Tomita's Algorithm: Extensions and Applications

Proceedings of the first
Twente Workshop on Language Technology

Rob Heemels, Anton Nijholt, Klaas Sikkel (eds.)

Memoranda Informatica 91-68
September 1991

ISSN: 0924-3755

Vakgroep SETI
Faculteit Informatica
Universiteit Twente
Postbus 217
7500 AE Enschede
PREFACE

The Computer Science department of the University of Twente intends to host a series of *Twente Workshops on Language Technology* (TWLT). The first of these workshops has been held in Enschede, The Netherlands, on March 22, 1991, under the title *Tomita’s Algorithm: Extensions and Applications*. The papers presented at this workshop are contained in this volume. An overview of the history of (generalized) LR parsing has been added as an introduction.

"Tomita’s algorithm" is the generalized LR parsing algorithm introduced by Masaru Tomita in a book published in 1985 and in some papers and reports before that date. Although the method is fairly straightforward and has also been thought of by others, it never had the clear exposition as presented by Tomita. Since then, others have investigated the method both theoretically and empirically. Some of their results can be found in the proceedings of the first and second *International Workshop on Parsing Technologies (IWPT)*, held in Pittsburgh, PA, August 1989 and Cancun, Mexico, February 1991. The papers concerning generalized LR parsing presented at the first IWPT will also appear in a separate book, *Generalized LR Parsing*, M. Tomita (Ed.), Kluwer Academic Publishers.

This volume contains another set of papers concerned with generalized LR parsing and closely related methods. The authors stem from universities and industrial research and development groups. We are grateful to Philips–Eindhoven, Océ–Venlo and Vleermuis BV–Utrecht for the contributions that could be made by R. Leermakers, R. Heemels and G. van der Steen. We also thank Océ–Venlo for giving M. Lankhorst the opportunity to spend three months with the “Language and Speech” group of Océ and the Center of Machine Translation of Carnegie Mellon University, Pittsburgh, for giving H. Harkema the opportunity to spend six months at the Center. Harkema’s stay was supported by NUFFIC, a Dutch organization for stimulating international scientific contacts. The research performed by Lankhorst, leading to a paper included in this volume was directed by R. Heemels. The research performed by Harkema, leading to a joint paper in this volume, was directed by M. Tomita.

Many people contributed to the success of the workshop. Obviously, we should thank the contributors for their willingness to write the papers and present them at the workshop. We also thank the other participants for their contribution to the discussions.

Apart from the editors, M. Lankhorst helped with organizational tasks. Secretarial help before and during the workshop was provided by mrs. A.W. Hoogvliet-Haverkate. Last but not least we would like to thank the Faculty of Computer Science for their financial support for printing and distributing these proceedings.

Rob Heemels, Anton Nijholt, Klaas Sikkel (eds.),
September 1, 1991.
CONTENTS

(Generalized) LR Parsing: From Knuth to Tomita .................................. 1
Anton Nijholt (University of Twente)

Recursive Ascent Parsing ................................................................. 9
René Leermakers (Philips Research Laboratories, Eindhoven)

A Parsing Algorithm for Non-Deterministic Context-Sensitive Languages ........ 21
Henk Harkema (University of Twente),
Masaru Tomita (Carnegie Mellon University, Pittsburgh)

Unrestricted On-line Parsing and Transduction with Graph Structured Stacks ...... 33
Gert J. van der Steen (Vleermuis Software Research, Utrecht)

Substring Parsing for Arbitrary Context-Free Grammars ............................ 61
Jan Rekers (Centre for Mathematics and Computer Science, Amsterdam),
Wilco Koorn (Programming Research Group, University of Amsterdam)

Detection and Correction of Morpho-Syntactic Errors in Shift-Reduce Parsing .... 69
Theo Vosse (NICI, University of Nijmegen)

Tomita’s Algorithm in Practical Applications ........................................ 79
Rob Heemels (Dept. of Language and Speech Technology, Océ, Venlo)

An Empirical Comparison of Generalized LR Tables ................................ 87
Marc Lankhorst (University of Twente)

Bottom-Up Parallelization of Tomita’s Algorithm .................................... 95
Klaas Sikkel (University of Twente)
(Generalized) LR Parsing: From Knuth to Tomita

Anton Nijholt

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

ABSTRACT

This paper is a short introduction to the research in LR parsing and its applications. It is concerned with the history of LR grammars and languages, LR parsing and parser optimization, generalizations of the LR grammar definition and parsing method, automatic parser construction, error handling and LR parsing for natural language. Moreover, it introduces the other papers on (generalized) LR parsing in these proceedings.

1 INTRODUCTION

These notes provide a short introduction to the field of LR parsing. That is, we mention the research topics that emerged in this field and some activities that have been going on since the early sixties of this century. These proceedings are devoted to generalized LR parsing, an extension of the LR parsing method. For computer scientists LR parsing was introduced in a paper by Donald Knuth in 1965 [Knuth65]. After the introduction of the method it became silent for a few years, until F.L. DeRemer presented more practical, i.e., efficient, versions of the algorithm and corresponding subclasses of the class of LR grammars, viz. the SLR and LALR grammars [DeRemer69]. Nevertheless, practical compilers and compiler writing systems for programming languages were still based on precedence or recursive descent parsing techniques. W.R. LaLonde wrote an influential paper in which he reported the construction of an LALR parser generator [LaLonde72]. It was shown that many time and space optimizations could be done on the constructed LR tables and empirical comparisons were made with other parsing techniques. LR parsing compared favourably with other methods and, moreover, it was claimed that existing 'natural' grammars for programming languages required little or no change in order to make them LALR. Some years later S.C. Johnson chose the LALR parsing technique for a compiler writing system that was made part of the UNIX operating system: Yacc, Yet another compiler-compiler [Johnson75]. Since then (programming) language designers had a simple tool to their disposal for generating LALR parsers. Due to these developments in computing science LR parsing has become the most commonly used parsing technique. Indeed, as mentioned by LaLonde in 1977, LR(k) grammars "... are popular because the grammars describe a large class of programming languages, the parser constructing techniques take a 'reasonable' amount of time, the parsing tables take a 'reasonable' amount of space, the parsers are fast, i.e., operate in linear time, and they have excellent error detection properties." A clear exposition of the different LR parsing techniques can be found in [Aho86].

In the first years after the introduction of the Chomsky hierarchy similar parsing methods were used by compiler builders and builders of natural language parsers. The results obtained by researchers in this area were shared. Results obtained in formal language theory could be used for programming language and natural language parsing. However, in computing science the main interest was in unambiguous grammars and there was a strong need for efficient, linear time, parsing methods in order to reduce compile time of programs. In natural language parsing the grammars had to capture the syntactic ambiguities of natural language. Therefore, efficient methods for the full class of context-free grammars had to be found. Since it turned out that parsing with respect to a context-free grammar for a (subset of a) natural language would lead to an explosion of the number of syntactic parses other researchers turned to non-context-free for-
malisms for natural language description. Sometimes these formalisms and parsing methods remained close to those for context-free grammars. Two aspects were important: 1) the embedding of syntax in formalisms and methods in which semantic information could be used to reduce the number of syntactic parses, and 2) notations that were more suitable for computational linguists to display and manipulate syntactic and semantic information. For some time augmented transition networks (ATNs) were used and some domain-dependent natural language processing systems were built based on ATNs [Woods70]. Other researchers put more emphasis on semantics and introduced more semantically oriented parsing methods, for example conceptual dependency parsing [Riesbeck75]. Results on the full class of context-free grammars were obtained by formal language theorists. Hardly any interest was displayed by computer scientists or computational linguists. Polynomial-time algorithms were obtained by Younger (the Cocke-Younger-Kasami algorithm [Younger67]) and Earley ([Ear-
ey68]). Earley had already published on LR parsers in 1967 and work done in 1964 had influenced Knuth's definition of LR grammars. The papers by Woods, Younger and Earley all appeared in computing science journals. Although it can be considered as a notational variant of the Earley and CYK method, (active) chart parsing, introduced by Kay (see [Kay80]), became more popular in the computational linguistics community than other context-free parsing methods. Its main advantage was a clear visualization of the parsing process.

In the 1980s there has been a revival of context-free grammars and parsing methods in computational linguistics. New grammar formalisms were introduced (e.g., generalized phrase structure grammars) with explicit context-free components and discussions were started whether or not natural languages are context-free (see e.g. [Savitch87] for a collection of papers). Moreover, there was a decrease in a reluctance to study and use formal methods from formal language theory in computational linguistics. To mention another example, see a collection of papers on computational complexity and natural language [Barto-
ton87]. Illustrative is also a remark in [Pullum84]. Pullum first notes that there has been a strong resurgence of the idea that context-free structure grammars could be used for the description of natural languages and next observes: "Techniques straight out of programming language and compiler design may ... be of considerable interest in the context of natural language processing applications." In 1985 the book Efficient Parsing for Natural Language by M. Tomita appeared [Tomita85]. In this book a generalized LR parsing method was introduced for the (almost full) class of context-free grammars. It was argued that the algorithm performed more efficiently than any other existing parsing algorithms in practice. Since then the method has been used in natural language interfaces, machine translation systems and speech-to-speech translation systems. Hence, not only the computer science community, but also those researchers working on natural language processing have become familiar with LR parsing. Many papers on (generalized) LR parsing appeared in the proceedings of the first International Workshop on Parsing Technologies [Tomita89]. Presently the generalized LR parser developed by Tomita at Carnegie Mellon University has been made part of a software package for practical natural language processing projects. The algorithm is augmented with unification modules. The package is publicly available [Tomita90].

2 Properties of LR Grammars and Languages

LR(k) stands for 'parsing from Left to right using Rightmost reductions and k symbols of lookahead'. LR(k) grammars are usually defined by introducing conditions on the rightmost derivations of context-free grammars. LR(k) grammars allow deterministic bottom-up parsing. The productions that have been used in a rightmost derivation of a sentence are recognized in their reversed order. This is done by reducing right-hand sides of productions that appear in the sentence and intermediate right sentential forms to their left-hand sides until the start symbol has been reached. The conditions in the LR(k) definition are such that each time the position of the right-hand side and the production to be used in the reduction are unique.

Many of the formal properties of LR(k) grammars have already been discussed in Knuth's original paper. Later papers have been concerned with practical approaches to the parsing problem and more formal treatments of the properties of LR grammars and parsing. Theoretical problems are e.g. the relationships with other classes of
grammars, decidability and equivalence problems and relationships with (deterministic) pushdown automata. There exist direct, but complicated, language preserving transformations from LR(k) to LR(1) grammars. The class of LR(1) languages coincides with the class of deterministic languages, i.e. the class of languages that can be accepted by deterministic pushdown automata. LR(k) grammars are unambiguous. Many subclasses of LR(k) grammars have been introduced. In many cases this was done to obtain even more efficient parsing methods.

3 LR PARSING AND PARSER CONSTRUCTION

Automatic LR parser construction is almost always based on the following idea of DeRemer. LR(0) grammars are not always sufficient to describe programming language constructs. However, if we give an LR(0) parser the possibility of a lookahead of one symbol then most constructs can be captured. Therefore automatic construction consists of first computing the LR(0) parser and then a method is used to incorporate a lookahead of one symbol in this parser. This lookahead computation has lead to the definitions of SLR(1) and LALR(1) grammars. The construction of an LR parser is in fact the construction of a finite automaton which is used as finite control of a deterministic pushdown automaton. The pushdown automaton acts as a shift-reduce parser. The finite automaton, i.e., its states and its transition function, is constructed from the production rules of the grammar. The automaton is mostly presented as a (parse) table. In the table it can be read which actions (shift a symbol from the input on the stack or pop symbols from the stack in order to effect a reduce action) and which state transitions have to be done. The actions are determined by the current state and the lookahead.

Methods have been introduced for increasing the parser efficiency. That is, increasing the speed of the parser and reducing its size. Especially for k>1 an LR parser can require an unpractical amount of space. In fact, the number of states of an LR(k) parser can be exponential in the size of the grammar. Even for k=1 the construction of an LR(1) parser can result in disastrous space requirements. SLR(1) and LALR(1) parsers are less general than LR(1) parsers but they can be constructed from an LR(0) parser and they require much less space. Several techniques have been introduced to improve LR parser efficiency. Early research concentrated on techniques which were applied to the constructed parse table. Later research concentrated on methods which could be included in the parser construction algorithm. One method to reduce parsing time is to eliminate semantically irrelevant chain reductions from an LR parser. It is possible to do this in such a way that states are merged. Hence, also the parser size is reduced. Reduction of states in an LR parser can also be obtained by eliminating states where the only action is a reduction. The generalization of this idea has lead to the introduction of 'default reductions', reductions which are performed without checking the lookahead. There exist many techniques for reducing sparse matrices. These compression methods can be used for LR parse tables. An important question is always whether the optimization techniques leave the error detection capabilities undisturbed. In general, optimization techniques for LR parsers can lead to enormous size reductions and speed increase. Therefore they should always be applied or be included in the construction process of a parser. It is not unusual that an optimized parser is twice as fast and requires only fifty percent of the space of the original parser. Recent books in which many results on LR parsing, parser construction and optimization can be found are [Sippu88] and [Sippu90].

Although optimization and error detection and recovery issues have drawn most of the interest, there are other issues which have received attention. There are results on parallel parsing, parsing ambiguous grammars (with the help of disambiguating rules), on-line parsing, unparsing and parsing of incomplete sentences. Moreover, incremental LR parsing (e.g., [Agrawal83] and [Gafer87]) and incremental and lazy parser construction (e.g., [Fischer80] and [Horpor90]) have been studied. An incremental parser makes it possible to reshape a parse tree when its corresponding string is modified without reparing the complete input string. Incremental parser construction can be useful if a grammar is often changed, for example during the design phase of a language. After each change a new parser must be generated. In incremental parser construction the parts of the parse table that are not affected by the grammar change are re-used. In lazy parser generation the parser is generated by need while parsing the input string. Only those parts of the parser are generated that are needed by the par-
4 LR GENERALIZATIONS

Many generalizations of LR(k) grammars have been explored. Kauth mentioned the LR(k,t) grammars for which the condition is relaxed that exactly the production that has been used in a rightmost derivation should be reduced in the process of converting intermediate sentential forms until the start symbol has been reached. Others have allowed unbounded lookahead. In this case the set of all strings over the alphabet is partitioned into regular sets and by determining to which set a certain lookahead belongs parsing decisions are made. Although parsing time remains linear it becomes possible to parse nondeterministic languages. LR type grammars and parsing methods have also been introduced for non-context-free classes of grammars. Methods have been presented for subsets of the indexed grammars [Sébesta78], context-sensitive grammars [Walters70], type-0 grammars [Turnbull79], [Harris85] and two-level grammars (Van Wijngaarden grammars, W-grammars) [Wegner80]. Moreover, it has been investigated for attribute and affix grammars whether attribute evaluation can be done during LR parsing and whether attribute values can be used in guiding the parsing process [opdenAkker88].

Various authors have explored the possibility to extend the LR parsing method such that a larger subset of the context-free grammars can be parsed while retaining the simplicity and the efficiency of the LR parsing method and the parser construction. The general idea is as follows. If a grammar fails to be LR the constructed LR parse table contains multiple entries. There are shift/reduce or reduce/reduce conflicts. Nevertheless the table can be used as a nondeterministic finite control of the pushdown automaton that acts as a shift-reduce parser. This is called nondeterministic LR parsing. However, if we want to parse a general context-free grammar, rather than making a nondeterministic choice during parsing, each possible choice should be explored. Some paths in the computation will fail, other paths (more than one when the grammar is ambiguous) will be successful. The possible choices that can be made during parsing of a sentence can be displayed in a computation tree. In order to walk the paths of the computation tree different techniques can be used. In a computational environment with more than one processor different paths can be followed in parallel. For example, whenever a nondeterministic choice has to be made we can introduce new processors and each processor continues with the current contents of the parse stack, the parse table and the remaining input. Observations like these have been given in the parsing literature. They have hardly been explored. In a more general setting this nondeterministic table driven parsing has been mentioned in [Lang74]. Computer scientists were not interested in general context-free grammars and especially not in such a method since it was much less efficient than existing methods. In order to obtain an efficient version of nondeterministic LR parsing some authors introduced a bound on the nondeterminism. For example, in [Frank79] bounded nondeterministic LR parsing is studied in order to allow efficient parallel parsing. If, for a given grammar, each computation tree for any input has at most m leaves, where m is a fixed number dependent on the grammar only, then m processors can be used to parse the grammar's sentences in parallel. It can be shown that there exists an infinite hierarchy of languages, starting with the deterministic context-free languages, based on the number m. Hence, by increasing the number of processors more context-free languages can be parsed with this model. Independently from Frank nondeterministic and bounded nondeterministic LR parsing has been studied in [Schäfer80] and [Schäfer82]. Theoretical observations on classes of bounded nondeterministic LR languages have been given in [Kintača85].

In [Tomita85] multiple entries in the parse table are handled with the help of a graph-structured stack, where the graph is directed and acyclic. As we mentioned above, for each path in the computation tree it is possible to introduce separate parsers, each performing actions on its own stack. Hence, in a breadth-first search of the computation tree a list of stacks has to be maintained. Tomita's contribution to nondeterministic LR parsing is to make it deterministic by the introduction of stack combination techniques. They are based on the following observations. From a certain point, due to the situation on top of the stacks, different stacks may start to behave exactly the same. If this happens, the tops of the stacks can be unified and these stacks can be combined into a tree. From its root only one continuation needs to be represented (if the stack grows). Similarly, when a multiple entry is encountered there is no need to make copies of the
Stacks. We can as well keep the bottom portion of the stack and split it according to the multiple entries. The result of the combination of these two ideas is Tomita's graph-structured stack. In order to improve the space efficiency of the algorithm it is also necessary to have an efficient representation of all the parse trees that are constructed. This is done by introducing techniques for the sharing of subtrees and the merging of top nodes of certain subtrees. The algorithm can be based on any type of LR parse table (LR(0), SLR(1), LALR(1), LR(1), etc.). The main properties of the algorithm thus obtained are [Tomita85]:

- It is more efficient if the grammar is 'close' to LR.
- It is less efficient if the grammar is 'densely' ambiguous.
- It is not able to handle cyclic grammars.
- Empirical results show that in practical applications the algorithm is significantly more efficient than Earley's algorithm.

The similarities between Earley's algorithm and the generalized LR algorithm have been made explicit in [Sikkel90]. Tomita's ideas about the graph-structured stack were independently developed by Van der Steen [vanderSteen87]. Van der Steen extended the technique to type-1 (an extension of Walters' parsing algorithm) and type-0 grammars (an extension of Turnbull's algorithm). An application of Tomita's algorithm in computer science is reported in [Rekers92]. The application concerns the derivation of programming environments from formal language definitions. Generalized LR parsing was selected as the basis for the syntactic tools. The system allows the interactive development of syntax definitions. Grammar writers are given maximal freedom. They are not forced to ensure beforehand that the grammar parts with which they are experimenting do not lead to ambiguity. However, if a grammar is LR the constructed parser should be comparable in speed with LR parsers. The generalized LR parser which is described is based on an LR(0) parse table. Due to the application of the algorithm it became necessary to develop techniques for, among others, lazy and incremental generalized LR parser construction. Tomita's method is slightly adapted so that it can handle cyclic grammars.

5 About the Papers

In these proceedings eight invited papers presented at the first Twente Workshop on Language Technology (March 1991) can be found. The first paper, Recursive Ascent Parsing by René Leermakers, throws a completely new light on deterministic and nondeterministic LR parsing by giving a functional description of LR parsers. This formulation allows relatively simple correctness proofs for LR parsers and by using a technique from functional programming known as memo-functions this functional LR parser can parse non-LR grammars with a cubic time complexity. Due to its formulation there is no reference to a pushdown or a graph-structured stack. However, they appear implicitly as a procedure stack induced by functional programming language concepts. The results on LR parsing in this paper are extended to grammars with productions with regular expressions in the right-hand sides (extended context-free grammars) and to a functional definition of the Marcus lookahead parsers. The author claims that for non-LR grammars there is no reason to use Tomita's algorithm since his algorithm has the possible advantages of Tomita's parser for natural language grammars, while being polynomial.

In the last chapter of Tomita's book some suggestions for future work can be found. One of the suggestions concerns the extension of the generalized LR algorithm to context-sensitive grammars. Reference is made to Walters' deterministic shift-reduce parse algorithm for a subset of the context-sensitive grammars, utilizing a parse table similar to an LR parse table. Extension of the algorithm should make it possible to handle multiple entries in this parse table. In the paper A Parsing Algorithm for Nondeterministic Context-Sensitive Languages by Henk Harkema and Masaru Tomita the extension to context-sensitive grammars is explored. Walters' algorithm uses two stacks: a parse stack to keep track of the parsing process and an input buffer on which, among others, results of reduce actions are pushed. In the extended version both stacks are replaced by graph-structured stacks. The authors give a clear exposition of the method and discuss improvements of its efficiency.

In the paper Unrestricted On-line Parsing and Transduction with Graph-Structured Stacks by Gert van der Steen a comprehensive program-generator is discussed for parsing and transduc-
tion based on the generalized LR technique and suitable for grammars in the full Chomsky hierarchy and for grammar formalisms like, among others, augmented transition networks, transformational grammars and augmented context-free grammars (attribute grammars). It is expected that the linguist writes a grammar in a so-called unifying formalism, a formalism in which we recognize type-0 grammar rules and the possibility to assign variables (attributes) to nonterminal symbols. The program-generator converts the grammar into a program written in a code for an abstract machine, the Parallel Transduction Automaton (PTA), designed by the author. It can be considered as an extension of a two-stack machine, where the stacks are directed acyclic graphs in order to allow generalized LR parsing for context-free, context-sensitive and type-0 grammars. The system and its underlying LR theory has been developed in the early eighties. Recently it has been extended with an error-repair facility. Among the applications are the parsing of several natural languages, the recognition of compound words and the transformation of graphemes into phonemes.

In the next paper, Substring Parsing for Arbitrary Context-Free Grammars by Jan Rekers and Wilco Koorn, we return to context-free language parsing. The task of a substring recognizer is to determine whether a string is a substring of a sentence. A substring parser generates trees for the possible completions of the substring. In this paper a substring recognizer and parser are introduced based on the generalized LR algorithm. The important difference with a few other algorithms for substring parsing is that it works for the full class of context-free grammars and that there is no need for specially generated parse tables. The same tables as used in the original parser are used for the substring parser. Applications of substring parsing include syntax error recovery, syntax-directed editing and incremental parsing. It is not clear whether the present algorithm is sufficiently efficient to be used for practical applications.

The next two papers deal with (generalized) LR parsing algorithms in practical natural language applications. In Detection and Correction of Morpho-Syntactic Errors in Shift-Reduce Parsing Theo Vosse describes grammar-independent error detection in a shift-reduce parser for augmented context-free grammars. The errors that are considered are morpho-syntactic errors, errors that are related to the derivation and inflection of words (agreement violations, spelling and typing errors, punctuation errors, etc.). In general these errors can not be detected by spelling checking mechanisms. What is needed is a parser that handles ungrammatical input. In the paper a standard shift-reduce parser based on an LR parse table is extended so that it can handle augmented context-free grammars and it is shown how the parser helps in detecting and sometimes correcting the different types of errors. It is also discussed which adaptations have to be made to the generalized LR algorithm in order to parse augmented context-free grammars and to obtain similar error detection capabilities. The parser has been implemented in a Dutch grammar checker.

In Tomita's Algorithm in Practical Applications Rob Heemels discusses the advantages of the generalized LR algorithm for natural language applications. The paper is based on his experiences with the algorithm in a natural language parser developed by the language and speech technology group of Océ. One important requirement of a natural language processing system is the possibility to handle ungrammatical input. In the paper a classification of errors in ungrammatical sentences is given. Most of the errors caused by the user are grammatical ones, followed by style, typing and spelling mistakes. The author presents some requirements of a robust parsing algorithm, discusses properties of Tomita's generalized LR algorithm and concludes that it is not only one of the fastest and most efficient algorithms, but it has also the potential to meet most of the requirements for a robust parser.

In An Empirical Comparison of Generalized LR Tables Marc Lankhorst compares the efficiency of generalized LR parsing for LR(0), SLR(1), LALR(1) and LR(1) parse tables, respectively. The comparison is made on the basis of empirical data. Use is made of the grammars and the sentence sets of [Tomita85] which were used by Tomita to compare Earley and generalized LR parsing. From the grammars the different parse tables were constructed and their sizes were compared. With the sentence sets parsing efficiency of the different parsers was compared. It turned out that for one of the larger grammars, due to a lack of memory, the LR(1) parse table could not even be constructed. From the comparisons with respect to space the conclusions are that LR(1) tables are unsuitable, LALR(1) tables are the best choice, but if space is a constraint or
ease of construction is important, then LR(0) tables are advised. As mentioned earlier in this introduction, in [Rekers92] generalized LR parsing is based on LR(0) parse tables exactly for the reason of ease of construction. Surprisingly, perhaps, using LR(1) tables turns out to be slower than LALR(1), SLR(1) and even LR(0). This confirms a remark of Billot and Lang [Billot89], who found (in a somewhat different context) that adding sophistication to chart parsing schemata may harm, rather than improve the efficiency. Lankhorst clearly illustrates and explains this phenomenon in the context of a Tomita parser.

The final paper, *Bottom-Up Parallelization of Tomita's Algorithm*, is by Klaas Sikkels. A parallel version of the generalized LR algorithm is presented. The algorithm uses a pipeline of processors, one for each word in the sentence. Each processor (or process) runs an adapted version of the Tomita parser. Symbols are passed along the pipeline from right to left. The leftmost processor will deliver a parse of the sentence. In the paper ordering requirements for the passing of symbols are discussed. In order to prevent communication bottle-necks communication saving rules are introduced. These rules can be used by the parser to detect whether a symbol will be used by a processor further down the pipeline. If not, there is no need to pass it from one processor to another. In the paper it is announced that the algorithm will be tested against some natural language grammars.

**REFERENCES**


[Harris85] L.A. Harris. SLR(1) and LALR(1) parsing for unrestricted grammars. Mathematics Department, University of Kentucky, Lexington, Kentucky 40506, 1985.


Recursive Ascent Parsing

René Leermakers
Philips Research Laboratories,
P.O. Box 80.000, 5600 JA Eindhoven, The Netherlands
E-mail: leermake@prl.philips.nl

ABSTRACT

LR-parsing is defined in a purely functional way, without constructing a push-down automaton for each grammar. In this functional formulation, LR-parsing turns out to be analogous to recursive descent parsing. The resulting parsers have a correctness proof that is extremely simple compared to the conventional ones. For non-LR grammars the time-complexity of a functional LR-parser is cubic if the functions that constitute the parser are implemented as memo-functions, i.e. functions that memorize the results of previous invocations. Memo-functions also facilitate a simple way to construct a very compact representation of the parse forest. For LR(0) grammars, our algorithm is closely related to the recursive ascent parsers recently discovered by Kruseman Aretz [1] and Roberts [2]. Extended BNF grammars (grammars with regular expressions at the right hand side) can be parsed with a simple modification of the LR-parser for normal CF grammars. An interesting family of look-ahead LR-parsers was introduced by Marcus [3] as models for the processing of natural language in the human mind. We define the Marcus parsers in a functional way, thereby uncovering their remarkable mathematical elegance.

1 INTRODUCTION

The theory of LR-parsing is still underdeveloped, as is exemplified by the absence of proper correctness proofs of LR-parsers in most textbooks. In this paper it is shown that, instead of constructing a push-down automaton for each grammar, LR-parsers are more naturally defined using the language of functions. The functional formulation can be seen as a generalization of the well-known recursive descent parsing technique and allows a surprisingly simple correctness proof. The relation between the conventional formulation and the one proposed here, is that the stack of the push-down automaton corresponds to the stack of activated functions in the functional formulation.

For LR(0) grammars, our result implies a deterministic parser that is closely related to the recursive ascent parsers discovered by Kruseman Aretz [1] and Roberts [2]. In the general non-deterministic case, the parser has cubic time complexity if the parse functions are implemented as memo-functions [4], which are functions that memorize and re-use the results of previous invocations. Memo-functions are easily implemented in most programming languages. The notion of memo-functions is also used to define an algorithm that constructs a cubic representation for the parse forest, i.e. the collection of parse trees.

It has been claimed by Tomita that non-deterministic LR-parsers are useful for natural language processing. In ref. [5] he presented a discussion about how to do non-deterministic LR-parsing, with a device called a graph-structured stack. With our parser we show that no explicit stack manipulations are needed; they can be expressed implicitly with the use of appropriate programming language concepts.

This paper’s proof of the correctness of LR-parsing is its major contribution to parsing theory. One of its lessons is that the CF grammar class is often the natural one to proof parsers for, even if these parsers are devoted to some special class of grammars. If the grammar is restricted in some way, a parser for general CF grammars may have properties that enable smart implementation tricks to enhance efficiency. As we show below, the relation between LR-parsers and LR-grammars is of this kind.

Often, standard CF grammars are rather limited in their strong generative power. Extended BNF notation, allowing rules to have regular ex-
pressions at the right hand side, is a useful extension, for that reason. It is not difficult to generalize our parser to cope with extended grammars, although the application of LR-parsing to extended BNF grammars is well-known to be problematic [6].

Marcus [3] introduced look-ahead LR-parsers that are claimed to be akin to the processing of natural language in the human mind. Although formal descriptions and proofs of Marcus parsers do not seem to exist, the essential ideas can be captured in a functional definition of a family of parsers, along the lines of our formulation of LR(0) parsing. The ensuing mathematical elegance leads us to the conclusion that conventional look-ahead parsers are mere artifacts when compared to our version of Marcus parsers.

We first present the recursive descent recognizer in a way that allows the desired generalization. Then we obtain the recursive ascent recognizer and its proof. If the grammar is LR(0) a few implementation tricks lead to the recursive ascent recognizer of ref. [1]. Subsequently, the time and space complexities of the recognizer are analyzed, and the algorithm for constructing a cubic representation for parse forests is given. To show the extendability of the theory we next discuss how to parse grammars written in Extended BNF notation. Finally we give our formalization of Marcus’ ideas and establish a natural family of look-ahead parsers.

