as defined above, give the derivation of 'zaphod likes anything' as shown in figure 1 on the following page.*

3 Proof Nets

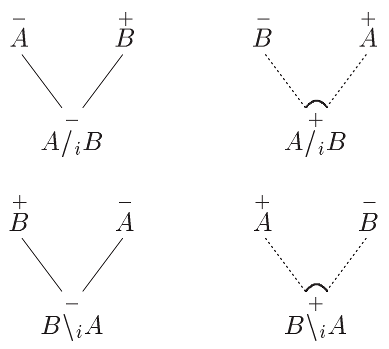
As you will have noted, the natural deduction calculus has two components: a logical and a structural. For reasons of efficiency, Grail uses a different but equivalent calculus, where this division is even more clear. The logical component is based on proof nets, which are proof-theoretic innovations developed for linear logic (see for example [Girard 87] or [Danos 90]), whereas the structural component is a term rewrite system we will discuss in the next section.

Before we introduce proof nets, we need an auxiliary notion of *polarity*. The polarity of a formula

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{zaphod} \vdash np \quad \dfrac{\text{likes} \vdash (np\backslash_a s)/_a np \quad [\text{p}_1 \vdash np]^1}{\text{likes} \circ_a \text{p}_1 \vdash np\backslash_a s}[/E]}{\text{zaphod} \circ_a (\text{likes} \circ_a \text{p}_1) \vdash s}[\backslash E]}{(\text{zaphod} \circ_a \text{likes}) \circ_a \text{p}_1 \vdash s}[Ass2]}{\text{zaphod} \circ_a \text{likes} \vdash s/_a np}[/I]^1 \quad \text{anything} \vdash (s/_a np)\backslash_a s}{(\text{zaphod} \circ_a \text{likes}) \circ_a \text{anything} \vdash s}[\backslash E]$$

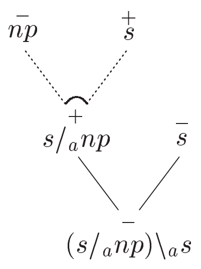
Figure 1: Derivation of ‘zaphod likes anything’

is either positive or negative, which we will indicate here by a superscript. The intuition is that negative formulas are outputs and that positive formulas are inputs. Formulas from the lexicon start out as negative, whereas the polarity of the goal formula starts out as positive. We decompose formulas according their polarity as follows



$\overset{-}{A}$ $\overset{+}{B}$ $\overset{-}{A/_i B}$
$\overset{-}{B}$ $\overset{+}{A}$ $\overset{+}{A/_i B}$
$\overset{+}{B}$ $\overset{-}{A}$ $\overset{-}{B\backslash_i A}$
$\overset{+}{A}$ $\overset{-}{B}$ $\overset{+}{B\backslash_i A}$

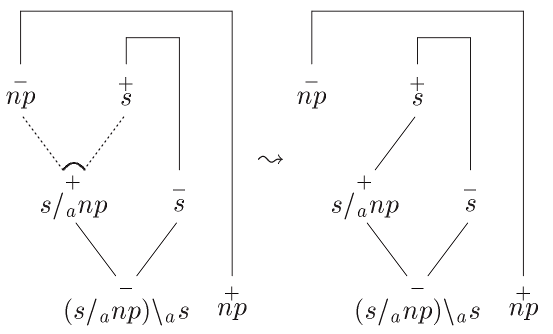
Example 5 *We would decompose the lexical entry for ‘anything’ given in example 1 as follows indicating that it produces an np as output and an s as both input and output.*



$\overset{-}{np}$ $\overset{+}{s}$
$\overset{+}{s/_a np}$ $\overset{-}{s}$
$\overset{-}{(s/_a np)\backslash_a s}$

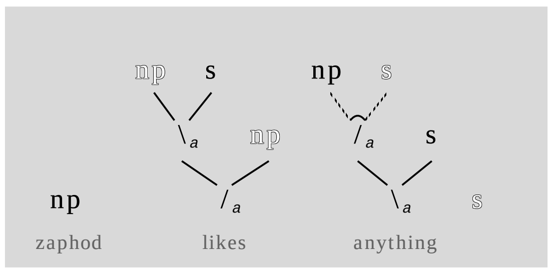
You will have noticed we distinguish between solid and dotted links. The solid links correspond to ‘mirrored’ versions of the elimination rules from natural deduction, the dotted links to the introduction rules. We can see the dotted links as some sort of ‘switches’ which we can be set to either the left or the right. In order to produce a *proof net* we need to connect all atomic formulas is such a way that we satisfy all input/output requirements and that for each switching the resulting graph is acyclic and connected.

