PARSING AS NATURAL DEDUCTION

Esther König

Universität Stuttgart
Institut für Maschinelle Sprachverarbeitung,
Keplerstrasse 17, D-7000 Stuttgart 1, FRG

Abstract

The logic behind parsers for categorial grammars can be formalized in several different ways. Lambek Calculus (LC) constitutes an example for a natural deduction[1] style parsing method.

In natural language processing, the task of a parser usually consists in finding derivations for all different readings of a sentence. The original Lambek Calculus, when it is used as a parser/theorem prover, has the undesirable property of allowing for the derivation of more than one proof for a reading of a sentence, in the general case.

In order to overcome this inconvenience and to turn Lambek Calculus into a reasonable parsing method, we show the existence of "relative" normal form proof trees and make use of their properties to constrain the proof procedure in the desired way.

1 Introduction

Sophisticated techniques have been developed for the implementation of parsers for (augmented) context-free grammars. [Pereira/Warren 1983] gave a characterization of these parsers as being *resolution based theorem provers*. Resolution might be taken as an instance of Hilbert-style theorem proving, where there is one inference rule (e.g. Modus Ponens or some other kind of *Cut Rule*) which allows for deriving theorems from a set of axioms. In the case of parsing, the grammar rules and the lexicon would be the axioms.

When categorial grammars were discovered for computational linguistics, the most obvious way to design parsers for categorial grammars seemed to apply the existing methods: The few combination rules and the lexicon constitute the set of axioms, from which theorems are derived by a resolution rule. However, this strategy leads to unsatisfactory results, in so far as extended categorial grammars, which make use of combination rules like functional composition and type raising, provide for a proliferation of derivations for the same reading of a sentence. This phenomenon has been dubbed the *spurious ambiguity problem* [Pareschi/Steedman 1987]. One solution to this problem is to describe normal forms for equivalent derivations and to use this knowledge to prune the search space of the parsing process [Hepple/Morrill 1989].

Other approaches to cope with the problem of spurious ambiguity take into account the peculiarities of categorial grammars compared to grammars with "context-free skeleton". One characteristic of categorial grammars is the shift of information from the grammar rules into the lexicon: grammar rules are mere combination schemata whereas syntactic categories do not have to be atomic items as in the "context-free" formalisms, but can also be structured objects as well.

The inference rule of a Hilbert-style deduction system does not refer to the internal structure of the propositions which it deals with. The alternative to Hilbert-style deduction is natural deduction (in the broad sense of the word) which is "natural" in so far as at least some of the inference rules of a natural deduction system describe explicitly how logical operators have to be treated. Therefore natural deduction style proof systems are in principle good candidates to function as a framework for categorial grammar parsers. If one considers categories as formulae, then a proof system would have to refer to the operators which are used in those formulae.

[1] "natural deduction" is used here in its broad sense, i.e. natural deduction as opposed to Hilbert-style deduction

The natural deduction approach to parsing with categorial grammars splits up into two general mainstreams both of which use the Gentzen sequent representation to state the corresponding calculi. The first alternative is to take a general purpose calculus and propose an adequate translation of categories into formulae of this logic. An example for this approach has been carried out by Pareschi [Pareschi 1988], [Pareschi 1989]. On the other hand, one might use a specialized calculus. Lambek proposed such a calculus for categorial grammar more than three decades ago [Lambek 1958].

The aim of this paper is to describe how Lambek Calculus can be implemented in such a way that it serves as an efficient parsing mechanism. To achieve this goal, the main drawback of the original Lambek Calculus, which consists of a version of the "spurious ambiguity problem", has to be overcome. In Lambek Calculus, this overgeneration of derivations is due to the fact that the calculus itself does not give enough constraints on the order in which the inference rules have to be applied.

In section 2 of the paper, we present Lambek Calculus in more detail. Section 3 consists of the proof for the existence of normal form proof trees relative to the readings of a sentence. Based on this result, the parsing mechanism is described in section 4.

