Natural Language Parsing

Methods and Formalisms

ACL/SIGPARSE WORKSHOP

Proceedings of the sixth
Twente Workshop on Language Technology

K. Sikkel & A. Nijholt (eds.)
PREFACE

TWLT is an acronym of Twente Workshop(s) on Language Technology. These workshops on natural language theory and technology are organised by Project Parlevink (sometimes with the help of others), a language theory and technology project conducted at the Department of Computer Science of the University of Twente, Enschede, The Netherlands. Each workshop has proceedings containing the papers that were presented. For the contents of these proceedings consult the last pages of this volume.

Previous workshops.

TWLT6 was devoted to natural language parsing. Just as with its predecessors, the program of the workshop consisted of presentations by a selective group of international researchers. The majority of speakers came from outside the Netherlands. The general aim was to present the state of the art in the research and development of natural language parsing. As such the following topics were present in the contributions: parsing for spelling correction, head-corner parsing and unification, the logical structure of language, practical comparisons between parallel parsing algorithms, tree adjoining grammar parsing, Definite Clause Grammar parsing, multiple agents processing, comparison of ALE and PATR, parsing ill-informed input, information from punctuation, et cetera. We regret that papers by B. Lang and M. Tomita were not available for inclusion in these proceedings.

We are grateful to the ACL (Association of Computational Linguistics) special interest group on parsing (SIGPARSE) for their support to this workshop. The workshop was also made possible by financial support provided by PTT Research.

A workshop is the result of a concerted effort of many people. Obviously, we are grateful to the authors and the organisations they represent for the efforts that made the workshop successful. Charlotte Bijron, Alice Hoogvliet-Haverkate and Joke Lammerink took care of administrative and organisational tasks. The workshop took place at the Vrijhof, the cultural meeting point of the University of Twente. We thank the participants for being there and contributing to the discussions. We hope that TWLT7, the sixth workshop in the series, with its topic "Computer-Assisted Language Learning", on June 16 and 17, 1994 will match the success of this and previous workshops.

Anton Nijholt
Klaas Sikkel

December 10, 1993
CONTENTS

Introductory Paper:

Natural Language Parsing: An Introduction
A. Nijholt (University of Twente) 1

Workshop Papers:

Typology and Logical Structure of Natural Languages
V. Manca (University of Pisa) 23

Data Oriented Parsing as a General Framework for Stochastic Language Processing
R. Bod (University of Amsterdam) 37

A Comparison of ALE and PATR: Practical Experiences
M. Stefanova & W. ter Stal (University of Sofia / University of Twente) 47

A Practical Comparison between Parallel Tabular Recognizers
J.P.M. de Vreught (University of Delft) 63

Head-Corner Parsing of Unification Grammars: A Case Study
M. Verlinden (University of Twente) 71

A Multi-Disciplinary Approach to a Parsing Algorithm
M.-J. Nederhof (University of Nijmegen) 85

Semantic-Oriented Chart Parsing with Defaults
Th. Stürmer (University of Erlangen) 99

The Parsing Problem for Tree-Adjoining Grammars
G. Satta (University of Venice) 107

A Single Formalism for a Wide Range of Parsers for DCGs
F. Barthélémy (University of Lisbon) 117

Multi-Agent Systems in Natural Language Processing
E. Ćsuhał-Varjú and R. Abo-Alež (Hungarian Academy of Sciences, Budapest) 129

Coordination as a Parsing Problem
C. Cremers (University of Leiden) 139

Bounded Incremental Parsing
M. Wirén (University of Saarbrücken) 145

Robust Parsing and Grammar Checking of Free Word Order Languages
V. Kubon and M. Plátek (Charles University, Prague) 157

Punctuation and Parsing of Real-World Texts
V. Srinivasan (University of Mainz) 163

Robust GLR Parsing for Grammar-Based Spelling Correction
T.G. Vosse (University of Leiden) 169
Sponsors and Support

We gratefully acknowledge help from:

ACL / SIGPARSE

Special Interest Group on Parsing of the ACL
(Association of Computational Linguistics)

PTT Research, Leidschendam
NATURAL LANGUAGE PARSING: AN INTRODUCTION

Anton Nijholt
University of Twente, P.O. Box 217,
7500 AE Enschede, The Netherlands
anijholt@cs.utwente.nl

ABSTRACT

This is a short introduction to the topics of this workshop. We discuss different approaches to the parsing problem, the role of syntax as a knowledge source, parallel approaches to natural language processing, competence vs. performance in parsing, non-traditional approaches to the parsing problem ( neural networks, genetic algorithms and simulated annealing) and approaches to robust analysis and interpretation.

1. INTRODUCTION

In the near future computer systems have to be accessible for users that wish to use natural language rather than any formal programming or query language and rather than any graphical menu and icon based interface. Moreover, more than now there will be applications which require the processing of (spoken) natural language. Here we think of applications where information is asked (e.g., for train and airline travels, theatre performances, etc.), reservations for tickets are made, orders are booked, etc. In addition we think of help systems that inform a user in a natural language dialogue how to use an information system, an expert system or complex machinery. In general, interfaces to such systems should allow different possibilities for access. Which (combination) will be chosen has to be decided by the user. Commands, mouse-clicked menus, pointing at the screen and the use of spoken or typed natural language are among the possibilities.

At this moment there are already acceptable commercial natural language interfaces to commercial databases. Questions have to be typed by the user. For speech the situation is different. Experiments with isolated word recognition in speech processing show increasing success. Many research projects emerge in which recognition and processing of continuous speech are considered. Until now, however, they have not lead to commercial products and it seems that this can not be expected in the near future. Looking at speech two research approaches can be distinguished. We can continue to improve existing speech technology aiming at speaker-independent and reliable continuous speech recognition. However, in this way progress is slow and it is generally assumed that, although this research should continue, it is necessary to have an approach in which speech processing is integrated with syntactic, semantic and pragmatic processing. This should yield speech recognition systems that produce input for, e.g., a natural language processing system that converts natural language utterances to SQL queries to a relational database or to an information retrieval query to a text database. In this way we can investigate how continuous speech recognition can profit from syntactic and semantic knowledge and from knowledge about the domain the natural language processing system is expected to provide information (e.g., the contents of the database that will be queried). This demands the integration of speech recognition and natural language understanding. It also demands an adaptation of existing formalisms that describe syntax and semantics. In this workshop, however, for speech we will only look at ways the connection between recognition and understanding can find its way into the formalisms. 'Robustness' will play an important role in modelling this connection and many papers presented here are concerned with 'robustness'.

Whatever access medium is considered, interfaces should be co-operative and user-friendly. Among others, this means that unexpected, unclear and incorrect input by the user should nevertheless be processed in an acceptable or informative manner. Such systems are called robust. 'Unexpected' means that rules of language use are not satisfied. The 'straightforward' functionality of the interface can not be satisfied and language utterances can go beyond

---

1 Parts of this paper have been published before. See Nijholt[1991], [1993a] & [1993b].
this 'straight-forward' functionality. In a natural language dialogue (with the help of a keyboard or by speech) this can mean that a user introduces new topics or asks questions in an incoherent way and which are beyond the range of the system. These are violations of pragmatic rules. Research in the area of language analysis for natural language interfaces to information systems can not be restricted to pragmatically well-formed language utterances. The same holds true for syntax and semantics. Users violate rules of syntactic and semantic well-formedness of sentences. Words can be (partly) absent, they can be spelled or pronounced incorrectly, spoken sentences are interrupted, etc. Therefore, in the case of automatic continuous speech recognition the result is not a proper series of recognised words, but a so-called word lattice, a graph that shows the possible words with probabilities for alternatives and word boundaries. The use of (extra-) linguistic knowledge will reduce the 'size' of a word lattice.

Solution of the problems of continuous speech recognition asks for an approach in which acoustic, linguistic and domain knowledge is integrated. With keyboard input robustness can be obtained by integrating linguistic and domain knowledge. Also, even when language utterances are well-formed, natural language analysis requires an integrated approach to make sentences comprehensible. Notice that words are ambiguous and ambiguity leads to a combinatorial explosion of possible analyses when the knowledge sources are called upon independently of each other. Well-formed sentences can contain constituents, e.g. pronouns, that refer to (parts of) other sentences in the discourse and resolution of these references - requiring all possible types of knowledge - is necessary to obtain a complete sentence understanding.

2. Syntax as a Knowledge Source

2.1 Introduction
In a well known textbook ([Allen][1987]), the different traditional forms of knowledge that are needed to understand sentences are distinguished: phonetic and phonological, morphological, syntactic, semantic, pragmatic and world knowledge. Finer distinctions have been made. In a dialogue participants have intentions behind utterances, they are planning their contribution to the dialogue. Hence, knowledge about participants and their plans is another source that has to be called on when processing sentences. In addition, texts and conversations have structure. Theory has been developed to model possible structures and to relate plans and communication goals to utterances in these structures. Another source of knowledge that has to be used.

Generally, in order to understand a sentence three phases of processing are distinguished. In the parsing phase a sentence is processed into a structural description using syntactic and morphological knowledge, in the semantic interpretation phase the structural description is mapped, using semantic knowledge, into a logical form which represents the (literal) meaning of the sentence (independent of the context), and in the contextual interpretation phase this representation is mapped into a final representation of the effects of understanding the sentence. These phases suggest a sequential and/or incremental processing of sentences and a modular and procedural approach. However, grammars for (subsets of) natural languages are - whatever formalism is used - very large and highly ambiguous. This ambiguity causes computational problems. Too many syntactic parses are constructed before semantic analysis can take place. Several approaches to 'interleaved' or 'integrated' syntactic and semantic analysis can be found in the literature. However, none of these approaches has become a standard. Semantic grammar and case frame grammar parsing have been introduced. From Artificial Intelligence (AI) knowledge based ideas have been imported. Schank's conceptual dependency parsing almost bypasses syntax. The use of frames and scripts and related knowledge representation formalisms allow the interpretation of sentences in a context that gives priority to world knowledge rather than syntactic or semantic knowledge. The obvious problem, hardly addressed in AI, is the choice of contexts of interpretation during sentence processing.

There exist a lot of formal approaches to aspects of language use and language description. There does not exist an integrated approach and the approaches that can be distinguished do not cover all facets and are often developed independently of each other. Morphology and syntax are reasonably understood phenomena. This does not mean that syntax is a computationally manageable problem. Large grammars describing a considerable part of a natural language are rare. Because of the large
number of rules they are difficult to extend and change without losing consistency. Although there are many competing grammatical formalisms, syntactic formalisms are often based on context-free formalisms with added attributes or features which enforce agreement (e.g., auxiliary-verb agreement) and complements (subcategorization of the verb, e.g., whether it allows no NPs, one object NP or an object and an indirect object NP to follow).

2.2 Decomposition, Models and Modularity
Models of sentence processing may or may not refer to a decomposition with several levels for phonology, morphology, syntax, semantics and pragmatics. Natural language processing systems can be built for quite practical reasons and therefore performance properties can be much more important than the attempt to reflect ideas available in linguistic or psychological theories. Moreover, since practical systems do not always have to deal with the full range of natural language sentence constructs - or with an unlimited domain of discourse - the 'natural' decomposition is not necessarily present in language processing systems.

From a psychological and linguistic point of view computer models of human sentence processing should be consistent with theories developed in those fields. With a model it should be possible to simulate phenomena of human sentence processing. Psycho-linguistic experiments (e.g. on the correspondence between the sequence of psychological processes and the sequence of operations whereby a grammar generates a structural description) and theoretical results (e.g., the time and space complexity of recognition and parsing) can play a role in determining the psychological plausibility of parsing models. Moreover, the design of these models can be inspired by these results. Theoretical topics that play a role are, e.g., weak and strong generative capacity, learnability, succinctness of description and parsing efficiency (Berwick & Weinberg[1982], Abney & Johnson [1991]). Experimental psychological data can tell us how lexical decisions are made, whether humans parse deterministically or whether on hearing a word all its senses are activated in the human brain. Much can be learned from deviations of ordinary language use. Being able to model this deviation, e.g. of an aphasia patient, is highly informative for a model of an ordinary language user who does not provide us with cues about ways of understanding sentences.

Human sentence processing has been explained with a serial model (Miller[1962], Fodor, Bever and Garrett[1974]). Usually, these models use a 'syntax-first' approach (see also Frazier and Fodor[1978]), where the syntactic processing task must be successful before semantic processing can begin, which in turn must precede pragmatic processing. If in this model of linear interaction between levels of knowledge, higher level information cannot be used to correct decisions at lower levels, this approach leads to the already mentioned combinatorial explosion of all legal syntactic possibilities. For that reason especially syntax and semantics often closely interact in models of sentence processing, leading to interleaved or integrated syntactic-semantic processing.

Choosing a parser for a natural language processing system is less simple and more important than often is assumed. A first choice is that between a syntactically and semantically based grammar and parser. In Wittenburg[1986] considerations on making this choice are mentioned. Semantically based grammars offer customization of the system for the domain and they offer robustness. On the other hand, syntactically based grammars are more general, can be used for many different domains and offer modularity in design. In the MCC Lingo project described by Wittenburg a portable natural language interface had to be designed. A graph-unification-based formalism for the representation of the grammar was chosen, while the grammar approach was that of Combinatory Categorial Grammar. Agenda driven chart parsing (see also section 4) with its possibility to control and restrict the enumeration of parses using heuristics, statistical information and semantics (domain knowledge) in maximally flexible ways turned out to be the natural choice for a parsing algorithm in this project.

When syntactic and semantic processing takes place simultaneously, parsing is said to be done in an integrated way. A simple example is parsing according to a semantic grammar, where syntactic categories are replaced by semantic categories in the grammar rules. More sophisticated examples are conceptual dependency and word expert parsers. Parsing rules operate with both syntactic and semantic information. Syntactic and semantic knowledge are compiled together in one set of
rules and no separate representation of syntactic structure is built. In Lytinen[1986] it is argued that rather than statically integrating syntactic and semantic knowledge in the parsing rules they should be combined dynamically at parse time, that is, in his view, semantic knowledge should determine the application of syntactic rules. A further exploration of this view can be found in Lytinen[1991]. Obviously, rather than talking about parsing strategies and architectures in general, it is often more important to consider an approach in relation to a domain of application. For example, ATRANS (Automatic Funds TRANSfer Telex-Reader) is a system that processes money transfer messages sent by banks (Lytinen & Gershman[1986]). These messages are written by people who don't care about spelling and syntax as long as their message is clear for human interpreters. In such a domain semantically oriented parsers, such as conceptual dependency parsers, will certainly yield better results than parsers based on correct English grammar rules.

In Tomita and Carbonell[1987] a 'universal parser architecture' for knowledge based machine translation is presented. It assumes syntactic grammars for different languages and semantic knowledge bases for different domains. A grammar compiler integrates a syntactic grammar, morphological rules and a domain knowledge base into a so-called 'syn/sem grammar' in PATR-like notation. This grammar is further compiled into an augmented context-free grammar that can be given to an LR table generator for a generalized LR parsing algorithm. In Fig. 1 the architecture of the system is presented.

The need for semantic or discourse context restriction checks during syntactic analysis can always clearly be illustrated by example sentences and analyses from the literature. A corpus considered by Martin, Church & Patil[1987] contained the sentence In as much as allocating costs is a tough job I would like to have the total costs related to each product and with their syntactically based grammar they found 958 parses for this sentence. No satisfactory performance can be expected with an exhaustive enumeration of these parses. The need for integrating thematic and contextual knowledge during parsing is also supported by similar examples in the context of speech understanding. Tomabechi & Tomita[1988] present an integrated parser (using concurrent processing of syntax and semantics during parsing) which nevertheless provides the spoken Japanese input atamagaitai (I have a headache) with 57 interpretations.

2.3 Towards Parallel Processing
In an influential paper Marslen-Wilson[1975] reported about experiments that gave psychological evidence to the idea that processing at each level of natural language description can
constrain and guide simultaneous processing at other levels. That is:

... evidence that sentence perception is most plausibly modelled as a fully interactive parallel process: that each word, as it is heard in the context of normal discourse, is immediately entered into the processing system at all levels of description, and is simultaneously analysed at all these levels in the light of whatever information is available at each level at that point in the processing of the sentence.

Models in which this type of sentence processing can be displayed are called interactive (parallel) sentence processing models. During parsing, a system based on this model is capable of using any type of knowledge at any moment it needs. These models may have different appearances. They can take the form of a hierarchical system in which natural language processing tasks are assigned to different processors or processes and in which every knowledge source interacts with every other. A proposal for such a system can already be found in Winograd[1972]. For example, using the same knowledge sources as before, for each source we can define one process (cf. Fig. 2).

Obviously, this may lead to a complicated model in which the knowledge sources must know how to communicate and interact with every other. In the blackboard model of knowledge source interaction (cf. Fig. 3), the multiple sources can process in parallel and co-operatively by means of a globally accessible 'blackboard' on which they can write and read intermediate results. The knowledge sources communicate and interact solely through the blackboard.

In the models described so far only a modest number of different processes (roughly, one for each knowledge source) are distinguished. Parallelism is task-oriented. It is possible to consider parallelism on a less global level and

![Fig. 2 A hierarchical parser.](image)

![Fig. 3 The blackboard model.](image)

more in agreement with the view expressed by Marslen-Wilson. This is done by considering words as active agents that interact with each other and possibly with other knowledge sources in order to obtain a meaning representation of a sentence. An even more fine-grained level of parallelism is the connectionist or neural network (NN) approach. In a huge network of extremely simple processing elements language processing is 'coded' into spreading of activation and converging of activation towards a pattern that somehow represents the meaning of a sentence. In the following sections examples of different types of parallelism in natural language processing (NLP) can be found.

3. MODULAR APPROACHES TO NLP

Generally, parsing is the process in which an input text is mapped into some internal representation. This representation may take the form of a syntactic parse tree augmented with semantic restrictions, a case frame or conceptual dependency representation, or some other semantic network structure. In obtaining this representation, knowledge from different levels of the hierarchy of knowledge sources has to be invoked. It has to be investigated how this can be done. A serial model does not necessarily imply that, e.g., a lexical analysis can only be done once the phonetic analysis is finished or that a syntactic analysis should strictly follow the lexical analysis. It is quite possible that either one of many possible analyses on one level is sent to a next level before obtaining the other analyses, or that a primary and partial analysis is sent to a next level of analysis. Hence, in a computer model this 'serial' processing may require various processors performing tasks in parallel, despite the fact that the information flow is serial. This so-called 'pipeline' processing can be extended to include feedback from a processor to a processor working at a previous
level. A possible candidate formalism for pipeline processing is the cascaded ATN of Woods. A cascaded Augmented Transition Network (Woods[1980]) consists of a sequence of cooperating ATN's with separate domains of responsibility. Each ATN in the sequence takes its input from the output of the previous ATN. Elements of an output sequence of an ATN are generated by executing output operations on the arcs of the ATN. In Woods' view, a cascade of ATN's can correspond with a 'serial' decomposition of natural language description into levels of phonology, lexicon, syntax, semantics, and pragmatics.

The cascaded ATN can be considered as a generalisation of the RUS Parser (Bobrow[1978]) which had a 2-stage cascade. Although the formalism allows pipeline processing, in Woods' view it will not be used that way. For example, as soon as a (partial) phrase is derived that allows useful semantic processing, then the syntactic ATN is interrupted and control is passed over to the semantic component. Feed-back given by this component may cause the syntactic component to continue from where it was interrupted or to backtrack when by semantic or higher level processing a wrong path of analysis is detected. Due to this filtering role of higher-level processes it is not useful to continue processing at a lower level before information is passed down from a higher level. As an example of a cascade-like formalism we mention the multilevel parsing formalism which can be found in Gehrke[1983]. The cascade consists of a syntactic and a semantic component and a few stages concerned with pragmatics. Since the formalism is part of a task-oriented dialogue system the pragmatic components deal with knowledge of the world, the dialogue partners and the course of the dialogue. Also in this case of CATN use no parallelism is foreseen.

The main example of a blackboard system in natural language processing is the Hearsay-II speech understanding system from the early 1970s. In multiprocessor simulation experiments the suitability for parallel processing of the blackboard system was shown. In Valkonen et al.[1987] a blackboard model for dependency parsing is presented. A scheduler controls the knowledge sources (functional schemata and binary dependency relations) and the linking of partial dependency trees into a parse tree. Dependency syntax in combination with a blackboard to be used by cooperating knowledge sources has also been matter of research in the Speech Understanding System SUSy: cf. Possio&Rollent[1987]. In this system top-down expectations and bottom-up predictions are integrated into a case-frame instantiating and filling approach to the parsing problem.

A limited example of a heterarchical parser is a system proposed by Huang and Guthrie[1985]. It consists of four semantic and two syntactic processes (see Fig. 4). In each process tasks are allocated to different processors (from an Intel Hypercube multiprocessor system). The syntactic processes - one for constructing a Sentence, one for constructing an NP - are therefore able to consider syntactic alternatives in parallel. The semantic processes are concerned with tasks as finding meaningful adjective-noun word sense pairs (the AN process), subject-verb word sense pairs (the SV process) and verb-object word sense pairs (the VO process). These processes are consulted by the syntactic processes. When a syntactic process returns several meaningful pairs of senses parsing continues for each of the possibilities in parallel. A parse may be blocked if no meaningful pair is returned.

Huang and Guthrie's approach is meant to be implemented in a logic programming language. Especially object- or actor-oriented approaches to the parsing problem often deal with some form of integrated parsing. In many cases these approaches are well-suited for designing parallel implementations. Phillips[1984] describes a left-corner parsing algorithm with a pseudo-parallel and breadth-first control structure which controls transmitting parsed constituents to a semantic interpretation component. The latter eliminates competing syntactic constituents. The object-oriented approach, with objects associated with grammar rules, simplifies a possible parallel implementation. A slightly more complete integrated parser (implemented in Prolog) is a
system of Uehara et al.[1985] based on Lexical Functional Grammar (LFG). Producing the f-structure of a sentence in an LFG is viewed as message passing between actors, where actors correspond with grammar rules or with lexical items.

In the discussion so far, parallelism corresponded to different knowledge sources, syntactic categories or grammar rules. Another approach, initiated by S.L. Small, is to consider parsing as the interaction of relatively small distributed knowledge components associated with words. In order to take into consideration the inherent parallelism required for sentence comprehension, Small introduced a theory of word expert parsing (WEP). When reading a sentence, each word triggers a 'word expert', a program attached to a word. This word expert starts processing and by interacting with other word experts and some higher-order processes it tries to determine the intended meaning of the word in its context. Hence, it will ask questions to other word experts and it is ready to answer questions posed by other word experts. The interactions are performed in a coroutine fashion: when information is asked execution is suspended until the information becomes available. By mutual agreement the word experts decide on the meaning of (a fragment of) the sentence. The potential parallelism in this approach has been explored by Adriaans[1989]. Rather than having a coroutine control, in Adriarc's Parallel Expert Parser (PEP) the parallel interaction of experts is stressed. The interaction takes place through a structured blackboard. Instead of having each expert associated with a word, in the parallel approach experts are associated with concepts from different levels (e.g., word level, constituent or clause level).

4. MODULARITY AND THE SYNTAX LEVEL OF DESCRIPTION

4.1 Introduction

In the previous sections attention was focused on language processing and parallelism emerged due to assigning tasks involving the application of different types of knowledge to processors. It is also possible to consider one level of sentence perception and investigate how a parallel approach 'inside' that particular level can be given and see whether it can be integrated in an overall parallel approach. Much effort has been spent to obtain efficient (serial) methods of syntactic analysis of sentences with respect to context-free grammars. Hence, rather than viewing parsing as the process in which an input text is mapped into an internal representation in which constituent relations are presented (as, e.g., in case-role representations of sentences), now parsing amounts to syntactic analysis and the result is a syntactic tree representation of a sentence. The question arises whether the existing methods can be adapted to a parallel processing view. The answer to this question is affirmative. General context-free language parsing according to the Cocke-Younger-Kasami (CYK), the Earley and the Tomita algorithm can be given various parallel implementations. Moreover, due to the advent of parallel logic and parallel object-oriented programming languages, parallel logic and object-oriented approaches to the parsing problem have been designed in rather natural ways. The parsing schemes that have been introduced include schemes which use more than one traditional syntactic parser, schemes where separate processes are devoted to 'non-deterministic' choices during parsing, schemes where the number of processes depends on the length of the sentence being parsed and schemes where the number of processes depends on the grammar size. An overview of the various algorithms can be found in Nijholt[1990].

4.2 Processor Configurations

Designing parsing methods from the point of view of available or desired processor configurations has led to a variety of methods. Attacking the context-free parsing problem with more than one processor almost always means using identical processors, that is, processors that run the same software and compute the same function. In some cases this can amount to having several asynchronous parsers working on the same input string (e.g., Fischer[1975] and Lozinskii and Nirenburg[1986]). Approaches involving multi-processor shared memory computers, pipelines of processors and arrays of processors can be illustrated with the CYK algorithm.

For convenience, assume that grammars are in Chomsky Normal Form (CNF), hence, each rule is of the form \( A \rightarrow BC \) or \( A \rightarrow a \). For any input string \( x = a_1 \ldots a_n \) an upper-triangular \( (n+1) \times (n+1) \) table \( T \) will be constructed. Initially, table entries \( t_{ij} \) with \( i < j \) are empty. Assume that the input string, if desired terminated with an end
marker, is available on the matrix diagonal (cf. Fig. 5).

(1) Compute \( t_{ij+1} \) as \( i \) ranges from 0 to \( n-1 \), by placing \( A \) in \( t_{ij+1} \) exactly when there is a production \( A \rightarrow a_{i+1} \) in \( P \) (the set of productions).

(2) In order to compute \( t_{ij} \), if \( j > i \), assume that all entries \( t_{kj} \) with \( k \leq j \), \( k \neq i \) and \( k \neq j \) have been computed. Add \( A \) to \( t_{ij} \) if, for any \( k \) such that \( i < k < j \), \( B \in t_{ik} \), \( C \in t_{kj} \), and \( A \rightarrow BC \) is a production rule and \( A \) is not already present in \( t_{ij} \).

(3) String \( x \) belongs to the language if and only if \( S \) is in \( t_{0,n} \).

It can be shown that this algorithm requires \( O(n^3) \) time.

\[
\begin{array}{|c|c|c|c|c|}
\hline
a_1 & t_{01} & t_{02} & t_{03} & t_{04} & t_{05} \\
\hline
a_2 & t_{12} & t_{13} & t_{14} & t_{15} & \\
\hline
a_3 & t_{23} & t_{24} & t_{25} & \\
\hline
a_4 & t_{34} & t_{35} & \\
\hline
a_5 & t_{45} & \\
\hline
S & & & & & \\
\hline
\end{array}
\]

Fig. 5 The upper-triangular CYK-table.

The algorithm is presented in such a way that computation can proceed column by column, but also diagonal by diagonal. A parallel implementation can be given with an array of processors, as depicted in Fig. 6. For each entry \( t_{ij} \) of the strictly upper-triangular table there is a processor \( P_{ij} \) which receives table entries (i.e., sets of nonterminals) from processors \( P_{i+1,j} \) and \( P_{i,j+1} \). Processor \( P_{ij} \) transmits the table entries it receives from \( P_{i+1,j} \) to \( P_{i,j+1} \) and the entries it receives from \( P_{i+1,j} \) to \( P_{i,j+1} \). Processor \( P_{ij} \) transmits the table entry it has constructed to processors \( P_{i+1,j} \) and \( P_{i,j+1} \). Fig. 6 shows the interconnection structure for \( n=5 \).

![Fig. 6 Processor configuration for CYK.](image)

Moreover, a communication scheme can be given such that each processor has to store only a fixed number of table entries in order to make the correct matches \( t_{ik} \) and \( t_{kj} \) for computing \( t_{ij} \).

In order to obtain other implementations it is necessary to return to the original sequential algorithm. Its efficiency is obtained by introducing a detailed data structure (a 2-dimensional array) in which we store nonterminals at certain positions. Let us call a table entry an item set and its contents items. Instead of a 2-dimensional array we can use a 1-dimensional array of item sets, one for each word of the sentence. However, then the items have to be more complicated. Now they are of the form \( f(i,A)_j \), where \( i \) is a position marker. The algorithm now takes the following form. For any input string \( x = a_1 \ldots a_n \), item sets \( I_1, \ldots, I_n \) will be constructed.

1. \( I_1 = \{0,A \mid A \rightarrow a_i \text{ is in } P \} \).
2. Having constructed \( I_1, \ldots, I_{j-1} \), construct \( I_j \) in the following way. Initially, \( I_j = \emptyset \).
2.1 Add \( \{j,A_1 \} \) to \( I_j \) for any production \( A \rightarrow a_j \) in \( P \).

Now perform step (2.2) until no new items will be added to \( I_j \).

2.2 Let \( \{k,C \} \) be an item in \( I_j \). For each item of the form \( \{i,B \} \) in \( I_k \) such that there is a production \( A \rightarrow BC \) in \( P \) add \( \{i,A \} \) (provided it has not been added before) to \( I_j \).
A string has been recognised as soon as $[0,S]$ appears in the item set. Clearly, more than in the 2-dimensional version, the algorithm has a left-to-right nature. Nevertheless, due to the fact that during construction of an item set the items that have been constructed can be sent to their right neighbours, we can obtain a pipeline of parallel processors (cf. Fig. 7).

![Fig. 7 Pipeline for CYK parsing.](image)

A communication scheme can be given such that each processor requires $O(n)$ memory and total parsing time is $O(n^2)$. Indeed, the methods become less efficient (at least theoretically), but we obtain more freedom, in the sense that different configurations and numbers of processors can be used. As a next step, let us forget clever data structures and consider a 'flat' blackboard-like structure consisting of an agenda and a work area. The items will be of the form $[i,A,j]$, where $i$ and $j$ are again position markers. The input is $a_1 a_2 \ldots a_n$ and initially agenda and work area are empty.

(1) Initially, items $[i-1,A,i+1]$, $1 \leq i \leq n$ with $A \rightarrow a_i$ in $P$ are added to the agenda.

Now keep repeating the following step until the agenda is empty.

(2) Remove any member of the agenda and put it in the work area. Assume the chosen item is of the form $[j,Y,j]$. For any $[i,X,i]$ in the work area such that $A \rightarrow a_i$ in $P$ add $[i,A,j]$ (provided it has not been added before) to the agenda.

If $[0,S,n]$ appears in the work area, then the input string has been recognised.

In the algorithm no specific order is prescribed for removing items from the agenda. One of several strategies has to be chosen. The algorithm is extremely inefficient and from a theoretical point of view this does not change if we add more processors. In practice, however, since the size of the grammar has a stronger influence on the efficiency than the length of the sentence, it will matter. Hence, if we have a set of processors at our disposal, then each processor can execute loop (2) of the algorithm. That is, we have a number of asynchronous processors and each processor picks up an item from the agenda, puts it on the list of employed items on the work area, cycles through this list and, after having verified that they have not been entered before, puts newly created items on the agenda. When it has finished the list it can pick up a new item from the agenda and start anew.

This implementation resembles one discussed in Grishman and Chitrao[1988] for a shared memory computer (the NYU Ultra-computer, a MIMD parallel processor) and one discussed in Thompson[1989] for the BBN Butterfly shared memory computer. The Grishman and Chitrao parser is intended to become part of a natural language processing system which uses augmented context-free grammars - context-free grammars augmented with procedural restrictions which enforce syntactic and semantic constraints. A reason to prefer asynchronous processors is that checking conditions on rules may vary widely in computation time. An other reason is that a parse table is large and typically sparse and no efficient use of processors can be made.

Our aim has been to show that a traditional serial parsing method can be given different (parallel) implementations. What we have shown for the CYK method can also be shown for the Earley method. In order to obtain similar implementations for the Earley method it is necessary to throw away the inherently sequential prediction operation that is used for 'top-down filtering'. This operation improves efficiency for the serial version, but it introduces a left-to-right order which prevents some parallel implementations. Since the CYK and Earley methods can be viewed as more formal descriptions of the chart parsing methods, everything that has been discussed here applies to these methods.

It should be noted that although we discussed parallel implementations of sequential algorithms, there remains the freedom to access them in non-parallel ways. For example, in Fig. 8 it is quite possible to use one processor and with it profit from the many possible flexible ways and strategies items can be combined into parses.
in parallel). These languages invite designers and programmers to take a different view on the (syntactic) language processing problem and to introduce new and interesting methods. Parallel logic programming languages such as GHC (Guarded Horn Clauses), PARLOG and Concurrent PROLOG have been used to build natural language processing systems. In Matsumoto[1987] an example of the logic approach can be found. In Tanaka and Numazaki[1989] GHC is used to implement a parallel version of Tomita’s generalised LR parsing algorithm. In the papers of Matsumoto the terminal and nonterminal symbols are defined as parallel processes. In the paper of Tanaka and Numazaki each LR-table entry is called a process.

Similarly, a parallel object-oriented point of view can be advocated, as is done by Yonezawa and Ohsawa[1988]. They use the parallel object-oriented programming language ABC1/1 to implement a parsing system which is obtained by translating a collection of context-free grammar rules into a configuration of message-passing, cooperating units (agents). Each occurrence of a terminal or nonterminal symbol in a grammar rule is represented as an agent in the system, the messages that are sent consist of control data or partial subtrees.

5. FROM GENERAL COMPETENCE TO DOMAIN PERFORMANCE

5.1 No Comprehensive Computational Models
A returning discussion, which we only mention now, is the difference between a holistic and a reductionist view of language. In the holistic view language processes should be described as cognitive strategies employed by the brain. Knowledge used by these strategies is shared with other strategies, e.g. for vision. A model of a phenomenon of the brain, such as language, is achieved from a model of the brain as a system as a whole. In the reductionist view the individual phenomena are modelled and there is not necessarily an account of the relation between phenomena in a unified whole. In the connectionist literature ‘holistic’ is often used to express the interaction between all possible types of knowledge in solving problems.

Whatever viewpoint, lack of convincing results suggest that computational models of real natural language and its use will not be available
in the near future. For that reason researchers in NLP started looking for applications that allow restricted language use. For a natural language interface to a database containing information about flight times there is no need to be able to conduct a dialogue with a user about sports events. The domain provides the pre-specified context in which questions, remarks and commands can be interpreted. Examples of commercial NLP systems exist which indeed owe their success to a well-chosen domain of application.

The lack of success led to criticism on the rule-based approach towards natural language. It is assumed that the huge amount of linguistic data and the necessary interaction between knowledge sources during analysis do not allow a manageable exhaustive description. Conclusions that have been drawn differ. It has led to discussions about a possible evolutionary account of language universals, about the necessity of being able to decide between grammatical and non-grammatical strings, about the necessity of complete syntactic trees, about the use of statistical methods, and, starting in the 1980s, the use of NNs and genetic algorithms in language analysis. The question has also been raised whether not more attention should be paid to grammars that describe performance rather than competence. Hence, there have been proposals and arguments for describing natural languages with FSA since recursion in language use is always highly restricted.

5.2 Regular versus Context-Free/Context-Sensitive: Performance versus Competence?
There have been hot discussions about the relationships between the generating power of the grammars from the Chomsky hierarchy and natural languages. Examples of language constructs from sometimes well known, but more often less well known or exotic languages have been used to make statements about the non-context-freeness of natural languages. Proofs have been criticised because the arguments involved not only syntactic, but also semantic constraints.

Approaching the question from another end, it has been shown that several linguistically motivated formalisms all define the same class of so-called ‘mildly context-sensitive’ languages (Joshi et al. [1991]). These formalisms include the combinatory categorical grammars, the head grammars and the tree adjoining grammars.

Approaching the question from a 'performance' viewpoint, it has been noticed that ordinary language users (non-linguists) never need unbounded recursion (embedding) or sentences of unbounded length. See e.g. Miller [1962], Suppes [1972], Reich & Well [1977], De Roeck et al. [1982], Koene [1985], Schütze & Reich [1990] and Hasida [1990]. For example, Suppes remarks that:

"From the standpoint of empirical application, one of the more dissatisfying aspects of the purely formal theory of grammars is that no distinction is made between utterances of ordinary length and utterances that are arbitrary long, for example, of more than 10^50 words. One of the most obvious and fundamental features of actual spoken speech or written text is the distribution of length of utterances, and the relative sharp bounds on the complexity of utterances, because of the highly restricted use of embedding or other recursive devices."

while Schütze and Reich notice that:

"While it is still a point of contention, recent carefully controlled experiments suggest that the human syntactic mechanism, without semantic or pragmatic cues, and without the aid of pencil and paper to construct sentences, does have a sharp level of one or two levels of embedding."

And, as has been remarked by Rumelhart & McClelland in their PDP books:

"As we have already seen, one can make an arbitrary computational machine out of linear threshold units, including, for example, a machine that can carry out all the operations necessary for implementing a Turing machine; the one limitation is that real biological systems cannot be Turing machines because they have finite hardware. ... We have not dwelt on PDP implementations of Turing machines and recursive processing engines because we do not agree with those who would argue that such capabilities are of the essence of human computation."

The emerge of NNs in the area of NLP has given a new impetus to this discussion. One reason is that a NN has a fixed size, and that in several approaches to NN processing of natural language this size determines the maximum of sentence length that can be handled. A second reason is that
5.3 Probabilistic Parsing Approaches
In 1949 science advisor Warren Weaver, impressed by Shannon's Mathematical Theory of Communication, suggested to use a statistical (information-theory-like) approach to the description of language ("entropy speaks the language of language") and even to language translation. However, Chomsky's cognitive approach in the early fifties was taken more seriously. The last five years have seen a growing interest in statistical methods and corpus-based linguistic research. A corpus can be used to provide examples and test material, but also to provide a basis of a statistical model of language. The decrease in interest in these models after the late 1950s has been explained by pointing at the 'anti-empiricist, anti-numerical, pro-symbolic trend in the Zeitgeist' during these years (Liberman[1991]). However, in the 1990s we see statistical approaches not only to speech processing, but also to tasks as message parsing, machine translation or even the translation of English specifications (prescribing the behaviour of a telephone switching system) into an automated testing language. Liberma argues that enormous corpora will be needed to obtain statistical models, since rather than modelling the regularities of speech and language, in this approach we are modelling regularities of the world.

Simple surface properties, such as which word will probably follow another word or will follow a sequence of words, can be described using statistical information (nth order Markov chains), although useful for some tasks, this information does not help in finding the most probable constituent structure of a sentence. Structure is described with the rules of the grammar. Hence, if we confine ourselves to syntax, then rather than designing formal grammars that are based on our intuitions about correct sentences, grammars can be based on a corpus of sentences that once have been uttered or written down. The corpus is finite and it allows the distribution of probabilities to grammar rules or parse actions (e.g., shift and reduce actions in a shift-reduce parser). This leads to the introduction of stochastic grammars and parsers. The statistical information can guide the parsing process and reduces the number of syntactic trees that have to be considered. Clearly, the parsing process can be guided to a wrong solution. The hope is that the resulting grammar or parser behaves well on sentences that are not present in the corpus. Assigning and inferring probabilities for rules of context-free grammars according to this idea can be found in, e.g., Fu & Booth[1975] and Ophoff [1992]. In criticism on the stochastic grammar approach the localness and context-independence of assigning probabilities to rules is emphasized. A general problem is also the question whether a corpus is representative enough to have a high enough level of statistical significance. Clearly, the same problem arises with corpus-based NN learning of NLP tasks.

6. SUBSYMBOLIC, BIO-SYNTHETIC AND GENETIC PARALLELISM: PARSING AND LEARNING

6.1 Connectionist Parsers
Connectionist or neural network models of human language processing allow massive parallelism, distributed representation of knowledge and low degradation of overall behaviour in the face of local errors. Usually, in the connectionist approach we have a network of simple processing elements which function independently and in parallel. There is no central controller. The nodes of the network (i.e., the processing elements) have an activation level which is recomputed iteratively. The links between the nodes are weighted. Each node computes an output which is a function of its weighted inputs and the current activation level of the node. Initial activation of nodes in an input layer may spread to a pattern of activity at output nodes or may lead to the convergence of the network through cycles of spreading activation to a pattern of activity at a configuration of nodes which represents a solution of the problem for which the network has been built. Hence, in the case of language processing this pattern of activity on a cluster of nodes represents the meaning of a sentence or the meaning of a text processed so far. Learning takes place through changes in the weights of the connections. This is done during a learning phase and according to a learning rule. It is possible to choose between different network paradigms and learning rules, for example, backpropagation. There does not exist a generally accepted type of network for sentence processing. Therefore existing approaches are rather divers. It should be mentioned that a massively parallel system can be part of a larger system for which it performs a specialized task. Obviously, even if a process is
sequential of nature, it may be possible to distinguish subtasks which allow a (massively) parallel processing approach. For example, in online parsing words are recognized from left to right. Nevertheless, in mapping the sequence of words into a meaning representation many subtasks can be performed in parallel.

In the case of connectionist context-free language parsing, the activity on a cluster of nodes should represent the parse tree for a sentence that has been parsed. Connectionist context-free parsing methods can be found in papers by Fanty[1985] (a straightforward and simple connectionist implementation of the CYK method), Selman and Hirst[1987] (Boltzmann machine parsing, i.e., parsing which amounts to computation according to a parallel stochastic relaxation scheme using simulated annealing), Howells[1988] (a relaxation algorithm which utilizes decay over time, together with a competition for available activation), Charniak and Santos[1987] (not really connectionist, but ideas are borrowed from the connectionist approach) and Nakagawa and Morl[1988] (a parallel left-corner parser incorporated in a learning network). One aim of the latter authors is to show that some phenomena of human parsing can be explained with a connectionist parser. Considered are the understanding of 'garden-path' sentences, the time required to recognize deeply nested sentences and the explanation of 'parsing preferences' in syntactically ambiguous sentences. Similar to the Fanty translation of the CYK method it is possible to translate the Earley or the Rytter parsing algorithm to connectionist methods (cf. Sikkel and Nijholt[1990]). With these methods it can be shown that syntactic parsing can be done in $O(n)$ (CYK and Earley) or even $O(\log n)$ (Rytter) time, where $n$ is the length of the sentence. Although there are some approaches in designing connectionist context-free parsing systems which come close to marker-passing systems, most of the systems are value-passing systems as described above.

One theme which emerges in these papers is the finite size of networks. A parse tree is represented as a network of activated and connected nodes. Building a network in advance means that the length of a sentence that can be parsed is limited by the size of the network. In order to get round this problem some authors allow the building of the network during parsing. A second problem that emerges is, that although the above mentioned connectionist approaches are interesting, it is not at all clear which approach should be chosen. None of them gives the impression that a connectionist approach is 'natural' for context-free language parsing. Similarly, it seems rather unnatural to implement in neural networks processing methods which are based on grammar formalisms like unification and lexical-functional grammar. See, however, Slack[1984].

More natural connectionist approaches become visible if the traditional methods of language analysis are replaced by strongly interactive distributed processing of word senses, case roles and semantic markers (see e.g. Waltz and Pollack[1985], McClelland and Kawamoto[1986], Pollack[1987] and Cottrell[1989]). As an example, in the Waltz and Pollack paper syntactic, semantic and pragmatic information is represented in the same network and used to obtain a case-frame like representation of a sentence in the neural network. It is also in this area that connectionism is used to explain human parsing mechanisms for natural language sentences.

Themes that emerge here deal with knowledge representation and learning issues. In a localist representation each concept of the problem domain resides in a particular processing element. Simple, but when a processing element fails or gets lost, than the corresponding knowledge is also lost. With a distributed representation a concept is represented by some pattern of activation on a large number of processing elements. This approach offers more flexibility when processing knowledge which is incomplete or uncertain and when learning from examples. In a localist view the nodes in the network represent words (or morphemes) from the lexicon, syntactic categories, word senses, cases (Agent, Object, Destination, etc.) and context settings. In a distributed view the meaning of, e.g., a word is distributed among many nodes. Each aspect of the meaning has its own node. Similarly, a case role can be viewed as a set of case role descriptors, where each descriptor requires one node. Distributed representations can be set by hand (see, e.g., the microfeature-based representations in Waltz and Pollack[1985] and McClelland and Kawamoto[1986]) or they can be obtained with an automatic learning procedure (cf. Hinton[1986], Miikkulainen and Dyer[1988] and Miikkulainen[1990]). The near impossibility to encode distributed representations by hand often
leads to the conclusion that the representations must be learned from examples. In this way
distributed representations encode statistical regularity in the training data of the network.
Learning can start from a tabula rasa (an initially random network) or from a network that is
'genetically prewired'.

In Miikkulainen's paper word meanings are encoded in a distributed way and the
representations are learned automatically with a backpropagation method. More specifically, in
this paper a parallel distributed processing (PDP) architecture is presented for parsing, representing
and paraphrasing sentences with relative clauses. The system architecture consists of four
subnetworks: an act parser, a sentence parser, a sentence generator and an act generator. The act
parser produces a (possibly incomplete) case-role representation for each act fragment of a sentence
with relative clauses, the sentence parser produces the complete case-role representations of the act
fragments (including referents of 'who'-pronouns). The subnetworks are three-layer backpropagation
networks. The words that may appear in sentences are stored in a distributed form in a lexicon. The
distributed word representations are obtained during the learning phase of the backpropagation
networks. The representations adapt to the regularity in the use of words in the training
examples. Therefore, 'similar' words obtain similar representations. In the performance phase, for
each word that is read its distributed representation is fed into the act parser network. In
this network expectations about possible act representations are formed until they are narrowed
down to one possible representation. Since an act fragment may be interrupted by a relative clause,
incomplete case-role representations may be passed to the sentence parser. The latter combines
the incomplete case-role representations and determines the referents of the who pronouns in
the sentence. For training and testing some sentence templates were used (e.g., The woman,
who helped the girl, blamed the man, three verbs were used (help, hit and blame) and each verb
could have only specified nouns as its agent and patient (no other case roles were considered).

Also in more recent research we see attempts of learning language or aspects of language with
neural networks rather than a direct implementation of parsing methods in neural nets. The networks learn (mostly with backpropagation)
to map sentences into a syntactic/semantic
representation. See e.g. for Jain[1991] for robust spoken language parsing with a modular recurrent
BP net. An extension of the parser has been used in a speech-to-speech translation system (a
conference registration dialog task). Recent research on connectionist natural language
processing is reported in Drossaers & Nijholt[1992]. It should be mentioned that presently there is more interest in
fundamental properties and problems that underlie natural language processing and that are also of
interest for other applications. Recognition of sequences of patterns is such a problem.
Backpropagation networks, Kohonen's self-
organising feature maps and Hopfield networks have been adapted and extended in order to
become able to recognize sequences.

Marker-passing and spreading-activation
models of language processing have been intro-
duced which differ from the usual neural network
models. There is no clear-cut distinction between
these models. All offer the advantages of parallel
processing. Usually, but not always, in marker-
passing nodes are either active or not and only
discrete markers are passed. In spreading activa-
tion models nodes have degrees of activation. An
example of a spreading-activation model is
ATLAST (Granger et al.[1984]) in which spreading activation models an inference making
process employing knowledge from different
levels of language description.

6.2 Annealing Parsers
Parsing can be viewed as searching the space of
all possible parse trees for a tree that fits the
current input sentence. Simulated annealing can
be viewed as a control mechanism for certain
search techniques. Usually it is described as a
stochastic technique for finding optimal solutions
to combinatorial problems. As such it can be used
in a special type of neural network, the Boltzmann
machine, a value-passing architecture with
probabilistic processing elements. Selman and
Hirsh[1987] gave a solution to the parsing problem
in a Boltzmann machine with simulated annealing.
Context-free language parsing becomes an energy
minimisation problem and parallel, stochastic
relaxation is used to simulate annealing for
obtaining a representation of a parse tree at
thermal equilibrium. The stochastic factor in the
search process allows the system to escape from
local minima during a search. As they mention, a
next step in their research should be the addition
of a semantic component to extend the disam-
boguation capability of the network. In this way an interactive processing model can be obtained. An extended network should be able to map syntactic constituents into, for example, case roles.

Another example of an annealing parser, but now presented outside the context of connectionist networks, is a parser introduced by Sampson [1988]. The approach is part of statistics-based natural language research in which no clear-cut grammatical/ungrammatical distinction of sentences is assumed. In Sampson's approach possible tree structures are generated at random with the aim to choose a tree with a high plausibility. For that reason it is necessary to have a tree-evaluation function, to define a class of local changes that can be made to a tree and to define an annealing schedule. Tree evaluation is based on statistics. The local changes to trees consist of selecting a node at random, disconnect its subtree and locate it somewhere else. For the latter operation some constraints are defined.

Generating tree structures from elementary parts and evaluating them with respect to a given sentence is also the approach to the parsing problem presented by Kempen and Vosse[1988]. Another similarity is the use of simulated annealing as control mechanism. However, their approach seems to be more promising and it allows for semantic and pragmatic influences on the tree formation process.

Syntactic trees are constructed out of so-called segments. A segment consists of two nodes connected by an arc. The arc is labelled with a syntactic function (Head, Subject, Object, etc.) and feature matrices are associated with the nodes of a segment. In Fig. 9 two segments with arc labels and feature matrices are displayed.

![Fig. 9 Two segments](image)

By unification of feature matrices a foot node and a root node of a segment can be merged and a concatenation of segments is obtained. Similarly, roots of segments can be unified. As an example, from the Kempen and Vosse paper, consider the function of the auxiliary verb DO segment with the Subject segment, as displayed in Fig. 10.

![Fig. 10 Syntactic tree formation](image)

Parsing consists of syntactic tree formation out of segments. Metaphorically, this can be compared with the biosynthesis of proteins. The idea is that in a 'test tube' - called the Unification Space - nodes will continuously hit upon each other, each time feasibility of unification is checked, and the probability of unification depends on the activation levels of the nodes and a grammatical fitness function. This function matches word order in the parse tree with that of the input sentence. In addition it can take into account semantic, pragmatic and lexical factors. Hence, this tree formation process subscribes to the interactive sentence processing models. The segments are retrieved from a lexicon in response to words recognised. Due to the activation level decay merged nodes may separate again and stronger unifications become possible. If the parsing process is successful the global excitation level of the Unification Space decreases and an equilibrium with a stable, 'frozen' configuration of segments, the parse tree, is obtained. Although not implemented that way, the idea is that unification may occur in parallel. From the performance of this parser on sentences containing syntactic and semantic difficulties the authors conclude that they have designed a computational model of a psychologically plausible parser.

6.3 Genetic Parsers

Ideas similar to those presented by Sampson can be found in De Weger[1990]. However, instead of simulated annealing, a genetic algorithm is adapted for context-free language analysis. Usually, in genetic algorithms, codings representing solutions of problems consist of bit strings. For parsing other codes seem to be more natural. In this paper, the search space consists of integer representations of syntactic trees. The evaluation function which defines the fitness of a parse tree is based on the occurrences and the positions of
words. Two recombinant operators are defined: mutation (replacing a random subtree by a randomly created parse tree with the same top node) and (tree) cross-over (from two random trees two new trees are constructed by swapping subtrees). Experimental results show that with these suitably defined operations genetic algorithms can be applied to the parsing problem. Obvious extensions of this work include the introduction of tree or grammar dependent genetic operations, the introduction of features and unification in the tree formation process, and the introduction of semantic and contextual influences on genetic tree formation.

In this example, a genetic algorithm was used, given a grammar, to find parse trees for sentences. Hence, the algorithm acts as a parser. Genetic algorithms have also been used to learn parsers (and grammars) for sets of example sentences. That is, genetic algorithms, as probabilistic optimisation algorithms, have been used to learn (induce) machines that accept languages corresponding to certain grammars as well as learning grammars generating certain languages (grammatical inference). These problems are NP-complete, that is, we can not expect to find an efficient algorithm that learns the correct machine or grammar (Gold[1978]). However, heuristic search techniques allow efficient processes to approximate the correct solution. Genetic algorithms have been used to learn descriptions for FSA for accepting regular languages (Zhou & Grefenstette[1986]) and for learning descriptions of context-free grammars (Wyard[1991]) or pushdown automata (PDA) that accept context-free languages (Sen & Janakiraman[1992]). Learning requires a given set of legal and illegal sentences of the language. In the latter paper the search space consists of codings of (deterministic) PDA into bit strings that can be manipulated by the recombinant operations of the genetic algorithm. The evaluation function that measures the quality of the bit string first decodes the bit string to a set of PDA rules and then the PDA is simulated on the input string. Penalties are given if simulation does not lead to acceptance. For each PDA (bit string) the fitness follows from the accumulation of penalties over all training instances. Similar experiments, with better results, have been done by Huijser[1993] at the University of Twente and are being done by Lankhorst of the University of Groningen. It is not at all clear at this moment how the results for formal language examples translate into real natural language description and use.

7. **ROBUST ANALYSIS AND INTERPRETATION**

7.1 Robust Parsing: From Speech Signal to Domain Knowledge

In 1994 new research on robust natural language analysis will be started by the Parlevink research group of the University of Twente. The research that will be performed has to build on three existing sub projects in the Parlevink research.

1. Current Ph.D. research on "Parallel Parsing of Natural Language". In this research a uniform framework is designed for the specification of methods for syntactic analysis and unification for language analysis. The framework allows to make comparisons between different algorithms at a global level and it allows the translation of the insights that are gained to new algorithms, more suitable for certain applications or implementations. For example, a number of parallel implementations of algorithms have been introduced. Also sequential implementations that deviate from the usual left to right processing of sentences can be deduced. This research will be concluded in 1993. See Nijholt[1993a,b] and Sikkel[1993].

2. The research on the application of statistical techniques in language analysis methods. In general these techniques can be found in both syntax driven and semantic driven approaches to language analysis. Until now most of the attention in the research group has been directed towards the syntactic approach. A new model of stochastic grammar has been introduced that allows the association of context-dependent probabilities with production rules. Current research concentrates on the generative power of this formalism, consistency properties and methods for parameter estimation based on corpus analysis. At this moment we don't expect that research on connectionist natural language processing (cf. Nijholt[1993c]), despite inherent possibilities for robust and integrated processing of language, will become part of the research proposed here. The main reason is that research in this area until now has not produced results that can be used in systems that model language use and behaviour in a particular, realistic, domain of discourse.
(3) Ph.D. research on dialogue modelling for natural language interfaces. In this research the emphasis is on semantics. A uniform framework for the representation of those aspects of meaning that are relevant in the context of a man-machine dialogue is strived for. Robustness, approached from a semantic viewpoint, is part of the research.

The research proposed here starts with the syntactic and unification framework mentioned in (1) and is expected to make the translation to methods that are suitable for the description and processing of sentences which are not necessarily well-formed (according to the grammar formalism that is used), which may have missing information and which may have words with associated probabilities and vague word boundaries. One of the starting points is again that parallel implementations and methods that deviate from the usual left-to-right processing need to be considered. Another starting point is that by using a (possibly simplified) model for domain knowledge (e.g., an abstraction of the database) and the results and insights gained in research on information dialogue modelling, it should be possible to map sentences onto a relevant meaning representation.

In order to deal with ill-formed input the following global approaches for constraint-based formalisms (attribute/unification grammars) can be distinguished:

- transform grammatically ill-formed input into grammatical input; continue analysis according to the formalism
- combine partially created structures (obtained from parsing and unification) using semantic knowledge

A technique to obtain partial structures is to partially delay unification/evaluation during parsing (Tomatschek[1993]). In this way a trade-off can be obtained between rejecting parse paths by violating constraints in the grammar rules and constraints expressed by feature information.

In order to obtain more robustness of a parser and to obtain more natural 'feels less' interaction between a user and the natural language processing system domain-specific knowledge should be brought to bear when interacting about grammatical deviations. Another, related approach to obtain robustness, is advocated in Carbonell and Hayes[1987]. Their multiple construction-specific parsing strategies allow the use of the appropriate knowledge (syntactic, semantic, task and domain specific knowledge) at the right time. Among the illustrations of the advantages of their approach they discuss the CASPAR system, a parser that provides a natural language command interface to an interactive computer system.

7.2 Integration: Speech and Augmented Grammars

In the first phase of the research the integration of speech recognition and syntactic formalisms should take place. A possible example of the integration we aim at can be found in Kita et al.[1991], where a Hidden Markov Model / LR system is obtained by combining a 'phonemic grammar' with a generalised LR syntax grammar. An other example is the integration of morphological and syntactic analysis based on the (generalised) LR algorithm as presented in Tanaka et al.[1993]. In general we can assume that an acoustic phonetic front-end produces a network of phonemes that can be input to a speech parsing system. A word lattice is another starting point for a parsing system for speech that should be taken into consideration. Research examples for this aspect can be found in Carter[1992], who uses a left-corner backtracking algorithm and Vosse[1993], who uses the generalised LR algorithm of Tomita. This research should be done in the context of the unifying syntactic and unification framework obtained in earlier research and parsing methods studied in this framework, e.g. generalised LR parsing and head-corner parsing (Sikkel & Op den Akker[1993]). In Vosse's research the generalised LR algorithm has been extended to augmented (attributed) context-free grammars. As is well known in the compiler technology literature input errors can sometimes be handled by adding so-called error rules to a grammar (see again Vosse[1993]). This is an other line of research that has to be considered. At this moment we do not foresee an extension in the direction of using prosodic information for solving syntactic and other ambiguities during language processing. One line of research that will be followed is the design of a model that describes the ways syntax, semantics and other knowledge can be called to assistance for continuous speech recognition.

In Lavic and Tomita[1993] robustness with respect to noise in the input (spontaneous speech) and limited grammatical coverage is obtained by ignoring unrecognisable parts of a sentence. This skipping is accommodated in the generalised LR
algorithm. The main problem with this approach is the discrimination between good and bad parse results and the development of computationally tractable heuristics. The research proposed here aims at integrating a 'skipping' algorithm with methods that are supported by other than syntactic (that is, context-free grammar rules) knowledge sources.

7.3 Integration: Statistical Techniques
Embedding statistical techniques in the grammar formalisms will lead to (new) definitions of stochastic grammars and probabilistic parsing methods. Research views which are of particular interest for the research proposed here can be found in Briscoe & Carroll [1993], Church & Mercer [1993], Ng & Tomita [1991], Wright et al. [1991] and Ophoff [1992]. Apart from this research, Op den Akker [1993] suggestions for new approaches to probabilistic parsing can be found and these suggestions are topics of current M.Sc. research. In the second phase of the research described here it will be necessary to investigate how to model the use of statistical information during (syntax-driven) language processing. For example, one may think of a model that gracefully encompasses trigrams and grammatical constraints in one formalism. Part of this investigation can be based on the following observations.

A stochastic grammar (not necessarily context-free, it is also possible to look at stochastic variants of formalisms that have been introduced in computational linguistics, such as tree adjoint grammars or unification grammars) induces a probability distribution for the occurrences of n-grams (sequences of letters or words of length n), and for n-grams that occur in a certain context, for example after a certain prefix of a sentence. During recognition of erroneous input this analytically obtained probability distribution can be used. The question needs to be answered how the existing parsing methods can be provided with such a statistical theory based method of error detection (and if necessary, correction) such that they can be made robust with respect to the analysis of incomplete and erroneous input. Research can start by looking at techniques for error correction based on the theory of stochastic finite automata. Obviously, the possible integration with speech recognition as discussed before should be maintained.

In the next phase it should be investigated how an extension can be designed in which methods are used where, based on statistical n-gram analysis of corpora, (linguistic) grammatical categories are assigned to phrases in texts. For example, there exist methods that look for occurrences of noun phrases, as such a rather arbitrary restriction. Results of this research look promising (more than 90% of the noun phrases are recognised correctly). Extension and generalisation (towards the recognition of other and possibly more complex structures than noun phrases) of this method of analysis in combination with an analysis based on a given fragmentary stochastic grammar can lead to very robust parsing when the statistics includes information about often occurring erroneous and incomplete sentences.

7.4 Integration: Semantic and Domain Knowledge
In the research proposed we assume that robust language analysis is done in the context of a natural language interface to a database with SQL as formal query language. This does not mean that the results of this research are only of interest for this application. Robust language analysis is of importance for any natural language processing system and research mentioned above is performed by researchers interested in, among others, spelling correction, information retrieval or portable natural language interfaces. However, in this phase of the projected research we want to choose an application in order to show the usefulness of the research in the previous phases and to add a few extensions which will make it possible to embed the research results in a more comprehensive approach to natural language processing. Moreover, the application is chosen because it is the main research application of present Parlevink research, it gives access to commercial applications and it is the topic of a joint research effort with PTT Research on the design of a natural language interface to a theatre information system.

For the chosen application natural language utterances have to be translated into SQL queries. Generation of queries can be done from a so-called context frame or a dialogue act, a representation of the meaning of a dialogue so far and as far relevant for formulating queries. Knowledge that will be constructed in such a frame should be used to connect, if necessary, fragmentary results of syntactic and semantic
analysis and to make the meaning representation more complete (see e.g. the papers of Senelff[1992a,b] and Senelff et al.[1992] about the TINA system for similar ideas, Lafferty et al. [1992] and Kirtner & Lytinen [1991]). In particular attention has to be paid to the Core Language Engine (Alshawi et al. [1992]). CLE has a unification-based syntactic and semantic analysis and both the relations between the distinguished levels of semantic representation and the role domain knowledge plays during analysis are described. CLE is an example of a 'multi-level semantics' system. Multi-level representation of semantics allows a distinction between the use of domain-dependent and domain-independent knowledge. As such, it provides robust parsing with a framework in which linguistic and domain knowledge can be embedded. In addition, in CLE we find the use of preference rules that assign weights to sentence interpretations in order to obtain a most plausible interpretation. This approach coincides with the approach that is advocated in this Ph.D. research proposal. It will be interesting to investigate where a possible integration of partial parse results at the frame or discourse history level - as mentioned above - can fit in the multi-level representation of semantics as advocated in the CLE approach. An alternative for the choice of a frame-like representation formalism from artificial intelligence can be, due to the results of the earlier phases of the research, a more logic-linguistic formalism such as Montague semantics, discourse representation theory or situation semantics. This will however heavily be determined by the application of the research results in (a portable) natural language interface to data bases.

8. Further Reading


9. Literature


Charniak, E. and E. Santos [1987]. A connectionist context-free parser which is not context-free, but then it is not really connectionist either. Proceedings of the Ninth Annual Conference of the Cognitive Society, 1987.

Church, K.W. & R.L. Mercer [1993]. Introduction to the special issue on computational linguistics


Wyard, P. [1991]. Context free grammar induction using genetic algorithms. 4th *Int. Conf. on Genetic Algorithms*.


Typology and Logical Structure of Natural Languages

Vincenzo Manca
Dipartimento di Informatica
Università di Pisa
Corso Italia, 40 - 56125 Pisa - Italy
e-mail: manca@di.unipi.it

Keywords: Formal Grammar, Linguistic Analysis, Representation Formalism, Universal Grammar.

1 INTRODUCTION

The relationship between logic and grammar has been a fundamental theme since the early investigations of both disciplines. The grammatical theories of Plato, Aristotle and Dionysus Trax, which are the basis of all further grammatical studies, are strongly connected to a logical analysis of language.

In the Trivium/Quadrivium partition of medieval culture, grammar was a discipline in the trivium together with logic (Dialectic) and Rhetoric. Furthermore, Scholastic logic was mostly devoted to Speculative Grammar (see [5]). In fact, language was considered as a speculum (mirror) of reality (modus essendi), and semantics (modus significandi) was a formal tool for investigating this mirroring function. Logicians who follow this approach were also called Modistae; their conception of semantics was very modern, because meanings were conceived of in terms of abstract logical entities, and the Scholastic approach constituted the background of the 'Logic' and 'Grammar' of Port Royal (two fundamental treatises from 17th century) where we can find the terminological and conceptual apparatus of usual grammars of European languages (see [36]).

The idea of a logical structure of language was considered with great interest by Leibniz. He projected the definition of a Characteristica Universalis, as a representation tool for reasoning (see Grammaticae Cogitationes in [31]). Moreover, Mathematical Logic was originally an analysis of mathematical language aimed at avoiding paradoxes, and fundamental notions of mathematical logic emerged from rigorous linguistic analyses (Frege, Carnap, Peano, Russell, Tarski, Church, see for example [11], [33]). Note that important early treatises on Mathematical Logic [35], [11] were particularly concerned with the logical analysis of conversational language.
In the 1960s, Chomsky, inspired by Post [34], introduced generative grammars as formal tools to describe the syntax of natural languages (see [7], [8], [9], [10]). In this approach the relationship between logic and language focuses on the notion of a formal system as a device for generating structures (theorems or phrases) that is crucial in theoretical computer science (computation models and programming languages).  

In the 1970s, Montague, in a series of seminal papers (see [25], [26], [27], [28], [39]), introduced intensional (tensed) logic (a predicative high order logic) where it is possible to express formally the semantics of natural languages. Montague’s syntactic approach in [27] is very close to the algebraic approach developed in the specification of abstract data types, see for example [17]). Another important approach to the formal semantics of natural language is based on Situation Theory (see [2]).