Example 6 *Below, we show the decompositions of a negative $(s/_a np)\backslash_a s$ formula and a positive np formula together with a complete linking of all atomic formulas. The resulting structure is not a proof net, however, because we can set the single switch to the right, as shown, after which we will have produced a graph which is both cyclic and disconnected.*



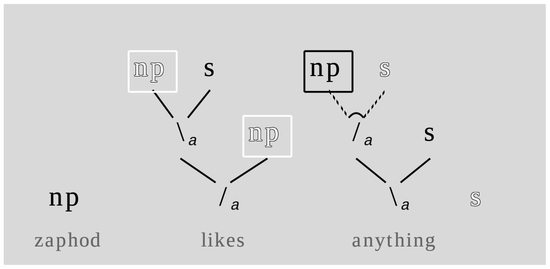
$\overset{-}{np}$ $\overset{+}{s}$ $\overset{+}{s/_a np}$ $\overset{-}{s}$ $\overset{-}{(s/_a np)\backslash_a s}$ $\overset{+}{np}$
$\rightsquigarrow$
$\overset{-}{np}$ $\overset{+}{s}$ $\overset{+}{s/_a np}$ $\overset{-}{s}$ $\overset{-}{(s/_a np)\backslash_a s}$ $\overset{+}{np}$

Returning to the derivation of example 4, when Grail returns from the lexicon it will display the formula decomposition trees as follows

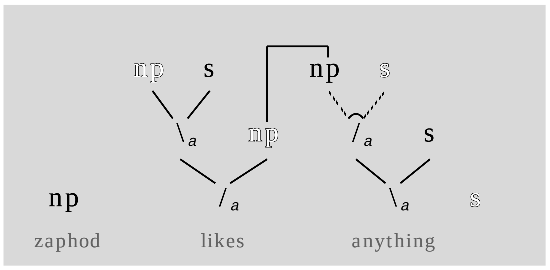


with positive atomic formulas drawn in white and negative atomic formulas drawn in black.

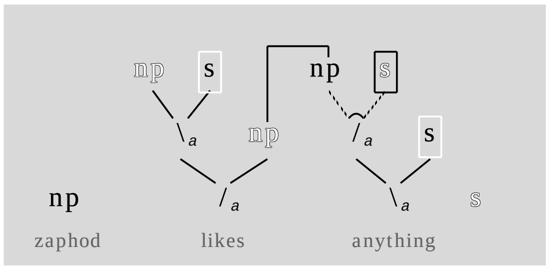
Now we need to connect all positive and negative formula occurrences in such a way that no switching produces either a cyclic or a disconnected graph. We can let the parser handle this automatically or we can manually guide the computation by selecting which formula we want to link first. In this case we select the rightmost np formula, indicated with a black box, and Grail shows us which formulas we can link it to by drawing a white box around them.

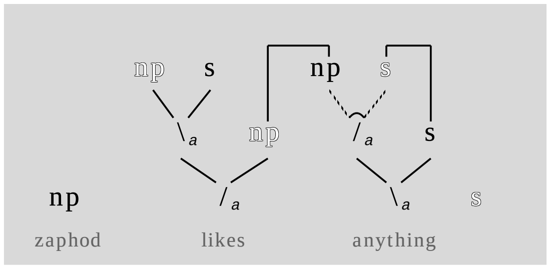


Using our educated guess that the np we have selected is the one 'likes' expects to its right instead of to its left, we link it as follows