2 Lambek Calculus

In the following, we restrain ourselves to cut-free and product-free Lambek Calculus, a calculus which still allows us to infer infinitely many derived rules such as Geach-rule, functional composition etc. [Zielonka 1981]. The cut-free and product-free Lambek Calculus is given in figures 1 and 2. Be aware of the fact that we did not adopt Lambek's representation of complex categories. Proofs in Lambek Calculus can be represented as trees whose nodes are annotated with sequents. An example is given in figure 3. A lexical lookup step which replaces lexemes by their corresponding categories has to precede the actual theorem proving process. For this reason, the categories in the antecedens of the input sequent will also be called *lexical categories*. We introduce the notions of *head*, *goal category*, and *current functor*: The *head* of a category is its "innermost" value category: The head of a basic category is the category itself. The head of a complex category is the head of its value category. The category in the succedens of a sequent is called *goal category*. The category which is "decomposed" by an inference rule application is called *current functor*.

Basic Category:
a constant
Rightward Looking Category:
if *value* and *argument* are categories,
then *(value/argument)* is a category
Leftward Looking Category:
if *value* and *argument* are categories,
then *(value\argument)* is a category

Figure 1: *Definition of categories*

axiom scheme

(axiom) $x \rightarrow x$

logical rules

(/:left) $\frac{T \rightarrow y \qquad U, x, V \rightarrow z}{U, (x/y), T, V \rightarrow z}$

(/:right) $\frac{T, y \rightarrow x}{T \rightarrow (x/y)}$

(\:left) $\frac{T \rightarrow y \qquad U, x, V \rightarrow z}{U, T, (x\backslash y), V \rightarrow z}$

(\:right) $\frac{y, T \rightarrow x}{T \rightarrow (x\backslash y)}$

T non-empty sequence of categories;
U, V sequences; x, y, z categories.

Figure 2: *Cut-free and product-free LC*

$$\dfrac{\dfrac{\dfrac{\text{the president of Iceland}}{np/n,\ n,\ (n\backslash n)/np,\ np \rightarrow np}}{\dfrac{n,\ (n\backslash n)/np,\ np \rightarrow n}{np \rightarrow np}} \qquad \dfrac{n,\ n\backslash n \rightarrow n}{n \rightarrow n \quad n \rightarrow n}}{np \rightarrow np}$$

Figure 3: *Sample proof tree*

2.1 Unification Lambek Calculus

Lambek Calculus, as such, is a propositional calculus. There is no room to express additional constraints concerning the combination of categories. Clearly, some kind of feature handling mechanism is needed to enable the grammar writer to state e.g. conditions on the agreement of morpho-syntactic features or to describe control phenomena. For the reason of linguistic expressiveness and to facilitate the description of the parsing algorithm below,

we extend Lambek Calculus to Unification Lambek Calculus (ULC).

First, the definition of *basic category* must be adapted: a basic category consists of an atomic category name and feature description. (For the definition of feature descriptions or feature terms see [Smolka 1988].) For complex categories, the same recursive definition applies as before. The syntax for categories in ULC is given informally in figure 4 which shows the category of a control verb like "persuade". We assume that variable names for feature descriptions are local to each category in a sequent. The (/:left)- and (\:left)-*inference rules* have to take care of the substitutions which are involved in handling the variables in the extended categories (figure 5). Heed that the substitution function σ has scope over a whole sequent, and therefore, over a complete subproof, and not only over a single category. In this way, correct variable bindings for hypothetic categories, which are introduced by "right"-rules, are guaranteed.

```
((s([<pred>:persuade])
    <subj>:Subj
    <obj>:Obj
    <vcomp>:VComp])
  \np(Subj)
 )/(s(VComp)
    \np(Obj))
)/np(Obj)
```

Figure 4: *Sample category*

$$\frac{T \to y_2 \quad \sigma(U, x, V \to z)}{U, (x/y_1), T, V \to z} \qquad \sigma(y_1) = \sigma(y_2)$$