Nowadays, these approaches, often extended and integrated (see [19], [32], [12]), are the prevalent paradigms in logical studies of language. Their primary objective is to provide formal tools for describing the syntax and semantics of a given natural language. This is in principle different from a Universal Grammar as a formal theory of linguistic phenomena common to all (or to most) natural languages. This topic is central in our discourse and is related to the following fields:

- The search for a Universal Language, or less pretentiously for artificial languages for interlingual communication, or specific conversational purposes, such as: Volapuk, Interlingua, Latino sine flexione, Esperanto, Glossa, Lincoz, Loglan (see [22], [14], [6], [16]).

- The study of logical formalisms for knowledge representations, in the perspective of Mathematical Logic, Artificial Intelligence and Psycholinguistics (see [27], [29], [21], [9], [19]).

- The investigations into language typology, to classify and compare natural languages, not only with respect to their genetic basis, but also according to general linguistic features [18], [37]).

In the field of language typology, Greenberg’s seminal paper develops a structural analysis of 30 languages (European, African, Asiatic, Oceanian, Amerindian) that shows the existence of language universals [18]. His results can be synthesized by 45 propositions. The following examples give an idea of the approach.

**Proposition 1 (Greenberg’s Universal 1)**
In declarative sentences with a nominal subject and object, the dominant order is almost always one in which the subject precedes the object.

**Proposition 2 (Greenberg’s Universal 2)**
In languages with prepositions, the genitive almost always follows the governing noun, while in languages with postpositions it almost always precedes.

**Proposition 3 (Greenberg’s Universal 3)**
Languages with dominant VSO (Verb + Subject + Object) order are always prepositional.

**Proposition 4 (Greenberg’s Universal 31)**
If either the subject or object noun agrees with the verb in gender, then the adjective always agrees with the noun in gender.

**Proposition 5 (Greenberg’s Universal 32)**
Whenever the verb agrees with a nominal subject or nominal object in gender, it also agrees in number.

**Proposition 6 (Greenberg’s Universal 36)**
If a language has the category of gender, it always has the category of number.

**Proposition 7 (Greenberg’s Universal 42)**
All languages have pronominal categories involving at least three persons and two numbers.

Greenberg’s analysis was developed by means of usual grammatical notions, therefore a natural idea arises: if interesting linguistic universals can be stated in usual grammatical terms, then it is reasonable to search for a logical universal grammar based on traditional grammar.

In this context, the usual linguistic analysis of European grammatical tradition (phonology, morphology, lexicon, parts of speech, logico-syntactic categories (subject, predicate, object complement, genitive, attribute, apposition, etc.)) has two great limitations. It is Greek-Latin oriented and often its concepts are based on informal, semantic and pragmatic aspects.

Below we show that an interesting part of traditional grammars can be reconstructed logically without the aforementioned limitations.
2 SOME GENERAL LINGUISTIC PRINCIPLES

Ferdinand de Saussure, one of the founders of structural linguistics, based his famous Course [38] on three general linguistic principles:

- Arbitrariness
- Linearity
- Articulation²

Consider an utterance of a given language, according to the principle of arbitrariness, there is no extralinguistic reason that determines the identification of its minimal linguistic units. According to the principle of linearity, the linguistic elements are arranged in a linear order, and for the principle of articulation, the linguistic aggregation into bigger units is organized on different levels. On each level, aggregations are obtained by applying grammatical constructions (see Figure 1).

At any articulation level, the rules of linguistic aggregation can be classified into three fundamental kinds:

- selection rules
- position rules
- categorization rules

Selection rules state which elements can be combined together; position rules determine the relative order of combined elements; and finally, categorization rules classify linguistic units and aggregates, according to general or specific categories of a given language. Greenberg's Universals 31, 32 are about selection rules; and Universals 1, 2, 3 about position rules; Universals 36, 42 about categorization rules.

Selection rules may be very different in distinct languages. For example, in several languages a nominal group and a verbal group can combine only if they agree on the person category (singular/plural); while in other languages (e.g. many Semitic languages) they have to agree on the gender category too. In languages based on a case system, nouns and adjective can combine only if they have the same case.

According to position rules, languages can be classified into several types. For example: type SVO (when Subject + Verb + Object is the dominant order of propositional components), VSO, SOV; Pre and Post (when prepositions are more frequent than postpositions); NA (Noun + Adjective) and AN; NG (Noun + Genitive) and GN.

Selection and position rules are connected. In fact, when selection rules are more rigid, position rules can be more flexible and vice versa; namely, when there is a case system, propositional components (subject, verb, object) can be arranged in any order.

Categorization rules are more problematic to identify. In a sense they relate to the grammar of a language rather than to the language itself. In fact, some grammatical descriptions could use some categories that in a different descriptive setting could be avoided. Therefore, it is interesting to find linguistic categories that have a general linguistic relevance.

2.1 SOME GENERAL LINGUISTIC CATEGORIES

If we consider the articulation levels of languages, then we can classify linguistic units into the following categories³:

- monemes (phonemes and graphemes: a, b, c, . . .)
- morphemes (under, stand, able, s, it, is)

²This principle is not explicitly stated in de Saussure's Course, but is widely assumed.

³The linguistic terminology we use may sometimes differ from other usages present in literature. Where traditional terms seem to be somewhat misleading we prefer some neologisms (e.g. grammeme, synagma, morphematic, taxematic).
- 26 -

Figure 2: The structure of a syntagma

- **taxemes** (*understand, understandable, it, is*)
- **syntagmas** (*it is understandable*)

Morphemes constitute the **roots** of a language, they are subdivided into **themes** and grammemes. Taxemes (usually corresponding to the words) are aggregations of morphemes. Particles are morphemes which are also taxemes. Lexemes are taxemes which are not morphemes. Grammemes identify the grammatical constructions; those that are inside lexemes are called affixes. Syntagmas (usually corresponding to the phrases) are aggregations of taxemes. Paradigms are sets of lexemes with the same themes.

A graphic representation of the structure of a syntagma is given in Figure 2.

A definition that formalizes this viewpoint, in the algebraic spirit of Kalmar (see [24]), is as follows.

**Definition 1 (Aggregation Categories)**

The system $GAC_L$, of Grammatical Aggregation Categories a language $L$, is defined by the following sets:

- the finite set $Mon_L$ of its monemes
- the finite set $Morph_L$ of its morphemes
- the set $Theme_L$ of its themes
- the set $Gramm_L$ of its grammemes
- the finite set $Tax_L$ of its taxemes
- the set $Particl_L$ of its particles
- the set $Lexem_L$ of its lexemes
- the set $Paradigm_L$ of its paradigms
- the set $Syntagm_L$ of its syntagmas

These sets satisfy the following conditions, where: for any set $A$, $A^*$ indicates the free monoid generated by $A$ (juxtaposition stands for the monoid operation on $Syntagm_L$; hyphen stands for the monoid operation on $Morph_L$; blank space stands for the monoid operation on $Tax_L$).

$\text{Morph}_L \subseteq \text{Mon}_L^*$

$\text{Tax}_L \subseteq \text{Morph}_L^*$

$\text{Syntagm}_L \subseteq \text{Tax}_L^*$

$\text{Morph}_L = \text{Theme}_L \cup \text{Gramm}_L$

$\text{Particl}_L = \text{Tax}_L \cap \text{Gramm}_L$

$\text{Lexem}_L = \text{Tax}_L \setminus \text{Particl}_L$

$\text{Paradigm}_L \subseteq \mathcal{P}(\text{Lexem}_L)$.

These grammatical categories allow us to express precisely and simply, some basic linguistic classifications which have been identified since research in this field began (see [20]).

A morphematic language is a language where every morpheme is also a taxeme. In this case every grammeme is always a particle and every lexeme is a syntagma. Morphematic languages have two fundamental types: i) isolating or analytical languages, such as Chinese and other Asiatic languages, and ii) fusional or synthesitical languages, such as Mexican and other Amerindian languages (see [20] and [3])$^4$.

In isolating languages every morpheme maintains its original form in the syntactic aggregation; on the contrary, in fusional languages the combined morphemes change form according to the syntagma where they are inserted; therefore different allomorphs of the same morpheme can be identified. Taxeme aggregation, in writing, is usually expressed by blank spaces in isolant languages and by hyphens (often implicit) in fusional languages. This notation suggests that in isolating languages morphemes are somewhat like words (of European languages); while in fusional languages syntagmas approximate to words. The

$^4$In the Mexican language, Humboldt found a particularly interesting fusional aggregation where some morphemes are directly or anaphorically incorporated into the verbal morpheme.
two opposite tendencies morpheme—word and syntagma—word, of isolating and fusional lan-
guages, respectively, are connected to the linguis-
tic alteration. In fact, monemes and morphemes are
more stable, as linguistic units, than lexemes
or syntagmas (the roots of a language are fossil
remains of dead languages). This explains the
millenary immutability of Chinese.

Taxematic languages are those where the dis-
tinction between taxemes and morphemes is real,
for example European languages, and most other
languages. To exemplify the difference in tax-
ematic aggregation, consider the isolating and
fusional representations of the following English
propositions:

• This formalization is understandable.
• Th is form al is ation is under stand able.
• Th-is-form-al-is-ation-is-under-stand-able.

In taxematic languages we find the dichotomy
isolating/fusional at a taxeme level; i.e., mor-
phemes constituting a taxeme (themes and af-
fixes) can be combined by simple concatenation,
or in a way that two or more morphemes pro-
sduce a single morph which expresses them. Lan-
guages that present the former phenomenon are
agglutinative (e.g. Turkish, see [23] or some-
what differently some Bantu languages), the oth-
ers are inflectional (e.g. Indo-European, Semitic
languages) and typically morphemes of number,
gender, case, tense, mood, voice, aspect are in-

grated into inflectional affixes.5

Of course, there are not pure language aggre-
gation types; i.e., a language may have different
aggregations in a few cases; nevertheless what is
relevant is the dominant aggregation behavior.

Taxematic languages, due to the intrinsic na-
ture of their taxemes that force them to have af-
fixes, implicate the possibility of classifying tax-
emes with respect to the kind of affixes they in-
clude. On this basis the usual parts of speech
of European languages are defined (in Swahili, a
Bantu language, there is a complex nouns clas-
sification, based on affixes, that determines com-
pletely the selection rules of the language). Ac-


5 It is interesting to note the geographical distribution of
these phenomena (see [37] for an updated genetic clas-
sification of roughly 5000 languages).

According to Leibniz (see [31] C 287): Particles which
are clearly of different nature are badly confused under
the name of 'adverb', for example, what has 'whether' in
common with 'briefly'?
• the generality of concepts they express, that usually belong to basic syntactic constructs (see later) or to the following basic semantic fields:
  - time
  - space
  - number
  - order
  - quantity
  - comparison
  - inference
  - aspect
  - modality\(^7\)
  - classification\(^8\)

In conclusion, traditional parts of speech have no universal character: the only general distinction about taxemes is the lexemes/particles partition, which is often the only natural one in morphemic languages (see [4]) and is adopted in artificial languages defined on the basis of a rigorous logical analysis (see [6]).

2.2 GENERAL SYNTAXIC CATEGORIES

In this section we outline a general theory of logico-syntactic structure.

Postulate 1 In every language there are syntagmas with the following fundamental syntactic types: i) proposition (assertive, imperative, interrogative), ii) predicate, iii) substantive. \(\diamond\)

Postulate 2 A syntagma with one of the three fundamental syntactic types is a category (propositional, or predicative, or substantival). A syntagma that is not category is a syncategoremata. \(\diamond\)

Postulate 3 In every language any category \(\varphi\) of (syntactic) type \(\Gamma\) that is not a taxeme includes a category \(\ker(\varphi)\) of type \(\Gamma\) that is its kernel. \(\diamond\)

Postulate 4 A category \(\varphi\) is nuclear if \(\varphi = \ker(\varphi). \ \diamond\)

Postulate 5 Any propositional category includes a predicative category. \(\diamond\)

Postulate 6 If \(\varphi\) is a nuclear proposition, then its predicate \(\text{pred}(\varphi)\) is the biggest (w.r.t. the string inclusion \(\subseteq\)) predicative category included in \(\varphi\). \(\diamond\)

Postulate 7 If \(\varphi\) is a nuclear proposition, then the subject of \(\varphi\), \(\text{subj}(\varphi)\) is the string difference

\[\text{subj}(\varphi) = \varphi \setminus \ker(\varphi). \ \diamond\]

Postulate 8 For every proposition \(\varphi\), \(\text{subj}(\varphi)\) is a substantive. \(\diamond\)

Postulate 9 For every category \(\varphi\), if it is not nuclear, then the modifier\(^9\) of category \(\ker(\varphi)\) in \(\varphi\) is the string difference

\[\text{ad}(\varphi) = \varphi \setminus \ker(\varphi). \ \diamond\]

Postulate 10 A coordinative particle is a particle that, combined with two categories of the same type, yields a category of that type. \(\diamond\)

Postulate 11 If the modifier \(\gamma\) of a category \(\varphi\) is constituted by a non coordinative particle and by a substantive, then \(\gamma\) is a complement of \(\ker(\varphi)\). \(\dagger\) \(\diamond\)

Let Subst, Pred, Prop be the set of substantival, predicative and propositional categoremata respectively, let String be the set of strings of monemes, and consider:

\[\Gamma \in \{\text{Subst, Pred, Prop}\}\]

the following are general schemas of syntactic constructions that we indicate with an evident set-theoretic notation:

\(\dagger\)In traditional grammar, if the modifier \(\gamma\) of a category \(\varphi\) is a nominal category, and \(\ker(\varphi)\) is nominal or substantive, then \(\gamma\) is an attribute of \(\ker(\varphi)\). If the modifier \(\gamma\) of a category \(\varphi\) is a substantive category, and \(\ker(\varphi)\) is substantive, then \(\gamma\) is an apposition of \(\ker(\varphi)\).

\(\dagger\)In traditional grammar, if \(\varphi\) is nominal, then \(\gamma\) is a genitive of \(\ker(\varphi)\) (in \(\varphi\)). If \(\varphi\) is verbal and \(\gamma\) is a substantive, then \(\gamma\) is a direct complement (or object complement) of \(\ker(\varphi)\). If distinction nominal/verbal has a universal character, also the notions of direct complement, genitive, attribute and apposition are universal. We will not deal with this problem here, though it is crucial for developing a more explicative theory of linguistic universals.
• predication : Subst × Pred → Prop
• determination : Pred → Subst
• relativization : Prop → Pred
• quantification : Pred → Subst
• modification : Pred × Γ → Γ
• complementation : Pred × Subst → Pred
• factualization : Pred ∪ Prop → Subst
• quotation : String → Subst
• coordination : Γ × Γ → Γ
• enunciation : Prop → Prop

Besides these fundamental syntactic constructions, there are two fundamental syntactic functionalities:

• anaphora
• deixis

Anaphora is accomplished by usual anaphoric pronouns that refer to linguistic entities. Deixis occurs with usual deictic particles, such as personal and demonstrative pronouns that refer to extralinguistic entities.

3 Logical Representation of Propositions

In this section we present a logical formalism \( \mathcal{F} \) based on the analysis previously developed.

Assume two sets of indeterminates, which may be decorated with subscripts or superscripts:

• a set of substantival indeterminates indicated by lower Latin letters;
• a set of predicate indeterminates indicated by lower Greek letters;

In addition, assume a special substantival constant \( \odot \) denoting the enunciative situation and a set of atomic predicates represented by lexemes of a natural language (e.g. English nouns, verbs and adjectives), that we feel free to specify (even implicitly) when they occur in the examples.

Let us begin with an example of logical representation. Consider the proposition:

The man is speaking.

This is represented in \( \mathcal{F} \) with the following formula:

\[ \vdash (\text{man.} \odot) \in (\text{speech.} \odot (\text{time.} \odot)) \]

or also:

\[ a := \downarrow (\text{man.} \odot) \quad (1) \]
\[ t := \downarrow (\text{time.} \odot) \quad (2) \]
\[ \vdash a \in (\text{speech.} t) \quad (3) \]

where:

• \( \downarrow \) is the determination operator corresponding to the particle the;
• \( \in \) is the predication operator;
• \( \vdash \) is the assertion operator;
• \( := \) is the anaphoric operator;
• the dot is the complementation operator;
• round parentheses indicate the construct \( \text{modifier(mmodified)} \);
• \( \text{man} \) and \( \text{speech} \) are atomic predicates.

Now introduce the symbols for the fundamental operators of \( \mathcal{F} \), where: \( \text{Cat, Cat'} \) stand for any two categoremata; \( \text{Prop} \) for any proposition; \( \text{Pred} \) for any predicative categoremata; \( \text{Subst, Subst}' \) for any two substantives; \( \text{Ind} \) for any indeterminate:

• \( \text{Subst.}\text{Pred.} \), \( \text{Subst} = \text{Subst}' \), \( \text{Subst} < \text{Subst}' \) for predication;
• \( \downarrow \text{Pred}, \uparrow \text{Pred}, \downarrow \text{Pred} \) for determination;
• \( \text{Prop(Ind)} \) for relativization of proposition \( \text{Prop(Ind)} \), w.r.t the indeterminate \( \text{Ind} \) occurring in the proposition;
• \( \nabla \text{Pred} \) for universal substantivation over predicate \( \text{Pred} \);
• \( \text{Pred(Cat)} \) for modification of \( \text{Cat} \) with the modifier \( \text{Pred} \);
• \( \text{Pred.Subst} \) for complementation (left associative) of \( \text{Pred} \) with the complement \( \text{Sub} \);
• \( [\text{Prop}], [\text{Pred}] \) for factualization;
There is a pen on the table.

This means that in the enunciative situation there is something that is a pen, something that is a table, and the first thing is placed on the second:

\[
\begin{align*}
a & := (pen, \circ) \\
b & := \downarrow (table, \circ) \\
\models (a \in on, b).
\end{align*}
\]

The case is very different for the proposition:

Give me a pen!

In fact, here the pen is not specified by the enunciative situation. For this reason we consider a second determination operator, such that \((?, Pred)\) represents, in a first approximation, an unspecified element among those satisfying the property determined by \(Pred\). In this manner we can formalize ‘Give me a pen!’ by:

\[
\begin{align*}
a & := ?(pen, \circ) \\
b & := \downarrow (speak, \circ) \\
c & := \downarrow (listen, \circ) \\
\{ c \in give, a. (receive, b) \}.
\end{align*}
\]

It is outside the scope of this paper to outline a formal semantics of \(\mathcal{F}\). In fact, we only want to find a coherent notational system which can identify and represent the main logical notions on which linguistic constructions are based. To this end, it is sufficient to assume that:

- substantives represent individuals (of some - maybe high order - universe) of these fundamental kinds: objects, classes, numbers, quantities, qualities, instants, spaces, events, actions, situations (also combined together);

- predicates represent properties over these individuals;

- propositions represent assertions or questions or commands involving individuals.

We do not specify these assumptions in more detailed and rigorous form. The intended meaning of a logical formalization of a proposition has to be obtained by giving content to the indeterminates, according to the determinative formulas, in order to satisfy the assertive or imperative or interrogative formulas where they occur. For example, in the above formalization, starting from
Formula 10, we can see that \( c \) has to give an object \( a \) to a receiver \( b \); \( c \) and \( b \) are determined by the context, while \( a \) has to be selected from any pen that is present in the context (where several pens or no pen may be present).

The distinction between \( \| \) and \( \| \) operators must not be confused with the distinction between definite and indefinite articles which exists in many languages. In fact, depending on the context, the right representation of an indeterminate article can be obtained either with the \( \| \) operator, or with the operator \( \| \).

Operator \( \downarrow \) is Peano-Russell's descriptive iota operator, while \( \| \) is very similar to Hilbert-Bernays' eta operator. Reichenbach uses them to describe logically the defined/undefined difference of natural language, but Reichenbach's treatment is insufficient in several cases. In fact, often a substantive has only an intensional content, without any extensional meaning. It is not necessary to recall the traditional examples of philosophic logic, based on unicorns, to understand that: propositions such as 'Give me a pen!' or 'There is no man' present this phenomenon at different levels. This intrinsic intensionality of some substantives is one of the main points where the logic of natural language diverges from classic mathematical logic. It was a cornerstone in Scholastic debates about meaning, where the intension/extension dichotomy appears in several contexts (suppositio materialis, suppositio formalis, de re, de dicto). In order to treat intensional aspects of natural language, the semantical approach of [28] is based on a high-order logical setting of intensional logic. But this approach too, where unicorns are definitively treated, does not consider important intensional aspects related to non assertive discourse, such as call by name evaluation of programming languages (as opposed to call by value evaluation).

In our approach the two different determinative operators allow us to express an important difference related to intensional aspects of interrogative and imperative discourse. In fact, we have two kinds of formula: determinative such as \( a := \text{Sub} \) or \( a := \text{Pred} \), and performative (assertive, interrogative, imperative). A determinative formula is specified if it assigns a substantive to an indeterminate, without any application of the \( \| \) operator; otherwise it is unspecified.

Semantics of a formalization, whatever its nature, has to be conceived of in such a manner that: on the one hand, it assigns some formal structure to a given performative formula constructed ac-

3.2 RELATIVIZATION AND QUANTIFICATION

Relativization allows us to obtain a predicate from a proposition where an indeterminate occurs. In many languages this construct is expressed by relative constructions. For example, from

\[ a \in \text{love.} \]

abstracting with respect to \( a \), we can obtain the predicate

\[ \hat{a} \in \text{love.} \]

that identifies the property of an individual \( a \) such that \( \text{love.} \). A relativization can be applied on a proposition where another internal relativization occurs. For example, the property of loving \( b \) who loves \( c \) can be obtained by relativizing the proposition:

\[ (a \in \text{love.} \hat{b}) \wedge (\hat{b} \in \text{love.} c) \]

with respect to \( a \). In this case the relativization is represented with a double hat relative to the indeterminate \( a \), i.e.:

\[ (\hat{\hat{a}} \in \text{love.} \hat{b}) \wedge (\hat{b} \in \text{love.} c) \].

Of course, this rule can be generalized: the indeterminate relativization has \( n+1 \) hats if some relativization with \( n \) hats occurs in the proposition. This kind of relativization is present, but in a simpler form, in Reichenbach (see [35]) and is very similar to the lambda abstraction introduced by A. Church (see [11]), which is the basis of lambda calculus. But, apart from a different notation, there is an important technical difference between our hat relativization and Church's lambda abstraction: what is usually called the
alpha conversion rule does not hold for hat relativization, i.e., no indeterminate renaming is possible; therefore:

\[ \hat{a} \epsilon \text{love}.b \neq \check{c} \epsilon \text{love}.b. \]

This feature is fundamental in modeling the anaphoric pronouns of natural language by means of indeterminates and in developing a special treatment of quantification that overcomes the usual inadequacies of the classic mathematical logic apparatus with respect to natural language.

The basic problem with logical representations of natural language quantifications is their incapacity to express in substantival form categoremes such as 'Every man'. The problem becomes crucial when quantification occurs in anaphoric contexts. In literature (see [28], [21]) donkey-sentences and love-sentences are examples of this phenomenon:

1. Every farmer who owns a donkey beats it.
2. Every man loves a woman who love him.

In the first proposition the indefinite term 'a donkey' is dominated by a universal quantification that forbids fixing any anaphoric assignment for it. In the second proposition the same phenomenon occurs, but there is also a reference to the quantificational substantive. If we use the usual logical operators of first order logic we get the following formulas:

\[ \forall x y (\text{farmer}(x) \land \text{donkey}(y) \land \text{own}(x, y) \rightarrow \text{beat}(x, y)) \]

\[ \forall x y (\text{man}(x) \land \text{woman}(y) \land \text{love}(y, x) \rightarrow \text{love}(x, y)) \]

These formalizations impose an implicative structure that is not present in the corresponding propositions; moreover quantificational substantives are expressed with a completely different quantificational structure. In other words, in this manner we cannot formalize the categorema 'Every man', but only propositions where it occurs.

Now we present a solution of donkey and love sentences that is intrinsically related to the previous treatment of relativization.

Let \( P(x) \) be a predicate where the indeterminate \( x \) occurs, and \( P_{\text{red}} \) the predicate obtained from it by means of relativization w.r.t. the indeterminate \( x \), then \( (\nabla P_{\text{red}}) \) represents the substantive 'every \( P \)' where \( P \) is the property relative to \( P_{\text{red}} \). Consider the formalizations of donkey and love sentences, in order to show that this symbolization is adequate in the presence of anaphoric references dominated by quantifications.

\[ \models \nabla (\hat{a} \epsilon \text{farmer} \land (\hat{a} \epsilon \text{own}.(b \epsilon \text{donkey}))) \epsilon \text{beat}.b \]

\[ \models \nabla (\hat{a} \epsilon \text{man} \land \forall (\text{be} \text{woman} \epsilon \text{love}.\check{a}) \epsilon \text{love}.b) \]

The following is an example where it is essential to consider the order of relativization of indeterminates.

A man who loves all the women who love him.

In fact, we need an application of operator \( \nabla \) to a proposition where both indeterminates \( a \) and \( b \) occur:

\[ \nabla (b \epsilon \text{woman} \land \forall \check{a}) \]

the corresponding formalization in \( F \) of all the phrase is:

\[ (\hat{a} \epsilon \text{man} \land \forall (\hat{a} \epsilon \text{love} \land \forall (\text{be} \text{woman} \epsilon \text{love}.\check{a})). \]

### 3.3 Modification and Complementation

These two operations express modificative and complementative constructions of natural language, usually realized by attributive phrases, adverbs, or by complementative particles. In \( F \) there is a clear distinction between a modification such as 'big man' and a complementation such as 'page of the book': the former leads to \( \text{big}(\text{man}) \), the latter leads to (page \( \) 1 (book)). Particles play either modificative or complementative roles: by changing the meaning of predicates or by extending the number of arguments of the predications and by indicating complementation roles: relationship, causality, finality, location, instrument, quality, quantity. For example, in English 'look at', 'look after', 'look for', 'look forward' express different binary relations where the same basic verb is modified by different prepositions; while in the expression 'to take a with b' the predicate 'take' is extended to represent a binary relation, where the role of the argument \( b \) is specified by 'with'. Unfortunately, similar roles are expressed by different particles in different contexts and the same particle can express very different roles in different contexts.11 For example, in English we have:

11The knowledge of many languages is, to a great extent, based on the correct use of complementation particles. Often, the choice of a specific particle confers expressive peculiarities which are very important at a con-
• We agree about most things.
• Let us agree on a date!
• He was waiting for her.
• He left for Paris.

In $\mathcal{F}$, if $Pred$ represents a verb, then the role of a complementation can be expressed by a suitable modification. For example, $Pred$'s $finality(a)$ indicates that $a$ is a substantive that specifies the meaning of $Pred$ with a final role.

Modification, complementation and predication (at different levels) are related to the vagueness of natural language, inasmuch as they 'fuse' meanings. Sometimes it is important to reduce vagueness, and a formal setting is an essential tool for such a task; nevertheless, vagueness cannot be completely avoided, if some intrinsic pecularities of natural language have to be represented. For example, the semantic role of 'big' in modifications such as: $\text{big(man)}$ or $\text{big(success)}$ is certainly vague. The metaphoric power of language is based on different mechanisms of meaning combinations. A deep understanding of these mechanisms is a complex task, related to a general explanation of metonymy and metaphor, and perhaps beyond the realm of logic. However, at a level of logical formalization, it is appropriate to represent modification and complementation differently from predication. To show the importance of this distinction, consider the formalization, in Reichenbach style, given in [41], of the sentence:

\[ \text{The three bitterly crying children walked home fast.} \]

Here the constructs of predication, modification, and complementation have the same logical representation:

\[ \exists x y t f g. \\
\text{child}(x, t) \land f(x) \land \text{cry}(f, t) \land \text{bitter}(f, t) \land \\
\text{mention}(x, a) \land g(x) \land \text{walk}(g, y, t) \land \\
\text{fast}(g, t) \land \text{home}(y, t) \land \text{have}(x, y, t) \land \\
\text{prec}(t, i_0) \land \text{time}(i_0, i_0). \]

One inadequacy of the above formalization is due to the arbitrariness in the arguments of predication. A formula $\text{walk}(g, y, t) \land \text{to}(g, y, t)$ instead of $\text{walk}(g, y, t)$ would be more appropriate. In fact, if these children were walking along a river $z$, then the information about walking would lead to something like $\text{walk}(g, y, z, t)$, but now, how can we distinguish between the roles of its arguments? By using a binary walking predicate, we could uniformly extend the previous formalization with the formula:

\[ \text{walk}(g, t) \land \text{to}(g, y, t) \land \text{river}(z, t) \land \text{along}(g, z, t). \]

Another inadequacy is the redundant occurrence of the temporal parameter, and again, its arbitrary position: why not consider the formula

\[ \text{g}(z, t) \land \text{walk}(g) \land \text{to}(g, y) \]

by moving the temporal parameter as an argument of the predicate $g$? The third inadequacy is the way modification is treated; for it implies reading any predication in the entire context of the formalization. Namely, the predication $\text{bitter}(f, t)$ can be correctly understood only if we consider that it occurs with the predication $\text{cry}(f, t)$. Finally, there is a very subtle implicit assumption: some linguistic information is represented directly by first order predicates: $\text{have}$, $\text{home}$, $\text{mention}$, $\text{time}$, $\text{prec}$; the others are second order predicate: $\text{walk}$, $\text{cry}$, $\text{bitter}$, $\text{fast}$. What is the basis of the distinction? It seems that it is dependent on the context and on the particular language on which is based, and of course, this is a limit for a formal theory of logical representation of natural language. The same proposition leads in $\mathcal{F}$ to the following formalization where the aforementioned inadequacies are avoided:

\[ \text{s} := \text{situation} \]
\[ a := \downarrow (3(\text{child}), a) \]
\[ b := \downarrow (\text{house}, a) \]
\[ t := \downarrow (\text{time}, \circ) \]
\[ t' := \downarrow (\text{time}, s) \]
\[ \varphi := (\text{fast}(\text{walk}), \text{destination}(b)) \]
\[ \psi := \text{bitter}(\text{cry}) \]
\[ \models a \in (\varphi \land \psi \{t' < t\}). \]

3.4 FACTUALIZATION AND QUOTATION

A general principle of language is that any linguistic entity can become something which can be spoken about; i.e., any categorem can be transformed into a substantive. For example, in the proposition
Eating too much is dangerous.

the verb 'to eat' becomes the subject of a proposition. In a first approximation, its formalization in $\mathcal{F}$ is:

$$[\text{big(eat)}] \text{edanger}$$  

a more detailed formalization could be the following:

$$r := \downarrow \text{right(eat(quantity))}$$  

$$q := \exists \text{eat(quantity)}$$  

$$e := [(\text{eat.quantity(q)} \land (q > r)]$$  

$$\models e \in \text{danger}.$$

Of course the considered proposition is intrinsically vague (what is the right quantity of eating?). But this is not a problem of the formalization, which only aims at translating adequately the linguistic form in terms of formal operations: if there is some vагueness in the proposition considered, this has to remain in its formal translation too.

Substantiation is also present in linguistic contexts like: 'to think that', 'to believe that', 'Amundsen's flight'. Moreover, usual subordination relationships between propositions can be represented by means of predications. For example if $P$ and $Q$ are propositions, then 'P because Q' in $\mathcal{F}$ can become something like:

$$\models [P] \epsilon \text{cause.}[Q].$$

The role of the quotation operator is evident. It is necessary in treatment of linguistic entities as mentioned signs rather than as used signs, where we speak about names of things rather than the things themselves.

3.5 Coordination and Enunciation

Coordinations allow us to connect in conjunct or disjunct form two categoremata of the same type by providing a categorema of that type. We consider negation as a particular coordination with one argument.

Enunciation is an operation that states the performative function of a proposition. In $\mathcal{F}$ we can express assertive, interrogative and imperative propositions. For example, the following simple question:

Where is the station?

leads to the formalization:

$$a := \downarrow (\text{station. o})$$  

$$b := \exists \text{place}$$  

$$\exists b(a \in \text{in } b).$$

3.6 Anaphora and Deixis

Anaphora is an essential feature of language and an essential tool in constructing formalizations in $\mathcal{F}$. The idea of using anaphoric elements in a logical setting for natural language representations is developed in a systematic manner by Kamp in [21]. Nevertheless the structure of $\mathcal{F}$ is very different from the Discourse Representation Structures presented in [21], where little consideration is devoted to the classical grammatical apparatus.

The treatment of deixis in $\mathcal{F}$ is similar to that presented by Reichenbach in [35], where the constant $\theta$ corresponds to the constant $\odot$ of $\mathcal{F}$. However, Reichenbach's treatment of reflexive tokens is more general, and perhaps beyond the aims of a linguistic formalization, inasmuch as it is oriented towards a philosophical analysis of semiotic aspects of deictic concepts.

4 Developments and Applications

The outlined formalism can be developed in many directions. First of all, it could be used in real situations, by introducing notational extensions which are useful in practice. Moreover, in specific semantic domains it could be specialized in order to deal adequately with very frequent semantic and pragmatic contexts.

Formalism $\mathcal{F}$ can be seen on an intermediate level between the semantical level investigated by Montague, and the syntactic level of a generative grammar. Thus, it would be interesting to study: on the one hand, how $\mathcal{F}$ can be translated into some kind of intensional logic, and, on the other hand, how a formalization in $\mathcal{F}$ can be considered as an input of linguistic generative algorithms producing correct texts. Of course, a semantics of $\mathcal{F}$ stated in some procedural style could be another interesting possibility to investigate.

Another formal development of $\mathcal{F}$ could be obtained by studying inferential systems within it, that express common sense reasoning or axioms stating synonymy relationships between paraphrases.

An interesting aspect of $\mathcal{F}$ is the identification
of a general lexicon, possibly based on roots common to many languages, where conceptual and interlingual representations can be developed.

Finally, given a natural language \( L \), let us indicate by \( F(L) \) the formalism \( F \) based on lexemes of \( L \). If \( L' \) is another natural language, then an intrinsic possibility of \( F \) is the transformation of logical representations of \( F(L) \) into logical representations of \( F(L') \), only by stating a correspondence between the lexicons of two languages. This could be seen as a lexicon directed translation of logical representations. Formally:

\[
P \downarrow \\
\phi(\lambda_1, \ldots, \lambda_k) \downarrow \\
\phi(\lambda'_1, \ldots, \lambda'_k) \downarrow \\
P'
\]

where \( \phi(\lambda_1, \ldots, \lambda_k) \) indicates a formalization in \( F(L) \) of a proposition \( P \), with lexemes \( \lambda_1, \ldots, \lambda_k \), and \( \lambda'_1, \ldots, \lambda'_k \) are the respective lexemes of \( L' \), and \( \phi(\lambda'_1, \ldots, \lambda'_k) \) is the formalization of \( F(L') \) obtained from \( \phi(\lambda_1, \ldots, \lambda_k) \) by replacing any lexeme of \( L \) by the corresponding lexeme of \( L' \).

An important question arises now: can we identify suitable hypotheses, that guarantee a semantical equivalence between propositions \( P \) of \( L \) and \( P' \) of \( L' \)? This result, considered as a sort of homeomorphism principle, could be an important result for theoretical and practical investigations about a formal theory of translation.

REFERENCES


DATA ORIENTED PARSING AS A GENERAL FRAMEWORK FOR STOCHASTIC LANGUAGE PROCESSING

Rens Bod
OTS, University of Utrecht
ILLC, University of Amsterdam
Spuistraat 134, 1012 VB Amsterdam
The Netherlands
rens@alf.let.uva.nl

ABSTRACT

In this paper we compare stochastic tree substitution grammars with other grammars in the context of formal language theory. Stochastic tree substitution grammars (STSGs) constitute the formal basis for "data oriented parsing" — a recently developed approach to stochastic language processing which works directly with an annotated corpus. It is shown that the stochastic (string and tree) languages generated by stochastic context free grammars (SCFGs) can also be generated by STSGs. On the other hand, there are STSGs generating stochastic languages that cannot be generated by SCFGs. By putting constraints on STSGs, their behavior can be experimentally compared with the behavior of other stochastic grammars.

1 INTRODUCTION

The current tradition of natural language processing systems is based on linguistically motivated competence grammars of natural languages. However, it turns out that as soon as such a grammar characterizes a non-trivial part of a natural language, almost every input string of reasonable length gets a large number of different analyses. Since most of these analyses are not perceived as plausible by a human language user, there is a need for distinguishing the plausible parse(s) of an input string from the implausible ones. Stochastic language models tackle this problem by assuming that the most plausible parse of an input string is the parse with the highest probability.

Many instantiations of this idea estimate the probability of a parse by assigning application probabilities to the rewrite rules of a context free grammar (Suppes, 1970; Garside et al., 1987; Fujiyoshi et al., 1989; Jelinek et al., 1990; Corazza et al., 1991; Black et al., 1992; Briscoe & Carroll, 1993; Schabes et al., 1993). By definition of a context free derivation, each rewrite rule is independent of the context within which the nonterminal appears; therefore the probability of a context free derivation (and its corresponding parse tree) is the product of the probabilities of its rewrite rules.

It has been widely recognized that unconditional application probabilities of context free rewrite rules are not really adequate for estimating the probability of a parse, since they do not describe how the probability of syntactic structures or lexical items depends on their context. In order to overcome this lack of context-sensitivity, several alternative approaches have been proposed. In (Magerman & Marcus, 1991; Magerman & Weir, 1992), a stochastic CFG is combined with part-of-speech trigram statistics. Another approach applies probabilities to elementary trees (Resnik, 1992; Schabes, 1992; Schabes, 1993), while in (Black et al., 1993; Black, Garside & Leech, 1993), the probability of a rewrite rule is taken to be stochastically dependent on the rules that generated its left-hand side.

The Data Oriented Parsing (DOP) approach, suggested in (Schabes, 1990, 1992) and developed in (Bod, 1992, 1993), distinguishes itself from the other statistical approaches in that it omits the step of articulating a formal grammar. Instead, it employs a manually annotated corpus as if it were a stochastic grammar. An input string is parsed by combining subtrees from the corpus. In this view, every subtree can be considered as an elementary tree. As a consequence, one parse tree can usually be generated by several derivations that involve different subtrees. This leads to a statistics where the probability of a parse is equal to the sum of the probabilities of all its derivations. It
is hoped that such an approach can accommodate all statistical properties of a language corpus.

So far, the DOP approach has been implemented and tested on a set of hand-parsed strings from the Pennsylvania Treebank, yielding high parsing accuracy (Bod, 1993a). In the present paper, we study the model from a mathematical perspective. We treat DOP in the context of formal language theory and study its status with respect to the theory of stochastic languages. We will also report on some new experimental results.

2 DATA ORIENTED PARSING

Let us illustrate how DOP works with a simple imaginary example. Suppose that a corpus consists of only two trees, where every node is labeled with only lexical or syntactic information.

![Diagram of two tree structures]

Suppose that we employ one operation for combining subtrees, indicated as $\otimes$; it identifies the leftmost nonterminal leaf node of one tree with the root node of a second tree (i.e., the second tree is substituted on the leftmost nonterminal leaf node of the first tree). Then the (ambiguous) sentence "She put the dress on the table" can be parsed in several ways by combining subtrees from the corpus. For instance:

![Diagram of different parse trees]

Other derivations may yield the same parse tree; for instance:

![Diagram of another parse tree]

Thus, a parse can have several derivations involving different subtrees. Using the corpus as our stochastic grammar, we estimate the probability of substituting a certain subtree on a specific node as the probability of selecting this subtree among all subtrees in the corpus that could be substituted on that node. The probability of a derivation can be computed as the product of the probabilities of the subtrees that are combined.

As an example, we calculate the probability of the last derivation given above. The first subtree $(\text{S(NP(she), VP)})$ occurs twice in the corpus among a total of 78 subtrees rooted with an S. Thus, its probability of being selected is $2/78$. The second subtree occurs once among a total of 60 subtrees that can be substituted on a VP, hence, its probability is $1/60$. The probability of selecting the subtree NP(he,dress) is $1/18$, since there are 18 subtrees in the corpus rooted with an NP, among which this subtree occurs once. The probability of the resulting derivation is therefore equal to $2/78 \times 1/60 \times 1/18 = 1/42120$. The next table shows the probabilities of all three derivations given above.

<table>
<thead>
<tr>
<th>Derivation</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st example</td>
<td>$1/78 \times 1/18 \times 1/8$</td>
</tr>
<tr>
<td>2nd example</td>
<td>$2/78 \times 1/60 \times 1/18$</td>
</tr>
<tr>
<td>3rd example</td>
<td>$2/78 \times 1/60 \times 1/18$</td>
</tr>
</tbody>
</table>
This example illustrates that a statistical language model which defines probabilities over parses by taking into account only one derivation, does not accommodate all statistical properties of a language corpus. Since we do not know in advance which lexical, syntactic or semantic relationships are statistically interesting, we should take into account the probabilities of all derivations of a parse tree. By defining the probability of a parse as the sum of the probabilities of all its derivations, no relationship that might possibly be of statistical interest is ignored.

3 FORMAL ASPECTS

There can be various instantiations of the Data Oriented Parsing approach as proposed in (Scha, 1990). The version of DOP illustrated above we will call a Stochastic Tree Substitution Grammar (STSG), which is a stochastic enrichment of a Tree Substitution Grammar (Joshi & Schabes, 1991). An STSG is a DOP model if its elementary trees and corresponding probabilities are extracted from a manually annotated natural language corpus. In this section, we abstract from the use of a corpus and study the properties of STSGs in general.

Definition 1 Stochastic Tree Substitution Grammar
A Stochastic Tree Substitution Grammar G is a five-tuple \((V_N, V_T, S, R, P)\) where
- \(V_N\) is a finite set of nonterminal symbols.
- \(V_T\) is a finite set of terminal symbols.
- \(S\) \(\in\) \(V_N\) is the distinguished symbol.
- \(R\) is a finite set of finite trees whose interior nodes are labeled by nonterminal symbols and whose yield nodes are labeled by terminal or nonterminal symbols.
- \(P\) is a function which assigns to every tree \(t\) \(\in\) \(R\) a probability \(p(t)\). For a tree \(t\) with a root \(\alpha\), \(p(t)\) is interpreted as the probability of substituting \(t\) on \(\alpha\).

We require, therefore, that \(0 < p(t) \leq 1\) and \(\sum_{\alpha \in \Gamma} p(t) = 1\).

If \(t_1\) and \(t_2\) are trees such that the leftmost nonterminal yield node of \(t_1\) is equal to the root of \(t_2\), then \(t_1 \circ t_2\) is the tree that results from substituting \(t_2\) for this leftmost nonterminal yield node in \(t_1\). The partial function \(\circ\) is called leftmost substitution. We will write \((t_1 \circ t_2) \circ \ldots \circ t_{n-1}\) as \(t_1 \circ t_2 \circ \ldots \circ t_{n-1}\), and in general \((\ldots (t_1 \circ t_2) \circ \ldots \circ t_{n-1}\) as \(t_1 \circ t_2 \circ \ldots \circ t_{n-1}\). For reasons of conciseness we will often speak of substitution while intending leftmost substitution.

A leftmost derivation generated by an STSG G is a tuple of trees \((t_1, \ldots, t_n)\) such that \(t_1, \ldots, t_n\) are elements of \(R\), the root of \(t_1\) is labeled by \(S\) and the yield of \(t_1 \circ \ldots \circ t_n\) is labeled by terminal symbols. The set of leftmost derivations generated by \(G\) is thus given by \(\text{Derivations}(G) = \{(t_1, \ldots, t_n) / t_1, \ldots, t_n \in R \land \text{root}(t_1) = S \land \text{yield}(t_1 \circ \ldots \circ t_n) \in V_T^+\}\). For reasons of conciseness we will often speak of a derivation while intending a leftmost derivation. A derivation \((t_1, \ldots, t_n)\) is called a derivation of tree \(T\), if \(t_1 \circ \ldots \circ t_n = T\). A derivation \((t_1, \ldots, t_n)\) is called a derivation of string \(s\), if \(\text{yield}(t_1 \circ \ldots \circ t_n) = s\).

A parse tree generated by an STSG \(G\) is a tree \(T\) such that there is a derivation \((t_1, \ldots, t_n) \in \text{Derivations}(G)\) for which \(t_1 \circ \ldots \circ t_n = T\). The set of parse trees, or tree language, generated by \(G\) is given by \(\text{Parses}(G) = \{T \in \text{Derivations}(G) : t_1 \circ \ldots \circ t_n = T\}\). For reasons of conciseness we will often speak of a parse while intending a parse tree. A parse whose yield is equal to string \(s\), is called a parse of \(s\).

A string generated by an STSG \(G\) is an element of \(V_T^+\) such that there is a parse generated by \(G\) whose yield is equal to the string. The set of strings, or string language, generated by \(G\) is given by \(\text{Strings}(G) = \{s \mid s \in \text{Parses}(G) \land s = \text{yield}(T)\}\).

The probability of a derivation \((t_1, \ldots, t_n)\) is equal to the product of the probabilities assigned to the trees \(t_1, \ldots, t_n\).

The probability of a parse is equal to the probability that any of its derivations occurs. Since the derivations are mutually exclusive, the probability of a parse is equal to the sum of the probabilities of all its derivations.

The probability of a string is equal to the probability that any of its parses occurs. Since the parses are mutually exclusive, the probability of a string is equal to the sum of the probabilities of all its parses. Thus, the probability of a string is equal to the sum of the probabilities of all its derivations.

The stochastic tree language generated by \(G\) is the set of pairs \((x, p(x))\) where \(x\) is a tree from the tree language generated by \(G\) and \(p(x)\) the probability of that tree.

The stochastic string language generated by \(G\) is the set of pairs \((x, p(x))\) where \(x\) is a string from the string language generated by \(G\) and \(p(x)\) the probability of that string.

In the following, we will compare STSG with other grammars in the context of formal language theory. In order to do so, we need some definitions and a convention.
Definition 2 Properness of Grammars
A grammar is called proper if it only such nonterminals can be generated whose further rewriting can eventually result in a string of terminals.

Example: the context-free grammar \((\{S, A\}, \{a\}, S \rightarrow S a, S \rightarrow \alpha a, S \rightarrow \alpha A\}) is proper, since there is a generation \(S \rightarrow \alpha A\) which can never result in a string of terminals.\(^1\)

Definition 3 Finite Ambiguity of Grammars
A grammar is called finitely ambiguous if there is no finite string that has infinitely many derivations.

Example: the context-free grammar \((\{S\}, \{a\}, S \rightarrow S a, S \rightarrow \alpha S, S \rightarrow \alpha A\}) is not finitely ambiguous, since the string \(a\) has infinitely many derivations.

Convention 1
All grammars under consideration in this paper are assumed to be proper and finitely ambiguous.

Definition 4 Weak Stochastic Equivalence\(^2\)
Two stochastic grammars are called weakly stochastically equivalent, if they generate the same stochastic string language.

Note that if two stochastic grammars are weakly stochastically equivalent they are also weakly equivalent (i.e., they generate the same string language).

Definition 5 Strong Stochastic Equivalence\(^3\)
Two stochastic grammars are called strongly stochastically equivalent, if they generate the same stochastic tree language.

Note that if two stochastic grammars are strongly stochastically equivalent they are also strongly equivalent (i.e., they generate the same tree language), and weakly stochastically equivalent.

The first grammar we will compare with STSG is the so-called Stochastic Context-Free Grammar (SCFG).

Definition 6 Stochastic Context-Free Grammar\(^4\)
A Stochastic Context-Free Grammar \(G\) is a five-tuple \((V_N, V_T, S, R, P)\) where
- \(V_N\) is a finite set of nonterminal symbols.
- \(V_T\) is a finite set of terminal symbols.
- \(S \in V_N\) is the distinguished symbol.
- \(R\) is a finite set of productions each of which is of the form \(\alpha \rightarrow \beta\), where \(\alpha \in V_N\) and \(\beta \in (V_N \cup V_T)^*\).
- \(P\) is a function which assigns to every production \(\alpha \rightarrow \beta \in R\) a probability \(p(\alpha \rightarrow \beta)\), for which holds that \(0 < p(\alpha \rightarrow \beta) \leq 1\) and \(\Sigma \alpha p(\alpha \rightarrow \beta) = 1\).

The notions derivation, string language, and tree language are defined as for CFGs. The probability of a derivation (and its corresponding parse tree) generated by an SCFG is equal to the product of the probabilities associated with the productions applied. The probability of a string generated by an SCFG is equal to the sum of the probabilities of all its derivations. The stochastic string language and stochastic tree language generated by an SCFG are defined as for an STSG.

Proposition 1
For every STSG there exists a weakly stochastically equivalent SCFG.

Proof of Proposition 1
Given an STSG \(G\), we convert every tree \(t \in R\) into a context-free production \(root(t) \rightarrow \text{yield}(t)\). This may lead to multiple occurrences of productions, since different trees may have the same root and yield. (Note that these productions define the same string language as \(G\).) To every such production a probability is assigned which is equal to the probability of the tree from which the production is derived. The resulting SCFG \(G'\) may be redundant, as it may contain equivalent productions with different probabilities. Now it is easy to see that the sum of the probabilities of all derivations of a string in \(G\) is equal to the sum of the probabilities of all derivations of this string in \(G'\). This means that \(G\) and \(G'\) assign the same probability to every string in their string language. Thus, \(G\) and \(G'\) are weakly stochastically equivalent. □

Proposition 2
For every SCFG there exists a weakly stochastically equivalent STSG.

---

\(^1\) In (Jelinek et al., 1990), an algorithm is described that determines whether or not a grammar may be made proper by the elimination of rules (p. 31).

\(^2\) What we call weak stochastic equivalence is usually known as stochastic equivalence; cf. (Fu, 1974).

\(^3\) In (Bod, 1993a), this type of equivalence is called superstrong equivalence.

\(^4\) This definition follows the definitions of a stochastic context-free grammar in (Booth, 1969), (Fu, 1974), (Wetherell, 1980), (Fujisaki et al., 1989), (Jelinek et al. 1990).
Proof of Proposition 2

Given an SCFG $G$, we convert every production $\alpha \rightarrow \beta \in \mathcal{R}$ into a unique tree $t$ of depth one such that $\text{root}(t) = \alpha$ and $\text{yield}(t) = \beta$. To every such tree a probability is assigned which is equal to the probability of the production from which the tree is derived. The resulting STSG $G'$ generates the same string language and tree language as $G$. Now it is easy to see that for every derivation in $G$ there is a unique derivation in $G'$ with the same probability. Thus, the sum of the probabilities of all derivations of a string in $G$ is equal to the sum of the probabilities of all derivations of this string in $G'$. This means that $G$ and $G'$ assign the same probability to every string in their string language. Thus, $G$ and $G'$ are weakly stochastically equivalent.

From the propositions 1 and 2 the following corollary can be deduced.

Corollary 1

The set of stochastic string languages generated by STSGs is equal to the set of stochastic string languages generated by SCFGs.

Proposition 3

For every SCFG there exists a strongly stochastically equivalent STSG.

Proof of Proposition 3

Consider the proof of proposition 2. Since $G$ and $G'$ generate the same tree language and every derivation in $G$ corresponds to a unique derivation in $G'$ with the same probability, $G$ and $G'$ are strongly stochastically equivalent.

Proposition 4

There exists an STSG for which there is no strongly equivalent SCFG.

Proof of Proposition 4

Consider the following STSG $G$ consisting of one tree with a probability equal to 1:

```
  S
 / \
S S
 /  \\
S b
```

The tree language generated by $G$ is equal to the set containing only the above tree. Thus, an SCFG is strongly equivalent with $G$ if it generates only the above tree. An SCFG which generates the above tree should consist of the productions $S \rightarrow Sb$ and $S \rightarrow a$. But such an SCFG generates more than just the above tree. Contradiction.

Proposition 5

There exists an STSG for which there is no strongly stochastically equivalent SCFG.

Proof of Proposition 5

Consider the proof of proposition 4. Since strong stochastic equivalence implies strong equivalence there is no SCFG which is strongly stochastically equivalent with $G$.

From the propositions 3 and 5 the following corollary can be deduced.

Corollary 2

The set of stochastic tree languages generated by SCFGs is a proper subset of the set of stochastic tree languages generated by STSGs.

Though corollary 2 seems an important result, it mainly follows from the property that STSGs are not always strongly equivalent with SCFGs. In the context of stochastic language theory, however, we are not so much interested in tree languages as in stochastic tree languages. Thus, it is more interesting to compare stochastic tree languages of strongly equivalent grammars.

Proposition 6

There exists an STSG for which there is a strongly equivalent SCFG but no strongly stochastically equivalent SCFG.

Proof of Proposition 6

Consider the following STSG $G$ consisting of three trees that are all assigned with a probability of $\frac{1}{3}$:

```
  S
 / \
S S
 /  \\
S b
```

```
  S
 / \
S S
 /  \\
S b
```

```
  S
 / \
S a
```

The three tree languages generated by $G$ are $t_1$, $t_2$, and $t_3$. However, no SCFG can generate all three trees with the same probability. Therefore, $G$ is strongly equivalent with an SCFG but not stochastically equivalent.
The string language generated by $G$ is \{ab$^k$\}. It is easy to show that $G$ is consistent$^5$. The only (proper) SCFG $G'$ which is strongly equivalent with $G$ consists of the following productions.

$$S \rightarrow Sb \quad (1)$$
$$S \rightarrow a \quad (2)$$

$G'$ is strongly stochastically equivalent with $G$ if it assigns the same probabilities to the parse trees in the tree language as assigned by $G$. Let us consider the probabilities of two trees generated by $G$, i.e. the trees represented by $t_1$ and $t_2$.$^6$ The tree $t_2$ has exactly one derivation: by selecting tree $t_3$. The probability of this tree is hence equal to $1/3$. The tree $t_1$ has two derivations: by selecting tree $t_1$, or by combining trees $t_2$ and $t_3$. The probability of this tree is equal to the sum of the probabilities of its two derivations, which is equal to $1/3 + 1/3*1/3 = 4/9$.

If $G'$ is strongly stochastically equivalent with $G$, then it should (at least) assign the probabilities $4/9$ and $1/3$ to the trees $t_1$ and $t_2$ respectively. The tree $t_3$ is exhaustively generated by production (2), thus the probability of this production should be equal to $1/3$: $p(S \rightarrow a) = 1/3$. The tree $t_1$ is exhaustively generated by applying productions (1) and (2), thus the product of the probabilities of these productions should be equal to $4/9$: $p(S \rightarrow Sb) = 4/9$. By substitution we get $p(S \rightarrow Sb) = 1/3$, which implies that $p(S \rightarrow Sb) = 4/9$. This means that the probability of production (1) should be larger than $1$, which is not allowed. Thus, $G'$ cannot be made strongly stochastically equivalent with $G$. ☐

$^5$A stochastic grammar $G$ is called consistent if $\sum_{x \in L(G)} p(x) = 1$ (where $L(G)$ is the string language generated by $G$). That the grammar $G$ in the proof of proposition 6 is consistent can be seen as follows. String $a$ is exhaustively generated by tree $t_2$, thus its probability is $1/3$. String $ab$ is generated either by $t_1$ or by combining $t_2$ and $t_3$, thus its probability is $1/3 + 1/3*1/3 = 4/9$. String $abb$ is generated either by combining $t_2$ and $t_3$ or by combining $t_2$, $t_3$, and $t_3$, thus its probability is $1/3 + 1/3 + 1/3*1/3 = 4/9$. In general, the probability of string $ab^n$ is $4/9n+1$ for $n \geq 2$. The sum of the probabilities of all strings is $1/3 + \sum_{n=2}^{\infty} \frac{4}{9^n} = 1/3 + 4*\frac{1}{9} = 1/3 + 4*1/9 = 1$. We will not investigate the conditions under which stochastic grammars are consistent; the reader is referred to (Booth & Thompson, 1973) and (Fu, 1982).

$^6$Note that the trees $t_1$ and $t_3$ are both elements of the set of (elementary) trees $R$ of $G$ and of the tree language generated by $G$.

Another grammar we want to compare with STSG is the so-called Weighted Context Free Grammar (WCFG).

**Definition 7** Weighted Context Free Grammar (WCFG)$^7$

A Weighted Context Free Grammar $G$ is a five-tuple $(V_N, V_T, S, R, W)$ where $V_N$ is a finite set of nonterminal symbols, $V_T$ is a finite set of terminal symbols, $S \in V_N$ is the distinguished symbol, $R$ is a finite set of productions each of which is of the form $\alpha \rightarrow \beta$, where $\alpha, \beta \in (V_N \cup V_T)^*$. $W$ is a function which assigns to every production $\alpha \rightarrow \beta$ of $R$ a weight $w(\alpha \rightarrow \beta)$ associated with the application of this production.

The weight of a derivation (and its corresponding parse tree) generated by a WCFG is equal to the product of the weights associated with the productions applied. The weight of a string generated by a WCFG is equal to the sum of the weights of all its derivations. The weighted tree language generated by $G$ is the set of pairs $(x, w(x))$ where $x$ is a tree from the tree language generated by $G$ and $w(x)$ the weight of that tree.

From the proof of proposition 6, we raise the question as to whether there exists for the STSG $G$ a strongly equivalent WCFG which assigns to every tree a weight equal to the probability of that tree assigned by $G$. Although a weighted grammar is not a stochastic grammar, this question is interesting since we saw that if to the rewritings $S \rightarrow Sb$ and $S \rightarrow a$ the weights $4/3$ and $1/3$ are assigned respectively, the trees $t_1$ and $t_3$ receive weights that are equal to the probabilities of these trees assigned by $G$.

**Proposition 7**

There exists an STSG for which there is a strongly equivalent WCFG but for which there is no WCFG whose weighted tree language is equal to the stochastic tree language of the STSG.

**Proof of Proposition 7**

Consider the same STSG $G$ as in the proof of proposition 6. The only (proper) WCFG $G'$ which is strongly equivalent with $G$ consists of the following productions.

$$S \rightarrow Sb \quad (1)$$
$$S \rightarrow a \quad (2)$$

$^7$This definition follows (Fu, 1982).
Let us consider the probabilities of three trees generated by $G$, tree $t_1$, tree $t_3$ which were given above, and tree $t_4$ which is given below.

![Diagram of a tree structure]

We have already seen that the probabilities of $t_1$ and $t_3$ assigned by $G$ were equal to $4/9$ and $1/3$ respectively. The tree $t_4$ has two derivations: by combining the subtrees $t_2$ and $t_1$, or by combining subsequently the subtrees $t_2$, $t_2$ and $t_3$. Its probability is equal to the sum of the probabilities of these two derivations, which is $1/3 * 1/3 + 1/3 * 1/3 * 1/3 = 1/9 + 1/27 = 4/27$.

If the weighted tree language of $G$ is equal to the stochastic tree language of $G$, then $G$ should assign the weights $4/9$, $1/3$ and $4/27$ to the trees $t_1$, $t_2$ and $t_4$ respectively. From the proof of proposition 6 it is directly derived that $w(S\rightarrow Sb) = 4/3$ and $w(S\rightarrow xa) = 1/3$. The tree $t_4$ is exhaustively generated by applying subsequently the productions (1), (1) and (2); thus the product of the weights of these productions should be equal to the probability assigned to $t_4$ by $G$, i.e. $4/27$. Thus, $w(S\rightarrow Sb) * w(S\rightarrow Sb) * w(S\rightarrow xa) = 4/27$, which, by substitution, leads to $4/3 * 4/3 * 1/3 = 4/27$, or to $16/27 = 4/27$, which is a contradiction.

4 EXPERIMENTAL ASPECTS

In (Bod, 1992), it is shown that STSGs can be parsed with conventional $n^3$ parsing techniques. In order to select the most probable parse of a sentence, it does not necessarily compare all parses, since there can be exponentially many of them (Church & Patil, 1983). Viterbi's algorithm enables us to find the most probable derivation of a sentence in cubic time (Viterbi, 1967; Fujisaki et al., 1989; Wright et al., 1991). However, this algorithm is not adequate for finding the most probable parse, since in an STSG, the most probable derivation is not necessarily the most probable parse. In (Bod, 1993b), it is shown that by incorporating Monte Carlo techniques into a chart parser, the most probable parse of a sentence can be estimated as accurately as desired, making its error arbitrarily small in polynomial time. More specifically, the time complexity of estimating the most probable parse by a Monte Carlo CKY parser is given by $O(|G|n^3e^{-n})$, where $|G|$ is the size of the grammar (i.e. the sum of the lengths of the yield of each elementary tree), $n$ is the length of the sentence, and $e$ is the estimation error.

Experimental evaluations of our model may be carried out with any corpus of strings annotated with labeled trees. For our experiments, we used a corrected version of the Air Travel Information System (ATIS) corpus (Hempel, 1990) in the Pennsylvania Treebank (Santorin, 1990, 1991; Marcus et al., 1993). This corpus is of interest since it is used by the ARPA community to evaluate their grammars and speech systems.

We used the standard method of randomly dividing the corpus into a 90% training set and a 10% test set. The 675 trees from the training set were used as the corpus of our DOP model, from which the subtrees and their relative frequencies were derived, resulting into a corresponding STSG. The 75 part-of-speech sequences from the test set served as input strings that were parsed using the training set. To establish the performance of the system, the parsing results were then compared with the trees in the test set. (Note that the "correct" parse was decided beforehand, and not afterwards.)

To measure accuracy, one often uses the notion of bracketing accuracy, i.e. the percentage of brackets of the analyses that are not "crossing" the bracketings in the Treebank (Black et al., 1991; Harrison et al., 1991; Pereira & Schabes, 1992; Grishman et al., 1992; Schabes et al., 1993). We believe, however, that the notion of bracketing accuracy is too poor for measuring the performance of a parser. A test set can have a high bracketing accuracy, whereas the percentage of sentences in which no crossing brackets are found (sentence accuracy) is extremely low. In (Schabes et al., 1993), it is shown that for sentences of 10 or 20 words (taken from the Wall Street Journal corpus), a bracketing accuracy of 82.5% corresponds to a sentence accuracy of 30%, whereas for sentences of 20 to 30 words a bracketing accuracy of 71.5% corresponds to a sentence accuracy of 6.8%. We shall

---

8In (Hammersley & Handscomb, 1964), an overview on Monte Carlo methods is given. For Monte Carlo significance testing, we refer to (Besag & Clifford, 1989).
employ the even stronger notion of exact match accuracy, defined as the percentage of the test sentences for which the best analysis matches exactly with the test set analysis in the Treebank.

It is one of the most essential features of our approach, that all combinations of arbitrarily large corpus subtrees are taken into consideration to estimate the probability of a parse. In order to test the usefulness of this feature, we performed different experiments constraining the depth of the subtrees extracted from the training set. The depth of a subtree is defined as the length of its longest path. We calculated for every depth the exact match accuracy, distinguishing between derivation accuracy and parse accuracy. The derivation accuracy refers to the percentage of the test sentences for which the parse generated by the most probable derivation is equal to the test set parse in the Treebank. The parse accuracy refers to the percentage of the test sentences for which the most probable parse is equal to the test set parse in the Treebank. The following table shows the results of these experiments.

<table>
<thead>
<tr>
<th>depth of subtrees</th>
<th>derivation accuracy</th>
<th>parse accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>52%</td>
<td>52%</td>
</tr>
<tr>
<td>≤2</td>
<td>47%</td>
<td>87%</td>
</tr>
<tr>
<td>≤3</td>
<td>49%</td>
<td>92%</td>
</tr>
<tr>
<td>≤4</td>
<td>57%</td>
<td>93%</td>
</tr>
<tr>
<td>≤5</td>
<td>60%</td>
<td>93%</td>
</tr>
<tr>
<td>≤6</td>
<td>65%</td>
<td>95%</td>
</tr>
<tr>
<td>unbounded</td>
<td>65%</td>
<td>96%</td>
</tr>
</tbody>
</table>

Derivation Accuracy and Parse Accuracy for the ATIS Corpus

The table shows that the derivation accuracy and the parse accuracy are equal if the depth of the subtrees is constrained to 1. This is not surprising, as for depth 1, STSG is equivalent with SCFG where every parse is generated by exactly one derivation. What is remarkable, is, that the derivation accuracy decreases if the depth of the subtrees is enlarged to 2. If the depth is enlarged further, the derivation accuracy increases again.

The parse accuracy, on the other hand, increases dramatically when enlarging the depth of the subtrees from 1 to 2. The parse accuracy keeps increasing, at a slower rate, when the depth is enlarged further. The highest (parse) accuracy is obtained by using all subtrees from the training set: 72 out of the 75 sentences from the test set are parsed correctly.

Acknowledgements

The author is indebted to Remko Scha and Martin van den Berg for valuable comments on earlier versions of this paper.