2 RECURSIVE DESCENT

Consider CF grammar $G$, with terminals $V_T$ and non-terminals $V_N$. Let $V = V_N \cup V_T$. A well-known top-down parsing technique is the recursive descent parser. Recursive descent parsers consist of a number of procedures, usually one for each non-terminal. Here we present a variant that consists of functions, one for each item (dotted rule). We use the unorthodox embracing operator [1] to map each item to its function. (we use greek letters for arbitrary elements of $V^*$)

\[ [A \rightarrow \alpha.\beta] : N \rightarrow 2^N \]

where $N$ is the set of integers, or a subset $0..n_{\text{max}}$, with $n_{\text{max}}$ the maximum sentence length, and $2^N$ is the powerset of $N$ The functions are defined by the following specification:

\[ [A \rightarrow \alpha.\beta](i) = \{ j | \beta \rightarrow^* x_{i+1}...x_j \} \]

with $x_1...x_n$ the sentence to be parsed. A recursive implementation for these functions is given by ($b \in V_T, B \in V_N$)

\[ [A \rightarrow \alpha](i) = \{ i \} \]

\[ [A \rightarrow \alpha.B\gamma](i) = \{ j | \gamma \in [A \rightarrow \alpha.B\gamma](i + 1) \} \]

\[ [A \rightarrow \alpha.B\gamma](i) = \{ j | k \in [B \rightarrow \delta](i) \land \gamma \in [A \rightarrow \alpha.B\gamma](k) \} \]

We keep to the custom of omitting existential quantification (here for $k, \delta$) in definitions of this kind.

The proof is elementary and based on

\[ \beta \rightarrow^* x_{i+1}...x_j \equiv \]

\[ (\beta = \epsilon \land i = j) \lor \]

\[ \exists_\gamma (\beta = x_{i+1} \land \gamma \rightarrow^* x_{i+2}...x_j) \lor \]

\[ \exists_{B \gamma \delta} (\beta = B \gamma \land B \rightarrow \delta \land \]

\[ \delta \rightarrow^* x_{i+1}...x_k \land \gamma \rightarrow^* x_{k+1}...x_j) \]

If we add a grammar rule $S' \rightarrow S$ to $G$, with $S' \notin V$ then $S \rightarrow^* x_1...x_n$ is equivalent to $n \in [S' \rightarrow .S](0)$.

The recursive descent recognizer works for any CF grammar except for grammars for which \[ \exists_{\alpha.\beta}(A \rightarrow \alpha \land \alpha \rightarrow^* \alpha.\beta) \]. For such left-recursive grammars the recognizer does not terminate, as execution of $[A \rightarrow \alpha][i]$ will lead to a call of itself. The recognition is not a linear process in general: the function calls $[A \rightarrow \alpha.B\gamma](i)$ lead to calls $[B \rightarrow \delta](i)$ for all values of $\delta$ such that $B \rightarrow \delta$ is a grammar rule.

3 THE ASCENT RECOGNIZER

The problem with left-recursion can be solved, at the same time avoiding some unnecessary non-determinism.

The mechanism for reducing non-determinism is the merging of functions corresponding to a number of competing items into one function. Let the set of all items of $G$ be given by $I_G$. Subsets of $I_G$ are called states, and we use $q$ to be an arbitrary state. We associate to each state $q$ a function, re-using the above operator [1],

\[ q : N \rightarrow 2^{I_G \times N} \]

that meets the specification

\[ q(i) = \{ (A \rightarrow \alpha.\beta, j) | A \rightarrow \alpha.\beta \in q \land \beta \rightarrow^* x_{i+1}...x_j \} \]
As above, the function reports which parts of the sentence can be derived. But as the function is associated to a set \( q \) of items, it has to do so for each item in \( q \). If we define the initial state \( q_0 = \{ S' \rightarrow S \} \), we have that \( S \rightarrow * x_1 \ldots x_n \) is equivalent to \( \{ S' \rightarrow S, n \} \in [q_0](0) \).

To be able to construct an implementation of \( [q] \) that has no problems with left-recursive grammars, we need a couple of definitions. Let \( \text{ini}(q) \) be the set of initial items for state \( q \), derived from \( q \) as the smallest solution of

\[
\text{ini}(q) = \{ B \rightarrow \nu | B \rightarrow \nu \land \ A \rightarrow \alpha.B \beta \in q \cup \text{ini}(q) \}
\]

An alternative non-recursive definition, that will be used below, is

\[
\text{ini}(q) = \{ B \rightarrow \nu | B \rightarrow \nu \land \ A \rightarrow \alpha.B \beta \in q \land \beta \Rightarrow * B \gamma \}.
\]

The double arrow \( \Rightarrow \) denotes a left-most-symbol rewriting with a non-\( \epsilon \) grammar rule, i.e.

\[
\alpha \Rightarrow \beta \equiv \exists B \gamma \delta (\alpha = B \gamma \land \beta = \delta \gamma \land B \rightarrow \delta \land \delta \neq \epsilon)
\]

The transition function \( \text{goto} \) is defined by \( (B \in V) \)

\[
\text{goto}(q, B) = \{ (A \rightarrow \alpha.B \beta) | (A \rightarrow \alpha.B \beta \in (q \cup \text{ini}(q))) \}
\]

A recognizer that has no problems with left-recursion may be obtained by relating to each state \( q \) not only the above \( [q] \), but also a function that we take to be the result of applying operator \( \lceil i \rceil \) to the state:

\[
[q] : V \times N \rightarrow 2^{I_\alpha \times X}
\]

It has the specification

\[
[q](B, i) = \{ (A \rightarrow \alpha.B \beta)(A \rightarrow \alpha.B \beta \in q \land \beta \Rightarrow * B \gamma \land \gamma \Rightarrow * x_{i+1} \ldots x_j) \}
\]

For \( i > n \) (\( n \) is the sentence length) it follows that \( [q](i) = [q](B, i) = \emptyset \), whereas for \( i \leq n \) the functions are recursively implemented by

\[
[q](i) = \left\{ \begin{array}{l}
\{ (A \rightarrow \alpha.B \beta)(A \rightarrow \alpha.B \beta \in q \land \beta \Rightarrow * B \gamma \land \gamma \Rightarrow * x_{i+1} \ldots x_j) \} \cup \\
\{ (A \rightarrow \alpha.B \beta)B \rightarrow \epsilon \in \text{ini}(q) \land (A \rightarrow \alpha.B \beta) \in [q](B, i) \} \cup \\
\{ (A \rightarrow \alpha, i)A \rightarrow \alpha. \in q \}
\end{array} \right.
\]

Proof:
First we notice that

\[
\beta \Rightarrow * x_{i+1} \ldots x_j \equiv \\
\exists \gamma (\beta \Rightarrow * x_{i+1} \gamma \land \gamma \Rightarrow * x_{i+2} \ldots x_j) \lor \\
\exists \gamma (\beta \Rightarrow * B \gamma \land B \rightarrow \epsilon \land \\
\gamma \Rightarrow * x_{i+1} \ldots x_j) \lor (\beta = \epsilon \land i = j)
\]

Hence

\[
[q](i) = \left\{ (A \rightarrow \alpha.B \beta)(A \rightarrow \alpha.B \beta \in q \land \beta \Rightarrow * B \gamma \land \gamma \Rightarrow * x_{i+1} \ldots x_j) \lor \\
(A \rightarrow \alpha.B \beta) \in [q](B, i) \lor \\
(A \rightarrow \alpha, \in q) \right\}
\]

This is equivalent to the earlier version because we may replace the clause \( B \rightarrow \epsilon \) by \( B \rightarrow \epsilon \in \text{ini}(q) \). Indeed, if state \( q \) has item \( A \rightarrow \alpha.B \beta \) then any state \( q' \) (not only \( [q] \)) that contains \( A \rightarrow \alpha.B \beta \) will have item \( A \rightarrow \alpha.B \beta \in \text{ini}(q) \) included in \( [q](i) \).

For establishing the correctness of \( [q] \) notice that \( \beta \Rightarrow * B \gamma \) either contains zero steps, in which case \( \beta = B \gamma \), or it contains at least one step:

\[
\exists \gamma (\beta \Rightarrow * B \gamma \land \gamma \Rightarrow * x_{i+1} \ldots x_j) \equiv \\
\exists \gamma (B = B \gamma \land \gamma \Rightarrow * x_{i+1} \ldots x_j) \lor \\
\exists \gamma (B = B \gamma \land \gamma \Rightarrow * B \delta \land \\
\delta \Rightarrow * x_{i+1} \ldots x_k \land \gamma \Rightarrow * x_{k+1} \ldots x_j)
\]

Hence \( [q](B, i) \) may be written as the union of two sets, \( [q](B, i) = S_0 \cup S_1 : \)

\[
S_0 = \{ (A \rightarrow \alpha.B \gamma, j)(A \rightarrow \alpha.B \gamma \in q \land \\
\gamma \Rightarrow * x_{i+1} \ldots x_j) \}
\]

\[
S_1 = \{ (A \rightarrow \alpha.B \gamma, j)(A \rightarrow \alpha.B \gamma \in q \land \\
\beta \Rightarrow * C \gamma \land C \rightarrow \beta \delta \land \\
\delta \Rightarrow * x_{i+1} \ldots x_k \land \gamma \Rightarrow * x_{k+1} \ldots x_j) \}
\]

By the definition of \( \text{goto} \), if \( A \rightarrow \alpha.B \gamma \in q \) then \( A \rightarrow \alpha.B \gamma \in \text{goto}(q, B) \). Hence, with the specification of \( [q] \), \( S_0 \) may be rewritten as

\[
S_0 = \{ (A \rightarrow \alpha.B \gamma, j)(A \rightarrow \alpha.B \gamma \in q \land \\
\gamma \Rightarrow * x_{i+1} \ldots x_j) \cup \\
(A \rightarrow \alpha.B \gamma, j)(A \rightarrow \alpha.B \gamma \in q \land \\
\gamma \Rightarrow * x_{i+1} \ldots x_j) \}
\]

\[
A \rightarrow \alpha.B \gamma, j)(A \rightarrow \alpha.B \gamma \in q \land \\
\gamma \Rightarrow * x_{i+1} \ldots x_j) \cup \\
(A \rightarrow \alpha.B \gamma, j)(A \rightarrow \alpha.B \gamma \in q \land \\
\gamma \Rightarrow * x_{i+1} \ldots x_j)
\]
The set $S_1$ may be rewritten using the specification of $\bar{q}(C, k)$:

$$S_1 = \{(A \rightarrow \alpha \beta, j) | (A \rightarrow \alpha \beta, j) \in \bar{q}(C, k) \land C \rightarrow B \delta \land \delta \rightarrow ^* x_{i+1} \ldots x_k\}.$$  

Also, as before, $\beta \Rightarrow C \gamma$ implies that all items $C \rightarrow \nu$ are in $\text{ini}(q)$, and the existence of $C \rightarrow B \delta$ in $\text{ini}(q)$ implies $C \rightarrow B \delta \in \text{goto}(q, B)$:

$$S_1 = \{(A \rightarrow \alpha \beta, j) | (A \rightarrow \alpha \beta, j) \in \bar{q}(C, k) \land C \rightarrow B \delta \in \text{ini}(q) \land (C \rightarrow B \delta, k) \in \text{goto}(q, B)(i)\}.$$  

\[\square\]

In the computation of $\bar{q}(0)$, functions are needed only for states in the canonical collection of LR(0) states \cite{7} for $C$, i.e. for every state that can be reached from the initial state by repeated application of the goto function. Note that in general the state $\emptyset$ will be among these, and that both $\emptyset(i)$ and $\emptyset(B, i)$ are empty sets for all $i \geq 0$ and $B \in V$.

The recognition functions may be cast in a form that will be convenient in the following sections, using the definitions:

$$\text{pop}(A \rightarrow \alpha B, \beta) = A \rightarrow \alpha \beta$$

$$\text{lhs}(A \rightarrow \alpha \beta) = A$$

$$\text{final}(A \rightarrow \alpha \beta) = (|\beta| = 0)$$

with $B \in V$, and $|\beta|$ the number of symbols in $\beta$ (with $|\epsilon| = 0$). Then the functions become

$$\bar{q}(i) = \bar{q}(x_{i+1}, i + 1) \cup \{(I, j) | B \rightarrow \epsilon \in \text{ini}(q) \land (I, j) \in \bar{q}(B, i)\} \cup \{(I, i) | I \in q \land \text{final}(I)\}$$

$$\bar{q}(B, i) = \{(\text{pop}(I, j), I, j) \in \text{gotto}(q, B)(i) \land \text{pop}(I) \in q\} \cup \{(I, j) | (K, j) \in \text{gotto}(q, B)(i) \land \text{pop}(I) \in \text{ini}(q) \land (I, j) \in \bar{q}(\text{lhs}(I), k)\}$$

4 Deterministic variants

One can prove that, if the grammar is LR(0), each recognizer function for a canonical LR(0) state results in a set with at most one element. The functions for non-empty $q$ may in this case be rephrased as

$$\bar{q}(i) :$$

if, for some $I, I \in q \land \text{final}(I)$

then return $\{(I, i)\}$

else if $B \rightarrow \epsilon \in \text{ini}(q)$

then return $\bar{q}(B, i)$

else if $i < n$

then return $\bar{q}(x_{i+1}, i + 1)$

else return $\emptyset$

\[\fi\]

$$\bar{q}(B, i) :$$

if $\text{gotto}(q, B)(i) = \emptyset$ then return $\emptyset$

else let $(I, j)$ be the unique element of $\text{gotto}(q, B)(i)$. Then:

if $\text{pop}(I) \in q$

then return $\{(\text{pop}(I), j)\}$

else return $\bar{q}(\text{lhs}(I), j)$

\[\fi\]

Reversely, the implementations of $\bar{q}(i)$ and $\bar{q}(B, i)$ of the previous section can be seen as non-deterministic versions of the present formulation, which therefore provides an intuitive picture that may be helpful to understand the non-deterministic parsing process in an operational way.

Each function can be replaced by a procedure that, instead of returning a function result, assigns the result to a global (set) variable. As this set variable may contain at most one element, it can be represented by three variables, a boolean $b$, an item $R$ and an integer $i$. If a function would have resulted in the set $\{(I, j)\}$, the global variables are set to $b = \text{TRUE}$, $R = I$ and $i = j$.

A function value $\emptyset$ is represented by $b = \text{FALSE}$. Also the arguments of the functions are superfluous now. The role of argument $i$ can be played by the global variable with the same name, and $\text{lhs}(R)$ can be used instead of argument $B$ of $\bar{q}$. Consequently, procedure $\emptyset$ becomes a statement $b := \text{FALSE}$, whereas for non-empty $q$ one gets the procedures (keeping the names $\bar{q}$ and $\bar{q}$, trusting no confusion will arise):

$$\bar{q} :$$

if, for some $I, I \in q \land \text{final}(I)$

then $R := I$

else if $B \rightarrow \epsilon \in \text{ini}(q)$
then \( R := B \rightarrow e; \quad [q] \)
else if \( i < n \)
then \( R := x_{i+1} \rightarrow x_{i+1}; \quad i := i + 1; \quad [q] \)
else \( b := FALSE \)
fi

\[ [q] : \]
\[ \text{goto}(q, \text{lhs}(R)) ; \]
if \( b \) then
if \( \text{pop}(R) \in q \)
then \( R := \text{pop}(R) \)
else \( [q] \)
fi
fi

Note that these procedures do not depend on the details of the right hand side of \( R \). Only the number of symbols before the dot is relevant for the test "\( \text{pop}(R) \in q \)". Therefore, \( R \) can be replaced by two variables \( X \in V \) and an integer \( i \), making the following substitutions in the previous procedures:

\[ R := A \rightarrow \alpha. \quad \Rightarrow \quad X := A; i := |\alpha| \]
\[ R := \text{pop}(R) \quad \Rightarrow \quad i := i - 1 \]
\[ \text{pop}(R) \in q \quad \Rightarrow \quad i \neq 1 \lor X = S' \]
\[ \text{lhs}(R) \quad \Rightarrow \quad X \]

After these substitutions, one gets close to the recursive ascent recognizer as it was presented in [1]. A recognizer that is virtually the same as in [1] is obtained by replacing the tail-recursive procedure \([q]\) by an iterative loop. Then one is left with one procedure for each state. While parsing there is, at each instance, a stack of activated procedures that corresponds to the stacks that are explicitly maintained in conventional implementations of deterministic LR-parsers. That it leaves this stack administration to the programming language is an important practical advantage of the recursive ascent formulation of deterministic LR-parsing, in addition to its being obviously correct.

5 Complexity

For LL(0) grammars the recursive descent recognizer is deterministic and works in linear time. The same is true of the ascent recognizer for LR(0) grammars. In the general, non-deterministic, case the recursive descent and ascent recognizers need exponential time unless the functions are implemented as memo-functions [4]. Memo-functions memorize for which arguments they have been called. If a function is called with the same arguments as before, the function returns the previous result without recomputing it.

In conventional programming languages memo-functions are not available, but they can easily be implemented. Devices like graph-structured stacks [5], parse matrices [8], or well-formed substring tables [9], are in fact low-level realizations of the abstract notion of memo-functions. The complexity analysis of the recognizers is quite simple. There are \( O(n) \) different invocations of parser functions. The functions call at most \( O(n) \) other functions, that all result in a set with \( O(n) \) elements (note that there exist only \( O(n) \) pairs \((I, j)\) with \( I \in I_q, i \leq j \leq n \)). Merging these sets to one set with no duplicates can be accomplished in \( O(n^2) \) time on a random access machine. Hence, the total time-complexity is \( O(n^2) \). The space needed for storing function results is \( O(n) \) per invocation, i.e. \( O(n^2) \) for the whole recognizer.

The above considerations only hold if the parser terminates. The recursive descent parser terminates for all grammars that are not left-recursive. For the recursive ascent parser, the situation is more complicated. If the grammar has a cyclic derivation \( B \rightarrow^* B \), the execution of \([q](B, i)\) leads to a call of itself. Also, there may be a cycle of transitions labeled by non-terminals that derive \( e \), e.g. if \( \text{goto}(q, B) = q \land B \rightarrow e \), so that the execution of \([q](i)\) leads to a call of itself. There are non-cyclic grammars that suffer from such a cycle (e.g. \( S \rightarrow SB, S \rightarrow e \)). Hence, the ascent parser does not terminate if the grammar is cyclic or if it leads to a cycle of transitions labeled by non-terminals that derive \( e \). Otherwise, execution of \([q](B, i)\) can only lead to calls of \([p](i)\) with \( p \neq q \) and to calls of \([q](C, k)\), such that either \( k > i \) or \( C \rightarrow^* B \lor C \neq B \). As there are only finitely many such \( p, C \), the parser terminates. Note that the ascent recognizer correctly terminates for any grammar, if the recognizer functions are implemented as memo-functions with the property that a call of a function with some arguments yields \( \emptyset \) while it is under execution. For instance, if execution of \([q](i)\) leads to a call of itself, the second call is to yield \( \emptyset \). For the recursive descent parser the same is true if it is written in a slightly different way (we will do so in the next section). A remark of this kind, for the recursive descent parser, was first made in ref. [9]. The recursive descent parser then becomes
virtually equivalent to a version of the standard Earley algorithm [10] that stores items $A \rightarrow \alpha \beta$ in parse matrix entry $T_{ij}$ if \( \beta \rightarrow \ast x_{i+1} \ldots x_{j} \), instead of storing it if \( \alpha \rightarrow \ast x_{i+1} \ldots x_{j} \).

The space required for a parser that also calculates a parse forest, is dominated by this forest. We show in the next section that it may be compressed into a cubic amount of space. In the complexity domain our ascent parser beats its rival, Tomita’s parsing method [5], which is non-polynomial: for each integer \( k \) there exists a grammar such that the complexity of the Tomita parser is worse than \( n^k \).

In addition to the complexity as a function of sentence length, one may also consider the complexity as a function of grammar size. It is clear that both time and space complexity are proportional to the number of parsing procedures. The number of procedures of the recursive descent parser is proportional to the number of items, and hence a linear function of the grammar size. The recursive ascent parser, however, contains two functions for each LR-state and is hence proportional to the size of the canonical collection of LR\((0)\) states. In the worst case, this size is an exponential function of grammar size, but in the average natural language case there seems to be a linear, or even sublinear, dependence [5].

6 Parse forest

Usually, the recognition process is followed by the construction of parse trees. For ambiguous grammars, it becomes an issue how to represent the set of parse trees as compactly as possible. Below, we describe how to obtain a cubic representation in cubic time. We do so in three steps.

In the first step, we observe that ambiguity often arises locally: given a certain context \( C[\cdot] \), there might be several parse subtrees \( t_1 \ldots t_k \) (all deriving the same substring \( x_{i+1} \ldots x_j \) from the same symbol \( A \)) that fit in that same context, leading to the parse trees \( C[t_1], C[t_2], \ldots, C[t_k] \) for the given string \( x_1 \ldots x_n \). Instead of representing these parse trees separately, repeating each time the context \( C \), we can represent them collectively as \( C[t_1, \ldots, t_k] \). Of course, this idea should be applied recursively. Technically, this leads to a kind of tree-like structure in which each child is a set of substructures rather than a single one.

The sharing of context can be carried one step further. If we have, in one and the same context, a number of applied occurrences of a production rule \( A \rightarrow \alpha \beta \) which share also the same parse forest for \( \alpha \), we can represent the context of \( A \rightarrow \alpha \beta \) itself and the common parse forest for \( \alpha \) only once and fit the set of parse forests for \( \beta \) into that. Again this idea has to be applied recursively. Technically, this leads to a binary representation of parse trees, with each node having at most two sons, and to the application of the context sharing technique to this binary representation.

These two ideas are captured by introducing a function \( f \) with the interpretation that \( f(\beta, i, j) \) represents the parse forest of all derivations from \( \beta \in V^* \) to \( x_{i+1} \ldots x_j \), for all \( i, j \) such that \( 0 \leq i \leq j \leq n \). The following recursive definitions fix the parse forest representation formally:

\[
f(\varepsilon, i, j) = \{[] | i = j\},
\]

\[
f(a, i, j) = \{a | j = i + 1 \land x_{i+1} = a\},
\]

for all \( a \in V_T \),

\[
f(A, i, j) = \{(A, f(\alpha, i, j)) | A \rightarrow \alpha \land \alpha \rightarrow \ast x_{i+1} \ldots x_j\},
\]

for all \( A \in V_N \),

\[
f(AB\beta, i, j) = \{(f(A, i, k), f(B\beta, k, j)) | i \leq k \leq j \land A \rightarrow \ast x_{i+1} \ldots x_k \land B \land \beta \rightarrow \ast x_{k+1} \ldots x_j\},
\]

for all \( A, B \in V \).

The representation for the set of parse trees is then just \( f(S, 0, n) \).

We now come to our third step. Suppose, for the moment, that the guards \( \alpha \rightarrow \ast x_{i+1} \ldots x_j \) and the like, occurring above, can be evaluated in some way or another. Then we can use function \( f \) to compute the representation of the set of parse trees for sentence \( x_1 \ldots x_n \). If we make use of memo-functions to avoid repeated computation of a function applied to the same arguments, we see that there are at most \( O(n^2) \) function evaluations. If we represent function values by references to the set representations rather than by the sets themselves, the most complicated function evaluation consumes an additional amount of storage that is \( O(n) \): for \( j - i + 1 \) values of \( k \) we have to perform the construction of a pair of (copies of) two references, costing a unit amount of storage each. Therefore, the total amount of
space needed for the representation of all parse trees is $O(n^2)$.

The evaluation of the guards $\alpha \rightarrow x_{i+1} \ldots x_j$ etc. amounts exactly to solving a collection of recognition problems. A top-down parser is possible that merges the recognition and tree-building phases, by writing

$$f(A, i, j) =$$

$$\begin{cases}
\{(A, f(\alpha, i, j)) | A \rightarrow \alpha \land f(\alpha, i, j) \neq \emptyset\}, & \text{for all } A \in V_N,
\end{cases}$$

$$f(AB\beta, i, j) =$$

$$\begin{cases}
\{(f(A, i, k), f(B\beta, k, j)) | i \leq k \leq j \land 
\quad f(A, i, k) \neq \emptyset \land f(B\beta, k, j) \neq \emptyset\}, & \text{for all } A, B \in V,
\end{cases}$$

the other cases for $f$ being left unchanged. Note the similarity between the recognizing part of this algorithm and the descent recognizer of section 2. Again, this parser is a cubic algorithm if we use memo-functions.

Another approach is to apply a bottom-up recognizer first and derive from it a set $P$ containing triples $(\beta, i, j)$ only if $\beta \rightarrow^* x_{i+1} \ldots x_j$, and at least those triples $(\beta, i, j)$ for which the guards $\beta \rightarrow x_{i+1} \ldots x_j$ are evaluated during the computation of $f(S, 0, n)$ (i.e., for each derivation $S \rightarrow^* x_1 \ldots x_k \alpha x_{j+1} \ldots x_n \rightarrow^* x_1 \ldots x_k \beta x_{j+1} \ldots x_n \rightarrow^* x_1 \ldots x_n$, the triples $(\beta, i, j)$ and $(A, k, j)$ should be in $P$). The simplest way to obtain such $P$ from our recognizer is to assume an implementation of memo-functions that enables access to the memoized function results, after executing $[q_0](0)$. Then one has the disposal of the set

$$\{(\beta, i, j) | [q](i) \text{ was invocated and } (A \rightarrow \alpha, \beta, j) \in [q](i)\}$$

Clearly, $(\beta, i, j)$ is only in this set if $\beta \rightarrow^* x_{i+1} \ldots x_j$. Note, however, that no pairs $(A \rightarrow \beta, j)$ are included in $[q](i)$ (except if $A = S'\)$. We remedy this with a slight change of the specifications of $[q]$ and $[q]$, defining $q \equiv q \cup \text{init}(q)$:

$$[q](i) = \{(A \rightarrow \alpha, \beta, j) | A \rightarrow \alpha, \beta \in q \land 
\quad \beta \rightarrow^* x_{i+1} \ldots x_j\}$$

$$[q](B, i) = \{(A \rightarrow \alpha, \beta, j) | A \rightarrow \alpha, \beta \in q \land 
\quad \beta \rightarrow^* B_\gamma \land \gamma \rightarrow^* x_{i+1} \ldots x_j\}$$

A recursive implementation of the recognition functions now is

$$\begin{align*}
[q](i) &= \{(I, j) \in [q]|x_{i+1}, i + 1}\} 
\quad I, j \in [q](B, i) \cup 
\quad \{(I, i) | I \in \emptyset \land \text{final}(I)\}
\end{align*}$$

$$\begin{align*}
[q](B, i) &= \{(\text{pop}(I, j))(I, j) \in [q]|x_{i+1}, i + 1}\} 
\quad I, j \in [q](B, i) \cup 
\quad \{(I, i) | I \in \emptyset \land \text{final}(I)\}
\end{align*}$$

If we define, for this revised recognizer,

$$P = \{(\beta, i, j) | [q](i) \text{ was invocated and } (A \rightarrow \alpha, \beta, j) \in [q](i)\} \cup 
\quad \{(A, i, j) | [q](i) \text{ was invocated and } (A \rightarrow \beta, j) \in [q](i)\} \cup 
\quad \{(x_{i+1}, i + 1) | 0 \leq i < n\},$$

it contains all triples that are needed in $f(S, 0, n)$, and we may write the forest constructing function as

$$\begin{align*}
f(A, i, j) &= \{(A, f(\alpha, i, j)) | A \rightarrow \alpha \land (\alpha, i, j) \in P\}, 
\quad \text{for all } A \in V_N,
\end{align*}$$

$$\begin{align*}
f(AB\beta, i, j) &= \{(f(A, i, k), f(B\beta, k, j)) | (A, i, k) \in P \land 
\quad (B\beta, k, j) \in P\}, 
\quad \text{for all } A, B \in V,
\end{align*}$$

the other cases for $f$ being left unchanged again. There exists a representation of $P$ in quadratic space such that the presence or absence of an arbitrary triple can be decided upon in unit time. As a result, the time complexity of $f(S, 0, n)$ is cubic.

7 **Extended BNF Grammars**

An extended BNF grammar consists of grammar rules with regular expressions at the right hand side. Every extended BNF grammar can be translated into a normal CF grammar by replacing each right hand side by a regular (sub)grammar. The strong generative power is different from CF grammars, however, as the degree of the nodes in a derivation tree is unbounded. To apply our recognizer directly to extended BNF grammars, a few of the foregoing definitions have to be revised.
As before, a grammar rule is written \( A \to \alpha \), but with \( \alpha \) now a regular expression with \( N_\alpha \) symbols (elements of \( V \)). Defining \( T_\alpha^+ = 1...N_\alpha \) and \( T_\alpha = 0...N_\alpha \), regular expression \( \alpha \) can be characterized by

1. a mapping \( \phi_\alpha : T_\alpha^+ \to V \) associating a grammar symbol to each number.
2. a function \( \text{succ}_\alpha : T_\alpha \to 2T_\alpha^+ \) mapping each number to its set of successors. The regular expression can start with the symbols corresponding to the numbers in \( \text{succ}_\alpha(0) \).
3. a set \( \sigma_\alpha \in 2^{T_\alpha} \) of numbers of symbols the regular expression can end with.

We refer to \( T_\alpha \) as the set of nodes. Note that node 0 is not associated to a symbol in \( V \) and is not a possible element of \( \text{succ}_\alpha(k) \). It can be element of \( \sigma_\alpha \) though, in which case there is an empty path through the regular expression.

We define an item as a pair \((A \to \alpha, k)\), with the interpretation that number \( k \) is 'just before the dot'. The correspondence with dotted rules is the following: Let \( \alpha = B_1...B_i \), then \( \alpha \) is a simple regular expression characterized by \( \phi_\alpha(k) = B_k \), \( \text{succ}_\alpha(k) = \{k + 1\} \) if \( 0 \leq k < i \), \( \text{succ}_\alpha(0) = \emptyset \), and \( \sigma_\alpha = \{i\} \). Item \((A \to \alpha, 0)\) corresponds to the initial item \( A \to \alpha \) and \((A \to \alpha, k)\) to the dotted-rule item with the dot just after \( B_k \).