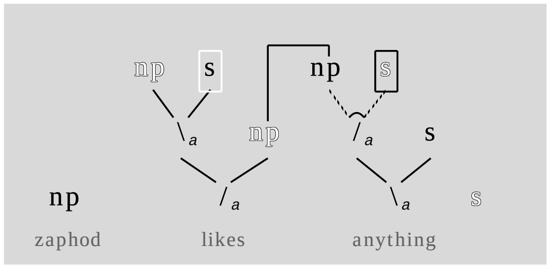


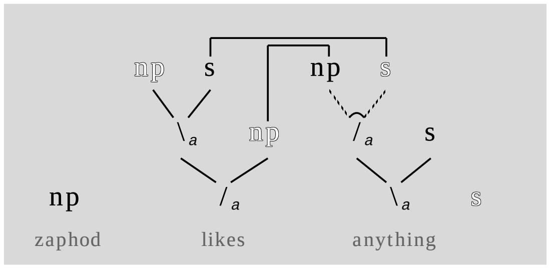
Grail keeps track of the other possible choice for us, so we don't have to worry about making a wrong choice. Next we select the s, as shown below, and we again have two choices for linking, of which we choose the rightmost one.



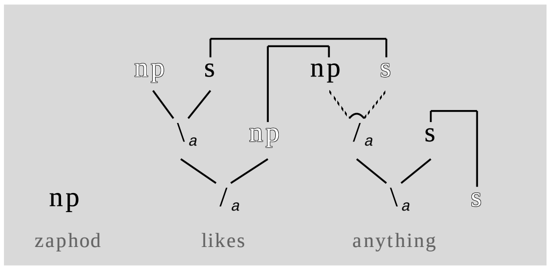


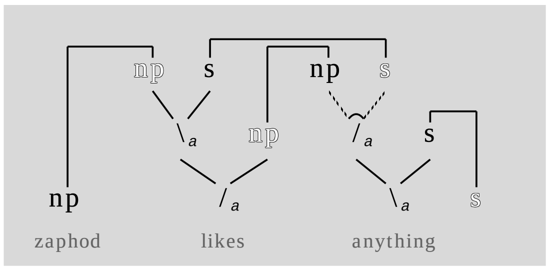
Now when we set the switch to the right, we will have a cyclic subgraph, so continuing past this point could never produce a proof net. In this case, we are forced to backtrack to the last choice we made where the rightmost negative s is no longer available, so we take the only remaining option.





Finally, we connect the last two np and s formulas, which can be done in only one way, and check that the resulting structure is acyclic and connected for both switchings.





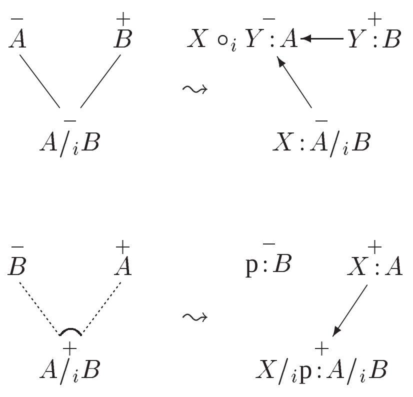
After connecting all atomic formulas, we know that all input/output requirements are satisfied and proceed to the next stage of the computation, where we check if all structural constraints can also be met. Should this fail, we return here to try for possible other linkings.

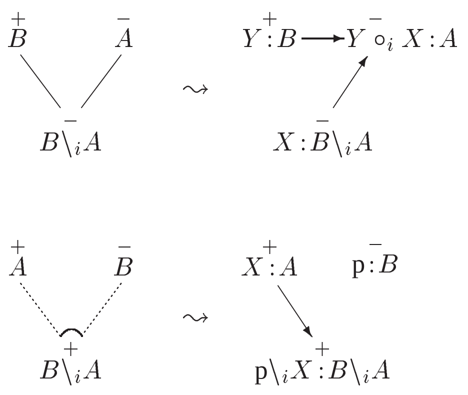
4 Structure

The proof nets discussed in the previous section only take care of the input/output requirements of formulas. As we saw, there are also *structural* requirements which a valid derivation must satisfy. How strict these requirements are depends on the available structural rules.

In Grail, these constraints are checked by a simple term rewrite system as used in [Moortgat 97]. Each connective has its own rewrite rule, which is fixed, and the user can specify any number of structural rules, which correspond directly to a like number of rewrite rules.