References


K. Church and R. Patil, 1983. *Coping with Syntactic Ambiguity or How to Put the Block in the Box on the Table*, MIT/LCS/TM-216.


A comparison of ALE and PATR; practical experiences

Milena Stefanova  Wilco ter Stal

Department of Computer Science
University of Twente
Postbus 217, 7500 AE Enschede
Email: {stefanov, terstal}@cs.utwente.nl

Abstract

We describe our experiences with two unification-based grammar formalisms, namely ALE and PATR, used for syntactic and semantic analysis of NL texts. These formalisms can be considered to be high-level tools, like expert-system shells, for writing a NL grammar and for parsing sentences.

However, there are a number of aspects, such as ease of expression, augmentability, linguistic felicitousness and computational efficiency, in which ALE and PATR differ.

The experiment is carried out in the context of the Plinius project which requires, a.o. a large sublanguage grammar covering a fragment of English.

We will describe the formalization and implementation of our linguistic ideas in PATR and ALE. Subsequently, we will evaluate both exercises thereby focusing on both practical and theoretical properties of the two formalisms. The purpose of the experiment is to reveal whether there are motivated reasons concerning which kind of formalism can be employed best given the particular Plinius task.

1 Introduction

Currently, there are several unification-based formalisms available which can be put to use for specifying a NL grammar. Examples of such grammar formalisms are PATR, (Gazdar & Mellish [1989]; Shieber [1986]), ALE (Carpenter [1992]), ELU (Estival [1990]) and STUF (Oliva [1991]). A common feature of these formalisms is that no particular linguistic theory is advocated. In other words, these formalisms can be considered high-level tools, like expert-system shells, for writing a grammar. However, there are a number of aspects, such as ease of expression, augmentability, linguistic felicitousness, computational efficiency, in which the aforementioned formalisms differ. Depending on the task at hand one should weight these aspects and make an appropriate choice for one particular formalism. For instance, if a large-scale grammar is to be designed for batch processing of real texts, the flexibility and the augmentability of the formalism seems to be more important than the computational efficiency. At the other hand if one is engaged in the specification of a small grammar for a real-time application, such as a dialogue system, the computational efficiency, i.e. the use of time and memory resources, is of crucial importance.

In this article we will discuss our preliminary experiences with two specific formalisms, namely PATR and ALE. We are experimenting with both formalisms in the context of the Plinius project which requires, a.o., a large sublanguage grammar covering a fragment of English. We will focus on both practical and theoretical properties of the two formalisms. It is not our aim to provide a complete formal, mathematical comparison of the two formalisms.

1.1 Overview of remaining sections

First we will briefly discuss the Plinius project which forms the context of our research. Subsequently, we will have to describe what linguistic 'theory' we would like to formalize in order to make the comparison feasible. In addition we will discuss, in § 2, our engineering approach towards the NLP part of the system and the grammatical information we are using. In § 3 we will show
how this grammatical information is actually encoded in the PATR formalism. We will give an overview of the format of rules and lexical entries and explain the resulting output of parsing a sentence with PATR. In § 4 we will give an short description of ALE and show how the grammatical from § 2 and additional sortal information is specified in ALE. Section 5 contains an evaluation of the two formalisms. In § 6 we will summarize our experiment and point out some directions for further research.

2 The Plinius project

The Plinius project is aimed at developing a system which is capable of semi-automatically extracting domain-specific knowledge from the title and abstract of scientific publications in the field of ceramic materials. The knowledge base resulting from the Plinius project should have economical potential, that is, the benefits of using the knowledge base should outweigh the costs of developing it. In order to attain this goal we have made the following design decisions:

- use of abstracts
- interactive resolution of complex linguistic problems
- limitation to subslanguage
- reuse of (linguistic) resources
- application of an ontology

An ontology consists of a limited vocabulary of unambiguous concepts and their interrelations. The Plinius ontology presents a conceptual framework which structures the domain of ceramics at the knowledge level. Concepts from the Plinius ontology can be employed to express relevant knowledge on ceramic materials, such as their chemical composition, their (mechanical) properties, and the processes to produce them.

For a detailed review of the other design decisions and the Plinius ontology we refer to (Mars et al. [1993]; Vet & Mars [1993]). In the following section we will concentrate on NLP and more specifically on the grammar engineering part within Plinius.

2.1 Grammar engineering in the Plinius project

In this section we will discuss characteristics of the Plinius grammar. Initially, our goal was to construct a grammar by taking an existing broad-coverage grammar and tailoring it to the envisaged need within Plinius. However, large broad-coverage grammars are hardly available and the experiments we carried out with a well-known exception, the grammar of the ALNT project (Briscoe et al. [1989]), yielded unsatisfactory results. Moreover, due to the subslanguage employed in the abstracts we expected to encounter only a limited number of syntactic constructions. Therefore, we decided to develop our grammar in-house.

The second observation concerning the Plinius grammar is that it is rather eclectic. That is to say, that we do not confine ourselves to a particular grammatical theory such as LFG, HPSG or GB. Instead we employ several notions from different theories and implemented grammars, such as the grammar in the ANLT and Pundit (Paramax Systems Corporation [1992]). Examples of such notions are head, modifier, complement and operator.

A third property of our grammar is that we presuppose a close parallelism between the syntax (structural configuration) and the semantic representation of a sentence. However, we will notice a difference in the nature of the semantic representation yielded by PATR in comparison to the representation rendered by ALE.

A final property of our subslanguage grammar is that we would like to integrate so-called selectional or sortal restrictions, see (Alshawi & Carter [1992]), during parsing in order to rule out spurious structural analyses. Structural ambiguities, caused by for instance various potential PP-attachments, are resolved during sentential analysis.

We decided to specify a first version of our grammar in PATR. This decision was motivated by three reasons. The first was that PATR seemed expressive enough for the Plinius purpose. Secondly, the PATR formalism has a wide acceptance amongst linguists and is free to obtain. Thirdly, the SETI group of the University of Twente took interest in developing an efficient parser working with a context-free grammar annotated with features. In order to test and compare the performance of this parser with other parsers it seemed a good idea to express
the grammatical information in a standard formalism, namely PATR.

3 Encoding the Plinius grammar in PATR

3.1 The PATR formalism

The original PATR formalism\(^1\) has been developed at SRI International, Menlo Park, around 1984. However, the formalism rapidly changed into what has become known in the literature as PATR-II. The PATR-II formalism is extensively described in (Shieber [1986]), where it is proposed as a kind of 'lingua franca' to formalize and implement different theories of grammar. From now on we will use the term 'PATR' to denote the PATR-II formalism.

The intuitive idea behind the PATR formalism is that language constructs (sentences, phrases, words) can be associated with information packages. The sort of information associated with the aforementioned language constructs usually concerns syntactic and semantic information. Information packages are notated in so-called feature structures (henceforth: FSs).\(^2\)

Parsing in PATR consists of incrementally unifying the FSs associated with words and phrases according to, or driven by, the rules of the grammar. For a discussion of how the parsing strategy precisely operates on rules and feature structures, we refer to (Verbrugge [1993]).

The grammar rules are in fact just like ordinary context-free phrase structure rules of the form \(A \rightarrow B C\), except for the fact that the nodes of the rules are annotated with constraints in terms of path equations of feature structures. For instance, the rule in figure 1 says that an \(S\) can be built from an \(NP\) and a \(VP\), if the associated constraints are satisfied. Similarly, grammar rules in the Plinius grammar consist of a phrase structure part which says how a particular mother node can be rewritten as a sequence of \(n\) daughter nodes and a bundle of path equations which operate on FSs associated with the nodes at the rule. In figure 2 an example of a rule (in Prolog notation) is given.

\[X1 \rightarrow [X2, X3]\]
\[X1: \text{category} = \text{S}\]
\[X2: \text{category} = \text{NP}\]
\[X3: \text{category} = \text{VP}\]
\[X2: \text{agreement} = X3: \text{agreement}\]

Figure 1: An example of a grammar rule in PATR.

\[X1 \rightarrow [X2, X3]: -\]
\[X1: \text{cat} = \text{S},\]
\[X2: \text{cat} = \text{NP},\]
\[X3: \text{cat} = \text{VP},\]
\[X3: \text{syn\_sem} : \text{head} : \text{voice} = \text{active},\]
\[X1: \text{syn\_sem} : \text{operator} = \text{null},\]
\[X1: \text{syn\_sem} = X3: \text{syn\_sem},\]
\[X1: \text{syn\_sem} : \text{args} : \text{subject} = X2: \text{syn\_sem},\]
\[X3: \text{arg1} = X1: \text{syn\_sem} : \text{content} = \text{arg0},\]
\[X1: \text{syn\_sem} : \text{mod} = \text{null}.\]

Figure 2: An example of a grammar rule in the Plinius grammar.

In the next subsection we will briefly describe which features are actually used in the Plinius grammar.

3.2 Feature structures for lexical and phrasal categories in Plinius

The choice for features in the Plinius grammar is organized in such a way that the result of a successful parse of a sentence yields (1) a conventional parse tree and (2) a FS describing syntactic and semantic information.\(^3\) In what follows we intend to make clear how the syntactic and semantic information in Plinius is notated in terms of FSs. First, we will give the general format of the information associated with lexical categories (words), and then we will explain the format of phrasal categories. Finally, we give some examples of FSs of (parts of) sentences taken from an abstract text.

\(^1\)PATR is an acronym for 'parsing and translate'.
\(^2\)Also called Attribute-Value Matrices (AVMs), throughout this paper we will use the term feature structures.
\(^3\)Note that when a sentence is syntactically ambiguous the parser will give several FSs (and parse trees) per sentence, unless of course the grammar and lexicon employ selection restrictions in order to rule out spurious parses.
3.2.1 Lexical categories

The overall format of the FSs associated with the lexical entries is the following:

\[
\text{word} \Rightarrow \left[ \begin{array}{c}
\text{cat} : \ldots \\
\text{syn.sem} : \left[ \begin{array}{c}
\text{head} : \ldots \\
\text{content} : \ldots \\
\end{array} \right] \\
\end{array} \right]
\]

Thus each lexical entry in Plinius consists of a feature \text{cat} with values such as \text{noun}, \text{verb}, \text{aux} etc. This feature is used to construct the parse tree based on the nodes of the context-free skeleton of the grammar. The motivation for generating a parse tree stems for the fact that trees are much easier to inspect. The feature \text{syn.sem} consists of a bundle other features namely \text{head} and \text{content} which should be completed with syntactic and semantic information respectively. The dots after features indicate particular values which can of course differ per word. For instance, the lexical entries for the words \text{materials}, \text{dense} and \text{exhibits} amount to:

\[
\text{materials} \Rightarrow \left[ \begin{array}{c}
\text{cat} : \text{noun} \\
\text{syn.sem} : \left[ \begin{array}{c}
\text{head} : \left[ \text{agr} : \left[ \text{num: plur} \right] \right] \\
\text{content} : \left[ \text{relation: MATERIAL} \right] \\
\end{array} \right] \\
\end{array} \right]
\]

\[
\text{dense} \Rightarrow \left[ \begin{array}{c}
\text{cat} : \text{adj} \\
\text{syn.sem} : \left[ \begin{array}{c}
\text{head} : \left[ \text{qua} : \text{=} \right] \\
\text{content} : \left[ \text{relation: DENSE} \right] \\
\end{array} \right] \\
\end{array} \right]
\]

\[
\text{exhibits} \Rightarrow \left[ \begin{array}{c}
\text{cat} : \text{verb} \\
\text{syn.sem} : \left[ \begin{array}{c}
\text{head} : \left[ \text{agr} : \left[ \text{num: sing} \right], \text{per} : 3 \right] \\
\text{vform} : \text{fin}, \text{tense} : \text{present} \\
\text{relation} : \text{EXHIBIT.1} \\
\text{content} : \left[ \begin{array}{c}
\text{arg0} : \text{el} \\
\text{arg1} : \text{x} \\
\text{arg2} : \text{y} \\
\end{array} \right] \\
\end{array} \right] \\
\end{array} \right]
\]

3.2.2 Phrasal categories

In our grammar a phrasal category corresponds with a traditional grammatical category such as NP, AP, VP and PP. The general FS format of a phrasal category is:

\[
\text{phrase} \Rightarrow \left[ \begin{array}{c}
\text{syn.sem} : \left[ \begin{array}{c}
\text{operator} : \ldots \\
\text{head} : \ldots \\
\text{args} : \ldots \\
\text{mod} : \ldots \\
\text{content} : \ldots \\
\end{array} \right] \\
\end{array} \right]
\]

We will shortly discuss the function of these features.

The \text{operator} feature

The purpose of the \text{syn.sem|operator} feature is to provide a placeholder for linguistic expressions which function as some kind of operator. Examples of such expressions are \text{quantifiers} and \text{co-ordinators}. The idea is that these expressions operate on the syntactic and semantic content of a phrasal category and should be marked.

The \text{head} feature

The \text{syn.sem|head} feature again contains an FS representing syntactic information concerning the \text{head} of a phrase. The \text{syn.sem|content} feature contains the semantic information of the head of the phrase.

Informally, the head of a phrase coincides with the most prominent word in the phrase. For instance, in an NP the head usually is the noun, in a VP the main verb usually constitutes the head. A number of syntactic and semantic properties of an entire phrase are derived from the head of the phrase. Examples of these properties are: \text{number}, \text{tense}, \text{predicate/argument structure}.

In the Plinius grammar the \text{head} feature can have several values specific to the different categories. For instance, it does not make sense to talk about the tense of an adjectival phrase, hence the feature \text{tense} will not occur as a value in the \text{head} feature of an AP.

The \text{args} feature

The \text{syn.sem|args} feature of a phrase can be instantiated with the so-called \text{complements} of a phrase. This can be best illustrated by means of an example. Take for instance the verb \text{exhibits}. This transitive verb requires a subject-NP and an object-NP. In case a sentence like (1) is encountered, the grammar constructs an FS equal

\[\text{A path or a path equation of a FS is written as feature|feature or feature|feature|feature|feature|feature|value.}\]
(1) Materials exhibited elongation

\[
\begin{align*}
\text{operator} : & \ldots \\
\text{head} : & \begin{cases} 
  \text{vform} : \text{fin} \\
  \text{voice} : \text{active} \\
  \text{tense} : \text{past}
\end{cases} \\
\text{syn} \text{.sem} : & \\
\text{args} : & \begin{cases} 
  \text{subject} : \text{NP}1 \\
  \text{object} : \text{NP}2
\end{cases} \\
\text{mod} : & \ldots \\
\text{content} : & \begin{cases} 
  \text{relation} : \text{EXHIBIT.1} \\
  \text{arg}0 : e1 \\
  \text{arg}1 : z1 \\
  \text{arg}2 : z2
\end{cases}
\end{align*}
\]

Note that the path equations \text{syn} \text{.sem}[\text{args}][\text{subject}] == \text{NP}1 and \text{syn} \text{.sem}[\text{args}][\text{object}] == \text{NP}2 are abbreviations. Parsing the whole sentence will yield an FS where the syn sem features for the NPs materials and elongation are substituted for \text{NP}1 and \text{NP}2.

The mod feature
The mod feature is a storage place for those constituents which are modifiers of the head. For instance, the syn sem features of the modifying PP in a test in sentence (2) are stored in the mod feature of (2').

(2) Materials exhibited elongation in a test

\[
\begin{align*}
\text{operator} : & \ldots \\
\text{head} : & \begin{cases} 
  \text{vform} : \text{fin} \\
  \text{voice} : \text{active} \\
  \text{tense} : \text{past}
\end{cases} \\
\text{args} : & \ldots \\
\text{mod} : & \begin{cases} 
  \text{relation} : \text{EXHIBIT.1} \\
  \text{arg}0 : e1 \\
  \text{arg}1 : z1 \\
  \text{arg}2 : z2
\end{cases}
\end{align*}
\]

The content feature
The semantic translation of most nouns, verbs, adjectives and prepositions, given in terms of predicate-argument structures, is expressed in the content feature.

An important aspect of (2'), which holds for FSs in general, is the fact that parts of an FS can be structure shared with another FS. In particular we make use of this property in case of the content feature. In case of the FS (2') this is illustrated by the fact that the first argument of the EXHIBIT.1 relation, e1, functions as a parameter used as the second argument in the IN.1 relation.

The values of the content features in the FS per sentence, are used to construct a so-called Quasi Logical Form (QLF). The idea of QLFs, as described in (Eijck & Alshawi [1992]; Sloat & Rentinier [1993]; van der Vet et al. [1993]), is that they form a suitable datastructure for storing grammatical information relevant to further semantic and discourse processing. That is to say that QLFs form the input for an additional process which maps expressions in QLF onto the final representation in terms of the Plinius ontology.

An example of a QLF for sentence (2) is given in (2''). Note that the QLF in fact is a linear notation of the collected content features in (2').

\[
(2'') \ [\text{MATERIAL}(x1) \ & \ \text{ELONGATION}(x2) \ & \ \text{EXHIBIT.1}(e1, x1, x2) \ & \ \text{TEST}(x3) \ & \ \text{IN.1}(e1, x3)]
\]

4 Encoding the Plinius grammar in ALE

In this section we describe relevant properties of the Attribute Logic Engine (ALE) and our experiences with developing a grammar in ALE. The version of ALE we are using runs under Quintus Prolog.
4.1 Description of the Attribute Logic Engine

ALE combines properties of three types of grammar and knowledge representation formalisms, namely, unification-based formalisms such as PATR and STUF, KL-ONE-like systems such as CLASSIC and Definite Clause Grammars.

ALE allows to model linguistic knowledge and knowledge about a certain domain as two separate parts in a single formalism. In fact, in our ALE grammar we are able to distinguish between two ontologies - one, describing concepts and relations in the Plinius domain, and another one describing the domain of linguistic knowledge. Elements of the linguistic domain are abstract entities like grammatical categories, inflexions of verbs, agreement, etc. In this fashion, a complex domain consisting of two subdomains can be specified in ALE as depicted in figure 3.

Elements of a domain are modeled by so-called typed feature terms (or descriptions). Every typed feature term denotes a set of typed feature graphs in a feature graph algebra, which coincide with the abstract objects of the linguistic domain and the objects of the Plinius ontology.

An advantage of this approach is that one might not only describe in the lexical entries syntax and syntax-driven (compositional) semantics, but also the actual meaning of the words in terms of the non-linguistic ontology. In such a way we are able to combine the benefits of both ontology-driven and syntax-driven semantics for representing sentence (and text) meaning.

The main properties of ALE can be summarized as follows. Firstly, ALE is a classification oriented formalism, which has much in common with systems for terminological knowledge representation. Types used in ALE play the same role as concepts used in KL-ONE-like systems for representing knowledge about a certain domain. As is well-known, see Brachman et al. [1991], these systems consist of two parts, namely a terminological part, or T-Box, and an assertional part, or A-Box. In a T-Box concepts about a certain domain are defined. In the assertional part we can define the objects of the domain under consideration in terms of the T-Box. Two important operations are employed in such systems namely concept and object classification. Concept classification computes the hierarchy (or subsumption preorder) of the concepts, and object classification performs pattern matching between components of the A-Box and proper concepts of the T-Box.

The state of affairs in ALE is similar. The type system in ALE corresponds to the T-Box, and the lexicon and the grammar rules - to the A-Box of a KL-ONE system. Every type is defined as a subtype of another type, and, optionally, a set of features with possible values are listed.

Assertions in ALE can be statical or dynamical. Statical assertions are in the lexicon, whereas dynamical assertions are made in the grammar rules.

The operation of concept classification is performed during compile time when the full type hierarchy is computed. There is an additional inference operation which is similar to object classification.

The second property of ALE is that it is a unification-based formalism. An ALE grammar is a context-free grammar in which feature graphs coincide with non-terminals and where natural language words are considered to be terminals. The most important operation is unification which is necessary when structure sharing is indicated in a feature graph. What differs ALE from the PATR is that feature structures employed in ALE are well-typed, see (Carpenter [1982]). This means that every feature structure is of a certain type and that every feature appropriate for this type must be present and must have a proper value. Using typed feature structures increases significantly time and memory ef
ficiency of the parsing algorithm, see Carpenter [1993].

The third characteristic of ALE is that definite clause programming can be integrated into an ALE grammar. That is to say that Prolog-like goals can be added to grammar rules. However, instead of first-order logic terms only feature terms may be used. Conditions under which a certain goal is true are specified as definite clauses in the same way as in Prolog. Having these procedural attachments, arithmetics, processing of lists and other recursive data operations can be easily incorporated into the parsing process.

For a more detailed overview of ALE the reader is referred to (Carpenter [1992]).

4.2 Implementing the Plinius grammar in ALE

4.2.1 Format of an ALE grammar

In an ALE grammar one can distinguish three main components, namely the type hierarchy, the lexicon, and the grammar rules. Additional components, such as lexical rules and definite clauses, can be used to facilitate the writing of the lexicon and the grammar rules, respectively. We will explain briefly the format of the three main components in an ALE grammar.

The first thing to be specified is the type hierarchy. In order to do this we need two finite sets TYPE and FEAT of types and features respectively. In fact, the type hierarchy can be specified by defining two relations Sub \( \subseteq \) TYPE \( \times \) TYPE and Intro \( \subseteq \) TYPE \( \times \) FEAT \( \times \) TYPE. Type specifications have the following format\(^8\):

\[
\begin{align*}
a &\text{ sub } [a_1, a_2, \ldots, a_n], \\
intro &\text{ [f}_1:b_1, f_2:b_2, \ldots, f_k:b_k].
\end{align*}
\]

There are several restrictions on the relations Sub and Intro, see Carpenter [1991], so that these relations meet the conditions appropriate for inheritance hierarchies and totally well typed systems. In § 4.2.2 we give an example of a part of the type hierarchy for Plinius.

Besides the type hierarchy, we have to specify lexical entries. Lexical entries are notated in the following format:

\[
<\text{word}> \Rightarrow D.
\]

\(^8\) \(a, a_1, a_2, \ldots, a_n; b_1, b_2, \ldots, b_k\) are types and \(f_1, f_2, \ldots, f_k\) are features.

Here <word> is a natural language word and D is a feature term. Note that it is not necessary to describe lexical entries fully in terms of the type hierarchy. One may specify only the features and types known for the moment, the missing types and most general features are inferred by the compiler. Additionally, the lexicon developer is able to define macros. These macros are abbreviations for complex feature structures and are expanded during compile time. For instance a word like car can be specified in the lexicon as (3) and amounts to (3') after compilation.

\[\text{(3) car } \Rightarrow \text{(syn: (cat: noun, agr: @agreement(3, sg))} )\]

\[\text{(3') sign}
\text{ SYN: syntax}
\text{ CAT: noun}
\text{ AGR: agreement}
\text{ PERSON: 3}
\text{ NUMBER: sg}
\text{ SUBCAT: list_sign}\]

In the remaining sections of the paper we will use the following orthographic convention. Features are written in uppercase letters, types are written in lowercase letters.

The format of grammar rules is as follows:

\[<\text{rule name}> \text{ rule (D)} \Rightarrow \]

\[\text{cat} > \text{(D}_1), \]
\[\text{cat} > \text{(D}_2), \]
\[\vdots\]
\[\text{cat} > \text{(D}_n), \]
\[\text{goal} > G_1, G_2, \ldots, G_k.\]

Here D, D1, D2, ..., Dn are feature terms and G1, G2, ..., Gk are Prolog-like goals in which instead of first-order logic terms, feature terms are employed. Note that adding these goals is not obligatory for writing a grammar rule. In section 4.2.4. we give an example of a grammar rule in ALE.

4.2.2 Specifying the PLINIUS type hierarchy in ALE

In order to implement the PLINIUS grammar in ALE we first have to specify the type hierarchy. As was mentioned in § 4.1, in the type specification we will define two ontologies - one for the linguistic knowledge and one for the knowledge of
the domain of material science, based on Vet & Mars [1993], van der Vet & Mars [1993].

For specifying the linguistic ontology, we will use more ore less the same basic notions as described in § 2.1.

Natural language words and phrases all belong to the concept of sign or in terms of ALE, feature structures (FSs) representing natural language words and phrases are all of type sign. As in HPSG (Pollard & Sag [1987]), all signs have a feature SYNSEM defined on them. Words and phrases are distinguished by adding subtypes x, x1, x2 to the type sign, rather than by adding an additional feature DAUGHTERS to phrasal signs. This decision is made in order to simplify the syntactic information captured in FSs.10

The syntactic and semantic contents of a sign are described by a FS of type synsem, see figure 2, which in fact is a value of the feature SYNSEM. Syntactic information is stored as a value of the CATEGORY feature of the synsem. This feature contains information concerning head and subcategorization possibilities of a sign. Note that the type hierarchy reflects syntactic principles, borrowed from HPSG (Pollard & Sag [1987]), such as the subcategorization principle, the head-principle and principles determining a certain phrase to be of type x, x1 or x2.

Semantic information is described as a FS which is the value of the CONTENT feature. This FS is of type sem_obj which has several subtypes, as displayed in figure 4.

The main subtype is property. Semantic representations of common nouns, verbs, adjectives, prepositions are of type property. Every FS of type property has the following features defined on it: PARA (from parameter) standing for a particular world object having this property, RESTR (from restrictions) which value has a number of relations in which the parameter takes place, and the feature DET standing for a certain quantifier for every phrase.11 In fact FSs of type sem_obj encode in a proper way the representations which are the intended output of the grammar.

There is a subtype world of the type sem_obj which represent the domain under consideration (in our case the domain of material science). The type world has a number of subtypes standing for the concepts of the ontology. A graphical representation of this type hierarchy is given in fig. 7. An overview of the linguistic type hierarchy is given in fig. 6, for both figures the reader is referred to the appendices.

In the next section we will explain briefly the specification of the Plinian lexicon in ALE.

4.2.3 Lexical entries in ALE

We will give a few examples of some Plinian lexical entries in ALE, in order to clarify the relation between the linguistic and non-linguistic ontology.

Feature structures for lexical entries for the substantive parts of speech (verbs, nouns, etc.) amount, after compilation time, to:

\[
\text{synsem}
\]
\[
\begin{align*}
\text{CATEGORY:} & \quad \begin{cases} 
\text{head:} \text{parts. of speech} \\
\text{SUBCAT:} \text{list. synsem} 
\end{cases} \\
\text{CONTENT:} & \quad \begin{cases} 
\text{proper}: \text{determiner} \\
\text{PARA:} \text{world} \\
\text{RESTR:} \text{non. empty. set} 
\end{cases}
\end{align*}
\]

Note that the parameter, PARA is of a type which is a subtype of world, i.e., it is a FS representing a certain concept of the ontology. For example, the value of the PARA for the word temperature can be the following FS:

\[
\begin{align*}
\text{quantity} \\
\text{HAS.NAME:} \text{temperature} \\
\text{HAS.UNIT:} \text{kelvin} \\
\text{HAS.VALUE:} \text{value}
\end{align*}
\]

Using such kind of structured parameters allows

\[11\]

There is a definite determiner \text{dx} which combines with verbs.

\[9\]
Corresponding to the bar-levels in the \text{x}-scheme in GB.

\[10\]
The syntactic information about the constituents of a phrase is not directly relevant for the ultimate goal of Plinian.
to point exact links among the parameters of a relation, i.e., to give a detailed semantics to this relation. In case of verbs this can be illustrated as follows. The parameter of a verb is taken to be an event which may occur in the restricted world under consideration. The parameters of the subcategorization phrases of a verb should fill certain slots in the verb parameter. In such a way the exact relation between verb and its arguments is established. An example of the CONTENT-feature of the verb exhibits is given below:

```
property
   DET: determiner
   PARA: [1] having material property
         MATERIAL: [2] material
         PROPERTY: [3] material property
         non_empty_set
   REST: [4] REST
```

When the verb exhibits is combined with its arguments from the subcategorization list, the values of the features, ARG1 and ARG2, are filled with more specific information. Specifically, they can have values which are subtypes of the types material and material property respectively. In such a way the parser is able to construct a semantic representation of the sentence in terms of the non-linguistic ontology.

4.2.4 Grammar rules in ALE

Grammar rules have the format described in 4.2.1. The main difference which distinguishes ALE rules from PATR rules, i.e., grammar rules augmented with feature-value pairs, is that additional Prolog-like goals can be incorporated.

Having this possibility one can (partially) solve problems concerning the semantic representation of modifiers, plurals, coordination, and negation (see § 5.1), as well as cases in which real numbers are used as values of certain quantities.

For instance, the semantic principle behind the role of modifiers is that modifiers place additional restrictions on the parameter of the modified phrase. One can easily see that this principle may be implemented by the following rule for prepositional postmodifiers:

```
x1_x1_pp rule
(x1,
   synsem:(category:X1_category,
            content:(det:X1_determiner,
                      para:X1_para,
                      restr:L))
   ==> cat>(x1,
            synsem:(category:X1_category,
                    content:(det:X1_determiner,
                              para:X1_para,
                              restr:L1)),
            cat>(x2,
                synsem:(category:(head:(prep,
                                predicative:yes),
                               subcat:NULL),
                        content:(det:X1_det,
                                 para:X1_para,
                                 restr:L2))))
```

goal> append(L1,L2,L).

Definite clauses may also be used to deal with plural nouns. One would like to express that plural nouns represent some kind of set. In order to obtain an appropriate representation one can employ instead of single-value parameters, parameters with list-values. Then, in the lexicon one can specify that singular nouns have a parameter list with exactly one element. Plural nouns are as-
sociated with a parameter list containing at least two elements. In case a plural noun is encountered in phrases like three materials, the following goal can be added to the rule for recognizing this NP:

\[
\text{goal} \rightarrow \text{length}(X,Y).
\]

where \( X \) is the parameter of the word materials, which is a list, and \( Y \) is a natural number representing the parameter of three, in this case \( Y = 3 \).\(^{12}\)

5 Evaluation of ALE and PATR

In this section we will discuss and compare different aspects of the two formalism. We will first focus on the theoretical aspects of the two formalisms and then discuss our practical experiences.

5.1 Theoretical issues

Underlying feature algebra

We already mentioned that the interpretation of the feature terms of ALE is given in a feature graph algebra. In fact the logic of ALE is a simplified form of feature logic, similar to the logic described in (Smolka [1988]). A special property of this feature logic is that the feature graphs employed for interpretation are totally well typed, since they conform to a number of appropriateness and type hierarchy conditions, see (Carpenter [1991]).

In case of PATR it is possible to give an interpretation in terms of a feature graph algebra, but these feature graphs will be unttyped.

Disjunction and negation

In ALE it is possible to specify disjunctive and negative information. In the following example we show how negation can be modeled by using definite clause programming. Suppose that we would like to state that the PERSON feature of agreement is not 3. Then, one can use the following rule:

\[
\text{xx rule}
\]

\[
\ldots
\]

\[\Rightarrow \text{cat} \ldots
\]

\[\text{cat} \in \text{syntex:content:agr:} X, \]

\[\text{cat} \ldots
\]

\[\text{goal} \rightarrow \text{not_third}(X).^{13}\]

\[\text{not_third}(1).
\]

\[\text{not_third}(2).
\]

In PATR it is not possible to specify negative or disjunctive information.

Explicit versus Implicit typing

An advantage of ALE is that, due to explicit typing of FSs, errors are detected already at compile time. That is to say that most of the work of unification is done in advance, i.e. before parsing a sentence.

PATR employs so-called *implicit typing*. The compiler assigns a type to every FS and in case this is not possible the FS is not consistent with previous FSs. Unfortunately, this error can only be discovered during parsing.

Singleton types

There are no singleton types in the logic of ALE. That means that structure sharing has to be indicated where it is necessary to point out that two FSs are identical, even in case where these FSs are intended to be just singleton types.

In PATR singleton types coincide with atoms.

5.2 Practical experiences

Simplicity of the formalism

It may be obvious from the description of ALE that writing a grammar in ALE is rather laborious. Especially, the type hierarchy forces a grammar writer to be very precise in choosing features. Fortunately, the formalism is *lexical oriented* which means that the number of rules remains limited and that most of the grammatical information is specified at the lexical entries.

The learning curve of PATR is much lower and the less rigid use of features allows the developer to create a grammar much faster.

Correcting and augmenting the grammar

In ALE corrections should be made on a global level, i.e. in the type hierarchy. PATR allows a more flexible and local view on grammar rules. Corrections can be implemented directly in a rule, as long as the feature perlocations remain intact.

\[\text{Or (NOT third(k))}, \text{where third(3)}.\]

\(^{12}\)This is not implemented yet since the current ALE version does not allow the use of arithmetics and numerical values. However, this will be possible in the new version of ALE (Carpenter, p.c).

\(^{13}\)Or (NOT third(k)), where third(3).
As far as concerns augmenting both grammars we can not make any conclusive remarks yet. Constructions occurring in our Plinius corpus that should be incorporated in both formalisms are a.o.: Relative clauses, Comparative constructions, Partitive NPs and Coordinated phrases. Our intuitions concerning the implementation of these linguistic phenomena is that the PATR framework provides a better point of departure.

Incorporating selections restrictions
In § 2.1. we stated that we would like to employ selectional restrictions in order to resolve structural ambiguities. It is straightforward that the strong typing used in ALE can be applied for this task. For instance, the structured parameters, as presented in § 4.2.3, prevent the combination of certain modifiers and modified phrases.

In PATR we could obtain the same effect by specifying an additional feature TYPE at lexical entries and by writing extra path equations at a grammar rule. In these constraints we would check, for instance, whether the TYPE of a modifier is compatible with the TYPE of a modified phrase. It is clear that this is a rather cumbersome solution, compared to the built-in type system and type inferencing mechanism in ALE.

Nature of output representation
We have shown that ALE allows to integrate non-linguistic knowledge into the process of parsing. Every meaningful word is linked to a proper concept (type) of the Plinius ontology. One of the consequences of this aspect of ALE is that parsing a sentence yields directly a semantic representation in terms of the Plinius ontology. Further discourse processing can operate upon these output FSs and results in the envisaged output of the Plinius project.

The current version of the Plinius grammar in PATR is implemented in a such a way that generated semantic representations are underspecified. After the parser delivers a QLF we have to design an additional process which links the intermediate representation to the final representational structures of the ontology.

Coverage of the grammars
The coverage of the grammar in ALE is rather limited, we experimented with a grammar which could handle; simple active sentences, VP complement structures, VPs with postmodifiers, NPs with adjectival and prepositional modifiers.

The grammar of PATR has a wider scope, it includes the structures described above extended with: Passives, Coordinated APs, Numerical APs, Coordinated PP, Infinitival constructions, simple Comparatives and Compounds.

Performance
Unfortunately, we can not (yet) provide any detailed figures regarding the parsing times of both formalisms. The experiences up until now with parsing sentences in both formalisms are very good. Note that in ALE most of the time is taken by compiling the type hierarchy (Carpenter [1993]), once this process is completed parsing a sentence of 10 words takes less than a second.

For details regarding time and memory efficiency of the head-corner parser which operates on the PATR rules we refer to Verlinden [1993].

6 Discussion and further research
We gave an overview of our experiences with coding a grammar in two unification-based formalisms, namely PATR and ALE. It is clear that the formalisms demonstrate both advantages and disadvantages. A major advantage of PATR is its flexibility which is useful for developing a grammar rapidly. A disadvantage of PATR is that it appears to be not very suitable for semantic processing of sentences. For instance, the incorporation of semantic (domain) knowledge in the same formalism, in order to prevent ambiguities and to generate an appropriate semantic structure, is not straightforward. Contrary to PATR it is very well possible in ALE to link domain knowledge with the process of parsing. A consequence of this aspect of ALE is that knowledge and grammar engineers need to collaborate closely.

Currently, we are not able to draw any firm conclusions regarding the choice in favor of one particular formalism. This is mainly due to lack of profound experience with constructing a real-size grammar in ALE. However, we plan to proceed with (the new version) of ALE. In addition, we would like to investigate the idea of combining 'best of both worlds'. In practice, this amounts to parsing sentences with PATR into QLFs, see § 3.2.2, and translating these QLFs into typed FSs in ALE. Inferencing, for instance for discourse purposes, will then takes place in ALE which is more suitable for this task.

Another option is to investigate whether the PLEUK environment (Calder & Humphreys
[1993]) offers valuable support for the task of specifying the Plinius grammar. Ultimately, the usefulness of these experiments can only be estimated on the basis of the rapid development of a grammar with a sufficient coverage and appropriate output.

Acknowledgment
We would like to thank Hidde de Jong for proofreading a draft version of this paper.

References


G. Gazdar & C. Mellish [1988], Natural Language Processing in Prolog: an introduction to computational linguistics, Addison-Wesley, Wokingham.


Stuart Shieber [1986], “An introduction to Unification-Based Approaches to Grammar,” CSLI Lecture Notes No. 4, Stanford, CA.

KO van der Sloop & Gerrit Rentier [1993], “The PLUS Grammar,” ITK, Esprit P5254, Tilburg.


Margriet Verlinden [1993], “A head-corner parser for unification grammars,” in Proceedings of the TWLT 6, Natural Language Parsing: Methods and Formalisms, Anton Nijholt & Klaas Sikkel, eds., University of Twente, Enschede, the Netherlands, this volume.

Paul E. van der Vet & Nicolaas J.J. Mars[1993],

"Plinius intermediate report," University of Twente, Memorandum UT-KBS-93-36, Enschede, the Netherlands.
Figure 6: Graphical representation of linguistic type hierarchy of Plinius as implemented in ALE.
Figure 7: Graphical representation of non-linguistic type hierarchy of Planus as implemented in ALE.
A Practical Comparison between Parallel Tabular Recognizers

J.P.M. de Vreught
Delft University of Technology, Faculty of Applied Physics*
Lorentzweg 1, 2628 CJ Delft, The Netherlands
J.P.M.deVreught@CP.TN.TUDelft.NL (Hans de Vreught)

Abstract

The parallel versions of the CYK, Earley’s, and the double dotted algorithm are tested with the third Tomita grammar and sentence set. The algorithms ran in two versions: a standard version and a version that filtered out bad items that didn’t fit in the context. Although it isn’t possible to draw statistical sound conclusions from such a specialized test, the results in this paper suggest that filtering out bad items is more effective than running the algorithms in parallel.

1 Introduction

In this paper we will make a practical comparison between three parallel tabular recognizers. All three algorithms run in $O(n)$ time with $O(n^2)$ processors on a CRCW PRAM using $O(n^2)$ space. The algorithms use upper triangular matrices to encode partial parses by means of items. Items are modified production rules that give information over what part of the production rule already has been processed.

We will start this paper with the preliminaries in section 2 and the three algorithms in section 3. In section 4 we will describe how we have made the measurements and we will present the results. In section 5 these results are analyzed in further detail. Section 6 ends the paper with the conclusions that can be drawn from this comparison.

2 Preliminaries

In the formal part of this paper we will describe a context-free grammar by $G = (V, \Sigma, P, S)$, where $V$ is the vocabulary, $\Sigma$ is the terminal alphabet, $P$ are the production rules, and $S$ is the starting symbol. In derivations we will denote by $\alpha\beta\gamma \Rightarrow_\beta \alpha\delta\gamma$ that only $\beta$ is allowed to be rewritten, thus $\beta \Rightarrow \delta$. Furthermore, we will assume that $\alpha_1...\alpha_m$ is the input string and the empty string is denoted by $\lambda$.

We will run two versions of the three algorithms that follow in the next section. The standard version is straightforward but in the specialized version bad items are pruned from being entered into the matrices. The filtering technique used will be driven by looking at the context of the part of the string already recognized. The testing of whether an item is good or bad will be quite simple by means of an $O(1)$ table look up.

To compute the table we precompute the following functions for certain string values (this depends solely on the grammar). The functions compute a ‘lookaside’ (i.e. a ‘lookback’ and a ‘lookahead’) similar to the well known lookahead functions used in compiler design (see the ‘dragon’ books [ASU66, AU77]).

Definition 2.1 We define the following functions (with left marker $[$ and right marker ] not in $V$):

- $\text{First}(\alpha) = \{ w \in \Sigma \cup \{\} | [S] \Rightarrow_{\alpha}^* \beta \alpha \gamma \Rightarrow_{\alpha}^* \beta \omega \delta \text{ for some } \alpha, \beta, \gamma, \text{ and } \delta \in (V \cup \{[,]\})^* \}$
- $\text{Last}(\alpha) = \{ w \in \Sigma \cup \{\} | [S] \Rightarrow_{\alpha}^* \beta \alpha \gamma \Rightarrow_{\delta}^* \delta \omega \gamma \text{ for some } \alpha, \beta, \gamma, \text{ and } \delta \in (V \cup \{[,]\})^* \}$
- $\text{Precede}(\alpha) = \{ w \in \Sigma \cup \{\} | [S] \Rightarrow_{\alpha}^* \beta \alpha \gamma \Rightarrow_{\beta}^* \delta \omega \gamma \text{ for some } \alpha, \beta, \gamma, \text{ and } \delta \in (V \cup \{[,]\})^* \}$
- $\text{Follow}(\alpha) = \{ w \in \Sigma \cup \{\} | [S] \Rightarrow_{\alpha}^* \beta \alpha \gamma \Rightarrow_{\gamma}^* \beta \omega \delta \text{ for some } \alpha, \beta, \gamma, \text{ and } \delta \in (V \cup \{[,]\})^* \}$

Let $\text{Name}$ stand for $\text{First}$ or $\text{Last}$:

$$\text{Name}(X) = \bigcup_{\alpha \in X} \text{Name}(\alpha)$$
Where necessary we will assume \( a_0 = [ \) and \( a_{n+1} = ] \).

Although we could use the formal definition of a context-free grammar to describe a natural language grammar, this is inconvenient: if we would include all words found in a natural language into the grammar, the grammar would be gigantic. Therefore it is more convenient to only use categorial classes to where the words belong to. Therefore we will describe a natural language grammar by

- \( G = (V, \Sigma, P, S) \) where \( \Sigma \) is a finite set of word classes,
- a dictionary which defines the classes of all words and also to which set of categories \( c \subseteq \Sigma \) a word \( w \) belongs.

The algorithms will not run on the input sentence \( w_1 \ldots w_m \), but on the categorial description \( c_1 \ldots c_n \). The modifications to the algorithms such that they are able to cope with such a grammar and input are only minor and will be tacitly assumed to be have done.

3 The Algorithms

We will use three parallel algorithms for our comparison:

- the CYK algorithm (section 3.1),
- Earley’s algorithm (section 3.2),
- the double dotted algorithm (section 3.3).

The CYK algorithm is the only one that requires a grammar in CNF. For just that algorithm we transform our input grammar in CNF by using the method described in [Har78]. For the other algorithms we leave the grammar unchanged.

3.1 The CYK Algorithm

The parallel CYK algorithm is a parallelized version of the Cocke-Younger-Kasami algorithm for grammars in CNF. The CYK algorithm was originally described in [Hay62, You67, Kas65]. The algorithm can be characterized by the following relation.

Definition 3.1 We will define the relation \( \text{CYK} \subseteq \{0, \ldots, n\}^2 \times (V \setminus \Sigma) \) as follows:

- If \( A \to a_j \in P \) then \( \langle j - 1, j, A \rangle \) is in \( \text{CYK} \) for any \( j \in \{1, \ldots, n\} \).
- If \( A \to BC \in P \) and \( \langle i, k, B \rangle \) and \( \langle k, j, C \rangle \) are in \( \text{CYK} \) then \( \langle i, j, A \rangle \) is in \( \text{CYK} \).
- Nothing is in \( \text{CYK} \) except those elements which must be in \( \text{CYK} \) by applying the preceding rules finitely often.

The item \( \langle i, j, A \rangle \) of the second rule is said to be obtained by means of a combination operation (see figure 1).

![Figure 1: Parallel CYK operation](image)

The relation \( \text{CYK} \) is called a partial parse relation. It can be shown that \( \text{CYK} = \{(i, j, A)| A \Rightarrow^* a_i \ldots a_j \} \) and thus \( S \Rightarrow^* a_1 \ldots a_n \) iff \( \langle 0, n, S \rangle \) is in \( \text{CYK} \).

In the algorithm we will use a set \( C \) to define a context filter. For the standard version of the following algorithm we will assume \( C = \{0, \ldots, n\}^2 \times (V \setminus \Sigma) \). For this choice of \( C \) the test whether the items are in \( C \) is trivially true. It can be proved that for this choice of \( C \) the computed \( \text{CYK} \) is equal to \( \text{CYK} \).

We can also choose an alternative definition for \( C \). We will restrict \( C \) such that only items that fit the context are allowed: \( \langle i, j, A \rangle \) fits the context iff \( a_i \in \text{Precede}(A) \) and \( a_{j+1} \in \text{Follow}(A) \). For this choice of \( C \) we call the computed \( \text{CYK} = \text{CYK}_{1,1} \).

**CYK-Recognizer**

\[
\text{CYK}' := \emptyset
\]

for \( i := 1 \) to \( n \) do

for each \( j := 0 \) to \( n - i \) in parallel do

if \( i = 1 \) then

\[
\text{CYK}' := \text{CYK}' \cup \{(j, j + i, A) \in C|A \Rightarrow a_{i+1} \in P\}
\]

else

for each \( k := 1 \) to \( i - 1 \) in parallel do

\[
\text{CYK}' := \text{CYK}' \cup \{(j, j + i, A) \in C|A \Rightarrow BC \in P\}
\]
and \((j,j + k, B) \in \text{CYK}'\)
and \((j + k, j + i, C) \in \text{CYK}'\).

return whether or not \((0, n, S) \in \text{CYK}'\)

3.2 Earley’s Algorithm

The parallel Earley algorithm is a parallelized version of Earley’s algorithm. Earley’s algorithm was originally described in [Ear68, Ear70]. The parallel algorithm can be characterized by the following partial parse relation.

**Definition 3.2** The relation \(E \subseteq \{0, \ldots, n\}^2 \times \{A \rightarrow \alpha \cdot B | A \rightarrow \alpha \cdot B \in P\}\) is defined as follows:
- If \(A \rightarrow \alpha \in P\) then \((j, j, A \rightarrow \cdot \alpha) \in E\) for \(j \in \{0, \ldots, n\}\).
- If \((i, j - 1, A \rightarrow \alpha \cdot a_i \gamma) \in E\) then \((i, j, A \rightarrow \alpha a_i \gamma) \in E\).
- If \((i, k, A \rightarrow \alpha \cdot B \gamma) \in E\) and \((k, j, B \rightarrow \beta \cdot \gamma) \in E\) then \((i, j, A \rightarrow \alpha B \cdot \gamma) \in E\).
- Nothing is in \(E\) except those elements which must be in \(E\) by applying the preceding rules finitely often.

The item \((i, j, A \rightarrow \alpha a_i \gamma)\) in the second rule is said to be obtained by means of a scanner operation, while the item \((i, j, A \rightarrow \alpha B \cdot \gamma)\) in the third rule is said to be obtained by means of a completer operation (see figure 2).

It can be shown that \(E = \{(i, j, A \rightarrow \alpha \cdot B) | A \rightarrow \alpha \beta \Rightarrow \gamma, a_i \ldots a_j \text{ and so } S \Rightarrow^* S \text{ for some } S \rightarrow \alpha \in P\}\).

In the algorithm we will use a set \(E\) to define a context filter. For the standard version of the following algorithm we will assume \(E = \{0, \ldots, n\}^2 \times \{A \rightarrow \alpha \cdot B | A \rightarrow \alpha \beta \in P\}\). For this choice of \(E\) the test whether the items are in \(E\) is trivially true. It can be proved that for this choice of \(E\) the computed \(E'\) is equal to \(E\).

We can also choose an alternative definition for \(E\). We will restrict \(E\) such that only items that fit the context are allowed: \((i, j, A \rightarrow \alpha \cdot B)\) fits the context iff \(a_i \in \text{Precede}(A)\) and \(a_j + 1 \in \text{First}(\beta \cdot \text{Follow}(A))\). For this choice of \(E\) we call the computed \(E' = E'_{1,1}\).

Earley’s Recognizer:

\[
E' := \emptyset \\
\text{for } i := 0 \text{ to } n \text{ do} \\
\quad \text{for each } j := 0 \text{ to } n - i \text{ in parallel do} \\
\]

\[\text{if } i = 0 \text{ then} \\
E' := E' \cup \{(j, j, A \rightarrow \cdot \alpha) \in E \mid [A \rightarrow \alpha \in P]\} \\
\text{else} \\
E' := E' \cup \{(j, j, A \rightarrow a_i \gamma) \in E \mid [j, j + 1, A \rightarrow \alpha \cdot a_i \gamma] \in E'\} \\
\text{for each } k := 1 \text{ to } i - 1 \text{ in parallel do} \\
E' := E' \cup \{(j, j + 1, A \rightarrow \cdot \gamma) \in E \mid [j, j + 1, A \rightarrow \alpha \cdot a_i \gamma] \in E'\} \\
\text{while } E' \text{ still changes do} \\
E' := E' \cup \{(j, j + 1, A \rightarrow \alpha B \cdot \gamma) \in E \mid [j, j + 1, A \rightarrow \alpha B \cdot \gamma] \in E'\} \\
\text{and } (j + 1, j + i, B \rightarrow \cdot \beta) \in E' \} \\
E' := E' \cup \{(j, j + 1, A \rightarrow \alpha B \cdot \gamma) \in E \mid [j, j + 1, A \rightarrow \alpha B \cdot \gamma] \in E'\} \\
\text{and } (j + 1, j + i, B \rightarrow \cdot \beta) \in E' \} \\
\text{return whether or not } (0, n, S \rightarrow \cdot \alpha) \in E' \text{ for some } S \rightarrow \alpha \in P\]

3.3 The Double Dotted Algorithm

The double dotted algorithm is a slightly changed and improved version of the one that can be found in [Vre93]. The parallel algorithm can be characterized by the following partial parse relation.

**Definition 3.3** The relation \(D \subseteq \{0, \ldots, n\}^2 \times \{A \rightarrow \alpha \cdot B \cdot \gamma | A \rightarrow \alpha B \gamma \in P\}\) is defined as fol-
lows:

- If $A \rightarrow \lambda \in P$ then $(j, j, A \rightarrow \cdots) \in D$ for any $j \in \{0, \ldots, n\}$.

- If $A \rightarrow a_\beta \gamma \in P$ then $(j - 1, j, A \rightarrow a_\beta \gamma) \in D$ for any $j \in \{1, \ldots, n\}$.

- If $A \rightarrow a_\beta B \gamma \in P$ and $(i, j, B \rightarrow \cdots) \in D$ then $(i, j, A \rightarrow a_\beta B \gamma) \in D$.

- If $(i, k, A \rightarrow \cdots a \chi \gamma) \in D$ and $(k, j, A \rightarrow \cdots a \chi \gamma) \in D$ then $(i, j, A \rightarrow \cdots a \chi \gamma) \in D$ for any $X \in V$.

- Nothing is in $D$ except those elements which must be in $D$ by applying the preceding rules finitely often.

The item $(i, j, A \rightarrow a_\beta B \gamma)$ of the third rule is said to be obtained by means of the inclusion operation, while item $(i, j, A \rightarrow a_\chi \gamma)$ is said to be obtained by means of the concatenation operation (see figure 3). This last operation is a specialized version of the one that can be found in [Vre93].

It can be shown that $D = \{(i, j, A \rightarrow \alpha \beta \gamma) | A \Rightarrow \alpha \beta \gamma \Rightarrow^*_\alpha a_{i+1} \cdots a_j \gamma\}$ if $\beta = \lambda$ then $\alpha \gamma = \lambda$, and if $|\beta| > 1$ then $\alpha = \lambda$ and thus $S \Rightarrow^* a_1 \cdots a_n$ iff $(0, n, S \rightarrow \cdots a_\cdot) \in D$ for some $S \rightarrow a \in P$.

![Diagram of double dotted operations](image)

**Figure 3:** Double dotted operations

In the algorithm we will use a set $D$ to define a context filter. For the standard version of the following algorithm we will assume $D = \{(A \rightarrow \alpha \beta \gamma | A \rightarrow \alpha \beta \gamma) \in P\}$. For this choice of $D$ the computed $D'$ is equal to $D$.

We can also choose an alternative definition for $D$. We will restrict $D$ such that only items that fit the context are allowed: $(i, j, A \rightarrow a_\beta \gamma \gamma) \rightarrow \ast (\text{context})$ if $a_i \in \text{Last}(\text{Precede}(A \beta) \chi)$ and $a_{i+1} \in \text{First}(\gamma \text{Follow}(A))$. For this choice of $D$ we call the computed $D' = D_{1,1}$.

**Double-Dotted-Recognizer:**

$D' := \emptyset$

for $i := 0$ to $n$

for $j := 0$ to $n - i$ in parallel do

  case $i$

    0:

      $D' := D' \cup \{(j, j, A \rightarrow \cdots) \in D$

      $|A \rightarrow \lambda \in P\}$

    1:

      $D' := D' \cup$

      $\{(j - 1, j, A \rightarrow a_\beta a_{i+1} \cdots a_j \gamma) \in D$

      $|A \rightarrow a_\alpha \gamma \in P\}$

    2:

      $D' := D' \cup$

      $\{(j, j + i, A \rightarrow a_\beta a_{i+1} \cdots a_j \gamma) \in D$

      $|X \in V, (j, j + i, A \rightarrow a_\alpha X \gamma) \in D$

      $\land (j + 1, j + i, A \rightarrow \cdots a_\cdot) \in D'\}$

      for $k := 1$ to $i - 1$ in parallel do

        $D' := D' \cup$

        $\{(j, j + i, A \rightarrow a_\beta a_{i+1} \cdots a_j \gamma) \in D$

        $|X \in V, (j, j + i, A \rightarrow a_\alpha X \gamma) \in D$

        $\land (j + 1, j + i, A \rightarrow \cdots a_\cdot) \in D'\}$

      $D' := D' \cup$

      $\{(j, j + i, A \rightarrow a_\beta \gamma \gamma) \in D$

      $|X \in V, A \rightarrow a_\beta \gamma \gamma \in P$

      $\land (j, j + i, B \rightarrow \cdots) \in D'\}$

      return whether or not $(0, n, S \rightarrow \cdots a_\cdot) \in D'$

for some $S \rightarrow a \in P$

4 The Measurements

The grammar and sentence set that are used in the comparison are corrected versions of the third Tomita grammar and sentence set [Tom86, HV91]. For each recognizer two versions are used: the standard version and an improved version. The standard versions compute CYK, E, and D, while the improved versions compute CYK, E, and D, and D_{1,1}. There are three interesting things to measure:
Figure 4: Sentence length versus the sequential time

Figure 5: Sentence length versus the sequential time

Figure 6: Sentence length versus the parallel time

Figure 7: Sentence length versus the parallel time
Figure 8: Sentence length versus the total number of items

Figure 9: Sentence length versus the total number of items

- the sequential time (see figures 4 and 5),
- the parallel time (see figures 6 and 7),
- the space (see figures 8 and 9).

The determination of the amount of space isn't that difficult. Probably the best and most honest way is to count the number of items in the matrices. This might look dishonest because a CYK item is more simple than an Earley item, which is again simpler than a double dotted item. However, all can be coded in a single machine integer.

Measuring the time, however, is a problem since this strongly depends on the machine used and also on how the various parts of the algorithms are implemented in the programs. A reference to an item in the matrix or the addition of an item to the matrix is counted as one time step.

Although the recognizers that were used, are parallel recognizers, the machine on which the recognizers ran, is not. For the determination of the sequential time and the number of items this isn't a problem, but of course it is a problem if we want to measure parallel time. We have made our parallel measurements in the following ways.

All three algorithms have the structure that the parallel loop over $j$ contains a parallel loop over $k$ followed by a sequential while loop (absent in the CYK algorithm). For each of the cells on the diagonal we count the number of items entered in or being referenced to during the computation of that cell. For the loop over $k$ we count only the largest amount of work that had to be done in that loop since the loop was assumed to be done in parallel. For the while loop we will only count the time for the cell that had taken the most time to be computed as the time needed to compute that diagonal. We have made the following assumptions:

- The architecture is able to resolve read and write conflicts.
- Communication between processors is without costs.
- All and only the parallel parts run completely asynchronously. All other parts run synchronized.

With the given diagonal by diagonal approach, this will give the best possible bound that we can achieve.
5 The Comparison

Some warnings are in order. First of all, we have only used one moderate size grammar and one small sentence set. Thus, it is impossible to state statistically sound conclusions. Secondly, Tomita's choice for this grammar and sentence set is undocumented and inspection of both shows some peculiarities. Some of the pictures seem to suggest some sort of linear correspondence. However, for all algorithms and measurements there is absolutely no reason to suspect that there is such a correspondence.

When we look at the results we can make the following observations:

- The recognition of sentences is highly influenced by the choice of sentences. Even recognition times of sentences of the same lengths vary wildly. This is especially true for parallel recognition. The pictures shows highly scattered clouds of points.

- The way we computed the recognition matrices, is not the way to compute the matrices in parallel. Maximum overall speedup of 4.3 and 4.6 for CYK and CYK\textsubscript{1,1}, 3.3 and 3.1 for E and E\textsubscript{2,1}, and 4.2 and 3.5 for D and D\textsubscript{1,1} look rather disappointing.

- The results of actively filtering bad items out, does seem to pay off. Both space and time is reduced considerably. When we compare the standard with the improved versions we get the following results.

For the total number of items we see that the context filter for CYK reduces the number with a factor 3.7, for Earley this is 8.3, and for the double dotted one it is 4.8. For the sequential time we see an increased speed of a factor 3.3 for CYK, 13 for Earley, and 9.1 for the double dotted algorithm. In the parallel algorithms the gain achieved for CYK is a factor 3.5, for Earley this is a factor 12, and it is 7.6 for the double dotted one.

6 Final Remarks

If we would just look to the results of this paper, we see that filtering is much more effective than running the algorithms in parallel. In practice the situation for the parallel algorithms is even worse: we have assumed an almost ideal architecture with respect to communication costs and read and write conflicts.

The reason for the disappointing parallel behavior is that many processors stay idle because only some cells on the diagonal need extensive computations. Hopefully an island driven approach will lead to better results.

Although filtering looks quite promising, there are a few drawbacks. First and most of all, it is theoretically known that there exist grammars for which filtering has little to no effect. Second, the comparison was done with just one grammar and one sentence set. And finally, we compared the parallel results with the results that would be obtained if the (parallel) algorithms would run sequentially. The spectacular numbers given here don't necessarily apply to purely sequential algorithms.

The reason for the speed up in the case of the filtering is believed to be that since the number of items found in the matrix cells is so much less that there is far less work to do with the matrix cells. The speed up grows even faster than the rate of the reduction of the space requirements (which is to be expected due to the various loops).

References


Head-Corner Parsing of Unification Grammars:
A Case Study

Margriet Verlinden

Department of Computer Science, University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands
email: margriet@cs.utwente.nl

ABSTRACT

The head-corner chart parsing algorithm by Sikkel and Op den Akker in [SA92] defines a new way to grammatically analyse sentences of a context-free language. In this algorithm sentences are parsed in another order than as usual from left to right. In informal terms, head-driven parsing is a strategy for syntactic analysis in which we start by looking for the main word. There is a linguistic motivation for the notion head, but in this paper the subject is looked at only from a computing science perspective.

The head-corner parsing algorithm has been extended to unification grammars and an implementation of the extended algorithm has been written. In this paper will be presented: the original algorithm, the extension, some aspects of the implementation and results of comparisons of the head-corner parser with some other parsers.

1 THE ALGORITHM

The algorithm for head-corner chart parsing from [SA92] was the basis for the implementation of a parser. A formal description of this algorithm will be given in this section. In [SA92] more versions of the algorithm and proofs of correctness can be found. A shorter version has been published as [SA93].

1.1 NOTATIONAL CONVENTIONS

We use the following notations. Nonterminals are denoted by \( A, B, \ldots \in \mathcal{N} \); terminals by \( a, b, \ldots \in \Sigma \). We write \( V \) for \( \mathcal{N} \cup \Sigma \), with \( X, Y, \ldots \) as typical elements. Strings in \( V^* \) are denoted by \( \alpha, \beta, \ldots \).

A context-free grammar \( G \) is a 4-tuple \((N, \Sigma, P, S)\), with \( P \) a set of productions and \( S \) the start symbol. A lexicon \( L \) for \( G \) is a 3-tuple \((\Sigma, E, W)\), with \( W \) the words of a natural language, \( E \) a set of lexical entries of the form \( b \rightarrow \text{word} \), with \( b \in \Sigma \) and \( \text{word} \in W \). The sentence to be parsed is often denoted by the lexical categories of its words: \( a_1 \ldots a_n \). Place markers \( i, j, k \ldots \) are used to indicate positions in the sentence. The symbol \( a_i \) is located between positions \( i-1 \) and \( i \). The derivation arrow \( \Rightarrow \) is defined by:

\[ aA\beta \Rightarrow a\alpha\gamma\beta \quad \text{if there is a production } \alpha \in P \text{ of the form } A \rightarrow \gamma. \]

Furthermore \( \Rightarrow^* \) is the transitive and reflexive closure of \( \Rightarrow \).

1.2 HEADS

To define the algorithm for head-corner chart parsing we need to define the notions head, head grammar and head-corner first:

- A context-free head grammar is a 5-tuple \((N, \Sigma, P, S, r)\), with \( r \) a function that assigns a natural number to each production in \( P \). Let \( |p| \) denote the length of the right-hand side of \( p \). Then \( r \) is constrained to \( r(p) = 0 \) for \( |p| = 0 \) and \( 1 \leq r(p) \leq |p| \) for \( |p| > 0 \).

- The head of a production \( p \) is the \( r(p) \)-th symbol of the right-hand side; the head is empty if and only if the production is empty.

- The head-corner relation \( >_h \) on \( \mathcal{N} \times (\mathcal{V} \cup \{ \varepsilon \}) \) is defined by:

\[ A >_h U \text{ if there is a production } p = A \rightarrow \alpha \in P \text{ with } U \text{ the head of } p. \]

More use will be made of the transitive and reflexive closure of \( >_h \), denoted as \( >^*_h \).
In a much more practical notation for head grammars, the function \( r \) is not defined explicitly, but the head of each production is simply underlined. An example is given below.

\[
\begin{align*}
S & \rightarrow NP \ VP \\
VP & \rightarrow \text{verb} \ NP \\
NP & \rightarrow \text{det} \ noun
\end{align*}
\]

1.3 ITEMS

The head-corner chart algorithm is related to the Earley chart parser ([Ear70]). Both are based on the recognition of items. Items represent parts of a phrase that have been or have to be parsed. For the head-corner parser we distinguish the following kinds of items:

\[
\begin{align*}
[l, r, A], & \quad \text{predict items,} \\
[l, r; A; B \rightarrow \alpha \beta \gamma, i, j], & \quad \text{head-corner items,} \\
[a, j - 1, j], & \quad \text{terminal items.}
\end{align*}
\]

Recognition of items should be interpreted as follows. A predict item \([l, r, A]\) is recognized if a constituent \( A \) is being looked for, located somewhere between \( l \) and \( r \). A head-corner item \([l, r; A; B \rightarrow \alpha \beta \gamma, i, j]\) is recognized if \([l, r, A]\) is looked for, \( A \Rightarrow^* B \), and \( \beta \Rightarrow^* a_{i+1} \ldots a_j \) has been established. A terminal item \([a, j - 1, j]\) is recognized if the \( j \)-th word of the sentence belongs to lexical category \( a \). An item is called recognizable

for a given sentence if a chart parser (following rules that will be introduced next) will add the item to the chart sometime or other. A sentence is correct if and only if \([0, n, S; S \rightarrow \delta^*, 0, n]\) is recognizable. For a more formal treatment and correctness of the head-corner parser see [SA92].

We start the parser with a chart that contains all recognizable terminal items. When the \( j \)-th word belongs to different categories, say \( a \) and \( b \), then both items \([a, j - 1, j]\) and \([b, j - 1, j]\) are present in the initial chart. Furthermore, the item \([0, n, S]\) is needed initially. New items can be derived from already recognized items by applying one of the operators, defined in Section 1.4. The chart parsing algorithm stops when all recognizable items have been added to the chart (and no other items).

1.4 OPERATORS

The definition of the operators is given in Figure 1. The operators predict, scan and complete can work in two directions from the parsed part of an item. In Figure 1 both possibilities are given for each operator. The turnstyle (\( \tau \)) notation is a convenient shorthand, to be interpreted as:

"if each of the the arguments (i.e. items left of \( \tau \)) has been recognized then the item right of \( \tau \) can be recognized also"

As mentioned in Section 1.3 the operators are used to recognize new items and add them to the chart. The conventional way to preserve that a deterministically defined set of items is added to the chart, is to make use of a second data structure called agenda. Initially, the chart contains only the terminal items; item \([0, n, S]\) is on the agenda. At each step of the algorithm, an item (the current item) is taken from the agenda and added to the chart. For each operator it is checked whether new items can be recognized from the current item and other relevant items present in the chart. These new items are added to the agenda. In Figure 2 this general chart parsing algorithm is written schematically.