Figure 5: *(/:left)-rule in ULC*

$$\dfrac{\dfrac{\text{np/n, n, } (n\backslash n)/np, \text{ np} \to \text{np}}{\text{np} \to \text{np} \quad \dfrac{\text{np/n, n, } n\backslash n \to \text{np}}{\text{n} \to \text{n} \quad \dfrac{np/n, \text{ n} \to \text{np}}{\text{n} \to \text{n} \quad \text{np} \to \text{np}}}}}{\dfrac{\text{np/n, n, } (n\backslash n)/np, \text{ np} \to \text{np}}{\text{np} \to \text{np} \quad \dfrac{np/n, \text{ n, n}\backslash\text{n} \to \text{np}}{\dfrac{\text{n, } n\backslash n \to \text{n}}{\text{n} \to \text{n} \quad \text{n} \to \text{n}} \quad \text{np} \to \text{np}}}}$$

Figure 6: *Extra proofs*

3 Normal Proof Trees

The sentence in figure 3 has two other proofs, which are listed in figure 6, although one would like to contribute only one syntactic or semantic reading to it. In this section, we show that such a set of a possibly abundant number of proofs for the same reading of a sequent possesses one distinguished member which can be regarded as the representative or the *normal form* proof tree for this set.

In order to be able to use the notion of a "reading" more precisely, we undertake the following definition of structures which determine readings for our purposes. Because of their similarity to syntax trees as used with context-free grammars, we also call them "syntax trees" for the sake of simplicity. Since, on the semantic level, the use of a "left"-rule in Lambek Calculus corresponds to the functional application of a functor term to some argument and the "right"-rules are equivalent to functional abstraction [van Benthem 1986], it is essential that in a syntax tree, a trace for each of these steps in a derivation be represented. Then it is guaranteed that the semantic representation of a sentence can be constructed from a syntax tree which is annotated by the appropriate partial semantic expressions of whatever semantic representation language one chooses. Structurally distinct syntax trees amount to different semantic expressions.

A *syntax tree* t condenses the information of a proof for a sequent s in the following way:

1. Labels of *single-node trees*, are either lexical categories or arguments of lexical categories.
2. The root of a *non-trivial tree* has either
   (a) one daughter tree whose root is labelled with the value category of the root's label. This case catches the application of a "right"-inference rule; or
   (b) two daughter trees. The label of the root node is the value category, the label of the root of one daughter is the functor, and the label of the root of the other daughter is the argument category of an application of a "left"-inference rule.

Since the size of a proof for a sequent is correlated linearily to the number of operators which occur in the sequent, different proof trees for the same sequent do not differ in terms of size - they are merely structurally distinct. The task of defi-

ning those relative normal forms of proofs, which we are aiming at, amounts to describing proof trees of a certain structure which can be more easily correlated with syntax trees as would possibly be the case for other proofs of the same set of proofs.

The outline of the proof for the existence of normal form proof trees in Lambek Calculus is the following: Each proof tree of the set of proof trees for one reading of a sentence, i.e. a sequent, is mapped onto the syntax tree which represents this reading. By a proof reconstruction procedure (PR), this syntax tree can be mapped onto exactly one of the initial proof trees which will be identified as being the normal form proof tree for that set of proof trees.

It is obvious that the mapping from proof trees onto syntax trees (*Syntax Tree Construction - SC*) partitions the set of proof trees for all readings of a sentence into a finite number of disjoint subsets, i.e. equivalence classes of proof trees. Proof trees of one of these subsets share the property of having the same syntax tree, i.e. reading. Hence, the single proof tree which is reconstructed from such a syntax tree can be safely taken as a representative for the subset which it belongs to. In figure 7, this argument is restated more formally.