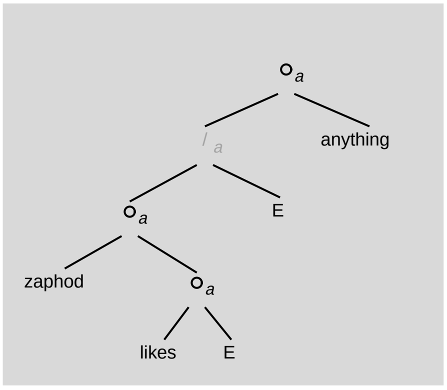
Given a proof net, we can compute the structural information from it in the following way, where the arrows indicate the flow of information in the graph



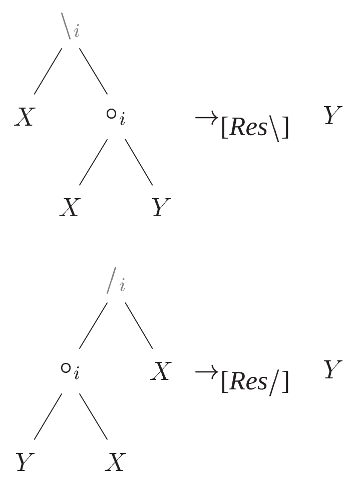


For the solid links, information travels exactly as we would expect given that they are just mirrored elimination rules; when we have a structure X assigned to $B\backslash_i A$ and a structure Y assigned to B, we assign $Y \circ_i X$ to A. For the dotted links, however, we want to demand the variable p occurs on the right (resp. left) branch of the tree X, as the corresponding introduction rules do. We implement this by assigning $X/_i\mathrm{p}$ to the $A/_iB$ formula, indicating that there we still have an unsatisfied constraint on X.

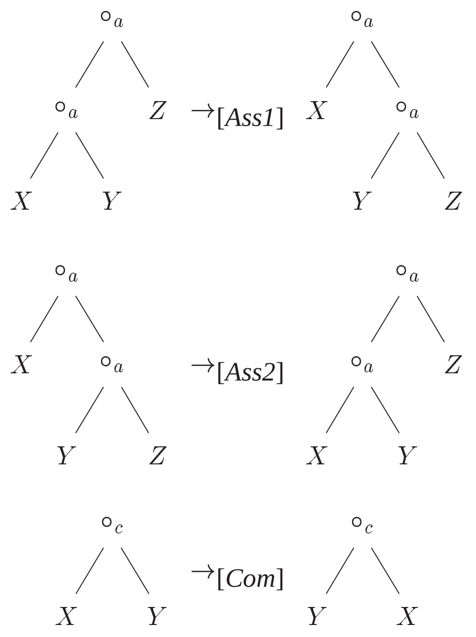
Example 7 *The structure computed by Grail for the example proof net of the previous section is the following, with the constructor $/_a$ drawn in dark grey to indicate this constraint still has to be satisfied*



The rewrite rules for the different connectives are called their *residuation rule*. For the constructors $/_i$ and $\backslash_i$, the following rules apply

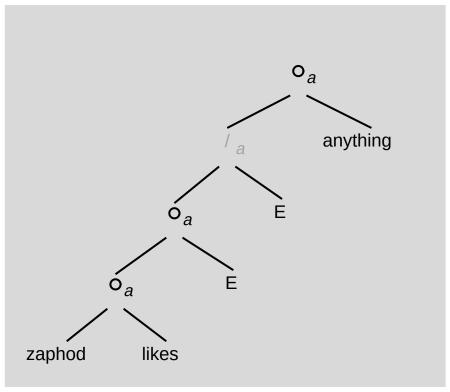


The structural rules correspond directly to rewrite rules, for example the structural rules from example 3 correspond to rewrite rules of the following form

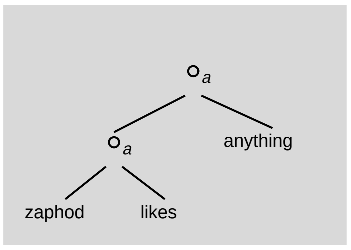


Grail allows you to click on each node of the tree to get a menu with all possible applications of a rewrite rule rooted at that node. You can select a promising rule from one of the menus, while Grail keeps track of any unvisited alternatives to your choice. When you get stuck you can either undo one of your previous choices or let the parser check if you missed a solution somewhere.