The head-corner operators make that the order of parsing is determined by the heads of the productions. The analysis of the sentence 'The cat catches a mouse' for the grammar in Section 1.2 illustrates this. The completed chart for this example is shown in Figure 3. From a completed chart the parse tree of the parsed sentence can be derived. For the example the parse tree is given in Figure 4.

2 EXTENSION TO UNIFICATION GRAMMARS

In unification grammars symbols of the grammar may have features and productions may have feature rules or feature constraints. These constraints determine how the values of the features are to be computed from other values by unification, or what the conditions on feature values are. A parser for unification grammars not only computes parse trees for a sentence but it also computes the feature values of all words and checks the conditions. After the grammar and lexicon are extended with feature constraints also the roles of feature structures in the items and operators have to be defined. All these adaptations of the algorithm will be described in this section.

2.1 GRAMMAR AND LEXICON

In Figures 5 and 6 examples of a grammar and lexicon with feature constraints are given. Linguistically much more interesting features can be
head-corner: there are three distinguished cases for head-corners $b, C, \varepsilon$.

(i) for $A \gamma^* b, B \rightarrow \alpha b \gamma \in P, l < j \leq r$:

\[ l, r, A, [b, j - 1, j] \vdash [l, r, A; B \rightarrow \alpha b \gamma, j - 1, j]; \]

(ii) for $A \gamma^* B, B \rightarrow \alpha C \gamma \in P$:

\[ l, r, A; C \rightarrow \delta, i, j \vdash [l, r, A; B \rightarrow \alpha C \gamma, i, j]; \]

(iii) for $A \gamma^* B, B \rightarrow \varepsilon \in P, l \leq j \leq r$:

\[ l, r, A, \vdash [l, r, A; B \rightarrow \varepsilon, j, j]; \]

predict:

\[ l, r, A; B \rightarrow \alpha C, \beta \gamma, i, j \vdash [l, i, C]; \]

\[ l, r, A; B \rightarrow \alpha \beta C \gamma, i, j \vdash [j, r, C]; \]

sem:

for $l < j \leq r$:

\[ [a, j - 1, j], [l, r, A; B \rightarrow \alpha a \beta \gamma, j, k] \vdash [l, r, A; B \rightarrow \alpha a \beta \gamma, j - 1, k]; \]

for $l \leq j < r$:

\[ [l, r, A; B \rightarrow \alpha \beta a \gamma, i, j], [a, j, j + 1] \vdash [l, r, A; B \rightarrow \alpha \beta a \gamma, i, j + 1]; \]

complete:

\[ [l, j, C; C \rightarrow \delta \gamma, i, j], [l, r, A; B \rightarrow \alpha C \beta \gamma, j, k] \vdash [l, r, A; B \rightarrow \alpha C \beta \gamma, i, k]; \]

\[ [l, r, A; B \rightarrow \alpha \beta C \gamma, i, j], [j, r, C; C \rightarrow \delta \gamma, j, k] \vdash [l, r, A; B \rightarrow \alpha \beta C \gamma, i, k]. \]

Figure 1: The operators for head-corner chart parsing.

program chart parser
begin
create initial chart and agenda;
while agenda is not empty do
  delete (arbitrarily chosen) current item from agenda;
  if current $\notin$ chart then
    add current to chart;
    add all items to agenda that can be recognized by
    any operator using current and items in chart
    fi
od
end.

Figure 2: General schema for a chart parser.
Figure 3: The completed head-corner chart for the sentence 'The cat catches a mouse'.

Figure 4: The parse tree of the sentence 'The cat catches a mouse'.
choose (see e.g. [StS03]) but here these examples will do. In this paper a notation is used that is almost identical to PATR-II ([Shi86]). As in PATR, we may replace formal variables \(X_0, X_1, X_2, \ldots\) by the syntactic categories (i.e. cat features) of these variables. The categories are not mentioned again in the feature constraints. This way the context-free parts of the grammar and lexicon are the same as in Section 1.

\[
S \rightarrow NP \ VP \\
S : syn : voice = active \\
S : sem : scene = VP : sem \\
S : sem : subject = NP : sem \\
VP : syn : number = NP : syn : number \\
VP : syn : person = NP : syn : person \\
VP \rightarrow verb \ NP \\
VP : sem : action = verb : sem \\
VP : sem : object = NP : sem \\
VP : syn : number = verb : number \\
NP : syn : number = verb : number \\
NP \rightarrow det \ noun \\
NP : sem = noun : sem \\
NP : syn : number = noun : number \\
NP : syn : number = noun : number \\
NP : det = noun = number \\
NP : det = noun = number
\]

Figure 5: Grammar with feature constraints

\[
det \rightarrow \text{'the'} \\
det \rightarrow \text{'a'} \\
noun \rightarrow \text{'cat'} \\
noun : sem = cat \\
noun : number = singular \\
noun : person = 3rd \\
noun \rightarrow \text{'mouse'} \\
noun : sem = mouse \\
noun : number = singular \\
noun : person = 3rd \\
verb \rightarrow \text{'catches'} \\
verb : sem = catching \\
verb : number = singular \\
verb : person = 3rd
\]

Figure 6: Lexicon with feature constraints

...tion and negation. So the distinction between grammar productions or lexical entries is decided by both the syntactical categories and the feature constraints.

For practical reasons the feature constraints are put together in feature structures, one structure for each constituent in a production or entry [see also [Shi86]). A common way to notate feature structures are attribute value matrices (AVMs). For example in Figure 7 the feature structures belonging to (the constituents in) the production \(NP \rightarrow det \ noun\) are represented as such matrices. In Figure 8 the feature structure belonging to (the noun in) the entry \(noun \rightarrow \text{'mouse'}\) is given.

\[
NP \rightarrow \begin{bmatrix}
\text{sem} : & 1 \\
\text{syn} : & \begin{bmatrix} nr : & 2 \end{bmatrix}
\end{bmatrix}
\]

\[
\begin{bmatrix}
det \rightarrow \begin{bmatrix} \text{number} : & 2 \end{bmatrix}
\end{bmatrix}
\]

\[
noun \rightarrow \begin{bmatrix}
\text{sem} : & \text{mouse} \\
\text{number} : & \text{singular} \\
\text{person} : & \text{3rd}
\end{bmatrix}
\]

Figure 7: Feature structures in \(NP \rightarrow det \ noun\)

Figure 8: Feature structure of the noun 'mouse'

If nothing is known about a structure the structure is empty, denoted as \([\ ]\). If a (partial) feature structure is reachable by different paths, this is denoted with a numbered box; the same number means same structure. This so-called co-reference\(^1\) is the main mechanism of unification grammars. Via coreferences information can be synthesized from the lexicon and grammar into the final feature structure of a sentence \(S\).

2.2 Items and Operators

In Section 2.1 the input for a head-corner parser for unification grammars has been described. In this section the main adaptions of the items and operators will be given.

\(^1\)reentrancy in [Shi86] on path-equivalence in [KR86]
First all constituents of the items get feature structures like the constituents in the grammar and lexicon have. The terminal items get their feature structures straight from the lexicon. If a word has two entries in the lexicon we have to create two terminal items, possibly with the same context-free part, but with different feature structures. In the initial predict item \([0, n, S]\) we know nothing yet about the features of \(S\), so \(S\) has an empty structure. Now the feature structures of all initial items are fixed.

All other items on the chart are results of operators applied to already recognized items. Therefore it is sufficient to give a description for each operator how the feature structures in the new items are determined by the ones in the old items. To get a compact description a few short notations are introduced:

\[
X_f \text{ in } p : \text{ the feature structure of constituent } X \text{ in production } p \text{ (as explained in Section 2.1);}
\]

\[
X_f \text{ in } it : \text{ the feature structure of constituent } X \text{ in item } it;
\]

\[
f_1 \sqcup f_2 : \text{ the result of unification of feature structure } f_1 \text{ with feature structure } f_2.
\]

In Figure 9 the description is given of constituents in newly recognized items. With this description the values of all features in the new items are fixed. Note that coreferences between constituents within an item have to be retained, because these embody constraints that are imposed by the lexicon and the productions. But the coreferences across items are undesirable. An item or a production can be used more than once for the recognition of a new item. In most cases the new items belong to different parses of one sentence. Therefore we do not want the new items to be connected via the old item respectively the production.\(^2\)

The simplest solution to this problem is to make copies of the old feature structures. Only in case of unification we do not copy the operands of the unification but we use a non-destructive unification algorithm. For all feature structures that depend on only one other feature structure we still have to make copies. As can be seen in Figure 9 many copies have to be made.

Parallel to the new lexicon and grammar an important difference with the context-free case is that an item can now occur more than once in the chart only with different feature structures. So the distinction between items is determined by both their context-free parts and the feature structures of their constituents. This distinction is important to determine whether a newly recognized item is on the chart already or should be added to the chart.

Finally the chart will contain one or more items of the form \([0, n, S; S \rightarrow \delta_\ast, 0, n]\). In each item of this form the feature structure of \(S\) (right of the semicolon) is a feature structure of the sentence \(\alpha_1 \ldots \alpha_n\) that has been parsed.

With the grammar and lexicon of Section 2.1 the chart for the sentence 'the cat catches a mouse' is the same as the one in Figure 3 apart from the feature structures. There were no words that occurred more than once in the lexicon (Figure 8) so the terminal items are the same. In the grammar (Figure 7) no production occurred more than once either, so there are no extra possibilities to apply a head-corner operator. The feature structure of the sentence (second \(S\) in item (15)) is given in Figure 10.

\[
S \rightarrow
\begin{bmatrix}
\text{syn : } [\text{voice : active}]
\text{sem : }
[\text{scene : } [\text{action : catching}]]
[\text{object : mouse}]
[\text{subject : cat}]
\end{bmatrix}
\]

Figure 10: Feature structure of the sentence 'The cat catches a mouse'.

3 \ THE IMPLEMENTATION

After adapting the head-corner algorithm I implemented a head-corner parser for unification grammars. As usual still many decisions had to be made to do so: A few of those decisions will be described in this section. I wrote the code for the head-corner parser in the imperative programming language Modula-2.

3.1 \ INPUT PROCESSING

A lot of information from the lexicon and grammar has to be made explicit. For example we say easily \('a \in \Sigma', 'A \overset{\ast}{\rightarrow} B'\) or 'for all \(p \in P\). To be able to handle all expressions used in the description of the algorithm we need to put the lexicon
head corner (i): \text{newitem} = [l, r, A; B \rightarrow \alpha \cdot \beta \cdot \gamma, j - 1, j] \text{ and } p = B \rightarrow \alpha \beta \gamma

A_f \text{ in newitem is as } A_f \text{ in } [l, r, A]
B_f \text{ in newitem is as } B_f \text{ in } p
X_f \text{ in newitem is as } X_f \text{ in } p, \text{ for all } X \in \alpha \gamma
b_f \text{ in newitem is as } (b_f \text{ in } [b, j - 1, j]) \cup (b_f \text{ in } p);

head corner (ii): \text{newitem} = [l, r, A; B \rightarrow \alpha \cdot C \cdot \gamma, i, j] \text{ and } p = B \rightarrow \alpha \cdot C \cdot \gamma

A_f \text{ in newitem is as } A_f \text{ in } [l, r, A; C \rightarrow \delta \cdot \iota, i, j]
B_f \text{ in newitem is as } B_f \text{ in } p
X_f \text{ in newitem is as } X_f \text{ in } p, \text{ for all } X \in \alpha \gamma
C_f \text{ in newitem is as } (C_f \text{ in } [l, r, A; C \rightarrow \delta \cdot \iota, i, j]) \cup (C_f \text{ in } p);

head corner (iii): \text{newitem} = [l, r, A; B \rightarrow \ldots, j, j] \text{ and } p = B \rightarrow \epsilon

A_f \text{ in newitem is as } A_f \text{ in } [l, r, A]
B_f \text{ in newitem is as } B_f \text{ in } p;

predict (left): \text{newitem} = [l, r, C]
C_f \text{ in newitem is as } C_f \text{ in } [l, r, A; B \rightarrow \alpha \cdot C \cdot \beta \cdot \gamma, i, j];

predict (right): \text{newitem} = [j, r, C]
C_f \text{ in newitem is as } C_f \text{ in } [l, r, A; B \rightarrow \alpha \cdot \beta \cdot C \cdot \gamma, i, j];

scan (left): \text{newitem} = [l, r, A; B \rightarrow \alpha \cdot \beta \cdot \gamma, j - 1, k]

A_f \text{ in newitem is as } A_f \text{ in } [l, r, A; B \rightarrow \alpha \cdot \beta \cdot \gamma, j, k]
B_f \text{ in newitem is as } B_f \text{ in } [l, r, A; B \rightarrow \alpha \beta \gamma, j, k]
X_f \text{ in newitem is as } X_f \text{ in } [l, r, A; B \rightarrow \alpha \beta \gamma, j, k], \text{ for all } X \in \alpha \beta \gamma
a_f \text{ in newitem is as } (a_f \text{ in } [a, j - 1, j]) \cup (a_f \text{ in } [l, r, A; B \rightarrow \alpha \beta \gamma, j, k]);

scan (right): \text{newitem} = [l, r, A; B \rightarrow \alpha \cdot \beta \cdot \gamma, i, j + 1]

A_f \text{ in newitem is as } A_f \text{ in } [l, r, A; B \rightarrow \alpha \beta \gamma, i, j]
B_f \text{ in newitem is as } B_f \text{ in } [l, r, A; B \rightarrow \alpha \beta \gamma, i, j]
X_f \text{ in newitem is as } X_f \text{ in } [l, r, A; B \rightarrow \alpha \beta \gamma, i, j], \text{ for all } X \in \alpha \beta \gamma
a_f \text{ in newitem is as } (a_f \text{ in } [a, j + 1]) \cup (a_f \text{ in } [l, r, A; B \rightarrow \alpha \beta \gamma, i, j]);

complete (left): \text{newitem} = [l, r, A; B \rightarrow \alpha \cdot C \cdot \beta \cdot \gamma, i, k]

A_f \text{ in newitem is as } A_f \text{ in } [l, r, A; B \rightarrow \alpha \cdot C \cdot \beta \cdot \gamma, j, k]
B_f \text{ in newitem is as } B_f \text{ in } [l, r, A; B \rightarrow \alpha \cdot C \cdot \beta \cdot \gamma, j, k]
X_f \text{ in newitem is as } X_f \text{ in } [l, r, A; B \rightarrow \alpha \cdot C \cdot \beta \cdot \gamma, j, k], \text{ for all } X \in \alpha \beta \gamma
C_f \text{ in newitem is as } (C_f \text{ in } [l, j, C_1; C_2 \rightarrow \delta \cdot \iota, i, j]) \cup (C_f \text{ in } [l, r, A; B \rightarrow \alpha \cdot C \cdot \beta \cdot \gamma, j, k]);

complete (right): \text{newitem} = [l, r, A; B \rightarrow \alpha \cdot \beta \cdot C \cdot \gamma, i, k]

A_f \text{ in newitem is as } A_f \text{ in } [l, r, A; B \rightarrow \alpha \beta \gamma, i, j]
B_f \text{ in newitem is as } B_f \text{ in } [l, r, A; B \rightarrow \alpha \beta \gamma, i, j]
X_f \text{ in newitem is as } X_f \text{ in } [l, r, A; B \rightarrow \alpha \beta \gamma, i, j], \text{ for all } X \in \alpha \beta \gamma
C_f \text{ in newitem is as } (C_f \text{ in } [j, r, C_1; C_2 \rightarrow \delta \cdot \iota, j, k]) \cup (C_f \text{ in } [l, r, A; B \rightarrow \alpha \beta \gamma, i, j]).

Figure 9: Extension of the operators to feature structures
Building feature structures
We can think of feature structures as directed acyclic graphs (DAGs). For example, the feature structure of *S* in Figure 10 can be represented as the graph in Figure 11 (see also [Shi86]). Those feature graphs were implemented with pointers.

![Diagram of feature graph](image)

Figure 11: The feature graph of the sentence 'The cat catches a mouse'.

Properties of the categories
After putting the grammar and lexicon in data-structures, first a list of all syntactic categories was made. For each category it was determined whether it was a terminal, a nonterminal, or even both. During testing with realistic grammars it turned out that a grammar writer possibly wants to use a lexical category as a nonterminal as well. The Plinius grammar (see Section 4) contains for example the production ‘*noun* → *noun noun*’.

Next the head-corner relation and its closure was determined. The transitive and reflexive closure of this relation (≻₆) was derived using an algorithm of Yellin ([Yel88]).

### 3.2 Operations on feature structures

We have seen in Section 3.1 that the implementation of the feature structures was not straightforward. Also during parsing the feature structures needed special attention.

**Copying**
For the independence between items we have to copy feature structures. In order to retain the coreferences within one feature structure, the copying was done as follows. To copy one feature structure a list was made of all the vertices in that structure. For each vertex it was noted whether it had been copied and which vertex was its copy. Walking through the original feature structure a new feature structure was built. If a vertex had been copied, the copy found in the list was used and otherwise a copy was made and used, and the list was updated.

By application of some operators we have to copy a whole production (out of the grammar or an item) together with all its interconnected feature structures. In these cases the feature structures were copied one by one using only one vertex-list for all feature structures. This way the coreferences between the feature structures belonging to that one production were retained.

**Unification**
We use a non-destructive adaption of Huet's unification algorithm ([Hue76]), that is similar to the congruence closure algorithm of Nelson and Oppen ([NO80]), cf. Wroblewski ([Wro87]). See Chapter 9 of [Sik83] for more details.

### 3.3 An alternative distinction between items

In Section 2.2 we talked about the distinction between items with feature structures. The most straightforward way to determine this distinction seems comparison of the context-free parts and of the feature structures. But the feature structures in the grammar production of a head-corner item can be connected by coreferences. This makes the comparison of two head-corner items with both a set of interconnected feature structures rather complicated. That is why I worked out another solution. This easier way to compare items will be described in this section.

Predict items and the predict part (before the semicolon) of head-corner items were compared just by looking at the context-free parts and the feature structures. For each terminal item and for the production part (after the semicolon) of each head-corner item a parse tree was created. These parse trees are only intuitively introduced.
In fact the trees are the usual parse trees (see Figure 4) annotated with the productions and lexical entries involved in the parsing. To annotate the trees we number the productions and the entries (up from 1). The parse tree for item (13) from the chart of the running example is given in Figure 12. Earlier items like (13) and (11) have unparsed strings outside the dots. All constituents in these strings are represented by empty branches in the parse trees. Terminal items get parse trees consisting of only one leave, the lexical entry.

This way all sources of feature information about the parsed phrase (all used grammar productions and lexical entries) are referred to in the annotated parse tree. Differences between the feature structures of items can only occur by use of different grammar productions or lexical entries, so all differences are noticed by comparison of the annotated parse trees.

In exceptional cases the feature structures of two items can be the same while their annotated parse trees are different. In these cases the grammar writer should think well about what he wants to achieve with the different productions and entries that lead to the same feature structures. If only the annotations in the parse trees are different it is a warning to the grammar writer for possible redundancy in the grammar or lexicon. If the parse trees differ in the constituents it is good to make a distinction between the items, because they belong to totally different parses.

Here we have arrived at another advantage (besides the easier comparison) of keeping the annotated parse trees in the items. We do not have to walk through the completed chart to find the parse trees. After parsing, each item of the form $[0, n, 5; 5 \rightarrow d_0, 0, n]$ is associated with one parse tree. For convenience a list of references to final items is also kept during parsing.

4 TEST RESULTS

Within the Department of Computer Science at the University of Twente there is a project on knowledge acquisition, called 'Plinius' (see also [StS93]). In this project a grammar and a lexicon with feature constraints are being developed. The Plinius corpus is a set of 400 English abstracts of technical articles on the mechanical properties of ceramic materials. Sentences from this corpus are being parsed with a simple parser. Sentences, grammar and lexicon of this project were used to test the head-corner parser.

4.1 HEAD-CORNER PARSER VERSUS PLINIUS PARSER

The Plinius parser is based on work of Gazdar and Mellish ([GM89]). It is also a parser for unification grammars. The parser works bottom-up and from left to right and has been written in PROLOG. All solutions are being looked for by means of backtracking. This mechanism makes the worst-case efficiency of the parser rather bad. That is why a more efficient parser was desired.
<table>
<thead>
<tr>
<th></th>
<th>large lexicon (52 words)</th>
<th>small lexicon (14 words)</th>
</tr>
</thead>
<tbody>
<tr>
<td>large grammar (20 productions)</td>
<td>sent.1, sent.2, sent.3</td>
<td>sent.1, sent.2, sent.3</td>
</tr>
<tr>
<td>small grammar (11 productions)</td>
<td>sent.1, sent.2</td>
<td></td>
</tr>
</tbody>
</table>

sent.1 = 'the material exhibited elongation', 1 parse
sent.2 = 'the material exhibited elongation in a test at temperatures', 5 parses
sent.3 = 'the material exhibited superplastic elongation in a tension test at temperatures and at strain rates', 7 parses

Figure 13: Test plan Plinius- versus head-corner parsing

<table>
<thead>
<tr>
<th></th>
<th>Plinius</th>
<th>sent.1</th>
<th>sent.2</th>
<th>sent.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>large lexicon large grammar</td>
<td>2,70</td>
<td>40,85</td>
<td>2903,10</td>
<td>48,4 min.</td>
</tr>
<tr>
<td>small lexicon large grammar</td>
<td>0,28</td>
<td>40,66</td>
<td>2893,95</td>
<td>48,2 min.</td>
</tr>
<tr>
<td>large lexicon small grammar</td>
<td>1,62</td>
<td>4,65</td>
<td>—</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Head-corner</th>
<th>sent.1</th>
<th>sent.2</th>
<th>sent.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>large lexicon large grammar</td>
<td>1,54</td>
<td>15,50</td>
<td>140,09</td>
<td>23,3 min.</td>
</tr>
<tr>
<td>small lexicon large grammar</td>
<td>1,38</td>
<td>15,02</td>
<td>139,38</td>
<td>23,2 min.</td>
</tr>
<tr>
<td>large lexicon small grammar</td>
<td>1,04</td>
<td>12,44</td>
<td>—</td>
<td></td>
</tr>
</tbody>
</table>

Figure 14: Parsing times of the Plinius- and the head-corner parser in seconds
After implementation of the head-corner parser a comparison has been made of the Plinius parser and the head-corner parser.

As the programs of the Plinius parser and the head-corner parser differ a lot, they can only be compared on the basis of the time they need to parse a certain sentence. The cpu-time is chosen because the real time varies strongly with the number of users on the machine. The needed cpu-time can be different on different machines so the programs have been tested on the same machine. Moreover, all tests have run ten times and the measurements have been averaged. The combinations of sentence, grammar and lexicon that have been tested are given in a schema (Figure 13). As testing was not the main purpose of the implementation, a limited number of cases has been tested. Testing with the large grammar and large lexicon is the most realistic situation. This situation can be simplified by reducing either the grammar or the lexicon. To learn about the influence of the size of grammar and lexicon, there have been testing in the two alternative situations too. The smaller grammar does not have enough productions to parse sentence 3 with. The Plinius grammar did not have heads so they were given in a linguistically usual way. Often the constituent in the righthand side that has the most feature information in common with the lefthand side, is taken as the head.

The results of the tests are in Figure 14. As we can see, the parsing time of the head-corner parser is much shorter than the time of the Plinius parser in most cases. The reduction varies from 36% to 95%. In two cases the Plinius parser is faster. But these are no realistic situations because either the grammar or the lexicon had been reduced artificially to reduce the parsing time of a specific sentence.

4.2 HEAD-CORNER PARSING VERSUS LEFT-CORNER PARSING

In [SA92] the head-corner parser is introduced by analogy to a left-corner parser. Left-corner parsers have been described by many authors. A left-corner parser parses sentences from left to right without jumping up and down the sentence. Usually left-corner parsing is described in simpler notations than used in this article. Less administration is needed because there are less possible orders for parsing. Still it is possible to describe left-corner parsing in the same notation as head-corner parsing. This looks a lot like head-corner parsing with a grammar that has all its heads extremely left. This way we can compare the efficiency of the left- and the head-corner parser by counting the number of created items.

Once the head-corner parser had been implemented it was easy to adapt it to use it as a left-corner parser. For the comparison the same three sentences were used as in the comparison with the Plinius parser, plus two sentences to learn about the effect of lexical ambiguities. Figure 15 shows the result of these tests. Sentences 4 and 5 have first been parsed with a lexicon with only

<table>
<thead>
<tr>
<th></th>
<th>sent.1</th>
<th>sent.2</th>
<th>sent.3</th>
<th>sent.4</th>
<th>sent.4*</th>
<th>sent.5</th>
<th>sent.5*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left-corner</td>
<td>32</td>
<td>126</td>
<td>381</td>
<td>28</td>
<td>69</td>
<td>32</td>
<td>33</td>
</tr>
<tr>
<td>Head-corner</td>
<td>37</td>
<td>205</td>
<td>962</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>39</td>
</tr>
<tr>
<td>change</td>
<td>+17%</td>
<td>+63%</td>
<td>+152%</td>
<td>+21%</td>
<td>-49%</td>
<td>+13%</td>
<td>+18%</td>
</tr>
</tbody>
</table>

* with a lexical ambiguity (‘yield’ as verb and as noun in the lexicon)

sent. 4 = ‘polycrystals yield hydroxyapatite’, 1 parse
sent. 5 = ‘hydroxyapatite increases the yield’, 1 parse

Figure 15: The number of created items at left-corner- and head-corner parsing

3The lexicon and grammar from Plinius are still being developed so their sizes are still far from realistic and terms like 'large' are only relative.
the right entry for 'yield' and again with a lexicon with two entries (*-marked column). As we can
see, in most cases the left-corner parser gives a better result.

Still, the head-corner parser can be of value. By introducing heads in the grammar productions the parsing can be more guided. If there are more discriminating parts in a phrase they can be parsed first in order to avoid false parses. Only the head-corner parser should be optimized in one very important respect. The head-corner parser described in this article suffers from the fact that it does not use alignment information. We take for example the sentence 'the elastic material bends in the test'. For every constituent 'the elastic material ...' similar constituents 'elastic material ...' and 'material ...' are recognized as well. An optimized head-corner parser should take only subjects that start at position 0. Important here is that the left-corner parser can be seen as a special case of the optimized head-corner parser, rather than the implemented head-corner parser: the left-corner parser does not leave gaps.

5 CONCLUDING REMARKS

Everything that has been noticed while working with the parser and that can be used for further work, will be discussed in this last section. First, a few stray observations are mentioned under the title 'Experiences'. Ways to improve the parser are given in Section 5.2. In the final section conclusions will be given.

5.1 EXPERIENCES

One limitation of the parser appeared when it was used in the Plinius project. The Plinius parser searches automatically for every possible category. The head-corner parser looks only for the category it is asked to look for by the initial predict item, in general \( [0, n, S] \). The head-corner parser can be adapted so that it searches all possible categories between the place markers 0 and n. But when this adaption had been made, the parser had become very slow. However, this adaption is probably not that much needed.

One benefit of the implementation is the calculated waste factor. The waste factor can be found empirically now and this may be helpful in developing theories on parsing. After counting all items that were needed to find the final item(s) the waste factor is calculated by:

\[
\text{waste factor} = \frac{\text{total}\# - \text{needed}\#}{\text{total}\#}
\]

For head-corner parsing of the sentences in Figure 15 the waste factor varies from 33\% to 87\%. The factor is higher for the longer sentences and charts. One thing noticed while calculating the waste factors is that the number of needed items differs sometimes for the left- and head-corner parsing of one sentence. Sometimes one or more extra predict items are used to achieve the same result.

5.2 POSSIBLE IMPROVEMENTS

During the eight months working with, reading and writing about the head-corner parser several suggestions for improvement of the parser have come up. They were found in literature or arose by 'playing' with the parser. Successively improvement of the preprocessing, typical improvements of a chart parser by using the place markers, and improvements with respect to the feature structures are discussed in this section.

PREPROCESSING

Of course the grammar and lexicon —among others— determine the performance of the parser. Actually we have to consider the grammar and lexicon a given input. But, we could ask the grammar writer to take care of a few things or we could adapt the input before parsing.

There are two sources of inefficiency in the grammar. Syntactic categories that are both terminal and nonterminal should be avoided. Head-recursive productions also lead to inefficiency.\(^4\) An interesting question for further study is whether the efficiency can be improved by special treatment of these constructions.

Moreover, we can improve the results of the parser by reducing the lexical ambiguity. One way to do so is to derive from the grammar precedence- and follower-relations between all categories. When we have a word with several lexical entries we can possibly forget about a few categories and shorten the initial chart. This is a simple adaption that can lead to a major efficiency gain.

PLACE MARKERS IN CHART PARSERS

In [SA92] two suggestions for improvement of the head-corner chart parser are given. First, when

\(^4\) A head recursive production is of the form \( A \rightarrow \alpha A \beta \), in analogy to left recursive productions.
an item \([...; A \rightarrow \alpha B \gamma, i, j]\) has been recognized by the head-corner rule, with \(\alpha \neq \varepsilon \neq \gamma\), it should be expanded either to \([...; A \rightarrow \alpha B \gamma, h, j]\) or to \([...; A \rightarrow \alpha B \gamma, i, k]\) but not in both directions; from either one a completed item \([...; A \rightarrow \alpha B \gamma, h, k]\) can be obtained. This was first suggested by Satta and Stock ([SS89]).

Second, the parser should use alignment information in order to avoid parsed parts of one sentence that do not fit together (see also Section 4.2).

Furthermore, the place markers of items can be used to organize the chart. This organization (like any arrangement) of the recognized items speeds up searches for items that — together with the current item — can be used as operands for an operation.

**Feature structures**

Finally, some actions on feature structures can be made more efficient and the feature structures themselves can be used to improve the performance of the parser.

Copying and unification of feature structures can be made more efficient by *subgraph sharing* (see [Kog90]). Subgraph sharing means that a vertex is only copied when it is actually used for different things. Until this double use, the vertex (and the feature graph connected to it) is shared by different feature structures. This way the number of vertices to be copied is reduced.

Besides being a source of inefficiency, feature structures can also be of help. Feature structures can be used for a form of *top-down filtering* as will be explained below. In the head-corner parser the predict items and the predict parts of the head-corner items contain feature information. This information could be used to check in an early stage whether the constituent that is being build can ever contribute to what is predicted. If it can not, we can stop (or ‘filter’5) such parses that are of no use. In order to do so, it is necessary to know what features from the lefthand side are inherited5 by the heads. In general this does not count for all features, so realizing the outlined filtering involves more than just unification of a predicted constituent with a found one. In order to improve the head-corner parser, top-down filtering is an option. But as long as this filtering is not realized (and it still is not) another choice of heads may be better. In stead of choosing heads on the basis of correspondence of feature infor-

---

5To inherit a feature: to have the same feature and with the same value.

---

**5.3 Conclusions**

We have successfully applied the head-corner parsing algorithm for unification grammars to the Plinius grammar. This confirms that head-corner parsing, previously described mainly on theoretical level, can be practically applied.

The left-corner parser, however, seems to be slightly more efficient than the head-corner parser. But it should be noticed that there is much room for optimization in the implementation of the head-corner parser, that will tip the balance (for this grammar) in favor of head-corner parsing.

As for unification grammars, in general the usefulness of head-corner parsing will strongly depend on the properties of the grammar. See also Bouma and Van Noord ([BN93]), who found that it depends on the grammar whether head-corner parsing is useful.

**Acknowledgements**

I would like to thank Mark Moll for writing the programming code for the unification of feature structures, and Wim van der Veer for programming an organized chart. Rieks op den Akker, Toine Andermarch, and Klaas Silkel have given helpful comments on a draft version of this paper.

**References**


A Multidisciplinary Approach to a Parsing Algorithm

Mark-Jan Nederhof *

University of Nijmegen, Department of Computer Science
Toernooiveld, 6525 ED Nijmegen, The Netherlands
E-mail: markjan@cs.kun.nl

ABSTRACT

In this paper we relate a number of parsing algorithms which have been developed in very different areas of parsing theory. We show that these algorithms are based on the same underlying ideas. The different parsing algorithms which are related include deterministic algorithms, a tabular algorithm, and a parallel algorithm.

By relating existing ideas, we hope to provide an opportunity to improve some algorithms based on features of others. A second purpose of this paper is to answer a question which has come up in the area of tabular parsing, namely how to obtain a parsing algorithm with the property that the table will contain as little entries as possible, but without the possibility that two entries represent the same subderivation.

1 INTRODUCTION

Left-corner (LC) parsing is a parsing strategy which has been used in different guises in various areas of computer science. Deterministic LC parsing with \( k \) symbols of lookahead can handle the class of LC\((k)\) grammars. Since LC parsing is a very simple parsing technique and at the same time is able to deal with left recursion, it is often used as an alternative to top-down (TD) parsing, which cannot handle left recursion and is generally less efficient.

Non-deterministic LC parsing is the foundation of a very efficient parsing algorithm [13], which is often used in natural language processing, and which is related to Tomita's algorithm and Earley's algorithm. It has one disadvantage however, which becomes noticeable when the gram-

\[
A \rightarrow a\beta_1 \mid a\beta_2 \mid \ldots
\]

where \( \alpha \) is not the empty string. After an LC parser has recognized the first symbol \( X \) of such an \( \alpha \), it will as next step predict all aforementioned rules (we abstain from discussing lookahead). This amounts to much non-determinism, which is detrimental both to the time-complexity and the space-complexity.

This problem of common prefixes is one of the problems of existing parsing techniques which have recently led to a number of new tabular parsing techniques such as head-corner parsing and head-driven parsing [19, 7, 2, 21]. It is not immediately clear however that some of the overhead of these algorithms does not deteriorate instead of improves the complexity in practical cases, at least where purely context-free parsing is concerned.

The much older PLR parsing technique has been introduced in [24] as a way of constructing small deterministic parsers with \( k \) symbols of lookahead for a proper subset of the LR\((k)\) grammars [23]. These PLR\((k)\) grammars properly contain the LC\((k)\) grammars.

A PLR parser can be seen as an LC parser for a transformed grammar in which some common prefixes have been eliminated and in this way PLR parsing is a partial solution to our problem.

However, PLR parsing only solves the problem of common prefixes when the left-hand sides of the rules are the same. In case we have e.g. the rules \( A \rightarrow a\beta_1 \) and \( B \rightarrow a\beta_2 \), where again \( \alpha \) is not the empty string but now \( A \neq B \), then even PLR parsing will in some cases not be able to avoid non-determinism after the first symbol of \( \alpha \) is recognized. We therefore go one step further and discuss extended LR (ELR) and common-

---

*Supported by the Dutch Organisation for Scientific Research (NWO), under grant 00-62-318
prefix (CP) parsing, which are algorithms capable of dealing with all kinds of common-prefixes without having to resort to unnecessary non-determinism. CP and ELR parsing are the foundation of a tabular parsing algorithm and a parallel parsing algorithm from the existing literature, but they have not been described in their own right.

To the best of the author's knowledge, the various parsing algorithms mentioned above have not been discussed together in the existing literature. The main purpose of this paper is to make explicit the connections between these algorithms. In this respect, we consider this paper to be a continuation of [13], which discussed algorithms that are all derived from left-corner parsing.

A second purpose of this paper is to show that CP and ELR parsing are obvious solutions to a problem of tabular parsing which can be described as follows. For each deterministic parsing algorithm there is a corresponding non-deterministic algorithm which can be realised using a parse table, where the parse table allows sharing of computation between different search paths. For example, deterministic LR parsing corresponds with Tomita's algorithm, which can be seen as a tabular realisation of non-deterministic LR parsing for grammars which are not LR [25].

In general, powerful deterministic parsing algorithms lead to efficient tabular parsing algorithms, provided the grammar can be handled almost deterministically. In case the original algorithm is very non-deterministic for a certain grammar however, sophistication of this algorithm which increases the number of parser states may lead to an increasing number of entries in the parse table of its tabular counterpart. This can be informally explained by the fact that each state represents the computation of a certain subderivation. If the number of states is increased then it is inevitable that at some point several states represent the same subderivation, which may lead to work being repeated during parsing. Furthermore, the parse forest (a compact representation of all parse trees) which is output by a tabular algorithm may in this case not be optimally dense.

We conclude that we have a tradeoff between the case that the grammar allows almost deterministic parsing and the case that the original algorithm is very non-deterministic for a certain grammar. In the former case, sophistication leads to less entries in the table, and in the latter case, sophistication leads to more entries, provided this sophistication is realised by an increase in the number of states. This is corroborated by empirical data from [1, 9], which deal with tabular LR parsing.

As we will explain, CP and ELR parsing have the nice property that they are more deterministic than most other parsing algorithms for many grammars, but their tabular realisation can never construct the same subderivation twice. This represents an optimum in a range of possible parsing algorithms.

This paper is organized as follows. Section 2 discusses non-deterministic left-corner parsing, and demonstrates how common prefixes in a grammar may be a source of bad performance for this technique. A multitude of parsing techniques which exhibit better treatment of common prefixes are discussed in Section 3. These techniques, including non-deterministic PLR, ELR, and CP parsing, have their origins in theory of deterministic, parallel, and tabular parsing. The application to parallel and tabular parsing is investigated more closely in Section 4. Data structures needed by the parsing techniques are discussed in Section 5.

If a grammar contains rules with empty right-hand sides, then this may complicate the parsing process. How we can deal with these rules is discussed in Section 6. Section 7 relates LR parsing to the ideas described in the preceding sections.

We will take some liberty in describing algorithms from the existing literature, since using the original descriptions would blur the similarities of the algorithms to one another. In particular, we will not treat the use of lookahead, and we will consider all algorithms to be non-deterministic unless indicated otherwise. We will only describe recognition algorithms, since each of the algorithms can be easily extended to yield parse trees as a side-effect of recognition.

The notation used in the sequel is for the most part standard and is summarised below.

A context-free grammar \( G = (T, N, P, S) \) consists of two finite disjoint sets \( N \) and \( T \) of nonterminals and terminals, respectively, a start symbol \( S \in N \), and a finite set of rules \( P \). Every rule has the form \( A \to \alpha \), where the left-hand side (lhs) \( A \) is an element from \( N \) and the right-hand side (rhs) \( \alpha \) is an element from \( V^* \), where \( V^* \) denotes \( (N \cup T) \). \( P \) can also be seen as a relation on \( N \times V^* \).

We use symbols \( A, B, C, \ldots \) to range over
$N$, symbols $X, Y, Z$ to range over $V$, symbols $\alpha, \beta, \gamma, \ldots$ to range over $V^*$, and $v, w, x, \ldots$ to range over $T^*$. We let $\epsilon$ denote the empty string.

The notation of rules $A \rightarrow \alpha_1, A \rightarrow \alpha_2, \ldots$ with the same rhs is often simplified to $A \rightarrow \alpha_1\alpha_2\ldots$

A rule of the form $A \rightarrow \epsilon$ is called an epsilon rule. Until Section 6 we will tacitly assume that grammars do not contain any epsilon rules.

The relation $P$ is extended to a relation $\rightarrow$ on $V^* \times V^*$ as usual. The reflexive transitive closure of $\rightarrow$ is denoted by $\rightarrow^*$.

We define: $B \not\rightarrow A$ if and only if $A \not\rightarrow B\alpha$ for some $\alpha$. The reflexive transitive closure of $\not\rightarrow$ is denoted by $\not\rightarrow^*$, and is called the left-corner relation.

A recognition algorithm can be specified by means of a push-down automaton $A = (T, \text{Alph}, \text{Init}, \rightarrow^*, \text{Fin})$, which manipulates configurations of the form $(\Gamma, v)$, where $\Gamma \in \text{Alph}^*$ is the stack, and $v \in T^*$ is the remaining input.

The initial configuration is $(\text{Init}, w)$, where $\text{Init} \in \text{Alph}$ is a distinguished stack symbol, and $w$ is the input. The steps of an automaton are specified by means of the relation $\rightarrow$. Thus, $(\Gamma, v) \rightarrow (\Gamma', v')$ denotes that $(\Gamma', v')$ is obtainable from $(\Gamma, v)$ by one step of the automaton. The reflexive transitive closure of $\rightarrow$ is denoted by $\rightarrow^*$. The input $w$ is accepted if $(\text{Init}, w) \rightarrow^* (\text{Fin}, \epsilon)$, where $\text{Fin} \in \text{Alph}$ is a distinguished stack symbol.

2 LC PARSING

For the definition of left-corner (LC) recognition we need stack symbols of the form $[A \rightarrow \alpha \cdot \beta]$, where $A \rightarrow \alpha\beta$ is a rule, and $\alpha \neq \epsilon$. Such a symbol is called an item. The informal meaning of an item is “The part before the dot has just been recognized, the first symbol after the dot is to be recognized next." For technical reasons we also need the items $[S' \rightarrow \cdot S]$ and $[S' \rightarrow \cdot S \epsilon]$, where $S'$ is a fresh symbol. We formally define the set of all items as

$$I = \{[A \rightarrow \alpha \cdot \beta] \mid A \rightarrow \alpha\beta \in P^1 \land (\alpha \neq \epsilon \lor A = S')\}$$

where $P^1$ represents the augmented set of rules, consisting of the rules in $P$ plus the extra rule $S' \rightarrow S$.

We now have

Algorithm 1 (Left-corner)

$A^{LC} = (T, \text{Alph}, \text{Init}, \rightarrow^*, \text{Fin})$, where $\text{Alph} = \{1\}$, 
$\text{Init} = [S' \rightarrow \cdot S], \text{Fin} = [S' \rightarrow S \epsilon]$, and transitions are allowed according to the following clauses.

1. $(\Gamma'[B \rightarrow \beta \cdot C], \alpha v) \rightarrow (\Gamma'[B \rightarrow \beta \cdot C\gamma][A \rightarrow \alpha \cdot \alpha_1], v)$

   where there is $A \rightarrow aa \in P^1$ such that $A \not\rightarrow^* C$

2. $(\Gamma'[A \rightarrow \alpha \cdot \alpha\beta], \alpha v) \rightarrow (\Gamma'[A \rightarrow \alpha\alpha \cdot \beta], v)$

3. $(\Gamma'[B \rightarrow \beta \cdot C\gamma][A \rightarrow \cdot \alpha], v) \rightarrow (\Gamma'[B \rightarrow \beta \cdot C\gamma][D \rightarrow A \cdot \delta], v)$

   where there is $D \rightarrow A\delta \in P^1$ such that $D \not\rightarrow^* C$

4. $(\Gamma'[B \rightarrow \beta \cdot A\gamma], \alpha v) \rightarrow (\Gamma'[B \rightarrow \beta A \cdot \gamma], v)$

The transition steps can be explained informally as follows. The automaton recognizes derivations in a bottom-up order. The first type of transition states that if symbol $C$ is to be recognized next, we start by recognizing the production in the leftmost “corner” of a derivation from $C$. The second type of transition reads an input symbol if that symbol is to be recognized next.

The third and fourth clauses both deal with the situation that the item on top of stack indicates that a complete rule has been recognized. The third clause predicts another rule in a bottom-up manner, so that eventually a larger subderivation may be recognized. The fourth rule represents the case that the subderivation obtained so far is already the one needed by the item just underneath the top of stack.

The conditions using the left-corner relation $\not\rightarrow^*$ in the first and third clauses together form a feature which is called top-down (TD) filtering. TD filtering makes sure that subderivations that are being constructed bottom-up may eventually grow into subderivations with the required root. TD filtering is not necessary for a correct algorithm, but it reduces the non-determinism, and guarantees the correct-prefix property (see also Section 3.3).

Example 1 Consider the grammar with the following rules:

$$
\begin{align*}
E & \rightarrow E + T \mid T \uparrow E \mid T \\
T & \rightarrow T \ast F \mid T \ast T \ast F \mid F \\
F & \rightarrow a
\end{align*}
$$

It is easy to see that $E \not\rightarrow E, T \not\rightarrow E, T \not\rightarrow T,$ $F \not\rightarrow T$. The relation $\not\rightarrow^*$ contains $\not\rightarrow$ but from the
reflexive closure it also contains $F \in^* F$ and from the transitive closure it also contains $F \in^* E$.

The recognition of $a \cdot a$ is realised by the following sequence of configurations:

<table>
<thead>
<tr>
<th>$E' \rightarrow \cdot E$</th>
<th>$a \cdot a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E' \rightarrow E[F \rightarrow a \cdot]$</td>
<td>$a$</td>
</tr>
<tr>
<td>$E' \rightarrow E[T \rightarrow F \cdot]$</td>
<td>$a$</td>
</tr>
<tr>
<td>$E' \rightarrow E[T \rightarrow T \cdot F \cdot]$</td>
<td>$a$</td>
</tr>
<tr>
<td>$E' \rightarrow E[T \rightarrow T \cdot F \cdot]$</td>
<td>$a$</td>
</tr>
<tr>
<td>$E' \rightarrow E[F \rightarrow a \cdot]$</td>
<td>$a$</td>
</tr>
<tr>
<td>$E' \rightarrow E[T \rightarrow T \cdot F \cdot]$</td>
<td>$a$</td>
</tr>
<tr>
<td>$E' \rightarrow E[T \rightarrow T \cdot F \cdot]$</td>
<td>$a$</td>
</tr>
</tbody>
</table>

Note that since the automaton does not use any lookahead, Step 3 may also have replaced $[T \rightarrow F \cdot]$ by any other item besides $[T \rightarrow T \cdot F \cdot]$ whose rhs starts with $T$ and whose lhs satisfies the condition of top-down filtering with regard to $E$, i.e. by $[T \rightarrow T \cdot F \cdot], [E \rightarrow T \cdot \uparrow E]$ or $[E \rightarrow T \cdot]$.

LC parsing with $k$ symbols of lookahead can handle deterministically the so-called LC($k$) grammars. This class of grammars is formalized in [18]. How LC parsing can be improved to handle common suffixes efficiently is discussed in [11]; in this paper we restrict our attention to common prefixes.

3 PLR, ELR, AND CP PARSING

In this section we investigate a number of algorithms which exhibit a better treatment of common prefixes.

3.1 PREDICTIVE LR PARSING

Predictive LR (PLR) parsing with $k$ symbols of lookahead was introduced in [24] as an algorithm which yields efficient parsers for a subset of the LR($k$) grammars [23] and a superset of the LC($k$) grammars. How deterministic PLR parsing succeeds in handling a larger class of grammars (the PLR($k$) grammars) than the LC($k$) grammars can be explained by identifying PLR parsing for some grammar $G$ with LC parsing for some grammar $G'$ which results after applying a transformation called left-factoring.

Left-factoring consists of replacing two or more rules $A \rightarrow \alpha \beta_1 \alpha \beta_2 \ldots$ with a common prefix $a$

by the rules $A \rightarrow \alpha A'$ and $A' \rightarrow \beta_1 \beta_2 \ldots$, where $A'$ is a fresh nonterminal. The effect on LC parsing is that a choice between rules is postponed until after all symbols of $a$ are completely recognized. Investigation of the next $k$ symbols of the remaining input may then allow a choice between the rules to be made deterministically.

The PLR algorithm is formalized in [24] by transforming a PLR($k$) grammar into an LL($k$) grammar and then assuming the standard realization of LL($k$) parsing. When we consider nondeterministic top-down parsing instead of LL($k$) parsing, then we obtain the formulation of nondeterministic PLR(0) parsing below.

We first need to define another kind of item, viz. of the form $[A \rightarrow a]$ such that there is at least one rule of the form $A \rightarrow \alpha \beta$ for some $\beta$. The set of all such items is formally defined by

$$ I^{PLR} = \{ [A \rightarrow a] \mid A \rightarrow \alpha \beta \in P^+ \land (\alpha \neq \epsilon \lor A = S') \} $$

Informally, an item $[A \rightarrow a] \in I^{PLR}$ represents one or more items $[A \rightarrow \alpha \cdot \beta] \in I$.

We now have

Algorithm 2 (Predictive LR)

$A^{PLR} = (T, Alph, Init, \rightarrow, Fin)$, where $Alph = I^{PLR}$, $Init = [S' \rightarrow ]$, $Fin = [S' \rightarrow S]$, and transitions are allowed according to the following clauses.

1. $(\Gamma[B \rightarrow \beta], a\alpha) \vdash (\Gamma[B \rightarrow \beta][A \rightarrow a], v)$
   where there are $A \rightarrow \alpha \alpha, B \rightarrow \beta C \gamma \in P^+$ such that $A \in^* C$

2. $(\Gamma[A \rightarrow a], a\alpha) \vdash (\Gamma[A \rightarrow a\alpha], v)$
   where there is $A \rightarrow \alpha \alpha \beta \in P^+$

3. $(\Gamma[B \rightarrow \beta][A \rightarrow a\alpha], v) \vdash (\Gamma[B \rightarrow \beta][D \rightarrow A], v)$
   where $A \rightarrow a \in P^+$ and where there are $D \rightarrow A\delta, B \rightarrow \beta C \gamma \in P^+$ such that $D \in^* C$

4. $(\Gamma[B \rightarrow \beta][A \rightarrow a\alpha], v) \vdash (\Gamma[B \rightarrow \beta A\gamma], v)$
   where $A \rightarrow \alpha \in P^+$ and where there is $B \rightarrow \beta A \gamma \in P^+$

Example 2 Consider the grammar from Example 1. Using Predictive LR, recognition of $a \cdot a$ is realised by the following sequence of configurations:
If we compare these configurations with those reached by the LC recognizer, then we see that here after Step 3 the stack element \([T \mapsto T]\) represents both \([T \mapsto T \cdot \star F]\) and \([T \mapsto T \cdot \star F]\), so that non-determinism is reduced. In this step still some non-determinism remains, since Step 3 could also have replaced \([T \mapsto F]\) by \([E \mapsto T]\), which represents both \([E \mapsto T \cdot \star E]\) and \([E \mapsto T \cdot \star E]\).

3.2 Extended LR Parsing

An extended context-free grammar has right-hand sides consisting of arbitrary regular expressions over \(V\). This requires an LR parser for an extended grammar (an ELR parser) to behave differently from normal LR parsers.

The behaviour of a normal LR parser upon a reduction with some rule \(A \mapsto \alpha\) is very simple: it pops \([\alpha]\) states from the stack, revealing, say, state \(Q\); it then pushes state \(\text{goto}(Q, A)\). (We identify a state with its corresponding set of items.)

For extended grammars the behaviour upon a reduction cannot be realised in this way since the regular expression of which the rhs is composed may describe strings of various lengths, so that it is unknown how many states need to be popped.

In [16] this problem is solved by forcing the parser to decide at each call \(\text{goto}(Q, X)\) whether

a) \(X\) is one more symbol of an item in \(Q\) of which some symbols have already been recognized, or whether

b) \(X\) is the first symbol of an item which has been introduced in \(Q\) by means of the closure function.

In the second case, state \(\text{goto}(Q, X)\) is pushed on top of state \(Q\) as usual. In the first case, however, state \(Q\) on top of the stack is replaced by state \(\text{goto}(Q, X)\). This is safe since we will never need to return to \(Q\) if after some more steps we succeed in recognizing some rule corresponding with one of the items in \(Q\).

A consequence of the action in the first case above is that upon reduction we need to pop only one state off the stack. A further consequence of this scheme is that deterministic parsing only works if a choice between case a) and case b) can be uniquely made. Further work in this area is reported in [10], which treats non-deterministic ELR parsing and therefore does not regard it as an obstacle to a working parser if the above-mentioned choice cannot be uniquely made.

We are not concerned with extended context-free grammars in this paper. However, a very interesting algorithm results from ELR parsing if we restrict its application to ordinary context-free grammars. (We will maintain the name "extended LR" to stress the origin of the algorithm.) This results in the non-deterministic ELR(0) algorithm that we describe below.

If we define the closure function from \(\mathcal{P}(I)\) to \(\mathcal{P}(I)\) by

\[
closure(Q) = Q \cup \{[A \mapsto \alpha] \mid \exists B \mapsto \beta \in Q. C \mapsto \gamma \in Q \wedge C \mapsto \star A\}
\]

then the usual goto function from \(\mathcal{P}(I) \times V\) to \(\mathcal{P}(I)\) for normal LR(0) parsing is defined by

\[
\text{goto}(Q, X) = \text{closure}([A \mapsto \alpha X \cdot \beta] \mid [A \mapsto \alpha \cdot X \beta] \in Q)\]

For ELR parsing however, we need two goto functions, \(\text{goto}_1\) and \(\text{goto}_2\), one for nonkernel items (items of form \([A \mapsto \alpha \beta]\)) and one for kernel items (the others).\(^3\) These are defined by

\[
\text{goto}_1(Q, X) = \text{closure}([A \mapsto \alpha X \cdot \beta] \mid [A \mapsto \alpha \cdot X \beta] \in Q)
\]

\[
\text{goto}_2(Q, X) = \text{closure}([A \mapsto \alpha X \cdot \beta] \mid [A \mapsto \alpha \cdot X \beta] \in Q \wedge \alpha \neq \epsilon)
\]

At each shift (where \(X\) is some terminal) and each reduce with some rule \(A \mapsto \alpha\) (where \(X = A\)) we may non-deterministically apply \(\text{goto}_1\), which corresponds with case b), or \(\text{goto}_2\), which corresponds with case a). Of course, one or both may not be defined on \(Q\) and \(X\), because \(\text{goto}_i(Q, X)\) may be \(\emptyset\), for \(i \in \{1, 2\}\).

We shortly present the ELR algorithm analogously to the LC and PLR algorithms. First, we remark that when using \(\text{goto}_1\) and \(\text{goto}_2\), each reachable set of items contains only items of the form \(A \mapsto \alpha \cdot \beta\), for some fixed string \(\alpha\), plus some nonkernel items. We will ignore the nonker-

\(^3\)Contrary to custom we treat the initial item \([S' \mapsto \epsilon]\) also as a nonkernel item.
nel items since they can be derived from the kernel items by means of the closure function.

This suggests representing each set of items by a new kind of item of the form \([\{A_1, A_2, \ldots, A_n\} \rightarrow \alpha]\), which represents all items \(A \rightarrow \alpha \bullet \beta\) for some \(\beta\) and \(A \in \{A_1, A_2, \ldots, A_n\}\). The set of all such items is formally given by

\[
I^{ELR} = \{\Delta \rightarrow \alpha | \emptyset \subseteq \Delta \subseteq \{A | A \rightarrow \alpha \beta \in P^1\} \land (\alpha \neq \epsilon \land \Delta = \{S\})\}
\]

where we use the symbol \(\Delta\) to range over sets of nonterminals.

Not all items in \(I^{ELR}\) may actually be used in the recognition process.

We now have

**Algorithm 3 (Extended LR)**

\(A^{ELR} = \langle T, \text{Alph}, \text{Init}, \cdot, \text{Fin} \rangle\), where \(\text{Alph} = I^{ELR}, \text{Init} = \{[S'] \rightarrow \}, \text{Fin} = \{[S'] \rightarrow \}\), and transitions are defined according to the following clauses.

1. \(\{\Delta \rightarrow \beta, av\} \vdash (\{\Delta \rightarrow \beta | [\Delta' \rightarrow \alpha]\}, v)\)
   where \(\Delta' = \{A | \exists A \rightarrow \alpha a, B \rightarrow \beta C \gamma \in P^1[\Delta \cup A \cup C]\}\) is non-empty

2. \(\{\Delta \rightarrow \alpha, av\} \vdash (\{\Delta' \rightarrow \alpha a\}, v)\)
   where \(\Delta' = \{A | A \in \Delta \rightarrow \alpha a \beta \in P^1\}\) is non-empty

3. \(\{\Delta \rightarrow \beta | [\Delta' \rightarrow \alpha]\}, v) \vdash (\{\Delta \rightarrow \beta | [\Delta'' \rightarrow \alpha]\}, v)\)
   where there is \(A \rightarrow \alpha \in P^1\) with \(A \in \Delta'\), and \(\Delta'' = \{D | \exists D \rightarrow A \delta, B \rightarrow \beta C \gamma \in P^1[\Delta \cup D \cup A \cup C]\}\) is non-empty

4. \(\{\Delta \rightarrow \beta | [\Delta' \rightarrow \alpha]\}, v) \vdash (\{\Delta'' \rightarrow \beta \alpha\}, v)\)
   where there is \(B \rightarrow \alpha \in P^1\) with \(A \in \Delta'\), and \(\Delta'' = \{B | B \in \Delta \rightarrow \beta A \gamma \in P^1\}\)

Note that Clauses 1 and 3 correspond with the application of \(go\) (on a terminal or nonterminal, respectively) and that Clauses 2 and 4 correspond with the application of \(go\).

**Example 3** Consider again the grammar from Example 1. Using the ELR algorithm, recognition of \(a \ast a\) is realised by the following sequence of configurations:

\[
\begin{align*}
1 & \{E' \rightarrow \} \\
2 & \{E' \rightarrow \{F \rightarrow a\} \ast \} \\
3 & \{E' \rightarrow \{T \rightarrow F\} \ast \} \\
4 & \{E' \rightarrow \{T \rightarrow T \ast\} \ast \} \\
5 & \{E' \rightarrow \{T \rightarrow T \ast]\[\{F \rightarrow a\} \} \\
6 & \{E' \rightarrow \{T \rightarrow T \ast\} \ast \} \\
7 & \{E' \rightarrow \{T, E \rightarrow T\} \} \\
8 & \{E' \rightarrow E\}
\end{align*}
\]

If we compare these configurations with those reached by the PLR recognizer, then we see that here after Step 3 the stack element \([T \rightarrow T \ast\] represents both \([T \rightarrow T \ast F]\) and \([T \rightarrow T \ast \ast F]\), but also \([E \rightarrow T \ast]\) and \([E \rightarrow T \ast E]\), so that non-determinism is even further reduced.

A simplified ELR algorithm, which we call the *pseudo ELR algorithm*, results from avoiding reference to \(\Delta\) in Clauses 1 and 3. In Clause 1 we then have a simplified definition of \(\Delta'\), viz.

\(\Delta' = \{A | \exists A \rightarrow a a, B \rightarrow \beta C \gamma \in P^1[A \cup C]\}\), and in the same way we have in Clause 3 the new definition \(\Delta'' = \{D | \exists D \rightarrow A \delta, B \rightarrow \beta C \gamma \in P^1[D \cup A \cup C]\}\).

Pseudo ELR parsing can be more easily realised than full ELR parsing, but the correct-prefix property can no longer be guaranteed (see also Section 3.3). Pseudo ELR parsing is the foundation of a tabular algorithm in [27].

Our presentation of ELR parsing is more complex than the original one in [16], which also compiles much of the computation into tables. Our presentation, however, simplifies comparison with the PLR algorithm and the algorithm from the next section.

### 3.3 Common-Prefix Parsing

One of the more complicated aspects of the ELR algorithm is the treatment of the sets of nonterminals in the left-hand sides of items. A drastically simplified version of this algorithm is the basis of a tabular algorithm in [28]. Since in [28] the algorithm itself is not described but only its tabular realisation,4 we take the liberty of giving this algorithm our own name: *common-prefix (CP) parsing*, since parsing of all rules with a common prefix is done simultaneously.5

---

4 An attempt has been made in [26] but this paper does not describe the algorithm in its full generality.

5 The original algorithm in [28] applies an optimization concerning unit rules (rules of the form \(A \rightarrow B\)) following [4]. This is irrelevant to our discussion however.
The simplification of this algorithm with regard to the ELR algorithm consists of omitting the sets of nonterminals in the left-hand sides of items. This results in

\[ I^{CP} = \{ \{ \rightarrow a \} \mid A \rightarrow a\beta \in P^1 \} \]

We now have

\textbf{Algorithm 4 (Common-prefix)}

\[ A^{CP} = (T, \text{Alph}, \text{Init}, \tau, \text{Fin}), \text{ where Alph} = I^{CP}, \text{Init} = \{ \rightarrow \}, \text{Fin} = \{ \rightarrow S \}, \text{ and transitions are allowed according to the following clauses.} \]

1. \( (\Gamma \rightarrow \beta), \alpha v \rightarrow (\Gamma \rightarrow \beta \rightarrow a), v \) where there are \( A \rightarrow \alpha a, B \rightarrow \beta C \gamma \in P^1 \) such that \( A \overset{\tau}{\rightarrow} C \)

2. \( (\Gamma \rightarrow \alpha), \alpha v \rightarrow (\Gamma \rightarrow \alpha a), v \) where there is \( A \rightarrow a\beta \in P^1 \)

3. \( (\Gamma \rightarrow \beta \rightarrow a), v \rightarrow (\Gamma \rightarrow \beta \rightarrow a), v \) where there are \( A \rightarrow a, D \rightarrow A\epsilon, B \rightarrow \beta C \gamma \in P^1 \) such that \( D \overset{\tau}{\rightarrow} C \)

4. \( (\Gamma \rightarrow \beta \rightarrow a), v \rightarrow (\Gamma \rightarrow \beta A), v \) where there are \( A \rightarrow a, B \rightarrow \beta A \gamma \in P^1 \)

The algorithms presented in the previous sections all share the \textit{correct-prefix property}, which means that in case of incorrect input the parser does not read past the first incorrect character. The simplification which leads to the CP algorithm inevitably results in the correct-prefix property to be lost.

\textbf{Example 4} Consider again the grammar from Example 1. It is clear that \( a + a \dagger a \) is not a correct string according to this grammar. The CP algorithm may go through the following sequence of configurations:

<table>
<thead>
<tr>
<th>( T_i )</th>
<th>( \alpha )</th>
<th>( \beta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \rightarrow \rightarrow \alpha )</td>
<td>( \rightarrow \rightarrow a )</td>
<td>( \alpha + a \dagger a )</td>
</tr>
<tr>
<td>( \rightarrow \rightarrow \beta )</td>
<td>( \rightarrow \rightarrow F )</td>
<td>( \alpha + a \dagger a )</td>
</tr>
<tr>
<td>( \rightarrow \rightarrow T )</td>
<td>( \rightarrow \rightarrow \beta \rightarrow F )</td>
<td>( \alpha + a \dagger a )</td>
</tr>
<tr>
<td>( \rightarrow \rightarrow E \rightarrow \beta \rightarrow T )</td>
<td>( \rightarrow \rightarrow \beta \rightarrow T )</td>
<td>( \alpha + a \dagger a )</td>
</tr>
</tbody>
</table>

We see that in Step 9 the first incorrect symbol \( \dagger \) is read, but recognition then continues. Eventually, the recognition process is blocked in some unsuccessful configuration, which is guaranteed to happen for any incorrect input\(^6\). In general, however, after reading the first incorrect symbol, the algorithm may perform an unbounded number of steps before it halts. (Imagining what happens for input of the form \( a + a \dagger a + a + \ldots + a \).)

Note that in this case, the recognizer might have detected the error at Step 9 by investigating the item \( \{ \rightarrow E \dagger \} \). In general however, items unboundedly deep in the stack need to be investigated in order to regain the correct-prefix property.

\[ \square \]

\section{4 Tabular Parsing}

Non-deterministic push-down automata can be realised efficiently using parse tables [1]. A parse table consists of sets \( T_{i,j} \) of items, for \( 0 \leq i \leq j \leq n \), where \( a_1 \ldots a_n \) represents the input. The idea is that an item is only stored in a set \( T_{i,j} \) if the item represents recognition of the part of the input \( a_1 \ldots a_j \). The table is gradually filled, first with items which can be added irrespective of other items, then with items whose insertion into some set \( T_{i,j} \) is justified by other items in other sets in the table.

We will first discuss a tabular form of CP parsing, since this is the most simple parsing technique from Section 3. We will then move on to the more difficult but also more interesting ELR technique, and apply an optimization which gives the tabular variant properties not shared by non-deterministic ELR parsing itself.\(^7\) Tabular PLR parsing is fairly straightforward and will not be discussed in this paper.

\subsection{4.1 Tabular CP Parsing}

For CP parsing we can give the following tabular version:

\textbf{Algorithm 5 (Tabular common-prefix)}

Sets \( T_{i,j} \) of the table are to be subsets of \( I^{CP} \).

Start with an empty table. Add \( \{ \rightarrow \} \) to \( T_{0,0} \).

Perform one of the following steps until no more items can be added.

1. Add \( \{ \rightarrow a \} \) to \( T_{i-1,i} \) for \( a = a_i \) and \( \{ \rightarrow \beta \} \in T_{j,i-1} \)

\[ ^6 \text{unless the grammar is cyclic, in which case the parser may not terminate, both on correct and on incorrect input} \]

\[ ^7 \text{This is reminiscent of the admissibility tests [8], which are only applicable to tabular realisations of logical push-down automata, but not to these automata themselves.} \]
where there are \( A \rightarrow aa, B \rightarrow \beta C \gamma \in P^1 \) such that \( D \vdash^* C \).