The predicate \( \text{final} \) for the new kind of items is defined by

\[
\text{final}(A \to \alpha, k) = (k \in \sigma_\alpha)
\]

Given a set \( q \) of items, we define

\[
\text{ini}(q) = \{(A \to \alpha, 0)| (B \to \beta, l) \in q \land k \in \text{succ}_\alpha(l) \land \phi_\alpha(k) \Rightarrow A \gamma \}
\]

The function \( \text{pop} \) becomes set-valued and the transition function can be defined in terms of it (remember: \( \bar{q} = q \cup \text{ini}(q) \)):

\[
\text{pop}(A \to \alpha, l) = \{(A \to \alpha, k)| l \in \text{succ}_\alpha(k) \}
\]

\[
\text{goto}(q, B) = \{(A \to \alpha, k)| \phi_\alpha(k) = B \land I \in \bar{q} \land I \in \text{pop}(A \to \alpha, k) \}
\]

A recursive ascent recognizer is now implemented by

\[
[q](i) =
\]

\[
[q](x_{i+1}, i + 1) \cup
\]

\[
\{(I, j)| I \in \text{ini}(q) \land \text{final}(J) \land (I, j) \in [q](\text{lhs}(J), i)\} \cup
\]

\[
\{(I, j)| I \in q \land \text{final}(I)\}
\]

\[
[q](B, i) =
\]

\[
\{(J, j)| J \in q \land J \in \text{pop}(I) \land (I, j) \in \text{goto}(q, B)(i)\} \cup
\]

\[
\{(I, j)| (J, k) \in \text{goto}(q, B)(i) \land \text{ini}(q) \land \text{pop}(J) \neq \emptyset \land (I, j) \in [q](\text{lhs}(J), k)\}
\]

The initial state \( q_0 = \{(S' \to S, 0)\} \), and a sentence \( x_1...x_n \) is grammatical if \((S' \to S, 0, n) \in [q_0](0)\). The recognizer is deterministic if

1. there is no shift-reduce or reduce-reduce conflict, i.e. every state has at most one final item, and in case \( q \) has a final item, \( q \) has no items \((A \to \alpha, j)\) with \( k \in \text{succ}_\alpha(j) \land \phi_\alpha(k) \in V_T \).
2. for all reachable states \( q \) and for all items \( I \) there is at most one \( J \in \bar{q} \) such that \( J \in \text{pop}(I) \).

In the deterministic case, the analysis of section 4 can be repeated with one exception: extended grammar items can not be represented by a non-terminal and an integer that equals the number of symbols before the dot, as this notion is irrelevant in the case of regular expressions. In standard presentations of deterministic LR-parsing of extended BNF grammars this leads to almost unsurmountable problems [6].

The parse forest can be created as before, on the basis of a small variation of the above parser:

\[
[q](i) =
\]

\[
[q](x_{i+1}, i + 1) \cup
\]

\[
\{(I, j)| I \in \text{ini}(q) \land \text{final}(J) \land (I, j) \in [q](\text{lhs}(J), i)\} \cup
\]

\[
\{(I, j)| I \in \bar{q} \land \text{final}(I)\}
\]

\[
[q](B, i) =
\]

\[
\{(J, j)| (I, j) \in \text{goto}(q, B)(i)\} \cup
\]

\[
\{(I, j)| (J, k) \in \text{goto}(q, B)(i) \land \text{ini}(q) \land \text{pop}(J) \neq \emptyset \land (I, j) \in [q](\text{lhs}(J), k)\}
\]

Define the set

\[
P = \{(A \to \alpha, i, j)| [q](i) \text{ was invoked and } ((A \to \alpha, k, j) \in [q](i)) \cup
\]

\[
\{(A, i, j)| [q](i) \text{ was invoked and } ((A \to \alpha, 0, j) \in [q](i)) \cup
\]

\[
\{(x_{i+1}, i, i + 1)| 0 \leq i < n\},
\]
and functions \( f \) and \( g \) by

\[
f(a, i, j) = \{ a | j = i + 1 \land a_{i+1} = a \}
\]

\[
f(A, i, j) = \{(A, g((\alpha, 0), i, j)) | A \rightarrow \alpha \land ((\alpha, 0), i, j) \in P \}
\]

\[
g((\alpha, m), i, j) = \{(f(\phi_0(l), i, k), g((\alpha, l), k, j)) | \ell \in \text{succ}_a(m) \land (\phi_0(l), i, k) \in P \land ((\alpha, l), k, j) \in P \} \cup \{[i] | i = j \land m \in \sigma_a \}
\]

As before, \( f(S, 0, n) \) produces a compact parse forest, if \( f \) and \( g \) are properly memoized. The pair \((\alpha, k)\) stands for the regular expression that is obtained from \( \alpha \) by taking node \( k \) as the initial one instead of 0. Two pairs \((\alpha_1, k_1)\) and \((\alpha_2, k_2)\) are equivalent if they reduce to equivalent regular expressions in this way. It may be profitable to identify equivalent pairs \((\alpha_1, k_1)\) and \((\alpha_2, k_2)\) in the construction of \( P \) and, subsequently, in the memoization of \( f \) and \( g \).

8 Marcus parsers

Marcus [3] has suggested a family of look-ahead parsers that should mimic the processing of natural language by humans. In particular, natural language grammars should be such that for some look-ahead the Marcus parser is deterministic. Here we are not interested in such claims about natural language but focus on the main ideas of the parser itself. A problem is that Marcus parsers have not yet been formulated very accurately, although an attempt has been made in ref. [11]. The family of recognizers defined below is a formalization of Marcus's ideas about look-ahead. A formalization of non-formal ideas is rarely unique however, and we may have found one that deviates slightly from Marcus' own intentions. From the mathematical point of view, however, our family of parsers seems to be the natural one.

In the following we use a few new notations. By \( k : \alpha \) we denote the \( k \)-prefix of \( \alpha \), with \( k \) a natural number. It is defined as follows. If \( \alpha = \varepsilon \) or \( k = 0 \) then \( k : \alpha = \varepsilon \). If \( \alpha \neq \varepsilon \) then 1 : \( \alpha \) is the first symbol of \( \alpha \). More generally, if \( \alpha \neq \varepsilon \) and \( k > 0 \),

- if \( 1 : \alpha \in V_T \) then \( k : \alpha = 1 : \alpha \).
- if \( 1 : \alpha \in V_N \) then

  if \( k > |\alpha| \) then \( k : \alpha = \alpha \)

  otherwise \( k : \alpha \) is the prefix of \( \alpha \) with length \( k \)

If the prefix \( k : \alpha \) is removed from \( \alpha \), one is left with a postfix referred to as \( \alpha : k \). We take the prefix and postfix operations to bind less tightly than concatenation. For instance, \( k : \alpha \beta \) means the prefix of \( \alpha \beta \).

The new family of parsers is based on a generalization of the notion of states. Whereas previously a state was a set of dotted grammar rules, it now becomes a set of objects \( \gamma \rightarrow \alpha \beta \), with \( \gamma \in V^* \) such that \( 1 : \gamma \) rewrites in one step to a prefix of \( \alpha \beta \). Correspondingly, we generalize some basic functions. Firstly, \( \text{ini}(q) \) is (re)defined to be the smallest solution of

\[
\text{ini}(q) = \{ B\mu \rightarrow \nu, B \rightarrow \nu \land \gamma \rightarrow \alpha \beta \in (q \cup \text{ini}(q)) \land B\mu = k : \beta \}
\]

for some \( k > 0 \). A non-recursive definition is also possible using a new kind of rewriting: instead of elements of \( V^* \times V^* \), we rewrite elements of \( V^* \times V^+ \), with a family of rewriting relations, denoted by \( \triangleleft \), with \( k \) a positive integer number. Their definition is

\[
(A\beta, \gamma) \triangleleft (k : \alpha \beta, k \gamma)
\]

whenever \( A \rightarrow \alpha \) is a grammar rule. The above function \( \text{ini} \) is definable in terms of \( \triangleleft \):

\[
\text{ini}(q) = \{ B\mu \rightarrow \nu, B \rightarrow \nu \land \gamma \rightarrow \alpha \beta \in q \land (k : \beta, k) \triangleleft (B\mu, \nu) \}
\]

Note that \( A\alpha \rightarrow B\beta \equiv (A, \alpha) \triangleleft (B, \beta) \) and both definitions of \( \text{ini}(q) \) come down to the corresponding ones in section 3, if one takes \( k = 1 \). The function \( \text{goto} \) has to be generalized as well, turning its second argument into an element of \( V^* \) instead of \( V \):

\[
\text{goto}(q, \delta) = \{ \gamma \rightarrow \alpha \delta, \beta \rightarrow \alpha \beta \in (q \cup \text{ini}(q)) \land \delta = k : \delta \}
\]

Now consider recognition functions

\[
[q](i) = \{ (\gamma \rightarrow \alpha \beta, j) | \gamma \rightarrow \alpha \beta \in q \land \beta \rightarrow^* x_{i+1} ... x_j \}
\]
\([q](\delta, i) = \{(\gamma \rightarrow \alpha, \beta, j) | \gamma \rightarrow \alpha, \beta \in q \land \\
(\beta: \beta, k) \xrightarrow{\delta, *}(\delta, \lambda) \land \\
\lambda \rightarrow \ast x_{i+1} \ldots x_{n}\}\}

For look-ahead parsers, it is customary to introduce a marker that signals the end of the input. We take \(\perp\) for this marker, i.e. we require \(x_{n+1} = \perp\). Then, if we define the initial state by \(q_0 = \{S' \rightarrow .S \perp\}\), one has that \((S' \rightarrow .S, \perp, n + 1) \in [q_0](0)\) is equivalent to \(S \rightarrow \ast x_1 \ldots x_n\).

As the specifications of the recognition functions differ only slightly from the ones of section 3, it will not come as a surprise that they can be implemented similarly:

\([q](i) = \\
\{(\gamma \rightarrow \alpha, \beta, j) | \gamma \rightarrow \alpha, \beta, j \in [q](x_{i+1}, i + 1) \} \cup \\
\{(\gamma \rightarrow \alpha, \beta, j) | B \rightarrow \epsilon \in \text{init}(q) \land \\
(\gamma \rightarrow \alpha, \beta, j) \in [q](B, i) \} \cup \\
\{(\gamma \rightarrow \alpha, i) | \gamma \rightarrow \alpha \in q \}
\]

\([q](0, i) = \\
\{(\gamma \rightarrow \alpha, \delta, \beta, j) | \gamma \rightarrow \alpha, \delta, \beta \in q \land \\
(\gamma \rightarrow \alpha, \delta, \beta, j) \in [goto(q, \delta)](i) \} \cup \\
\{(\gamma \rightarrow \alpha, \delta, \beta, j) | \gamma \rightarrow \alpha, \beta, j \in [q](\mu, i) \land \\
\mu \rightarrow \delta \nu \in \text{init}(q) \land \\
(\mu \rightarrow \delta \nu, i) \in [goto(q, \delta)](i)\}
\]

Proof:
The proof is similar to the one in section 3. The correctness of \([q]\) follows directly from

\[\begin{align*}
\beta & \rightarrow \ast x_{i+1} \ldots x_j \equiv \\
\exists_{\gamma}(k: \beta, \beta, k) & \xrightarrow{\delta, *}(x_{i+1}, \gamma) \land \\
\gamma & \rightarrow \ast x_{i+2} \ldots x_j \lor \\
\exists_{B:\gamma}(k: \beta, \beta, k) & \xrightarrow{\epsilon, *}(B, \gamma) \land B \rightarrow \epsilon \land \\
\gamma & \rightarrow \ast x_{i+1} \ldots x_j \lor \\
(\beta & = \epsilon \land i = j)
\end{align*}\]

To verify the correctness of \([q]\) note that

\[\begin{align*}
\exists_{\lambda}(\beta_1, \beta_2) & \xrightarrow{\delta, \lambda}(\delta, \lambda) \land \lambda \rightarrow \ast x_{i+1} \ldots x_j \equiv \\
(\beta_1 = \delta \land \beta_2 \rightarrow \ast x_{i+1} \ldots x_j) \lor \\
\exists_{\mu, \lambda, \nu}(\beta_1, \beta_2) & \xrightarrow{\mu, \lambda, \nu}(\mu, \lambda) \land \\
(\mu, \lambda) & \xrightarrow{\delta, \nu, \lambda}(\delta, \nu, \lambda) \land \\
\nu & \rightarrow \ast x_{i+1} \ldots x_1 \land \lambda \rightarrow \ast x_{i+1} \ldots x_j)
\]

Now if \(\gamma \rightarrow \alpha, \beta \in q\) we need this equivalence for \(\beta_1 = k: \beta\) and \(\beta_2 = \beta: k\). If \(k: \beta = \delta\) then \(\gamma \rightarrow \alpha, \beta, j \in goto(q, \delta)\) and \(\gamma \rightarrow \alpha, \beta, j \in \{ goto(q, \delta) \}(i) \) iff \(\beta \rightarrow \ast x_{i+1} \ldots x_j\). If, alternatively, \((k: \beta, \beta, k) \xrightarrow{\delta, *}(\delta, \nu, \lambda) \land \mu \rightarrow \delta \nu \in \text{init}(q)\) with \(\delta = k: \delta\nu\). Therefore, \(\mu \rightarrow \delta \nu \in goto(q, \delta)\) and \((\mu \rightarrow \delta \nu, i) \in \{ goto(q, \delta) \}(i) \) iff \(\nu \rightarrow \ast x_{i+1} \ldots x_j\). Finally, observe that \(\gamma \rightarrow \alpha, \beta, j \in [q](\mu, i) \) iff \(\lambda \rightarrow \ast x_{i+1} \ldots x_j\).

The fundamental reason for having the new states is that the items have longer right hand sides, so that it will occur less often that the right hand side of some item is a prefix of another one in the same state. As a consequence, the parser suffers from fewer shift-reduce and reduce-reduce conflicts and is deterministic for more grammars, possibly including natural language grammars. If it is deterministic, a number of implementation techniques may be applied, in the spirit of section 4. The look-ahead size \(k\) can be tuned to the grammar and may vary from state to state. Choosing \(k = 1\) for every state one recovers the parser of section 3. Whereas in LR(k) parsers the look-ahead consists of \(k\) terminals, with \(k\) fixed, in the Marcus parser it consists of at most \(k - 1\) elements of \(V\). This in general corresponds to an unbounded look-ahead in terms of terminals. For any value of \(k\), however, the Marcus parser look-ahead may be 0 elements of \(V\) for some reductions. Also, when there are \(\epsilon\)-rules, an element of \(V\) may derive 0 terminals. Hence, a finite look-ahead in terms of non-terminals may vanish in terms of terminals. It is therefore difficult to compare LR(k) parsers and Marcus parsers exactly. An interesting subject for future research would be to characterize the class of grammars that can be parsed deterministically with a Marcus parser.

9 Conclusions

The functional approach to LR-parsing provides a high-level view on the subject compared to the standard theory. The elimination of the explicit push-down automaton simplifies proof obligations enormously. Nevertheless, the functional implementation is as efficient as conventional ones. Also, the notion of memo-functions is an important primitive for presenting algorithms at a level of abstraction that can not be achieved without them, as is exemplified by this paper's presentation of both the recognizers and the parse forests.

For non-LR grammars, there is no reason to
use the complicated Tomita algorithm. If indeed non-deterministic LR-parsers beat the Earley algorithm for some natural language grammars, as claimed in [5], this is because the number of LR(0) states may be smaller than the size of $I_0$ for such grammars. Evidently, for the grammars examined in [5] this advantage compensates the loss of efficiency caused by the non-polynomiality of Tomita’s algorithm. The present algorithm seems to have the possible advantage of Tomita’s parser, while being polynomial.

Marcus look-ahead parsers as formulated in this paper are so natural from the point of view of LR-parsing that one would really hope that they will indeed prove to be natural in the linguistic sense as well. In any case, this paper’s formalization of Marcus’ ideas should be helpful to the linguistic application. It follows from what we have found, however, that linguistic inquiries may lead to advances in computer science as well.

ACKNOWLEDGEMENT

A considerable part of this research was done in collaboration with Lex Augusteijn and Frans Kruseman Aretz. I thank Joep Rous and Paul Gorissen for their suggestions to improve the presentation. All are colleagues at Philips Research.

REFERENCES


A parsing algorithm for non-deterministic context-sensitive languages

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

Masaru Tomita
Center for Machine Translation, Carnegie Mellon
University, Pittsburgh, PA 15213-3890, USA
email: mt@nl.cs.cmu.edu

ABSTRACT

In this paper a shift-reduce algorithm for the recognition of sentences of non-deterministic languages defined by context-sensitive grammars is introduced. The algorithm is a composite of Tomita’s efficient parsing method for non-deterministic languages and a procedure for processing deterministic context-sensitive languages, proposed by Walters. The algorithm that is presented in this paper employs two (graph-structured) stacks, one of which is used for representing the context of the several active processes (an extension of the traditional input buffer). The other stack is the normal parsing stack. The two stacks are linked by means of a set of pointers. When manipulating context-sensitive languages in an LR-manner, there is no need for a separate goto table anymore. In case of a reduce action, all the symbols of the left-hand side of the rule under consideration are pushed back into the input buffer. The state which appears on top of the stack when having popped off the symbols of the right-hand side, and the first symbol in the input buffer determine the next action.

1 INTRODUCTION

1.1 Context-sensitive grammars and languages

Before we jump into the basic operations of the algorithm, we have to define what exactly a context-sensitive grammar and ditto language is. According to Walters [12], a context-sensitive grammar (CS grammar) is a quadruple \( \{ V_T, V_N, S, P \} \), where \( V_T \) is a finite set of terminal symbols, \( V_N \) is a finite set of non-terminal symbols, \( S \in V_N \) is the distinguished or start symbol, and \( P \) a finite set of rules. The intersection of the sets \( V_T \) and \( V_N \) is the empty set, the union of the sets \( V_T \) and \( V_N \) is called the vocabulary \( V \). The \( p \)-th rule is denoted \( Y_{p1}, Y_{p2}, \ldots, Y_{pm}, \rightarrow X_{p1}, X_{p2}, \ldots, X_{pn}, \) where \( 1 \leq m_p \leq n_p, Y_{p1}, X_{p1} \in V \) and \( Y_{p1}, \ldots, Y_{pm} \not\in V_T^* \) (the set of all nonnull strings over \( V_T \)). If \( m_p \geq 2 \), rule \( p \) is a context-sensitive rule; if \( m_p = 1 \), rule \( p \) is a context-free rule. \( Y_{p1}, \ldots, Y_{pn} \) is the subject or left-hand side of rule \( p \) and \( X_{p1}, \ldots, X_{pn} \) is its right-hand side.

The set of all sentences generable by a CS grammar \( G \) forms the context-sensitive language defined by \( G \). A context-free grammar is a CS grammar all of whose rules are context-free or are of the form \( A \rightarrow \lambda \) (the null string) for \( A \in V_N \). If a context-free grammar contains no rules which have the null string as right-hand side, it is clearly also a CS grammar.

1.2 Tomita’s parsing algorithm

The Tomita parsing algorithm adapts Knuth’s well-known parsing algorithm for LR(k) grammars to non-LR grammars, including ambiguous grammars ([8], [9], [10], [11]). If a grammar is ambiguous then at some point in the analysis of an ambiguous sentence two different parsing actions must be possible that lead to the two distinct analyses of the sentence. The parsing table
of such a grammar will therefore contain multiple entries. Knuth's algorithm is not capable of dealing with this kind of non-determinism. Tomita's algorithm is an extension of Knuth's method, that can handle parsing tables with multiple entries.

When a parsing process in Tomita's algorithm encounters a multiple action, the stack is split, thus creating a new process for each entry. Because the first part of the stack of the new process is exactly the same as the original stack, it is not necessary to copy the whole stack. When a stack is split, the stack is therefore represented as a tree, the bottom of the stack corresponding to the root of the tree.

Whenever two or more processes have the same state number on top of their stacks, they will behave in exactly the same manner until the tops will be popped by a reduce action. It is obvious that these processes should be combined by unifying their top nodes.

With these stack splitting and combination techniques, the stack becomes an directed acyclic graph called a graph-structured stack. Each path through the graph from the joint node representing the start state, to whichever end node delineates an ordinary LR parse stack.

1.3 Walters' parsing algorithm

Walters also takes Knuth's algorithm as a base and adjusts it to handle context-sensitive grammars [12]. The CS(k) processor is basically the same as an LR(k) processor: a one-way input tape, a stack to keep track of the parsing process, an LR-table and a control mechanism that uses the table to guide the process. The LR-table normally consists of an action part and a goto part. The goto part is used to conclude what state to push on top of the stack after a reduce action. The action part contains the actions (shift, reduce or accept) to be performed on the stack. In a reduce action a number of states is popped off the stack. The number of states that is popped off is equal to the length of the right-hand side of the rule used in the reduction. After a reduce action, the entry of the goto part of the table corresponding to the new topmost state on the stack and the (non-terminal) symbol of the left-hand side of the rule gives the new state.

The way a CS(k) processor handles a reduce action differs slightly, but decisively, from the LR(k) approach. In addition to popping a number of states from the stack, the left-hand symbols of the rule that is used in the reduction are pushed on the input tape. This makes the goto part of the action table superfluous, because now the next action can simply be determined by looking up the appropriate entry in the action part, according to the new state on top of the stack and the first symbol on the input tape (which is the first symbol of the left-hand side of the used rule). Since this rule can be context-sensitive, this symbol is possibly a terminal symbol.

Using Walters' method we can picture the parsing process as a constant exchange of symbols between the stack and the input buffer. Hence, the input buffer acts as a second stack.

2 THE BASIC OPERATIONS

The configuration of the recognizer comprises two stacks and a set of so-called pointer elements. One of the stacks is Tomita's graph-structured stack, the other is Walters' extended input buffer, which likewise is a graph-structured stack. Stack \( \Gamma_r \), formerly Tomita's stack, is the one the symbols of the input buffer are pushed onto. In all the following figures \( \Gamma_r \) is depicted on the left. The input buffer, which is now a stack in its own right, is called \( \Gamma_v \). When a reduce action is performed, the symbols of the left-hand side of the rule used in the reduction are added to this stack. In the figures \( \Gamma_r \) is on the right. In order to link each process represented in \( \Gamma_r \), with its own set of contexts in \( \Gamma_v \), a set of pointer elements \( \Pi \) is created. Each element of \( \Pi \), called pointer element or context marker, points to one process on the left and to one or more contexts on the right. The pointer elements are depicted between the two stacks, as asterisks within circles.

Besides \( \Pi \), there are three more sets playing an important role in the algorithm: \( A \), \( S \) and \( \Sigma \). Set \( A \) is the set of active triples. An active triple is an triple for which the control mechanism has to decide what the next action is that should be performed on the process the triple is representing. If the action is shift, a triple containing information about the action is added to \( S \). The shift procedure uses these triples in \( S \) to actually execute the shift actions. In case of a reduce action a triple is added to \( \Sigma \). This set is used by the reduce procedure.

Using these sets enables the algorithm to develop the two stacks and pointer set by merely adding vertices, edges and elements to them.
There are no removals of vertices or edges, nor deletions of pointer elements. As a result we do not really pop away items from the stack; we leave the items and mark them inactive by removing triples from $A$, as in Tomita's algorithm [11].

### 2.1 Initial configuration

At the very beginning the input stack $\Gamma_0$ has a vertex for each symbol of the input string. Furthermore a vertex labeled $\# \equiv 0$ is created in $\Gamma_0$. This vertex demarcates the end of the input string. Stack $\Gamma_0$ simply consists of one single vertex, an initial state vertex labeled $0$. The two stacks are linked by a pointer element of set $\Pi$. Figure 1 gives the initial framework for the recognizer if the input string is $e c f$.

![Figure 1: initial situation of the stacks and the pointer set](image)

The first recognizer process is started up by adding triple $(v, \sigma, l)$ to the set of active triples $A$. The control mechanism takes over and determines what action is next.

### 2.2 Shift

A shift action is characterized by a triple $(\sigma, s, l)$, where $\sigma$ is a pointer element, $s$ is the state to go to after the shift action and $l$ is the vertex in $\Gamma_0$ to be shifted. Figure 2 and 3 which portray parts of the stacks and the pointer set before and after a shift action, will illustrate the operation.

![Figure 2: situation of the stacks before the shift operation](image)

Pointer element $\sigma$ points to the process whose stack begins with the state and symbol vertices labeled 3 and D respectively and to two contexts starting with the symbols A, b, D and c, C, F. At this moment we are not interested in the remainder of the stacks nor in the rest of the pointer elements. We focus on the implementation of the shift action.

We are looking at the active triple $(v, \sigma, l)$. The vertices $v$ and $l$ and the pointer element $\sigma$ give the control mechanism all the information needed to look up the next action in the action table. Suppose the entry in the action table for state 3 and symbol c, which is the first symbol of the current context, is ‘shift 5’. The triple $(\sigma, 5, l)$ is handed over to the shift procedure. This procedure, not surprisingly called the shifter, will create two new vertices in $\Gamma_0$, one labeled with the symbol to be shifted and one labeled with the new state. Furthermore a new pointer element is added to $\Pi$. This element connects the new state vertex with all the successors of vertex $l$. In this case there is only one successor, the vertex labeled C. Note that the original contexts are left unchanged; nothing is removed, only added. Figure 3 illustrates the operation of the shifter.

![Figure 3: situation of the stacks after the shift operation](image)

Triple $(q, \tau, z)$ is added to $A$, the set of active triples. If $l$ has more than one successor, more than one new active triple will be added to $A$.

Shifting is one of the two ways of exchanging symbols between the two stacks. At a shift action only one symbol at a time is moved unchanged from $\Gamma_0$ to $\Gamma_0$.

### 2.3 Reduce

A reduce action is the other occasion on which symbols are moved from one stack to another. The direction now is from $\Gamma_0$ to $\Gamma_0$. The number of symbols that is exchanged depends on the production rule that is used. The number of vertices stripped from $\Gamma_0$ equals twice the length of the right-hand side of the rule (the vertices labeled with the symbols and the matching state vertices). The symbols of the left-hand side of the rule are pushed onto $\Gamma_0$.

Consider the configuration of the stacks as depicted in Figure 4. The triple $(v, \sigma, l)$ is processed by the control mechanism which takes a look at the action table and decides that, say, a reduce action using rule 4 should be executed. Hence
the control mechanism renders a triple \((\sigma, 4, l)\) to the reducer. Assume for the moment that rule 4 is \(A \cdot b \rightarrow C \cdot D\). The reducer will look for a path of twice the length of the right-hand side of rule 4, starting in \(w\) and whose symbol vertices match the symbols of the right-hand side of rule 4. If there is already a pointer element pointing to the last vertex of this path, the reducer simply adds a new context to that pointer element by creating vertices in \(\Gamma\), for the symbols \(C\) and \(D\) of the rule’s left-hand side and connecting them with the pointer element. If there is no such pointer, the procedure will create one and then attaches the symbols to its right. In both cases, the vertex labeled with \(\gamma_{pm}\), the last symbol of the left-hand side, will be connected to \(l\). Performing ‘reduce 4’ in the previous figure, results in the situation presented in Figure 5.

Figure 5: situation of the stacks after the reduce operation

In this example vertex \(w\) has no pointer element pointing to it, so a pointer element \(\tau\) is created. After the reduction, triple \((w, \tau, q)\) is added to the set of active triples. Because in general there can be only one vertex \(w\), only one new triple is added to \(A\) in case of a reduction.

3 A NECESSARY IMPROVEMENT

The implementation of the basic operations shift and reduce as sketched in the last sections is straightforward. When tested with some grammars the reduce action turned out not to be very efficient.

3.1 The problem

Figure 6: a situation that causes inefficiency

The main drawback of the first version of the algorithm is that it doesn’t provide any rules for merging nodes in the stack which represent the context of a process. As a consequence, a stack configuration as pictured in Figure 6 can occur. The process depicted in this figure has two contexts, both starting with the symbol \(A\). Because the algorithm uses one symbol of the context for lookahead, the same action will be performed twice on this process, one time for each context. This is, of course, useless.

Figure 7: execution of the first shift action

Suppose the action table tells us that ‘shift 5’ is the action to be executed when a process is in the state 3 and symbol \(A\) is the first symbol of the input stack. Then Figure 7 shows us the next step in the trace of the parser. In this figure the first symbol of the second context is shifted onto the shift stack. Because of the first context, one action ‘shift 5’ still remains to be executed on the process with context marker 1. We cannot simply discard this action, saying we already did a ‘shift 5’ action. If we do so and bluntly proceed with processing context marker 2, we loose the rest of the first context (containing the symbols \(b, D\) et cetera).

Figure 8: situation of the stacks after the second shift operation

What if the second ‘shift 5’ is executed straightforward? Take a look at Figure 8. There are no
more active triples left containing context marker 1, in other words: there are no more actions for this process. In the state vertex labeled 3 stack Π, splits in two identical parts, ending in two different contexts. The obvious thing to do, is to merge the vertices of these identical parts, as shown in Figure 9.

![Figure 9: merging vertices in Π](image)

The process ending in context marker 2 now has two lookahead symbols and therefore two actions attached to it (unless these actions are the same).

Is this merge-when-shift approach a solution to the problem? Not really. The same action, in this case 'shift 5', is still performed twice. The problem of having two identical vertices in Π is more or less shifted onto the shift stack and solved there, when it is too late. And what happens if the double action is a reduce action instead of a shift action?

There shouldn't be two or more contexts of a process starting with the same symbols in the first place. So the solution lies not in the merge-when-shift approach, but in a merge-when-reduce approach.

### 3.2 Merge-when-reduce approach

Let's take a second look at the problem. In a merge-when-reduce approach the situation appearing in Figure 10 should be impossible.

![Figure 10: situation of the stacks that shouldn't occur](image)

How can we prevent this from happening? We have to take back a step in the history of this graph and look at Figure 11.

![Figure 11: situation of the stacks before the reduce operation](image)

is the right time for the merge-when-reduce approach to come into action and to prevent the stack from slipping in the prohibited configuration of Figure 10. It is clear what should be done: just merge the identical vertices labeled A. The result of this strategy is shown in Figure 12.

![Figure 12: merging vertices in Π](image)

In fact, this way of acting seems to be a very simple solution for the problem of performing the same actions twice. Alas, there are some snakes in the grass.

These difficulties are related to the fact that a context marker can be part of an active or inactive triple. Consider the situation in Figure 13.

![Figure 13: challenging the merge-when-reduce approach](image)

In this case, the action 'shift 5' of pointer element 1 is executed before the reduce action has taken place. There are no more actions left for pointer element 1, so there are no more active triples containing that element. The next thing to do is the reduce action. Execution with the simple merge-when-reduce approach yields a new configuration, displayed in Figure 14.

![Figure 14: new configuration](image)

There we go again! The reduce action adds a triple holding context marker 1 to A. Since the first symbol of its context is still A, the action for this context marker is 'shift 5' afresh. Two
problems arise: we already did a 'shift 5' action on context marker 1, and by activating a triple with context marker 1 again, we not only add a new context starting with the symbols A, B and C, but also re-activate the old context with symbols A, C, f etcetera. Note that in Figure 14 the vertex labeled C can be reached from context marker 1 as well as from context marker 3, both of which are part of an active triple. In fact, the processes of which an edge points to vertex C are the same. They look different because they are in different phases (symbol A not yet shifted, symbol A already shifted).

Back to the figure. Since the vertex A is already shifted, it seems we are too late to apply our merge-when-reduce approach. However, if the reducer knows that symbol A is already shifted at the moment of executing the reduce action, it can catch up with the active process. This happens in Figure 15. A triple (v, σ, t) is added to the set of active triples, where v is the state vertex in \( \Gamma_r \) labeled 5, σ pointer element 3 and t the vertex in \( \Gamma_r \) labeled b.