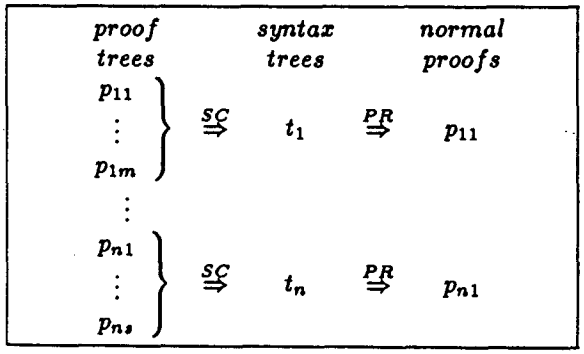

Figure 7: *Outline of the proof for normal forms*

We want to prove the following theorem:

Theorem 1 *The set of proofs for a sequent can be partitioned into equivalence classes according to their corresponding syntax trees. There is exactly one proof per equivalence class which can be identified as its normal proof.*

This theorem splits up into two lemmata, the first of which is:

Lemma 1 *For every proof tree, there exists exactly one syntax tree.*

The proof for lemma 1 consists of constructing the required syntax tree for a given proof tree.

The preparative step of the syntax tree construction procedure SC consists of augmenting lexical categories with (partial) syntax trees. Partial syntax trees are represented by λ-expressions to indicate which subtrees have to be found in order to make the tree complete. The notation for a category c paired with its (partial) syntax tree t is $c : t$.

A *basic category* is associated with the tree consisting of one node labelled with the name of the category.

Complex categories are mapped onto partial binary syntax trees represented by λ-expressions. We omit the detailed construction procedure for partial syntax trees on the lexical level, and give an example (see fig. 8) and an intuitive characterization instead. Such a partial tree has to be built up in such a way that it is a "nesting" of functional applications, i.e. one distinguished leaf is labelled with the functor category which this tree is associated with, all other leaves are labelled with variables bound by λ-operators. The list of node labels along the path from the distinguished node to the root node must show the "unfolding" of the functor category towards its head category. Such a path is dubbed *projection line*.

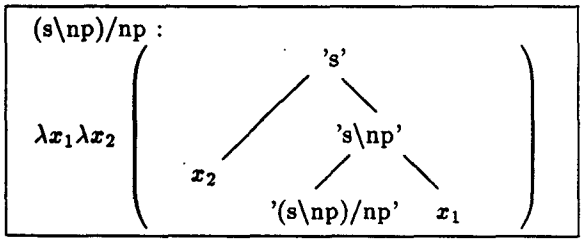

Figure 8: *Category and its partial syntax tree*

On the basis of these augmented categories, the overall syntax tree can be built up together with the proof for a sequent. As it has already been discussed above, a "left"-rule performs a functional application of a function t_f to an argument expression t_a, which we will abbreviate by $t_f[t_a]$. "right"-rules turn an expression t_v into a function (i.e. partial syntax tree) $t_f = \lambda t_a t_v$ by means of λ-abstraction over t_a. However, in order to retain the information on the category of the argument and on the direction, we use the functor category itself as the root node label instead of the afore mentioned λ-expression.

The steps for the construction of a syntax tree along with a proof are encoded as annotations of the categories in Lambek Calculus (see figure 9). An example for a result of Syntax Tree Construction is shown in figure 10 where "input" syntax trees are listed below the corresponding sequent, and "output" syntax trees are displayed above their sequents, if shown at all.

Since there is a one-to-one correspondence between proof steps and syntax tree construction steps, exactly one syntax tree is constructed per successful proof for a sequent. This leads us to the next step of the proof for the existence of normal forms, which is paraphrased by lemma 2.

Lemma 2 *From every syntax tree, a unique proof tree can be reconstructed.*

The proof for this lemma is again a constructive one: By a recursive traversal of a syntax tree, we obtain the normal form proof tree. (The formulation of the algorithm does not always properly distinguish between the nodes of a tree and the node labels.)