2. Add \([\rightarrow a]/a \) to \( T_{j,i} \).
   \( a = a_i \) and \([\rightarrow a]/a \in T_{j,i-1}\)
   where there is \( A \rightarrow aa \beta \in P^1 \).

3. Add \([\rightarrow a]/a \) to \( T_{j,i} \).
   \( a = a_i \) and \([\rightarrow a]/a \in T_{j,i} \)
   where there are \( A \rightarrow \alpha, D \rightarrow A \gamma, B \rightarrow \beta C \gamma \in P^1 \) such that \( D \vdash^* C \).

4. Add \([\rightarrow a]/a \) to \( T_{k,i} \).
   \( a = a_i \) and \([\rightarrow a]/a \in T_{k,i} \)
   where there are \( A \rightarrow a, B \rightarrow \beta A \gamma \in P^1 \).

Report recognition of the input if \([\rightarrow \Sigma]/\Sigma \in T_{0,n} \).

**Example 5** Consider again the grammar from Example 1 and the (incorrect) input \( a \vdash a \vdash a \).
After execution of the tabular common-prefix algorithm, the table is as given by Figure 1.

The items which correspond with those from Example 4 are labelled with \((0),(1),\ldots\).
These labels also indicate the order in which these items are added to the table.

A form of tabular CP parsing without top-down filtering (i.e., without the checks concerning the left-corner relation \( \vdash^* \)) is the main algorithm in [28].

Without the use of top-down filtering, the references to \([\rightarrow \beta]/\beta \) in Clauses 1 and 3 are clearly not of much use any more. When we also remove the use of these items, then we obtain the following new versions of these clauses:

1. Add \([\rightarrow a]/a \) to \( T_{i-1,i} \).
   where there is \( A \rightarrow aa \in P^1 \).

3. Add \([\rightarrow a]/a \) to \( T_{j,i} \).
   where there are \( A \rightarrow \alpha, D \rightarrow A \gamma \in P^1 \).

In the resulting algorithm, no set \( T_{i,j} \) depends on any set \( T_{k,h} \) with \( k < i \).
In [22] this fact is used to construct a parallel parser with \( n \) processors \( P_1, \ldots, P_n \), with each \( P_i \) processing the sets \( T_{i,j} \) for all \( j > i \).
The flow of data is strictly from right to left, i.e., items computed by \( P_i \) are only passed on to \( P_{i+1}, \ldots, P_n \).

4.2 Tabular ELR Parsing

The tabular form of ELR parsing allows an optimization which constitutes an interesting example of how a tabular algorithm can have a property not shared by its non-deterministic origin.

First note that we can compute the columns of a parse table strictly from left to right, that is, for fixed \( i \) we can compute all sets \( T_{j,i} \) before we compute the sets \( T_{j,i+1} \).

If we formulate a tabular ELR algorithm in a naive way analogously to Algorithm 5, as is done in [10], then for example the first clause is given by:

1. Add \([\Delta' \rightarrow a]/a \) to \( T_{i-1,i} \).
   where \( \Delta' = \{ A \mid \exists A \rightarrow a, B \rightarrow \beta C \gamma \in P^1[B \in \Delta \land D \vdash^* C] \} \)
   is non-empty.

However, for certain \( i \) there may be many \([\Delta \rightarrow \beta]/\Delta' \in T_{i-1,i},\) for some \( j, \) and each may give rise to a different \( \Delta' \) which is non-empty. In this way, Clause 1 may add several items \([\Delta' \rightarrow a]/a \) to \( T_{i-1,i} \), some possibly with overlapping sets \( \Delta' \).
Since items represent computation of subderivations, the algorithm may therefore compute the same subderivation several times.

We propose an optimization which makes use of the fact that all possible items \([\Delta \rightarrow \beta]/\Delta' \in T_{i-1,i} \)
are already present when we compute items in \( T_{i-1,i} \): we compute one single item \([\Delta' \rightarrow a]/a \),
where \( \Delta' \) is a large set computed using all \([\Delta \rightarrow \beta]/\Delta' \in T_{i-1,i} \), for any \( j \). A similar optimization can be made for the third clause.

We now have:

**Algorithm 6 (Tabular extended LR)**

Sets \( T_{i,j} \) of the table are to be subsets of \( I^{ELR} \). Start with an empty table. Add \([\{S']/S \rightarrow\} \) to \( T_{0,0} \).
For \( i = 1, \ldots, n \), in this order, perform one of the following steps until no more items can be added.

1. Add \([\Delta' \rightarrow a]/a \) to \( T_{i-1,i} \).
   where \( \Delta' = \{ A \mid \exists A \rightarrow a, B \rightarrow \beta C \gamma \in P^1[B \in \Delta \land D \vdash^* C] \} \)
   is non-empty.

2. Add \([\Delta' \rightarrow a]/a \) to \( T_{i,j} \).
   where \( \Delta' = \{ A \mid A \rightarrow a, B \rightarrow \beta C \gamma \in P^1[B \in \Delta \land D \vdash^* C] \} \)
   is non-empty.

3. Add \([\Delta' \rightarrow a]/a \) to \( T_{i,j} \).
   where \( \Delta' = \{ A \mid A \rightarrow a \in P^1 \} \)
   is non-empty.

4. Add \([\Delta' \rightarrow \beta a]/a \) to \( T_{i,j} \).
   where \( \Delta' \) is a large set computed using all \([\Delta \rightarrow \beta]/\Delta' \in T_{i-1,i} \).
where there is $A \rightarrow \alpha \in P^t$ with $A \in \Delta'$, and 
$\Delta'' = \{ B \in \Delta \mid B \rightarrow \beta A \gamma \in P^t \}$ is non-empty.

Report recognition of the input if $[\{S'\} \rightarrow S] \in T_{0,n}$.

Informally, the top-down filtering in the first and third clauses is realised by investigating all left corners $D$ of nonterminals $C$ (i.e. $D \not\rightarrow C$) which are expected from a certain input position. For input position $i$ these nonterminals $D$ are given by

$S_i = \{ D \mid \exists \beta \exists \gamma \Delta \rightarrow \beta \in T_j, \exists B \rightarrow BC \gamma \in P^t \\
B \in \Delta \land D \not\rightarrow C \} \}$

Provided each set $S_i$ is computed just after completion of the $i$-th column of the table, the first and third clauses can be simplified to:

1. Add $[\Delta' \rightarrow \alpha]$ to $T_{i-1,i}$ for $a = a_i$ where $\Delta' = \{ A \mid A \rightarrow \alpha \alpha \in P^t \} \cap S_{i-1}$ is non-empty.

2. Add $[\Delta'' \rightarrow A]$ to $T_{j,i}$ for $[\Delta' \rightarrow \alpha] \in T_{j,i}$ where there is $A \rightarrow \alpha \in P^t$ with $A \in \Delta'$, and

$\Delta'' = \{ D \mid \beta \rightarrow D \rightarrow \alpha \beta \in P^t \} \cap S_j$ is non-empty which may lead to more practical implementations.

Note that we may have that the tabular ELR algorithm manipulates items of the form $[\Delta \rightarrow \alpha]$ which would not occur in any search path of the non-deterministic ELR algorithm, because in general such a $\Delta$ is the union of many sets $\Delta'$ of items $[\Delta' \rightarrow \alpha]$ which would be manipulated at the same input position by the non-deterministic algorithm in different search paths.

With minor differences, the above tabular ELR algorithm is described in [28]. A tabular version of pseudo ELR parsing is presented in [27].

4.3 FINDING AN OPTIMAL TABULAR ALGORITHM

In [20] Schabes derives the LC algorithm from LR parsing in the same way that ELR parsing can be derived from LR parsing (Section 3.2). The LC algorithm is obtained by not only splitting up the goto function into goto1 and goto2 but also splitting up goto2 even further, so that it non-deterministically yields the closure of one single kernel item. (This idea was described earlier in [10], and more recently in [15].)

Schabes then argues that the LC algorithm can be determined (i.e. made more deterministic) by manipulating the goto functions. One application of this idea is to take a fixed grammar and choose different goto functions for different parts of the grammar, in order to tune the parser to the grammar.

In this section we discuss a different application of this idea: we consider various goto functions which are global, i.e. which are the same for all parts of a grammar. One example is ELR
parsing, as its goto function can be seen as a
determinized version of the goto function of LC
parsing. In a similar way we obtain PLR pars-
ing. Traditional LR parsing is obtained by taking
the full determinization, i.e. by taking the normal
goto function which is not split up.\footnote{Schabes
more or less also argues that LC itself can be
obtained by determinizing a non-deterministic TD
generator. (In lieu of TD parsing he mentions Earley's
generator, which is its tabular realization.)}

We conclude that we have a family consisting
of LC, PLR, ELR, and LR parsing, which are
increasingly deterministic. In general, more
deterministic algorithms require more parser states.
For example, the LC algorithm requires a num-
ber of states (the items in \( I \)) which is linear in
the size of the grammar. By contrast, the LR
algorithm requires a number of states (the sets of
items) which is exponential in the size of the
grammar.

The differences in the number of states com-
plicates the choice of a tabular algorithm as the
one giving optimal behaviour for all grammars.
If a grammar is very simple, then a sophisticated
algorithm such as LR may allow completely
deterministic parsing, which requires a linear
number of entries to be added to the parse table, mea-
sured in the size of the grammar.

If, on the other hand, the grammar is very am-
biguous such that even LR parsing is very non-
deterministic, then the tabular realisation may at
worst add each state to each set \( T_{i,j} \), so that the
more states there are, the more work the parser
needs to do. This favours simple algorithms such
as LC over more sophisticated ones such as LR.
Furthermore, if more than one state represents
the same subderivations, then recognition of that
subderivations may be done more than once, which
leads to parse forests (compact representations of
collections of parse trees) which are not optimally
dense [1, 17, 13].

Schabes proposes to tune a parser to a gram-
mar, or in other words, to use a combination of
parsing techniques in order to find an optimal
parser for a certain grammar. This idea has un-
til now not been realised. However, when we try
to find a single parsing algorithm which performs
well for all grammars, then the tabular ELR algo-
rithm from Section 4.2 may be a serious can-
didate, for the following reasons:

- For all \( i, j \), and \( \alpha \) at most one item of the
  form \([\Delta \rightarrow \alpha]\) is added to \( T_{i,j} \). Therefore, identical
  subderivations are not computed more than
  once. [Note that this is a consequence of the
  optimization of top-down filtering at the be-
  ginning of Section 4.2.) Note that this also
  holds for the tabular CP algorithm.
- ELR parsing guarantees the correct prefix
  property, contrary to the CP algorithm. This
  prevents computation of all subderivations
  which are useless with regard to the already
  processed input.
- ELR parsing is more deterministic than LC
  and PLR parsing, because it allows process-
ing of all common prefixes to be shared. It is
  hard to imagine a practical parsing technique
  more deterministic than ELR parsing which
  also satisfies the previous two properties (see
  also Section 7).

5 Data structures

The most straightforward data structure for han-
dling items in \( I^{CP} \) is a trie. The vertices of the
trie represent the items in \( I^{CP} \). From a vertex
representing \([\rightarrow \alpha]\) to a vertex represent-
ing \([\rightarrow \alpha A]\) there is an edge labelled \( A \). In order
be able to detect when a complete rhs has been
recognized, each vertex representing \([\rightarrow \alpha]\) should
have a label \( COMPLETE(\alpha) \), which is a set of all
nonterminals \( A \) such that \( A \rightarrow \alpha \in P^T \).

For the purpose of top-down filtering we also
need to label each vertex representing \([\rightarrow \alpha]\)
with \( LHS(\alpha) \), which is a set of all \( A \) such
that \( A \rightarrow \alpha \beta \in P^T \), for some \( \beta \). Note that
\( COMPLETE(\alpha) \subseteq LHS(\alpha) \), for all items \([\rightarrow \alpha]\).

Example 6 The following represents the items
of \( I^{CP} \) in a trie, according to the grammar from
Example 1.

\[
\begin{align*}
(\{E\} \oplus \{F\}) & \rightarrow \{E\} \\
(\{T, E\} \oplus \{F\}) & \rightarrow \{T\} \\
(\{T\} \oplus \{F\}) & \rightarrow \{T\} \\
(\{E\} \oplus \{T\}) & \rightarrow \{E\} \\
(\{E\} \oplus \{F, T\}) & \rightarrow \{F, T\} \\
(\{T\} \oplus \{F\}) & \rightarrow \{T\} \\
(\{T\} \oplus \{F, T\}) & \rightarrow \{F, T\} \\
(\{T\} \oplus \{F\}) & \rightarrow \{T\} \\
(\{T\} \oplus \{F, T\}) & \rightarrow \{F, T\} \\
\end{align*}
\]

The sets \( LHS(\alpha) \) are given beneath the vertices, the
non-empty \( COMPLETE(\alpha) \) are given above the ver-
tices. \( \square \)

The data structure used in [22] for CP pars-
ing without top-down filtering is inspired by LR.
tables. First we take an initial set of items to be the set of all nonkernel items, and then we generate new sets of items using a version of the goto function which is stripped of the application of the closure function. Each set of items then corresponds with one LR state, and a table representing the goto function and a table containing reduce entries are constructed, analogously to those normally used for LR(0) parsing. It is easy to see that this structure is equivalent to a trie.

Essentially the same ideas may be used for the representation of items in $I^{PLR}$ and $I^{ELR}$, except that we should then partition the items into a collection of tries. For PLR parsing, each trie represents the items $[A \rightarrow a]$ for some fixed $A$.

For ELR parsing, the root of each trie represents an item $[\Delta \rightarrow X]$ such that for some other item $[\Delta' \rightarrow \beta]$ we have $\Delta = \{A | \exists A \rightarrow XA \in P^+ | B \in \Delta' \wedge A \rightarrow^* C \}$. There is an edge labelled $X$ from a vertex representing $[\Delta \rightarrow a]$ to a vertex representing $[\Delta' \rightarrow \alpha X]$ if $\Delta' = \{A \in \Delta | A \rightarrow \alpha X \beta \in P^+ \}$. Note that the tries for $I^{ELR}$ may have subtries in common.

For tabular ELR parsing we may need more tries. The root of each trie represents an item $[\Delta \rightarrow X]$ such that for some $S_i$ which may be calculated during parsing

$$\Delta = \{A | A \rightarrow X \in P^+ \} \cap S_i$$

$$= \bigcup_{[\Delta' \rightarrow \beta] \in T_{i,i}} \{A | \exists A \rightarrow XA \in P^+ | B \in \Delta' \wedge A \rightarrow^* C \}$$

Since for all $X$ the number of subsets of $[A | A \rightarrow X \in P^+]$ is generally small, the calculation of $[A | \exists A \rightarrow XA \in P^+ | B \in \Delta' \wedge A \rightarrow^* C \}$ for $X$ and $[\Delta' \rightarrow \beta]$ may be computed statically. Also the unions of these sets for different $[\Delta' \rightarrow \beta]$ may be computed statically. These results may be stored into tables which can be used during parsing. In addition, the outcome of the complete calculation may be memoized for $X$ and $i$.

This implementation of tabular ELR parsing is in contrast to the algorithm in [28], which makes use of a single trie for $I^{CR}$, combined with (elementwise) calculation of the $S_i$ and of the sets of nonterminals in the left-hand sides of items, which obviously does not give optimal time complexities.

6 Epsilon Rules

There are two ways in which epsilon rules may form an obstacle to bottom-up parsing. The first is non-termination for backtrack parsing caused by a special form of left recursion, called hidden left recursion [13, 14], which means that $A \rightarrow Ba$, $B \rightarrow \epsilon$, and $\alpha \rightarrow^* A\beta$, for some $A, B, \alpha$, and $\beta$ (the left recursion is "hidden" by the empty-generating nonterminal $B$).

Secondly, we explain informally how for tabular parsing, epsilon rules may interfere with optimization of top-down filtering. In Section 4.2 we made use of the fact that before we calculate the sets $T_{j,i}$ for some fixed $i$, we have already calculated the sets $T_{j,i-1}$. We can then prepare for top-down filtering by combining all $[\Delta \rightarrow \beta] \in T_{j,i-1}$ in $S_{i-1}$ as explained.

However, suppose that we allow epsilon rules and that for some $[\Delta' \rightarrow \beta'] \in T_{j',i-1}$, for some $j'$, we have a rule $A \rightarrow \beta'BC$, where $A \in \Delta'$, and $B \rightarrow \epsilon$. Then for finding a derivation $B \rightarrow^* \epsilon$ we need top-down filtering, for which we have combined in $S_{i-1}$ all $[\Delta \rightarrow \beta] \in T_{j,i-1}$, for any $j$. After we have found this derivation however, we find that some $[\Delta'' \rightarrow \beta'B]$ should be in $T_{j',i-1}$. But this is in conflict with our assumption that we had already found all $[\Delta \rightarrow \beta] \in T_{j,i-1}$, for any $j$.

We propose treatment of epsilon rules analogously to [13, 14], which amounts to merging epsilon-rule elimination with the parser construction, without actually transforming the grammar. We omit details because of space limitations.

7 BEYOND ELR PARSING

We have conjectured in Section 4.3 that there is no parsing algorithm more deterministic than ELR parsing which allows a practical tabular algorithm with the property that subderivations may never be computed more than once. Below we give an informal explanation of why we feel this conjecture to be true.

For tabular ELR parsing, the above property was ensured in Section 4.2 by applying an optimization of top-down filtering. The result of this optimization is that, for example, Clause 1 adds a single item $[\Delta \rightarrow a]$ to $T_{i-1,i}$ where a naive tabular ELR parser would add several items $[\Delta' \rightarrow a]$ to $T_{i-1,i}$ for different $\Delta'$, such that $\Delta$ is the union of all such $\Delta'$. A similar fact holds for Clause 3.

Let us now consider how this idea can be translated into the realm of LR parsing. For LR parsing, the stack symbols (the states) represent sets of items from $I$. If two states representing say $J$ and $H$ are both added to some set $T_{i,j}$ of the
table of a tabular LR parser, then we have that a subderivation is being computed twice if some kernel item belongs to both $J$ and $H$. We can avoid this by adding a single state to $T_{i,j}$ representing $J \cup H$, by analogy of the above-mentioned optimization for tabular LR parsing.

However, the number of sets of items that we will then need may become prohibitively large, even larger than the number of sets we need for traditional LR parsing. (In case we do not compile the operations on sets of items into an LR table, but compute the sets during parsing, then the algorithm actually comes down to a tabular LC algorithm.) Furthermore, merging two states leads to new problems later on when a reduction has to be performed which is only applicable for one of the two original states.

We conclude that there is no practical tabular LR parsing algorithm which has the property that subderivations cannot be computed more than once. The main obstacle is the number of sets of items which would be required to apply the above optimization. However, mixing LR parsing with simpler kinds of parsing, as suggested in [20], may lead to feasible tabular algorithms, since in that case the number of states which are required may be smaller than that for full LR parsing.

How the top-down prediction of rules or nonterminals may be incorporated in LR parsing is discussed in a number of papers, which only take into account deterministic parsing. For example, [5] describes how top-down prediction of nonterminals may be mixed with LR parsing. LR parsing is performed until a certain nonterminal $A$ can be predicted with certainty, with which we mean that it is certain that this nonterminal will occur least-most in the derivation under construction. Then a specialised LR automaton is activated for the recognition of $A$. After this has been completed, normal LR parsing continues.

A similar idea is described in [6]. LR parsing is performed until a unique item $\{ A \rightarrow \alpha \beta \}$ can be predicted with certainty. TD parsing is then activated consecutively for the symbols in $\beta$, until TD parsing cannot be done deterministically, and then LR parsing takes over recursively. After recognition of the symbols in $\beta$, normal LR parsing continues.

The generalized LC algorithm in [3] makes use of an annotated grammar, where each rhs is divided into two parts, the first being the (generalized) left corner. LR parsing is performed until we reach a state of which the set of items contains some item $\{ A \rightarrow \alpha \beta \}$ whose left corner is $\alpha$. It should then be possible to predict this item by certainty (this is a constraint on the grammar), and specialised LR automata are then activated for the consecutive recognition of the symbols in $\beta$. After this has been completed, normal LR parsing continues.

For general grammars, in particular for grammars which are very ambiguous, TD prediction with certainty of nonterminals or rules may not be possible very often. We may however generalize the above ideas by allowing non-deterministic prediction of nonterminals or rules, even if these cannot be predicted with certainty. The resulting algorithms may be used as starting-points for tabular algorithms. An example is the head-driven chart parser from [2], which is essentially a tabular realisation of non-deterministic generalized LC parsing without LR states.

Algorithms which are more deterministic than LR parsing, such as Marcus’ algorithm [12] fall outside the scope of this paper.

8 Conclusion

We have discussed a range of different parsing algorithms, which have their roots in compiler construction, expression parsing, and natural language processing. We have shown that these algorithms can be described in a common framework.

We further discussed tabular realisations of these algorithms, and concluded that we have found an optimal algorithm, which in most cases leads to parse tables containing fewer entries than for other algorithms, but which avoids computing identical subderivations more than once.

Acknowledgements

The author acknowledges valuable correspondence with Klaas Sikkil, René Leermakers, François Barthélémy, Giorgio Satta, Yves Schabes, and Frédéric Voisin.

---

9 There is a complication with generalizing the algorithm from [6] in this way, since non-deterministic prediction of different nonterminals may lead to different computations of the same subderivation. In other words, different search paths may lead to recognition of the same derivation, which is an objectionable property for a parsing algorithm. These problems do not occur for [3] or [6], provided no nonkernel items are predicted.
REFERENCES


[31] Third International Workshop on Parsing Technologies (IWPT 3), Tilburg (The Netherlands) and Durbuy (Belgium), August 1993.
Semantic-Oriented Chart Parsing with Defaults

Thomas Stürmer
University of Erlangen and Nuremberg
Computer Science, Department of Artificial Intelligence (IMMD-8)
Am Weichselgarten 9, D-91058 Erlangen, Germany
e-mail: thomas@immd8.informatik.uni-erlangen.de

ABSTRACT

We present a computational model of incremental, interactive text analysis.

The model is based on an active chart and supports interleaved syntactic and semantic processing. It can handle intra- and intermodular constraints without forcing the use of the same formalism for the description of syntactic and semantic knowledge. An essential part of the model are defaults which guide an analysis algorithm based on our approach to compute the most plausible solution. We will argue that the resulting computation can be understood as semantic-oriented parsing.

We will also show how our model can be abstracted into a NL understanding system architecture, that is applicable to a wide range of NL understanding problems.

1 INTRODUCTION

There is a small number of popular natural language (NL) systems for very special purposes, restricted domains or linguistic scopes, or poor linguistic expressive power [Sei92] in use. More general real world NL systems, which should be the goals of NLP research [PG93, page 1], are still far away.

1.1 OBSERVATIONS ON NLP

The main reason for this lack of systems lies in the enormous complexity of NL. Managing this complexity is a hard (possibly the central) problem for at least two reasons:

Nearly every NL utterance is highly ambiguous or vague in different ways. An interpretation of an utterance like “Do it with the small one!” depends on numerous factors which must be taken into account. Lacking information about these factors causes uncertainty—the main reason for misunderstanding even in natural communication. This sort of uncertainty is inherent to natural language and cannot be avoided in principle.

Ambiguity additionally increases for another reason: the lack of techniques capable to model behaviour of such complexity. Designing a model as a description for solving a problem, and in consequence building a large system, enforces detailed investigation. In terms of computer science: One has to identify modules, which should be independent models for subproblems, and relations between those modules (e.g. hierarchical, sequential, etc.), that allow to describe the initial problem in terms of the subproblems. In models for NLP, the modules usually correspond to the well known levels of linguistic abstraction. Unfortunately this modularization causes ambiguity local to those levels since they lack the information that belongs to other levels.

1.2 DESIRED BEHAVIOUR

Supposing there is no better modularisation than the traditional one at hand, one has to develop and use techniques that support the availability of as much information as possible in a sophisticated way. Obviously a very strong interaction between those modules for practical NLP is necessary: Intermediate results obtained by one of them can reduce a great number of ambiguities that occur in another, and vice versa. Interaction between modules doesn’t make sense without synchronous and stepwise processing. For “real time” applications of NLP those steps should be ir-incremental [Wir90, Wir92], i.e. incremental from left to right (in the order words occur in an NL utterance).\(^1\)

As mentioned above, communication of infor-

\(^1\) This proposition is quite intuitive. It does not necessarily apply for speech signal processing, indexing techniques in semantics, etc.
formation will not generally lead to one single solution of a NLP problem. In spite of that we usually want a NL understanding system not to process all possible but only the most plausible interpretation of a given input.

To achieve this behaviour, an algorithm for solving NLP problems has to make good guesses about lacking information and the way problem solving should proceed. If bad guesses have been made, the algorithm has to provide means for their revision.

Based on these observations we present in the following sections a computational model of incremental, interactive text analysis. We focus especially on the linguistic levels of syntax and semantics.

The model is based on an active chart and supports interleaved syntactic and semantic processing. It can handle intra- and intermodular constraints without the use of the same formalism for the description of syntactic and semantic knowledge. An essential part of the model are defaults which guide an analysis algorithm based on our approach to compute the most plausible solution. We will argue that the resulting computation can be understood as semantic-oriented parsing.

We will also show how our model can be abstracted into a NL understanding system architecture, that is applicable to a wide range of NL understanding problems.

2 PRELIMINARIES

2.1 BASIC LINGUISTIC ASSUMPTIONS

We assume that NL is *compositional* in its structure and meaning. The meaning of an utterance is a function of the meaning of its parts and their structural relations.

In written texts those parts are *morphemes, words, phrases, sentences, discourse segments*, and at last *texts*.

Structural descriptions of words, phrases, and sentences uniquely consist of descriptions of the *constituent structure* (c-structure) and *functional structure* (f-structure).2

Our notion of meaning is that of a *truth value semantics* [vSW91]. We distinguish between3

- *semantic potential*, which is generated by *semantic construction* using just abstract *linguistic knowledge*,
- *discourse representation*, generated by *semantic resolution* using contextual knowledge and discourse information, and
- *relevant information contents*, generated by *semantic inference* using world knowledge.

The meaning of a word, a phrase, or a sentence is represented by *objects* in discourse and *propositions* about these objects.

Figure 1 shows a simple functional model of the participating linguistic levels, the flow of information, and the use of rules and knowledge sources.

```
written language ⇒ morphological analysis
                     ⇒ lexical analysis
                     ⇒ syntactic analysis
                     ⇒ semantic construction
                     ⇒ semantic resolution
                     ⇒ semantic inference

internal representation
```

Figure 1: Levels of linguistic abstraction for a simple linear model of text understanding

2.2 MANAGING INTERACTION

Approaches for managing interaction that are currently discussed in literature can be devied into two classes:

*Interaction by architecture:*

Interaction is realized explicitly by a great variety of means, eg. with *blackboards* [Cra88, Cra91], by *hierarchical systems*, [Pep93], or by *multi-agent systems* [Cra92, Wel93]. Some of

---

2While this notions originate in the LFG theory [Bre82], they can be found in most grammar formalisms [MK92].

3In the description of our notion of meaning we slightly modified [Pin93]. It differs from the classical notions in-

---

The references mentioned here are just an arbitrary selection among numerous others.
these approaches were developed or proposed especially for psycholinguistic motivated language modelling, e.g. [Br87] or the interactive incremental architecture [Gö92a].

They all have in common a more or less significant control problem. The complexity of the problem has partially become the complexity of the communication.

Interaction by unique formalism:

Interaction is realized by the use of a unique linguistic knowledge representation formalism for different linguistic levels and a unique processing mechanism, e.g. HPSG and feature unification [PS87] (the constraint pool metaphor, [F’87]). Such approaches provide an elegant means for formulating communication, which may be one reason of the popularity of HPSG.

The computational complexity is desasterous [Se92]. It seems to be difficult to cover a significant part of semantics or pragmatics without extending the pure framework [Pi93].

2.3 The notion of plausibility

There are two promising ways for getting good guesses: statistical methods, e.g. [Ney91, EN91, KN91, Han94], and psycholinguistic principles, e.g. [Fod91, H’92, Hen93].

The importance of statistics for speech recognition (statistical pattern matching) was never in doubt, and neither was the responsibility of symbolic AI for higher linguistic levels. Nowadays the impact of symbolic on probabilistic processing leads to hybrid approaches [Han94]. E.g. dynamic programming approaches to parsing [Ney91], or adaption of recognition processes to temporary circumstances [EN91] were developed.

Probabilistic measures for the occurrence of particular phenomena can be obtained and represented by statistical analysis of huge corpora. This can be achieved e.g. by training of Hidden-Markov-Models (HMMs). Each possible phenomenon has to occur at least once in the training set5, otherwise it gets the probability 0. With respect to the potentially infinite number of NL utterances one has to train different levels of linguistic abstraction (e.g. the level of syntax), obtaining several statistical models.

In ambiguous situations we call the alternative statistically plausible that gets the highest probability or score.

Psycholinguistic researchers develop models for human language performance, that behave close to results of psycholinguistic experiments. This models can be used to predict decisions humans would take on ambiguous communication situations, applied to linguistic competence modelling.

The currently most successful approach is that of SOUL-processing [H’92]: it applies semantic oriented processing principles (SOPP) to unification based language processing.

Observations on human behaviour lead to the following semantic oriented processing model: Human language processing is primarily based on structural processing. It is hr-incremental. Semantic-conceptual and discourse knowledge is immediately integrated (therefore processing is interactive). A single solution is performed, guided by SOPP:

- A good guess to handle lexical ambiguities is to take the most specific of all possibilities (lexical strength principle).
- For structural (attachment) ambiguities,
  - attach a constituent to a phrase whose lexical head was instantiated before (head attachment principle),
  - choose an attachment that assigns a θ-role to the ambiguous constituent (θ-attachment principle), or
  - attach it to the phrase currently working on (late closure principle).

We call this processing model psycholinguistically plausible.

Supposed that natural communication mainly consists of both statistically and psycholinguistically plausibly constructed utterances, an analysis strategy anticipating plausibility should succeed in most cases without having to revise guesses.

3 The Approach

In this section we concentrate on our approach for combining the NLP levels of syntactic analysis and semantic construction for sentence processing.

5Semantics-oriented unification-based language (SOUL-) processing.
In a first step a nondeterministic model of interleaved syntactic and semantic NLP is presented. The second step specifies a λ-incremental most-plausible-1st search upon this model, using default reasoning techniques.

We show the current state of implementation.

Finally we outline how our approach might be extended to a model of coherent text understanding.

3.1 The Nondeterministic Model

Following the basic assumptions of section 2.1 we chose as means for modelling the single levels:

- A feature-based grammar formalism similar to PATR-2.
- A semantic construction formalism: λ-DRT.

We represent corresponding structural and semantic information as a label of an edge in an active chart. Figure 3 shows an active chart representing a possible parse of a sentence. An active chart represents grammatical structures in inactive edges that span the corresponding part of an utterance. Active edges are used to represent hypotheses about applicable rules, or rules that where partially applied and lack at least on constituent.

The active chart allows the representation of "top-down" hypotheses and "bottom-up" construction of linguistic data. The fundamental rule describes this process in terms of interaction between active and inactive edges. It determines the formation and representation of all structural relations between the word and phrase of a sentence dependent on a grammar and a lexicon without specifying a concrete strategy.

Description of the Syntax

PATR-2-like grammar formalisms have a context-free (cf) backbone and attribute-value structures for representing c-structure and f-structure respectively [Shi86]. A grammar rule consists of a cf production rule and a number of constraint equations. The rule is annotated with a rating. See figure 2 for an example.

These grammar rules apply in the usual way both to c-structure and f-structure of constituents. See figure 3 for an example of a cf-tree representation in a chart, and for an application of the constraint equations to f-structure. An f-structure is integrated into the chart as extension of the label of its edge.

The expressive power of this formalism is sufficient for our purpose. We can handle all desired syntactical phenomena, depending on a concrete grammar.

---

*For a detailed description of active chart-parsing, see [Wi92]. For a careful elaboration of the connection between c- and f-structure and chart parsing, see [MK92].

Figure 2: A PATR-2 rule for an S.

Figure 3: A chart representation of a phrase structure tree and an application of the above PATR-2 rule to a pair of feature structures.
DESCRIPTION OF THE SEMANTICS

\( \lambda \)-DRT [Fil93, Pin93] is a combination of Montague Semantics and Discourse Representation Theory (DRT) [KR93].\(^8\) It is based on partial and predicative discours representation structures (DRS) which are combined along the constituent structure\(^9\) by \( \lambda \)-conversion in a typed \( \lambda \)-calculus, grounding on lexical entries. See figure 4 for an example.

\[
\begin{align*}
\lambda S & \ x \\
& \quad +S(z) \\
& \quad \lambda w \lambda R \\
& \quad \ y \\
& \quad \bone(y) \\
& \quad \with(w,y) \\
& \quad +R(w) \\
\lambda \lambda w \lambda R & \ with(w,x) \\
& \quad \y \\
& \quad \bone(x) \\
& \quad \with(w,y) \\
& \quad +R(w) \\
\lambda R & \ x \\
& \quad +R(x) \\
\end{align*}
\]

\( \lambda \)-DRSes

Figure 4: Construction of the semantic potential of a verbal phrase VP with \( \lambda \)-DRSes

This approach enables us to handle anaphora resolution, Generalized Quantifiers, quantifier scope ambiguity, modality, and tense.

COMBINING SYNTAX AND SEMANTICS

While words, phrases, and sentences have simultaneously corresponding c-structure, f-structure, and semantic representation, we can formulate a strong interaction constraint in our model: Only edges with valid grammatical information and a valid corresponding semantic interpretation may be inserted into a chart.

In some cases additional constraints between modules need to be expressed, e.g. for handling free word order.

For this interactions need to be reflected in our grammar rules, we have to augment their format by (1) the semantic operation that applies to the semantics of the attached constituents, and (2) additional constraints between states of modules. See figure 5 for an example.

Within this framework a solution can be defined as a inactive edge that spans the whole utterance and has the expected category.

\[
S \rightarrow NP \ VP
\]

\[
\begin{align*}
(NP \ cat) &= np \\
(VP \ cat) &= vp \\
(S \ cat) &= s \\
(NP \ head \ agr) &= (VP \ head \ agr) \\
(S \ head) &= (VP \ head) \\
(S \ head \ subj) &= (NP \ head)
\end{align*}
\]

\( \text{compose NP VP} \)

\( \text{(SYN arg1) = (SYN subj)} \wedge \text{(SEM arg1) = subj} \)

Figure 5: An augmented PATR-2 rule for including semantics.

According to the fundamental rule beginning with a initial hypothesis and the lexical entries as label of initial inactive edges, all possible solutions for syntax and semantics can be generated.

3.2 FIXING THE SEARCH STRATEGY

Each consecutive pair of an active and an inactive edge is a candidate for applying the fundamental rule. In a chart there are probably many of those candidates, but the model described above provides no order on them.

Usually these pairs are kept in an agenda. By specifying the read- and write-operations on this agenda, a concrete search strategy is determined, e.g. a "first-in first-out" strategy will result in depth first search.

For the \( \lambda \)-incremental "most-plausible-1st" strategy we have to prevent the agenda from working on implausible pairs. We cannot simply throw those pairs away: If the search algorithm has made bad guesses, they can become plausible. In pathological cases candidates may switch plausible and implausible more than once.

We achieve the desired behavior by the use of reason-maintenance techniques [Bec94]. We formulate constraints for the dependency of edges and configurations of active and inactive edges. Additionally, the constraints formulated in the grammar rules can be handled the same way.

3.3 THE CURRENT IMPLEMENTATION

The current system is based on a cf chart parser, that was extended with feature structures to deal with PATR-2 grammar rules. Further development lead in two different directions: The PATR-
2 parser was augmented by strategy rules that perform incremental AO* search [Närk92]. Alternatively the PATR-2 parser was extended by λ-DRSe, but with incrementality in mind [Fis93].

The latter semantic parser was combined with an LTMS (TRUMP, [Kiec92]), that maintains the consistency of syntactic-semantic constraints and of currently believed edges and configurations.

The parsing algorithm performs considerably semantic-oriented parsing; it processes normal sentences straightforward with minimal overhead. In "pathological" cases it produces all possible edges.

The whole system is implemented in SCM und runs under any R4RS-compliant Scheme.

3.4 Possible Extensions

Our model can be extended into different directions.

The results of our semantic constructions are DRSes. Sentential DRSes can directly be used as an input for a semantic resolution level. Charts for a sequence of texts can be put together, and the arising discourse information can be collected in a discourse edge, which has in its scope all or a number of sentences analysed before. For this level we can adopt knowledge about discourse structure. An early integration of discourse knowledge in the analysis process will be realised analog to the integration of the semantic construction level.

Semantic inference is highly dependent to the concrete application. It subsumes a great variety of different purposes. A careful integration of some of them into the chart construction process should be possible in analog.

The "lower" side, morphology and even speech, can be integrated by parsing word lattices as initial input in the chart. This will be future work in parts of the VERBMOBIL parser (Hans Weber).

4 Conclusion

We have presented an approach for interactive incremental Our approach can be characterized in short by this essential properties

- Symbolic problem description
  - modularisation as set of specialized problem solvers,
  - representing alternative data structures,
- representing guesses,
- explicit representation of control: representing alternative tasks.

- Probabilistic and psychological principle based ordering of the problem space: the notion of plausibility.
- Specification of an algorithm that
  - solves the problem in a "most plausible first" order,
  - tracks back in an "intelligent" way.

It introduces an incremental method for constructing sentential DRSes, which will help to bridge the gap between the syntactic and the discourse level.

Acknowledgements

I want to thank many dear colleagues and students for fruitful discussions and commenting. Thanks especially to Ingrid Fischer for the pictures she made.

References


THE PARSING PROBLEM
FOR TREE ADJOINING GRAMMARS*

Giorgio Satta
Department of Computer Science
University of Venice
via Torino, 155
30172 Mestre – Venezia
Italy
satta@moo.dsi.unive.it

ABSTRACT

An interesting relationship is studied between the well known computational problem of boolean matrix multiplication and the computational problem of parsing a sentence in a tree adjoining language. First, we show that any algorithm for the solution of the latter problem can easily be converted into an algorithm for boolean matrix multiplication. This reduction provides evidence that a straightforward method that improves the known time upper bound for tree adjoining grammar parsing may be hard to find. Conversely, by restricting to the subclass of linear tree adjoining languages we show how a method for the solution of the corresponding parsing problem can be obtained from any algorithm for boolean matrix multiplication. As a consequence, using less than cubic time methods for boolean matrix multiplication we can improve the already known time upper bound for linear tree adjoining grammar parsing. Both our reductions reveal which features of the tree adjoining grammar parsing problem are responsible for its claimed difficulty.

1 INTRODUCTION

The class of Tree Adjoining Grammars (TAG's) was first introduced in [4] and [3]; since then, for-

*I am grateful to Bernard Lang and Yves Schabes for helpful discussion on topics related to this paper. Part of this research was conducted while the author was a post-doctoral fellow at the Institute for Research in Cognitive Science, University of Pennsylvania, 3401 Walnut Street, Philadelphia PA 19104-6228, USA. The research was sponsored by the following grants: ARO DAA1 03-89-C-0331; DARPA N00014-89-J-1853; NSF IRI 80-16592; and Ben Franklin 913-3078C-1.

mal and computational properties of this class have been extensively investigated, both in a theoretical vein and in view of possible natural language processing applications (see for instance the presentation in [2] and the references therein). A large variety of parsing algorithms for TAG's has been proposed in the literature. Dynamic programming has proved to be the most successful paradigm and the least time upper bound that has been attested in this way is \( O(|G| |w|^6) \) for the random access model of computation, \(|G|\) being the size of the input grammar and \(|w|\) the length of the input string. In recent years, improvement of such a worst-case running time has been a common goal for many researchers, but up to present time the TAG parsing problem has strongly resisted all such attempts.

This paper reports a summary of recent work of the author on the topic of TAG parsing (see [8] and [9]). Two main results are presented and discussed. First of all, the TAG parsing problem is related to the well known computational problem of boolean matrix multiplication. This is done in such a way that time upper bounds for TAG parsing can be transferred to time upper bounds for the latter problem. More precisely, we show how any algorithm for TAG parsing that improves the \( O(|G| |w|^6) \) time upper bound can be converted into an algorithm for boolean matrix multiplication running in less than \( O(m^3) \) time, \( m \) being the order (number of rows/columns) of the input matrices. We remark that boolean matrix multiplication has been object of investigation for many years, and methods that are asymptotically faster than \( O(m^3) \) are known, but the more considerable the improvement turned out to be, the more complex the involved computation was found. Indeed, the huge constants in-
volved in the running time of the fastest methods render prohibitive their practical application, given current computer hardware. As a matter of fact then, the design of practical algorithms for boolean matrix multiplication that considerably improve the cubic time upper bound is regarded as a very difficult enterprise. Therefore, a consequence of our first result is that TAG parsing should also be considered as having the status of a problem “hard to improve”, and there is enough evidence to think that methods for TAG parsing that are asymptotically faster than $O(|G|n^5)$ are unlikely to be of any practical interest, i.e., will involve very complex computations.

As a second result, we show an even closer relationship between the TAG parsing problem and boolean matrix multiplication, by partially reversing the direction of the above reduction. We investigate a subclass of TAG’s, called Linear TAG (LTAG), and discuss a construction that maps any instance of the LTAG parsing problem to an instance of the boolean matrix multiplication problem. In this way, we obtain that time upper bounds for boolean matrix multiplication can be transferred to time upper bounds for the parsing problem of the subclass LTAG. Such a result has its theoretical significance, since already known tabular methods for TAG parsing still exhibit $\Theta(|G|n^5)$ time performance for some worst case input grammars in the subclass LTAG. The best time upper bound known to date for boolean matrix multiplication is approximately $O(n^{2.376})$, as reported in [1]. Using our result, we can then transfer this upper bound to LTAG parsing, obtaining the improved time upper bound $O(|G|n^{1.758})$. Our first result is not contradicted, since the method obtained in this way should definitely be considered prohibitive, given the huge constants involved in its running time: at present, no straightforward method is known for boolean matrix multiplication that considerably improves the cubic time upper bound and that can be used in practical cases.

All the above results are obtained using deterministic polynomial time reductions between search problems. This kind of reduction is commonly used to provide hardness results for complexity classes not known to be included in $P$ ($P$ is the class of all languages decidable in deterministic polynomial time). The problems we investigate here are already known to be in $P$. However, no solution is known for these problems that works in deterministic linear time. Hence, the choice of almost linear, deterministic time reductions allows us to transfer upper bounds with the computational consequences we have discussed.

The remaining part of this paper is organized as follows. Next section presents the definition of tree adjoining grammar and restates the parsing problem for this class as a search problem. Sections 3 and 4 separately present and discuss the two main results. Finally, section 5 concludes with some remarks.

2 Preliminaries

The class of TAG’s and the associated notion of derivation are briefly recalled here; elementary familiarity is assumed on the part of the reader with the related definitions (see for instance [13] or [2]).

A tree adjoining grammar is denoted as a tuple $G = (N, \Sigma, S, I, A)$, where $N$ and $\Sigma$ are finite, disjoint sets of nonterminal and terminal symbols respectively, $S \in N$ is a distinguished symbol, $I$ and $A$ are finite sets of initial and auxiliary trees respectively. Trees in $I \cup A$ are called elementary trees and meet the following specification. Internal nodes in an elementary tree are labeled by nonterminal symbols. An initial tree has root labeled by $S$ and leaf nodes labeled by symbols in $\Sigma \cup \{\varepsilon\}$; an auxiliary tree has leaf nodes labeled by symbols in $\Sigma \cup \{\varepsilon\}$ with the addition of one node, called foot node, having the same nonterminal label as the root node. The size of a TAG $G$, written $|G|$, is defined as the sum of all nodes in the elementary trees of $G$.

A Linear TAG (LTAG) is a TAG in which every elementary tree is associated with a path, called spine, connecting its root node to one of its leaves. In the case of auxiliary trees, the spine ends in the foot node. By definition, nodes along the spine are the only sites at which adjunction (see below) can take place.

In TAG, the notion of derivation is based on a composition operation called adjunction, defined in the following way. Let $\gamma$ be an auxiliary tree having root (and foot node) labeled by $A \in V_N$. Let also $\gamma'$ be any tree containing a node $\eta$ labeled by $A$, and let $\tau$ be the subtree of $\gamma'$ rooted in $\eta$. The adjunction of $\gamma$ into $\gamma'$ at node $\eta$ results in a tree specified as follows (see Figure 1):

(i) the subtree $\tau$ is excised from $\gamma'$;
(ii) the auxiliary tree $\gamma$ replaces $\tau$ in $\gamma'$, with the root of $\gamma$ replacing the excised node $\eta$;
(iii) the subtree $\tau$ is attached to the resulting tree, with the foot node of $\gamma$ replacing $\eta$ in $\tau$. 
In TAG a derivation is the process of recursive composition of elementary trees using the adjunction operation; the resulting trees are called derived trees. Since adjunctions at different nodes can be performed in any order, we can directly adjoint derived trees into derived trees without altering the outcome of the derivation process.

Although TAG is a class of tree rewriting systems, a derivation relation can be defined on strings in the following way. Let $\gamma$ be an elementary tree and let $\gamma'$ be a tree obtained from $\gamma$ by means of zero or more adjunction operations. If the yield of $\gamma'$ is a string $x \in X^*$, that is $\gamma \in I$, we say that $\gamma$ derives $x$ in $G$. If the yield of $\gamma'$ is a string $xAy \in \Sigma^*\Sigma^*$, that is $\gamma \in A$, we say that $\gamma$ derives the pair $(x,y)$ in $G$. In particular, the set of all strings in $\Sigma^*$ that can be derived in $G$ is denoted by $L(G)$. In this perspective then, an elementary or a derived tree is seen as a structural description of a string (a pair of strings) derived by the grammar; such a description is called a parse tree. The space of all parse trees associated by the grammar with a string in the generated language is called a parse forest.

We introduce now the notion of subderivation. If an auxiliary tree $\gamma$ participates in the derivation of a sentence $w$, it will do so by deriving a pair $(x,y)$, $x, y \in \Sigma^*$, where strings $x$ and $y$ will appear as non-overlapping prefixes within $w$. In this case we say that the derivation of $(x,y)$ from $\gamma$ is a subderivation of a sentential derivation of $w$. As a consequence of the definition of parse forest, we have that all subderivations of the sentential derivations of $w$ can be read off from the parse forest of $w$.

We can now introduce a relation that will be used to restate the parsing problem for TAG as a search problem. Let $G = (N, \Sigma, S, I, A)$ be a TAG and $w = a_1a_2\cdots a_n$, $n > 0$, be an input string over $\Sigma$. For $1 \leq p \leq q \leq n$, we write $p,w_q$ to denote the substring $a_pq+1\cdots a_q$. A parse relation $R_p \subseteq A \times \{1,\ldots,n\}$ associated with the pair $(G, w)$ is specified as follows. For every auxiliary tree $\gamma$ in $G$ and for natural numbers $p, q, r$ and $s$, $1 \leq p \leq q \leq r \leq s \leq n$, $R_p(\gamma, p, q, r, s)$ holds if and only if:

(i) the pair $(p,w_q,w_s)$ can be derived by $\gamma$ in $G$, and
(ii) the derivation in (i) is a subderivation of a sentential derivation of $w$ in $G$.

The goal of a parsing algorithm for TAG is one of constructing a "suitable" representation for the parse forest of a given string, with respect to a given grammar. However, there is no common agreement in the literature on the requirements that such a representation should meet. Indeed, a trade-off is found between computational time and space in choosing among different representations of a parse forest. Note that, from an extreme perspective, the input itself can be considered as a highly compressed representation of the parse forest, one that needs a time-expensive process for parse tree retrieval. More explicit representations offer the advantage of time-efficient retrieval of parse trees, at the cost of an increase in storage resources. In practice, most commonly used algorithms solve the parsing problem for TAG's by computing a superset of a parse relation (defined as above) and by representing it in such a way that its instances can be tested in constant time; such a condition is satisfied by the methods reported in [13], [11], [7], [10], [6], [5] and [14]. From such a representation, time-efficient computations can be later used to retrieve parse structures of the input string.

On the basis of the previous observation, we assume in the following that the solution of the parsing problem involves (at least) the computation of a representation for $R_p$ such that its instances can be tested in constant time; we base our results on such an assumption. More precisely, an input instance of the tree adjoining

---

1. I owe this observation to Bernard Lang (p.c.).
grammar parsing problem is defined to be a pair \((G, w)\), \(G\) a TAG and \(w\) a string over \(\Sigma\), and the unique solution of such an instance is provided by an explicit representation of relation (set) \(R_p\) associated with \((G, w)\).

In the next sections we will deal with the TAG parsing problem and with the boolean matrix multiplication problem. The latter is specified as follows: given any pair \((A, B)\) of square boolean matrices, we solve the problem by providing the unique square matrix \(C\) such that \(C = A \times B\) (row-to-column product). We will denote by TGP and BMM the TAG parsing and the boolean matrix multiplication problem respectively.

3 The TAG Parsing Problem

This section states a result that has been obtained using a reduction from BMM to TGP. The general idea underlying the construction is informally presented to the reader and the computational consequences of the result are then investigated; see [8] for a formal treatment.

Both the BMM and the TGP problems are viewed here as search problems whose solutions are obtained by exploring a search space of elementary combinations. In the case of the BMM problem, the elementary combinations are the combinations of elements of the input matrices; in the TGP problem, the elementary combinations are taken to be single applications of the adjunction operation. In order to achieve our result, we then establish a (size preserving) correspondence between the two search spaces above. The general idea underlying the construction is to encode natural numbers using three positive integers; the encoding is then used to chop off matrix indices into smaller numbers that will be processed independently, as described in what follows.

Let \((A, B)\) be a generic instance of BMM, \(A\) and \(B\) of order (number of rows) \(m\). We will construct an instance \((G, w)\) of TGP such that the parse relation \(R_p\) that solves \((G, w)\) will straightforwardly provide the product matrix \(C\) that solves instance \((A, B)\). In this way, given any algorithm for the solution of the TGP problem, we can effectively construct an algorithm for the solution of the BMM problem. We represent the input matrices by means of grammar \(G\), which fixes \(|G|\) to a quantity \(O(m^3)\). This forces the choice of \(n\) to a quantity \(O(m^4)\).

First of all, observe that non-null elements \(a_{ik}\) and \(b_{kj}\) in \(A\) and \(B\) respectively force element \(c_{ij}\) to value 1 in the product matrix \(C\) if and only if \(k = k'\). The check of such a condition can be transferred to the computation of an adjunction operation in the target parsing problem using the following method. Fix a positive integer \(b\) to a (rounded) quantity \(m^4\). We encode each index \(i\) of the input matrices by means of positive integers \(i_1, i_2\) and \(i_3\), such that \(i_1\) is \(O(b^4)\) and \(i_2, i_3\) are \(O(b)\). Condition \(k = k'\) above is therefore reduced to the three tests \(k_{ih} = k'_{ih}, 1 \leq h \leq 3,\) which can be performed independently. The test \(k_1 = k'_1\) is precompiled into some auxiliary tree of \(G\); the tests \(k_2 = k'_2\) and \(k_3 = k'_3\) are performed by the parser using the input string, as explained below.

We construct a string \(w\) of distinguishable symbols by concatenating six "slices" \(w^{(b)}, 1 \leq h \leq 6,\) each slice of length \(O(b)\). We then encode the input matrices \(A\) and \(B\) within the target grammar \(G\); this is done by transforming each non-null element in the input matrices into an auxiliary tree of \(G\) in the following way. Non-null element \(a_{ik}\) is mapped into an auxiliary tree \(\gamma_i\) having root (and foot node) labeled by a symbol including integers \(i_1\) and \(k_1\). Moreover, \(\gamma_i\) will eventually derive a string pair \((x_i, y_i)\) with the following property. String \(x_1\) is the smallest substring of \(w\) including the symbol in the \(i_1\)-th position within slice \(w^{(1)}\) and the symbol in the \(i_2\)-th position within slice \(w^{(2)}\). Furthermore, string \(y_1\) is the smallest substring of \(w\) including the symbol in the \((k_1 + 1)\)-th position within slice \(w^{(6)}\) and the symbol in the \(i_3\)-th position within slice \(w^{(6)}\). This is schematically shown in Figure 2.

At the same time we map non-null element \(b_{kj}\) into an auxiliary tree \(\gamma_j\) having root labeled by a symbol including integers \(k'_1\) and \(j_1\). Crucial to our construction, \(\gamma_j\) will derive a pair of strings \((x_2, y_2)\) with the following property. String \(x_2\) is the smallest substring of \(w\) including the symbol in the \((k'_1 + 1)\)-th position within slice \(w^{(3)}\) and the symbol in the \(j_2\)-th position within slice \(w^{(3)}\). Furthermore, string \(y_2\) is the smallest substring of \(w\) including the symbol in the \(j_2\)-th position within slice \(w^{(4)}\) and the symbol in the \(k'_2\)-th position within slice \(w^{(4)}\). Let us call \(\gamma_1\) and \(\gamma_2\) the derived trees obtained from \(\gamma_1\) and \(\gamma_2\) as above. Observe that \(k_2 = k'_2\) and \(k_3 = k'_3\) if and only if the yields of \(\gamma_1\) and \(\gamma_2\) are exactly nested within \(w\); see again Figure 2.

To complete the construction of \(G\), we add an auxiliary tree \(\gamma_3\) with the following property. Tree \(\gamma_3\) can contribute to a sentential derivation of \(w\) in \(G\) if and only if \(\gamma_1\) and \(\gamma_2\) can be adjoined to it. This is in turn possible just in case integer \(k_1\)
in the root of $\gamma_1$ and integer $k'_1$ in the root of $\gamma_2$ are equal, as specified by the adjunction sites in $\gamma_3$, and the yields of $\gamma'_1$ and $\gamma'_2$ are exactly nested within $w$. It follows that, by deciding whether $\gamma_3$ contributes to a sentential derivation of $w$, the parser is able to perform the required test $k = k'$. Finally, index $k$ in its coded form is discarded in the derivation process above, while indices $i$ and $j$ are preserved in such a way that we can easily recover non-null element $c_{ij}$ by reading off the parse relation that solves the target instance $(G, w)$.

The particular encoding that we use for matrix indices requires the computation of a constant number of integer divisions. Such a computation can be carried out in an amount of time $O(\log^2(m))$ using standard algorithms. Since there are $O(m^2)$ elementary trees in $G$, the whole construction of the target instance of TGP takes an amount of time $O(m^2 \log^2(m))$. This is also a time upper bound on the computation of the product matrix $C$ out of the parse relation $R_p$ that solves the target instance of TGP, assuming that each instance of $R_p$ can be tested in constant time.

As a consequence of the above remark, we have that any time upper bound for the TGP problem can be transferred to an upper bound for the BMM problem, down to the time needed for the computation of the proposed reduction. The following statement gives an example (see [8] for a formal proof).

**Theorem 1** Let $A_p$ be an algorithm for the solution of the TGP problem having running time $O(|G|^p |w|^q)$, $p, q > 1$. Then any instance of the BMM problem can be solved in time $O(\max\{m^{2p+1}, m^2 \log^2(m)\})$, where $m$ is the order of the input matrices.

The above theorem implies that any method for the solution of the TGP parsing problem having running time $O(|G| |w|^6)$ will give us a method for boolean matrix multiplication having running time $O(m^{2.83})$. Likewise, any $O(|G| |w|^4)$ time method for the former problem will result in a $O(m^{2.6})$ time method for BMM. We remark here that, even if the involved constants hidden in our construction are large, the resulting methods will still be competitive with known methods for boolean matrix multiplication that improve the well known cubic time upper bound. At present, no algorithm is known for boolean matrix multiplication that considerably improves the cubic upper bound and that can be used in practical cases. Also, there is enough evidence that, if such a method exists, its discovery should be a very difficult enterprise. We conclude then that the TAG parsing problem should also be considered as having the status of a problem "difficult" to improve, and we have enough evidence to think that methods that are asymptotically faster than $O(|G| |w|^6)$ are unlikely to be of any interest for practical application purposes.

The technical details of the reduction overviewed in this section reveal that
Theorem 1 still holds when we restrict ourselves to the subclass LTAG. In the next section, we will show an even closer relationship between such a restriction and boolean matrix multiplication.

4 The LTAG Parsing Problem

The parsing problem for the subclass LTAG, abbreviated LTGP, is here related with the BMM problem, reverting thus the direction of the reduction discussed in the previous section. Again, we only discuss the general idea underlying the construction; formal details can be found in [9].

Our reduction is composed of two steps. First, we construct a square matrix $A$ out of the input instance of LTGP, whose elements are defined over a set that is associated with a non-commutative product. It holds that the transitive closure of $A$ with respect to the above product, written $A^*$, directly gives the desired representation of relation $R_p$ that solves the input instance of LTGP. In the second step, we use a reduction from the computation of $A^*$ to boolean matrix multiplication, that as been studied in [12]. We can view the problem of computing the transitive closure above as a problem of searching all paths in a directed graph, where each path is associated with a labeling symbol and path concatenation is defined by means of the non-commutative product. We start with the specification of how sub-trees of trees derived in $G$ can be represented as paths in a graph, since this is at the base of our construction.

When a tree is derived in $G$, whose yield matches (part of) $w$, each of its sub-trees can be represented by (an identifier for) its root node and the positions in $w$ that delimit its yield, called pointers. If the root node does not dominate a foot node, then the yield of the sub-tree is a single string and two pointers can be used to encode it. We call this a single pair representation. In the case the root node dominates a foot node, the yield of the sub-tree is a pair of substrings of the input, and four pointers are required. We call such a representation double pair. Using this representation, sub-trees can be combined according to two basic operations: tree concatenation constructs a new sub-tree out of its child trees; tree nesting adjoins a tree derived from an auxiliary tree, at the root of a sub-tree of a derived tree (see Figure 3).

To obtain our reduction, then, we map single and double pair representations into graph paths, in such a way that instances of the tree concatenation and tree nesting operations are translated into instances of path concatenation in the target problem. Our solution is based on the idea of combining pointers into pairs that will define the vertices of our target graph.

Let $(G, w)$ be an instance of LTGP, with $n > 0$ the length of $w$. The vertices of our graph are defined as

$$V = \{ (i, j) \mid 0 \leq i \leq j \leq n \}. \quad (1)$$

To be used later, we associate with $V$ a (strict) total order relation $\prec_V$, specified as

$$<_V = \{ (v_1, v_2) \mid v_1 = (i_1, j_1), v_2 = (i_2, j_2), i_1 < i_2 \lor (i_1 = i_2 \land j_2 > j_1 > 0) \}. \quad (2)$$

We use identifiers of nodes in the elementary trees of $G$ in order to define the set of labels for edges of the target graph:

$$L = \{ [\eta] \mid \eta \text{ is a node of } \tau \in (A \cup I) \}. \quad (3)$$

Assume now that $\eta$ is a node of some elementary tree $\alpha$. Let $\tau$ be a subtree of a tree derived from $\alpha$ in $G$, such that $\tau$ has root $\eta$ and is compatible with $w$, in the sense that the yield of $\tau$ is a substring or a pair of non-overlapping substrings of $w$. We must deal with three basic cases.

Case 1 Assume that $\alpha \in A$, $\eta$ belongs to the spine of $\alpha$, and let $p \leq q \leq r \leq s$ be the four pointers in a double pair representation of $\tau$. We represent $\tau$ by means of items (path label and incident vertices)

$$[\eta], (p, s), (q, r), \quad (4)$$

where we have encoded into single vertices the outermost and the innermost pointers of $\tau$. This
has the desired consequence of mapping instances of the tree nesting operation into instances of path concatenation. In fact, let $\gamma$ be a tree with

![Diagram](a)

Figure 4: Nesting of $\tau$ within $\gamma$ (a) translates into path concatenation in the chosen representation (b).

Consider first the case in which $\psi$ is the father node of $\eta$, and call $\eta'$ the right sibling of $\eta$. It follows that $\eta'$ belongs to the spine of $\alpha$. If there exists a subtree $\tau'$ with root $\eta'$ (see Figure 5(a)) such that $\tau$ and $\tau'$ can be concatenated compatibly with $w$, then $\tau'$ must be associated with pointers $q \leq q' \leq r' \leq s'$ in a double pair representation, and thus represented by items $[\eta'], (q, s')$ and $(q', r')$ according to (4). Note that the resulting path is adjacent with the path representing $\tau$ (see Figure 5(b)).

A second possibility is that $\psi$ is not the father node of $\eta$. From the definition of LTAG, no adjunction can ever be performed at any node below $\psi$ in the path from $\psi$ to $\eta$. Call again $\eta'$ the sibling node of $\eta$ and note that $\eta'$ does not dominate the foot node of $\alpha$. Assume that $\eta'$ is

![Diagram](a)

Figure 5: Concatenation of subtrees $\tau$ and $\tau'$ (a) translates into path concatenation in our representation (b).

Case 2 Assume now that $\eta$ is located at the left of the spine of $\alpha$ and let $p < q$ be the two pointers in a single pair representation of $\tau$. We have to map this representation in a way that is compatible with case 1. We start by observing that there must be a node $\psi$ in the spine of $\alpha$ that dominates $\eta$ and is the lowest node with such a property. If tree $\tau$ will ever be included in a successful analysis of string $w$, node $\psi$ (or some foot node that has replaced it after an adjunction) will be associated with four pointers (to positions within $w$) in such an analysis. Let $s'$ be the rightmost of these pointers. In our representation we associate $p$ and $q$ with $s'$. Note that we do not know which integers are possible values for $s'$; nevertheless we can make every possible guess for these values. Following this idea, we represent $\tau$ by means of items (path label and incident vertices)

$$[\eta], (p, s'), (q, s'),$$  

for every value of $s'$ such that $q \leq s' \leq |w|$. This choice maps instances of the tree concatenation operation in the source problem to path concatenation in the target problem, as explained in what follows.

![Diagram](a)

Figure 6: A second case of concatenation of subtrees $\tau$ and $\tau'$ (a) resulting in a path concatenation configuration in the target graph (b).
and \((q', s''')\), for every \(s''\) such that \(q' \leq s'' \leq |w|\).

Note that representations of \(\tau\) and \(\tau'\) with the same values for pointers \(s'\) and \(s''\) denote paths that are adjacent, and can therefore be combined in the target problem (see Figure 6(b)). Finally, a symmetrical configuration is found when \(\eta'\) is at the left of \(\eta\). In this case \(\tau'\) must be associated with pointers \(p' < p\) in a single pair representation. Then we represent \(\tau'\) by means of items \([\eta'\), \((p', q')\), \((p', p)\]\), for every value of \(s''\) such that \(p \leq s'' \leq |w|\).

**Case 3** If node \(\eta\) is located at the right of the spine of \(\alpha\), we have a simmetrical situation w.r.t. case 2. Let then \(p < q\) be the two pointers in a single pair representation of \(\tau\). We represent \(\tau\) by means of items

\[
[\eta], (p', q'), (p', p),
\]

for every value of \(p'\) such that \(0 \leq p' \leq p\). □

In the cases above we have been able to map all possible instances of the tree nesting and tree concatenation operations in \(G\) into adjacency configurations between paths over vertex set \(V\). At this point it is not difficult to specify a non-commutative product associated with label set \(\mathcal{L}\) as to restrict possible concatenations of paths to the only subtree combinations that are legal in \(G\). Furthermore, using the above prescription we can construct a set of (labeled) directed arcs \(E\), on the basis of the terminal symbols in \(w\) and the leaf nodes in the elementary trees of \(G\). In this way we obtain a (edge labeled) graph \(G_r = (V, E)\) whose transitive closure, computed with respect to the non-commutative product, provides the desired representation for relation \(R_p\), solving thus the instance of LTGP at hand. (Here we do not undergo the details of the construction of \(G_r\).

We point out that the total order \(\prec_{\mathcal{V}}\) in \(3\) induces a topological order on \(G_r\), that is \(G_r\) can be represented as an array \(A\) whose non-empty entries are all located in its upper triangular portion. In [12] a construction is presented that reduces the computation of the transitive closure of an upper-triangular square matrix to boolean matrix multiplication, when the closure is defined with respect to a non-commutative product. We can then use this construction to complete our reduction. We only state the final result here, and address the reader to [9] for a formal proof.