To find out whether a symbol is shifted or not, a pointer from the corresponding vertex in the reduce stack to the context marker of its shifted counterpart in the shift stack is needed. This pointer is not a part of any stack, nor the pointer set and therefore not shown in the figure. In our example the pointer points from vertex A in \( \Gamma_r \) to context marker 3.

Context marker 3 is part of a still active triple, so we can simply add a new action according to the lookahead symbol b. This method can easily be generalized for situations in which more than one vertex can be merged and more than one matching symbol of the context is shifted.

With this modification, the general behaviour of the reducer can be outlined as follows. First the reducer determines the new state of the process, which is given by the label of the new top of stack \( w \), after popping off the symbol and state vertices. If there is no context attached to this vertex (i.e. there is no pointer element pointing to \( w \)), the reducer creates a pointer element and vertices in \( \Gamma_r \) labeled with the symbols of the left-hand side of the rule used in the reduction and links them. If there is a context marker, say \( \tau \), the reducer starts looking for the longest path in the context of \( w \) whose symbols match the symbols of the left-hand side. When a matching symbol in \( \Gamma_r \) is already shifted, we can continue the matching process in the context belonging to the context marker of the shifted vertex. An illustration of this can be found in Figure 16 and 17, which will be explained shortly.

In case there is no matching path in \( \Gamma_r \), the reducer creates new vertices for the symbols and makes a right pointer of context marker \( \tau \) pointing to the first of the new vertices (which is labeled \( Y_{N+1} \)). On the other hand, if a path does exist but it doesn't cover all the symbols of the left-hand side, the reduce procedure creates vertices for the symbols that are left over. If the last vertex of the path is not shifted, an edge will be created between the last vertex of the path and the first of these new vertices, otherwise the context marker belonging to the last vertex of the path will be pointing to the first of the new vertices. In the former case there will be no triple added to \( A \).

The mainspring in Figure 16 is the stack ending in the vertices B and 2, and its context consisting of the symbols A, R and d. The symbols A, R and d are all shifted onto \( \Gamma_r \). (The vertices and edges resulting from shifting R and d are not shown in the figure.) The shifted symbols R and d are then reduced to C F. This adds the context C F to context marker 2. In the next action symbol C is shifted on the shift stack (context marker 3). Finally the symbols X, Y and Z are reduced to A C G. That leads to the situation of Figure 17.

The reducer starts looking for a matching path in the context to the right of context marker 1. Only the first symbol, A, matches. The vertex labeled A is shifted, so the reducer continues to
look for the rest of the symbols in the context of context marker 2. Here it finds another matching symbol. Because the vertex labeled with that symbol is also shifted the search continues in the context of context marker 3, without success. The reducer now has found the longest matching path. Since the last vertex of this path, labeled C, is shifted, a vertex labeled g is attached to context marker 3. The edge leaving vertex g ends in the vertex labeled E in $\Gamma_r$.

4 A BRIEF REMARK ABOUT THE ACTION TABLE

The parsing table for the algorithm can be obtained by using the existing LR parsing table construction methods ([2], [3], [4], [7]). One important modification for our recognizer is that each entry in the table should be a set of actions, rather than a single action [11]. However, since we are not only dealing with non-deterministic languages, but also with their context-sensitiveness, there turns out to be another difficulty to overcome for the table constructor.

Consider a grammar with, among others, the following 4 rules:

1. $S \rightarrow A\ B$
2. $B \rightarrow D$
3. $A\ D \rightarrow a\ d$
4. $A\ C^n \rightarrow a\ c^m\ (m \geq n)$

The left-hand side of rule 4 contains $n$ C's. The first state of the action table contains the items $S \rightarrow \cdot A\ B$ and $A\ D \rightarrow \cdot a\ d$, because of the rule 2 ($S \Rightarrow A\ B \Rightarrow A\ D \Rightarrow a\ d$). Now, add the left-recursive rule

5. $B \rightarrow B\ D$

to the grammar. The first state still contains the two dotted rules mentioned above. However, with the new rule in the grammar we also have to take a look at the infinite derivation $S \Rightarrow A\ B \Rightarrow A\ B\ D \Rightarrow A\ B\ D\ D \Rightarrow \ldots$, because the grammar might contain rules which make a derivation $B\ D\ \ldots\ D \Rightarrow C^n$ possible. In that case we should also add item $A\ C^n \rightarrow \cdot a\ c^m$ to the first state.

The question here is how the table constructor knows at which point to stop considering the infinite derivation $S \Rightarrow A\ B \Rightarrow A\ B\ D \Rightarrow A\ B\ D\ D \Rightarrow \ldots$ and its possible derivations to $A\ C^n$. Stopping after $j$ D's ($j < n - 1$) could mean that the constructor misses the derivation of $B\ D^j$ to $C^n$, and thus incorrectly omits item $A\ C^n \rightarrow \cdot a\ c^m$ from the first state. The answer to the question is given by the constraint $m_p \leq n_p$ on the lengths of the left-hand and right-hand sides of a context-sensitive rule. This constraint implies that the reducer can stop looking at the infinite derivation as soon as the length of $B\ D\ \ldots\ D$ is
greater than the length of $C^n$, since a string of $n$ symbols can never be derived from a string of length $n+1$.

In the context-free case this problem does not occur, because then you simply look at the symbol just after the dot, since the left-hand side of every rule consists of one non-terminal symbol.

5 Conclusions and Future Research

The combination of Tomita's algorithm for non-LR grammars and Walters' algorithm for context-sensitive grammars results in a shift-reduce algorithm for parsing sentences of non-deterministic languages defined by context-sensitive grammars. At this stage only a version that recognizes sentences is implemented (in Modula-2 and Lisp). This version is to be extended to a full parser. Walters proposed a notation to represent a parse for context-sensitive grammars [12].

An important feature of the algorithm, its complexity, is the subject of ongoing research. The literature on complexity is often limited to context-free grammars and algorithms ([5], [6]). The problem whether an input string is in the language generated by an acyclic context-sensitive grammar is probably polynomial for fixed grammars [1].

Another point of interest is a formal proof of correctness of the algorithm. This proof should be based on the proofs of correctness for Tomita's and Walters' algorithms.

6 Formal Specification of the Algorithm

6.1 Pre-defined functions and global variables

This section presents the pre-defined functions and global variables essential to understanding the algorithm.

\text{ACTION}(s, a) \quad \text{looks up the action table and returns one or more actions} \\
\quad \text{s is a state number and a is a terminal or non-terminal symbol.}

\text{SYMBOL}(l) \quad \text{takes a vertex in } \Gamma_s \text{ or } \Gamma_r \text{ as its argument and returns a symbol labeled with vertex } l.

\text{STATE}(u) \quad \text{takes a vertex in } \Gamma_s \text{ as its argument and returns a state number labeled with vertex } u.

\text{LEFTP}(\sigma) \quad \text{takes a pointer element as its argument and returns the vertex in } \Gamma_s \text{ the pointer element is pointing to.}

\text{SUC}(w) \quad \text{takes a vertex } w \text{ as its argument and returns the set of all vertices for which there exists an edge from } w \text{ to each of those vertices.}

\alpha_1, \ldots, \alpha_n \quad \text{input string of length } n.

G \quad \text{grammar.}

\Gamma_s \quad \text{graph-structured stack in which the shift actions take place.}

\Gamma_r \quad \text{graph-structured stack in which the reduce actions take place.}

\Pi \quad \text{set of context markers or pointer elements, linking the two stacks } \Gamma_s \text{ and } \Gamma_r.

r \quad \text{boolean variable indicating if the input string is recognized, if } r \text{ is TRUE the input string is accepted, else it is rejected.}

A \quad \text{set of active triples to be processed, each triple is of the form } (v, \sigma, l), \text{ where } v \text{ is a vertex in } \Gamma_s, \sigma \text{ a pointer element in } \Pi \text{ and } l \text{ a vertex in } \Gamma_r; A \text{ is initialized in PARSE, ACTOR removes elements from } A, \text{ RECOGNIZER and SHIFTER both add triples to the set.}

S \quad \text{set of triples } (\sigma, s, l) \text{ to direct the SHIFTER; } \sigma \text{ is a pointer element, } s \text{ is the state to go to after the shift action and } l \text{ is the vertex in } \Gamma_r \text{ whose symbol is to be shifted.}

R \quad \text{set of triples of the form } (\sigma, r, l) \text{ to direct the REDUCER; } \sigma \text{ is a pointer element, } r \text{ is the number of the rule to be used in the reduction and } l \text{ is the vertex in } \Gamma_r \text{ the new context should be attached to.}

Y_{pi}^{th} \quad \text{ith symbol of the left-hand side of the } p^{th} \text{ production.}

X_{pi}^{th} \quad \text{ith symbol of the right-hand side of the } p^{th} \text{ production.}
\( |p_r| \) length of the right-hand side of production \( p \).

\( |p_l| \) length of the left-hand side of production \( p \).

In the following sections the formal description of the algorithm is given, based on Tomita’s description [11].

### 6.2 Parse

The procedure Parse forms the top level of the algorithm. In Parse the sets \( \mathcal{R}, \mathcal{S} \) and \( \mathcal{A} \) are initialized. One vertex is created in \( \Gamma_\mathcal{E} \), labeled 0 and representing the start state. The input string is the initial context. Vertices labeled with these terminal symbols appear in in \( \Gamma_r \), along with edges connecting them. The initial state and context are connected by the first pointer element in \( \Pi \). A call to the procedure Recognize starts the actual recognition process. In the end the variable \( r \) contains the result.

**PARSE** \((G, a_1, \ldots, a_n)\)

- \( \Gamma_\mathcal{E}, \Gamma_r, \Pi \leftarrow \emptyset \)
- \( \mathcal{R}, \mathcal{S} \leftarrow \emptyset \)
- \( a_{n+1} \leftarrow ’\#’ \)
- \( r \leftarrow \text{FALSE} \)
- \( \text{create in } \Gamma_\mathcal{E} \text{ one vertex } v, \text{ labeled } 0 \)
- \( \text{create } n+1 \text{ vertices in } \Gamma_r, w_0, \ldots, w_n, \text{ labeled } a_1, \ldots, a_{n+1} \) respectively
- \( \text{create } n \text{ edges in } \Gamma_r \text{ from } w_i \text{ to } w_{i+1}, \quad 0 \leq i < n \)
- \( \text{create one vertex } \sigma \text{ in } \Pi, \text{ labeled } * \)
- \( \text{create two pointers from } \sigma \text{ in } \Pi \text{ to } v \text{ in } \Gamma_\mathcal{E}, \) and from \( \sigma \text{ in } \Pi \text{ to } w_0 \text{ in } \Gamma_r \)
- \( A \leftarrow (v, \sigma, w_0) \)
- **RECOGNIZER**
- **RETURN** \( r \)

### 6.3 Recognizer

The procedure Recognize manipulates the sets \( \mathcal{R}, \mathcal{S} \) and \( \mathcal{A} \) and thus decides in which order the actions will be executed. An important implementation detail is that before a reduce action can take place, all shift actions must have been performed. This synchronization is necessary because the reducer handles reduce actions depending on whether a vertex is shifted or not, as explained in an earlier section. When a vertex is scheduled to be shifted (i.e. element of \( \mathcal{S} \)) but not yet shifted the reducer will assume it will be not shifted, causing it to make wrong moves.

**RECOGNIZER**

- **REPEAT**
  - **WHILE** \( A \neq \emptyset \) **DO**
    - **ACTOR**
      - **WHILE** \( S \neq \emptyset \) **DO**
        - **SHIFTER**
          - **WHILE** \( R \neq \emptyset \) **DO**
            - **REDUCER**
  - **UNTIL** \( R, A, S = \emptyset \)

### 6.4 Actor

Each time the actor is called it removes one element from \( \mathcal{A} \) and decides what action should be done according to the action table. Because of multiple entries the ACTION function may return more than one action.

**ACTOR**

- **remove** one element \((v, \sigma, l)\) from \( \mathcal{A} \)
- **FOR ALL** \( \alpha \in \text{ ACTION(STATE}(v), \text{SYMBOL}(l)) \) **DO**
  - **IF** \( \alpha = \text{’accept’} \) **THEN**
    - \( r \leftarrow \text{TRUE} \)
  - **IF** \( \alpha = \text{’shift s’} \) **THEN**
    - **add** \((\sigma, s, l)\) to \( \mathcal{S} \)
  - **IF** \( \alpha = \text{’reduce p’} \) **THEN**
    - **add** \((\sigma, p, l)\) to \( \mathcal{R} \)
REDUCER

- remove one element $(\sigma, p, l)$ from $R$
- $v \leftarrow \text{LEFTP}(\sigma) \in \Gamma_v$
- FOR vertex $w \in \Gamma_v$ such that there exists a path in $\Gamma_v$ of length $2*|p_r|$ from $v$ to $w$ DO
  - IF there is an element $\tau$ in $\Pi$, such that there exists a pointer from $\tau$ to $w$ THEN
    - let $z_1, \ldots, z_j$ be the longest matching path in $\Gamma_\tau$, starting from $\tau$, the symbols of which match the left-hand side of production $p$
    - IF $j = 0$ THEN
      - create vertices $q_1, \ldots, q_{p_m}$ in $\Gamma_\tau$, labeled $Y_{p_1}, \ldots, Y_{p_{m}}$, respectively
      - create edges from $q_i$ to $q_{i+1}$ in $\Gamma_\tau$, $1 \leq i < p_{n_p} - 1$
      - create an pointer from $\tau$ to $q_1$
      - create an edge from $q_{p_{n_p}}$ to $l$ in $\Gamma_\tau$
      - add $(w, \tau, q_1)$ to $A$
    - ELSE
      - IF $j < |p|$ THEN
        - create vertices $q_{j+1}, \ldots, q_{p_{m}}$ in $\Gamma_\tau$, for the non-matched symbols of the left-hand side, labeled $Y_{j+1}, \ldots, Y_{p_{m}}$, respectively
        - create edges from $q_{j}$ to $q_{j+1}$ in $\Gamma_\tau$, $j < i < p_{n_p} - 1$
        - create an edge from $q_{p_{n_p}}$ to $l$ in $\Gamma_\tau$
        - IF $z_j$ is shifted THEN
          - create a pointer from the context marker $\lambda \in \Pi$, created when shifting vertex $z_j$, to vertex $q_{j+1}$
          - add $(s, \lambda, q_{j+1})$ to $A$, where $s$ is the vertex in $\Gamma_\tau$ $\lambda$ is pointing to
        - ELSE
          - create an edge from vertex $z_j$ to vertex $q_{j+1}$ in $\Gamma_\tau$
      - ELSE
        - IF $z_j$ is shifted THEN
          - create a pointer from the context marker $\lambda \in \Pi$, created when shifting vertex $z_j$, to vertex $l$
          - add $(s, \lambda, l)$ to $A$, where $s$ is the vertex in $\Gamma_\tau$ $\lambda$ is pointing to
        - ELSE
          - if not already created, create an edge from vertex $z_j$ to vertex $l$ in $\Gamma_\tau$
      - ELSE
        - create a pointer element $\tau$ in $\Pi$
        - create a pointer from $\tau$ to $w$ in $\Gamma_v$
        - create vertices $q_1, \ldots, q_{p_m}$ in $\Gamma_\tau$, labeled $Y_{p_1}, \ldots, Y_{p_m}$, respectively
        - create edges from $q_i$ to $q_{i+1}$ in $\Gamma_\tau$, $1 \leq i < p_{n_p} - 1$
        - create an pointer from $\tau$ to $q_1$
        - create an edge from $q_{p_{n_p}}$ to $l$ in $\Gamma_\tau$
        - add $(w, \tau, q_1)$ to $A$
6.5 Reducer

The reducer takes care of the reduce actions. It tries to handle each reduction as efficient as possible by using vertices already existing in $\Gamma_r$. The way the reducer achieves this goal is by making use of so called matching paths. The reducer is shown on a separate page.

6.6 Shifter

The procedure shifter, finally, shifts a vertex from stack $\Gamma_r$ to $\Gamma_s$. Because vertices are only shifted one at a time and merging is virtually impossible, the procedure is really straightforward.

**SHIFTER**

- remove one element $(\sigma, s, l)$ from $S$
- $v \leftarrow \text{LEFTP}() \in \Gamma_s$
- create two vertices $q$ and $w$ in $\Gamma_r$, labeled $s$ and $\text{SYMBOL}(l)$ respectively, and a pointer element $\tau$ in $\Pi$ labeled $*$
- create two edges in $\Gamma_r$ from $w$ to $v$ and from $q$ to $w$
- create a pointer from $\tau$ to $q$
- add a pointer from $l$ to $\tau$
- FOR ALL $z$ in $\text{SUC}(l) \in \Gamma_r$ DO
  - create a pointer from $\tau$ to $z$ in $\Gamma_r$
  - add $(q, \tau, z)$ to $A$

References


ABSTRACT

A program-generator for linguistic purposes is described with the following properties:
- The formalism unifies weak equivalent formalisms for Chomsky type-0 grammars, cascaded transduction grammars, regular expressions, pattern matching, augmented transition networks and attribute grammars.
- In the compiler extensions for the LR-table generation technique are implemented.
- The generated code runs on a formal machine which makes use of two dag-structured stacks and which maintains a parse forest for the efficient storage of parses of ambiguous grammars, like in the algorithm of Tomita.
- Parsing is done on-line; transduction is done with a finite delay.
- New bounds are obtained for the runtime of some pattern matching problems. For instance, the recognition in a string of a set of substrings which contain don't cares may be done in sub-linear time.
- The system (called Parspat) has been used in projects concerning the recognition of complicated patterns in tagged corpora of texts and music, for the parsing of a number of natural languages, for the transformation of graphemes into phonemes by a cascaded transduction grammar, for the recognition of compound words, for the classification of words in a number of languages, for the interactive detection of the structure of a document and its subsequent conversion into another format.

1 INTRODUCTION AND HISTORY

The Parspat system ("Parser for Pattern Grammars") originated in the years 1980-1986 in the Computer Department of the Faculty of Arts of the University of Amsterdam. The Department was confronted with a number of computational problems, stemming from research groups in the Departments for Language, Musicology, History and Library Sciences. Initially, the problems seemed to diverge widely. But they became gradually unified when we tried to develop formalisms for them, which were a unification of existing grammatical formalisms. The accepted formalisms are among the most widely used in scientific and industrial applications, like attribute grammars, augmented transition networks, augmented phrase structure grammars, some forms of transformational grammars, Chomsky type-0 grammars and transduction grammars. But also facilities for pattern matching are included. Extensions were made to LR parser theory. A new formal machine model (PTA, for Parallel Transduction Automaton) was developed in which all kind of ambiguities are handled in an efficient way. Also new bounds for pattern matching problems were achieved and implemented.

Our research has been published as a doctoral dissertation in (Van der Steen, 1987), which was reprinted and provided with an index in (Van der Steen, 1988). We left the University of Amsterdam in 1988. The system Parspat is still maintained and distributed to the industry by the original Computer Department, which extended it recently with an error-repair facility.

After completion of the development of our PTA we learned about the work of Tomita (1986). We identified his algorithm as akin to our algorithm, which extends it in a number of ways. In this paper we discuss some of these extensions in some detail. For more discussion we refer the reader to (Van der Steen, 1988) and to the latest version of the user manual of the Parspat system (Elstrodt, 1990).
MOTIVATION FOR THE
PARSPAT SYSTEM

Syntactic descriptions play an important role in
numerous applications from Linguistics, Artificial
Intelligence, Musicology, Biology, Shape Analysis.
They are a model of the apparatus by which we can
select meaningful impressions from the outside
world and by which we can express our messages to
that world.

Most of the formalisms for syntactic descriptions
stem from Linguistics. According to these formalisms
it is possible to generate, to recognize, to
parse and to transduce parts of sentences of a
language in a precise way. ”Syntactic Pattern
Recognition” (abbreviated "SPR") is, at this
moment, an emerging field (Gonzalez and
Thomason, 1978) (Fu, 1982) in which the syntactic
methods of Linguistics are applied to other fields. A
central role is played by the recognition and
transduction of parts of datastructures of atomic
symbols, formulated in some grammatical
formalism.

The potential applications of SPR are numerous.
All natural objects in which we can project some
ordering and all artificial objects in which some kind
of ordering or sequencing plays a role are candidates
for syntactic description and thus become objects for
a syntactic treatment.

In natural language applications the problem of
ambiguity is ubiquitous. Only systems which are
transparent for ambiguities arising from different
possible ways of perception and interpretation have a
chance to become instruments for semantic analysis.

There are a number of applications in
Computational Linguistics which were frustrated and
eventually stopped by the lack of efficient
computational strategies. The computer programs
should be of a sufficiently general and optimized
nature to become a stimulus instead of a hindrance to
new research-projects. The programming should
preferably be done in an automatic way, straight
from the formalism in which problems are ex-
pressed.

The attempts to automatically transform
grammars into recognizing, parsing or transducing
programs have their roots in Computer Science,
where a rich literature originated on automatic parser
generation for programming languages. These
techniques are now extended to the domain of
Computational Linguistics under the flag of
'Generalized LR parsing', and they can be extended
further to the domain of SPR.

In figure 1 we indicate for some typical
applications in SPR the desirable characteristics of
their implementation.

Current bottle-necks are : the limited availability
of on-line and real-time recognition, parsing and
transduction algorithms for a large number of
syntactic formalisms.

In this paper we concentrate ourselves on on-line
recognition, parsing and transduction.

The Parspat system generates efficient programs
for a formalism which unifies a number of popular
grammatical formalisms which are in use in
Computational Linguistics and in SPR. The core of
the runtime system consists of an algorithm which
can be viewed of as an extension to the algorithm of
Tomita.

In section 4 we will describe the functionality of
the Parspat system. In section 5 we describe the
extensions to the algorithm of Tomita which will be
worked out in further detail in the sections 6 and 7.
Typical applications:

<table>
<thead>
<tr>
<th>Application</th>
<th>Recognition</th>
<th>Parsing</th>
<th>Translation Line</th>
<th>Online</th>
<th>Real-Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>EEG analysis</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Shape analysis</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Pattern-involutions</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Text manipulation</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Exploration of texts</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Free-text database systems</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Text-to-speech systems</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Speech-recognition</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Grammatical analysis</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Machine translation</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

Figure 1: desirable characteristics of applications in SPR

4 FUNCTIONALITY OF THE PARSPAT SYSTEM

The Parspat system consists of a compiler and a runtime system which interprets code for a formal machine. The working of the system is depicted in figure 2.

4.1 Introduction to the unifying formalism

What is desirable in a formalism for the automatic processing of texts?

In a paper (1985) with the title "Hiding complexity from the Casual Writer of Parsers" Dahl writes as follows: "[I]t raises the question of whether it is indeed possible to construct a formalism that combines efficiency with high expressive power, and that hides from the user all details that can be automated, thus providing a simple way of describing the purely creative grammar-writing aspects. So far it is difficult to see just how much should be made transparent, just how much expressive flexibility is appropriate without the formalism becoming too powerful and introducing new problems on this account, and just how to ensure efficiency without burdening the user with machine-oriented concerns such as control".

These remarks reflect the current state of the genetic aspect of computational grammatical systems. We suggest that high expressive power can be reached by unification of frequently used formalisms and that program generation should hide the details that can be automated. Program generation should be possible for all formalisms, however powerful they may be.

Winograd (1983) discusses a large number of syntactic formalisms. Some of them have been used only experimentally, others are in use in practical natural language systems. We are interested in the latter category and in the question how we can build effective systems.

In this section we present a unifying syntactic formalism for which the Parspat system generates and executes programs.
Terminology

In the sequel we will use the following abbreviations:

- **cf** for context-free
- **cfg** for context-free grammar
- **ecfg** for extended context-free grammar
- **cs** for context-sensitive
- **csg** for context-sensitive grammar
- **lhs** for left-hand side (of a rule)
- **rhs** for right-hand side (of a rule)
- **FSA** for Finite State Automaton
- **PDA** for Push Down Automaton

Notations and definitions:
- a "symbol" is an atomic entity, such as a letter or a digit; in Parspat it is an Ascii-character
- an alphabet is a finite set of symbols
- a string (or word) is a finite sequence of symbols from some alphabet
- \( \varepsilon \) denotes the empty word

- \( A^* \) is the set of all strings over the fixed alphabet \( A \)
- \( \emptyset \) denotes the empty set

4.2 The unifying formalism

A grammar which is written according to the unifying formalism is called a "U-grammar".

**Definition.**
A U-grammar \( G \) is a 9-tuple \( (N, I, T, S, Z, C, R, P, M) \) where
- \( N \) is a finite set of nonterminal symbols; those symbols that are rewritten, as only symbol, at a lhs
- \( I \) is a finite set of intermediate symbols; intermediate symbols are introduced in a lhs of a grammar rule, where \( \text{lhs} > 1 \)
- \( T \) is a set of terminal symbols (not necessarily finite); terminal symbols may be read from the input
- \( N, I \) and \( T \) are disjoint sets; \( V \) is the union of \( N, I \) and \( T \)
- \( S \) is a distinguished start-symbol, with \( S \in N \) or \( S = \varepsilon \); if \( S = \varepsilon \) then the grammar is a transduction grammar, otherwise it is a Chomsky-type grammar
- \( Z \) is a finite set of variable symbols which can act as a parameter to a nonterminal symbol; expressions
of variables and constants may be assigned to a variable and Boolean expressions can be formed with them
- C is a finite set of cooperation symbols with which Boolean relations between rules can be expressed
- R is a finite set of report-numbers with which a position within a rhs can be marked
- P is a finite set of rewriting rules
- M is an ecfg which describes the syntax of the symbols and the rules; we refer to this syntax by the term "unifying formalism"; M is called the metagrammar of G.

\[ \Delta \]

In the sequel we will use the following notations:
- a, b, c, ..., (other than ε) are elements of T
- A, B, C, ..., are elements of N
- w, x, y, ..., are words in T*
- X, Y, ..., are elements of V
- \( \alpha, \beta, \gamma, ..., \) are words in V*

\[ \Delta \]

For a U-grammar G = (N, I, T, S, Z, C, R, P, M) only the set P of production-rules has to be specified by the grammar writer. Usually the metagrammar M has been fixed for a number of applications. The other sets are derived from P with the aid of M in the following way.

The first rule of P determines whether G is a phrase structure grammar ("PSG") or a transduction grammar ("TDG"). If the length of the lhs of the first rule is 1 then it is a PSG and the notion at the lhs is the start-symbol. Otherwise it is a TDG.

T, Z, C and R are determined by the syntax of the metagrammar. I is the set of symbols which can not be defined otherwise.

G will be transformed into a process \( G_p \) by a compiler and an interpreter. The semantics of G will be determined by the (input, output) pairs of \( G_p \).

It will depend upon the operating environment of \( G_p \) whether input and output are related to other processes, to disk-files or to strings which are provided by surrounding programs. For instance, in the Parspat system the compiler and the runsystem can be provided to a program as an external procedure written in a programming language, or they can be called from the shell of an operating system.

In general we abstract from the environment and use the name of the process \( G_p \) also as a function to denote the transformation of input into output. The sources of input and output stem from the functional environment of grammatical systems.

We will denote the input and output of a particular process \( G_p \) by a number of parameters and write \( G_p(I_n, L_{ex}, R_{ec}, P_a, O_u, R_p) \), where
- \( I_n \) is the input as a string in \( T^* \),
- \( L_{ex} \) is the lexicon which has to be used,
- \( R_{ec} \) is a Boolean variable which will receive the value true in case the recognition succeeds, and false otherwise,
- \( P_a \) is the parse which will be created on-line; if \( G_p \) is called with \( \varepsilon \) as actual value for \( P_a \) no parse will be produced,
- \( O_u \) is the output as a string in \((V+Z)^*\) : the symbols in \( V^* \) may be accompanied by variable symbols in \( Z \) with their value; if \( G_p \) is called with \( \varepsilon \) as actual value for \( O_u \) no output will be produced,
- \( R_p \) is a string in \( R^* \); if \( G_p \) is called with \( \varepsilon \) as actual value for \( R_p \) no reports will be produced.

\[ Definition. \]

A cascaded grammar C is a set \{ \( G_1, G_2, ..., G_{m-1}, G_m \) \}, \( m \geq 1 \), where \( G_1, G_{m-1} \) are TDG's and \( G_m \) is a TDG or a PSG.

A cascaded process \( C_p \) is a set \{ \( G_{p1}(I_{n1}, L_{ex1}, R_{ec1}, P_{a1}, O_{u1}, R_{p1}, B_{ld1}) ..., G_{pm}(I_{nm}, L_{exm}, R_{ecm}, P_{am}, O_{um}, R_{pm}, B_{ldm}) \) \}, \( m \geq 1 \). All processes \( G_{p1}, ..., G_{pm} \) run in parallel. The output of one process may be the input to another one. This is indicated by parameters with the same name.

\[ \Delta \]

In the unifying formalism six components are distinguished:
1. the BNF notation for Chomsky type-0 grammars,
2. regular expressions,
3. reserved notions for patterns, trees and references between lhrs and rhs's,
4. the use of variables,
5. notations for Boolean constructs,
6. notations for output.
In the following subsections some components will be discussed in more detail. The notations for Boolean constructs and for trees in the input are only allowed when a PFA can be constructed. Because we are interested in this paper in extensions to Tomita-style parsing we leave them out of the discussion. Interested readers are referred to (Van der Steen, 1988).

4.2.1 The basic formalism: Chomsky type-0 grammars and transduction grammars

The formalism of general rewriting is, among others, provided in: transformational grammars, string-replacement, graph- and tree grammars, cs-grammars for phonological rewriting and the formalisms for machine translation. Term-rewarding systems are in use in compilers and other symbol manipulation systems.

In the linguistic tradition, grammars are commonly used as a means to systematically describe the sentences of a language. This can either be from a generative or an analytic point of view. In the generative case the grammar specifies the sentences by its possible derivations. In the analytic case a sentence of the language is parsed.

One of the most established formalisms for grammars is the BNF-notation. The use of non-terminals enables the sharing of common substructures and the recursion mechanism enables recursive writing.

In transduction, grammatical knowledge, expressed in the grammar, is used to perform the operations insert, delete and change on the input, resulting in the "transduced" output. The result of a single transduction remains possible input of other grammar rules, until no more rules can be applied.

One essential difference between recognition and parsing on the one hand, and transduction on the other, is that transduction lacks a start symbol (or: distinguishing symbol). The transduction varies freely over the input. All transduction rewrite rules will operate in parallel. Transduction stops when no more rules are applicable. This process may give rise to a number of ambiguous transductions.