(axiom)	$x : t \rightarrow x : t$
(/:left)	$\dfrac{T \rightarrow y{:}t_a \qquad U, x{:}t_f[t_a], V \rightarrow z{:}t}{U, (x/y){:}t_f, T, V \rightarrow z{:}t}$
(/:right)	$\dfrac{T, y \rightarrow x{:}t}{T \rightarrow (x/y){:}'(x/y)'(t)}$
(\:left)	$\dfrac{T \rightarrow y{:}t_a \qquad U, x{:}t_f[t_a], V \rightarrow z{:}t}{U, T, (x\backslash y){:}t_f, V \rightarrow z{:}t}$
(\:right)	$\dfrac{y, T \rightarrow x{:}t}{T \rightarrow (x\backslash y){:}'(x\backslash y)'(t)}$

T non-empty sequence of categories;
U, V sequences; x, y, z categories;
t, t_a, t_f partial syntax trees.

Figure 9: *Syntax Tree Construction in LC*

Proof Reconstruction (PR)

Input: A syntax tree t with root node label g.
Output: A proof tree p whose root sequent s with antecedens A and goal category g, and whose i daughter proofs p_i $(i = 0, 1, 2)$ are determined by the following method:
Method:

- *If t consists of the single node g, p* consists of an s which is an instantiation of the *axiom scheme* with $g \rightarrow g$. s has no daughters.
- *If g is a complex category x/y resp. $x\backslash y$ and has one daughter tree t_1*, the antecedens A is the list of all leaves of t without the leftmost resp. the rightmost leaf.. s has one daughter proof which is determined by applying Proof Reconstruction to the daughter tree of g.
- *If g is a basic category and has two daughter trees t_1 and t_2*, then A is the list of all leaves of t. s has two daughter proof trees p_1 and p_2. C is the label of the leaf whose projection line ends at the root g. t_1 is the sister tree of this leaf. p_1 is obtained by applying PR to t_1. p_2 is the result of applying PR to t_2 which remains after cutting off the two subtrees C and t_1 from t.

Thus, all proofs of an equivalence class are mapped onto one single proof by a composition of the two functions Syntax Tree Construction and Proof Reconstruction. □

4 The Parser

We showed the existence of relative normal form proof trees by the detour on syntax trees, assuming that all possible proof trees have been generated beforehand. This is obviously not the way one wants to take when parsing a sentence. The goal is to construct the normal form proof directly. For this purpose, a description of the properties which distinguish normal form proofs from non-normal form proofs is required.

The essence of a proof tree is its nesting of current functors which can be regarded as a partial order on the set of current functors occuring in this specific proof tree. Since the current functors of two different rule applications might, coincidently, be the same form of category, obviously some kind of information is missing which would make all current functors of a proof tree (and hence of a syntax tree) pairwise distinct. This happens by stating which subsequence the head of the current functor spans over. As for information on a subsequence, it is sufficient to know where it starts and where it ends.

Here is the point where we make use of the expressiveness of ULC. We do not only add the start and end position information to the head of a complex category but also to its other basic subcategories, since this information will be used e.g. for making up subgoals. We make use of obvious constraints among the positional indices of subcategories of the same category. The category in figure 11 spans from position 2 to 3, its head spans from 1 to 3 if its argument category spans from 1 to 2.