**Theorem 2** Let \(A_M\) be an algorithm for the solution of the BMM problem with running time \(O(m^3)\), \(m\) the order of the input matrices and \(p \geq 2\). Then any instance of the LTGP problem can be solved in time \(O(|G||w|^{2p})\). □

If the standard cubic time method for boolean matrix multiplication is used in the above theorem, we get a non-standard parsing method for the class LTAG working with the time upper bound \(O(|G||w|^3)\); this is also the time upper bound achieved by standard methods for TAG parsing (see section 3). More interestingly, Theorem 2 can be combined with the asymptotically fastest method known to date for boolean matrix multiplication, which runs in approximately \(O(m^{2.373})\) time as reported in [1]. This results in the time upper bound \(O(|G||w|^{1.725})\) for LTGP, which is an improvement of the previously known upper bound. We remark again that the performance of the method in [1], although asymptotically superior, is not comparable with that of the standard cubic time algorithm in practical cases, due to the huge constants involved in its running time. In a sense, this confirms us the result in section 3, indicating that improvement of the \(O(|G||w|^3)\) time upper bound, even if restricted to subclass LTGP, may not be of any interest in current applications or for other practical purposes.

5 Remarks

In previous sections we have related the complexity of tree adjoining grammar parsing to the complexity of boolean matrix multiplication. As already discussed, the notion of parse forest is an informal one, and there is no common agreement on which specifications such a structure should meet. The obtained results are based on the assumption that a parsing algorithm for TAG should be able to provide a representation for a parse forest such that instances of the parse relation \(R_p\) can be retrieved in constant time (see section 2). Whatever the specifications of the output parse forest structure will be, it seems quite reasonable to require that an explicit representation of relation \(R_p\) can be extracted from the output in linear time with respect to the size of the output itself, thus without affecting the overall running time of the method. This requirement is satisfied by all TAG parsers that have been presented to date in the literature.

As a second point, the studied constructions provide interesting insight in the structure of the TAG parsing problem. We see for instance that the major source of complexity in the problem derives from cases of properly nested adjunctions: such cases are responsible of a (bounded) amount of nondeterminism in the computation, forcing a
method to detect how a string divides into subparts according to the adjunction of some derived
tree into another.

In section 4 we have used the computation of the closure of an upper-triangular square matrix
as an intermediate search problem in our reduction. This problem is directly related with a gen-
eralization of the context-free grammar parsing problem called lattice parsing (see [5]) where we
have to parse, using a context-free grammar, all sentential forms that can be recognized by a finite
state automaton. Hence, we have found that the basic operation in TAG derivation, that is tree ad-
joining, is related with self-embedding by means of a change of representation. Unfortunately, if
we drop the linearity restriction assumed in section 4, we find that the above closure problem
involves a full (non-upper-triangular) square matrix. This problem is not known to date to be re-
ducible to boolean matrix multiplication, and hence the $O(|G|^2|w|^{3})$ time upper bound for the
general TAG parsing problem still remains the best one we know at present.

REFERENCES


A Single Formalism for a Wide Range of Parsers for DCGs

François Barthélemy
Departamento de Informática,
Universidade Nova de Lisboa,
2825 Monte de Caparica
Portugal
frb@fct.unl.pt

ABSTRACT

In this paper, we present the APOC-II system for prototyping parser generators for Definite Clause Grammars (DCGs). Many algorithms may implement a DCG parser (SLD-resolution, chart parsing either top-down or bottom up, etc.) and different techniques may be employed to handle their non-determinism (backtracking, tabulation, etc.). APOC-II is based on a formal framework that clearly distinguishes between tree construction and handling non-determinism, thus enabling the modular specification of these two components of a parser generator. A DCG compiler reflects a parsing algorithm and translates the DCG into a possible non-deterministic extended push-down automaton (XA). A parser combines the generated automaton with an interpreter. The interpreter may use several approaches to handle the automaton's non-determinism, including backtracking and tabulation. The modular design of APOC-II allows code sharing and a free exchange of components to produce a parser generator.

Keywords: Definite Clause Grammars, Parsing, Push-Down Automata

1 INTRODUCTION

When developing a new system, it is extremely convenient to work with a prototype that may be rapidly modified and tested. APOC-II\(^1\) facilitates the prototyping of parser generators for DCGs.

The system is based on a general formal framework in which most known parsing techniques may be formulated. Expressing different strategies in a single formalism can shed light on similarities between theoretical foundations (see [10]). A common scheme gives us a legitimate platform for comparing the different parsing algorithms.

Many parsers may implement a given grammar, and we would like to pick and choose between elements that compose a parser generator. For example, an LL parsing strategy is not practical for left recursive grammars whereas bottom up is, and LR parsing strategies work poorly with non-LR grammars. APOC-II allows us to prototype different strategies quickly and to exchange with ease components that make up DCG compilers or that handle non-determinism.

Parsing with respect to a DCG involves two interrelated tasks: building a tree by combining elementary trees (the clauses), just as for a context-free grammar, and performing a unification for each node of the tree. A new equation appears at the moment when a clause instance is grafted to a tree. The graft may be invalid if unification fails.

The context-free part of DCG parsing, namely parsing according to the context-free backbone of the DCG, can be expressed using a Push-Down Automaton (PDA). Well-known techniques let one automatically compile any context-free grammar into a non-deterministic PDA.

We extend these compilation techniques and the PDA formalism itself to account for the equation solving process. We call the new formalism X-automaton (XA, for eXtended Push-Down Automaton). A grammar is a purely declarative description of a language. An X-automaton, on the other hand, incorporates both the grammar and an operational semantics to parse text written in the language and build syntax trees.

Like PDAs, XAs may be non-deterministic, i.e., different evolutions are possible from a given state. And as with PDAs, they may be executed in several ways: direct execution (restricted to de-
terministic automata), backtracking, tabulation, graph-structured stacks. In APOC-II, the parsing strategy and the treatment of non-determinism are orthogonal issues. The interpreter ignores the strategy embodied by the automaton—it executes the automaton whatever the strategy. Alternative implementations of the compiler (LR, LL, Left-corner, Earley) that produces the automaton from a DCG may be combined seamlessly with different interpreters (backtracking, tabulation). The modular architecture allows us to extend the system with new algorithms as desired and to reuse code conveniently (e.g., the same unification algorithm is used by all parsers).

The system currently offers more than twenty different parsing algorithms. Moreover, many new variants can be easily implemented by adding a small number of modules.

The next section is devoted to the presentation of the parsing of languages defined by DCGs. Section 3 contains the definition of X-automata. Section 4 describe two algorithms that compile any DCG into an X-automaton. The first one produces a top-down automaton, the second a bottom-up. Other compilation methods are discussed. The fifth section touches on the execution of X-automata. The sixth section is devoted to the architecture of APOC-II. The last section is a short comparison with related work. Some APOC-II results are given in appendix A.

2 DCGs, SYNTAX TREES, AND PARSING

Definite clause grammars are now well known, so the following reminders are quite brief. More details can be found in the original paper [12] and in [1].

DCGs are an extension of context-free grammars but differ in that:

- non-terminals are first-order terms instead of simple symbols. The use of logic variables allows one to express equality of otherwise unknown parts of terms.

- some literals called procedure calls can appear in the right-hand side of rules. They express additional conditions on non-terminals' arguments. The corresponding procedures are defined by a logic program.

The grammar obtained by removing all the arguments of every non-terminal and all the procedures calls occurring in a DCG is called the underlying context-free grammar of the DCG, or its context-free skeleton.

In the following, to simplify the presentation, we limit ourselves to procedures defined by pure definite clause programs (i.e., without negation, built-ins, or control predicates). These can be conveniently seen as grammar rules that derive into the empty string.

The usual way of defining the semantics of DCGs is a translation into definite clause programs. A DCG rule looks very much like a clause. There is only one implicit piece of information, namely the sequentiality of the symbols. This information is easily made explicit with two additional arguments added to the symbols. These two arguments, which are either integers [12] or difference-lists in some other translations, define the part of the string that the symbol derives. Variables allow us to express the sequentiality of substrings, without any other information about these substrings.

Another way of defining a semantics for DCGs is more grammatical. It consists in defining the notion of syntax tree.

Definition 1 DCG Syntax tree
A syntax tree of a DCG G is a labeled tree such that:

- every leaf node is labeled either by a terminal or by a couple (c, σ) where c is a unit clause of G (rule whose right-end side is empty) and σ is a substitution.

- every internal node is labeled by a couple (c, σ) where c is a rule of G and σ is a substitution.

- every leaf node labeled by a terminal t is the i-th child of a node labeled by a couple (c, σ) where c is a clause c₀ :: c₁, ..., cₙ, n ≥ i, and t = cᵢσ.

- every other node except the root is such that:
  - it is labeled by a couple (c, σ), where c is the clause c₀ :: c₁, ..., cₙ.
  - it is the i-th child of its parent.
  - the parent is labeled by (k, θ), where k is the clause k₀ :: k₁, ..., kₘ, and i ≤ m.
  - kᵢθ = c₀σ

The string associated with a syntax tree is the sequence of terminals formed by following the leaves of the tree in a preorder traversal. The
language defined by a DCG $G$ is the set of terminal strings associated with at least one syntax tree from $G$.

To parse is to try to find a syntax tree (or all the syntax trees) associated with a given string.

A pre-order can easily be defined by extending the usual instance relation in a straightforward way. This allows to work modulo subsumption. Only equivalence classes are considered, thus reducing the search space for parsing. In particular, systematically renaming clauses is sound because the trees obtained cover all the equivalence classes.

Following definition 1, an equation $k_1\theta = e_0\alpha$ appears for each node of the tree (except the root), identifying the head of the clause labeling this node with the relevant atom of the body of the clause labeling its parent. The minimal requirement for this equation to be satisfied is obtained by unifying the two atoms, after renaming. A global substitution for the whole tree is the result of the resolution of all the equations of the tree. We don't enter the technical details on these points. They can be found in any good book on logic programming.

One point we wish to emphasize is that unifications can be delayed. An interesting property of unification is that it depends only on the equations being considered, not on the order in which they are taken into account. The unifier may be computed incrementally—as the tree grows and the equations appear. But it can also be computed afterwards.

As shown by Deransart and Maluszynski in [5], a tree can be built considering only the context-free constraints of the grammar, that is, matching only predicate symbols instead of unifying the complete terms. This gives a structure for a candidate syntax tree. Of course, to transform it in an actual parse tree, all the unifications must be performed, and the result must be applied on the tree. But this can be done afterwards, once the whole tree has been built.

Between the two extremes—unification as soon as possible (at the moment a new clause instance is grafted to a partial syntax-tree) or delaying until a whole tree is constructed—many intermediate steps are available. Notice that no unification is performed before the corresponding part of the tree has been built.

Parsing with DCGs may be divided in two distinct parts:

- Constructing the tree skeleton, that is the tree labeled by clauses and renamings. This part is similar to a context-free parsing with the underlying context-free grammar of the DCG.

- Solving the equations of this skeleton.

The two parts of parsing may be mixed in many different ways. A sound and complete parsing method blends a sound and complete context-free parsing method with the consideration of all the tree equations.

3 X-AUTOMATA

We now define a new kind of automata, specially designed to perform DCG parsing following the principles exposed in the previous section. They are an extension of push-down automata classically used for context-free parsing. The aim of the extension is to handle the contextual constraints of the DCGs (the equations).

Informally, an X-automaton is a stack automaton where each level of the stack stores four pieces of information:

- an atom, a kind of node label
- a substitution, that is, a part of the answer substitution (possibly with additional irrelevant variables)
- an equation set that contains delayed unifications
- a pointer to the input string that separates what has been read from what remains

Following Bernard Lang, we consider a machine whose only storage is a stack. There is no state indicator and no string pointer. This implies that all relevant information must be encoded in the stack. For instance, the state of an automaton, if necessary, has to be stacked in such a way that the current state appears in the top element.

A configuration of the automaton reduces to a stack. There are three stack operations:

- PUSH pushes a tuple on top of the stack
- HOR changes the top of the stack for another tuple
- POP replaces the top two tuples by only one, possibly different from these two
Any combination of the usual pushing and popping operations with or without a state change can be translated respectively into PUSH and POP. For instance, the classical pop without changing the state, consists only in removing the top element of the stack. This can be done by one new POP: it replaces the two top elements by the former second one. Similarly, if the state changes, this second element has to be modified. The new POP is able to do this as well. State changes with no change in the stack are achieved by the HOR.

**Definition 2 X-automaton.**

An X-automaton is a 5-tuple \((\Sigma, V, \text{init}, F, T)\) where

- \(\Sigma\) is a graded alphabet
- \(V\) is a denumerable set of variables
- \(\text{init}\) is a nullary initial predicate
- \(F\) is a set of final predicates
- \(T\) is a finite set of transition functions, respecting the requirements described below in definition 3

**Definition 3 Requirements for the transition functions (abbreviated as "transitions"):**

- they are partial functions from stacks to stacks
- each transition applied on any stack in its domain performs one of the three basic operations PUSH, HOR, and POP. We call PUSH-transition (respectively, HOR-transition, POP-transition) a transition that performs a PUSH (respectively, a HOR, a POP).
- whether or not a stack belongs to the domain of a transition only depends on the top of the stack, namely the top tuple for PUSH and HOR-transitions and the two top tuples for a POP-transition.

In other words, a transition is any function that locally applies one of the three basic operations (PUSH, HOR, or POP) to the top of the stack.

In the rest of this paper, we will indicate a stack using the Prolog list notation, the top being on the left-hand side. We will use \(a, b, c, d\) as a notation for the 4-tuples in the stack.

**Definition 4 Configuration**

Let \(A = (\Sigma, V, \text{init}, F, T)\) be an X-automaton.

- the stack \([< \text{init}, \text{id}, 0, 0 >]\) where \(\text{init}\) is the initial predicate, \(\text{id}\) is the identity substitution, and \(0\) the empty set, is a configuration of \(A\) called initial configuration.
- The stack obtained by the application of a transition \(t\) of \(T\) to a configuration \(c\) (belonging to the domain of \(t\)) is a configuration of \(A\).

**Definition 5 Computation**

Let \(A = (\Sigma, V, \text{init}, F, T)\) be an X-automaton. A computation of \(A\) is any sequence \(c_0, c_1, \ldots, c_n\), \(n \geq 0\), of configurations of \(A\) such that:

- \(c_0\) is the initial configuration:
  \([< \text{init}, \text{id}, 0, 0 >]\).
- \(\forall i, 0 < i \leq n, \exists t \in T\) such that \(t(c_{i-1}) = c_i\).

**Definition 6 Success**

Let \(A = (\Sigma, V, \text{init}, F, T)\) be an X-automaton. A computation of \(A\) is said to be a success if it contains only two tuples: the top tuple must have its first component (the atom) belong to \(F\) and its last component (the integer) equal to the length of the input string; the second tuple of the stack must be the initial one.

We need a language to write the transitions. We will use a small functional language inspired by ML (and especially the CAML dialect [18]), in which functions can be described by pattern matching. We change the syntax slightly in order to use Prolog notation for lists, and the previously defined syntax for tuples. To distinguish between logical variables (for instance those appearing in the grammar) and the transition language variables, each of the latter will be prefixed by a \(\$\). The most useful operations such as unification, substitution, renaming, and the scanning of the string, are predefined in the language. Substitution application and substitution composition are denoted by an infix dot.

Example of a transition:

\[
\text{function } [< \$, \$, \$, \$, \$ > ] \rightarrow \text{let } \sigma = \text{unify}(\$, (p(3, X))), \text{ in}
[< q(X, Y), \sigma, \sigma, 0, \$, >, \$, \$, \$, \$ > ]
\]

We suppose here that an exception is raised by the function unify when the unification fails. This exception can be captured when the failure of the unification doesn't imply a failure of the transition application. The domain of the above function is the set of stacks whose top tuple has an
atom unifiable with $p(34, X)$ as first component.
Notice that this transition adds a new tuple on the stack, so it is a PUSH transition.

Transitions can be named by a let: let $f =$
function $[.. . . . .]$ $\to \ldots$

4 Compiling Grammars into X-Automata

Now that we have defined X-automata and their behavior, we show that they can be used to describe many kinds of parsers. This section is devoted to algorithms that compile arbitrary DCGs into X-automata. These algorithms are the heart of a parser generator. We first give two basic algorithms: the first produces a top-down automaton and the second a bottom-up automaton. Then we discuss other compilation techniques using our formalism.

4.1 Top-down Parsing

We introduce new predicates ci-k-i (for clause instance), where $k$ is anything uniquely identifying a clause, and $i$ is an integer. This integer denotes a position in the body of the clause $k$, that separates the already recognized part of this body from the part that is still to be processed. The integer is the number of atoms in the body that have been recognized so far. A ci-k-i predicate contains exactly the same information as the dotted rules used by Earley and LR parsing algorithms. For instance, let 1 : noun-phrase :- article, noun. be a rule. ci-1-1 is equivalent to the dotted rule noun-noun :- article, noun and means that the parser tries to use the 1st clause, has already recognized an article, and still has to parse a noun.

Algorithm 1 Top-down compiler

For each clause $k : k_0 :: k_1, \ldots, k_n$, such that $k_0$ is unifiable with the goal $g$ by a unifier $\sigma_k$, create the following transition:

1. function $[\langle \text{init}, \text{id}, \emptyset, 0 \rangle \to$

   let $\theta \text{eta} = \text{renaming} (k)$ in

   let $\sigma = \text{unify}(k_0, \theta, g)$ in

   $[\langle \text{ci-k-0}, \sigma, 0, 0 \rangle,$

   $\langle \text{init}, \text{id}, \emptyset, 0 \rangle]$}

Furthermore, for each such transition, ci-k-n is a final predicate of the automaton.

For each clause $c : c_0 :: c_1, \ldots, c_m$, for all $i$, $0 \leq i < n$, such that $c_{i+1}$ is a non-terminal, and for each clause $d : d_0 :: d_1, \ldots, d_m$, such that $c_{i+1}$ and $d_0$ are unifiable, create the two transitions:

2. function $[\langle \text{ci-c-i}, \theta, 0, 0 \rangle \to$

   let $\theta \text{eta} = \text{renaming}(d)$ in

   let $\sigma = \text{unify}(c_{i+1}, \theta, d_0, \theta)$ in

   $[\langle \text{ci-c-0}, \theta, \sigma, 0, 0 \rangle,$

   $\langle \text{ci-c-i}, \theta, \sigma, 0, 0 \rangle]$

3. function $[\langle \text{ci-c-i}, \theta, 0, 0 \rangle \to$

   $[\langle \text{ci-c-i+1}, \theta, \theta, 0, 0 \rangle \to$

   $\langle \text{End} \rangle]$

For each clause $c : c_0 :: c_1, \ldots, c_n$, for all $i$, $0 \leq i < n$, such that $c_{i+1}$ is a terminal, create the transition:

4. function $[\langle \text{ci-c-i}, \theta, 0, 0 \rangle \to$

   let $\sigma = \text{unify}(c_{i+1}, \theta, \text{scan}(\theta X + 1))$

   $[\langle \text{ci-c-i+1}, \theta, \sigma, 0, 0 \rangle \to$

   $\langle \text{End} \rangle]$

The work of top-down X-automata should seem quite obvious to people aware of SLD-resolution with the standard strategy. Transitions of type 1 and 2 correspond to the non-deterministic choice of a clause to reduce a (sub-)goal. The chosen clause must be renamed with fresh variables.

In usual SLD-resolution, the result of unifications is immediately applied to the remaining subgoals. Here, we do it lazily, when popping. The current computed substitution is stored only in the top tuple. Type 3 transitions propagate this substitution one level down the stack.

The type 4 transition reads a terminal. There is a unification in these transitions because variables may stand for a terminal in some rules. The stored substitution is applied in order to rename them according to the renaming of the rule.

At each level of the stack, the substitution contains the composition of the renaming of the corresponding node and the answer substitution. This substitution is used mainly for the renaming (cf., SS2 in transition 3).

4.2 Bottom-up Parsing

Among the many ways of expressing a bottom-up method, we choose those that seem the most understandable, not necessarily the most efficient.
Algorithm 2 Bottom-up compiler

1. Create the transition:
   \[ \langle s1, s2, \emptyset, sX > \mid \$End \rangle \rightarrow \langle \text{scan}(sX + 1), id, \emptyset, sX + 1 >, \langle s1, s2, \emptyset, sX > \mid \$End \rangle \]

2. For each clause \( k : k_0 :: k_1, \ldots, k_n \), create the following transition:
   \[ \langle s1, s2, \emptyset, sX > \mid \$End \rangle \rightarrow \langle \text{let theta = renaming}(k) \rangle \]
   \[ \langle \text{ci-k-0}, \theta, \emptyset, sX >, \langle s1, s2, \emptyset, sX > \mid \$End \rangle \]

3. For each clause \( k : k_0 :: k_1, \ldots, k_n \), create the following transition:
   \[ \langle \text{ci-k-n}, \emptyset, \emptyset, sX > \mid \$End \rangle \rightarrow \langle k_0, \emptyset, id, \emptyset, sX > \mid \$End \rangle \]

4. For each clause \( k : k_0 :: k_1, \ldots, k_n \) and each position \( i \leq i < n \), create the transition:
   \[ \langle sA, sS_1, \emptyset, sX >, \langle sA, sS_2, \emptyset, sY > \mid \$End \rangle \rightarrow \langle \text{let sigma = unify}(k_{i+1}, sS_1, sA) \rangle \]
   \[ \langle \text{ci-k-i+1}, sS_2, sS_1, \emptyset, sX >, \langle s1, s2, \emptyset, sX > \mid \$End \rangle \]

5. Add a final transition to filter the success. Let \( g \) be the goal.
   \[ \langle sA, \emptyset, \emptyset, sX >, < i, id, \emptyset, 0 > \rightarrow \langle \text{let sigma = unify}(sA, g) \rangle \]
   \[ \langle \text{success, sS, sigma, \emptyset, sX >, < i, id, \emptyset, 0 >} \]

Intuitively, the transitions have the following meanings:

1. You can always push a terminal on the stack and advance one step in the string.
2. You can always try to reduce any rule.
3. If you have recognized the entire body of a rule and synthesized the substitution that instantiates that rule, then you may reduce and obtain the head of the clause, on which this substitution applies.
4. If you have already recognized the \( i^{th} \) first symbols of the body of a rule, and the \( i + 1^{th} \), is on the top of the stack, then you have recognized the \( i + 1 \) first atom of the rule. Unification must be performed, with the proper renaming, and the answer substitution must be passed along.

This kind of automaton is highly non-deterministic.

4.3 OTHER PARSEING TECHNIQUES

Many other parsing methods can be implemented by algorithms analogous to our two examples. The author has already written LR(0), left-corner bottom-up, Earley, and Earley with restriction compilers. Notice that using the LR(0) technique in no way restricts the class of grammars considered. If the grammar is not LR(0), then the LR(0) parser will be non-deterministic, but it is correct and complete. Similarly, Prolog does not restrict us to LL(0) grammars, since it supports non-determinism with backtracking.

Many of the parsing algorithms developed for DCGs can be encoded into X-automata plus an execution scheme, and many of the context-free parsing methods can be extended in order to cope with arguments of non-terminals. There are however some limitations to our formalism.

The first is that the formalism is meant for left-to-right parsing. The state of the automaton with respect to the input string is given by a unique pointer that separates the left part already recognized from the right part left to scan. The same device can be used for right-to-left parsing, but not for a non-unidirectional parsing. The reason for our choice is that we are mainly interested in left-to-right parsing and that we think that the sequentiality of the string is a strong property that is worth exploiting. Nonetheless, a small change in the formalism would allow non-sequential parsing.

The second main limitation of our framework is that it implies a completely static compilation of the grammar. It expresses more easily those algorithms that in some way have a context-free behavior, i.e., the search strategy depends only on the label of the search tree node considered, not on the shape of the context of this node. We have only considered these kinds of algorithms because they are the most common. XAs don't work well with more dynamic techniques (such as, for instance, those involved in the AKL model).

These limitations are not very strong. We address the way two devices of great interest for DCG parsing, look-ahead and unification delay, can be encoded within X-automata.

Look-ahead involves examining tokens of the string before considering the corresponding nodes of the parse tree in order to eliminate parsing choices certain to fail. A static analysis of each
non-terminal gives the set of terminals that can appear in the string in some correct parse tree when considering a node labeled by this non-terminal, with respect to the parsing technique being used.

We may simply compute look-ahead sets using context-free techniques on the context-free backbone of the grammar. More precise results may be obtained by extending these techniques to take into account part of the non-terminals’ arguments.

To add a one symbol look-ahead to a transition

\[
\text{function}[< \text{s}, \ldots, \text{x} >, \ldots] \rightarrow \text{...text-of-the-function...}
\]

just write:

\[
\text{function}[< \text{s}, \ldots, \text{x} >, \ldots] \rightarrow \\
\text{if member(scan(\text{x} + 1), lookahead-set(\text{s}))} \\
\text{then ...text-of-the-function...}
\]

Where lookahead-set retrieves the statically computed set of terminals that can follow \text{s}. Look-ahead of arbitrary length may be added not only to LR automata, but also to top-down and bottom-up automata.

Another way to vary parsing strategies is to delay unifications. The third field of the stack tuples is used to store the equations until they are reduced by unification.

Delaying unification is a way of simulating attribute-grammar-like evaluation of DCGs, where a context-free technique parses the whole string, just collecting the equations that are eventually reduced in a special order.

This is of course the most simple means to use the delay mechanism. It can be used more locally, as is done with the classical freeze predicate.

Another important point we can only address briefly in this paper concerns the integration of extra-logic devices to perform side-effects. The methods we present here apply on pure DCGs. Unfortunately, pure DCGs often lack practical expressive power. To overcome this limitation, adding some imperative features seems necessary.

However, to preserve the versatility of X-automata, these additional devices must remain independent of the parsing strategy. If built-ins or imperative constructs rely on an evaluation order, the grammars can’t be parsed using another order. For instance, the behavior of some Prolog built-ins depends on the groundness of their arguments. This groundness is highly dependent on the evaluation strategy. The programmer uses his knowledge about it when writing a program/grammar.

In our opinion, constraints are an elegant way to express extra-logic features such as built-in arithmetic or input/output, and allow us to preserve a purely declarative semantics for grammars. Some work on the subject has been done in [4]. Many other research projects deal with the use of constraint logic programming in the grammatical field, usually with very different motivations.

5 Executing X-Automata

Expressing different kinds of parsers in the same formalism is interesting from a declarative point of view. It allows comparison between alternative methods expressed in the same words. In addition to this advantage, X-automata can be executed.

The usual ways of executing push-down automata may be extended to X-automata: executing deterministic automata directly, using a backtracking mechanism, using graph-structured stack, using tabulation.

Limiting ourselves to the execution of deterministic automata is probably too strong a restriction. Some work on the characterization of classes of grammars that lead to deterministic parsers using a given strategy (mainly LR) was done some years ago in the field of affix grammars ([7], [15]).

Automata can be extended by stacking choice points and backtracking on failures. This poses no particular difficulty. It is the usual way to handle the non-determinism of definite clause programs. This may be used with the Prolog goal-directed strategy, but it is not restricted to it. It has also been successfully used (among others) by Nilsson [11] with an LR method.

The third alternative is to use tabular techniques. They consist in decomposing computations in elementary steps and storing partial results in tables (or other data structures) in which they can be found when needed, without re-computation. Some instances of tabular techniques: for logic programming, ODLT by Sato and Tamaki [17] and Earley Deduction by Pereira and Warren [13]; for parsing, CKY [2] and Earley [6].

In [8], Lang gives a tabular procedure to execute any Push-Down automaton. The idea is to
compute the complete set of stack tops that may appear in any calculation of the PDA. The reachable stacks, and especially successful ones, can be extracted from this set of tops.

This approach was later extended to more complex stack automata. [3] proposes a sufficient condition on a stack machine for such a tabular technique to be sound and complete. X-automata do not generally satisfy this condition, but the author shows in [4] that the X-automata generated by the algorithms given above, and some others (LR, Earley) do.

6 APOC-II: A MODULAR PROTOTYPE

APOC-II's implementation of XAs has been designed for the experimentation and comparison of DCG parsing algorithms. We wanted an open system allowing quick prototyping of as many parsing algorithms as we could find. We made as few assumptions as possible to allow for unforeseen developments.

APOC-II's modularity allows different parser generator implementations to share code. The key to this modularity was to isolate logical independencies between parts of the algorithms.

At a global level, we distinguish between the compiler, which converts a grammar into anXA, the interpreter, which parses a string following anXA, and the pretty printer, which prints anXA. These modules share theXA internal representation, the unification algorithm, et al.

Within each module, we decomposed things in homogeneous tasks. For instance, the compiler is divided into:

- a parser for the grammars
- a static analyzer that computes look-ahead sets
- a module for terms representation, unification, and related operations
- a module describing the tree-traversal
- a module for transition representation, storage, and manipulation
- a control loop

Compilers may share any of the code that implements these tasks. For example, the Earley and LR compilers differ in tree traversal.

Two features of our logical modules make their management a little tricky. The first is that we may have alternative implementations of a module and we want to put off as long as possible the moment when a choice between implementations must be made. Our design must not rely on which implementation is finally chosen.

The second characteristic of our system is that the alternative implementations of a module are not always strictly equivalent. For instance, chart parsing and backtracking executions both involve a lexer that reads the input string. Whereas chart parsing only needs to read sequentially one token after another, backtracking execution requires backtracking in the string. There are two slightly different views of the same logical module. Of course, we could impose that a module implement the cartesian product of all the functions needed by all the possible users. But this would be restrictive and counter flexibility and versatility.

Fortunately, we don't have to impose such a constraint. We found a language that provides exactly the kind of modularity we needed: Alcool 90 [14]. Alcool 90 is a typed functional member of the ML family. It extends usual ML instructions with an original module system. The external view of an Alcool module is an interface containing the set of functions with their signatures (types) the module implements, and possibly a set of functions used by, but not defined in, the module. Such a module can be instantiated by specifying the real implementation of these functions.

At compile time, when a function is used, the compiler searches in the library for all possible implementations and asks the user to choose between them. A type-checking procedure ensures that only safe combinations of modules (i.e., each module gets exactly what it needs) are performed.

Consider again the example of the lexer. Suppose we have two implementations of a lexer, one that just reads sequentially a stream and provides the function next-token and another that puts the tokens in an array before advancing in the stream and provides two functions next-token and backtracking. A chart parser uses only next-token, so it can use one or the other of the implementations, whereas a backtracking parser needs backtracking and therefore can only use the second one.

At the moment, the different parser choices offered by APOC-II for each module are:

- Tree construction: LL, LR, pure bottom-up, left-corner bottom-up, Earley without restriction, Earley with the maximal restriction.
• **Look-ahead:** 0 or 1 symbol (LR excepted, only LR(0))

• **Non-determinism:** backtracking, tabular breadth-first execution.

Combining these alternatives gives 22 different parsers\(^2\). Two of them (backtracking execution of pure bottom-up with or without look-ahead) always loop.

New modules can be added in each of the three above categories. Variations are also possible for tabular execution by changing the order of the computations. This is not a minor issue, since this order affects efficiency and even termination.

7 RELATED WORK

We don’t know of any other attempts to design a multi-strategy parsing machine for DCGs. Most papers describe a specific algorithm either including control or letting Prolog take care of it. They usually can be expressed by an X-automata compiler plus a control technique.

The closest work is in the area of logic programming: Lang ([9]) designed a machine called a **logical push-down automaton (LPDA)** that allows the encoding of the evaluation of a definite clause program using one of several resolutions methods. Top-down, bottom-up, and Earley are considered. This is a generalization of the techniques developed for context-free parsing [8].

This is the principal source of inspiration behind the present work. X-automata are an extension of LPDA, that overcome some of their limitations. Whereas an XA transition may be any arbitrary function respecting the stack mechanism, LPDA’s have a fixed and rigid form, consisting in

- a systematic renaming
- a unification with a fixed pattern
- the application of the unifier to a fixed pattern to be stacked

LPDA are therefore more static than XA. They can’t generally encode devices such as look-ahead, delayed unification, and restriction [18]. They can describe only one way of renaming.

The drawback of this extended expressive power is that XA’s lose the property of soundness of the tabular procedure given in [9]. It must be proved separately for each subclass corresponding to a given strategy.

8 CONCLUSION

We presented a system for prototyping parser generators for definite clause grammars. This modular prototyping system is based on a single formalism to represent a wide range of parsers. This formalism separates the parsing strategy and the way non-determinism is handled. The most common parsing algorithms can be expressed within this framework. Many algorithms may reuse code thanks to the modular architecture. Furthermore, the prototype extends easily to new algorithms by adding new modules. Appendix A shows the results of some experiments carried out with small grammars.

9 ACKNOWLEDGEMENTS

The author is grateful to Ian B. Jacobs and Gabriel Pereira Lopes for their help.

REFERENCES


A AN EXPERIMENTAL STUDY OF COMPUTATION SHARING

This appendix is devoted to an example of experiment with APOC-II.

We ran several parsers for the same little grammar:

**Grammar 1**:

\[
s → [s].
\]

\[
s → s, s.
\]

This is a very ambiguous grammar for the language \( a^+ \). Every binary structure is allowed. For each execution, we counted the number of transition applications, and for tabular ones, we also counted the number of entries of the table at the end of the parsing. The results are given in the arrays of figure 1. We stopped backtracking executions when the first solution was found, whereas tabular execution computed every solution.

The grammar can be augmented by adding an argument to the non-terminal \( s \) that carries a representation of a parse-tree:

**Grammar 2**:

\[
s(a) → [s].
\]

\[
s(s(X,Y)) → s(X), s(Y).
\]

Using this grammar, we obtain the results of figure 2.

With backtracking, the results are the same for grammar 1 and grammar 2. Since only one tree is considered at once, building this tree costs nothing. On the other hand, tabular execution computes a compact representation of all the parse-trees, namely a shared forest. There are two kinds of sharing: subtree sharing and context sharing (see figure 3). The argument added to \( s \) in grammar 2 represents a lone tree, not a shared forest (this would require graphs instead of terms, and a different unification). This prevents context sharing. Therefore, tabular computations are more costly using grammar 2 than using grammar 1. Notice that this costly argument is not really needed, since the table itself is a representation of the shared forest.

Things are not always as simple as in our example, since the argument can be used to disallow some of the parses.

Arrays given in figures 1 and 2 show the behavior of the alternative parsers with respect to context sharing.
Figure 1: parsing with grammar 1

Earley+ stands for Earley without restriction, Earley- for Earley with the maximal restriction, bup for pure bottom-up and lc for left-corner. The number between brackets is the length of the look-ahead. For tabular execution, the first integer is the number of transition applications and the second one is the final number of entries in the table. When an execution was too long (after an hour or so), we gave up and wrote down a $t$ in the corresponding cell of the array.

The results for a backtracking execution are the same as for grammar 1.

Figure 2: parsing with grammar 2

Figure 3: structure sharing
Multi-agent Systems in Natural Language Processing

Erzsébet Csuhaí-Varjú *
and
Reda Abo Alez

Computer and Automation Institute
Hungarian Academy of Science
H-1111 Budapest, Kende u. 13-17.
Hungary
e-mail: h264csu@ella.hu

ABSTRACT

We propose a multi-agent framework of natural language, based on cooperating/distributed grammar systems. The model is illustrated by some representations.

1 INTRODUCTION

Natural languages are such systems which are made up of a number of modules that interact in a nonsimple way. In these systems, the whole is more than the sum of the parts, in an important pragmatic sense that, given the properties of the parts and the laws of their interactions, it is not trivial matter to infer the properties of the whole. The classical research activity in natural languages was utilized for studying different types of centralized modules. Instead of understanding the reality of the natural language by finitely describing the behavior as a whole, we will try to show that the task of understanding and/or generation can be performed on the base of both the individual behavior of the system's components and some knowledge on the communication strategy of these components.

Representing modules by multi-agent systems appears to be a natural approach. Multi-agent systems form a primary research area in contemporary Distributed Artificial Intelligence ([3]). Following [4], we list some main characteristics of multi-agent systems which we use in building our model in the sequel. The area of multi-agent systems is concerned with coordinating intelligent behavior among collections of autonomous agents; how they can coordinate their knowledge, goals, skills and plans, jointly to take action or to solve problems. The agents may have and may be working toward a single global goal, or they may have and may be working toward separate individual goals that interact. The agents share knowledge about problems and solutions and about the processes of coordination. The coordination of the task can be quite complicated and there are situations where there is no possibility for global control, globally consistant knowledge, globally shared goals, even global representation of a system. Such special systems are open systems of Hewitt ([12]) which

- are composed of independently developed parts in continuous evolution,
- concurrent and asynchronous, have decentralized control based on debate and negotiations, they exhibit many local inconsistencies;
- they consist of agents with bounded knowledge and bounded influence;
- they have no fixed global boundaries visible to the agents constituting the system.

Another important field is the theory of autonomous agents with emergent functionality ([16],[5],[6]). Here the functionality of an agent is viewed as an emergent property of its intensive interaction with its dynamic environment. Agents are constructed from modules obtained by task-level decomposition. The communication among modules is reduced to the minimum and there is no global representation or a global planner of the activity of an agent. The activity of an agent emerges by the interaction of the behaviours of the modules. Each autonomous agent

---

*Research was supported by Hungarian Research Foundation OTKA no. 4295
accomplishes its own individual task or a global task. Such situations are studied in detail in Decentralized Artificial Intelligence ([10]).

2 The model

The question 'How did the human use, represents, store, and retrieve the information', argue the role of simultaneous human abilities to understand and generate one or more languages.

Minsky ([18]) has developed a hypothesis based on viewing human mind as an organized society of intercommunicating agents, the so called society theory of mind. According to Minsky's hypothesis, mental activities appear from interactions between agents organized in some local hierarchies. An application of Minsky's model for modelling complex symbol systems was presented in Kelemen [13].

Using grammars, we can speak about which notions can be expressed and defined in an intuitive sense. With the idea of distribution and cooperation, represented in the society theory of mind, how can the paradigm of grammar systems ([8,9]) affect and be used for understanding and generation of NL as human mental activity which is based on natural intelligence and tools?

By using the above concepts, we investigate the proper ways for formulating and imitating these activities to aid the process of automation including machine translation, question-answer systems, and man-machine interfaces.

Our model originates from the notion of the cooperative/distributed grammar system ([7]), of the parallel communicating grammar system ([20]) and of the stratified grammar system ([14]) and unifies the properties of these generative paradigms of cooperative systems.

In the following we will demonstrate our strategy for handling the language model. The model has different functional modules which have their own cooperating strategy. Each module has its competence and its own grammar system. The grammar system consists of a number of components (grammars) which have their own role in the derivation of the sentential form.

The grammars with different strategies, forming a multi-agent symbol system, participate in rewriting of a given shared sentential form. One possibility is where the components work simultaneously, each having its own sentential form for rewriting. According to the type of activity: Generation or Analysis, one of the grammars is distinguished as the master grammar. It can ask from the other grammars the current strings as they are. In this way the communicated strings become a part of the string generated by the master grammar. The language generated in this manner by the master grammar is then considered the language generated by the whole system. In this case the module is organized as a parallel communicating grammar system. Sequential organization of the grammar system is allowed, too. In this case the component grammars are associated with local and/or global (start/stop) context conditions which control the grammars' entering into the derivation process and stopping with it. Start and stop conditions can be defined on the base of local (global) competence in contributing to the derivation, too. This organization corresponds to the organization of the cooperating/distributed grammar system of ([7],[8],[9]).

By the above models, the strategies in the system are the strategies of derivation followed by each of its component, and the strategies of cooperation of components.

It is obvious that, understanding and generation needs cooperation at least of the phonological, morphological, lexical, syntactic, referential, semantic modules and textual module. Modules are clustered in sets, which, following Minsky, we call strata, and we assume that a linear ordering of these strata is given. The work of the system starts with the first stratum in this ordering and continues from stratum to stratum until a terminal string is obtained. Every component of a stratum must effectively participate in the rewriting. These strata are dynamically defined depending on the current state of a global task and on the specialization of the components. Thus, modules are organized as a stratified grammar system of [14].

In the following two sections we represent a language generation approach demonstrating cooperation and distributivity among different modules represented by different grammars. Moreover, there is a visible parallelism in the generation process.

In the grammar system for each module, a question arises whether only one component is active in every moment, or all components are
active (maybe not, this is imposed by an external conditions), or more components are active and their set is dynamically determined by the sentential form to be written.

3 Applications

We consider generation as an immediate verbalization of the parts of computable conceptual structure in situations that are predictable in advance. Generation has two main phases: the planning of the content of what to say (restrictive planning algorithm) and the decision of how to say it. Generation is a complex realization of progressive, cooperative and distributed activities; where the linguistic module (how to say) is processing partial conceptual structures which are previously computed from the conceptual module (what to say), the latter can run simultaneously and add more conceptual structures. As a result, during generation the contents (information) can be verbalized in due time.

We mention two main approaches to ensure that the contents can be verbalized in due time. The first is the systemic grammar which is organized as a set of loosely related choice systems. Each choice adds some constraints to the final sentence form (i.e. a grammar as a filter), but the final structure of the sentence is determined only by the combination of all the different choices made ([11]).

In this framework the grammar translates decisions of the planner into constraints of the syntactic form of the sentence. The planner itself needs no knowledge of the syntactic consequences of its decisions. Other modules make decisions based on other choice systems and add other additional constraints.

The other approach uses a more distributed control. To generate an utterance from a representation, either it uses a realization function that maps conceptual or semantic network fragments to phrases or it uses an explicit grammar that directs the generation process.

Here the grammar is not used to control the generation process, rather other techniques are used to produce partial semantic and syntactic structures, which are then processed by the grammar to fill in the remaining details. In accordance with this idea, Danlos implemented a pair of grammars to produce an utterance ([16]).

It may happen that specified information is not sufficient to identify a single sentence uniquely, or unspecified but required information is missing. To avoid these situations, leading to the successive specification of the complete sentence structure, the linguistic module has to provide a feedback for the selection of what is missing or what to say next by the conceptual module. To make this possible, we need some uniform representation for the content, structure, and lexical items in the sentence.

A framework that is useful for this purpose is functional unification grammar {1}, which is well-suited for investigating the interaction between the grammatical issues and the content planning issues. This framework slightly differs from the unification grammar ([2]). The syntactic ordering restrictions are given by patterns in structures.

To illustrate how the generation process looks like see the following example:

(S (FIRST-NP (np))
 MAIN-V (verb ROOT x TENSE y NUMBER Z)
 NUMBER = NUM(FIRST-NP) = NUM(MAIN-VERB)
 OBJ (np)
 MOD (pp)
 PATTERN (FIRST-NP, MAIN-VERB, OBJ, MOD))
 (NP DET (art)
 HEAD (noun)
 PATTERN (ART NOUN ... MOD))
 (NP PRO (pro)
 PATTERN (PRO))
 (NP NAME (name)
 PATTERN (name))
 (PP PREP (prep)
 POBJ (np)
 PATTERN (PREP POBJ))

The complete sentence is constructed by unifying structures together where the pattern slot determines the syntactic order of constituents by syntactic grammar {2}.

The special power of this approach for generation arises from the fact that semantic and discourse information can be encoded in the same notation as well. This requires introducing a notational variant of the logical form based on case
The output conceptual module is just as:

(S TENSE past
AGENT NUMBER Singular
AGENT PERSON 3
VOICE active
ACTION TYPE buy1
AGENT John1
THEME pen10
FROM POSSESSION Mary1)

The case roles become slots, and new slots are introduced for the tense, and specifications as needed.

With this representation, the mapping to the syntactic form can be done by unification operations that equate roles like AGENT with FIRST-NOUN. For example, if the discourse module chooses between active and passive sentence forms, then it has simply to pick one of the following sentence forms that map the case roles to the desired syntactic positions.

(SF FIRST-NP(
NP REF = `AGENT)
OBJ (NP REF = `THEME)
VOICE active)
(SF FIRST-NP(
NP REF = `THEME)
AUX be
MODS (PP PREP by
POBJ (NP REF = `AGENT)
VOICE passive)

By unifying one of the active and passive structures into the conceptual structure, the new structure will be:

(S TENSE past
AGENT NUMBER Singular
AGENT PERSON 3
VOICE active
ACTION TYPE buy1
AGENT John1
THEME pen10
FROM POSSESSION Mary1
FIRST-NP (NP REF = `AGENT)
OBJ (NP REF = `THEME))

The overall structure of the sentence has been determined, but none of the subconstituents of the structure have been planned.

The structure is taken by a grammar based lexical analyzer which plans how to realize each slot-value pair in the structure. For instance, the pair ACTION TYPE/buy1 can be realized using the main verb with root buy.

Moreover, the tense analysis can be done by a separate module that then simply unifies its results back into the structure form being constructed.

MAIN-VERB (verb ROOT buy
TENSE past
NUM Singular)

If it is decided that the AGENT/John1 will be realized as noun phrase as (NP REF John1) in the FIRST-NP slot, a grammar based reference module might choose to realize this as a pronoun unifying in the structure (NP PRO he). Moreover, the THEME/pen1 pair must be realized as an NP in the OBJ slot. The reference module might reason that this object has not been introduced before and choose to introduce it with an indefinite noun phrase e.g. (NP DET a HEAD pen).

Finally, once all these choices are made, the remaining details of the sentence structure are determined by unifications with the proper syntactic rules in the linguistic module to produce a final representation of the sentence such as:

(S FIRST-NP (NP PRO he
AGENT NUMBER Singular
AGENT PERSON 3
REFERENCE John1
PATTERN (PRO))
MAIN-VERB (verb ROOT buy
TENSE past
NUM 3)
OBJ (NP DET a
HEAD pen)
REF pen10)
PATTERN (DET HEAD))
MODS (PP PREP from
POBJ (NP name Mary
REF Mary1)
PATTERN (PREP POBJ))
An algorithm was implemented in SL-TRANS2, a spoken language translation system to generate sentences ([19]). With this method, it is possible to produce all structures in parallel.

Generation requires particular needs upon syntactic description and processing in a parallel fashion such as:

- The grammar should support vertical rather than horizontal orientation. The rules should emphasize the growing of syntactic structures by individual subtrees (ideally, only by one branch).

- When the whole syntactic structure of an utterance is built, it should be possible to decide for every partial structure whether it is locally complete.

- One should not assume that the chronological order in which syntactic parts are attached during generation, corresponds to linear order of the resulting utterance.

4 THE LINGUISTIC MODULE IN PARALLEL PROCESSING

In the linguistic module, the syntactic level is divided into two sublevels:

- the first level at which the syntactic structure is constructed in parallel way and

- the second level at which there exists one process that performs the inflection and linearization for every coming part from the first level.

4.1 REPRESENTATION

The representation and the derivation devices are separated. In order to allow vertical orientation, the knowledge concerning immediate dominance and linear precedence (ID/LP), is divided into ID and LP structures. The ID structures are divided into those that describe basic syntactic parts and others that describe either the circumstances or the relation between complex structures. Properties, classification and quality of ID structures makes it possible to interpret such structures theoretically based on dependency grammar (implicit grammar) [7].
The basic syntactic parts are defined as phrases that consist of a set of related constituents. A basic part must access the information that is necessary to build up a minimal syntactically accepted utterance. Therefore, each basic part has a constituent that determines the characteristic properties of the whole phrase, which is denoted as the head of the phrase. All other constituents are denoted as participants or as complements that are immediately dependent on the head. Basic parts build up syntactic constraints for their complements. Although this seems to emphasize horizontal orientation, these constraints are not redundant and therefore do not violate the view of vertical orientation. So, the central knowledge source of both previous sublevels is modified to a unification-based head driven grammar [1].

Circumstances or adjuncts are not part of the basic parts, because they are not necessary for building minimal accepted structures. They are added to an object by augmenting the object's context with their new labeled slots to yield a complex part. The corresponding ID structures of circumstances are applied only when the basic part already exist.

Let us assume an example. If sentence is denoted as

\[(S \Rightarrow NP1,V2,NP3),\]

then the basic part for constructing a sentence is:

\[
S \text{ (agreement,} \\
LP(0,sentence), \\
LP(1,subject), \\
LP(2,verb), \\
LP(3,direct object)). \\
NP1\text{(agreement,} \\
\text{case:nominal,} \\
LP(1,subject)). \\
V2 \text{ (agreement,} \\
LP(2,verb), \\
\text{functor(part1,part2))}. \\
NP3\text{(usually local agreement,} \\
\text{case:accusative,} \\
LP(3,direct object)).
\]

4.2 Generation and Parallel Processing

The generator is augmented by operations which are suitable for parallel generation.

In order to construct syntactic structures in parallel way, every dependent object has to solve the following two main subtasks:

- Building up connections to other objects.
  While new communication links are build up, participating objects exchange their syntactic features.
- Mapping itself into the next level.

4.2.1 The first subtask

Objects at the first level are created during the evaluation of local translation rules [7] which describe the relation between semantic and syntactic knowledge in a distributed way. They are local because every rule describes the mapping of some semantic parts to some syntactic parts of the ID structures.

The central viewpoint for the description of the basic parts is that a dependency relation is defined for the constituents of a phrase. For example, the predicate `functor(part1, part2)` as one element of the feature set of the basic part for constructing a sentence where the functor governs the two participants corresponds to the syntactic relation `verb2(np1,np3)`, where the verb is the essence and the two noun phrases are the context of this translation process.

In principal this local mapping is still a caseframe based on functional grammar [9], where a semantic head (e.g. functor of predicate) and its deep cases (e.g. agent, benefactive) are related to the corresponding syntactic head (e.g. verb) and its syntactic frame (e.g. subject, or direct object constituents).

The ID structures are processed at the first level in parallel way. At this level, basic parts are considered as independent and active components. In order to realize this, every basic part is associated with its own active translation rules by which the underlying structure of a part reflects the underlying dependency relation of that part.
If the dependencies between a head functor and its modifiers are strong as in the case of complements, then this information has to be formulated in increasing, parallel manner.

During the mapping, lexical material which is determined through the choice of words, directs and restricts the choice of possible syntactic structures. The choice of words is performed in two steps. First, while conceptual parts are translated into semantic parts, the content words—e.g., verbs and nouns—are selected. Second, the selection of function words, e.g., prepositions which depend on the meaning of the verb, is performed during the mapping between the dependency level and semantic level.

The topology of the network of active translating processes represents the corresponding dependency-based tree structure.

Every new created concept tries to integrate itself into the topology of the already existing objects. By this way, syntactic parts should be expanded in the following three ways:

- The existing structure is expanded downward (a new part B becomes the daughter node of a previous part A), where the new object is taken up into the context of another object. Syntactically, this means that the new object represents an immediately dependent constituent. In the following example the new object becomes the daughter node of a previous basic. Suppose the following:

  \[
  \text{functor(part1, part2) } = \\
  = \text{verb2 (np1, np3) = likes(John, X)} \\
  X = \text{NP3 = proper noun = Mary,} \\
  \text{and likes(John, X), then} \\
  \text{likes(John, Mary) = verb2 (np1, np3)}
  \]

- The existing structure is expanded upward (a new part B becomes the root node of a previous part A), where the new object binds an already existing object. In the following example, the new verbal object binds two complements. Suppose the following:

  \[
  \text{X = proper noun = John,} \\
  Y = \text{proper noun = Mary,} \\
  \text{functor(part1, part2) } = \\
  = \text{verb2 (np1, np3) = verb2(X, Y),} \\
  \text{verb2 = likes, then} \\
  \text{verb2 (np1, np3) = likes(John, Mary)}
  \]

- The existing structure is inserted (B becomes the daughter node of A, which already has a daughter and this daughter may become the daughter of B), where the new object must be replaced with an already existing object. Consider the following sentence:

  \text{John plays tennis \ldots uh \ldots table tennis.}

Suppose the following:

  \[
  \text{functor(part1, part2) } = \\
  = \text{verb2 (np1, np3) } = \\
  = \text{plays(John, tennis),} \\
  \text{X = proper noun = John,} \\
  \text{Y = noun = tennis,} \\
  \text{Y = noun = tennis,} \\
  \text{but Y = noun phrase = table tennis, then} \\
  \text{verb2 (np1, np3) =} \\
  = \text{plays(John, table tennis).}
  \]

This is in contrast to the approach in which a generator driven by a syntactic system—the ATN—that interpreted semantic networks which were designed by Winograd (1983), Goldman (1975) and Shapiro (1982) ([2]). All syntactic parts are defined as node-arc-node structures, and parts have to branch only by one element at time in a sequential mode. The problem with using this approach to drive the entire generation process is that it forces a certain ordering of decisions about the surface structure of the sentence that must be followed by the system.

4.2.2 The second subtask

In order to facilitate spontaneous dialogue, the feature set of the head functor of the associated part is checked to see if it contains sufficient information for inflecting and linearizing it. For every part, this information has to be specified under the head’s local feature. Actual values come either from a governing constituent, e.g., case for dependent nouns of a verb, from dependent elements by means of structure sharing, e.g., person and number for a verb from the subject constituent, or from the lexicon e.g., gender for nouns. When all subfeatures of the local feature have values (not empty), then the basic or complex part is said to be locally complete.

The reason for the part of a conceptual object not to be locally complete is that these objects
that share values with the local feature set are either not yet created or not yet locally complete themselves (for instance a verb is locally incomplete when a phrase that has to fill the subject role is not yet created). In the first case, where objects that share values with the local feature set are not yet created, the missing objects are requested (using the object’s context) from that object of the semantic level above that has created the requested object (a feedback). In the second case, where objects that share values with local feature set are not yet locally complete themselves, the objects are invited to determine the missing values.

The order of the parts in an utterance is syntactically constrained by the linear precedence LP part of the central knowledge source. It is assumed that the order of activation of the conceptual parts which is determined by the pragmatic knowledge, should be maintained if it is syntactically accepted. Otherwise the parts are reordered by means of relevant LP-structure. Therefore, the order of the input parts affects the variations of the linear order in the resulting utterance (e.g. by deciding whether to realize an active or passive form).

Lastly, there remains the process which performs the inflection and linearization for every coming part to second level that are locally complete from the first level.

5 Final Remarks

In the previous sections progressing approaches from central module to distributed parallel processing modules for language generation were described, gaining features such as:

- The possibility of processing free word order languages by separating the knowledge concerning immediate dominance and linear precedence (ID/LP) controlling mechanisms where the ID is further divided into those that describe basic syntactic parts and others that describe either the circumstances or the relation between complex structures.

- The decisions about the actual structure of the sentence are not made last after the more functional properties of the structure is considered.

- Cooperative task performing which is constructed from modules obtained by task-level decomposition where an agent cooperates with other agents to perform its own individual or a global task. The functionality of an agent is viewed as an emergent property of its intensive interaction with its dynamic environment.

The crude brackets referred to the minimal number of cooperative grammar-based modules that are needed for the generation process. Moreover, some other cooperative modules have to be taken into account, for example a set of inference rules to digest the conceptual representation and produce a structure that is suited for the task of verbalization (content), or as pragmatic and stylistic issues for word choice, syntactic choice, sentence inclusion, or the criteria that distinguish between variations. (The sentences either have different effects or differ in voice.)

Many vital questionable areas have raised such as:

- Natural languages are devices which are used to produce through time an evolving collection of symbol structures. That is, in writing steps of derivation, a component may either use a production for rewriting the current string or may wait. The question is what happens when synchronization is not supposed and presented in NL system.

- Restrictive planning of the content of what to say and using functional unification grammar allow a much more flexible ordering of decisions than other formalisms and investigates the interaction between the grammatical issues and the content-planning issues. Pereira emphasized in [21] that the computational tractability of complex constraints in unification based formalism remains an open problem. The question is now that whether there is a need for designing a grammar with global and local constraints which will consider all the previous issues or what we need is to design an attachable constraints language.

- There is a fundamental division between those who take the view that generation is simply parsing in reverse, and those who feel that the deeper issues in text organization and planning are where the real action is. The question now is that whether the grammar is reversible at least on the level of utterance realization in the linguistic module.
REFERENCES


COORDINATION AS A PARSING PROBLEM

Crit Cremers
University of Leiden
Department of General Linguistics
Postbox 9515, 2300 RA Leiden, The Netherlands
email: cremers@rullet.leidenuniv.nl

ABSTRACT

Declarative grammar is essentially unable to specify the variety of coordinative structures in natural languages adequately. Parsers nevertheless can be upgraded to handle coordination more subtly than by brute force. This upgrading involves knowledge of the grammar to be parsed and differentiation of parsing stages.

1. COORDINATION AS A GRAMMAR PROBLEM

Natural languages have structures of the form

\[ l_{	ext{sentence}} X Y_1 \text{ coord } Y_2 Z \]

where \text{ coord} is an element equivalent to \text{ and} or \text{ or}. The substring \(Y_1\) \text{ coord} \(Y_2\) is a \text{ coordination}; the element \text{ coord} is said to be the \text{ coordinator}, and \(Y_1\) and \(Y_2\) are dubbed the left and the right \text{ coordinate}, respectively. The substrings \(X\) and \(Z\) may be seen as the left and the right \text{ context}, respectively, of the \text{ coordination}; either of them may be empty.

Coordinations in languages like Dutch and English have several unique and rather nasty properties. The most prominent among them is that the coordinates \(Y_1\) and \(Y_2\) are grammatically equivalent with respect to the context \(X\) \text{ coord} \(Z\), and that there are hardly any other wellformedness conditions on coordination in structures like (1). From this some other characteristics emanate. For example, there is no canonical class of coordinates: any locally wellformed string structure can be a coordinate. Furthermore, coordinations can occur almost anywhere in a structure, if only the equivalence condition is respected. As a consequence, coordinations may themselves be a (part of a) coordinate or occur in the context of another coordination. Moreover, the connection between \(Y_1\) and \(Y_2\) is so tight that the right coordinate can be syntactically incomplete, borrowing its combinatory power from the left coordinate. Integrating these properties, the following sentence is grammatical (coordinators and boundaries of coordinations are marked):

\[ \{\text{Enkele parvenus' met en enkele yuppen zonder} \ [\text{strafblad of schulden} \ \{\text{teasden en verkochten} \ \text{aan ervaren van} \ [\text{huurden en verhuurden} \ \text{voor beginnende} \} \ \text{spekulanten die} \ {\\{\text{niet hier maar wel in A'dam} \ \text{woonden beleggen of} \ \{\text{zijn wel hier maar niet in A'dam}\}} \ \{\text{taitrije nieuwe en enkele oudere} \ [\text{woonblokken en (bedrijfs- of winkelpanden)}]\}

'Some parvenus with and some yuppies without police record or debts leased and sold to experienced or hired and let for starting scalpers that not here but in Amsterdam wanted to invest or not here but in Amsterdam numerous new and some older housing blocks and business or commercial properties'

Coordinators, as it seems, can occur almost everywhere - the main restriction here is the position adjacent to another coordinator - and affect everything that is open to semantic variation. So, if we take a description like (3) instead of (1), neither \(X\) nor \(Z\) conveys specific information on suffixes or prefixes, respectively, which can be involved as a coordinate in the coordination.

\[ X \text{ coord } Z \]

In other words, both \(X\) and \(Z\) can perfectly well serve as substrings of wellformed sentences in which no coordinator at all occurs - abstracting away, for the sake of argument, from the possibility that in a particular coordination the prefix of \(Z\) is elliptic. Consequently, neither \(X\) nor
Z can specify which particular suffix or prefix is involved in a given coordination.

It is evident, then, that the grammar of natural language can handle coordination only in a very liberal manner with respect to possible sites and possible coordinates. As a matter of fact, coordination lives in the free-trade zone of most grammars that aim at covering it. Phrase structure grammars come up with recursive rule schemata of type \( X \rightarrow X \ and \ X \), introducing coordinators syncategorically. Flexible constituency grammars of the categorial breed resort to assigning a completely ungrounded type to coordinators: \( \langle \forall x \rangle x \), taking anything and its typological clone into itself; see e.g. Moortgat (1988) and Wittenburg (1986). Both approaches are bound to fail, I believe, but for different reasons. For context free phrase structure grammar, the rule is too strict since it can only be instantiated by constituents. In higher order phrase structure grammar, the rule scheme will have infinitely many instantiations. Treating coordinators as ungrounded combinatory types, seems to do the job but only by giving up all control of the derivation. In particular, an ungrounded type does not allow for an account of the internal structure of the coordinates.

Recently, it has been suggested that coordinations or parts of them may be seen as constituents in their own right: Coordination or Boolean Phrases. Grooveld (1994) offers extensive discussion of several proposals along these lines. In general, one can say that they restructure (1) as

(4) \[ \text{sentence } X \{ \text{co-phrase } Y_1 \text{ coord } Y_2 \} Z \]

A common feature of these proposals is that they try to normalize what appears to be exceptional. Coordinators are considered to be heads of phrases that are not selected by any other phrase and do not select any particular phrase themselves. Inevitably, the resulting phrase is to be treated in a particular manner, both in relation to other phrases and with regard to the way its syntactic valency is computed internally. The main result of allowing for something like Coordinated Phrases is that coordinations can be addressed by the grammar with finite means. The main problem, however, remains: there is no relevant top-down prediction of the syntactic and semantic properties of these phrases and their constituent parts. The informational content of a node CoPhrase hardly exceeds the informational content of the leaf Coordinator. It is fair to say, I believe, that this criticism extends to the propositions of Sag et al. (1985). They construct coordination in terms of categorial underspecification, which is a fascinating enrichment of the theory of categories (for an application, see Grooveld 1994), but also fails to determine the scope of coordination in a given coordinated structure.

By and large, there is reason to doubt that all aspects of coordination can be dealt with by homogeneous declarations in the grammar. Fortunately, there is no reason to believe that the grammar of natural language is strictly declarative. In the remainder of this paper I would like to consider the option that coordination is a mainly procedural phenomenon.

2. Coordination as a Problem for Parsing

As things are, parsers had better not rely, for the task of recognizing and structuring coordination in natural language, on the information encoded in explicit grammars. That information, to the extent that it is adequate, is either infinite or zero. The most remarkable and almost paradoxical feature of coordination in this respect is that in grammatical sentences there is, at the level of non-terminal strings, for each coordinate exactly one consistent answer to the question which are its coordinates. That is, in (5), considered as a grammatical string of non-terminals, exactly one non-empty suffix of the a-sequent and exactly one non-empty prefix of the b-sequent will qualify as coordinates.

(5) \[ \text{sentence } a \ldots \text{ coord } b \ldots \text{ bun } \]

There appears to be a wide gap between the grammatical means to specify coordinated structures and the determinism of coordination in specific cases. Thus, grammars are simply not equipped to provide adequate backing for the parser's main task regarding coordination: to determine the coordinates, i.e. to determine the left corner of \( Y_i \) and the right corner of \( Y_j \) at input (1).

Let us establish that a parser must identify the coordinates whenever a coordinator occurs in the input. A parser that fails to determine the extension or the content of the coordinates, will also fail to assign any meaningful structure to the
input as a whole and to specify the semantic impact of the coordinator itself (boolean, semi-boolean or non-boolean, for example).

The problem with which coordination confronts a natural language parser, is easily described:

(6) For any input $X$ coord $Z$ and a hypothesis that the input is of category $C$, find a partition $\langle \text{prefix}(X), \text{suffix}(X) \rangle$ of $X$ and a partition $\langle \text{prefix}(Z), \text{suffix}(Z) \rangle$ of $Z$ such that
   (a) $\text{suffix}(X)$ and $\text{prefix}(Z)$ are grammatically equivalent, and
   (b.1) both $X$ $\text{suffix}(Z)$ and $\text{prefix}(X)$ $Z$ are meaningful expressions of category $C$, or
   (b.2) $X$ $\text{suffix}(Z)$ is a meaningful expression of category $C$, and there is some completion $Y$ of $\text{prefix}(Z)$ with respect to $\text{suffix}(X)$ and $\text{prefix}(X)$ $Y$ $\text{suffix}(Z)$ is a meaningful expression of category $C$.

Condition (6a) requires the coordinates to be syntactically, semantically and prosodically idempotent. A condition of this or similar import is designed to rule out coordinates like

(7) * ... (omdat) Jan aardappelen en onsmakelijk at
    ... (because) Jan potatoes and distastefully atc

Although both ...Jan aardappelen at and ... Jan onsmakelijk at are wellformed and meaningful, the coordination is not. In this case, the ungrammaticality could be predicted by any serious grammatical declaration of coordination. One cannot expect, however, the grammar to predict the ungrammaticality of the following mismatches where, again, the coordinates separately are combinable with the context:

(8) * Haar gaf hij en ik gaf Kees een zoen
    'her gave he and I gave Kees a kiss'
(9) * ... omdat Jan met smaak aardappelen en gisteren onsmakelijk at
    'because Jan with relish potatoes and yesterday distastefully ate'

The alternative conditions (6b) express the general requirements for 'normal' continuous coordinations (b.1) and elliptic discontinuous instances (b.2). I will neglect here the question whether coordination is essentially phrasal or sentential, but see Cremers (1993: chapter 2) for an extensive argument in favour of the sentential position.

The procedure sketched in (6) essentially introduces a guess with regard to the interdependent partitions of $X$ and $Z$. If the grammar does not give effective declarative hints as to the internal and external structure of the coordination in the input, the parser must check a number $O(n^2)$ of possible analyses for each coordinator in an input of order $n$. Thus the complexity of the parsing procedure for coordinated structures equals $k^*O(n^2)^kC$, where $k$ is the number of coordinators in the input and $C$ the complexity of the coordination-free part of the procedure. This is an intolerable increase in complexity, given our knowledge that for each particular coordinator only one pair of coordinates can do the job.