Nonterminals may be suffixed by the "cover symbol" \(\lambda\). During parsing or transduction a covered nonterminal will be replaced by the terminals which are at the leaves of its associated parse tree(s). The current scope for covering at the lhs is the governing cs rule. The current scope for covering at the rhs is the current regular expression. If the notion is not known then the scope is widened to the surrounding regular expression, etcetera.

When a U-grammar \(G= (N, I, T, S, Z, C, R, P, M)\) is reduced to a Chomsky type-0 grammar the sets of symbols \(I, Z, C\) and \(R\) are empty. \(M\) will describe that the rules in \(P\) are in general of the form:

\[X_1, X_2, \ldots, X_n \colon:: Y_1, Y_2, \ldots, Y_m,\]

where \(X_i, Y_j \in N \cup T\).

Depending on the maximum values allowed for \(n\) and \(m\) and restrictions on the use of terminals and nonterminals four different kinds of rules may be distinguished, which give rise to the classical "types" of phrase structure grammars and the Chomsky hierarchy.

The first extension of the BNF formalism is that we allow the rhs's to be empty \((m=0)\) for type-3, -2 and -0 grammars. The second extension concerns TDG's.

example of a CSG

a csg which generates the language \( \{a^n b^n c^n d^n \mid n \geq 1\} \) (Levelt,1973):

\[Z \colon:: E, Z, F \mid a, b, c, d.
E, a \colon:: a, E.
F \colon:: d, F, d.
E, b \colon:: a, b, b.
c, F \colon:: c, c, d.
\]

examples of a TDG

1 \[A, B \colon:: B, A
A \colon:: a, c
B \colon:: b
C \colon:: c
\]

! The input 'bac' will be rewritten as 'acB'!

2 \[E, SG \colon:: e, SG.
SG \colon:: CONS, CONS & 'k,j'.
CONS \colon:: 'k, i, l, o, u'.
!
\]

An 'e' followed by a string of 2 consonants which is not 'kl' will be rewritten to 'E'!
3 \quad A^\wedge < A, A^\wedge .

A :: a..z.

With this transduction grammar all double lower case characters will be singled.

Four processes for interpretation of a U-grammar can be distinguished.

1. Generation with a phrase structure grammar \( G \). Input is the starting symbol, output a sentence which consists of terminal symbols. The process itself consists of a stepwise derivation.

**Definition.**
A derivation in \( G \) is a sequence \( \alpha_1, \alpha_2, \ldots, \alpha_{m+1} \), \( m \geq 0 \), of strings such that for each \( i, 1 \leq i \leq m \), there are strings \( \beta_i, \gamma_i, \delta_i, \zeta_i \) such that
\[
\alpha_i = \beta_i \gamma_i \delta_i, \quad \alpha_{i+1} = \beta_i \gamma_i \delta_i, \quad \text{and} \quad \gamma_i :: \zeta_i \in P.
\]
Associated with each of \( \alpha_1, \ldots, \alpha_m \) there must be a pair \( \langle p, r \rangle \) denoting that \( \gamma_i :: \zeta_i \) is the \( p \)-th rule of \( P \) and the first symbol of \( \gamma_i \) is the \( r \)-th symbol of \( \alpha_i \). Each \( \alpha_i \) is a line of the derivation, and the process of applying a rule to one line to produce the next line is a step of the derivation.

The sequence \( \alpha_1, \ldots, \alpha_{m+1} \) is said to be a derivation of \( \alpha_{m+1} \) from \( \alpha_1 \), and \( \alpha_{m+1} \) is said to be derivable from \( \alpha_1 \).

\( \Delta \)

2. Recognition with a phrase structure grammar. Input is a string \( w \in T^* \). Output is the answer "yes" or "no" depending on whether \( w \) can be generated by \( G \) or not. With on-line recognition this answer will be given after the reading of each symbol.

3. Parsing with a phrase structure grammar. Input and output are the same as with a recognizer, but the process will also reconstruct all possible derivations.

The following definition of a **parse** as a two-dimensional description of a set of derivations follows closely the one of Walters (1970). It generalizes the definition of a parse tree for cfg's.

**Definition.**
A parse of \( \beta \) from \( \alpha \) is a bracketed diagram showing how \( \beta \) is derived from \( \alpha \). Such a parse is obtained from any derivation of \( \beta \) from \( \alpha \) by writing down \( \beta \), then bracketing the rhs of the last rule used in the derivation, writing the subject of the rule above the bracket, and associating the rule number with the bracket. The string resulting from the replacement of the bracketed symbols by those above the bracket is the penultimate line of the derivation. If this bracketing is continued until all the steps in the derivation have been utilized, the result is a parse of \( \beta \) as \( \alpha \). The set of derivations that would yield the same parse under this construction is the set described by the parse.

\( \Delta \)

The equality of two parses can be tested after a structure-preserving transformation of a 2-dimensional bracketed parse into a string-representation.

**Example.** In the phrase structure grammar

(1) \( S :: A, B, C \).

(2) \( A, B :: a, B \).

(3) \( C :: D \).

(4) \( B, D :: b, d \).

a derivation and the corresponding parse are

\[
S <1,1>
\]
\[
A, B, C <3,3>
\]
\[
A, B, D <2,1>
\]
\[
a, B, D <4,2>
\]
\[
a, b, d
\]

\[
\begin{array}{c}
1 \\
R B C
\end{array}
\]

\[
\begin{array}{c}
2 \\
B D
\end{array}
\]

\[
\begin{array}{c}
3 \\
a b d
\end{array}
\]

**Definition.**
Corresponding to every parse is a unique lefmost (rightmost) derivation, which can be constructed from the parse as follows.

Write down \( \alpha \). Find the lefmost (rightmost) bracket that has no brackets above it, and delete it. The next line of the derivation is determined by \( \langle p, r \rangle \), where \( p \) is the rule number labeling the deleted bracket and \( r \) is the position of the lefmost (rightmost)
bracketed symbol. Repeat this process until all brackets have been deleted.

\[ \Delta \]

The derivation in the example above is the rightmost one. The leftmost derivation is

\[
\begin{align*}
S & <1,1> \\
A, B, C & <2,1> \\
a, b, C & <3,3> \\
a, b, D & <4,1> \\
a, b, d & 
\end{align*}
\]

The Parspat system delivers a parse in the form of a string from which the 2-dimensional bracketed parse can be reconstructed. For our example this string is:

\[ S_1(A_2[ a \ B \ 4[ b \ d ]_4 ]_2 \ B \ C \ 3(D)_3 )_1 \]

The brackets are labeled with the number of the rule. The type-1 and -0 rewritings are indicated by square brackets. The rhs of a rule which is used in a derivation is attached to the first symbol of its lhs.

This accounts for the 1-dimensionality. With the aid of the number of the rule it can be reconstructed for the 2-dimensional parse how far its brackets have to extend to the right (when the size of the lhs is fixed).

This representation may serve as a canonical representation for a 2-dimensional parse. It works for all types of Chomsky-grammars. With it the equality of parses may be tested.

4. Transduction with a transduction grammar \( G \). Input is a string \( \alpha_1 \in T^* \), output a string \( \alpha_{m+1} \in (T^+)^* \). \( \alpha_{m+1} \) is said to be a translation of \( \alpha_1 \). In a cascaded grammar \( C = \{ G_1, G_2, \ldots, G_m \} \) the strings \( \alpha_i \) are element of \( (T^+)^* \) for every \( G_i \) (1\leq i \leq m).

The definition of a transduction is nearly the same as that of a derivation. The only difference is that the lhs and the rhs of the rules are interchanged.

**Definition.**

A transduction in \( G \) is a sequence \( \alpha_1, \alpha_2, \ldots, \alpha_{m+1}, m \geq 0 \), of strings such that for each \( i, 1 \leq i \leq m \), there are strings \( \beta_i, \gamma_i, \delta_i, \zeta_i \) such that

\[
\alpha_i = \beta_i \; \gamma_i \; \delta_i, \; \alpha_{i+1} = \beta_i \; \zeta_i \; \delta_i, \; \text{and} \; \zeta_i \; :: \; \gamma_i \in P.
\]

**Definition.**

\( \alpha_m \) is a normal form of \( \alpha_1 \) if \( \alpha_m \) is a translation of \( \alpha_1 \) and no rule is applicable to \( \alpha_m \).

\[ \Delta \]

We repeat from Dershowitz (1985) three desirable properties for transduction grammars:

1. **termination**- no infinite derivations are possible
2. **confluence**- each string has at most one normal form (this is also called the Church-Rosser property)
3. **soundness**- equal strings are only rewritten to equal strings, that is, there is only one possible derivation; we call this property also "unambiguous".

Each of these properties is in general undecidable. Dershowitz (1985) shows that confluence is decidable for terminating systems.

We are interested in the automatic construction of transducers for general transduction grammars, whether these grammars have the desirable properties or not, and in a classification of heuristics, stemming from applications, which may be built into the transducers in order to obtain one or more of the desirable properties. We therefore allow also for transduction grammars which do not have the desirable properties.

The following example shows some transductions with the ambiguous (and therefore not sound) transduction grammar:

\[
\begin{align*}
(1) & \ a, a :: A. \\
(2) & \ b, a :: A. \\
(3) & \ c, a :: a, c. \\
(4) & \ c, b :: b, c. \\
(5) & \ A :: a, b.
\end{align*}
\]

and the input string \( \alpha_1 = pabcq \).

The transduction process according to this grammar terminates, is not confluent and is not sound.

(Transductions 1 and 2 do not reject confluence because they rewrite to the same normal form, but 2 and 3 do; 1, 2 and 3 reject soundness because equal terms are rewritten to different terms.)
Central in the processes for recognition and transduction is the (re)construction of a derivation. In this paper we study the on-line aspects of these processes and restrict the interpretation to on-line interpretation.

An on-line interpretation consists of an on-line (re)construction of a derivation and of the creation of the normal form(s) within a finite delay after the reading of the input string, where input and output possibly overlap in time with each other.

Heuristics, stemming from the practice of phonological rewriting, may be activated to reduce the number of ambiguities:
- if 2 rhs's are applicable, the longest one is chosen
- rewriting according to a rule is performed as soon as possible; the piece of text which was matched by the rhs of the rule is no longer subject to other transformations.

4.2.2 Regular expressions

Application
If regular expressions are used in the rules of a cfg then the grammar is called an ecfg. Ecfg's are quite naturally used in phonological transduction, morphological analysis and pattern matching. Regular expressions provide for a compact notation of repetition and embedded alternation. Syntactic diagrams and recursive transition networks are easily rewritten into regular expressions.

Functioning within formalism
Regular expressions could be used in the lhs.

Semantics
A notion is the most simple regular expression. If r, r1 and r2 are regular expressions then the following ones are also regular expressions:

- expression meaning
  - $r \cdot r$ : concatenation of r1 and r2
  - $r \cup r$ : r1 or r2
  - $[r]^*$ : 0 or more repetitions of r
  - $[r]^+$ : 1 or more repetitions of r
  - $[r]^1$ : 0 or 1 r.

A regular expression can be rewritten as a weak equivalent cfg, which means that the same language will be generated but not with the same derivations. In the parse of a regular expression no special marker will signal the appearance of the expression.

Example
$$[a \cdot b]^1 \cdot [c \cdot d \cdot [E]^+]^* \cdot [f]$$
is a regular expression.

Example.
We combine the subformalism of type-0 grammars with the subformalism of regular expressions. In the type-0 grammar of figure 3 some EST principles from transformational grammar theory are used. (The grammar was provided by J. van den Hock.) The drawing of one of the parses is based upon the output of the Parspat system.
"de man belt om de vrouw te zien op"

Parse:

Figure 3: parsing of a transformational grammar
Example

We combine the subformalism of transduction grammars with the subformalism of regular expressions.

Formula manipulation systems have a long tradition. Knuth (1968, vol. 1, page 337) gives examples on the computational treatment of symbolic differentiation. Dershowitz (1985) redefines the rules for symbolic differentiation as general rewrite rules and studies the formal properties of general rewrite systems. We repeat some of the grammar rules for symbolic differentiation:

\[ \begin{align*}
D_X x : 1. \\
D_X \alpha : 0. \\
D_X (\alpha + \beta) : D_X \alpha + D_X \beta. \\
D_X (-\alpha) : -D_X \alpha. \\
D_X (\alpha \beta) : \beta D_X \alpha + \alpha D_X \beta.
\end{align*} \]

where \( D_X \) is the differentiation operator and 'a' stands for any constant symbol other than \( x \). \( \alpha \) and \( \beta \) are variables of the rewrite system and match any term, while \( x \) is a constant of the system and matches only itself.

The rules can be rewritten in the unifying formalism as indicated in figure 4.

---

\[ \begin{align*}
! differentation ! \\
1 < D, \text{Var} . \\
0 < D, \text{Constant} . \\
D^\wedge, \text{Expr1}^\wedge, \text{^'+}, \text{Expr2}^\wedge :: D, (', \text{Expr1}, '\+', \text{Expr2}, ')'. \\
D^\wedge, \text{Expr1}^\wedge, '\-', D^\wedge, \text{Expr2}^\wedge :: D, (', \text{Expr1}, '\-', \text{Expr2}, ')'. \\
', D^\wedge, \text{Expr}^\wedge :: D, (', '\-', \text{Expr}, ')'. \\
\end{align*} \]

\[ \begin{align*}
! distribution ! \\
\text{Expr2}^\wedge, D^\wedge, \text{Expr1}^\wedge, '\+', \text{Expr1}^\wedge, D^\wedge, \text{Expr2}^\wedge :: D, (', \text{Expr1}, '\+', \text{Expr2}, ')'. \\
\end{align*} \]

\[ \begin{align*}
! simplifications ! \\
0 < \text{Expr}, '\ast', 0. \\
0 < 0, '\ast', \text{Expr}. \\
\text{Expr}^\wedge < 0, ['+', '\-', 1], \text{Expr}. \\
\text{Expr}^\wedge < \text{Expr}, ['+', '\-', 1], 0. \\
\end{align*} \]

\[ \begin{align*}
! lexical symbols ! \\
\text{E} :: [''], T, ['+', '\-', 1], T. \\
T :: F, ['+', '\-', 1], F, 1. \\
F :: \text{VarConst} \mid D, (', E, ')'. \\
\end{align*} \]

\[ \begin{align*}
! arithmetic expression ! \\
\text{Expr} :: \text{E}. \\
\end{align*} \]

Figure 4: a grammar for formula manipulation

On the basis of these rules the Parspat system performs symbolic differentiation. More rules can be added for division etc. and for more simplification.

For more examples of linguistically motivated transduction grammars we refer the interested reader to chapter 8 of (Van der Steen, 1988) and to the User manual of the Parspat system (Elstrodt, 1990).
4.2.3 Notions for pattern matching

In a number of applications the notions "don't care" and "arb" are necessary. In pattern matching they are used as "wild-card's", in syntactic analysis as "any intervening material". "Don't cares" and "arb's" may also appear in texts.

The "arb" is named after the built-in function "arb" of Snobol. It matches a number of characters up to the character(s) which may be expected after the arb, according to the grammar.

The "line" plays an essential role in the unification of parsing and pattern matching. It allows for the multiple matching of phrases in an input.

On the lexical level ranges of terminal symbols may be useful, for instance in morphological and phonological grammars and grammars for the description of texts.

The unifying formalism of the Parspat system contains these formalisms. The method of LR table construction was extended in order to generate efficient programs in linear time for these subformalisms. For instance, it is possible to denote in a pattern grammar that a Sentence has to contain a NP and a VP, in any order, which are surrounded by other material which has not to be defined further.

We refer to (Van der Steen, 1988).

4.2.4 Notation of Actions

The extensions to the formalism for variables, for Boolean constructs between rules and for output are all denoted within braces. The actions within braces can be placed on well defined places within a rhs.

We refer to (Van der Steen, 1988).

4.2.5 Variables

application

Variables are present in a number of grammar-formalisms. They may be divided into three classes:

- formal parameters bound to nonterminal symbols (like attribute grammars and Prolog),
- global variables, like those used in atn-grammars (scope: the whole grammar),
- local variables, (scope: the grammar rule in which they appear).

The possible operations on variables may be summarized by:

- Snobol-like conditional-assignment to variables from the text,
- assignment of expressions of other variables,
- tests on (expressions of) variables; a desirable feature (for unification) is postponed evaluation and assignment when variables do not have a value,
- matching on the value of variables and assigning substructures to other variables.

The only feature which is not allowed in the unifying formalism is the use of global variables. The consequence for ATN grammars is that the variables which are used in a sub-ATN have to be passed by parameters.

functioning within formalism

Nonterminals and lexicon terminals may be enriched with parameters. The parameters of nonterminals at a rhs have to be preceded by the declarations "1.0." or "1O.0.", which correspond respectively with the features "inherited", "synthesized" and the combination of both, which are familiar in attribute- and affix-grammars. Parameters of lexicon terminals need the "O:" declaration. In the latter case the actual value(s) will be supplied by the lexicon.

The value of a variable is a string of arbitrary length. Within an expression variables may be concatenated, together with string-constants and the last read-in character.

A variable is declared by its first appearance in a rhs. The scope of the variable is the grammar rule with that rhs.

Operations on and with variables are denoted within the action-brackets. The operations are: assignment and test.

In an assignment, an expression with variables and constants is assigned to a variable.

A test concerns the truth-value of a Boolean expression. If this value is "false" then the current recognition path stops.

lexical considerations

Variables are denoted as strings of characters and digits. The concatenation operator is '1'. A string-constant is a string of Ascii-characters between
quotes. In the string the symbols '(' and ')' are
reserved and act as tree symbol. The reserved symbol
'%' stands for the last symbol that is read in.

restriction in Parspat

Unification of variables will be implemented in a
later stage. In the current implementation variables
which appear in an expression need to have a value
assigned to them when the expression is evaluated.

```
S(NGETAL):: NP(NGETAL), VP(VGETAL) [NGETAL=VGETAL].
NP(O:Ngetal):: [LIDW], [BIJVNW]*, NAAMW(Ngetal).
VP(O:Vgetal):: FINNW(Vgetal), [INFNW]*.
LIDW::
  'de' | 'het' | 'een'.
BIJVNW::
  'groen',[e].
NAAMW(O:N)::
  'man metje' [N:=E'], [s [N:=M']].
FINNW(O:N)::
  'wa', [s [N:=E'] | 'ren' [N:=M']].
INFNW::
  'gevonden'.
```

Figure 5: a grammar with variables

After recognition the name of the variable NGETAL
with its value will be brought to the output.
3. assignment from a lexicon :
NP:: $det(gender), $adj(singplu1),
$noun(singplu2) [singplu1 = singplu2 ].

We assume that the lexicon contains the value for
the gender of a "det" and the multiplicity of an "adj"
and of a "noun".

4.2.6 Boolean negation within a
rule

The Boolean negation (complementation) is desirable
with "rewriting in context" (phonological rewriting)
and in query-languages for free-text database systems.
In (Aho, Hopcroft and Ullman, 1974, p. 419) it is
argued that Boolean operators in regular expressions
may shorten their length, and therefore may provide
for a more compact notation. In the Parspat system
the negation operator is implemented in combination
with those sub-formalisms for which a FSA can be
constructed and for transduction grammars which are
not recursive.

For a further discussion we refer to (Van der
Steen, 1988).

4.2.7 The lexicon

application

Terminals may be single ASCII characters or lexicon
symbols, which are preceded by a '$'. The Parspat
runsystem cooperates with a lexicon with the
datastructure of a trie.

In figure 6 an example is given of a trie (supplied
by J. Skolnik). A star denotes the end of an entry.

There are a number of properties of tries which make
them useful for the automatic processing of text.
They are :
- the alphabetical order of entries is preserved
- common prefixes of entries are stored only once
- the lookup of an entry is an on-line process : during the reading of an entry from the input one
  may walk in parallel through the trie, getting an
  indication of whether further progress is possible
- because of this on-line property it is a good
  companion for on-line syntactic analysis
- multi-words may be stored and processed in a
  simple way
entries may have any length; no fixed storage has to be reserved.

Trie:

Contents of the trie:

\[
\begin{array}{c}
\text{do} & \text{m} & \text{e} & \text{an} \\
\text{nkey} & \text{e} & \text{at} & \text{are} \\
& \text{dope} & \text{e} & \text{tal} \\
& \text{dot} & \text{at} & \text{tal} \\
& \text{me} & \text{tal} & \text{tal} \\
& \text{men} & \text{tal} & \text{tal} \\
& \text{metal} & \text{tal} & \text{tal} \\
& \text{more} & \text{tal} & \text{tal} \\
\end{array}
\]

Figure 6: a trie structured lexicon

Kunst and Blank (1982) discuss the merits of the trie-datastructure for morphological analysis.

In his survey on access methods for text, Faloutsos (1985) categorizes the trie as the only datastructure which enables access to a list of keywords in a time which is proportional to the length of the keyword searched for. However, if the trie has to be made external on disc, a more than trivial implementation is necessary in order to let it compete with the widely known family of B-trees and with hash-coding methods. Because of the absence of such a non-trivial implementation Faloutsos further ignores tries on external memory.

The Parspat runsystem makes use of an efficient implementation of such an external trie (Skolnik, 1982). In particular the number of disc-accesses which are needed for the lookup of an entry is optimized. Each position in the trie may be characterized by a set of internal pointers. Functions exist to ask for the current position, to store a position and to locate on a position. By storing positions in the trie itself the external trie may be transformed into an external dag or an external network.

The "lookup" operation is performed on-line and will be initiated when a lexicon terminal in the grammar has to be matched. After the reading of the next character three messages may be reported by the lexicon function: failure (no such entry), success (entry found) and proceed (continuation possible). The last two messages may be combined when an entry is found which is the prefix of another entry. In the case of failure the current derivation will stop.

Entries in the lexicon are strings of arbitrary length and may be followed by one or more lexical notions, separated by a delimiter. The categories will be successively assigned to the variables which are denoted as parameters with the lexicon terminal. The first parameter will always return the matched entry.

The semantics for on-line recognition according to a grammar can be extended to the on-line processing of a lexicon which has the trie-datastructure. By combining these semantics the external lexicon can be treated as an integral part of the grammar.

4.2.8 Output

4.2.8.1 Reports

"Reports" are used in order to trigger other processes during recognition. They enable a "syntax-driven" approach with a strict separation between syntax and semantics. The following remarks concern on-line recognition. The reports will be brought to the output string as soon as possible. If there is only one derivation this means: immediately. When there exist two or more derivations the reports are stored within the parse. At the moment that some derivation stops and only one derivation exists then the stored reports are brought to the output string: in that respect the outside environment is not aware of temporary ambiguities, but responses may be delayed.

4.2.8.2 Output-variables

With variables all kind of structures can be built. They can be brought to the output string. If during recognition, parsing or transduction no rhs is applicable for a character it is brought to the output string. This happens as soon as possible. In the case of a phrase-structure grammar the output string will contain (after successful recognition) the start
symbol, accompanied by its variables. These variables are also brought to the output string, in the form of pairs (variable, value). In the case of ambiguities all sets of pairs will be written.

With transduction the same thing happens, but then more symbols can appear in the output, together with their associated variables. Cascaded grammars are treated as a whole: an output string will become visible when it is not denoted as an input string for some other grammar.

A more elaborate example of the use of variables and a lexicon.

In figure 7 we show a piece of a grammar which describes "Simplified English", as defined by Fokker B.V. (trademark), which language is in use for the writing of technical manuals. The grammar is written by B. van der Korst by order of BSO Research. The chosen subset of the unifying formalism is suitable for the implementation of attribute grammars or apsg's ("augmented phrase structure grammars"). The grammar makes use of a lexicon. The parse which has to be constructed is not the internally constructed parse but a dependency tree which is built up in a variable.

Examples of (simple) imperative sentences, written in Simplified English:

1. install the clamp to connect the duct to the duct assembly.
2. do not use a cigarette lighter to find a gas leak.
3. apply the protective compound with a soft-bristle brush on the clean area of the fuselage skin.

Basic rules (only for imperative sentences):

1. rule for imperative sentences:

   IMP:: [ FADJ ]*, [ do,not ], Vrb, [ COM ], [ FADJ ]*.

2. rule for 'free adjuncts':

   FADJ:: Adv | PP | INF | ADVCL.

3. rule for the complement of a verb:

   COM:: AP | NP |
   NP, [ Prt, [ PP | INF ] ]
   Prt, [ NP ], [ PP ]
   PP | INF.

Additions to the basic rules:

1a. actions with variables for the construction of dependency trees
   b. parameters for the passing of subtrees

2. lexicon symbols with attributes; lexicon symbols are preceded by a $'

3. tests with variables for the management of over-generation (congruency, sub-categorization)
4. other rules (e.g. coordination & ellipsis)

IMP(O:tree):=
  [ FADJ(fadj) { fadj1:=fadj1|fadj } ]*.$vrb(vrb,type,compl,prt), [ COM(com,ty,compl,prt) ]
    E { type='intr' & compl=" &prt=" } 1, [ FADJ(fadj) { fadj2:=fadj2|fadj } ]*
    { tree:=[E-GOV,"lvrbl|fadj1|compl|fadj2\|f"] } 1, 'do','not', $vrb(vrb,type,compl,prt), [ COM(com,ty,compl,prt) ]
    E { type='intr' & compl=" &prt=" } 1, [ FADJ(fadj) { fadj2:=fadj2|fadj } ]*
    { tree:=[E-GOV,"ldo'|fadj1|f",E-INT|not,E-INFC,"lvrbl|compl"]|fadj2\|f"] }.

FADJ(O:fadj): [ FADJ(fadj) { fadj1:=fadj1|fadj } ]
  'do','not', $vrb(vrb,type,compl,prt), [ COM(com,ty,compl,prt) ]
    E { type='intr' & compl=" &prt=" } 1, [ FADJ(fadj) { fadj2:=fadj2|fadj } ]*
    { tree:=[E-GOV,"ldo'|fadj1|f",E-INT|not,E-INFC,"lvrbl|compl"]|fadj2\|f"] }.

FADJ(O:fadj): [ Sadv(fadj) ]
  PP(prp,fadj) 1
  INF(fadj) 1
  ADVCL(fadj)
  J1 { fadj1:=[E-CIRC,"ladj1"] }.

COM(O:com,I:type,I:compl,I:prt):=
  [ AP(pred) NP(pred) ] 1 { type="kwa' & com:=\{E-PRED,"l\pred\|f"] } 1
  NP(obj) { type='trans' & com:=\{E-OBJ,"l\obj\|f"] },
    { S.part(p) { p=prt &com:=comll\{E-P,"ipl\|f"] } 1, [ PP(prp,pp) { prp=compl &com:=comll\{E-PREC,"l\pp\|f"] } 1
      PP(prp,pp) { compl=advc &com:=comll\{E-ADV,",l\pp\|f"] } 1
      INF(inf) { compl=inf' &com:=comll\{E-INFC,"l\inf\|f"] } 1
  $prt(p) { p=prt &com:=comll\{E-P,"ipl\|f"] },
  [ NP(obj) { type="trans' &com:=comll\{E-OBJ,"l\obj\|f"] } 1, [ PP(prp,pp) { prp=compl &com:=comll\{E-PREC,"l\pp\|f"] } 1
    PP(prp,pp) { prp=compl &com:=comll\{E-PREC,"l\pp\|f"] } 1
    PP(prp,pp) { compl=advc &com:=comll\{E-ADV,",l\pp\|f"] } 1
    INF(inf) { compl=inf' &com:=comll\{E-INFC,"l\inf\|f"] }.

E:::
A few words in the lexicon:

(the values for variables are written after each word; some words have more values for variables)

compress#Svr#1#into
compress#Svr#3#into
disconnect#Svr#1#from
disconnect#Svr#3#from
do#Svr#1#
does#Svr#1#
writes#Svr#3#no
often#Sadv

Figure 7: an augmented phrase structure grammar for Simplified English

5 OVERVIEW OF EXTENSIONS TO TOMITA'S ALGORITHM

The basic algorithm of Tomita performs online parsing of a linear input string according to a cfg. The cfg is compiled into the usual LR-tables. The parser maintains a dag-structured stack and a parse forest.

When the Parspat system is compared with the basic algorithm of Tomita the following extensions can be observed.

With regard to the formalism:

1a. the BNF notation for Chomsky type-0 grammars: the compiler generates code for lhs's and for rhs's; a second dag-structured stack is added in order to handle type-1 and -0 grammars and transduction grammars which may be cascaded; a set of "connectors" is added which "arbitrate" between the two stacks;
1b. the notation for cover symbols: symbols which are stored in the parse forest are again treated as input symbols;
2. regular expressions: a distinction is made in the parse forest for different items within an itemset; the compiler generates code in order to discriminate between the different items;
3. reserved notions for patterns: these are treated by the compiler, as an extension to LR-theory;
4. the use of variables: the compiler generates symbolic code for operations on variables; because of the regular expressions the values of variables are stored in the first stack at the level of items in an itemset;
5. notations for Boolean constructs: these are treated entirely by the compiler, as an extension to LR-theory;
6. notations for output: reports are stored in the parse-forest; the output of reports and variables and of transduced input is handled by a general processor which prunes the stacks and the forest as soon as possible; when objects are not longer referenced they are brought to the output;
7. notations for input: the parser cooperates with a trie-structured lexicon on external memory;
8. cooperation with a trie-structured external lexicon.

With regard to efficiency:

9. of space: the LR-tables are replaced by (compressed) programs with symbolic instructions; the instructions are interpreted by the parser;
10. of time: the compiler constructs, if possible, a FSM for transduction grammars; in that case the parser only needs to maintain the second stack;
11. of time and space: when no parse has to be delivered to the user the parser maintains in the parse forest only those nodes which are needed to store the output of reports.
6 A PARALLEL TRANSDUCTION AUTOMATON (PTA)

6.1 The evolution of automata and program generators for the grammars in the Chomsky-hierarchy

The usual goal for program generators for Chomsky type grammars is to generate a parser. They are therefore called "parser generators".

There is a close correspondence between the hierarchy of Chomsky grammars and the following formal automata, as depicted in figure 8.

<table>
<thead>
<tr>
<th>sentences generated by a Chomsky grammar of type</th>
<th>can be recognized by a</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>FSA (Finite State Automaton)</td>
</tr>
<tr>
<td>3</td>
<td>FSA</td>
</tr>
<tr>
<td>2</td>
<td>PDA (Push Down Automaton)</td>
</tr>
<tr>
<td>1</td>
<td>LBA (Linear Bounded Automaton)</td>
</tr>
<tr>
<td>0</td>
<td>2SM (Two Stack Machine)</td>
</tr>
</tbody>
</table>

Figure 8: relation between grammars and automata

The machine models

The formal machines which we listed above form a hierarchy, in the same way as the corresponding types of grammars. It will be slightly easier to start the explanation with a 2SM rather than with a FSA. The use of a 2SM for parsing is depicted in figure 9 (from Turnbull (1975)).