Example 8 *Given the structure of example 7, we can only apply the* [*Ass2*] *conversion and obtain the following tree*



Now we are in the right position to apply the [*Res*/] *conversion, after which we have a valid structure. We could also choose to apply* [*Ass1*] *at this point, after which we have seen the complete search space.*

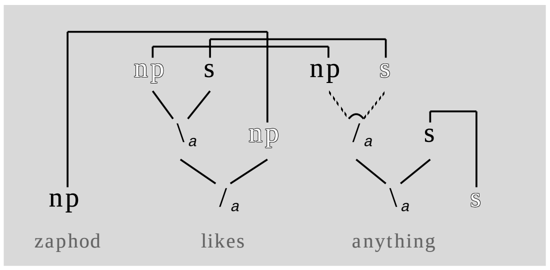


After we have applied the appropriate rewrite rules, Grail can convert the proof net to a more human-readable natural deduction proof like the one shown in figure 1.

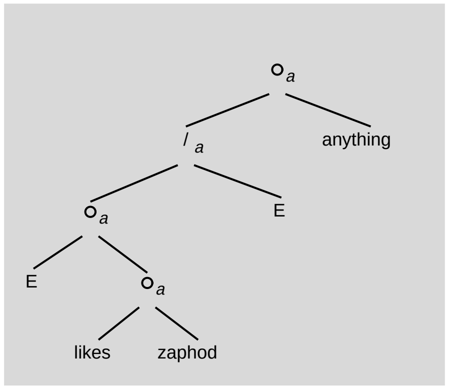
For a typical grammar designed in Grail, there are a lot more choices to make when using the rewrite rules and the search space can involve several thousands of structures. User interaction can considerably improve the performance by allowing the user to perform the intended structural conversions himself. When designing a grammar, it is often enlightening to see Grail (ab)use your carefully chosen structural rules in unintended ways, showing surprising, linguistically incorrect predictions of your grammar, or to see it fail to satisfy a critical constraint, pointing to a missing or not sufficiently general structural rule.

Example 9 *The structure computed for example 7 showed us we needed the* [*Ass2*] *structural rule.*

There is also a second proof net for 'zaphod likes anything', namely the following



for which Grail computes the following structure



Should we want to be able to meet all structural constraints here, we would need the [*Com*] *structural conversion for mode a. However, by adding this rule our grammar fragment would make the incorrect prediction that word order in English is completely free, so rejecting this structure is justified.*

5 Conclusions

Though we have skimmed over a lot of the details, we hope to have given an impression of the basic operation of the Grail parser and of how it implements the underlying logical framework. It has been our experience that interactive parsing is a great improvement over fully automated parsing which helps both in the teaching categorial grammars and in the designing and debugging of grammar fragments.

References

[Danos 90] Danos, V., *La Logique Linéaire Appliquée à l'étude de Divers Processus de Normalisation.* Thèse de Doctorat, Université de Paris VII, 1990.

[Gabbay 96] Gabbay, D., *Labeled Deductive Systems I.* Clarendon Press, Oxford, 1996.

[Girard 87] Girard, J.Y., *Linear Logic.* Theoretical Computer Science **50**, 1987, pp. 1-102.

[Girard e.a. 95] Girard, J.Y., Y. Lafont and L. Regnier (eds.), *Advances in Linear Logic*, London Mathematical Society Lecture Notes, Cambridge University Press, 1995.

[Lambek 58] Lambek, J., *The Mathematics of Sentence Structure.* American Mathematical Monthly **65**, 1958, pp. 154-170.

[Moortgat 97] Moortgat, M., *Categorial Type Logics.* Chapter 2 of Benthem, J. van, and A. ter Meulen (eds.) *Handbook of Logic and Language.* Elsevier, 1997.

[Morrill 94] Morrill, G., *Type Logical Grammar. Categorial Logic of Signs*, Kluwer, Dordrecht, 1994.

[Roorda 91] Roorda, D., *Resource Logics: A Proof Theoretical Study.* PhD Thesis, University of Amsterdam, 1991.

A Obtaining Grail

Grail and its source code are available under the GNU General Public License by anonymous ftp from `ftp.let.uu.nl` in directory `/pub/users/moot/`. See the `README` file for the latest details.

To run Grail it is necessary to have SICStus Prolog version 3.5 or later, together with the Tcl/Tk libraries, installed on your computer. In addition you will need $\LaTeX 2_\epsilon$ for the natural deduction output.