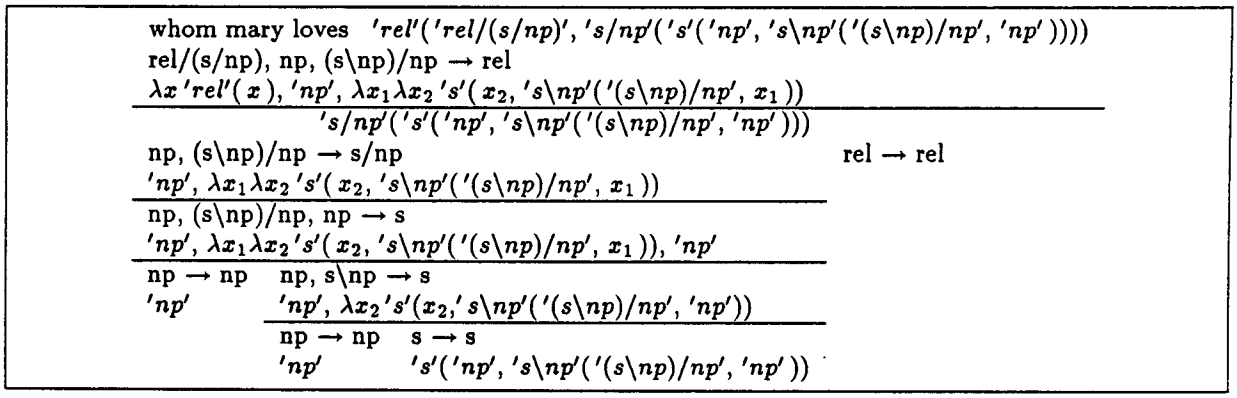

Figure 10: *Sample syntax tree construction*

The augmentation of categories by their positional indices is done most efficiently during the lexical lookup step.

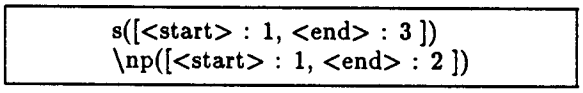

Figure 11: *Category with position features*

We can now formulate what we have learned from the Proof Reconstruction (PR) procedure. Since it works top-down on a syntax tree, the characteristics of the partial order on current functors given by their nesting in a proof tree are the following

Nesting Constraints:

1. *Right-Rule Preference*: Complex categories on the righthand side of the arrow become current functors before complex categories on the lefthand side.
2. *Current Functor Unfolding*: Once a lefthand side category is chosen for current functor it has to be "unfolded" completely, i.e. in the next inference step, its value category has to become current functor unless it is a basic category.
3. *Goal Criterium*: A lefthand side functor category can only become current functor if its head category is unifiable with the goal category of the sequent where it occurs.

Condition 3 is too weak if it is stated on the background of propositional Lambek Calculus only. It would allow for proof trees whose nesting of current functors does not coincide with the nesting of current functors in the corresponding syntax tree (see figure 12).

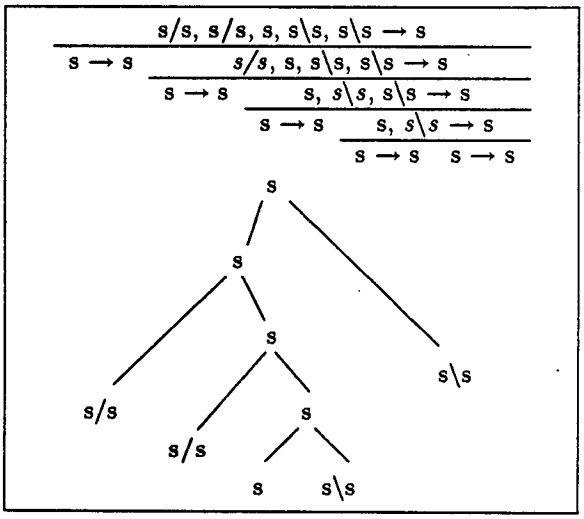

Figure 12: *Non-normal form proof*

The outline of the parsing/theorem proving algorithm P is:

- A sequent is proved if it is an instance of the *axiom scheme*.
- Otherwise, choose an *inference rule* by obeying the nesting constraints and try to prove the premises of the rule.

Algorithm P is sound with respect to LC because it has been derived from LC by adding restrictions, and not by relaxing original constraints. It is also complete with regard to LC, because the restrictions are just as many as needed to rule out proof trees of the "spurious ambiguity" kind according to theorem 1.

4.1 Further Improvements

The performance of the parser/theorem prover can be improved further by adding at least the two following ingredients:

The positional indices can help to decide where sequences in the "left"-rules have to be split up to form the appropriate subsequences of the premises.

In [van Benthem 1986], it was observed that theorems in LC possess a so-called *count invariant*, which can be used to filter out unpromising suggestions for (sub-)proofs during the inference process.