A 2SM is composed of an L-stack (a stack), an R-stack (really an output restricted deque (Knuth 1968) ), and a finite control. Initially the input resides in the R-stack, beginning at the left end. The control is designed so that it either shifts a symbol from the left end of the R-stack and pushes it into the L-stack, or applies a rewriting rule to the top of the L-stack.

Figure 9: a 2SM
The latter operation is known as a reduction. It entails removal of symbols from the top of the L-stack and insertion of some related symbols (related by the productions) into the left end of the R-stack. Shift reduce parsers (Aho and Ullman, 1972) operate like this model. Because they are based on CFG's they may only push a single symbol into the R-stack during a reduction operation. Instead of using the R-stack the symbol can be pushed on the L-stack. In that case we are left with a PDA (figure 10).

![Figure 10: a PDA](image)

Various parsing techniques differ only in how the control operates and how it is constructed. Research into parsing based on formal grammars has been directed at increasing the power of the control and, consequently, enlarging the class of languages that can be parsed. LR(k) grammars correspond to the largest set of CF languages that can be deterministically parsed from left to right in this model, looking ahead a bounded number of symbols (=k) in the input. LR(k) grammars correspond to the class of CF languages recognizable by a deterministic PDA.

Still lower in the hierarchy are the type-3 and -4 grammars. For recognition they do not need a stack and we are left with figure 11.

![Figure 11: a FSA](image)

In the parsing methods which we shall review the finite control consists of a number of states. In this case the last machine model consists of a FSA.

### 6.2 Considerations about lookahead calculations

The construction of an LR(0) table is the basis for the construction of an LR(k) table. It is also the basis of the algorithm of Earley (1970) for general CF
parsing in cubic time and of the construction of the
automata of Walters and Turnbull, who generalized
the LR(k) approach for deterministic type-1 and -0
grammars which we will review in the next subsec-
tions.

In this paper we are concerned with the parsing
of generalized, ambiguous type-0 grammars. We are
therefore not interested in classes of grammars which
can be parsed deterministically by using k symbols
lookahead. In our PTA lookahead will be profitable
only for optimization purposes.

In that respect it is useful to observe the relation
between LR(0) and LR(k) table construction. In the
original paper of Knuth (1965) each LR(k) item
contains a lookahead string which is used in every
algorithm related to the construction of an LR(k)
table. In the subsequent literature on LR parsing this
approach was more or less maintained. However,
this is not necessary. The calculation of lookahead
can be performed after the construction of an LR(0)
table. The resulting simplification in the explana-
tion of the construction of LR(k)-tables is worded by
Heilbrunner (1981): "It is not a simple task to give a
convincing explanation of the commonly used LR(k)
definition. There are detailed tutorials on LR parsing
which do not even state it (Aho and Johnson, 1974),
(Horning 1974). The technical apparatus for a
presentation of LR theory along the now traditional
lines of (Aho and Ullman, 1972) involves a lot of
tedious details which may prove the results but
certainly obscures the ideas. A comprehensive and
formal treatment using these methods is contained in
(Geller and Harrison, 1977). We claim that basically
simple ideas and a few clever tricks are sufficient to
explain and prove correct in an intuitively clear and
still formal way Knuth's original LR algorithm
(Knuth, 1965) and DeRemer's (1969,1971) and
Pager's (1977) variants. We shall see that a grammar
is an LR(k) grammar if and only if the
straightforward nondeterministic bottom-up parsing
algorithm for this grammar can be made
deterministic by eliminating superfluous moves***

Superfluous moves are caused by inadequate
states. Our LR treatment of ambiguous grammars in
the Chomsky hierarchy will concentrate on the
treatment of inadequate states in the generated LR(0)
tables. Because the inadequacies can not all be
resolved by lookahead calculations we will resolve
them by efficient parallel parsing. Some lookahead
calculations could be useful in order to improve
efficiency, but they are not necessary.1

In the papers of Walters (1970) and Turnbull
(1975) the original algorithm of Knuth (1965) for
the construction of parsing tables is followed, with
lookahead strings contained within items. Their
algorithms for the construction of parsing tables
collapse to the algorithm for the construction of
LR(0) tables when lookahead calculations are not
taken into account. Again, we are interested in the
parsing of ambiguous type-0 grammars. It seems
therefore appropriate to adopt the algorithm for the
construction of LR(0) tables as our basic algorithm
for the construction of the control of the PTA and to
develop techniques for the calculation of lookahead
for some of the inadequate states later on.

6.3 The relation between the finite
control and the LR(0) table
according to Knuth, Walters
and Turnbull

Knuth, Walters and Turnbull present the working of
the finite control as a sequence of configurations.

A configuration of an LR processor is a pair
whose first component is the stack contents and
whose second component is the unexpended input.
Walters extends this towards a configuration for a
CS processor: a configuration has two stacks, L and
R. The content of the L-stack is $S_0S_1...S_n$ ($S_0...S_n$
are states) and the content of the R-stack is $X_0$. Turnbull adds the current state as a separate third
component. The others denote the current state as the
top of the L-stack, and we will follow that
convention. X is the top symbol on R. For cf
grammars $X \in T_\omega()$ and $\beta \in T^*$, for cs- and type-0
grammars $X \in V_\omega()$ and $\beta \in V^*$.

As for the LR case, the CS processor starts in
the configuration $(S_0, \omega)$, where $\omega$ in $T^*$ is the

---

1 With the extension to type-1 and type-0 grammars
new ways for the treatment of lookahead become
possible. For instance, Turnbull (1975) shows that
in the extension to non-ambiguous type-1 and -0
grammars lookahead can be eliminated by adding
more context to grammar rules. This can be done in
an automatic way, based upon lookahead calculations
for the LR(0) table.
input string. If the automaton reaches the configuration \((S_0S_1...S_nX\beta)\) then it has reconstructed a rightmost derivation of \(\omega\) from \(x_1...x_nX\beta\), where \(x_1...x_n\) in \(V^*\) is the reconstructed rightmost derivation up till now.

We will explain how the four instructions shift, reduce, accept and error in the LR-table direct the control for a CS processor.

A next move of the configuration is determined by the symbol at the top of the L-stack \(S_n\) (the current state), the symbol at the top of the R-stack X and the consultation of \(\text{ACTION}[S_n,X]\) in the LR-table (see figure 12).

According to the result of the consultation the next move is

a. shift S: \[(S_0S_1...S_nX\beta) \Rightarrow (S_0S_1...SnS,\beta)\]

b. reduce \(Y_1...Y_m: \gamma: (S_0S_1...S_nX\beta) \Rightarrow (S_0S_1...Sn-1,Y_1...Y_m\beta),\)

where \(\gamma = r\). For cfg's \(m=1\).

Remarks:
1. In the standard LR-algorithm for cfg's with lookahead calculations chains of reduces always lead to a shift; moreover, \(S_n\) will contain the reducing item with the dot at the end, which is the reason that the X will only be inspected as a part of the lookahead; in that case the next move is defined as \[(S_0S_1...S_nX\beta) \Rightarrow (S_0S_1...Sn-1S,X\beta),\]

where \(\text{ACTION}[S_{n-1},Y]\) is shift S.

2. In the algorithms of Walters and Turnbull for type-1 and -0 grammars it is guaranteed, by the calculation of lookahead, that \(\text{ACTION}[S,X]\) will be a shift.

c. accept : the control stops. \(S_n\) has to be an accepting state.

d. error : the control calls some error reporting-., and eventually, an error-recovery routine.

Figure 12: next move function within a traditional CS-processor

The algorithm for the control itself is very simple: it starts in the configuration \((S_0, a_1...a_n)\) and performs next moves as long as no accept or error shows up.

Walters proves the equivalence of CS(k) grammars and DLBA's: a set of strings is accepted by some DLBA iff it is generated by some CS(k) grammar. In his case the 2SM is reduced to a DLBA. He also proves that the sentences of a CS(k) grammar can be parsed in a time proportional to the length of their derivations.

The control which he constructs for CS(k) grammars need not halt for every input.

Turnbull (1975) studies type-0 grammars for which parsers can be constructed which will detect any errors as soon as possible. These parsers are guaranteed to halt. We quote from (Turnbull, 1975, p. 3-1): "In general terms, our goal is to define and examine the class of languages that can be parsed deterministically in a single left to right scan without backtracking. Since backtracking and multiple passes are inherently inefficient operations, this class is the largest set of languages that are practical to use as programming languages."

Both Walters and Turnbull observe that the steps from one configuration to another can be thought of as grammatical productions, in reverse, operating on the configurations. If no instruction can be applied to a configuration, then an error has been found. Based upon this idea Walters reconstructs from a DLBA a CS(1) grammar, and proves the equivalence.

6.4 The extension of a 2SM to a PTA

A PTA consists of 2 stacks in the form of a dag-structure (directed acyclic graph), a set of connectors and a parse forest (figure 13).

Each connector connects nodes in the L-stack with nodes in the R-stack. In principle a connector is a 4-tuple (active node\(s_L\), projective node\(s_L\), active
nodes\textsubscript{R}, projective nodes\textsubscript{R}).\textsuperscript{1} (The indexes L and R refer to the L- and R-stack respectively). Active nodes project themselves, on the reading of an input symbol, to projective nodes. The number of possible projective nodes is limited and is calculated by the compiler.

The operations on both stacks are in essence the same. The usual instructions for a stack, like pop, push, inspect top and change top are maintained. If an active node\textsubscript{L} is connected to an active node\textsubscript{R} then the intersection is taken of the two sets of possible continuation symbols. These symbols are found in the LR-tables for the compiled states which are attached to these nodes. For the symbols which are in common the corresponding actions in the LR-table are executed for the two nodes. After zero or more reduces a shift will follow, resulting in a push on the stack. This is performed by the creation of a new leaf in the dag and the creation of an edge from the last active node to that leaf node. All leaf nodes which result from actions taken for nodes in one active set (in the same dag) form a projective set. Before a new leaf node is created it is checked whether there already exists a leaf node in the projective set with the same number of a compiled state. In that case no new leaf node is created but an edge is formed from the last active node (the last one in a possible chain of reduces) to that already existing leaf node in the set of projective nodes. (This corresponds with "Earley parsing"). A reference to the resulting set is stored in a new connector.

A third dag is automatically maintained (if necessary): the parse forest. This parse forest will be built on-line and reflects the current status of all valid parse trees. In the case of TDG's (transduction grammars) it will be used also as an association list for the nonterminals which appear in a rhs. Evaluation of these nonterminals will only happen when necessary ("lazy evaluation").

In the parse forest the report-output finds its natural, but temporarily, place. If the parse forest has become one single parse tree then this output will be brought automatically to the outside world and released from the parse tree.

The way of storing of the report-output may be called: "deep binding". It is not necessary to reference these values until they have to be brought to the outside world. This in contrast with the variables. They are stored with "shallow binding", with a copying of the reference to their values from stacknode to stacknode.

Garbage collection is done entirely "on the fly" by keeping reference counts for each dagnode and its associated substructures. When the PTA has treated the endmarker in the input no datastructures are left.

The reference counts play a special role during transduction: when the reference count of a stacknode in L becomes 0 this is an indication that at that point no more transductions can be performed; the associated symbol will then be brought to the output file, together with the coordinates of its neighbours. In the case of ambiguous TDG's a dag with symbols from translations is in that manner brought to the outside world.

The translation is then produced with a finite delay (see also section 2.2.1). With finite delay we mean that output of the transducer becomes available after the reading of a finite number of input symbols, while the total number of characters in the input can be indefinite.

It is now possible to relate to the PTA the machine models which we described above and the parsing methods which make use of them.

The PTA can be seen as a general machine model which can be reduced to the former machine models. The 2SM consists of linear L- and R-stacks and of the finite control. A LBA, a PDA and a FSA are simplified models of a 2SM. The PTA extends the 2SM in the following ways:

- the L- and R-stack both become a dag
- a third dag is added for the storage of a parse forest
- connectors are added which connect sets of active and projective nodes in the L-and R-dag with each other
- storage is added for reports and variables.

The purpose of the sets of active and projective nodes is that they serve to maintain a polynomial runtime complexity for each connector. If there is only one stack (the L-stack) only one connector is functioning. In that case we get back Tomita's

\textsuperscript{1} We will see later that connectors also have references for variables and the lexicon.
The Parspat compiler only generates code for the smallest machine model that is possible. For the runtime behaviour the lowest bounds are valid which are achieved at this moment in theoretical research for the different sub-formalisms (as far as constructive methods are concerned). Furthermore, the PTA can be influenced by user-heuristics in order to decrease the number of ambiguous transductions.

These heuristics concern the treatment of inadequate states. In the LR-table of an inadequate state there may be denoted for a symbol one shift and a multiple number of reduces, to be subdivided in reduces for cf-rules and type-1 or -0 rules. The user of the PTA may indicate with global switches if a choice has to be made, and which kind of choice.

Code will be generated for the following (possibly combination) of sub-formalisms.
- for a FSA: Chomsky-type 3 or 4 with regular expressions, Arbs don’t care and arb and line, Booleans;
- for a PDA: Chomsky-type 2 with regular expressions, no ambiguity, don’t care and arb and line;
- for a PTA with 1 dag-stack: Chomsky-type 2 with regular expressions, all ambiguities, don’t care and arb and line, Booleans, variables;
- for a PTA with 2 dag-stacks: Chomsky-type 0 or 1 or transduction with regular expressions, all ambiguities, don’t care and arb and line, Booleans, variables.

The adjective “parallel” in “Parallel Transduction Automation” has two meanings:

1. The execution of code which was produced automatically for a formalism in which, conceptually, each rule operates in parallel with other rules.
2. The execution of this code on a parallel machine.

Meaning 1 has nothing to do with meaning 2. As we will see shortly the execution of the code can be done wholly on a sequential machine. But the machine can be organized in such a way that as much as possible can be done in parallel. The organization of the parallel processes does not stem from the inherent parallelism of the formalism but from the formal description of the instructions of the machine. In such a way an important gain in efficiency is achieved.

For the compact reference of paths within a dag we use the concept of a condensed notation and a condensed path.

In a condensed notation we denote for a node zero or one successor. For instance, the condensed notation for the dag

is \((1, 2), (2, 6), (3, 5), (4, 5), (5, 6), (6, e), (7, e))\).

A condensed path contains a pair with the beginning and end-node and it contains zero or more pairs of exceptions: if the successor of a node in the path is different from the successor which is named in the condensed notation of the dag then the node and its successor in the path have to be denoted as an exception.

Thus the path (1, 3, 5, 6) is represented as a condensed path by \((1, 6), (1, 3))\). The begin and the end of the condensed path are 1 and 6; at 1 has to be chosen as its successor 3 and not 2, which was specified in the condensed notation of the dag.

The use of a condensed notation is motivated by reasons of economy: long paths in a dag with nodes with low degrees are referenced substantially shorter.

The nodes of the dags contain information, which are described precisely in (Van der Steen, 1988). We give there also an elaborate example of the working of a PTA for an highly ambiguous cfp with regular expressions and reports, together with pictorial representations of the dag structures during input.

6.5 Transduction with a PTA; example

We will now discuss informally how transductions are performed with the PTA.

On-line recognition with a 2SM can be extended towards transduction with finite delay. We compute LR-tables for the lhs’s and the rhs’s of the rules. We choose an example for which the generated states are adequate. In that case the dag-structured L- and R-stacks of the PTA remain flat.

Our grammar is:

\[ b, a \rightarrow a, b \]

The compiled LR(0) table for the left hand sides of the grammar is:

**State 1:**

items: \(-1\) b, a \rightarrow a, b.

Table entry for shifts: generate "b" and goto state -2
(in runtime: "push state -2 on the R-stack")
State -2:
items: b, -2 a :: a, b.
table entry for reduces: generate "a" and reduce (in runtime: "go back to the state where this item originated")

The compiled LR(0) table for the right hand sides of the grammar is, per state:

State 1:
items: b, a :: 1 a, b.
table entries:
a : shift 2 (in runtime: "push state 2 on the L-stack")
. (every other character) : shift 1

State 2:
items: b, a :: 1 a, b. (all transduction rules may start in each state)
  b, a :: a, 2 b.
table entries:

---

L-stack:

```
  0  2  1
```

R-stack:

```
-1  0
```

Read 1st symbol 'a': shift to state 3(2)

```
  0  2  1  3  2
```

Read 2nd symbol 'a': shift to state 4(2)

```
  0  2  1  3  2  4  2
```

Read 3rd symbol 'b': reduce and put state -2(-1) on the R-stack

```
  0  2  1  3  2
```

The entries on a stack have a reference count. If a reference count becomes 0 the entry is deallocated. If an entry at the bottom of the L-stack is deallocated, then its associated symbol is emitted. By assigning unique labels to the emitted symbols (e.g. increasing integers) a dag may be outputted (in the case of an ambiguous grammar).

In our example only the essential entries are shown. Not shown are the items in each entry which reference the entry where they originated. If there are no more items which reference an entry than that entry will be deallocated.

We show the on-line operation of the generated parser for the input "aab..." in figure 14.
Generate from state -2(-1) symbol 'b' as input for state 3(2)  
This causes a shift in state -2(-1) to state -3(-2) and a reduce in state 3(2)  
Therefore put state -4(-1) on the R-stack

```
1 0 2 1
```

Generate from state -4(-1) symbol 'b' as input for state 2(1)  
This causes a shift in state -4(-1) to state -5(-2) and a shift from state 2(1) to state 5(1)

```
1 0 2 1 5 1
```

*State 2(1) is not longer referenced. The symbol 'b' will be emitted.*  
Generate from state -5(-2) symbol 'a' as input for state 5(1)  
This causes a reduce in state -5(-2) and a shift from state 5(1) to state 6(2)

```
1 0 5 1 6 2
```

Generate from state -3(-2) symbol 'a' as input for state 6(2)  
This causes a reduce in state -3(-2) and a shift from state 6(2) to state 7(2)

```
1 0 5 1 6 2 7 2
```

The 2nd stack is empty. We resume reading from the input.

Figure 14: on-line operation of the generated parser for a non-ambiguous Transduction Grammar

Up till now the emitted output is:

```
1 b 5
```

where the superscripted digits denote the unique labels of the entries which were deallocated.

Two 'a's are waiting on the stack for a possible transduction.

Note that the entries on the R-stack act as generators of symbols. That is the reason why regular expressions, arbs, lines, ranges and Boolean negations at the lefthand side of type-0 rules cause no problems.\(^1\)

For the case of ambiguous type-0 grammars the reader may imagine for the 2 stacks dag-structures. If a shift/reduce-reduce conflict is encountered a node in the 2nd stack will project into more directions, which effect is responsible for the creation of a tree-structure. If the lhs of a rule contains a regular expression then the tree-structure may become a dag-structure.

7 EXTENSIONS TO THE LR-TABLE CONSTRUCTION METHOD

For the following sub-formalisms we made an extension to the algorithm for the creation of LR-tables:

- regular expressions
- concerning symbols:
  - ranges of terminal symbols
  - "don't cares"
  - "arb's" and "lines"
- concerning Boolean expressions:
  - negations.

\(^1\) However, up till now we did not meet an application for them.
These extensions are given in full detail in (Van der Steen, 1988) and imply new lower bounds for pattern matching, combined with an integration within LR-parsing. For instance, the recognition of don't cares may now be done in linear and even in sub-linear time.

8 ACKNOWLEDGEMENTS

The author gratefully acknowledges the support of his former colleagues at the Faculty of Arts of the University of Amsterdam, particularly Marijke Elstrodt, Pieter Masereeuw, Joos Skolnik, Bieke van der Korst, Jan de Jong and Willem Meijs.

9 REFERENCES


Elstrodt, M. et al. (1990), Users Manual for the Parspat System, Computer Department Faculty of Arts, University of Amsterdam, about 100 pp.


Van der Steen, G.J. (1988), A program generator for recognition, parsing and transduction with syntactic patterns. CWI Tract 55, Centre for Mathematics and Computer Science, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands, 284 pp.


SUBSTRING PARSING
FOR ARBITRARY CONTEXT-FREE GRAMMARS

Jan Rekers
Centre for Mathematics and Computer Science
P.O. Box 4070, 1009 AB Amsterdam, The Netherlands
email: rekers@cwi.nl

Wilco Koorn
Programming Research Group, University of Amsterdam
P.O. Box 41882, 1009 DB Amsterdam, The Netherlands

abstract

A substring recognizer for a language \( L \) determines whether a string \( s \) is a substring of a sentence in \( L \), i.e., \( \text{substring-recognize}(s) \) succeeds if and only if \( \exists \), \( w: usw \in L \). The algorithm for substring recognition presented here accepts general context-free grammars and uses the same parse tables as the general context-free parsing algorithm from which it was derived. Substring recognition can be useful for noncorrecting syntax error recovery and for incremental parsing. By extending the substring recognizer with the ability to generate trees for the possible contextual completions of the substring, we obtain a substring parser, which can be used in a syntax-directed editor to complete fragments of sentences.

1 INTRODUCTION

A recognizer for a language \( L \) determines whether a sentence \( s \) belongs to \( L \). A substring recognizer performs a more complicated job, as it determines whether \( s \) can be part of a sentence of \( L \).

A recently developed substring recognition algorithm [Cor89] uses an ordinary LR parsing algorithm with special parse tables. For ordinary parsing, this parsing algorithm is limited to LR(1) grammars, but the more complicated nature of substring recognition limits it to bounded-context grammars (see Section 3).

We describe a substring recognition algorithm that does not suffer from this drawback. It accepts general context-free grammars and uses the same parse tables as our ordinary parser. Our algorithm is based on the pseudo-parallel parsing algorithm of Tomita [Tom86], which runs a dynamically varying number of LR parsers in parallel and accepts general context-free grammars. Next, we extend the substring recognizer into a substring parser that generates trees for the possible completions of the substring.

2 APPLICATIONS

Before discussing existing proposals to substring parsing (Section 3) and our approach to it (Section 4), we mention some applications of the technique.

2.1 Syntax error recovery

In its simplest form, a parser stops at the first syntax error found. If it has to find as many errors in the input as possible, it can try to correct the error in order to continue parsing. Spurious errors are easily introduced, however, if the parser makes false assumptions about the kind of error encountered.

Substring parsing can be used to implement noncorrecting syntax error recovery. If an ordinary parser detects a syntax error on some symbol, the substring parser can be started on the next symbol to discover additional syntax errors. Using this method, it is not necessary to let the parser make any assumption about how to correct
the error, or to let it skip input until a trusted symbol is found.

Richter defines noncorrecting syntax error recovery with the aid of substring parsing and interval analysis in a formal framework [Ric85]. He proves that his technique does not generate spurious errors, but is not explicit about its implementation. He notes, however, that there are difficulties in keeping the substring parser deterministic due to a limitation on the class of grammars accepted. Our technique could be useful here, as it implements the required substring analysis for general context-free grammars.

2.2 Completion tool

In Section 5 we will show how the substring recognizer can be extended such that it generates parse trees for the possible completions of a substring. As the total number of possible completions will often be infinite, only generic completions are generated. A syntax-directed editor could use these to complete fragments of sentences in accordance with the grammar used, or to guess the continuation of what the user is typing.

Snetting presents a technique to complete the right-hand side of unfinished sentences [Sne80]. We will discuss parts of his method in Section 5.3.

2.3 Incremental parsing

Another application for substring parsing is incremental parsing. An incremental parser builds the parse tree for the current version of its input text while it re-uses the parse tree generated for the previous version as much as possible. We will first sketch two possible solutions for the problem of incremental parsing, and next suggest a third solution based on substring parsing.

Re-use parser states

Incremental parsing can be performed by attaching parser states to tokens [Cel78, AD83, Yeh83]. After a modification has been made, the parser is restarted in a saved state, at a point in the text just before the modification. Parsing stops when the parser reaches a token after the modification in an old configuration (if ever).

These methods are very good as to minimizing the amount of recomputation after a modification, but require a huge amount of memory for storing the states of the parser (parse stacks with partial parse trees as elements).

Abbreviate sentence

Ghezzi and Mandrioli present an alternative technique for incremental parsing [GM79, GM80]. If the string \( z \hat{x} \hat{y} \) is modified to \( z \hat{x} \hat{y} \), where \( \hat{x} \) and \( \hat{y} \) have length \( k \), with \( k \) the look-ahead used by the parser, then the parse trees previously generated for \( z \) and \( y \) are still valid after the modification. All subtrees previously generated for \( x \) and \( y \) can thus be abbreviated by their top non-terminals, which minimizes the length of the string to be reparsed.

This technique is both time and space efficient, but is not applicable to general context-free parsing as it requires a fixed look-ahead. In our particular case, we need incremental parsing in a syntax-directed editor that uses the Tomita parser. By running a varying number of LR-parsers in parallel, the Tomita parser adjusts its look-ahead dynamically to the amount needed, and is thus not limited to an a priori known \( k \).

Reparse a subtree only

Incremental parsing can also be achieved in another manner: after a modification has been made in the text, find the substring \( s' \) belonging to the smallest subtree that contains the modification in the stored parse tree. If the type of this subtree is \( T \) and \( s' \) can be parsed as a tree of type \( T \) also, the old subtree can be replaced by the new one. If \( s' \) fails to parse, it may be the case that the modification introduced a syntax error, or that the subtree has been chosen too small. These two cases must be distinguished, as the incremental parser proceeds in a different way in each case.

A substring parser can provide a hint as to which of the two possibilities is actually the case. If the substring parser fails on \( s' \), the modification will be syntactically incorrect in any context, and an error message can be given. If the substring parser succeeds, a larger subtree is chosen and parsing is retried.

This can be more time consuming than remembering parser states, but the amount of memory needed is far less. We consider using this scheme in the syntax-directed editor GSE [Koo], but it has to be investigated further as the technique still performs a lot of work twice.
3 RELATED WORK

Cormack [Cor89] describes a substring parse technique for Floyd’s class of bounded context or BC(1,1) grammars [Flo64], and implements the substring parser Richter mentions [Ric85]. A grammar is BC(1,1) if for every rule \( A ::= \alpha \beta \) some sentential form contains \( a\alpha b \) where \( \alpha \) is derived from \( A \) then \( \alpha \) is derived from \( A \) in all sentential forms containing \( a\alpha b \). This class is smaller than LR(1). The solution of Cormack consists in using an ordinary LR automaton, but a special parse table constructor. The sets of items generated do not only contain items of the form \( A ::= \alpha \gamma \beta \) but also “suffix items” of the form \( A ::= \ldots \gamma \beta \). These suffix items denote partial handles whose origins occur before the beginning of the input. The generated parse tables are deterministic, provided that the grammar is BC(1,1). This substring parser is used for noncorrecting error recovery in a parser for Pascal. The BC(1,1) limitation on the grammar caused problems in the definition of Pascal, which where alleviated by permitting the parse table generator to rewrite the grammar if necessary.

Lang describes a method for parsing sentences containing an arbitrary number of unknown parts of unknown length [Lan88]. The parser produces a finite representation of all possible parse trees (often infinite in number) that could account for the missing parts. The implementation of this method is based on Earley parsing [Ear70]. The basic idea of Lang’s method is that “in the presence of the unknown subsequence “\(*\)”, scanning transitions may be applied any number of times to the same computation thread, without shifting the input stream.” This process terminates, as parsers in the same state are joined and the number of states is finite.

This method is very elegant and powerful, and can be used as a substring parser (by providing it with the string “\(*\)”). We will not use it, however, as it is more general than what we need. Whether it would be sufficient enough for interactive purposes is unclear.

4 SUBSTRING RECOGNITION

4.1 Tomita parsing

We base the implementation of our substring parser on Tomita’s parsing algorithm. This algo-

4.2 The grammar

The grammar according to which the substring recognition algorithm works, should not contain useless symbols. According to [HUr79, p. 88], a symbol \( X \) is called useful if there is a derivation \( S \Rightarrow aX\beta = \varepsilon w \) for some \( \alpha, \beta \) and \( w \), where \( w \) is in \( T^* \). Useless symbols can be identified easily, and all rules in which they appear should be removed from the grammar. Such a clean-up operation does not affect the language recognized.

Unreachable symbols and rules do not influence our method of substring parsing, as these are already ignored by the parse table generator. This is due to the fact that LR parse tables are generated top-down, starting with the start symbol of the grammar, and that unreachable symbols and rules are, by definition, unreachable from the start symbol.

Symbols and rules which cannot produce any terminal string should be removed from the grammar however. These can cause the substring parser to succeed on a string \( s \), while no string \( uvw \) exists in \( L \).

\footnote{Grammars in which \( A \Rightarrow \epsilon A \) is a possible derivation.}
4.3 The algorithm

If we have to determine whether a string $a_0 \cdots a_n$ is a substring of a sentence in a language $L$, we start the substring recognition process by generating, for each state directly reachable under $a_0$, a parser with this state on its stack. These parsers will process $a_1 \cdots a_n$.

We will show how an individual parser processes an action, but we will not discuss the management of the different parsers, as this is done in the same way as in ordinary Tomita parsing.

The parser obtains an action from the parse table with the state on top of its stack and with input symbol $a_k$. This can be a shift, error or reduce-action, and is processed in the following manner:

- A (shift state')-action is processed as in normal parsing: state' is pushed on the stack and the parser is ready to process $a_{k+1}$.
- An (error)-action removes the parser from the set of active parsers.
- A (reduce $A ::= \alpha\beta$)-action is processed as follows:
  - If there are at least $|\alpha\beta| + 1$ entries on the parse stack the reduce action is performed as in normal parsing: $|\alpha\beta|$ entries are popped off the stack, and the parse table is consulted, with the state remaining on top of the stack and $A$, to obtain a state to push on the stack again. The parser is now ready to continue the processing of $a_k$.
  - If there are only $|\beta|$ entries on the stack, only $\beta$ has been recognized of $A ::= \alpha\beta$; $\alpha$ lies before $a_0$ and should produce (a part of) a prefix of $a_0$. This is possible, as all non-terminals in $\alpha$ can produce some terminal string, and all terminals in $\alpha$ trivially do. So the reduction $A ::= \alpha\beta$ may be performed, and parsing may continue in the states which can be reached directly by a transition under $A$. For each of these valid states a new parser is started with that state on the stack. These parsers all proceed to process $a_k$.
  - If there are exactly $|\alpha\beta|$ entries on the stack, $a_0 \cdots a_{k-1}$ reduces to $\alpha\beta$, but the context in which $A$ is to be used is unknown. This is handled in the same way as the previous case.

If there are no parsers left alive after the processing of $a_n$, the substring parser fails. If there are parsers left, these are currently recognizing rules $A ::= \alpha\beta$, of which (a part of) $\alpha$ has been recognized. As every $\beta$ can produce some terminal string, these rules can all be finished. This means that the substring parser succeeds if there are parsers remaining after the processing of $a_n$.