This complexity-increasing trial and error procedure can be related to the proposition, made by Cremers (1993: p. 142 - 143), that there will be no particular marked state of a genuine parse at which both coordinates of a coordination are revealed. In other words, a parser that follows any fixed overall strategy, cannot determine whether or not it has reached a state in which the coordinates show up. For example, an incremental parsing procedure will combine a coordinate and its context without being aware or being warned that this reduction step may destroy the integrity of the coordinate as such.

3. RESOLVING COORDINATION

In this section it will be argued, following Cremers (1993: chapter 3) that the lack of declarative power regarding coordination of the grammar and the complexity increasing effects thereof on parsing can be compensated by upgrading the parser with metaknowledge of the coordination-free part of the grammar. Of course this knowledge is grammar-dependent, and consequently the resolution of coordination will become a grammar-dependent but extragrammatical procedure.

The main idea is this. Treat coordinators as grammatically inert items. Then, a parser can
initially compute as much of the input as the coordination-free part of the grammar permits. This partial computation being completed, the resulting state of the parse must contain information on the way in which the coordinator blocks further computation. That information can be used to predict where the left and the right corner of the coordination are to be found. In this vein, the parsing of coordinative input comprises some distinct stages:

(10) \textit{stage 1}: compute the grammar, neglecting coordinators

\textit{stage 2}: for each coordinator, identify its coordinates by inspecting the results of stage 1

\textit{stage 3}: compute the grammar on each coordinate and its context.

Cremers (1993; ch. 3) applies these tactics to the parsing of a rigid (as opposed to flexible) categorial grammar. The conclusion of that exercise is that to the left and the right of the inert coordinator two partial structures can be identified by their types (top nodes), which indicate where in the structure the corners of the coordination must be found by their being adjacent or non-adjacent to the coordinator. Given a maximally reduced partial analysis of the coordination-free part of the input, the parser can find, inspecting only the top nodes of the partial result, two partial structures \( L \) and \( R \) such that the following conditionals hold.

(11) Let \( \ldots \mathrm{A.L..} \quad \mathrm{coat} \quad \ldots \mathrm{R.B.} \ldots \) be a maximally reduced covering sequence of partial analyses of an input \( X \mathrm{coat} \ Z \). Then:

(a.1) if \( L \) is adjacent to \( \mathrm{coat} \), it properly contains the left corner of the coordination

(a.2) if \( R \) is adjacent to \( \mathrm{coat} \), it properly contains the right corner of the coordination

(b.1) if \( L \) is adjacent to \( \mathrm{coat} \) and \( R \) is not, the left neighbour of \( R \) is the right corner of the coordination

(b.2) if \( R \) is adjacent to \( \mathrm{coat} \) and \( L \) is not, the right neighbour of \( L \) is the left corner of the coordination

(c) if neither \( L \) nor \( R \) is adjacent to \( \mathrm{coat} \), both of them properly contain a corner of the coordination.

If \( L \) or \( R \) cannot be identified, or if both are adjacent to \( \mathrm{coat} \), no coordination can be established.

In order to identify \( L \) and \( R \), the parser has to make some moves linear to the number of partial structures in the covering sequence. The parser knows, however, what must be the grammatical properties of at least one of the types it is looking for. More importantly, these moves determine the \( \text{loci} \) of the corners of the coordination, and are not an order-increasing factor in the complexity of the computation as a whole.

The statements (a) and (c) of lemma (11) indicate that the corner of a coordination can be properly contained in one of the relevant partial structures. In order to determine exactly which is the corner, the partial structure must be decomposed (see also Steedman 1990). In addition to the existence of a pair \( \langle L.R \rangle \), this decomposition is a characteristic entailment of the strategy applied. The need for decomposition reflects what was referred to at the end of the preceding section: it is impossible for the parser to recognize a coordinate regularly. Note that there is no linguistic reason for this decomposition; as a matter of fact, the decomposed structure must and will be restored when the parser continues on its way in the standard mode of computing the grammar according to task (6b).

Decomposition under the conditions presented here does not increase the order of complexity of the procedure as a whole. The number of decompositional steps does not depend on the complexity of the input; its maximum equals the degree of embedding in the partial structure that is subjected to decomposition. Moreover, this decomposition is completely deterministic (cf. Cremers 1993: section 3.4.4).

It is clear that the information the parser is equipped with to find the landmarks \( L \) and \( R \) of a coordination depends on the grammar that is parsed. It is by no means clear, however, that adequate information in this respect can be derived from each and every grammar. It may well be the case that only grammars that impose some kind of fixed constituency give rise to specifications of the sort needed to determine the landmark structures. It is encouraging, however, that Grooteveld (1992, 1994), though dealing with grammar types, parsing models and linguistic principles that differ considerably from those that led to (11), nevertheless arrives at a system that has much in common with the proposal made here. She too defends an approach to coordination that crucially differentiates between stages of parsing and in
which the reconstruction of the coordination is brought about by specialized moves of the parser, steered by knowledge about the background grammar.

In general, then, this type of strategy affects the independence of the parser. This, I believe, is inevitable; it is the price to pay for deterministic efficiency. Coordination invites us, for efficiency’s and transparency’s sake, to reallocate part of our knowledge of language to the procedural module. Complying with this call entails that some knowledge of language is instantiated as knowledge of procedures in a dedicated parser. Coordination may therefore induce a shift of perspective on the relation between declarations and procedures.

REFERENCES

- Wittenburg, K.B. *Natural Language Parsing with Combinatory Categorial Grammar in a Graph-Unification-Based Formalism*. Dissertation. University of Texas at Austin. 1986
Bounded Incremental Parsing

Mats Wirén
Fachrichtung 8.7, Computerlinguistik
Universität des Saarlandes
Postfach 1150
D-66041 Saarbrücken, Germany
wiren@coli.uni-sb.de

Abstract

Ideally, the time that an incremental algorithm uses to process a change should be a function of the size of "the set of things changed" rather than the size of the entire current input. Building upon a previous notion of change, this paper investigates how and to what extent it is possible to give such a guarantee for a chart-based parsing framework. Two results are provided: first, it is shown that a previously proposed algorithm is unbounded incremental; secondly, it is outlined how a polynomial time bound can be obtained by refining this algorithm.

1 Introduction

1.1 Background

Parsing has traditionally been understood as a "batch-mode" or "once-only" process, in which an input text is mapped as a whole to an output representing an analysis of the text. However, in highly interactive and real-time applications — for example, grammar checking, structure editing and on-line translation — what is required is efficient analysis of a sequence of small changes of a text. Exhaustive recomputation is then not a feasible alternative. Rather, to avoid as much recomputation as possible, each update cycle must re-use those parts of the previous analysis that are still valid. We say that an algorithm is incremental if it uses information from an old analysis in computing the new analysis.

Ideally, each update cycle should expend an amount of work which is a polynomial function of the size of "the set of things changed" rather than the size of the entire current input. However, making this notion precise in a way which is independent of particular incremental algorithms is not always straightforward. Two early approaches along these lines are Goodwin [6, 7] (reason maintenance) and Reps [13] (language-based editing). More recently, Alpern et al. [1] and Ramalingam and Reps [12] have provided a framework for analysing incremental algorithms, in which the basic measure used is the sum of the sizes of the changes in the input and output.

This framework presupposes a problem instance (a representation of the current input) $P$ and a solution $S$ (the current output). When $P$ is changed to $P'$, the task of the incremental algorithm is to find a new solution $S'$. The new instance $P'$ is given as a modification $\Delta P$ to $P$, written with a composition operator as $P' = P \oplus \Delta P$. It is assumed that the modification of the input can be carried out in $O(|\Delta P|)$ time, where the generic notation $|X|$ is used for the size of $X$. The task of an incremental algorithm is to produce a change $\Delta S$ in the old solution such that $S \oplus \Delta S$ is a solution to $P \oplus \Delta P$. Nothing is then stipulated about the amount of information in $S$ that should be re-used in $S'$. Now, assume that, according to some measure, $|\Delta s_{min}|$ denotes the minimal $|\Delta S|$ such that $S \oplus \Delta S$ solves $P \oplus \Delta P$. Alpern et al. define

$$\delta = |\Delta P| + |\Delta s_{min}|$$

as the intrinsic size of a change.\footnote{We have modified the notation of Alpern et al. [1] slightly.}

The choice of $\delta$ is motivated as follows: $|\Delta P|$, the size of the modification, is in itself too crude a measure, since a small change in problem instance may cause a large change in solution or vice versa. $|\Delta s_{min}|$ is then chosen as a measure of the size of the change in the solution, since the time for updating the solution can be no less than this. The $\delta$ measure thus makes it possible to capture how well a particular algorithm performs relative to the amount of work that must be performed in response to a change.

An incremental algorithm is said to be bounded if it can process any change in time $O(f(\delta))$, that is, in time depending only on $\delta$. Intuitively, this means that it only processes the "region" where the input changes or output changes. Algorithms of this kind can then be classified according to their respective degrees of boundedness (see Ramalingam and Reps [12, section 3]). For example, an algorithm which is linear in $\delta$ is asymptotically optimal. Furthermore,
an incremental algorithm is said to be \textit{unbounded} if the time it takes to update the solution can be arbitrarily large for a given \( \delta \).

It might seem that what has been discussed so far has little relevance to natural-language processing, where incrementality is typically understood as the piecemeal assembly of an analysis during a single left-to-right\(^2\) pass through a text (or a spoken utterance). In particular, incrementality is often used as a synonym for \textit{interleaved} approaches, in which syntax and semantics work in parallel such that each word or phrase is given an interpretation immediately upon being recognized (see, for example, Bowers and Webber [2], Mellish [11], and Haddock [8, 9]). However, the two views are closely related: The "left-to-right view" is an idealized, psycholinguistically motivated special case, in which the only kind of change allowed is addition of new material at the end of the current text, resulting in piecemeal expansion of the analysis. Moreover, the interleaving is just a consequence of the fact that every piece of new input must, in some sense, be fully analysed in order to be integrated with the old analysis.

Wirén and Rönquist [16, 17] refer to the special case as left-to-right (LR) \textit{incrementality} and to the general case, in which arbitrary changes are allowed, as \textit{full incrementality}. The latter case has long been studied in interactive language-based programming environments (for example, Ghezzi and Mandrioli [4, 5]), whereas the only previous such work that we are aware of in the context of natural-language processing is Wirén and Rönquist [16, 16, 17].

\subsection{1.2 The Problem}

The aim of this paper is to adapt and apply the notion of bounded incremental computation to natural-language processing. Specifically, the paper investigates how the \( \delta \) parameter can be defined in a fully incremental, chart-based parsing framework, what the behaviour of the algorithm originally introduced by Wirén [15] is in terms of this measure, and how an improved, polynomially bounded algorithm can be obtained on the basis of this.

\subsection{1.3 Organization of the Paper}

The rest of this paper is organized as follows: Section 2 specifies a batch-mode version of chart parsing. Based on this, section 3 defines the problem of incremental chart parsing and the \( \delta \) notion as conceived within this framework. Using the previously introduced notions, section 4 then shows that the algorithm put forward by Wirén [15] is unbounded. Furthermore, section 5 outlines how instead a polynomially bounded algorithm can be obtained by refining the unbounded algorithm. Section 6 discusses what has been achieved and how this line of research can be continued. Finally, section 7 provides the conclusions.

\section{2 Batch-Mode Chart Parsing}

An incremental problem can be defined by specifying its batch-mode version and the set of allowable changes. To specify batch-mode chart parsing, we begin with some preliminary notions. First, \( R \) is the set of dotted context-free rules obtained from the grammar. A dotted rule \( X \rightarrow \alpha \cdot \beta \) corresponds to an \( X \) edge containing an analysis of constituents \( \alpha \) (the contents part), seeking constituents \( \beta \) (the needed part). \( Q \) is the set of (partial) analyses according to the grammar and lexicon. For the purpose of the parsing algorithms, we shall restrict our attention to a standard context-free grammar without cyclic or empty productions.

\begin{definition}[Chart] A \textit{chart} is a directed graph \( C = (V, E) \) such that \( V \) is a finite, non-empty set of vertices and \( E \subseteq V \times V \times R \times Q \) is a finite set of edges.
\end{definition}

The vertices \( v_1, \ldots, v_{n+1} \in V \) correspond to the linear positions between the tokens \( t_1 \cdots t_n \) of an \( n \)-token text.\(^3\) An edge \( e \in E \) between vertices \( v_i \) and \( v_j \) carries information about a (partially) analysed constituent between the corresponding positions. For brevity, we shall sometimes omit the \( Q \) component, writing an edge as a triple \( (v_i, v_j, X \rightarrow \alpha \cdot \beta) \).

We shall also need the following notions:

\begin{definition}[Crossing edge] We say that an edge \( e = (v_i, v_j, X \rightarrow \alpha \cdot \beta) \) \textit{crosses} a vertex \( v_s \) if \( i < s < j \). Furthermore, we say that \( e \) crosses a vertex interval \( v_s, \ldots, v_t \) if \( i \leq s \) and \( t \leq j \).
\end{definition}

\begin{definition}[Width of edge] We define the \textit{width} of an edge \( e = (v_i, v_j, X \rightarrow \alpha \cdot \beta) \) by the function \( \text{Width}(e) = j - i \). A zero-width edge is called a \textit{looping edge}.
\end{definition}

The algorithm makes use of an agenda (see Thompson [14]). Agenda tasks are created in response to tokens being read and edges being added to the chart, and may be ordered according to their priorities. To define the agenda, we make use of the

\footnotesize\(^2\)Strictly speaking front-to-back or beginning-to-end. In its strong form, left-to-right analysis means that each prefix of a string is parsed (interpreted) before any of the input beyond that prefix is read (on-line analysis; see Harrison [10, page 435]).

\footnotesize\(^3\)We shall use \( \tau \) interchangeably to denote a sequence and a set of tokens.
set of possible tokens $\textit{Tokens}$ and the set of possible edges $\textit{Edges}$.

**Definition 4 (Agenda)** We define the agenda as $\textit{Agenda} \subseteq \textit{Tokens} \cup \textit{Edges} \cup (\textit{Edges} \times \textit{Edges})$. We refer to the three types of tasks that it contains as scanning, prediction, and combination tasks, respectively.

Each agenda task is executed by a step of the algorithm below. We specify two versions of batch-mode chart parsing — the basic bottom-up (strictly speaking, left-corner) and top-down (Earley-style) strategies — assuming that the one or the other is chosen.

**Algorithm 1 (Batch-mode chart parsing)**

**Input:** A sequence of tokens $\tau = t_1 \cdots t_n$.

**Output:** A chart.

**Initialization:** If the top-down strategy is used, then add an agenda task corresponding to an initial top-down prediction $\langle v_j, v_k, S \rightarrow \alpha \rangle$ for each rule $S \rightarrow \alpha$, where $S$ is the start category of the grammar.

**Method:** For each token, create a scanning task. While the agenda is not empty, remove the next task and execute the corresponding step below:

**Scan:** Given a token $t$ at position $j$, for each lexical entry of the form $X \rightarrow t$, add an edge $\langle v_j, v_{j+1}, X \rightarrow t \rangle$. Adding new tasks to the agenda.

Informally, this means adding an inactive edge for each sense of the token. We refer to the new edges as *lexical* edges.

**Predict 1 (Bottom-up):** If the edge is of the form $\langle v_j, v_k, X \rightarrow \alpha \rangle$, then, for each rule of the form $Y \rightarrow X \gamma$, add an edge $\langle v_j, v_j, Y \rightarrow X \gamma \rangle$ unless it already exists. Adding resulting new tasks to the agenda.

Informally, this means adding an edge according to each rule whose right-hand-side category matches the category of the triggering inactive edge, provided that an equivalent edge does not already exist in the chart. We refer to the new edges as *predicted* edges.

**Predict 2 (Top-down):** If the edge is of the form $\langle v_i, v_j, X \rightarrow \alpha \cdot Y \gamma \rangle$, then, for each rule of the form $Y \rightarrow \gamma$, add an edge $\langle v_j, v_j, Y \rightarrow \gamma \rangle$ unless it already exists. Adding resulting new tasks to the agenda.

Informally, this means adding an edge according to each rule whose first left-hand-side category matches the first needed category of the triggering active edge, provided that an equivalent edge does not already exist in the chart. Again, we call the new edges *predicted* edges.

**Combine:** If the first edge is of the form $\langle v_i, v_j, X \rightarrow \alpha \cdot Y \beta \rangle$ and the second is of the form $\langle v_j, v_k, Y \rightarrow \gamma \rangle$, then add an edge $\langle v_i, v_k, X \rightarrow \alpha Y \beta \rangle$. Add resulting new tasks to the agenda.

Informally, this means adding an edge whenever the category of the first needed constituent of the active edge matches the category of the inactive edge. We refer to the new edge as a combined edge.

### 3 Incremental Chart Parsing

#### 3.1 The Problem

The overall incremental process can be thought of as a change-update loop, where each change of the input is immediately followed by a corresponding update of the output. To completely specify the state of this process, we shall make use of a configuration consisting of (a representation of) an input text $\tau$, a chart $C$ and an edge-dependency relation $D$ (to be defined in section 4.1). The problem of incremental chart parsing can then be specified abstractly as a mapping

$$f(\langle \tau, C, D \rangle, \Delta) \mapsto \langle \tau', C', D' \rangle$$

from an old configuration and a change $\Delta$, to a new configuration. We shall allow two kinds of change, namely, insertion and deletion of $m \geq 1$ contiguous tokens. We assume that a change $\Delta$, is encoded as a vertex pair $v_i, v_{i+m} \in V$ defining the update interval and, in the case of an insertion, a sequence of tokens $\tau = t_j \cdots t_{i+m}$. We furthermore assume that either the bottom-up or top-down strategy is chosen throughout a change-update session, and that, when the latter is chosen, the top-down initialization is made before the session is started.

#### 3.2 A General Vertex Mapping

How can $|\Delta_{\text{inst}}|$, the minimal update needed in response to a change, be established in a chart-based framework? One way of doing this is to compare the charts $C = \langle V, E \rangle$ and $C' = \langle V', E' \rangle$ that are obtained by batch-mode parsing of the texts before and after a change, respectively. We thereby obtain a measure which is independent of particular incremental update algorithms. Intuitively, only those edges that are in $C$ but not in $C'$ must be removed.
and only those edges that are in $C'$ but not in $C$ must be generated anew. If the change is small, then a large fraction of the edges are in $C \cap C'$ (that is, are unchanged).

However, to be able to compare the edge sets in the two charts, we must first establish a one-to-one mapping between their vertices. Let us consider the case in which a single token $t_i$ is deleted from an $n$-token text. The problem is that, because of the removed token, the two vertices $v_i$ and $v_{i+1}$ would seem to correspond to a single vertex in $V'$. However, we can regard this single vertex as consisting of a “left half” and a “right half”, which we assign different indices. In other words, after having increased each index of $v_{i+1}, \ldots, v_n \in V'$ by one, we “split” vertex $v_i$ and assign the index $i + 1$ to its “right half”. The incoming non-predicted edges as well as (looping) top-down predictions at the split vertex are then associated with its left half, and the outgoing non-predicted edges as well as (looping) bottom-up predictions are associated with its right half.\(^6\) The reason for dividing the predicted edges in this way is that a top-down prediction is made at the ending vertex of the triggering edge (that is, from the left), whereas a bottom-up prediction is made at the starting vertex of the triggering edge (that is, from the right).

The mapping can be generalized to the case in which $m$ contiguous tokens are deleted. This is done by increasing the index of each vertex from the “right half” of the split vertex and onwards by $m$ (instead of one). Furthermore, by using the same mapping but in the opposite direction, we can also cover insertion of $m$ contiguous tokens. To express this generalized mapping, assume that $\tilde{V}$ is the set of vertices of the larger chart and $V$ is that of the smaller chart. A deletion of $m$ contiguous tokens then involves a mapping from $\tilde{V}$ to $V$ and an insertion of $m$ tokens involves a mapping from $V$ to $\tilde{V}$. In terms of the indexing that holds before the vertices in $V$ are renumbered, and assuming that $\tilde{V}$ has $n+1$ vertices, we obtain the following bidirectional mapping:

- Vertices $\tilde{v}_1, \ldots, \tilde{v}_{i-1} \in \tilde{V}$ correspond to $v_1, \ldots, v_i \in V$, respectively.
- Vertex $\tilde{v}_i$ corresponds to the “left half” of vertex $v_i$.
- Vertices $\tilde{v}_{i+1}, \ldots, \tilde{v}_{i+m-1} \in \tilde{V}$ do not correspond to any vertices in $V$.
- Vertex $\tilde{v}_{i+m}$ corresponds to the “right half” of vertex $v_i$.

- Vertices $\tilde{v}_{i+m+1}, \ldots, \tilde{v}_{n+1}$ correspond to $v_{i+1}, \ldots, v_{n+1-m}$, respectively.

The mapping is thus established with respect to insertion or deletion of an arbitrary number of contiguous tokens.\(^6\)

### 3.3 The Size of a Change

Assume that $E$ and $E'$ are the sets of edges of the charts $C$ and $C'$ obtained by batch-mode parsing of a text before and after a change, respectively. Defining the size of the input change (modification) is trivial:

**Definition 5 (Size of input change)** Given an input change $\Delta_I$, we define the size of the input change as $|\Delta_I|$, the number of inserted or deleted tokens.

The size of the (minimal) output change can be defined on the basis of two edge sets as follows:

**Definition 6 (Size of output change)** We define the set of affected (decremented) edges\(^6\) as the set difference $A = E \setminus E'$ and the set of incremented edges as the set difference $I = E' \setminus E$. Furthermore, we define the output change as $\Delta_{C_{out}} = A \cup I$. Finally, we define the size of the output change as $|\Delta_{C_{out}}| = |A \cup I|$, the number of edges in $A \cup I$.

We are now ready to define the size of “the set of things changed”; that is, the (intrinsic) size of the change of the input and output:

**Definition 7 (Size of change)** We define the size of a change in the input and output as $\delta = |\Delta_I| + |\Delta_{C_{out}}|$.

The size of a change is thus defined independently of any incremental algorithm for performing the update.

### 3.4 An Example

As an illustration, the chart in figure 2 is obtained under bottom-up parsing, given the grammar in figure 1 and the sentence “The old man the tall ships”. If the token “tall” is removed, the chart in figure 3 is obtained. Vertex $v_9$ in figure 2 then corresponds to the left half of vertex $v_9$ in figure 3, and vertex $v_9$ corresponds to the right half of vertex $v_9$. Furthermore, $v_7$ corresponds to $v_9$. Clearly, the input change $\Delta_I$ consists of the token “tall”. The output change $\Delta_{C_{out}}$ consists of the affected set $A$, which

\(^6\) As mentioned above, we assume that only one or the other strategy is used, so that it is known beforehand which kind of predictions the chart contains.

\(^7\) These were called "regenerated" edges in Wirén and Rönning [17].
Figure 1: Example grammar and lexicon.

contains the three edges $\Delta_{26}$, $\Delta_{27}$ and $\Delta_{34}$ in figure 2, and the incremented set $I$, which contains the single edge $\Delta_{32}$ in figure 3. The size of the change is then $\delta = |\Delta_r| + |\Delta_{C_{inc}}| = 1 + 3 + 1 = 5$.

If instead “tall” is inserted before the last word in the sentence in figure 3, then the input change still consists of the token “tall”. However, the two sets making up the output change are reversed: the affected set contains the single edge $\Delta_{32}$ in figure 3 and the incremented set contains the three edges $\Delta_{26}$, $\Delta_{27}$ and $\Delta_{34}$ in figure 2. Thus, the size of the change is again 5.

4 An Unbounded Algorithm

4.1 Edge Dependencies

A key idea of the incremental chart-parsing algorithm put forward by Wirén [15, 16] is to use edge dependencies for keeping track of edges that have to be removed in response to a change. Essentially, the algorithm performs an update by removing the entire disturbance set and then generating all possible new edges, where the latter include both the incremented edges and any disturbed but non-affected edges.

An edge $e'$ is said to depend upon another edge or token if it is formed (derived) directly on the basis of $e$. Furthermore, if $e'$ is redundantly proposed by an edge $f$, then $e'$ can be said to depend (also) on $f$. By $e'$ being “redundantly proposed”, we mean that the parser attempts to add an edge that is equivalent to $e'$ to the chart, but that that edge is rejected by the standard redundancy test in chart parsing. In effect, $f$ provides an additional “justification” for $e'$.

Given a chart $C = (V, E)$ and a set of tokens $\tau$, these conditions correspond to the following dependency relation on $E$ and $\tau$:

Definition 8 (Edge dependency) We define $D$ as a binary relation on the set of chart edges and the set of tokens $E \cup \tau$ such that $D(s, d)$ holds if and only if $d \in E$ is formed, or is redundantly proposed, directly using $s \in E \cup \tau$ according to a chart-parsing algorithm. We say that $d$ is a dependent (or derivative) edge of $s$, and that $s$ is a source edge (token) of $d$.

Under a context-free grammar, only predicted edges can be redundantly proposed, whereas under a unification grammar such as PATR, both predicted and combined edges can be redundantly proposed. In the former case, the following dependencies are thus induced by the three chart-parsing steps:

- A predicted edge depends upon the edge that has triggered it and the edges that have redundantly proposed it, unless it is an initial top-down prediction, which does not depend on any edge.
- A combined edge depends upon the active and inactive edge that have formed it according to the combine step.
- A lexical edge depends upon the token that triggered it.

$D$ can be illustrated by the corresponding graph. Given the grammar in figure 1 and the sentence “The old man the ships”, the chart in figure 3 is obtained under bottom-up parsing. The resulting dependency graph is then shown in figure 4. The set of edges that depend (directly or indirectly) on the token “tall” in figure 2 contains the seven edges $\Delta_{26}$, $\Delta_{27}$ and $\Delta_{34}$, $\Delta_{35}$, $\Delta_{36}$, $\Delta_{37}$ and $\Delta_{38}$. Of these, only $\Delta_{26}$, $\Delta_{27}$ and $\Delta_{34}$ are affected, as mentioned in section 3.3. $\Delta_{35}$, $\Delta_{36}$, $\Delta_{37}$ and $\Delta_{38}$ are unchanged, on the other hand, because each one of them exist in the new analysis in figure 3, namely, $\Delta_{33}$, $\Delta_{34}$, $\Delta_{35}$ and $\Delta_{38}$, respectively. For example, $\Delta_{36}$ in figure 2 and $\Delta_{34}$ in figure 3 both represent the partial analysis $V P \rightarrow V N P$.

4.2 Disturbance Sets

On the basis of the dependency relation, Wirén and Rönquist [16, 17] define different disturbance sets, which contain edges that need to be removed from the chart in response to a token-level change. The abstract definition of this notion is as follows:

Definition 9 (Disturbance) We define a disturbance as a function $Disturb: \tau \rightarrow 2^E$. We call the set given by the function the disturbance set of a token $t_j \in \tau$.

In making the definition of the disturbance set with respect to a token $t_j \in \tau$ concrete, a natural first approach is to let it be identical to $D(t_j)$, the transitive closure of $D(t_j)$. Wirén and Rönquist [16, 17] prove that this set is complete with respect to the set of affected edges. Next, a reduction of the set is possible with respect to predicted edges that have several source edges that are combined: only the combined sources with the least width need to be included in the disturbance set. Wirén and
Figure 2: Chart of the sentence “The old man the tall ships” under bottom-up parsing. Inactive edges are drawn using continuous lines, active edges using dashed lines, and predicted (looping) edges are depicted below the vertices.

Figure 3: Chart of the sentence “The old man the ships” under bottom-up parsing.
Figure 4: Edge-dependency graph induced by \( D \). The nodes of the graph correspond to the chart edges in figure 3, whereas a dummy root node 0 is shown instead of nodes corresponding to the tokens. Furthermore, the dependency between the only redundantly proposed edge \( S_{10} \) and the source \( NP_{18} \) is not shown in the figure.
Rönnquist [16, 17] call this reduced set the primary disturbance set and again prove that it is complete with respect to the set of affected edges. Finally, in order to reduce the disturbance set under top-down parsing, a further possible reduction is to exclude top-down predictions altogether (again, see [16, 17]).

4.3 Incremental Complexity

The complexity analysis of the algorithm of Wirén [15, 16] yields that it is unbounded incremental in both its bottom-up and top-down version. The source of this is that the algorithm removes the entire disturbance set, which is unbounded in $\delta$, in response to a change.

For the purpose of analysing the incremental complexity of the algorithm, we assume that adding or removing an edge takes unit time. We also assume that no edge has more than a constant number of sources or dependants and, hence, that the time required to install or examine the dependencies of $k$ edges is $O(k)$. We shall consider a small change to an $n$-token text, namely, deletion of a token $t_j$ (thus, $m = 1$). However, an insertion within the text, which involves a disturbance set in connection with “splitting” the chart, gives the same result. We assume a chart $C = (V, E)$, and refer to the subchart including $v_1, \ldots, v_j$ (to the left of the update interval) as $C_l = (V_l, E_l)$ and the subchart including $v_{j-1}, \ldots, v_n$ (to the right of the update interval) as $C_r = (V_r, E_r)$. By “disturbance set” we mean any of the three variants mentioned in section 4.2. $|G|$ is a measure of the size of the grammar (namely, the number of occurrences of categories in the grammar).

We begin with two lemmas:

Lemma 1 Given deletion of token $t_j$, the disturbance set contains at least all the edges that cross the update interval $v_j, v_{j+1}$.

Proof Consider any non-lexical edge $e$ that crosses the interval $v_j, v_{j+1}$. Since $e$ does not loop, it must be a combined edge. According to algorithm 1, a combined edge spans the combined vertex interval of its two source edges. Therefore, there must be some other edge $e'$ that crosses the interval $v_j, v_{j+1}$ such that $D^*(e', e)$. This edge is then either lexical or combined.

- If it is lexical, then $D^*(t_j, e')$ and hence $D^*(t_j, e)$, since a lexical edge cannot be part of a dependency cycle. Hence the lemma follows in this case.
- If it is combined, then assume that $e$ and $e'$ belong to a dependency cycle, that is, $D^*(e', e)$ and $D^*(e, e')$. Now, since the formation of an edge relies on a prior formation (or existence) of source edges, there must be some combined edge $e''$ in the cycle that is formed prior to the other edges in the cycle. Because all dependencies are grounded in tokens by definition 8, and because combined edges span the intervals of their sources, it must then hold that $D^*(t_j, e'')$ and, hence, $D^*(t_j, e)$. The same holds if $e$ and $e'$ do not belong to a dependency cycle. Hence the lemma follows also in this case.

Lemma 2 For a small $m (m = 1)$ and a fixed $j \leq n$, the number of edges that cross the update interval is $|E_{cross}| = O(|G| \cdot n)$.

Proof If $m = 1$, the number of edges that cross the update interval is $|E_{cross}| = |E| - (|E_l| + |E_r|)$, where $|E| = O(|G| \cdot n^2)$, $|E_l| = O(|G| \cdot j^2)$ and $|E_r| = O(|G| \cdot (n - j + m)^2) = O(|G| \cdot (n^2 - 2nj - mn + 2mj + j^2 + m^2))$. Thus $|E_{cross}| = O(|G| \cdot (n^2 - j^2 - (n^2 - 2nj - 2mn + 2mj + j^2 + m^2))) = O(|G| \cdot (2nj + 2mn - 2mj - m^2))$. If $m = 1$ and $j \leq n$, the dominating term in this expression is $|G| \cdot n$.

The unboundedness of the algorithm is then an immediate consequence of the following theorem:

Theorem 1 The disturbance set is unbounded in $\delta$.

Proof The proof can be obtained by comparing problem instances of different sizes to which the same small change is made (say, $m = 1$). Suppose first that a change is made to an $n$-token text $\tau$ such that $\delta = |\Delta_\tau| + |\Delta_{G_m}|$. By lemma 2, the number of edges in the disturbance set is (at least) $O(|G| \cdot n)$. Now consider a monotonically expanded problem instance with $C' \supset C$, obtained by inserting $\Delta_\tau$ tokens at the end of $\tau$. Suppose that the same change is made, and that this is constructed together with the expanded problem instance in such a way that the size of the change is still $\delta$. In other words, the change is local to $C$ in the sense that no edges are added to $A \cup I$ in the expanded problem instance; that is, it holds that $(C' \setminus C) \cap (A \cup I) = \emptyset$. This can be achieved, say, by making the same small change to a noun phrase, and adding a prepositional phase to the noun phrase in the expanded problem instance such that there are no incremented edges within the noun-phrase interval. The number of edges in the disturbance set is then $O(|G| \cdot (n + \Delta n)^2)$ in the expanded case. However, since $\delta$ is constant, and since
$n$ is a variable that is independent of $\delta$, the disturbance set cannot be bounded by $\delta$. \vfill

As seen by the proof, the source of the unboundedness is that the size of the set of non-affected disturbed edges, which is first removed and then reinstalled by the algorithm, is independent of $\delta$.

Although the algorithm is unbounded in general, there are two important special cases in which it is linearly bounded, namely, insertion and deletion at the end of a text. To see this, assume first that $\Delta_r$ is an insertion at the end of a text $\tau$. Since LR-incremental chart parsing is monotonic, the affected set is necessarily empty. Therefore, $\Delta_{can}$ consists only of an incremented set $I$. But this set is also precisely what the incremental chart parser computes in response to the added input, since the end result of the batch-mode and incremental algorithms are the same. The only extra work made by the incremental algorithm is installation of edge dependencies with respect to the edges in $I$. However, this only amounts to a constant factor in the number of new edges $|I|$, and the time of the algorithm is therefore bounded by $O(|\Delta_r|) + O(|I|) = O(\delta)$ in this case.

Deletion at the end is the dual case of this, and $\Delta_{can}$ then only consists of an affected set $A$. But given that we restrict ourselves to the primary disturbance set (Wirén and Rönning [16, 17]), the edges in $A$ are precisely what the incremental algorithm removes, and hence it is linearly bounded by $O(|\Delta_r|) + O(|A|) = O(\delta)$ also in this case.

5 A Bounded Algorithm

5.1 Intuitive Idea

Intuitively, a bounded incremental algorithm only processes the region where the input or output changes during an update cycle. The problem in achieving this is that the affected and incremented edges are not a priori known — when the incremental update begins, only a set of potentially affected edges (the disturbance set) is known. However, the update can be limited by using a change-propagation algorithm (compare Ramalingam and Reps [12, page 21]): By initially retaining the disturbance set, new and old edges can be compared during reparsing. If a new edge $e'$ is different from the corresponding old edge $e$ (if this exists), then the dependants of $e$ are regarded as disturbed (potentially affected). If $e'$ is equivalent to $e$ in the sense of giving rise to the same derivative edges, then the dependants of $e$ are known not to be affected, and hence the reparsing process does not have to proceed beyond this point in the search space. In order to avoid extra computation, the disturbed edges should be visited in the order given by the dependency graph.

How can the points at which a change “dies out” be characterized? Since we are interested in characterizing the conditions under which two edges give rise to the same derivative edges, the contents part of an edge (that is, the right-hand side before the dot of the dotted rule) is irrelevant. For example, we want to say that the incremented edge $NP_{23}$ in figure 3 to be reparsing-equivalent with edge $NP_{34}$ in figure 2 although their dotted rules and parse trees are different; the dotted rule of the former is $NP \rightarrow Det N$. and that of the latter is $NP \rightarrow Det A N$. \footnote{To be able to localize changes as much as possible, we assume that the $q$ component of an edge $(v_i, v_j, X \rightarrow \alpha \cdot \beta \cdot q)$ only consists of a mother node and its immediate daughter nodes, and that the complete parse tree of an edge is obtained by recursively traversing its source edges.}

We can summarize this in the following definition:

Definition 10 (Reparsing-equivalent edges)

Assume a proposed edge $e$ and a disturbed edge $e' \in C$. We say that $e = (v_i, v_j, X \rightarrow \alpha \cdot \beta)$ and $e' = (v_i, v_j, Y \rightarrow \mu \cdot \nu)$ are equivalent from the point of view of reparsing if $i = s$, $j = t$, $X = Y$ and $\beta = \nu$. Furthermore, we then say that $e'$ is re-created.\footnote{This is a slight misnomer, since $e$ and $e'$ do not have to be equal.}

Inactive (combined or lexical) edges and predicted edges are special cases under this definition. In the former case, $\beta$ and $\nu$ are empty, and thus two inactive edges are reparsing-equivalent if $i = s$, $j = t$ and $X = Y$. In the latter case, $\alpha$ and $\mu$ are empty, and thus two predicted edges $e$ and $e'$ are reparsing-equivalent if $e = e'$.

5.2 The Algorithm

We now give a preliminary specification of a bounded incremental algorithm which handles one update cycle. (The essential differences compared to the unbounded algorithm are in the reparse and remove steps.)

Algorithm 2 (Incremental Chart Parsing)

Input: A configuration $\langle \tau, C, D \rangle$ and a change $\Delta_r$, which amounts to insertion or deletion of $m$ tokens $t_i, \ldots, t_{i+m}$.

Output: An updated configuration $\langle \tau', C', D' \rangle$.

Method: Do the following steps:

Modify the problem instance:
Insert or delete the modified tokens $\Delta_r$ into or from $\tau$. 

Prepare the chart: Do one of the following steps in the case of insertion and deletion, respectively:

**Insertion:** Renumber edges as follows: First, replace each edge \( \langle v_i, v_j, r, q \rangle \) where \( j \geq i \) and \( k \neq i \) with an edge \( \langle v_{j+m}, v_k, r, q \rangle \). Secondly, replace each edge \( \langle v_j, v_k, r, q \rangle \) where \( k > i \) with an edge \( \langle v_j, v_{k+m}, r, q \rangle \). Looping edges at the change vertex, which have the form \( \langle v_i, v_i, r, q \rangle \), are dealt with differently depending on whether their sources are to the left or right, which in turn depends on the rule-invocation strategy.

1. **Bottom-up case:** If the looping edge depends on an edge \( \langle v_k, v_j, r, q \rangle \) that is outgoing to the right (\( j > i \)), then it is replaced with an edge \( \langle v_{i+m}, v_{k+m}, r, q \rangle \) (in effect, it is moved).
2. **Top-down case:** If the looping edge depends on an edge \( \langle v_j, v_k, r, q \rangle \) that is incoming from the left or looping (\( k \leq i \)), then do nothing.

Finally, update the dependency relation \( D \) so that any edge \( \langle v_j, v_k, r, q \rangle \) such that \( j \leq i \) and \( k > i \) is made dependent on \( t_i \).

**Deletion:** Renumber edges as follows: First, replace each edge \( \langle v_j, v_k, r, q \rangle \) where \( j \geq i \) with an edge \( \langle v_{j-m}, v_k, r, q \rangle \). Then replace each edge \( \langle v_j, v_k, r, q \rangle \) where \( k > i \) with an edge \( \langle v_j, v_{k-m}, r, q \rangle \).

Reparse: In the case of an insertion, create a scanning task for each new token. Create a combination task for each active–inactive edge pair meeting at \( v_i \) in the case of deletion and at \( v_i \) and \( v_{i+m} \) in the case of insertion. Reparse such that the disturbed edges are visited in the order given by the dependency graph. Treat the disturbed edges as "sleeping", that is, they do not play any role in the parsing process as such. Whenever a new edge is proposed, check if an equivalent edge exists in the disturbance set according to definition 10. If so, install it, update \( D \) by letting the new edge inherit the dependencies from the old edge, and discontinue reparsing along this path (by refraining from adding any agenda items for the new edge). Mark the old edge as re-created.

Remove edges: Remove each edge that is in the disturbance set but not in the dependency set of any re-created edge.

5.3 Incremental Complexity

A detailed analysis of the complexity of algorithm 2 must await a more carefully worked out formulation. However, some preliminary observations can be made. To begin with, we adopt the same assumptions as with respect to the unbounded algorithm in section 4.3. As for the chart-preparation step, we assume that the renumbering is achieved implicitly by linking and unlinking vertices in a pointer structure and that we therefore do not have to actually update the entire chart when processing a change. In other words, we assume that this step can be made in \( O(\varepsilon) \) time. As for the reparsing step, given that only the incremented edges are added, and assuming that an edge comparison can be done in constant time, this step can be executed in \( O(|I|) \) time. Finally, given that the remove step only removes the affected edges, it can be executed in \( O(|A|) \) time. The algorithm as a whole then only requires \( O(\Delta_r + \varepsilon + |A| + |I|) = O(\varepsilon) \) time.

6 Discussion

The notion of boundedness is a recently developed criterion in the study of incremental computation and, according to Ramalingam and Reps [12, page 5], relatively few bounded incremental algorithms are currently known. In adapting this notion to natural-language processing, we have gone one step further than previous work in the sense that we define \( \varepsilon \) as a mathematical measure based on the result of the corresponding batch-mode computations. We thereby obtain a measure which is independent of any incremental algorithms for handling updates.

Boundedness is a strong criterion in the sense that only algorithms that are restricted to processing the region where the input or output changes qualify. As pointed out by Alpern et al. [1, page 32, 42 f.] and Ramalingam and Reps [12, page 39], it is important to recognize that an incremental algorithm with poor worst-case incremental behaviour may still be practical. Poor incremental behaviour means that the algorithm does not respond quickly to (some) small changes. However, it may still perform better than discarding the old solution and invoking a batch-mode algorithm. Even if the algorithm is unbounded, it may have a lower time bound in \( |P| + |S| \) (the size of the entire problem instance and solution) than the batch-mode algorithm. As a case in point, the unbounded algorithm in section 4 is clearly preferable to the batch-mode algorithm for the purpose of incremental update.

The fact that the incremental algorithms are linearly bounded (optimal) in the case of LR-incremental chart parsing means that they can be
applied to the batch-mode problem without changing the asymptotic behaviour. This should also work well in practice, since the cost of just maintaining dependencies only requires a small constant. This is notable, since the best batch-mode algorithm often will have superior asymptotic behaviour to that obtained when an incremental algorithm is applied to the batch problem (Ramalingam and Reps [12, page 4]). What this in turn means is that incremental chart parsing can be seen as a general utility in the sense of being efficiently applicable to both batch-mode and incremental problems.

There are several ways in which to carry the work reported here further. Obviously, another pass is needed at specifying and analysing algorithm 2, and it should also be implemented. A more general question is how the notion of bounded incrementality can be made more fine-grained in order to better reflect the granularity of actual grammars. In particular, can it be usefully translated to the level of feature structures under a unification grammar? Clearly, this would allow for much more thorough re-use of information than is possible when updates are performed only at the chart-edge level. Granted, fine-grained algorithms are not always more efficient than coarse-grained ones, but at least to find suitable trade-offs between fineness of grain of the update on the one hand and complexity of overhead on the other, translating the notions developed in this paper to the feature-structure level seems worthwhile.

A result related to this idea is provided by Alpern et al. [1], who show an exponential lower bound, namely \( \Omega(2^4) \), for the problem of handling arbitrary update of a circuit, a directed acyclic graph with values computed at the nodes.

7 Conclusions

Highly interactive natural-language applications require incremental parsing: although exhaustive re-analysis of the input can be fast enough for small problems, incremental algorithms are ultimately necessary in order to cope with larger problems. Ideally, the time that an incremental algorithm uses for processing a change should be a polynomial function of the size of the change rather than the size of the entire current input. To this end, we have adapted and applied a notion of bounded incremental computation to a chart-based parsing framework. Specifically, we have given a definition of incremental change which captures what must be done in response to a user change and which is independent of particular algorithms for incremental update; we have shown that the algorithm originally introduced by Wirén [15] is unbounded relative to this measure; and, as a way towards improving upon this, we have have given a preliminary specification of a polynomially (possibly linearly) bounded algorithm. In sum, there are at least two advantages of bounded incrementality. First, it provides a guarantee that the next update state is never more than a certain (bounded) computation away from the current state. This makes the system more predictable and also more natural in the sense that a small change is known to require less work than a big change. Secondly, in comparing the efficiency of different incremental algorithms, boundedness is a more sensitive probe than traditional worst-case analysis as a function of the size of the entire input. We therefore expect that the notion of boundedness will be of great value in developing enhanced models for incremental natural-language processing.

Acknowledgements

I would like to thank Ralph Rönnquist for e-mail discussions on the material presented here, Gregor Erbach for comments on a draft of the paper, and Peter Fritzson for originally alerting my attention to Ramalingam and Reps' paper. This research has been funded by the German Science Foundation (DFG) through the Sonderforschungsbereich 314, project N3 (BiLD).

References


ROBUST PARSING AND GRAMMAR CHECKING
OF FREE WORD ORDER LANGUAGES

Vladislav Kubon, Martin Plátek
Faculty of Mathematics and Physics,
Charles University
Malostranské nám. 25
CZ - 118 00 Praha 1
Czech Republic
e-mail: vkubon@CSPGUK11.BITNET,
platek@CSPGUK11.BITNET

ABSTRACT

The paper describes the technology being developed in the frame of a EC Joint Research Project PECO 2824 "Language Technology for Slavic Languages" - LATESLAV. The main objective of this project, which is carried simultaneously at universities in Saarbrücken, Prague, Sofia and Barcelona, is to develop a technology for creating "grammar based" grammar checkers for free word order languages.

The grammar checker is based on a theory of so-called deletion (erasion) automata. A new grammar formalism Deletion Automata Based Grammar (DABG) is being developed as a formal device that will control the processing. It allows to formulate the grammar rules not only in a classical "positive" way (rules for syntactically correct constructions), but also in a "negative" way (expectations of errors of certain kind). The parser tries to apply nondeterministically "positive" rules first. If it fails, the control is passed over to the "negative" part, which tries to identify an error in the input text and to correct it by means of deletion of one (or more) elements from the input. When the negative part is finished, it returns the control to the "positive" part again. Each time when the negative part performs its task, it writes a remark about the error into the tree of computation.

We discuss the logical specifications for both the grammar checker and a robust parser of free word order languages and the differences between them.

1. INTRODUCTION

The main goal of this paper is to show the difference in methods used for grammar checking and robust parsing, as they are being developed in the frame of an EC founded project LATESLAV - Language Technology for Slavic Languages (PECO 2824).

The languages under consideration in the project - Czech and Bulgarian - are both free word order languages, therefore it is not sufficient to use only simple pattern based methods for error checking. The stress is on grammar-based methods, which are much closer to parsing than pattern-based methods. We would like to show in this paper how the methods used for the error checking may be slightly modified and extended into methods for a robust parsing of relevant languages.

The goals and the means of robust syntactic parsing and grammar checking are to a great extent very similar, but nevertheless there are differences not only in goals, but also in the methods used.

The main goal of a grammar checker is to provide a user with suggestions as clear and unambiguous as possible to reduce the number of syntactic (stylistic) errors in the input text.

As to the robust syntactic parser, its goal is to create syntactic structures of the input sentence even in the case when the input sentence is grammatically incorrect. If the input sentence contains error(s), the resulting syntactic structure is in fact only similar to the structure of an input sentence - it is a structure of corrected input sentence. Even in this case we would like to distinguish between correct and incorrect parts of the input and the parts the correctness of which we are hardly able to say anything about.

This paper contains logical specifications, which are common for both purposes. We will point out the differences between grammar checking and robust parsing at relevant places of the paper.

It is necessary to stress that we are dealing with a surface syntactic analysis. Therefore also the errors which are taken into consideration are the surface syntax errors. Our system for identification
and localization of (surface) syntactic errors will consist of two basic modules - the module of lexical
analysis and the module of surface syntax checking. We will describe the second module, which is more
interesting from the point of view of this paper. Our approach to the problems of grammar checking and
robust parsing is based on dependency syntax.

2. ERROR CHECKING AUTOMATON

The core module of our system is an Error Checking Module. It recognizes grammatical
correctness of the given portion of text, or, in other words, it distinguishes the grammatically correct and
grammatically incorrect subsequences (not necessarily continuous) of lexical elements in the
input text.

The input of the Error Checking Module (ECM) consists of the results of the morphological
and lexical analysis. The exact form of the input elements is thoroughly described in [1]. For the
purpose of this paper it is only necessary to say that the data, representing one lexical element, are
contained in one complex feature structure. The attributes are divided into two groups, input and
output attributes. The ECM will try to reduce the input sequence by means of deleting symbols. The
deleted symbols are stored. They create the history of computation.

The whole process is nondeterministic - if there are more variants how to delete the symbols, all of
them are sooner or later taken into account.

A suitable device for this kind of processing the
input sentence is a nondeterministic list automaton
(NLA). In our work we are going to use its slightly
modified version, called Error Checking
Automaton (ECA). The storage of ECA is a
two-level, in each level two-way linear list composed of
data items (fields, cells). In the list there are two
Distinguished items: one at the leftmost end and
the other at the rightmost end of the list. These items
are called sentinels.

The first level represents the input and the
working storage of ECA, the other one ECA's
output. ECA has a finite state control unit with one
head moving on a linear (doubly linked) list of
items. In each moment the head is connected with
one particular cell of the list ("the head looks at
the visited field"). The actions of the working head are
delimit by the following five types of basic
operations that the head may perform on the list:
MOVE, DELETE, INSERT, SIGN and RESTART.
The operation of the type MOVE ensures the bi-di-
rectional motion of the head on the list. The
DELETE operation deletes the input field in the
input level, but it leaves the deleted field in the
output level. The INSERT operation adds a field
with a symbol to the output level, more exactly: to
the close neighborhood of the visited field. The
SIGN operation adds a special marker or sign
(pebble) to the chosen place of the list or removes
the pebble from the chosen place of the list. The
RESTART operation transfers ECA from the
current configuration to the (re)start configuration.
The (re)start configuration of ECA is any such
configuration in which ECA is in an initial
(unique) state and there is no sign placed in the list.

The processing of the ECA is controlled by
rules, written in a formalism called DABG, which
was developed especially for the project
LATESLAV. It is described in detail in [1]. The
theoretical background for erasing automata of
the above described kind can be found in [2] and
[3].

When the processing of ECA finished, the
input level contains only the sentinels.
The ECA autonation is logically composed of
the following three components:
(a) list automaton M;
(b) list automaton N;
(c) control module C.

2.1. The automaton M
The automaton works in cycles between (re)starts.
The function of the automaton M is (during one
cycle) to apply one rule of the control grammar to
the input. That means to decide (nondeterministically) which finite subsequence of
the text in the input level of the list is correct
(according to one rule of the control grammar), and
to delete this part from the input level. After that it
continues the work in the next cycle.

It means that if the input text is error free, the
automaton M will gradually repeat cycles and
delete (in at least one branch of its tree of
computations) all the input elements (except for the
sentinels).

The automaton M will consider the input
sequence correct, if the computation of M will
finish by deleting all the items (except for the
sentinels) from the input level of the list.

Two basic principles how to create rules for the
automaton M follow from the above mentioned
nature of the behavior of the automaton M:
1. M will contain only those (deleting) rules, for which there exists a sentence which will remain syntactically correct after the application of the rule.

2. There may not be a syntactically incorrect sequence of words in the input sequence which would be changed to a syntactically correct sequence of words by means of the application of a rule from M.

**Strong rules (S-rules)** are the rules, which meet the following requirement:

3. Any application of a rule will keep both correctness and incorrectness of the input sequence.

Clearly the S-rules meet also the requirements 1) and 2).

The subautomaton of M, which uses S-rules only, is called M_S.

One cycle (one compound step) of an automaton M (or M_S) looks from the technical point of view as follows:

In the first step the automaton searches through the input level for the place where to put the pebble. Then it checks whether there is a possibility to use one of the deleting rules in the close neighborhood of the pebble. In the positive case the deleting rule is applied, the pebble is removed and M (or M_S) will return to the initial configuration. The function of the pebble is to allow M to move away from the point, where the deletion would be applied, to check whether the constraints for the application of a rule are met and to come back to the marked spot.

### 2.2. The automaton N

The task of the automaton N is to find an error in the input text, to make an error mark and to remove the error causing word from the input. In one compound step the automaton N performs (nondeterministically) the following actions:

First, similarly as the automaton M, N locates the place where to put the pebble in the input level. Then it checks whether there is an error in the close neighborhood of the pebble. In the positive case it will mark the occurrence of the error to the output level of the list and correct the input level of the list by deleting some items from the environment of the pebble according to a particular rule of the control grammar of the automaton N.

**Definition:** The **limited error** is a string z, for which there are no y, w such that the string yzw is not a (grammatically) correct sentence (of a given language L). If z can be written in a form of

\[ z = v_0 u_1 v_1 u_2 v_2 \ldots \ u_k v_k \]

and there are also strings s, r such that

\[ su_1 u_2 \ldots \ u_k r \]

is a grammatically correct sentence, we call u = u_1 u_2 \ldots \ u_k the correction of z. The correction may be ambiguous, but there is always a trivial correction - performed by deleting the whole z.

The limited error is a type of error, which may be located in the given input string (we may delimit the region, which is affected by a particular error).

**Comment:** A **minimal limited error** is a string z, which is a limited error and there is no possibility how to shorten z from neither side and to preserve the property of being a limited error for z.

It is necessary to try to build the automaton N from as many rules as possible, which perform the corrections of minimal limited errors.

This is especially important in case that we are trying to build a grammar checker. In that case we need to distinguish between real errors and the errors brought in during the process of deleting the words from the input. As to the robust parsing, it is not necessary to make this distinction there - the main task of a robust parsing is to guarantee error free subsequences of the input and to provide a user with an 'elegant' error recovery and the distinction between real errors and other types of errors is not relevant.

For the purpose of grammar checking it is useful to distinguish between two types of minimal limited errors:

- **a)** peripheral (formal) - containing both sentinels
- **b)** real - all other minimal limited errors.

The subautomaton of the automaton N, which works with the rules performing the corrections of minimal limited errors only, will be called N_S.

### 2.3. The control module C

The C module which is also of the NLA type is a control submodule of the entire ECM module. At the beginning of the operation of ECM, the C module will call the automaton M, which will work as long as it is possible to shorten the input level list by deleting its items. If there are more possibilities at a certain moment of computation, the automaton M will choose only one of them, it will store the other into the stack and it will try to
apply another rule to the modified input. That means that M (as well as N) is searching the tree of possible computations depth-first. As soon as the automaton M cannot shorten the list in the input level any more and the input level does not contain only the sentencels, C will call the module N. This automaton removes one error from the input level, makes an error mark and transfers control back to the C module, which invokes the automaton M anew. Thus, the submodule C will repeatedly invoke the automata M and N (switching between them) as long as they are able to shorten the sequence of input elements.

3. ERROR CHECKING AND ROBUST PARSING

Clearly there may be some input sentences, which contain either syntactically correct subsequences of words, which cannot be handled by the rules of M, or syntactic errors, which are not covered by the rules of N. In this case both automata stop and the input level still contains a subsequence of input elements. Its contingent final emptying will be ensured by the C module that will note this emptying to the output level. Then the C module will transfer control to the next phase of the whole system.

The behavior of the C module will slightly differ at this point according to the type of activity it is performing. If the C module is a part of a grammar checker, it will make an error mark about the whole undeletable subsequence of the input. In case that the C module works in connection with robust parser, it should try to get more information about the remaining subsequence. It will try to remove some word(s) from the subsequence in order to find parts of this subsequence, which may be parsed correctly. In the worst possible case the 'robust' C will remove word by word, all the remaining subsequence from the input.

At this point it is necessary to clarify, what kind of output structure will be built by ECM. As we have already mentioned, our approach to the problem is oriented towards dependency syntax.

The use of DABG for creating the control grammars for M and N is not limited to dependency based approach only. Both the data structures (feature-structure based) and the DABG formalism allow to formulate the rules that will create the constituent structure of the sentence at the output level.

All the rules of control grammar for M and N delete the depending word from the input and put it into the output attribute of the governing word. At the end of the process there is a tree, which contains the information about all the words from the input, about the order of deletions and also all error marks made by the automaton N.

The switching between M and N will guarantee that any possible path in the tree of computation will result in a complete structure of a deletion tree.

The current best deletion tree is then compared to any new deletion tree. If the new tree is better (e.g. contain smaller number of errors or contains errors with a smaller weight), it is then stored for further use as the new current best result.

At the end of the whole process we have the best result (or a set of best results, e.g. when there are more possibilities how to parse the sentence), which contains all relevant information about errors present in the input sentence.

This approach will serve very well for the purpose of robust syntactic parsing. For grammar checking, it is better to use a more careful approach. If there are non-empty (the bigger the better) automata M\textsubscript{S} and N\textsubscript{S} for a given language, it is better to provide the user preferably with the error messages about errors that were not brought in by the deletion of some word in the input.

At the current stage of the work we distinguish the following two types of real errors:

a) If some peripheral rule of N\textsubscript{S} was applied immediately after the automaton M stopped for the first time on a certain path in the tree of computation of a particular input sentence.

b) If there were only the rules of M\textsubscript{S} and N\textsubscript{S} applied to a particular path in a tree of computation.

Clearly the tree with error marks of the type a) will be among the best results of the C for any reasonable comparison of results. To achieve a satisfying quality of error messages for the user, we have to assign to the errors of a type b) slightly smaller weight than to similar errors that were detected by other rules then the rules of M\textsubscript{S} and N\textsubscript{S}. It is of course possible to use other strategies how to provide the user with correct error messages.

In the previous paragraphs we have shown, that there are stronger requirements to grammar checking than to robust parsing. In the following comment it is the other way round.
Comment: For the purpose of grammar checking it is possible to choose the deleting rules of the automaton M according to the constraints 1, 2 or 3. That means that it is possible to use also the rules, which transform the meaning of the sentence.

In case of robust parsing it is necessary to create the control grammar of the automaton M from those deleting rules, which transform the sentence into the sentence with more general meaning. In this case we are not allowed to transform the meaning.

The main idea of the comment is, that if we are writing the grammar for a grammar checker, we should preserve only one invariant during the process of modifications (deletions) of the input sentence, and that it is an invariant expressed in constraints 1, 2 and 3. There are no other requirements on the rules of the control grammar M.

In case of robust parsing it is not so simple. The reason is that we not only need to identify the correct and incorrect parts of the input sentence, but also want to build a structure, representing the syntax of a particular sentence. This is a different point of view, which implies stronger requirements to the nature of rules of a control grammar. The more freedom is allowed for the grammar rules, the less realistic is the resulting syntactical structure.

4. CONCLUSION

In the previous paragraphs we have sketched the basic theoretical differences between grammar checker and robust parser of free word order languages, as they are viewed from the perspective of the project PECO 2824 - LATESLAV. The main goal of the project is to develop a methodology of solving the problem of "grammar based" grammar checking of relevant languages. We hope that the ideas presented in the paper may help us to achieve this goal successfully.

REFERENCES


[2] Oliva K., Plätzek M., Seznamové automaty a typy popisu povrchové syntaxe (List Automata and
the Types of Description of Surface Syntax). In: SOFSEM'90, Janské Lázně 1990.


The work described in this paper was performed in the frame of the Joint Research Project PECO 2824 - LATESLAV.
PUNCTUATION AND PARSING OF REAL-WORLD TEXTS

(Mr.) V. Srinivasan
School of Applied Linguistics and Cultural Studies, University of Mainz
76726 Germersheim
Federal Republic of Germany
email: srini@coli.uni-sb.de

ABSTRACT
Real-world texts currently require extensive pre-processing before being subjected to any kind of language processing by computers. Such pre-processing primarily involves the visual and graphematic aspects of written texts, both of which are potentially relevant for parsing. While parsing entries for lexicon insertion and lookup needs to strip punctuation, it can be passed on to the parser, whereas current approaches to parsing treat punctuation marks only as delimiters. Particularly for free word-order languages (German as against English), punctuation appears to be an essential (and neglected) aspect of syntax and can be used as a reliable indicator of certain syntactic structures while parsing.

1. INTRODUCTION
This paper addresses several issues concerned with the pre-processing of real-world texts, e.g. texts that are created by writers and translators working with a word processor or texts scanned in using an OCR system. To be commercially viable, any kind of language processing system — parsers, machine (aided) translation systems, automatic lexicon updating systems, spelling checkers etc. — will have to accept real world texts as input (HOBBS and others 1992). As it stands, all existing NLP systems require extensive pre-processing of the input text; in the worst case, these texts have to be manually rewritten as suggested by (FUTRELLE and others 1991).

The problems involved in pre-processing are multi-layered: at a very basic level, difficulties in pre-processing arise due to compatibility problems between machine types, file formats, character sets, type faces, fonts etc. The number of hacks, patches, kluge innovations and ad hoc solutions used particularly in the European context is itself an entertaining topic for a paper and will not be addressed here. The next level of difficulty arises from the fact that real world texts are often made up of running text and graphics in the widest sense of the word. Not seldom, these graphics themselves contain text that require processing. Only recently has this problem been tackled and has been discussed in detail in (BAIRD and others 1992). From the standpoint of computational linguistics and parsing technology however, the most interesting problems concerning real world texts are located at the third and bottom-most layer and are related to the nature of written language itself, arising from what Max Silberztein refers to as "formalization of not always well-defined habits of writing" (GROSS and PERRIN 1989:93).

Many of the problems discussed here were encountered during the development of TERMBASE, a multilingual extended phrasal lexicon for translators (SRINIVASAN 1993). This lexicon is the first stage of a comprehensive toolkit for a group of translators working in a corporate environment (PAILLET 1988). Currently this extended phrasal lexicon is used by translators to store and look up all kinds of language data: individual words, terms, phrases, sentences, even whole standard paragraphs. In the next phase, they will add tree representations of the terms in the database, in much the manner suggested by Neumann (NEUMANN 1994). In the final phase, a parser will be added in order to automate at least some of the translators tasks.

While written and spoken language share elements from a common repertoire, they have both very specific and individual characteristics, which need to be looked at in their own right. To be sure, speech can be expressed (transcribed) in writing and written language can be spoken (quoted). In the normal case however, people use language differently when they speak and write respectively.
Parsing technology must therefore adopt different strategies to analyze and process input from these different genres. Parsing technology can exploit certain regularities which occur in written language, and are specific to it.

2. VISUAL ASPECTS OF TEXTS

To begin with, real-world texts are physical, two dimensional entities in which textual information is visually organized. That this seemingly trivial fact has non-trivial linguistic significance was first pointed out by Geoffrey Nunberg (NUNBERG 1990). In general, it can be said that all real-world texts will contain visual reading clues of one kind or the other (be it merely a page number) and that every such clue has a "meta-lexical" function, in the sense that they will comment on the text. Any system that provides input to a parser has to first identify these visually organized elements such as text and paragraph titles, headers & footers, (This is the title of the text/paragraph), dates and addresses (This text was written on ... by ...), page numbers etc. The common characteristic linking these clues and setting them apart from the "body" of the text is the fact that they are exclusively visual — typeface and font alterations, position (top right etc...), white space setting them apart from the body of the text etc. The stringency of rules governing these visual markers can be appreciated by the simple fact that even human readers would be confounded without them.

The identification of such elements can be aided by other syntactic features. For example, both in English and German technical texts, titles will often not be syntactically well-formed sentences. They will omit articles wherever possible and will make sparing use of punctuation. They will not as a rule be terminated by a punctuation mark with the possible exception of a closing single or double quotes or an exclamation mark. Whereas English titles adopt a system of using uppercase initial letters for any word longer than four letters, German titles tend to be orthographically well-formed.

To sum up, a pre-processor will have to incorporate image processing techniques to identify and label these elements, despite the fact that such elements are made up of only of alphanumeric characters (BAYER and others 1992:32). Following this, it will have to use heuristically stated syntactic rules to confirm the initial identification. Needless to say, these rules will have to be specific to the language and to the type of the document in question. Most research findings in this area relate to scanned documents — see section 1 in (BAIRD and others 1992). In the case of documents written using a word processor, generating an image of the page (as it is done in the page preview function) and then processing it to identify the relevant portions ought to be within the scope of existing technology. Of particular interest for texts in German are findings of the kind reported by A. Dangel and F. Hönes at the DFKI in Kaiserslautern, where they are trying to automatically extract information from letters based on their visual layout.

3. TOKENIZING

Tokenizing the body of the text can be described as the transformation of two-dimensional text into a linear sequence of elements such that each element belongs to a predefined category. Most programming languages offer a facility to tokenize an input string - the catch is that the tokens that are extracted by this facility are those of the programming language that is used, not of natural language, due to which application developers have to write their own tokenizer. The second reason to do this is because different applications require different kinds of tokenization. Lexicon entries need to be stripped of punctuation, while parsers may even benefit if punctuation is retained (see section 3.2). Also, since different languages use different writing conventions, the tokenizer has to be language sensitive.