5 Conclusion

The cut-free and product-free part of Lambek Calculus has been augmented by certain constraints in order to yield only normal form proofs, i.e. only one proof per "reading" of a sentence. Thus, theorem provers for Lambek Calculus become realistic tools to be employed as parsers for categorial grammar.

General efficiency considerations would be of interest. Unconstrained Lambek Calculus seems to be absolutely inefficient, i.e. exponential. So far, no results are known as to how the use of the *nesting constraints* and the *count invariant filter* systematically affect the complexity. At least intuitively, it seems clear that their effects are drastic, because due to the former, considerably fewer proofs are generated at all, and due to the latter, substantially fewer irrelevant sub-proofs are pursued.

From a linguistic standpoint, for example, the following questions have to be discussed: How does Lambek Calculus interact with a sophisticated lexicon containing e.g. lexical rules? Which would be linguistically desirable extensions of the inference rule system that would not throw over the properties (e.g. normal form proof) of the original Lambek Calculus?

An implementation of the normal form theorem prover is currently being used for experimentation concerning these questions.

6 Acknowledgements

The research reported in this paper is supported by the LILOG project, and a doctoral fellowship, both from IBM Deutschland GmbH, and by the Esprit Basic Research Action Project 3175 (DYANA). I thank Jochen Dörre, Glyn Morrill, Remo Pareschi, and Henk Zeevat for discussion and criticism, and Fiona McKinnon for proof-reading. All errors are my own.

References

[Calder/Klein/Zeevat 1988] Calder, J.; E. Klein and H. Zeevat(1988): Unification Categorial Grammar: A Concise, Extendable Grammar for Natural Language Processing. In: Proceedings of the 12th International Conference Computational Linguistics, Budapest.

[Gallier 1986] Gallier, J.H. (1986): Logic for Computer Science. Foundations of Automatic Theorem Proving. Harper and Row, New York.

[Hepple/Morrill 1989] Hepple, M. and G. Morrill (1989): Parsing and derivational equivalence. In: Proceedings of the Association for Computational Linguistics, European Chapter, Manchester, UK.

[Lambek 1958] Lambek, J. (1958): The mathematics of sentence structure. In: Amer. Math. Monthly 65, 154-170.

[Moortgat 1988] Moortgat, M. (1988): Categorial Investigations. Logical and Linguistic Aspects of the Lambek Calculus. Foris Publications.

[Pareschi 1988] Pareschi, R. (1988): A Definite Clause Version of Categorial Grammar. In: Proc. of the 26th Annual Meeting of the Association for Computational Linguistics. Buffalo, N.Y.

[Pareschi 1989] Pareschi, R. (1989): Type-Driven Natural Language Analysis. Dissertation, University of Edinburgh.

[Pareschi/Steedman 1987] Pareschi, R. and M. Steedman (1987): A Lazy Way to Chart-Parse with Categorial Grammars. In: Proc. 25th Annual Meeting of the Association for Computational Linguistics, Stanford; 81-88.

[Pereira/Warren 1983] Pereira, F.C.N and D.H.D. Warren (1983): Parsing as Deduction. In: Proceedings of the 21st Annual Meeting of the Association of Computational Linguistics, Boston; 137-144.

[Smolka 1988] Smolka, G. (1988): A Feature Logic with Subsorts. Lilog-Report 33, IBM Deutschland GmbH, Stuttgart.

[Uszkoreit 1986] Uszkoreit, H. (1986): Categorial Unification Grammar. In: Proceedings of the 11th International Conference on Computational Linguistics, Bonn.

[van Benthem 1986] Benthem, J. v. (1986): Essays In Logical Semantics. Reidel, Dordrecht.

[Zielonka 1981] Zielonka, W. (1981): Axiomatizability of Ajdukiewicz-Lambek Calculus by Means of Cancellation Schemes. In: Zeitschrift für mathematische Logik und Grundlagen der Mathematik, 27, 215-224.