4.4 The parse table generator

The substring parser is controlled by the same parse table as our ordinary parser. To generate this parse table we use an extended version of the lazy and incremental parser generator IPG [HKR89]. The extension concerns the need of the substring parser to know all states which can be reached by a transition under a given symbol. This function needs global information about the parse table, which means that the whole parse table must be known. As a consequence, the lazy aspect of IPG cannot be exploited here and the parse table will always be fully expanded. The expanded parse table can of course also be used by the ordinary parser.

5 Substring Parsing

We extend the substring recognizer into a substring parser by generating parse trees for substrings. The possible parse trees for a substring $s$ are the parse trees of all sentences $uav$, for which $uav \in L$ holds. To limit the number of completions we allow $v$ and $w$ to consist both of terminals and non-terminals, and we generate a parse tree, corresponding to a sentential form $\sigma_1 \sigma_2$, only when the frontier of each of its subtrees contains at least one symbol of $s$; i.e., we do not generate subtrees whose frontier lies entirely within $\sigma_1$ or $\sigma_2$. The trees that we generate are the most general trees, as it is not possible to replace any of their subtrees by a non-terminal such that the frontier of the whole tree still contains $s$ as a substring. Even so, the number of completions can still be infinite. In Section 5.3 we will discuss how to limit this number still further.

5.1 Example of a completion

For the grammar of Figure 1 and the string ") + 5 then if", a possible completion is the sentential
START ::= Stat
START ::= Exp
Stat ::= if Exp then Stat
Stat ::= if Exp then Stat else Stat
Stat ::= Id := Exp
Exp ::= Id
Exp ::= Int
Exp ::= Exp + Exp
Exp ::= Exp * Exp
Exp ::= ( Exp )

Figure 1: A grammar

START

| Stat
| if Exp then Stat
| Exp + Exp if Exp then Stat
| ( Exp ) Int(5)

Figure 2: A completion of "") + 5 then if"

form

if ( Exp ) + 5 then if Exp then Stat

whose parse tree is given in Figure 2. To distinguish the leaves of s from those of \( \sigma_1 \) and \( \sigma_2 \), the former are underlined.

5.2 Generating the completions of a substring

LR parsers generate parts of parse trees during a reduction step. On reducing \( A ::= \alpha \), the parse stack contains the subtrees created for \( \alpha \). These are assembled in a new node of type \( A \) and the subtree created in this way is pushed on the stack. In the substring parser ordinary reductions are treated in the same way.

If the rule \( A ::= \alpha \beta \) is reduced with only nodes for \( \beta \) on the stack, however, additional nodes are created for \( \alpha \). In this way, the parse trees for the possible prefixes of \( s \) are created.

Parse trees for postfixes of \( s \) are created in the same way: after processing \( s \), the parser has to finish all rules which are in the process of being recognized. These are the rules in the kernel of the current state of the parser. If only \( \alpha \) has been seen from a rule \( A ::= \alpha \beta \), the rule is reduced and additional nodes are created for \( \beta \). It can even be the case that only \( \beta \) has been recognized from a rule \( A ::= \alpha \beta \gamma \), and that nodes must be created for both \( \alpha \) and \( \gamma \).

5.3 Further reduction of the number of possible completions

By producing only parse trees that are most general, the number of possible completions is reduced, but it is often still too large and not even always finite. We propose the following rules to limit this number still further:

1. The parse trees generated are kept as compact as possible by disallowing reductions of rules of the form \( A ::= \alpha A, A ::= \alpha A \beta, \) and \( A ::= A \beta \), where only \( A \) has actually been recognized and all elements of \( \alpha \) and \( \beta \) would produce elements in \( \sigma_1 \) or \( \sigma_2 \). Clearly, such reductions can be repeated infinitely often. They are undesirable as they only enlarge \( \sigma_1 \) or \( \sigma_2 \).

For example, the substring ") + 5 then if" also has a possible completion

if Exp + ( Exp ) + 5 then if Exp then Stat

whose parse tree is given in Figure 3. In this tree a subtree for the rule Exp ::= Exp + Exp has been inserted in the prefix.

2. The number of possible sentential forms for which parse trees are generated is now finite, but these can still have infinitely many parse trees as the grammar may be cyclic. Rekers describes how to parse and generate parse graphs.
for cyclic grammars [Rek]. The cycles generated in this graph can be removed by his routine \textit{remove-cycles}. The same approach can be used for substring parsing, and this results in a finite number of most general completions.

3. In the generation of the postfixes of \( s \) a choice can be made for the "simplest" completion. That is, if a substring can be completed according to both \( A ::= \alpha\beta \) and \( A ::= \alpha\gamma \), and \( |\beta| < |\gamma| \), we prefer \( A ::= \alpha\beta \). In the example of Figure 2 this rule forbids the choice of the "if-then-else" rule, as the "if-then" rule already applies. Snelting's rule "\textit{prefer reduce items over shift items}" [Sne90] is similar to ours. His rule can also be formulated as: if completion according to both \( A ::= \alpha \) and \( B ::= \alpha\gamma \) \((\gamma \neq \epsilon)\) is possible, then prefer \( A ::= \alpha \). We consider our rule more appropriate, as we take the case of \( \beta \) being non-empty but shorter than \( \gamma \) into account as well, and we only make the choice if the two rules reduce to the same non-terminal. Otherwise, the rule \( A ::= \alpha \) might be preferred over \( B ::= \alpha\gamma \), whereas the environment in which the substring is completed needs a tree of type \( B \).

6 MEASUREMENTS

Our first measurement compares the substring recognizer with the Tomita recognizer from which it was derived to learn the additional costs of substring parsing.\(^1\)

We have taken a grammar of about twenty rules and sentences of increasing length. These were parsed by the Tomita recognizer first. The resulting parse times are indicated in Figure 4 with a "\textit{\textsuperscript{T}}". Next, the same strings minus a randomly chosen prefix were given to the substring parser. The required times are indicated in Figure 4 with a "\textit{\textsuperscript{S}}".

The measurements show that the substring parser has a moderate overhead with respect to the normal parser. This overhead can be interpreted as the time needed for the substring parser to get on the "right track". As our next measurements show, the variations in this overhead are caused by the random cutting of the string. For some strings it takes longer than for others to dete-

termince of which language construct it can be a substring. The larger the grammar is, the more alternatives are available and therefore the higher the variation.

In Figure 5 we compared the time taken by the substring parser on 30 randomly chosen parts of Pascal sentences of 100 tokens. The dots indicate the amount of time needed and they are attributed with the first symbol of the substring. These measurements show that sentences starting with a token that can appear in many different contexts, like "\textit{Id}" or "\textit{)}", take more time to recognize than sentences starting with a disambiguating token like "\textit{:=}" or "\textit{else}".

7 CONCLUSIONS

The adaptation of the Tomita algorithm to substring parsing results in a very elegant and powerful algorithm. The main advantage of the fact that it accepts general context-free grammars and uses ordinary LR parse tables is that substring parsing can now be applied in a very general manner, instead of only to carefully written grammars and at the cost of an extra generation phase.

Substring parsing is slower than ordinary parsing, but this will not be a serious drawback for its application as an error recovery technique or as a completion tool. The use of the substring parser in incremental parsing, however, has to be investigated further.

ACKNOWLEDGMENTS

We would like to thank Nigel Horspool, who suggested to extend our implementation of the Tomita algorithm to substring parsing. Two years after this discussion we finally saw the need for such a technique and started a serious investigation. Next, we are grateful to Paul Hendriks who pointed out a valuable simplification in the treatment of incomplete reductions in the substring parser, and to Jan Heering for his careful reading of earlier versions of this paper.

\(^1\)The measurements have been performed on a SUN SPARC station. The programs have been written in Lisp. The time used by the lexical scanner has not been taken into account.
Figure 4: Comparison of the substring recognizer with an ordinary recognizer

Figure 5: Time needed by the substring parser on Pascal sentences of 100 tokens
REFERENCES


DETECTION AND CORRECTION OF MORPHO-SYNTACTIC ERRORS IN SHIFT-REDUCE PARISING

Theo Vosse

Nijmegen Institute for Cognitive Research and Information Technology
University of Nijmegen, Montessorilaan 3, 6525 HR Nijmegen, The Netherlands
email: vosse@nici.kun.nl

ABSTRACT
In this paper we will describe a systematic approach to detecting and correcting morpho-syntactic errors using a shift-reduce parser for Augmented Context-free Grammars. In particular adaptations to the Tomita parsing algorithm are studied.

1. INTRODUCTION
One of the most practical uses, if not currently the most practical use, of Natural Language Processing is grammar checking. In doing so there is no great need to take the semantics of natural language into account, at least not if we adhere to strict grammar checking only. Successful grammar checking requires three things: a comprehensive grammar, an efficient parser and a mechanism to detect and correct grammatical errors and to help correcting spelling errors. Assuming we have a shift-reduce parser at our disposal this paper describes how an error detection mechanism can be built that operates grammar-independently although the effectiveness of the mechanism strongly depends on the grammar’s quality. It is assumed that grammatical error correction can only be achieved if the parse tree the parser produces is as close to a grammatical tree as possible.

First we will discuss the types of errors we intend to detect and possibly correct, and the need for a parser. Then we will turn our attention shortly to Augmented Context-Free Grammars and shift-reduce parsers for these grammars with an eye towards applying them in error detection. Next we will discuss the adaptations to the Tomita algorithm needed to efficiently deal with erroneous input and take a look at how to use the parse results to correct the errors. Finally we will evaluate some of the results of this approach that we have achieved in using this scheme in a Dutch grammar checking system.

2. MORPHO-SYNTACTIC ERRORS
A large number of grammatical and morpho-phonological errors can be distinguished. Most of them are however beyond the scope of this paper. These are errors regarding the style and meaning of words and sentences, or errors that go beyond word and sentence level, such as wrong usage of deixis or language interference.

We will concern ourselves here with those errors which can be categorised under the name “morpho-syntactic” errors. In general these errors are related to the derivation and inflection of words. These errors are by far the most frequent and is therefore an interesting area for grammar checking and educational applications. We will now give a more detailed classification of the morpho-syntactic errors and some related errors.

2.1. Agreement violations
A typical syntactic error is agreement violation. Though none of the words in the sentence “she walk home” is incorrect, the sentence is ungrammatical. No simple spelling checking mechanism can detect the error, let alone cor-
rect it, since it is caused by a relation between two words that need not be direct neighbours. Detection and correction of this type of error thus requires a parser that can handle ungrammatical input.

2.2. Spelling and typing errors
The two most general types of error at word level are spelling errors and typing errors. The first is caused by a lack of competence whereas the second type is caused by a lack of performance. This results in two typical error patterns.

Wrongly spelled words often have a pronunciation that is roughly equivalent to that of the intended word. Usually this is caused by the deletion or insertion of a consonant or by the substitution of similar sounding vowels or diphthongs, e.g. onmiddelijk (should be onmiddëlijk) olieën (should be oliën) or rijlen. (should be reilen). Transposition typically does not occur.

Wrongly typed errors resemble the intended word in a different way. Insertion, deletion and substitution are related to the position of the characters on the keyboard and the sequence of the characters that make up the word. Transposition of two adjacent characters occurs rather often. Typically the pronunciation of a typing error does not even resemble the pronunciation of the intended word.

The most obvious characteristic of these two types of errors is that they often, but not always (see below), result in a sequence of characters that does not occur in a dictionary. Therefore even a simple spelling checker might notice them. However, most spelling correctors come up with more than one alternative spelling (it sometimes even seems that a corrector is considered better the more alternatives it suggests). So in order to maintain only the grammatical plausible alternatives a parser that can handle spelling errors is required.

Homophones words
An important subcategory of spelling errors and a frequent source of spelling and therefore grammatical errors are homophones words: words having the same pronunciation. Dutch examples are zee and zíj, sectie and sexy, word and wort and achteruit and achteruit. Such words can easily be replaced by one of its homophones counterparts in written text since the difference cannot be heard during dictation or because the difference is simply not known.

The greatest problem for current spelling checkers is again that this substitution goes unnoticed as the substitutes are legal words themselves. If this type of error needs to be detected, a parser is required since it often involves a change of syntactic category. We will see that the treatment of these errors strongly resembles the treatment of spelling errors. Unfortunately, a parser cannot detect all substitutions by homophones since a number of them have the same syntactic properties.

We will not discuss the errors caused by a typing error resulting in an existing word (such as rotsen and rosten), but their treatment of course is similar.

Homophones inflections
A special case of homophone words are words which differ only in inflection. This type of homophony is very frequent in Dutch and French. French examples are donner, donnez, donné, donnée, donnés and données or cherche, cherchez and cherchant. Dutch examples typically involve dit- errors, since -d-, -t- and -d- sound similar at the end of a word, while they often indicate some form of verb inflection. Examples are the forms gebeurt (third person singular, present tense) and gebeurd (past participle) of the verb gebeuren, word (first person, singular, present tense) and wordt (third person, singular, present tense) of the verb worden or besteden (infinitive and plural, present tense) and besteedden (plural, past tense).

However, unlike the general case of homophone words, homophones inflections, by their very nature, do not alter the syntactic category of the word, but rather its (morpho-syntactic) features. So we can regard this type of error as a homophone word or a spelling error, or as an agreement violation.
2.3. Structural errors
Generally speaking, structural errors such as constituent substitution, deletion or insertion do not very often occur in normal text, at least if we define the constituents at the normal level of NP, PP, VP etc. Specific types of structural errors which do occur frequently are described below.

Punctuation
When punctuation elements are considered to be constituents, punctuation errors are structural errors indeed. Most often these are caused by a superfluous use of commas as in “The boy saw, the girl, on her bike, yesterday”. Another elementary error is the deletion of the period at the end of a sentence. This requires that a parser should be prepared to deal with such minute insertions and deletions.

Constituent Order problems
The order of the words and constituents in a sentence will usually be correct. One typical example in which this might not be the case is the French adjective ordering, e.g. in “la maison grande” or “le rouge moulin”. This type of error is most typically made by non-native speakers. It can be seen as a feature violation error, but also as a structural error.

Word doubling
“Did someone actually spot the error in this sentence?” One of the most difficult errors to spot is word doubling. Especially at the end of a line it goes completely unnoticed. A parser surely notices it, but it should not break down because of this.

Errors in idiomatic expressions
Idiomatic expressions often cause problems for parsers since they do not have a regular syntactic structure and some of its words may be illegal outside the idiom. A Dutch example is te allen tijde (English: at all times), in which the word tijde only occurs within idiomatic expressions. Whenever it occurs in a normal sentence it must be considered to be a spelling error. An English example might be in lieu of. An extra problem is caused by spelling errors in idiomatic expressions. E.g. the expression above is more often than not written as te alle tijden, which consists of legal words and is syntactically correct as well.

Compounds
Somewhat similar to idiomatic expressions is the case of compound nouns, verbs, etc. In both Dutch and German these should be written as single words. However, under influence of the ever advancing Americanisation especially newly made-up compounds, such as tekstverwerker (text processor) and computer terminal are written separated by blanks, which would normally cause a parser to fail.

3. AUGMENTED CONTEXT-FREE GRAMMARS
Augmented Context-Free Grammars (ACFGs for short) form a proper basis for error detection and correction. Simply put, an ACFG is a Context-Free Grammar where each non-terminal symbol has a (finite) sequence of attributes each of which can have a set of a finite number of symbols as its value. In a rule the value of an attribute can be represented by a constant or by a variable.

Less informally, an ACFG G = (N, T, P, S, Var, Val) where N is a finite set of non-terminals and its number of attributes, T a finite set of terminals, P a set of production rules, S the start symbol, Var a set of variables and Val a finite set of value symbols. A production rule has the form

$$\alpha_0(v_{01} \ldots v_{0n_0}) \rightarrow \alpha_1(v_{11} \ldots v_{1n_1}) \ldots \text{ or } \alpha(v_1 \ldots v_n) \rightarrow t$$

where n_i is the number of attributes the non-terminal \(\alpha_i\) has and where v_{ij} is either a variable or a constant set of value symbols and t is a terminal. The language generated by G is defined in analogy to the normal definition of a CFL as \(L_G = \{ \alpha \mid S \Rightarrow^* \alpha \}\) but with agreement on the attribute values.
A simple fragment of an ACFG is for example:

1. $S \rightarrow NP \ (\text{Num nom}) \ VP \ (\text{Num})$
2. $NP \ (\text{Num _}) \rightarrow \text{Det} \ (\text{Num}) \ ADJs \ Noun \ (\text{Num})$
3. $NP \ (\text{Num Case}) \rightarrow \text{Pro} \ (\text{Num Case})$
4. $VP \ (\text{Num}) \rightarrow \text{Verb} \ (\text{Num intrans})$
5. $VP \ (\text{Num}) \rightarrow \text{Verb} \ (\text{Num trans}) \ NP \ (\_ \ acc)$
6. $ADJs \rightarrow$
7. $ADJs \rightarrow \text{ADJ} \ ADJs$

The derivation of a sentence might go like this:

$S \Rightarrow$
$NP \ (\text{sg3 nom}) \ VP \ (\text{sg3}) \Rightarrow$
$\text{Det} \ (\text{sg3}) \ ADJs \ Noun \ (\text{sg3}) \ VP \ (\text{sg3}) \Rightarrow$
$\text{Det} \ (\text{sg3}) \ Noun \ (\text{sg3}) \ VP \ (\text{sg3}) \Rightarrow$
$\text{Det} \ (\text{sg3}) \ Noun \ (\text{sg3}) \ Verb \ (\text{sg3 intrans}) \Rightarrow$

$a \ \text{man} \ \text{sings}$

In the actual implementation of the parser described in this paper, the grammatical formalism is slightly more complex as it uses strong-typed attributes and allows restrictions on the values the variables can take, which make grammar writing easier and parsing more reliable.

4. SHIFT-REDUCE PARSING WITH ACFGs

The construction of the parsing table is accomplished by means of standard LR-methods, e.g. SLR(0) or LALR(1), with the "stripped" grammar (i.e. leaving out the attributes). The parsing algorithm itself barely changes compared to a standard shift-reduce algorithm. The shift step is not changed except for the need to copy the attributes from lexical entries if we use a lexicon and a grammar with preterminals. The reduction step needs to be extended with a instantiation algorithm to compute the value of the variables and a succeed/fail result. It should fail whenever a instantiation fails or the value of a constant is not met.

To accomplish this, the trees which are stored on the stack should include the values resulting from the evaluation of the right-hand side of the reduced rule. This makes the instantiation step fairly straightforward. The variables can be bound while popping the elements from the stack. If a variable is already bound, it needs to be instantiated with the corresponding value on the stack. If this cannot be done or if a constant value in a rule does not match the value on the stack, the reduction step fails. A simple example (not completely) following the grammar sample above should clarify this.

In figure 1a parsing succeeds just as it would have done if only the context-free part of the grammar had been used. The only difference is that the symbols on the stack have attributes attached to them. In figure 1b however, parsing fails, not because the context-free part of the grammar does not accept the sentence (the parse table does contain an entry for this case), but because the unification of $p_1$ and $s_3$ in rule 1 causes the reduction to fail.

![Figure 1a. Parsing of "a man eats".](image)

![Figure 1b. Parsing of "a man eat".](image)
Note that the mechanism for variable binding is not completely equivalent to unification. It typically differs from unification in the reduction of the following two rules:

1. \( A \rightarrow \ldots B(X, Y) \ldots \)
2. \( B(X, X) \rightarrow \ldots \)

The reduction of rule 2 will leave two values on the stack rather than an indication that the two variables are one and the same. Therefore \( x \) and \( y \) may differ after the reduction of rule 1.

5. Parsing Erroneous Input

5.1. Agreement violations

Figure 1b shows one type of problem we are interested in, but clearly not the way to solve it. Though the parser actually detects the error, it does not give enough information about how to correct it. It doesn’t even stop at the right place\(^1\), since the incongruence is only detected once the entire sentence has been read. Therefore we need to alter the reduction step again. It should not fail whenever the instantiation of a variable fails or a constant in the left hand side of the rule being reduced does not match the corresponding value on the stack, but mark the incongruence and continue parsing as in figure 2. Then in this case the parsing result will contain an error message stating that the first attribute of the NP directly under \( S \) fails to agree with the value of the first attribute of its sister VP.

---

\(^1\) This of course is caused by the context-free part of the grammar. If we had created a unique non-terminal for every non-terminal feature combination, e.g. \( s \rightarrow NP_{sing}3 \_acc \_ VP_{sing}3 \), parsing would have stopped at the right place (i.e. between “man” and “eat”). This however depends mainly on the structure of the grammar. E.g. in Dutch the direct object may precede the finite verb, in which case agreement can only be checked after having parsed the object following the finite verb. Then the parser can never fail before the first NP following the finite verb, which is too late in general.

---

5.2. Spelling errors

Spelling errors are another type of error we are interested in. E.g. take the sentence the yellow cab stops. The spelling corrector of our word processor (MS-Word) offers two alternatives: yellow and yellows (1. n [U; (C)] a colour like that of butter, gold, or the middle part (YOLK) of an egg). Since the string yellow is obviously incorrect, it has no syntactic category and the sentence cannot be parsed. We might therefore try to substitute both alternatives and see what the parser comes up with, as in figure 3. This example clearly shows that the only grammatically correct alternative is yellow. In this way a parser can help a spelling corrector to determine the correct alternative. A (real) natural language parser should already have a way of parsing words with multiple syntactic categories (e.g. stop is both a noun and a verb), so the two entries for yellow could be parsed in the normal way. Afterwards the grammatical alternative(s) can be found by inspecting the resulting parse trees.

In order to handle errors caused by homophones as well, we need to extend this mechanism. When dealing with legal words we should use their syntactic categories plus the syntactic categories of possible homophones, or — to be on the safe side — every alternative a spelling corrector might give. Afterwards the parse trees need to be examined to see whether the original word or one of its alternatives is preferred.
5.3. Structural errors

The third and last category of errors we attempt to deal with, is formed by the structural errors. General techniques for parsing sentences in which this type of error occurs are difficult, computationally rather expensive and not completely fool-proof. For these reasons plus the fact that only a very limited number of structural errors occur in normal natural language, we have developed a different approach. Instead of having a special mechanism in the parser find out the proper alternative, the grammar should contain foreseen improper alternatives. Take as an example the language aⁿbⁿ. It is very simple to determine what is wrong with a sentence like “abbb”. There are either two superfluous b’s or the first b should be changed into an a. In case we are interested in the latter form of error, an error correcting grammar might look something like this:

\[
S \rightarrow \\
S \rightarrow A S B \\
A \rightarrow a \\
A \rightarrow b \text{ “Change } b \text{ into } a\text{”} \\
B \rightarrow b \\
B \rightarrow a \text{ “Change } a \text{ into } b\text{”}
\]

In which case the parsing of “abbb” goes like this:

0
3 a 0
1 A 0
8 b 1 A 0
1 A* 1 A 0
5 S 1 A* 1 A 0

As a result of this parse the first b in the sentence was reduced to A while warning that it should be substituted by an a. This approach can of course also be used in case of superfluous or missing punctuation or wrongly placed adjectives.

5.4. Parsing weights

Natural language is almost synonymous to ambiguity. Allowing errors to occur in the input only makes things worse. Even if we look at the simple toy grammar above and the sentence “they think” we might end up with a great number of useless parses. The word “think” may have different entries for first and second person singular, plural and infinitive, which would result in one parse tree without an error message and three parse trees all saying that the number of “they” does not agree with the number of “think”. By using sets of values instead of a single value this number can be reduced, but in general the number of parses will be very large. Especially with larger grammars and longer sentences there will be large quantities of parses with all sorts of error messages.

A simple way to differentiate between these parses is to simply count the number of errors, agreement viola-
tions, structural errors and spelling errors, in each parse and to order the parses accordingly. Then we only have to look at the parse(s) with the smallest number of errors. However, this concept of weight needs to be extended since not all errors are created equal. Some types of agreement violation simply never occur\footnote{As an example take the very general category of pronouns. This category is divided into subcategories by attributes. It is of course clear that no one will ever mistake a personal pronoun for a relative pronoun.} whereas others are often found in texts. Also, spelling errors and homophone substitution are frequent phenomena while structural errors are relatively rare. Suppose we have have a sentence like \textit{Word je broer geopereerd?} (English: \textit{Are your brother being operated?}). In Dutch this is a frequent error (see section 2.2), since the finite verb should indeed be \textit{word} if the subject were \textit{je} (which is either \textit{you} or \textit{your}) instead of \textit{je broer} (\textit{your brother}). The verb \textit{opereren (to operate)} has one (direct) object in an active sentence, and therefore there can only be a subject in the passive voice. So there are two possible errors: 1) \textit{word (are)} should become \textit{wordt (is)}, or 2) the transitivity of the main verb is wrong and \textit{broer (brother)} is the direct object (but has no article). The right choice is, obviously, 1. But how can a syntactic parser distinguish between these two alternatives? One solution lies in making distinctions in the grammar between these choices by giving the parse in which the transitivity is wrong a heavier penalty than the one in which the number of the finite verb is incorrect. In conclusion, the best way to distinguish parses is by the sum of their error weight as defined by the grammar.

6. WORD LATTICES

As said before, idiomatic expressions cause a parser a lot of trouble. We therefore propose that the parser should not operate directly on a linear sentence, but on a word lattice that has been prepared by a pre-processor. For a sentence like “hij kan te allen tijde komen logeren” (“he can come to stay at all times”) such a structure might look like figure 4. Instead of parsing each word of the expression \textit{te allen tijde} separately, the parser can take it as a single word spanning three words at once. Such lattices can of course become much more complex than this example.

Since we have a pre-processor that is able to combine multiple words into a single item, we might as well use it to aid the parser in detecting two further types of errors as well. The first one is the Dutch split compound. By simply joining all the adjacent nouns (under some restrictions) the grammar and the parser can proceed as if split compounds do not occur. The second error type is word doubling. The pre-processor can join every subsequent repetition of a word with the previous occurrence so that they will be seen as two distinct words and as one word (since not every occurrence of word repetition is wrong). Another possibility is to concatenate adjacent words when the concatenated form occurs as one entry in the dictionary. E.g. many people do not know whether to write "er op toe zien", "erop toe zien", "er optoe zien" or any other combination (though a parser might not always have the right answer either).

7. ADAPTATIONS TO THE TOMITA ALGORITHM

Now that we have seen the changes that need to be made to the stack and the shift and the reduce step and outlined the requirements for a pre-processor, we turn our attention towards the adaptations that need to be incorporated in the Tomita (1986) algorithm.

In general, the stacks and trees must be able to contain attribute values and error messages. The shift step needs to copy the attribute values onto the stack and should also mark the origin of distinct entries, such as the different alternatives of a spelling correction. The reduce step needs to com-
pute the new resulting values as well as the total sum of the error weights. Also notice that spelling errors have an error weight attached to them as well. This implies that freshly shifted entries may increase the total error weight. This demands that the criterion for merging stacks should be restricted: only those stacks that have the same attributes are allowed to merge.

This however does not suffice since the algorithm will then produce far too many parses at the same time, most of which we are not interested in. Since we are only interested in the parses with a minimal error weight, we might just as well create multiple stack groups, one for each total error weight. So, after every (shift or reduce) step, the resulting stack should be merged with stacks of the same weight. It is now possible to use only those stacks that have the lowest error weight, which speeds up parsing enormously. This also allows for one further optimisation: we are, generally speaking, not interested in parses with large numbers of errors. So we can impose a maximum error weight and simply discard those parses whose weight exceeds this threshold (e.g. by setting it to zero we only allow grammatical parses).

Unfortunately, this introduces a loss in efficiency. Tomita's algorithm, as well as any other "efficient" parsing algorithm, gains its efficiency from the fact that all stacks are at the same position in the input. But we just said that there was no need to continue those parses that have an error weight above the current minimum. This means that as soon as this minimum changes, because no error free parse can be found, we might be in trouble. The same problem also arises because of the word lattice. Therefore all stacks must not only know their error weight but also their position in the sentence. This leaves us with the notion of a current stack, which is the stack with the smallest error at the leftmost position in the sentence. Should there be two such (graph structured!) stacks, as soon as the current stack reaches the same position as an older stack, then they will have to be merged.

8. IMPLEMENTATION AND RESULTS
The parser as described above has actually been implemented in a Dutch grammar checker. We subjected this system to an informal Dutch spelling test used to test typists and secretaries. The test consists of 150 sentences varying in length between three and nine words. Every sentence has one word underlined and the subject's task is to decide whether or not that word is spelled correctly. This task has to be completed within ten minutes. If less than five per cent of the decisions is incorrect, the subject is considered to be a good speller; more than ten per cent means 'failed'.

The 75 errors in the test can be divided as follows: 59 non-words, all of which were recognized and of which 58 were corrected as intended; 14 legal words incompatible with the context, of which 11 were detected and corrected as intended; and two spelling errors in idiomatic expressions, which were both detected and corrected. In the case of two of the undetected legal words, which were incompatible with their context, the incompatibility was caused by the meaning of the sentence. One sentence read Vroeger behoorden zijn ouders zijn prestaties sterk (English: Formerly his parents influence his performance strongly).
The parser cannot discover that the tense of the verb is incorrect and should have been the homophonous beïnvloeden (influenced), because the meaning of vroeger (formerly) determines this in this context, and not its syntactic category. There were no error messages ("false alarms") on grammatical and well-spelled sentences and the program executed the test well within ten minutes.

9. CONCLUSIONS
The mechanism described above works quite well for certain types of errors. These are also the most frequent ones: spelling errors (resulting in non-existing words or in other existing words), agreement violations and punctuation errors. Errors that cannot be handled are those involving constituent deletion, superfluous insertion or transposition. Using a word-lattice however, it is possible to detect and correct the most frequent type of insertion (word doubling) as well as many split compounds. As a bonus it also allows us to parse idiomatic expressions and detect and correct errors in them as well. So, in conjunction with a word-level spelling corrector most grammatical errors can be effectively detected and corrected.

10. REFERENCES


TOMITA’s ALGORITHM IN PRACTICAL APPLICATIONS

Rob Heemels
Department of Language and Speech Technology
Océ Venlo B.V.
P.O. Box 101
5900 MA Venlo
Telephone: +31 77 593444
Telefax: +31 77 594313
E-Mail: rphe@oce.nl
May the 23. 1991

ABSTRACT

The parsing discipline has a longstanding tradition with its roots in compiler construction. In that area the main application was deterministic parsing. Numerous kinds of algorithms were designed for compiling programming language specifications. Soon it was clear that these kinds of specifications could best be expressed in a formalized language like a grammar. Those grammars were hardly ambiguous. Natural languages on the contrary are inherently ambiguous. Therefore the parsing methods could not simply be applied to this kind of constructions. In 1970 Earley developed an algorithm that was capable of producing all the possible analyses for an input sentence in a combined top-down bottom-up fashion. His algorithm is in fact a kind of chart parser. Before 1970 backtrack parsers were used. Their disadvantage is that they are extremely slow. Later on (1985) Tomita picked up the well known \(\text{LR} \) parse technique and extended it to ambiguous grammars. His Generalized \(\text{LR} \) Parsing Algorithm is at present one of the fastest and most efficient algorithms available and is applied to practical systems for natural language processing.