3.1 Tokenizing for lexica

Lexicon entries are tokenized primarily for indexing purposes. The entries are stripped of all punctuation marks before being subjected to morphological analysis (SPROAT 1992:5-7). The actual entries are then made accessible by means of a set of keys that are thus obtained. The stripping/segmenting algorithm that is used is fairly straightforward; the following procedure is adopted for each block of contiguous text:

- do while not end of block
- start
- read towards the right until a space, tab or a line feed is encountered
- recursively examine the read unit from the left and from the right for punctuation
marks until a letter or a digit is encountered stop
- list the punctuation marks and the remainder ("the word") as individual tokens

A similar algorithm has been suggested by Booth in (GARSIDE and others 1987:97-109). The individual "words" isolated by means of this procedure come surprisingly close to our intuitive visual perception of a word, however controversial this term may be linguistically. While this procedure strips away punctuation marks from the extremities, additional procedures are defined for the following cases:

1. Possessive singulars:
The unit's cover etc.
A normal feature in English. Known as the "Saxonian genitive", this is quite prevalent in German although spurned by linguists. For both English and German, everything to the right of the apostrophe is stripped away.

2. Range indicators:
Coi a 2-4" long strip of insulation ...
From 1990-1993, there was long...
The inch symbol would have been removed earlier. If the token consists only of numbers to the right and left of the hyphen, the entire token is deleted, since numbers need not be indexed.

3. Names of chemical compounds
2,6-Diaminohexanoic acid
(FUTRELLE's example)
A token embedding a punctuation mark but containing only digits and letters is left untouched.

4. Embedded parentheses
(amin-icids, (dis)colored
Embedded parentheses are simply deleted. Embedded hyphens are retained.

5. Hyphens
Both in English and German, the hyphen proves to be the most difficult character to process. In (poorly written) English, a hyphen is often used to build ad-hoc compounds or to disambiguate a context as in
C language-processing functions
as against
C-language processing functions

TERMBASE features other algorithms to treat compounds in which hyphens occur. Only the relevant ones have been discussed here. For a more exhaustive and contrastive treatment of compounds in English and German see (RACKOW and others 1992), (RACKOW 1992).

A line feed can generally be replaced by a space character (see pouring rules in NUNBERG 1990:73f). However when the last character of a line is a hyphen, the first "word" of the next line can either be interpreted as a continuation of the previous word or as the unsplit nth element of a hyphenated compound as in
...to pro-<line feed>
cess these entries...
and
...language-<line feed>
processing...

A hyphen is often (mis)used as a substitute for the en-dash. When this occurs at a line break, it will be preceded by a space and can be detected.

The situation is slightly better in German thanks to the fact that hyphenated compounds are fairly rare in German except in proper names. Then again, German uses the hyphen as a "morpheme placeholder" in multiword lexemes as in
Im- und Export
Fachbücher bzw. -zeitschriften
Berufs-, Real- und Hochschulen

To decide whether a hyphen located at the end of a line is a word break or a "placeholder", it is sufficient to recursively examine the next token until the first letter group is encountered if this group is "und", "bzw", or a letter group which in turn is followed by a hyphen, then the hyphen at the line break is a placeholder. In all other cases it is a word break provided the following group is not capitalized, or else a proper name. These compounds are indexed as shown below

Im- und Export as Im<export> und Export
Berufs-, Real- und Hochschulen as
Berufs<oschulen>, Real<oschulen> und Hochschulen

suggesting that the hyphen is a placeholder for a non-null trailing substring of the next element that is not und or bzw. and is not followed by a hyphen.

Fachbücher bzw.-zeitschriften as Fachbücher bzw. <Fachbücher>-zeitschriften

suggesting that the hyphen is a placeholder for a non-null leading substring of the previous element
that is not und or bwz. and is not preceded by a hyphen.

Both in German and in English, abbreviations are stripped of their trailing period:
  etc. becomes etc.
  A.C.L. becomes A.C.L. etc.

3.2 Punctuation and parsing
From existing literature on parsing, it would appear that punctuation plays a predominantly defining role in parsing. The only punctuation mark that is mentioned at any length in parsing literature is the period, and that too in its role as an unreliable end-of-sentence marker, since it is used in a number of other contexts.

While the focus on linguistic forms common both to spoken and to written language is certainly warranted, written language, as Nunberg convincingly argues, does exhibit interdependent features, notably punctuation, that merit study as a system in its own right.

In particular, Nunberg argues, a closer investigation of the scheme of “figural representation” will no longer support the persistent claims of punctuation being a poor attempt at recording intonational patterns. This certainly appears all the more so for German, which features at least some syntactically non-contingent rules for the placement of punctuation marks.

While punctuation (or lack of it) would alter the semantic content of sentences, or at worst cause ambiguity without however violating conditions of well-formedness in English, a German sentence will be rendered graphematically and syntactically ill-formed by a missing or a superfluous comma. This is not to imply that syntactically well-formed German sentences are free of ambiguity, quite the contrary:

The men who could swim, managed to save themselves.
The men, who could swim, managed to save themselves.
Those who could swim managed to save themselves.

Die Menschen, die schwimmen konnten.
Sie konnten den Menschen die

Sich retten konnten die Menschen, die schwimmen konnten.

The same applies to sentences featuring relative clauses, conditionals etc. From a universal grammatical standpoint, it appears that languages featuring a relatively freeword-order have a correspondingly larger set of syntax-motivated punctuation rules. Besides German, this seems to be true for some other languages like Czech.

Note that in German, "variations of the theme" are achieved by syntactic reordering, while preserving the position of the comma, or to put it in other words, the position of the comma is unaffected by changes to the semantics of the sentence; it is purely determined by syntactic constraints.

Any standard reference work on German grammar will give hard and fast syntactic rules for the placement of punctuation marks, see for example (ENGEL 1988). Consequently, punctuation marks can be used to define well-formedness conditions and can be relied upon to provide at least some clues to detect syntax patterns while parsing. It is therefore surprising, that even recent attempts to construct a catalogue of data to exemplify the syntactic patterns of German e.g. (NERBONNE and others 1993) make no mention of punctuation, an integral aspect of German syntax.

4 Concluding Remarks
The visual aspects of real-world texts have to be examined in greater detail.

The role of punctuation needs to be examined more closely for free word-order languages, in order to detect the extent to which they are syntactically motivated. This information can then be incorporated into the parsers for those languages that take written language as input.

For application developers, it would be advantageous to have a Standard Parser Input Format (SPIF perhaps?), so that all applications can have a standard interface to a parsing module. For the parsing community such a Standard would serve as a concrete starting point for syntactic analyses.
Acknowledgements

I am grateful to Prof. M. Pinkal and the staff at the Dept. of Computational Linguistics, University of Saarbrücken for providing me with research facilities. Special thanks also to Mats Wirén for drawing my attention to scant publications in the field of punctuation, to Brigitte Kreßn for valuable suggestions.

REFERENCES


ROBUST GLR PARSING FOR GRAMMATICAL SPACING CORRECTION

Theo Vosse
Experimental Psychology Unit, University of Leiden
P.O. Box 9555, 2300 RB Leiden, The Netherlands
+31 70 273407, vosse@ruisw.LeidenUniv.nl

ABSTRACT

Grammar-based spelling correction has been discussed before in Vosse (1991, 1992a, 1992b). These articles described the requirements of a grammar-based spelling corrector at a functional level and their implications for the system architecture. In this paper we will present the technical details concerning the robust parsing using Augmented Context-Free Grammars and its implementation in a Generalised LR-parser. We will also see how the results of such a parser can be used in spelling correction and discuss some aspects of the parser’s performance.

1. SENTENCE ANALYSIS IN SPELLING CORRECTION

Grammar-based spelling correction boils down to the question of how to use syntactic information in spelling correction. If we have a parser that only informs us that the sentence is ungrammatical, how do we find out what caused the ungrammaticality and how can we correct the problem? Thus, the parser should not only be able to analyse grammatical sentences, but also certain classes of ungrammatical sentences and present their (most probable) analysis together with an error indication. To determine the type of error, the parser must relax the constraints the grammar imposes on the sentence (cf. Kwasny and Sondheimer (1981), Jensen and Heidorn (1982), Carbonell and Hayes (1983), Weischedel (1983), Richardson (1985), Mellish (1989)). E.g., by allowing disagreement between subject and finite verb, an erroneous sentence such as "he go home" can be analyzed. A parsing mechanism that can analyse incorrect sentences as well as correct sentences without special adaptation of the grammar is presented in section 4.4.

After the parser has determined the analysis with the least number of violations of the grammatical rules, a syntactic corrector can inspect this analysis and propose corrections. In doing so, it should keep the alternatives as close to the original input as possible. E.g., the proper correction of "he go home" is he goes home and not the also grammatical Any

vulture getting caught sitting on my snowman gets clobbered. Because the meaning of the sentence is unknown, the general idea of the method applied here is that the stem of (content) words should not be changed and that word order should be changed as little as possible in order to maintain the meaning of the sentence. This requirement limits correction to changing the inflection, or finding an alternative to function words, such as articles, and occasionally deleting or inserting these. In most cases, there is some sort of disagreement between the features of two words, one of which can be seen as the central or dominating part of the phrase, also known as the head of the phrase, in which the two words occur. In principle, the head is not changed. E.g., in case of a mismatch between determiner and noun, as in deze huis (Eng.: these house), the determiner will be changed, giving dit huis (Eng.: this house), instead of the noun, giving dieze huizen (Eng.: these houses). Only in the case of subject-verb disagreement, the finite verb, which is commonly regarded as the head of a sentence, is changed. This rule guarantees that the meaning of the corrected sentence remains as closely as possible to the original intention.

2. DESCRIPTION OF GRAMMATICAL SENTENCES: ACFG

Augmented Context-Free Grammar (ACFG) is not the linguistically most elegant way to describe generalisations, but it captures them much better than plain CFG. Typical use of CFG is

\[ S \rightarrow NP_{nom\_sg\_1} \rightarrow VP_{sg\_1} \]
\[ S \rightarrow NP_{nom\_pl\_1} \rightarrow VP_{pl\_1} \]
\[ S \rightarrow NP_{nom\_sg\_2} \rightarrow VP_{sg\_2} \]
\[ S \rightarrow NP_{nom\_pl\_2} \rightarrow VP_{pl\_2} \]
\[ S \rightarrow NP_{nom\_sg\_3} \rightarrow VP_{sg\_3} \]
\[ S \rightarrow NP_{nom\_pl\_3} \rightarrow VP_{pl\_3} \]

In ACFG, every symbol can be associated with a number of features, such as number and person. In the rules, it is possible to put restrictions on the fea-
tutes of a certain symbol, e.g. demanding that the subject noun phrase be nominative, to use variables to cohere certain features of one or more symbols to be equal, and to determine the features of the left hand side non-terminal. A typical ACFG rule, corresponding to the previous CFG fragment, is

$$S \rightarrow \text{NP}(\text{nom Number Person}) \text{ VP(} \text{Number Person})$$

This rule states that the first feature of the NP has to have the value nom, and that NP's second feature must correspond to VP's first, and NP's third feature must correspond to VP's second, and the use of the variables Number and Person. This way of description is more compact and understandable than the CFG fragment.

More formally, an ACFG G is a 7-tuple \((N, T, P, S, F, V, C)\), where \(N\) is a set of non-terminal symbols, \(T\) is a set of terminals symbols (the categories, or parts of speech), \(P\) is a set of production rules, \(S \in N\) is the featureless start symbol, \(F\) is a set of feature symbols, \(V\) is a set of variables, and \(C:\{\text{NUT}\} \rightarrow N\), is a function assigning the number of features to each symbol. If the set \(A\), the alphabet of the production rules, is defined as

$$A = \{X, \{f_1, \ldots, f_{\text{C}(X)}\} \}$$

where \(\text{C}(X)\) is the number of features of \(X\) then

$$P \subseteq 2^A$$

Note that each symbol has a variable and a set of feature symbols attached to it. The symbol 1 represents the "don't care" variable. The set of features is the restriction on the allowed values for each feature.

The language \(L(G) = \{ \sigma \in P \mid S \Rightarrow^* \sigma \}\) of the rewrite relation \(\Rightarrow^*\) is defined as

$$\text{c}A\{f_1, \ldots, f_{\text{C}(A)}\}B \Rightarrow$$

$$\text{cA}\{f_1, \ldots, f_{\text{C}(A)}\}A_n\{f_{1}, \ldots, f_{\text{C}(A)}\}B$$

provided \(P\) contains a production rule

$$A\{v_0, f_{0,1}, \ldots, v_0, f_{0,\text{C}(A)}\} \Rightarrow$$

$$A\{v_1, f_{1,1}, \ldots, v_1, f_{1,\text{C}(A)}\} \ldots$$

$$A_n\{v_{n,1}, f_{n,1}, \ldots, v_{n,\text{C}(A)}\}$$

and there is a substitution that consistently copies all features from the left-hand side to the right-hand side and keeps values for identical variables compatible (if neither variables is 1). In this scheme, a symbol has a number of sets of features values instead of a number of single feature values, which makes grammar writing and parsing simpler. An example is the rule NP\{L{\small (L, (nom, acc, dat), (Number, F), (L, (3))} → Noun((Number, F))\}, which says that a noun can form a nominative, dative or accusative noun phrase, where the noun phrase has the same number as the noun and is always in third person.

Note that the rewriting may halt, unlike the rewriting of CFGs, if a substitution is chosen, which substitutes feature values that do not match any left-hand side. It is also customary, and in practice indispensable, to use a lexicon during parsing and have categories as terminal symbols instead of having characters as terminal symbols and put the words in the grammar. Then, if a sentence is to be generated, rewriting may also halt if a rewriting requires features that cannot be found in the lexicon.

### 3. TYPES OF GRAMMATICAL ERRORS

Now the formal side has been taken care of for the moment, the question of ungrammaticality returns: how can ungrammaticalities be discovered by means of a grammar? To answer this question, we must first know what kind of ungrammaticalities there are. What is the relation between an ungrammatical sentence and the corresponding grammatical one? If any? A more practical overview can be found in Vosse (1992a, 1992b), but more abstractly we can distinguish the following transformations.

- **Substitution.** Replacing a word or a non-terminal constituent with an incorrect alternative usually creates ungrammaticality. An example of the former would be *the well performance* instead of the *good performance*. Typing errors are another major cause of erroneous substitution.

- **Deletion.** Forgetting to write a word or an entire constituent can give rise to another form of ungrammaticality. The determiner is an often deleted category; and telegraphing—omitting the subject or the verb phrase in subordinate clauses—becomes more and more popular in Dutch, e.g. *Zou graag komen, maar afspraak* (Eng.: *Like to come, but appointment*).

- **Insertion.** Word repetition is the most frequent cause of superfluous words or constituents, especially when going over to the next line. Sometimes, auxiliary verbs are erroneously added.

- **Transposition.** Word or constituent transposition leading to incorrect order most often occurs in foreign language users. E.g., the order of nouns and their modifying adjectives in French may pose problems for native speakers of Dutch and English. The proper positioning of adverbs in English is also quite problematic for foreigners.

- **Feature mismatch.** Ungrammaticalities can be caused in ACFG analysis by non-corresponding features, e.g. subject-verb disagreement. Feature mismatch may be seen as a special case of substitution; in CFGs it is a real case of substitution, e.g. of the non-terminal VP \_\_3 by VP \_\_1. In what way, then, can ACFGs be used to cope with these ungrammaticalities? For a start, being an extension of a CFG, an ACFG can be used to find feature mismatches. We can say that an ACFG \(G\) is an extension of a CFG \(G'\) if \(G'\) has the same rules \(G\) has, but without the features and the variables. The language of \(G', L(G')\), contains the language of \(G, L(G)\). In particular, part (not necessarily all) of the sentences contained in \(L(G') - L(G)\) are sentences.
whose underlying derivational structure is identical to a correct string in \( L(G) \), but which have one or more feature mismatches. E.g., consider the sentence \( \sigma \in \mathcal{E}(G) \rightarrow L(G) \). Then, a context-free derivation \( S \Rightarrow S' \Rightarrow S'' \Rightarrow \sigma \) exists, where \( \rho_1, \ldots, \rho_n \) are rules in \( G \). There might also exist an ACFG derivation \( S \Rightarrow \rho_1 \Rightarrow \rho_2 \Rightarrow \sigma \), where \( \rho_1, \ldots, \rho_n \) are extensions of the rules \( \rho_1, \ldots, \rho_n \). We call \( \sigma \) a correction of \( \sigma' \).

Consequently, analysing a sentence using \( G' \) enables a language processing system to determine whether the sentence has a grammatical alternative, by applying the feature rules later in the parsing process or by allowing mismatches during analysis.

Unfortunately, a mechanism of comparable elegance cannot be given for the other types of errors, although general schemes to detect these structural errors exist (cf. Barnard and Holt (1982), Charles (1991)). These schemes, however, are mainly aimed at deterministic parsing and errors made in programming languages. They are also computationally expensive when used with ambiguous grammars and introduce very unlikely alternatives. Such mechanisms do not seem to be suited for the correction of structural errors, because the low rate of occurrence of structural errors cannot justify the expenses involved in accounting for such errors in general.

On the other hand, some structural errors do occur, e.g., missing or superfluous punctuation is a regular error. Such errors cause a breakdown in a parsing system, that has no built-in mechanism to deal with structural errors. To avoid this undesirable situation, ACFGs can be extended with error rules, i.e., rules that explicitly describe an expected structural error. E.g., rules that deal with a missing comma can be formulated as

\[
\text{requisite_comma} \rightarrow \lambda
\]

where \( \lambda \) is the empty string. If we ignore the features for the moment, we can define a context-free grammar with error rules as \( G = \langle N, T, P, S, E \rangle \) where each rule in \( P \) contains a possibly empty set of error messages from the set of error messages \( E \).

The language of \( G' \):

\[
L \{ (N, T, P, S, E) \} = \{ (\sigma, \Theta) | (S, \Theta) \Rightarrow^* (\alpha, \Theta) \}
\]

contains the strings plus the associated error messages and their position. The language of error-free sentences, \( \{ \sigma | (S, \Theta) \Rightarrow^* (\alpha, \Theta) \} \), is identical to the context-free language of the grammar without the error rules. This language may contain sentences that have other derivations with errors, because a sentence may be derivable in more than one way. Features can be added to this scheme as in section 2.

The combination of having both features and error rules in a grammar allows us to deal with most ungrammaticalities, assuming arbitrary structural errors do not occur. Therefore, ACFGs constitute a solid basis for robust parsing and error detection.

4. ANALYSIS

How can we use a dictionary and a grammar to analyse grammatically ill-formed sentences? In the previous section it was shown that ACFGs can be used to describe grammatical sentences, while also being able to account for certain classes of ungrammatical sentences. This section will show that incorporating ACFGs in existing parsing algorithms yields a system that not only analyses grammatical sentences, but is also able to analyse and correct large classes of ungrammatical sentences, improving spelling correction as a side effect.

4.1. Shift-reduce parsing with ACFGs

In order to be able to use the feature values, the shift-reduce algorithm must be adapted slightly, because shift-reduce parsing with features can be accomplished in the same way as shift-reduce parsing without features. If a sentence is grammatical according to an ACFG \( G \), it is also grammatical according to the CFG \( G' \) of which \( G \) is an extension. More formally, if a sentence \( \sigma \) can be derived using rules \( \rho_1 \ldots \rho_n \) of \( G' \) in a right-most derivation, then it is possible to derive a sentence \( \sigma' \) using rules \( \rho'_1 \ldots \rho'_n \) of \( G' \) in a right-most derivation, provided that \( \sigma \) is an extension of \( \sigma' \) and that each \( \rho_i \) is an extension of \( \rho'_i \). It follows that the parser \( eq \), the LR-table for \( G \) will have the same states and the same transition function as the parser \( eq \), the LR-table for \( G' \).

The adaptations to the shift-reduce algorithm are twofold. The shift step now includes copying the input symbol's features onto the stack. During the reduce step, the features on the stack must be combined and checked for equality; from these features, the rule's left-hand side features are computed. If there are unification errors, the reduce step fails, and the sentence is considered ungrammatical. Let us look at two examples, using the grammar fragment given above: he sleeps and he sleep. Figure 1a shows the consecutive stack configurations when parsing the former, and 1b when parsing the latter sentence.

In the pre-final step in figure 1a the reduction is made according to the first rule of the example grammar: \( S(\text{Num}) \rightarrow \text{NP(Num nom)} \text{VP(Num)}. \) Because it is legal to rewrite \( S(\text{sg3}) \Rightarrow \text{NP(\text{sg3 nom}) VP(\text{sg3)}) \) the reduction can be made by binding \text{Num} to \text{sg3}. This reduction cannot be made in the final step in figure 1b, because it is illegal to rewrite \( S(\text{?}) \Rightarrow \text{NP(\text{sg3 nom}) VP(\text{pl})}. \) In terms of reduction, it would require to bind \text{Num} to both \text{sg3} and \text{pl}, which is impossible. In other words, combining features in a reduction can be done by locally binding the values

\[ \text{Note that } \sigma \text{ and } \sigma' \text{ are not identical, because } \sigma \text{ may contain features.} \]
that are associated with the symbols on the stack to the variables mentioned in the rules on the corresponding places. If a binding cannot be resolved uniquely, the reduction cannot be made.

More technically speaking, at the moment of reduction the top of the stack contains the symbols \( A_n(f_1, \ldots, f_n, k) \) and the rule under reduction is

\[
A_n((v_{0,1}, f_{0,1}), \ldots, (v_{0,k_0}, f_{0,k_0})) \rightarrow \ A_1((v_{1,1}, f_{1,1}), \ldots, (v_{1,k_1}, f_{1,k_1})) \rightarrow \ A_2((v_{2,1}, f_{2,1}), \ldots, (v_{2,k_2}, f_{2,k_2}))
\]

What the reduce step has to do now, is to bind each variable \( v_{i,j} \) to the feature sets \( f_{i,j} \), while respecting the restrictions \( f_{i,j} \) for \( 1 \leq i \leq n \) and \( 1 \leq j \leq k_i \); for soundness, assume that \( f_0 = \emptyset \). If binding is successful, the features at the left-hand side can be computed – observe that, in agreement with the definition of the ACFG-rewriting rule, unbound variables can assume any value. Thus, the value of a variable \( v \in V \) is

\[
\text{value}(v) = \bigcap_{v_{i,j} = v} f_{i,j}
\]

i.e. the intersection of all the feature sets and the feature restrictions at the position where \( v \) is written. The condition \( f_{i,j} \cap f_{i,j} = \emptyset \) must hold for the positions at which \( v_{i,j} = v \). Each left-hand side feature set \( f_0, j = v_0, j \cap f_{0,j} \). This value must also be \( \neq \emptyset \). If one of the mentioned conditions does not hold, the reduction step fails; otherwise it replaces the top \( n \) symbols with \( A_0(f_0, \ldots, f_k, k_0) \).

This definition of variable binding is in complete agreement with the definition of rewriting in section 2. Although it resembles unification – this type of variable binding is sometimes called snapshot unification –, it is not exactly identical. The main difference is that after the reduction has been performed the relationship between features, that shared the same variable, ceases to exist, while this relation keeps existing when using real unification. An example may clarify this. Suppose an ACFG contains the two rules \( A(X,Y) \rightarrow B(X,Y) \) and \( B(X,X) \rightarrow C(X) \). After the reduction of the latter rule, the stack contains the symbol \( B \) plus two identical feature sets. If next the former rule is reduced, \( X \) and \( Y \) will receive the same value, but \( X \) and \( Y \) will not be identical as they would have been when real unification was used. This restriction does not seem to hinder grammar development too much, whereas maintaining the administration for unifications is an expensive task.

4.2. Error detection

We will now focus on the two error detecting functions of an ACFG. The parser's action when reducing an error rule is very simple: it keeps tabs on the error message attached to the rule and to which part of the sentence the error message applies, as shown by (5). As an example, take the requisite comma rule of section 3, and its application in a rule \( X \rightarrow A \). requisite comma B. After an A has been recognized, the parser either shifts a comma and reduces the non-error rule, or reduces the (empty) error rule. In the latter case, the error message is attached to the stack, and after parsing has finished, the position of the missing comma can be identified exactly.

A not unusual action can be taken in case of feature mismatch. However, it requires an additional adaptation of the reduction step. Instead of failing, the reduction step now has to make a note of the type of error before continuing parsing. In the example of figure 1b, it would imply reducing according to rule 1 and attaching an error message to \( S \) saying that the first feature of the NP he did not match with the first feature of the VP sleep. But how to continue parsing if the obstructing variable also occurs on the left-hand side of the rule? Should the final reduction in figure 1b give \( S(pl) \) or \( S(sg3) \)? Either the parser postpones the decision to a later stage, or it makes the decision immediately. If the first method is used, parsing continues with two parses instead of one, which has two clear disadvantages: it is a computationally expensive method and it only delays the decision in most cases: both choices, i.e. both sg3 and pl in this example, will appear in the final set of parses, in which case the decision still has to be made with the same data. Therefore, it makes more sense to make the decision right away.
The current formalism does however not provide us with enough information to make this quick decision. Thus, it should be extended with a way of declaring which error is the more severe one. The most flexible way of providing this information is (1) to attach an error weight to each feature of each symbol on the right-hand side of each rule, and (2) to attach a separate error weight to each error rule. While parsing, the parser maintains the total sum of the error weights, and, in case of ambiguity, chooses the parse with the least error weight. Before the formal definitions are given, a simple example might be appropriate. The rule from section 2 could be extended with error weights as follows.

\[ w_i + \sum_{ij} \begin{cases} w_{ij} \text{ if } \text{bind}(v_{ij}) \cap f_{ij} \cap f_{ij}' = \emptyset \\ 0 \text{ otherwise} \end{cases} \]

with \( \text{bind}(L) = F \). The value of the left-hand side features is defined as

\[ f_{0,j} = \begin{cases} \text{bind}(v_{0,j}) \cap f_{0,j} \text{ if } \text{bind}(v_{0,j}) \neq \emptyset \\ f_{0,j} \text{ otherwise} \end{cases} \]

The total error weight of a parse is the sum of the error weight of its reductions plus the error weight of its input sentence. In case of ambiguity, the best parse for a sentence is the parse with the lowest error weight. This definition of the reduce step will allow us not only to detect structural and feature agreement errors, but also to correct them.

### 4.3. Spelling errors behind word lattices

Now that the grammatical side has been taken care of, we must take a look at the input side. If we want to process ungrammatical natural language input, the input representation must be capable of holding lexical ambiguity, corrected spelling errors and idiomatic expressions.

The solution chosen here is to describe the sentence in a (directed a-cyclic) graph with a unique start and stop vertex, where each edge represents a word plus its category. So instead of a list of alternative categories, the parser now sees lexical ambiguity as a number of edges originating and ending in the same vertex. Each edge can be labelled with additional information, such as an error message or weight, or a transition probability. Such a graph is called a word lattice; its use is very common in e.g. speech recognition, where parts of words may overlap and some parts of the sentence may turn out to be meaningless noise. In spelling correction in texts, words cannot overlap, although their correction alternatives can; e.g., metaan is an existing word (Eng.: instantly), but it can also be split into meta+ean (Eng.: meta+toe) and me+teen (Eng.: with a). An example of a word lattice for the sentence *Eye saw the man that fell* is shown in figure 2. Idiomatic expressions can be treated in the same fashion, by adding a single edge for the entire expression plus, if possible and necessary, an edge for each separate word. For more examples of the use of word lattices in spelling correction, see Carter (1992).

### 4.4. The GLR algorithm for ACFGs with error detection and word lattices

We assume familiarity with the work on context-free natural language parsers, as described by Earley (1970), Aho and Ullman (1972), and, in particular, with the work on the GLR parser by Tomita (1986, 1991). For a comparison of Earley's parser with Tomita's, see Sikkel (1991).

---

\( ^4 \) As will be explained later, a spelling error causes a non-zero error weight in the input.
In this section, we will alter the GLR algorithm and its main data structure, the graph-structured stack. A (normal) graph-structured stack GSS is a doublet \((T,S)\), where \(T\) is a set of top nodes, and \(S\) is the stack itself. \(S\) is a set of 6-tuples \((n,s,\sigma,p,i,j)\), where \(n\) is the node index, \(s\) is the state, \(\sigma\) is the symbol at this node, \(p\) is a set of predecessor nodes, and \(i\) and \(j\) are integer numbers indicating that \(\sigma \Rightarrow a_i \ldots a_j\). The algorithm operates only on the nodes in \(T\).

The addition of word lattices is fairly straightforward. Instead of shifting only a single word at a time, it is possible to shift a number of words at once and add a node on the corresponding position. Therefore, a sentence is no longer seen as a sequence of word categories, but as a set of categories with a begin and end position. That is, a sentence is a set of triples \((c,i,j)\), where \(c\) is a category and \(i\) and \(j\) are integer numbers marking the first and last position of a word. In this representation, the sentence of figure 2 is \{\(\text{noun,1,1}\), \(\text{verb,1,1}\), \(\text{pron,1,1}\), \(\text{verb,2,2}\), \(\text{det,3,3}\), \(\text{det,3,4}\), \(\text{det,4,4}\), \(\text{noun,5,5}\), \(\text{relpron,6,6}\), \(\text{verb,7,7}\), \(\text{S,8,8}\)\}. Some precautionary measures must be taken. E.g. it does no longer make sense to use the concept "next word", as this word may differ for each path. Therefore, the pointer walks through the sentence once position at a time, shifting all words immediately following the current position and reducing all nodes that end at this position, using all categories that start at the next point as the look-ahead symbol.

The extension with grammar augmentations is more complicated. It no longer suffices to collect all points from the graph-structured stack to which the reductions lead. Instead, during the reduction, the features of the new non-terminal have to be computed from the feature values along the paths. Therefore, the definition of a node, the join step and the reduction function have to be adapted. A node now becomes a 7-tuple \((n,s,\alpha,f,p,i,j)\), where \(f\) is the set of all possible feature value combinations the symbol \(\alpha\) can have at this node, e.g. there might be a node \((42, 9, \text{NP}, \{\text{pl,nom}\}, \{\text{pl,acc}\}, \{\text{pl,dat}\}, \{6, 9\}, 1, 2)\), corresponding to a phrase such as the answers. The reduction function is now responsible for delivering a set of \((n,f)\), i.e. preceding nodes plus the features that were collected from the particular path that lead to that node. If a new node joins the stack, its features are simply copied. If it must be merged with an already existing node, its features must be added to the existing node's features.

We can now turn towards the incorporation of grammatical error detection capabilities into this algorithm. The first obvious extension is to incorporate the total error weight plus the errors into the nodes, which by now have become \(8\)-tuples \((n,s,\alpha,f,p,i,j,w,\tau)\), where \(w\) is the total error weight for this node, and \(\tau\) is the set of accumulated errors\(^5\).

The second extension that has to be made concerns the join operation. If a node joins the stack and is a candidate to be merged with another existing node, there are three possibilities: it has the same, a higher or a lower error weight. If it has the same error weight, things proceed as usual. If it has a higher error weight, the new node is simply discarded. If it has a lower error weight, the existing node must be replaced by the current node (while also removing the existing node from the list of active top nodes). The reason is very simple; if two nodes represent the same state for the parser, then they will both be treated identically, resulting in a duplicate set of parses with only a different error weight. Since we always prefer the tree with the lowest error weight we can safely discard the node with the higher error weight. Note that a node includes a feature and an error set. If the features of the two nodes do not overlap, the node with the higher error weight cannot be discarded. Therefore, it is wiser to remove the overlapping features from the node with the higher error weight, and completely remove nodes that have an empty feature set\(^6\).

The third adaptation concerns the reduction function. It should now provide us with the triple \((n',f',w',\tau')\), where \(w'\) is the weight of the error made in this particular reduction, and \(\tau'\) the corresponding errors. The fourth and final extension is another addition to the shift step, which can also introduce extra error weight, resulting from the input sentence. If a word exists and it must be offered to the parser as well as its correction alternatives, additional error weight for the corrections might be in place to prevent that a parse with a spelling correction is preferred over one without a correction\(^7\). A sentence is a set of \(5\)-tuples \((c,f,i,j,w)\), of categories, features, begin and end positions, and the error weight, although a sixth and a seventh element, the

---

Footnotes:

5. These are needed for the correction phase (cf. section 7).

6. Nodes with different error sets do not have to be compared. If the rest of both nodes of a graph-structured stack are compatible, have the same state, positions, etc., the sets of parse trees of both nodes can be united. If the nodes do not have the same error weight, one of them can be discarded, unless the user is interested in all possible parses.

7. If a word is not an existing word, there is no reason to add any extra error weight to the input, because it will only slow down parsing. If we do add extra error weight, all parses will be increased by this amount, thereby making the error weight redundant.
word itself plus its possible corrections, might also be added.

The only remaining technical detail is the exact result of the reduce step. It is possible for a single reduction to have multiple results, because an error can be corrected by assuming different values for different variables in the rule under reduction (cf. section 4.2). The reduction step either lists all possible outcomes, leaving it to the remainder of the parsing algorithm to filter out the best alternative, or returns only the variants with the least number of errors. If nodes with higher errors weights are discarded in favour of compatible nodes with a lower error weight, there is no difference between these two alternatives, except that the second alternative will be computationally cheaper. If all parses are required (cf. footnote 6), the first alternative must be chosen.

If Tomita’s algorithm is adapted along these lines, it will become a rather inefficient algorithm. The parser will always proceed with all parses, including the highly erroneous ones. This is quite useless as we are only interested in the parse with the lowest error weight. To improve parser efficiency, we can utilise the following property. Because all error weights are greater than or equal to zero, the error weight will (non-strict) monotonically increase from left to right. Consequently, it is only necessary to consider the nodes in the graph-structured stack with the lowest error weight and only the left-most ones of those. Thus, the parser may end up with different sets of active top nodes, each representing a certain combination of error weight and position. As soon as the parser has accepted a node with a certain error weight, it should stop after the nodes with that error weight have been exhausted. The final version of the algorithm can be found in the appendix. Because parses with a great number of errors are hardly interesting, the speed of this algorithm can be improved by using an upper limit on the error weight or a maximum allowed increment of the error weight per shift or reduce step or both.

4.5. Improving Speed
The main reason of Tomita’s algorithm’s speed is the use of an LR table. Lankhorst (1991) arrives to the conclusion that LALR(1) tables are the most efficient tables, but these perform only marginally better than the most other table construction techniques. Obviously, it is possible to increase the parser’s speed to rewrite the grammar. Rewriting is not always commendable: when encoding larger grammars for efficiency it is very easy to introduce errors that are difficult to recover. Besides, it makes the grammar less readable and therefore less maintainable and expandable.

On the other hand, especially broad coverage systems do not need to inspect all possible parses. E.g., spell and grammar checking systems are not interested in all variations of PP attachment, nor are such systems interested in the exact position of adverbial modifiers (cf. Jensen, Heidorn and Richardson (1993)), because these affect neither the spelling of words nor the word order in the sentence. Considering that these ambiguities appear as shift-reduce or reduce-reduce conflicts in an LR table, excessive parses can be eliminated by manually removing ambiguities from the LR table. Because such tables are not very open to inspection, it is more practical to allow for a priori conflict resolution rules in the grammar. E.g., a rule that says “if the choice is between reducing an NP or shifting a preposition, prefer shifting”, formulated as P → NP will remove all ambiguity from a simple grammar containing NP-PP ambiguity. Consequently, a parser using this table will come up with at most one parse for every sentence. A strategic choice of disambiguation rules can improve a parser’s performance (cf. section 4.6).

A different factor partially determining an ACFG parser’s speed is the storage of features. In the algorithm of the appendix, all features that a non-terminal node can have, are stored separately in a set. Although this does not influence the algorithm’s complexity — the number of features is finite — this type of storage may become a serious bottleneck when a large number of features is valid for a certain node. Therefore it is more economical to work with combinations of sets of features with features of combinations of features. E.g., instead of storing NP(sg3 nom) and NP(pl nom) separately, they could be stored as NP((sg3,pl) nom) instead. This scheme requires some additional computation, because two combinations of sets can only be merged if they differ in precisely one element. The details and more complicated aspects have been described by Dekkers, Nederhof and Sarbo (1992).

4.6. Performance
The performance of the GLR algorithm has extensively been discussed in the literature (cf. Tomita et al. (1991), which contains several articles concerning performance). All of these studies indicate that the algorithm parses its input approximately in time \(O(n^3)\) for common natural language grammars. We will now take a short look at the performance of the algorithm of section 4.5. We will do this systematically by using the (in)famous S-NP-VP-PP example (cf. figure 3a) and its feature-equipped variant (cf. figure 3b) found in so many books on natural language parsing. In the ACFG grammar, the penalty for violating the transitivity feature is 1 point, for violating the number feature, it is 2 points, and for violating the case feature, it is 10 points.

To analyse the amount of time needed for feature evaluation\(^8\), a comparison is made between grammars 4a and 4b. In the first run, the sentences of the scheme \(u + a + (p n)\) were parsed using the first

---

\(^8\) All tests were run five times on a DECStation 5000/125, and averaged; the times did not differ more than 21 clock tick, which is 1/60th of a second. The time was measured between generating the first stack frame and the acceptance of the first tree or the rejection of the entire sentence.
grammar, with the number of PPs ranging from zero to twenty. In the second and third run, the second grammar was used with the same sentences, only now with features, \( n(\text{sg}, \{\text{nom}, \text{acc}\}) \) \( v(\text{sg}, \text{trans}) \) \( p \) \( n(\text{sg}, \{\text{nom}, \text{acc}\}) \). In the second run, the code for maintaining the error administration was bypassed, while it was enabled in the third run. The results of these tests are shown in figure 4, labelled runs 1, 2 and 3. As this figure shows, parsing times are nearly equal.

To examine the effect of the number of errors on the parsing time, several conditions were compared. In run 4, the verb was made intransitive, which caused a penalty of one error point for each sentence. The effect was an increase in parsing time of approximately 60 percent, slightly less for shorter and slightly more for longer sentences. In run 5, the verb was made transitive but plural, causing a penalty of two error points. The effect in this run was a much smaller increase in parsing time, being only 45 percent, which decreased for longer sentences. The combined effect of runs 4 and 5 is disclosed in run 6, which features an intransitive plural verb, causing a three error point penalty in each sentence. This run displays a 120 percent increase, which is nearly flat, although slightly decreasing for longer sentences (which is surprisingly similar to the summed effect of run 4 and 5; a closer study of the figures shows that the effect in run 6 is almost identical to the summed effects of run 4 and 5 for all cases). Finally, in run 7, the noun in the final PP was changed from accusative to nominative only, which causes an error penalty of 10 points. However, as already discussed in the previous section, the implementation has an upper limit on the total error weight, which was set to 9 in this run. Thus, the parser did not accept a sentence but the first, the one without a single PP. The increase in parsing time was approximately 8 percent for the longer sentences. The results from run 4 through 7 can also be found in figure 4.

But how does the error position influence parsing time? Apparently, if the error occurs early in the sentence, parsing time increases more than when the error occurs towards the end of the sentence. This hypothesis is supported by the data in figure 5. In this test, the parser was given the sentence with twenty PPs, but with a case feature violation in one PP, causing an error penalty of one point, as in this run the case feature weight was set to one. Figure 5 shows the increase in parsing time when varying the position of the error from the first PP to the last PP compared to the parsing time of the errorless twenty PP sentence. The table evidently shows that the parsing time strongly depends on the position of error; the difference runs from nearly doubling the parsing time to an increase of less than 20 percent.

This effect is caused by the multiple stack mechanism in combination with the peculiar nature of this grammar. Until the error position is encountered, parsing runs as efficiently as in the errorless case. However, at the location of the error, the parser can take two possible paths: one involves shifting the noun (and attaching at least one following PP to the NP node the noun will become later on), the other involves reducing it to an NP (and subsequently combining this NP with the preceding preposition to a PP). The last step creates an error, so the parser only pursues the first path. However, after the next PP, the parser will again try to reduce the current noun plus that PP or continue with the deeper PP. At this point, reducing gives an error again, which creates a new position in the error stack, and is ignored by the parser for the time being. This process continues until the end of the sentence, at which point no error-free parse is available. Consequently, parsing resumes at the point the first error was encountered and follows its way through the sentence, joining all stacks with an error weight of one that were left behind. Since there are no errors left, the process now proceeds as usual. Recapitulating, the earlier the error occurs in the sentence, the less paths will be joined on the error-free stack, as a result of which parsing will take longer.

On the other hand, in run 5 it was shown that an early error position does not necessarily lead to a larger increase in processing time; on the contrary, the longer the sentence, the less parsing time increased. This effect has a similar explanation, but in this case detection of the error does not lead to a difference in error weight amongst ambiguous paths. After parsing the initial NP – the subject – the
parser will give an analysis for the following VP, regardless of person and number of the NP. Only after reaching the end of the sentence the parser will encounter an error. Although it might be expected that there should be only one erratic reduction in the entire process, this is not entirely true; due to a lack of synchronisation in GLR-parsing – in this particular case, there are as many S → NP VP reductions as there are prepositions – there is a number of error containing reductions. Each of these reductions slows down parsing when it introduces an error. As a result, the delay occurs – in this particular case – O(n) times. Because parsing takes more than O(n) time, the relative increase in parsing time decreases as the sentences become longer.

Another question that comes to mind is: how is parsing time affected by local errors that do not show up globally? As could be expected, parsing time is affected, but not seriously. To test this, the following rules were changed:

4'. VP(Num) → VP(Num) PP(pl)
5'. PP(Num) → p NP(Num acc)
8'. NP(Num Case) → NP(Num Case) PP(sg)

These rules only allow attachment of plural PPs to a VP and singular PPs to an NP. Thus, when parsing a sentence with only plural PPs, there is only one error-free parse, while all other parses contain an error weight of 2. The error-free parsing result is identical to deleting rule 8', so we can compare the parsing times with and without rule 8'. When parsing a sentence with only singular PPs, there are be multiple parses, and the result is identical to that of parsing without rule 4'. Testing the first case, plural PPs only, we find a linear parsing time that shows only the slightest increase, 0.5%, over running the same test without rule 8'. When testing the second case, singular PPs only, we find normal (near-cubic) parsing times, with a slight increase, 13%, when compared to parsing without rule 4'. Thus, we may conclude that local errors do not have an unacceptable influence to the performance of the algorithm.

Altogether, we can conclude that in the worst case a single error, occurring at a position in which it enforces stack splicing, can increase the parsing time with a factor O(n), giving the algorithm a O(nk+3) time complexity, where k is the number of errors in the sentence. As we also have seen, this behaviour can be avoided by keeping the introduction of errors away from points of ambiguity. E.g., if the grammar in figure 3b had accounted for PP modifiers at sentence level using S → S PP, the parsing times in run 5 would have been as bad as those in run 4', not to mention the effect it would have had on run 6. The reason is similar to the explanation given for run 4: the joining of stacks would have been made impossible at a highly ambiguous point in the parsing process. In a similar vein, by using extra non-terminals in the ACFG grammar, distinguishing the non-terminals that can have PP-attachment from those that cannot, the position effect in figure 5 can be prevented.

This suggests an automatic way of finding the problem areas in a grammar. Each entry in the action table should be inspected for shift-reduce and reduce-reduce conflicts. In case of reduce-reduce conflicts, inspection of the rules involved can reveal

---

These assertions have been verified.
possibly conflicting situations in which one rule reduces with a lower error weight than the other. In shift-reduce conflicts, the conditions for worst case behaviour can be determined and either be reported or automatically repaired by introducing an extra non-terminal - which may be a rather expensive solution, though. Manually solving shift-reduce and reduce-reduce conflicts does not only yield an improvement of parsing time in erroneous sentences, but also in error-free sentences, as conjectured in section 4.5. E.g., when using a larger grammar, resolving shift-reduce conflicts involving prepositions and adverbs in favour of shifting, may improve parsing time with 15%. All findings make clear that carefully examining the grammar to reduce ambiguity is very worthwhile.

5. CORRECTION, OR HOW TO IMPROVE A TREE

After the parse trees of minimal error weight have been supplied by the parser, any errors in the tree must be corrected. Errors fall into one of the classes, spelling errors, lattice errors, structural errors or agreement errors. In the following, these errors will be treated as if there was only a single tree. Later we shall look at the problem of ambiguity and of correcting trees in a forest without unpacking all trees, which is an expensive operation.

Spelling errors are denoted by a flag at the terminal node; all the corrector needs to do is to replace the input word by the alternative given by the spelling corrector. If there is a list of alternatives attached, the corrector should pick out the best scoring alternative with the correct category and matching features. When noticing an error rule, the corrector can give a simple warning. It is also possible to annotate each error rule in the grammar with an appropriate explanation or a specific correction procedure.

In case of agreement errors, the corrector does not need to rely on correction procedures provided by special error rules, as has been proposed by Schwind (1990). Instead, it can reconstruct the tree from the ACFG rules and the parser's error messages. The idea is to recompute the correct feature value on the basis of the values of surrounding nodes, because the context-free structure of the parse is sound (cf. section 2). E.g., when completing the parse in figure 1, the parse tree of the sentence *he sleeps* looks like figure 6. The bold node contains an error in the feature indicated by an asterisk.

By inspecting the rule $S(\text{Num}) \rightarrow \text{NP}(\text{Num nom}) \ 
\text{VP}(\text{Num})$ we can see that the correct value for $\text{VP}(\text{sg}1)$ is $\text{VP}(\text{sg}3)$. Because the next rule $\text{VP}(\text{Num}) \rightarrow v(\text{Num})$ requires that both symbols have the same feature value, $v(\text{sg}1)$ should be replaced by $v(\text{sg}3)$. At this point we can determine that the word *sleep* has been misspelt. By looking up other forms of the word, we find that (only) *sleeps* is of the category $v(\text{sg}3)$. Therefore, the correct sentence is *he sleeps*.

The general procedure is to recompute the feature values for each node in a top-down fashion and, at the points indicated by the parser's error message, copy the appropriate value from the parent or sibling.
nodes with the same feature. At each terminal leaf the correction procedure checks whether the computed feature values still correspond to the lexical definition of the original word. If not, the words that share the same stem are retrieved and scanned for a variant with compatible feature values. If such a word is found, it replaces the original word; if such a word cannot be found, an warning should be given that the word is misspelt.

As was already mentioned in section 2, not all \( \sigma \in L(G) \cdot L(G) \) necessarily have a correction. This is even more true for the parse trees that can be built of an erroneous sentence. A number of trees will not be correctable, and for some sentences, no tree is correctable using the scheme outlined above. The reason is that it is to some extent naive to think that replacing a single word will correct the error; some errors require the substitution of a complete sub-tree. If we take a look at the following grammar, this will become clearer.

\[
S \rightarrow A(X) \ B(X) \quad A(a) \rightarrow a \\
A(b) \rightarrow b \quad B(a) \rightarrow c \\
B(b) \rightarrow d
\]

The only two sentences derived by this grammar are \( ab \) and \( bd \). However, the context-free part also derives \( a \) and \( bc \). If one of these sentences is fed into the parser, it will produce a tree with one feature conflict, that cannot be resolved by the corrector, because it requires a change of rule. Although it might be possible in this particular case — it only requires the substitution of one symbol by another — it is not possible in general when the other rule contains symbols that are not in the original rule. The correction can only be made with certainty when there is a rule with an identical context-free structure or the extra introduced symbols have a unique lexical realisation, e.g. a comma or an indefinite article. If these conditions are not met, the parse tree cannot be corrected.

**Ambiguity**

Most natural language sentences have multiple syntactic parses. For sentences that contain errors the number of parses becomes even larger. How can a corrector select the best correction from a set of possible corrections? The answer involves a set of heuristics that assign a score to each correction. All trees have an initial score of 1. For spelling corrections, this score is multiplied by the normalised score of the spelling corrector\(^{10}\). Structural changes are disfavoured slightly, so that analyses which do not force a structural change are preferred. Splitting up illegal compounds is also disfavoured, but joining split compounds and correcting idiomatic expressions as well as deleting duplicated words are preferred in order to prevent too many missing hits. The idea behind these heuristics is that what appears to be an illegal compound in one parse may sometimes be a legal construction in another parse, but since this will not be the case most of the time, lowering the score has no effect if the compounding was really illegal. In contrast, if two consecutive nouns can be joined in one analysis while they are not joined in another analysis, joining them is usually correct because, if it is really unwanted no parse can be made in most cases; so, if it can be made, joining is probably necessary\(^{11}\). This type of reasoning also holds for the other examples. If the corrector is intended for use by a very competent author, the preferences may have to be changed.

This disambiguation procedure is based on the assumption that the corrector receives a set of trees. In case of a Tomita or Earle parser, it is rather a forest, in which trees share components. Since expanding a forest into a set of trees is too expensive, the procedure has to be adapted. In the first pass, the corrector only assigns goodness scores to each node. The score of a node is the product of the scores of its daughter nodes times the score for the correction attached to the node itself. If a node is shared, the maximum value of the shared components is taken. The goodness score of a terminal leaf is computed by triphone comparison of the original and the resulting word, which is 1.0 if these are equal. In the second pass the optimal tree is selected top-down by searching for the nodes with the highest values.

6. CONCLUSIONS

It is possible to use natural language grammars, written to describe grammatical sentences using features to assert some generalisations, for parsing both grammatical and ungrammatical sentences efficiently. By explicitly incorporating error rules, the coverage of ungrammaticalities increases. This sentence analysis system can be used to detect and correct frequent morpho-syntactic errors and help improve spelling correction as a side effect without too many computational resources.

ACKNOWLEDGMENTS

The author wishes to thank all people who contributed to the completion of this article: Alice

---

\(^{10}\) A correctly spelt word should receive a score of 1, and a misspelt word should score below 1.

\(^{11}\) This conclusion has been drawn from a few informal tests.
Dijkstra, Gerard Kempen, Anton Nijholt and Stan van de Burgi.

REFERENCES


Richardson, S. D. 1985. Enhanced Text Critiquing using a Natural Language Parser (RC 11332 (#51041)). IBM Thomas J. Watson Research Center.


APPENDIX. TOMITA’S ALGORITHM FOR ACFGs WITH WORD LATTICES, AND ERROR DETECTION.

Main:

\[ T := \{0\}; S := \{(0,q0,\lambda,\{(0\Delta,1,0,0,\varnothing)\})\}; i := 1; A := \varnothing; \]
While \( T \neq \varnothing \) (and \( A = \varnothing \))

Do

\[ V := \varnothing; \]

Let \( w_{\text{min}} = \min \{ w \mid n \in T, (n,q,\alpha,f,p,i,j,w,\tau) \in S \}; \]
Let \( i_{\text{min}} = 1 + \min \{ j \mid n \in T, (n,q,\alpha,f,p,i,j,w_{\text{min}},\tau) \in S \}; \]

For all \( (a,f,i_{\text{min}},w,\rho,\gamma) \in \text{sentence Do} \; \text{Complete}(a,i_{\text{min}},w_{\text{min}}) \; \text{Od;} \)

For all \( (a,f,i_{\text{min}},i,w,\rho,\gamma) \in \text{sentence Do} \; \text{Shift}(a,f,i_{\text{min}},i,w_{\text{min}},\rho,\gamma,w_{\text{min}}) \; \text{Od;} \)

\[ T := V \]

Od;

return Trees(n) for \( n \in A \)

Shift(a,f,i,j,w,\rho,\gamma,w_{\text{min}}):

For \( n \in T, (n,q,\alpha,f,p,i,j,w_{\text{min}},\tau) \in S \)

Do

If shift \( \in \text{ACTION}(q,a) \)

Then \( \text{Join}(n,\text{GOTO}(q,a),a,f,(n),i,j,w_{\text{min}},w,\{a: \rho \Rightarrow \gamma\}) \) to \( (T,S) \)

Fi;

\[ V := \{n\} \]

Od

Complete(a,i_{\text{min}},w_{\text{min}}):

For all \( n \in T, (n,q,\alpha,f,p,i,i_{\text{min}},w_{\text{min}},\tau) \in S \)

Do

For all \( X \rightarrow X_1 \ldots X_m \in \text{ACTION}(q,a) \)

Then \( \text{For all} \ (n',w',\tau',f') \in \text{Red}\{(n,q,\alpha,f,p,i,i_{\text{min}},w_{\text{min}},\tau),X \rightarrow X_1 \ldots X_m,m,(\lambda)\}, \)

\[ (n',q',\alpha',f',p',i',j',w',\tau') \in S \]

Do \( \text{Join}(n',\text{GOTO}(q',a),X,f',(n'),i',i_{\text{min}},w'+w',\tau') \) to \( (T,S) \) Od

If \( n = S \) and accept \( \in \text{ACTION}(q,a) \) Then \( A := \{n\} \) Fi;

\[ V := \{n\} \]

Od

\[ \text{Red}(n,q,\alpha,f,p,i,j,w,\tau),X \rightarrow X_1 \ldots X_k,0,l) = \{n,w',\tau',f'\} \]

\[ \text{Red}(n,q,\alpha,f,p,i,j,w,\tau),X \rightarrow X_1 \ldots X_k,m,l) = \bigcup_{w' \in \text{ACTION}(q,a)} \text{Red}(n',q',\alpha,f',p',i',j',w',\tau'),X \rightarrow X_1 \ldots X_k,l \times f) \]

Join(n,q,\alpha,f,p,i,j,w,\tau):

If \( (n',q,\alpha,f',p,i,j,w',\tau') \in S \)

Then \( i \text{ If } w' > w \text{ Then } f' := f; \)

If \( n' \in T \) Then \( T := \{n'\} \) Fi;

If \( f = \varnothing \) Then \( S := \{(n',q,\alpha,f,p,i,j,w',\tau')\} \) Fi;

\[ S := \{(n,q,\alpha,f,p,i,j,w,\tau)\}, T := \{n\} \]

Elif \( w' < w \) Then \( f \notin f \)

Then \( S := \{(n,q,\alpha,f,p,i,j,w,\tau)\}, T := \{n\} \)

Fi

Else \( f \notin f \) Then \( f := f', \tau := \tau', T := \{n\} \)

Fi

Else \( S := \{(n,q,\alpha,f,p,i,j,w,\tau)\}, T := \{n\} \)

Fi

* If a new element is added to \( T \), it should also be inspected.

** The error weight and the left-hand side features are computed according to formulas in section 4.2.
LANGUAGE processing shows every indication of becoming one of the main pillars of information technology in the next century. To keep up with this extraordinary development as it occurs, increasing numbers of people in Europe, North America, and East Asia are turning to LANGUAGE INDUSTRY MONITOR, currently the only single source of information about this strategic field. Each issue of this unique bimonthly newsletter offers its readers a timely and incisive roundup of news, background information, and analysis of the rapidly evolving world of language processing. Recent issues of LANGUAGE INDUSTRY MONITOR have included exclusive reports on:

- IBM’s translation strategy
- Soft-Art and Houghton Mifflin
- Why ARPA funds Speech and NLP research
- Boeing puts Simplified English to work
- Oracle’s linguistic software group
- Japanese MT: an overview
- Logos user Lexi-tech
- The Localization Industry Standards Association
- The CEC LRE program: round two
- Electronic publishing

If you too would like to receive LANGUAGE INDUSTRY MONITOR on a regular basis, contact:

LANGUAGE INDUSTRY MONITOR
EERSTE HELMERSSTRAAT 183
1054 DT AMSTERDAM, THE NETHERLANDS
Tel +31 20 685 0462 Fax +31 20 685 4300
INTERNET: COLINB@PARAMOUNT.NIKHEF,NL

LANGUAGE INDUSTRY MONITOR is independently published by Uitgeverij LIM, Reg. in Amsterdam no. 215.527

ANNUAL SUBSCRIPTION: NLG150/UK£50/FFR500/US$95 including airmail shipping
PARLEVINK Research Topics
The Parlevink Project started in January 1992. It did not start from scratch. In previous years research took place in the area of theory of formal and programming languages (theoretical computer science, compiler construction) and more and more this research became influenced by potential applications in the area of natural language processing. Currently the following three research directions are distinguished:

- research which concentrates on syntactic formalisms and where syntax is the starting point for studying the description and processing of semantic and pragmatic aspects of language;
- research which concentrates on the representation of meaning in dialogue modelling and where syntax is of secondary importance;
- research which concentrates on modelling language behaviour with the help of neural networks and where language learning and integrated use of syntactic, semantic and pragmatic knowledge are the main characteristics.

In 1993 and 1994 a start will be made with the integration of the different research tracks in the design of a natural language interface that allows a user to ask information about theatre performances in a city. This will take the form of joint research with PITT-Research and possibly other partners.

PARLEVINK Researchers
More than ten researchers, including Ph.D. students, are involved in the project. In the course of 1993 five Ph.D. students will be involved in the research. Programming support is provided by the Computing Laboratory of the Department of Computer Science. A large number of computer science students are performing their M. Sci. work in the project. It is not unusual that they spend part of their education in companies in the Netherlands or with research groups in the USA.

PARLEVINK Activities
PARLEVINK research is published in books, journals and proceedings of (international) workshops and conferences (COLING, ICANN, KONVENS, 1WPT, etc.). A complete list of publications is available on request. Twice a year a workshop (TWLT: Twente Workshop on Language Technology) is organised. Proceedings of these workshops are available. In 1991 there were workshops on Generalised LR Parsing and on Linguistic Engineering. In 1992: Connectionist Natural Language Processing and Pragmatics in Language Technology. Students and project members are informed about research, lectures and other activities during weekly meetings and in the PARLEBODE, a monthly newsletter.
Twente Workshops on Language Technology

The TWLT workshops are organised by the PARLEVINK project of the University of Twente. The first workshop was held in Enschede, the Netherlands on March 22, 1991. The workshop was attended by about 40 participants. The contents of the proceedings are given below.

Proceedings Twente Workshop on Language Technology 1 (TWLT 1)
Tomita’s Algorithm: Extensions and Applications
Eds. R. Heemels, A. Nijholt & K. Sikkel, 103 pages.

Preface and Contents
A. Nijholt (University of Twente, Enschede). (Generalised) LR Parsing: From Knuth to Tomita.
G.J. van der Steen (Vleermuis Software Research, Utrecht). Unrestricted On-Line Parsing and Transduction with Graph Structured Stacks.
R. Heemels (Occ Nederland, Venlo). Tomita’s Algorithm in Practical Applications.
M. Lankhorst (University of Twente, Enschede). An Empirical Comparison of Generalised LR Tables.
K. Sikkel (University of Twente, Enschede). Bottom-Up Parallelization of Tomita’s Algorithm.

The second workshop in the series (TWLT 2) has been held on November 20, 1991. The workshop was attended by more than 70 researchers from industry and university. The contents of the proceedings are given below.

Proceedings Twente Workshop on Language Technology 2 (TWLT 2)
Linguistic Engineering: Tools and Products.

Preface and Contents
A. Nijholt (University of Twente, Enschede). Linguistic Engineering: A Survey.
B. van Bakel (University of Nijmegen, Nijmegen). Semantic Analysis of Chemical Texts.
T. Vosse (NICI, Nijmegen). Detecting and Correcting Morpho-syntactic Errors in Real Texts.
A. van Rijn (CIAD/Delft University of Technology, Delft). A Natural Language Interface for a Flexible Assembly Cell.
J. Honig (Delft University of Technology, Delft). Using Deltra in Natural Language Front-ends.
D. van den Akker (IBM Research, Amsterdam). Language Technology at IBM Nederland.
The third workshop in the series (TWLT 3) was held on May 12 and 13, 1992. Contrary to the previous workshops it had an international character with eighty participants from the U.S.A., India, Great Britain, Ireland, Italy, Germany, France, Belgium and the Netherlands. The proceedings were available at the workshop. The contents of the proceedings are given below.

Preface and Contents

L.P.J. Veelenturf (University of Twente, Enschede). Representation of Spoken Words in a Self-Organising Neural Net.

P. Wittenburg & U. H. Frauenfelder (Max-Planck Institute, Nijmegen). Modelling the Human Mental Lexicon with Self-Organising Feature Maps.


W. Daelemans & A. van den Bosch (Tilburg University, Tilburg). Generalisation Performance of Back Propagation Learning on a Syllabification Task.

E.-J. van der Linden & W. Kraaij (Tilburg University, Tilburg). Representation of Idioms in Connectionist Models.

J.C. Scholtes (University of Amsterdam, Amsterdam). Neural Data Oriented Parsing.


M.F.J. Drossaers (University of Twente, Enschede). Hopfield Models as Neural-Network Acceptors.


R. Reilly (University College, Dublin). An Exploration of Clause Boundary Effects in SRN Representations.

S.M. Lucas (University of Essex, Colchester). Syntactic Neural Networks for Natural Language Processing.

R. Miikkulainen (University of Texas, Austin). DISCERN: A Distributed Neural Network Model of Script Processing and Memory.

The fourth workshop in the series has been held on September 23, 1992. The theme of this workshop was "Pragmatics in Language Technology". Its aim was to bring together the several approaches to this subject: philosophical, linguistic and logic. The workshop was visited by more than 50 researchers in these fields, together with several computer scientists. The contents of the proceedings are given below.

Preface and Contents

Proceedings Twente Workshop on Language Technology 4 (TWLT 4)
Pragmatics in Language Technology
D. Nauta, A. Nijholt & J. Schaake (University of Twente, Enschede). Pragmatics in Language Technology: Introduction.

Part 1: Pragmatics and Semiotics
J. van der Lubbe & D. Nauta (Delft University of Technology & University of Twente, Enschede). Semiotics, Pragmatism, and Expert Systems.
F. Vandamme (Ghent). Semiotics, Epistemology, and Human Action.
H. de Jong & W. Werner (University of Twente, Enschede). Separation of Powers and Semiotic Processes.

Part 2: Functional Approach in Linguistics
C. de Groot (University of Amsterdam). Pragmatics in Functional Grammar.
E. Steiner (University of Saarland, Saarbrücken). Systemic Functional Grammar.
R. Bartsch (University of Amsterdam). Concept Formation on the Basis of Utterances in Situations.

Part 3: Logic of Belief, Utterance, and Intention
J. Schaake (University of Twente, Enschede). The Logic of Peirce's Existential Graphs.
H. Bunt (Tilburg University). Belief Contexts in Human-Computer Dialogue.

The fifth workshop in the series took place on 3 and 4 June 1993. It was devoted to the topic "Natural Language Interfaces". The aim was to provide an international platform for commerce, technology and science to present the advances and current state of the art in this area of research.


Preface and Contents
F.M.G. de Jong & A. Nijholt (University of Twente). Natural Language Interfaces: Introduction.
L. Boves (University of Nijmegen). Spoken Language Interfaces.
J. Nerbonne (University of Groningen). NL Interfaces and the Turing Test.
J. Schaake (University of Twente). The Reactive Dialogue Model: Integration of Syntax, Semantics, and Pragmatics in a Functional Design.
D. Speelman (University of Leuven). A Natural Language Interface that Uses Generalised Quantifiers.
W. Menzel (University of Hamburg). Title.
G. Neumann (University of Saarbrücken). Design Principles of the DISCO system.

The sixth workshop in the series took place on 16 and 17 December 1993. It was devoted to the topic "Natural Language Parsing". The aim was to provide an international platform for technology and science to present the advances and current state of the art in this area of research, in particular research that aims at analysing real-world text and real-world speech and keyboard input.
Preface and Contents

A. Nijholt (University of Twente). Natural Language Parsing: An Introduction.

V. Manca (University of Pisa). Typology and Logical Structure of Natural Languages.

R. Bod (University of Amsterdam). Data Oriented Parsing as a General Framework for Stochastic Language Processing.

M. Stefanova & W. ter Stal (University of Sofia / University of Twente). A Comparison of ALE and PATR: Practical Experiences.

J.P.M. de Vreught (University of Delft). A Practical Comparison between Parallel Tabular Recognizers.

M. Verlinden (University of Twente). Head-Corner Parsing of Unification Grammars: A Case Study.


Th. Stürmer (University of Saarbrücken). Semantic-Oriented Chart Parsing with Defaults.

G. Satta (University of Venice). The Parsing Problem for Tree-Adjoining Grammars.

F. Barthélémy (University of Lisbon). A Single Formalism for a Wide Range of Parsers for DCGs.


C. Cremers (University of Leiden). Coordination as a Parsing Problem.

M. Wirén (University of Saarbrücken). Bounded Incremental Parsing.

V. Kubon and M. Platek (Charles University, Prague). Robust Parsing and Grammar Checking of Free Word Order Languages.

V. Srinivasan (University of Mainz). Punctuation and Parsing of Real-World Texts.

T.G. Vosse (University of Leiden). Robust GLR Parsing for Grammar-Based Spelling Correction.

The proceedings of the workshops can be ordered from Vakgroep SET1, Department of Computer Science, University of Twente, P.O. Box 217, NL-7500 AE Enschede, The Netherlands. E-mail orders are possible: byron@cs.utwente.nl. Each of the proceedings costs Dfl. 30.