1. Introduction

In this paper I will discuss some of the requirements of a natural language processing system which should be able to handle grammatical as well as ungrammatical input.

It is common practice to use a parser for analyzing natural language and to be more specific a robust parser. Robust parsing is related to the parsing process (see also [Heemels 1991]). This process analyzes sentences according to a grammar. Sentences which are in accordance with the grammar will be recognized and analyzed as such. In a practical application however, the user types grammatical sentences as well as sentences that deviate more or less from the grammar rules stated in the grammar. A parser would normally reject the latter sentences.

A robust parser on the contrary may in such a situation produce the best analysis possible. It shall be clear that this cannot be guaranteed for every input sequence. In [Tomita 1986:98-99] we find the following definition of robust parsing:

"One approach is to tolerate such extragrammaticalities as much as possible so that the system seldom rejects input sentences. This approach is often called robust parsing."

The errors in ungrammatical sentences can be classified into two groups:

1.1 Errors at competence level:

- unknown structures

(those not defined in the grammar). The grammar for a natural language processing system will never be complete. Therefore there will always be constructions that do not fall within the scope of the grammar.

1.2 Errors at user level:

- unknown words

These words are not listed in the lexicon. Language is dynamic so we can’t simply expect all words to be in the lexicon. Unknown words can be handled by the parser by inspecting the action table for categories that have actions and execute all of them (for details see [Tomita 1986:43-46]).
- 80 -

- **Misspelled words**
  
  E.g. vvor instead of voor (english for). A spelling corrector can produce the most likely alternatives. One remaining problem we have to face is the fact that a typing error can result in an existing word.

- **Wrong segmentation of the input**
  
  E.g. kwammet instead of kwam met. (english came with). This problem is difficult to solve. Wrong segmentations can be detected by the Lexicalizer. If a word cannot be found it is either a misspelled word or a cluster of several words.

- **Wrong order of constituents**
  
  A solution to this problem could be to extend the grammar with rules that describe the wrong sequences and let them produce penalties.

- **Feature violations**
  
  E.g. nominative instead of dative.

- **Superfluous elements**
  
  E.g. Hij kwam met met de trein (eng. He came by by train) instead of Hij kwam met de trein (english He came by train). Another problem is separation of words analogous to other known languages: the members of compounds in Dutch are often separated by blank space because of interference with English.

---

**Disagreement**

E.g. Hij zingen een lied. (eng: He sing a song) instead of Hij zingt een lied (eng: He sings a song). These errors can be tolerated if we extend the grammar formalism (CFG's) with affixes.

As a result of research done at Océ over 200 pages of reports and 120 VAX-messages on News-net 1225 errors were collected. They can be divided into four major groups as is shown in fig 1.

---

fig. 3 Example of an overall natural language processing system

Most errors are grammatical ones (656), followed by style(469), typing (62) and spelling-mistakes (34). A robust parser should be able to recognize as much of these errors as possible. However style errors are difficult to detect. Perhaps the best method to track them is by checking a precompiled list of common errors.
If we want to detect the errors above mentioned we can develop a blueprint of a natural language processing system (see fig. 3). First the Text Normalizer has to convert the input into a standard format. After that the Text Tokenizer can divide the input in tokens. The Text Lexicalizer looks up the words in the lexicon. The lexicon contains information about wordclasses, features and semantic information. The lexicalized tokens will be analyzed and input to the parser. The parser loads precompiled tables and starts to analyze the input. The tables can be generated using a table generator. For a choice of the most suitable kind of LR-parse table see [Lankhorst 1990]. Words that belong to more than one wordclass are parsed in parallel (breadth first). Whenever in the action table a reduction is encountered the Unificator is called to unify the feature structures. A semantic component in the unification module can reject parses that violate semantic restrictions. Unification can result in violations because of missing features and feature disagreement.

A scoring mechanism sorts the analyses in order of probability. The outcome of the parsing process is the well known parse forest.

2 The requirements of a robust parsing algorithm

In view of the previous section we can state the following requirements for a robust parsing algorithm:

- Speed, real time parsing
- external grammar
- should handle grammatical as well as ungrammatical input
- grammar should be written in a standard formalism
- grammar should be extended with affixes
- efficient handling of ambiguity
- should have a unification module

3 Properties of Tomita’s algorithm

Tomita’s algorithm has the potential to meet most of the requirements stated in the previous sections.

- Speed

From both Tomita’s and Shann’s [Shann 1989] test results it is clear that Tomita’s algorithm is one of the fastest algorithms currently available.

- Efficiency

The number of edges that will be created during the parse process will be reduced to a minimum. This could be achieved by the two methods of packing (analysis stack and parse stack).

- Compactness

The algorithm is very compact and will only be a small module in a larger natural language processing system.

- Precompilation of parsing tables

The parsing tables have to be created only once before the actual parsing process starts. This saves time during parsing. In Earley’s algorithm and other kinds of chart parsers much time is needed to consult the grammar, although variants exist that precompile references to the grammar. In Tomita’s algorithm the grammar is not needed during input analysis.

- Pseudo-parallelism

The parse stack branches whenever conflicts occur during parsing (local ambiguities). The stack is represented as a structured graph. If branches of the stack are in the same state they will be merged together. That guarantees that the input is never parsed twice. Pseudo-parallelism can be replaced by real parallelism as is shown by Numazaki & Tanaka [Numazaki & Tanaka 1990]. However at present their implementation does not build a parse forest. The latter is nevertheless essential in natural language processing.

- Practical experience

The algorithm has been used at several locations for over three years. As a result most of the bugs in the algorithm have been fixed [Tomita 1990].

- Adaptability
The algorithm can easily be extended with for instance a unification module (see [Tomita 1986:98-101]).

- **Elegance**

The algorithm as described in [Tomita 1986] is neat and elegant, clear and understandable.

- **History and literature**

Because the algorithm is an extension to Knuth's well known LR parsing technique and has been applied in various compilers, it has a rich history and many articles about this technique have been written. The theoretical limits and properties of the algorithm are sufficiently known (see for instance [Kipps 1989] and [Johnson 1989]).

- **The input is in a standard format**

The algorithm takes as input a context-free grammar, possibly extended with unification ([Tomita 1990]).

- **Handling of ambiguity**

The inherent ambiguity of natural language is stored in a compact and efficient manner. As a result parsing times do not grow exponentially with increasing sentence length.

- **Suitable for on-line parsing**

The algorithm processes a sentence strictly from left to right. The input can be handled in a piece meal fashion. If necessary the algorithm waits for a next input word.

- **Undo-possibility**

It is possible to unparse part of the input by erasing the previously stored information for that part of the input. So the user can correct typos himself.

- **Error detection**

Errors are discovered as soon as they emerge because the algorithm does not find an action in the action table.

- **Dialogue based ambiguity resolving**

In [Tomita 1986:103-120] a method is described for resolving ambiguities based on a dialogue with the user. Consulting the user in difficult or unclear situations is a recent trend in for instance PMT (Personal Machine Translation see [Boitet 1990]). The packed forest retains information about ambiguity of an input sentence. The writer of the grammar can add Explanation Templates to the rules of the grammar (see section 5 for an example) that are known to contribute to ambiguous analyses (for instance rules for preposition adjacency and coordination). By means of Explanation List Comparison ambiguity can be resolved during a dialogue with the user. The user simply chooses the correct interpretation by pointing at a menu item on the screen or alternatively may prefer to maintain the ambiguity. This technique can be applied for instance to machine translation.

- **Generalization**

The algorithm can also be used in the field of speech recognition. An example is the SPHINKX-system (see [Kita et al 1989]). This system is capable of speaker independent speech recognition. It uses a vocabulary of 3000 words.

It seems to be a nice thought to use the same parse technique for natural language parsing as well as for speech recognition.

4 The problem of ambiguity

Suppose a system like the one described in fig. 3 has been built. Next applications can be defined. They range from educational grammar tools to things like grammar checkers, grammar correctors, dialogue systems, machine translation systems etc. Some of them have to cope with one of the most persistent problems in the world of natural language processing: ambiguity. Examples of the vast ambiguity in natural languages are: adjunction of prepositions and coordination.

With respect to a practical system it is important that ambiguity is handled efficiently.

5 Disambiguating the input

Often it is desirable to disambiguate the input
for instance in the field of machine translation. In the example sentence *the rabbit saw an orange with a telescope* the ambiguity can be solved by asking the user. A menu appears and the user can either choose or keep the ambiguity. If he chooses the first option as is shown in fig. 7 the sentence interpretation is that *the rabbit sees the orange while looking through a telescope* (see fig. 8). Figures 4 and 5 show the grammar and the lexicon used in this example. The feature information is left out because of lack of space. Figure 6 represents a typical forest the Tomita parser would produce.

\[
\begin{align*}
NP & : *\text{article, noun} .
\hline
NP & : *\text{noun} .
\hline
NP & : NP, PP . ~ S[1][2] (1)is (2)$
\hline
PP & : *\text{prep, NP} .
\hline
S & : NP, VP .
\hline
S & : S, PP . ~ S[1][2] (1)takes place (2)$
\hline
TOP & : S, *\text{endmark} .
\hline
VP & : *\text{verb, NP} .
\end{align*}
\]

**Fig. 4. A sample grammar**

The toy grammar consists of 8 rules. The rules 3 and 6 do have a template attached. Each template has a head, a focus and a body (see also [Tomita 1986: 103-120]). Heads represent the kernel of a righthand side. The head of *the man in the street* is for instance *the man*. The focus is the element in a rule which will help to solve the ambiguities in a forest.

\[
\begin{align*}
. & (\text{endmark \text{...}})
\hline
\text{a} & (\text{article \text{...}})
\hline
\text{an} & (\text{article \text{...}})
\hline
\text{orange} & (\text{noun \text{...}}) (\text{adj \text{...}})
\hline
\text{rabbit} & (\text{verb \text{...}})
\hline
\text{noun} & (\text{...})
\hline
\text{saw} & (\text{verb \text{...}}) (\text{verb \text{...}}) (\text{noun \text{...}})
\hline
\text{telescope} & (\text{verb \text{...}}) (\text{noun \text{...}})
\hline
\text{the} & (\text{article \text{...}})
\hline
\text{with} & (\text{prep \text{...}})
\end{align*}
\]

**Fig. 5. A sample lexicon**

The lexicon entries consist of a word and a list of the categories they can belong to. In a fully fledged system there will be selection restrictions and semantic information added to the categories.

The result of the parsing process is contained in the parse forest. We see the top of the forest at index 20. From the list to the right of the equal sign we can conclude that the sentence was ambiguous. Normally the forest represents besides the syntactical trees the feature information and semantic properties of the nodes. To the left of the forest the violations of the nodes are shown (in this case all are zero).

\[
\begin{align*}
0 & \text{fa}[1] = (*\text{article} [] (((0)\text{the}))
0 & \text{fa}[2] = (*\text{noun} [] (((0)\text{rabbit})))
0 & \text{fa}[3] = (\text{NP} [] (((1) 2)))
0 & \text{fa}[4] = (*\text{verb} [] (((0)\text{saw})))
0 & \text{fa}[5] = (*\text{article} [] (((0)\text{an})))
0 & \text{fa}[6] = (*\text{noun} [] (((0)\text{orange})))
0 & \text{fa}[7] = (\text{NP} [] (((5) 6)))
0 & \text{fa}[8] = (\text{VP} [] (((4) 7)))
0 & \text{fa}[9] = (\text{S} [] (((3) 8)))
0 & \text{fa}[10] = (*\text{prep} [] (((0)\text{with})))
0 & \text{fa}[11] = (*\text{article} [] (((0)\text{a})))
0 & \text{fa}[12] = (*\text{noun} [] (((0)\text{telescope})))
0 & \text{fa}[13] = (\text{NP} [] (((1) 12)))
0 & \text{fa}[14] = (\text{PP} [] (((10) 13)))
0 & \text{fa}[15] = (\text{S} [] (((9) 14)))
0 & \text{fa}[16] = (\text{NP} [] (((7) 14)))
0 & \text{fa}[17] = (\text{VP} [] (((4) 16)))
0 & \text{fa}[18] = (\text{S} [] (((3) 17)))
0 & \text{fa}[19] = (*\text{endmark} [] (((0),)))
0 & \text{fa}[20] = (\text{TOP} [] (((15) 19),(18 19)))
\end{align*}
\]

**Fig. 6. The parse forest before disambiguation**

**Fig. 7. The dialogue with the user**

**Fig. 8. The forest after choosing the first option**
fig. 9. The dialogue with the user

If the user chooses the second option (fig. 9) then the interpretation of the sentence is that the orange has a telescope and the rabbit observes this (see fig. 10).

fig. 10 The forest after choosing the second option

6 Ungrammatical input

It is necessary to extend Tomita’s algorithm with some kind of unification mechanism to cope with ungrammatical input. Extragrammaticalities such as agreement errors are the result of feature violations. Those errors can be dealt with if one allows this kind of violations during the parsing process. However they must be less probable. This can be achieved by developing a scoring mechanism. If errors are very common the most efficient way to handle them is to add rules to the grammar which allow for these errors and mark them as being incorrect.

Conclusion

Tomita’s algorithm is an attractive algorithm in the field of natural language processing. In particular the parse forest is a promising candidate for the representation of the analyses. Together with the precompiled parse tables and the breadth first approach which makes on-line parsing possible it is a solid basis for a natural language processing module.

Bibliography


[Boitet 1990] Boitet, Christian

[Chien et al. 1990] Chien, Lee-Feng.

[Chiesinger 1979] Chiesinger, J.

[Devos & Gilloux 1990] Devos, L. & Gilloux, M.

[Earley 1968] Earley, J.

[Earley 1970] Earley, J.
An Efficient Context-Free Parsing Algorithm, CACM 13(2), 94-102.


On the relative efficiencies of context-free grammar recognizers. CACM 8, 5 (May 1965), 289-300.

[Jacobs 1990] Jacobs, P.
To Parse or Not to Parse: Relation-Driven Text Skimming.,

[Jain & Waibel 1989] Jain, Ajay & Waibel, Alex
A Connectionist Parser Aimed at Spoken Language.
Proceedings of the International Workshop on Parsing

YACC- Yet Another Compiler Compiler, CSTR 32, Bell

[Kasami 1965] Kasami, T.
An efficient recognition and syntax analysis algorithm for
context-free languages. Scientific Report AFCRL-65-758,
Air Force Cambridge Research Laboratory, Bedford, Mass.,
1965.

[Klijnssma 1986] Klijnssma, S.
Parsing van vraagzinnen in "natuurlijke" taal: twee
methoden. Afstudeer-verslag Tech. Universiteit Eindhoven,
1986.

On the translation of languages from left to right.

When Something is Missing: Ellipsis, Coordination and the
Chart. Proceedings of the Coling-90., Helsinki 1990,
184-189.

[van der Linden 1987] H.J.B.M.
Eindrapport Paser-Project Vooronderzoek, Nederlandse
organisatie voor toegepast natuurwetenschapelijk onderzoek

Theory of Syntactic Recognition for Natural Language.

[Numazaki & Tanaka 1990] Numazaki, Hiroaki & Tanaka,
Hozumi
A New Parallel Algorithm for Generalized LR Parsing.

[De Roeck 1983] De Roeck, A.
An Underview of Parsing, King, M. (ed) Parsing Natural

[Saito 1990] Saito, H.
Bi-directional LR Parsing from an Anchor Word for Speech
Recognition., Proceedings of the Coling-90., Helsinki 1990,

[Thurair 1990] Thurair, G.
Parsing for Grammar and Style Checking., Proceedings of the

[Tomita 1985] Tomita, M.
An Efficient Context-Free Parsing Algorithm for Natural
Languages and Its Applications, Ph.D. Thesis, Computer
Science Dept., Carnegie-Mellon University, Pittsburgh, PA,
1985.

[Tomita 1986] Tomita, M.
Efficient Parsing for Natural Language, Kluwer Academic

[Tomita 1989] Tomita, M.
Parsing Algorithms 2-Dimensional Language. Proceedings of
the International Workshop on Parsing Technologies.,

[Tomita 1990] Tomita, M.
The Generalized LR Parser/Compiler V8-4: A Software
Package for Practical NL Projects., Proceedings of the

[Verhaar 1986] Verhaar, Vincent
Een fouten verbeterende parser voor "natuurlijke" taal.

[Tomita 1987] Tomita, M.
An Efficient Augmented-Context-Free-Parsing Algorithm,

[Tomita 1988] Tomita, M.
Graph-Structured Stack and Natural Language Parsing. The
Proceedings of the 26th Annual Meeting of the Association for
Computational Linguistics, SUNY Buffalo, New York.

[Johnson 1989] Johnson, Mark
The Computational Complexity of Tomita's Algorithm.
Proceedings of the International Workshop on Parsing
[Kipps 1989] Kipps, James, R.

[Kita et al] Kita, Kenji & Kawahata, Takeshi & Saito, Hiroaki

On the translation of languages from left to right. Information and Control 8:6, 607-639, 1965.

[Lankhorst 1990] Lankhorst, Marc.


[Nijholt et al. 1989a] Nijholt, Anton

[Nijholt 1989b] Nijholt, Anton

[Nijholt 1990] Nijholt, Anton

[Shann 1989] Shann, Patrick

[Sikkel 1990] Sikkel, N.

[Sipper 1981] Sipper, S.

[van der Steen 1987] van der Steen, Gert Jan
A Program Generator for Recognition, Parsing and Transduction with Syntactic Patterns, Dissertatie Rijksuniversiteit Utrecht, 1987.

[Younger 1967] Younger, D. H.

Language as a cognitive process, Syntax, Addison-Wesley 1983.

[Wirén 1990] Wirén, Mats

[Yonezawa & Ohsawa 1988]
AN EMPIRICAL COMPARISON OF GENERALIZED LR TABLES

Marc Lankhorst
Computer Science Department, University of Twente,
P.O. Box 217, 7500 AE Enschede, The Netherlands

ABSTRACT

In this study, table sizes and parsing efficiency of Tomita’s algorithm with LR(0), SLR(1), LALR(1) and LR(1) parsing tables are compared on the basis of empirical data. From this comparison, it can be concluded that LALR(1) tables are the best choice regarding parsing time. These tables are about the same size as SLR(1) and noticeably smaller than LR(1) tables. If the size of the tables or the ease of construction is a determining factor, it can be advisable to use LR(0) tables.

As is known from the theory of ‘standard’ LR parsing, the use of LR(1) tables is expected to result in the best parsing efficiency compared to the other three types. However, this is not the case in Generalized LR parsing. LR(1) turns out to be significantly slower than the other types presented. We give an elegant explanation for this unexpected phenomenon.

1 INTRODUCTION

A well-known method for parsing a certain class of context-free languages is the LR algorithm, introduced by Knuth [1965]. One of the limitations this algorithm imposes, is that the grammars which are used must be unambiguous. In Tomita [1985] an extension to this algorithm is described which can parse almost all non-cyclic context-free languages. This variant, which Tomita calls generalized LR parsing, makes it possible to parse ambiguous natural language on the basis of the efficient LR algorithm.

Parsers based on the LR algorithm use LR parsing tables, consisting of an action and a goto part, which can be constructed directly from the grammar. There are many different types of these LR tables, which vary in space and parsing efficiency. In the following, we will be speaking of LR tables although the generalized LR tables used by Tomita’s algorithm differ from standard LR tables in one aspect: Conflicts (multiple entries) in the action part of the table are no longer forbidden.

Since the LR tables are the only part of (generalized) LR-like parsers that are grammar-dependent, the decision which type to use is quite important in the construction of such a parser.

To be able to make a well-founded choice between the different types of generalized LR tables, a comparison on both theoretical and practical aspects is necessary. In this paper, table sizes and parsing efficiency of four of the most widely used types are compared on the basis of empirical data. From this comparison we can draw conclusions considering which table type is the most suitable for use by Tomita’s algorithm.

2 LR TABLE TYPES

In this paragraph we discuss the main differences between the four different types of LR tables that we have looked at. These four types are: LR(0), SLR(1), LR(1) and LALR(1). We will not go into the construction methods of these types. These are extensively treated in e.g. Aho, Sethi and Ullman [1986].

LR(0) tables

LR(0), which stands for LR with 0 look-ahead, is the simplest type of LR table. As is stated in its name, this type employs no look-ahead symbol, which means that all reductions are carried out regardless of the next input symbol. In many situations such a reduction will not lead to a correct parse of the input, hence it was invalid and unnecessary work has been done. To prevent this, other methods, which use one or more look-ahead symbols, are invented. Three types which use one look-ahead are discussed below.

The main advantage of LR(0) tables is their relatively small size compared to the other types. This results from the fact that all reductions can be stored in one column instead of across the entire action table.

SLR(1) tables

The first of these types is SLR(1), which means Simple LR with 1 look-ahead. A reduction to a symbol X is carried out only if the next input symbol is an element of FOLLOW(X), the set of terminal symbols that can follow X in any derivation of the start symbol. If the next input symbol is not in this set, it is not possible to derive the start symbol after carrying out the reduction to X, so this reduction is invalid.
SLR(1) tables have the same number of states as LR(0) tables, but they are considerably larger because the reductions have to be stored separately for each valid look-ahead symbol.

LR(1) tables

The third type is LR(1), the canonical LR method with 1 look-ahead. In SLR(1) it is still possible that a superfluous reduction to X is carried out when the input symbol can, in general, follow X in a derivation of the start symbol, but not in this particular parse. LR(1) circumvents this problem by carrying more information in a state and splitting states when necessary. In this way some of these invalid reductions can be ruled out. It can be proved that LR(1) makes the best possible use of the look-ahead information, therefore it is called canonical LR(1).

A problem with LR(1) tables is their enormous size. The number of states can be tens of times larger than that of LR(0) and SLR(1) tables for the same grammars.

LR(1) tables have the best parsing efficiency of all single look-ahead types when standard LR parsing is regarded. We shall see that this is not the case in generalized LR parsing. Billot and Lang have found a similar result. They conclude that "sophistication in chart parsing schema (e.g. use of look-ahead) may reduce time and space efficiency instead of improving it" [1989, p. 143]. However, this is not widely known. Noohooor-Farshi e.g. wrongly states that "using non-optimized LR(1) tables will decrease the number of superfluous reductions in general." [1989, p. 187].

LALR(1) tables

Because the large size of the LR(1) tables poses a problem, LALR(1), which stands for Look-Ahead LR(1), was conceived. It forms a compromise between the parsing efficiency of LR(1) and the size of SLR(1) tables. The construction of LALR(1) tables can be regarded as the merging of LR(1) states into a table with the number of states of LR(0) and SLR(1). The parsing efficiency of LALR(1) is better than that of LR(1) and the size of the tables is comparable to SLR(1).

3 THE EXPERIMENT

The objective of the experiment was to gain insight in the aptness for Tomita’s algorithm of the four different types of LR tables described above.

Grammars

To conduct the experiment, we used three grammars, which are all given by Tomita [1985]. Grammar 3, formed by 225 productions, can be used on both the first and the second set of sentences (see below). Grammars 1 and 2, consisting of 8 and 44 productions respectively, can only be applied on the second sentence set.

Sentences

The input sentences are also derived from Tomita [1985]. The first sentence set consists of 40 ‘real life’ sentences, most of which are taken from actual publications.

The second set is made more systematically. The n-th sentence (1 <= n <= 8) is obtained by the following schema:
	noun verb det noun (prep det noun)^n-1

Tomita [1986, p. 81-82, 153] assumes that these sentences have the same ambiguity when parsed with all three grammars. This is not the case! Because of the absence of a recursive rule of the form ‘NP -> NP PP’ in the second grammar, this grammar allows less deep ambiguities than the first and third.

Method

To obtain the test results we have used an implementation of Tomita’s algorithm which was executed on a Sun Microsystems SPARCstation! Every input sentence was parsed five times in a row and this process was repeated four times. The average parsing time was computed from these 20 results to minimize potential inaccuracies.

Apart from the parsing time, the sizes of the graph-structured stack T and the packed forest T of Tomita’s algorithm were also determined. These sizes are a measure for the density of the packing that was applied. Without packing, T is exactly twice as large as T. This can be deduced from the formal description of Tomita’s algorithm.

In LR(0) tables all reductions to be carried out can be found in one column of the action table. To determine whether this is an advantage, we have also regarded LR(0) tables in which the reductions were filled in across the entire action table.

The sizes of the different types of tables for the three grammars can be found in figure 1. From this figure, which has a logarithmic scale, one can observe that the ‘normal’ LR(0) tables are the smallest and the LR(1)
Figure 1. Table size against grammar size for grammars 1, 2 and 3

Figure 2. Parsing time against ambiguity for grammar 1 and sentence set 2
tables are by far the largest. The LR(1) tables for grammar 3 are several megabytes in size and could not be computed due to a lack of memory. Therefore this size is estimated.

4 RESULTS

In figure 2 the parsing times of the sentence set 2 combined with grammar 1 are given for the different table types. This figure shows a small but suspicious detail: The LR(1) parsing times are somewhat slower than the SLR(1) and LALR(1) times, whereas one would expect LR(1) to be faster (see 2.3). But the differences are so small that they could be attributed to measuring inaccuracies.

However, figure 3 shows the same difference on a much larger scale. This cannot be some inaccuracy but requires a better explanation.

Explanation

The unexpected phenomenon described above can be explained as follows.

LR(1) tables contain in general (many) more different states than LR(0), SLR(1) and LALR(1) tables. This means that in a non-deterministic LR(1) parse of an ambiguous sentence, different states occur which would have coincided if LR(0), SLR(1) or LALR(1) tables were used.
Tomita's algorithm merges top vertices with the same state in the graph-structured stack. In the case of LR(1) less merging is possible due to the larger number of different states. This results in more top vertices than in the other three cases. Because reductions and shift actions are carried out on these top vertices, more work has to be done and longer parsing times are obtained.

An example of this can be seen in figures 4 and 5. Figure 4 shows the graph-structured stack during the LALR(1) parsing of a sentence from set 2 with grammar 2. Figure 5 shows the stack during the LR(1) parsing of the same sentence. Both figures show the situation immediately after shifting a *prep symbol onto the stack. In figure 4, the top vertices have just been merged. In the LR(1) case of figure 5, this is not possible because the top states are different. The next actions have to be carried out thrice in the case of LR(1), but only once in the LALR(1) case. This leads to a major difference in parsing time.

Packing is less efficient in the case of LR(1) than in the other cases. The stack sizes of the different types relative to LR(0) are shown in figure 6. From this figure, it can be seen that the LR(1) packing grows less and less dense compared to LR(0) when the ambiguity of the input is raised.

One would expect LR(1) to be the fastest method when no packing is used. As can be seen from figure 7, this is indeed the case. This fact forms a strong support for our explanation.

More results

Some more results are shown in figures 8 and 9. From both these figures, it is apparent that SLR(1) is significantly faster than LR(0) and LALR(1) is the fastest method. Also one can see that the advantage of 'normal' LR(0) over the 'filled in' LR(0) is not very large.

5 CONCLUSIONS

The first conclusion that can be drawn from the above is that LR(1) tables are unsuitable for the generalized LR parsing algorithm of Tomita. These tables are much larger than the other three types presented and result in significantly longer parsing times. This conclusion is not in accordance with most literature on standard LR parsing, which states that LR(1) is the most efficient single look-ahead LR type. This is definitely not the case in generalized LR parsing.

Secondly, we can conclude that from the four types presented, LALR(1) is the best choice if maximum parsing efficiency is required. The use of these tables can result in a 5-10% improvement in parsing time compared to LR(0). Also a gain in space efficiency of the same order of magnitude can be achieved.

If the size of the parsing tables is a constraint, LR(0) tables are advisable. These are several times smaller than the other types. Another advantage of LR(0) tables is their ease of construction.

Acknowledgement

This research was done during my stay with the Language and Speech group of Océ Nederland BV, Venlo, The Netherlands. I would like to thank Rob Heemels, Eddy van Vliembergen, Lou Cremers, Erik Hommersom and all my other colleagues at Océ for their help and support.

REFERENCES


Figure 6. Size of the graph-structured stack relative to LR(0)

Figure 7. Parsing time against ambiguity for grammar 2 and sentence set 2 without packing
Figure 8. Parsing time against ambiguity for grammar 3 and sentence set 1

Figure 9. Parsing time against ambiguity for grammar 3 and sentence set 2
BOTTOM-UP PARALLELIZATION OF TOMITA’S ALGORITHM

Klaas Sikkel
Computer Science Department, University of Twente,
PO Box 217, 7500 AE Enschede, The Netherlands
sikkel@cs.utwente.nl

ABSTRACT

A new parallel parsing algorithm is described, based
on Tomita’s generalized LR parser and related to a
well-known parallel version of Earley’s algorithm. It
uses a pipeline of processors, one for every word in
the sentence. Each processor parses all constituents
starting with its “own” word. While constructing such
a partial parse, constituents that have been parsed by
other processors higher up the pipeline may be used as
terminal symbols.

1. INTRODUCTION

Tomita’s generalized LR parser [Tomita85] is a
popular parsing algorithm for natural language applica-
tions. It combines the ability to handle arbitrary
acyclic context-free grammars with the efficiency of
the LR parser. Rather than a single parse stack, as in
traditional LR parsing, a graph structured stack is
maintained. All branches of the stack are seemingly
dealt with in parallel. This is pseudo-parallelism, in
fact, as the algorithm is clearly designed to run on
one processor only.

Tanaka and Numazaki have implemented several
generalized LR parsers in a parallel logic program-
ing language. Maintaining a graph structured stack
would require too much synchronization, therefore
they work in parallel on separate copies of linear
stacks [Tanaka89] or with tree structured stacks
[Numazaki90]. We look at the problem of parallel
generalized LR parsing from quite a different angle
— taking, in fact, a perpendicular view. Rather than
working through the sentence in LR fashion, we
remove the left-to-right restriction and introduce
processes that parse the sentence in bottom-up
fashion, starting at every word in the sentence in
parallel. Each process runs an adapted Tomita parser,
yielding the constituents that start with its “own”
word.

Tomita’s algorithm is closely related to Earley’s
algorithm [Earley70]. Our parallel Tomita parser is
similarly related to — and motivated by — a parallel
bottom-up Earley parser [Chiang84], [Nijholt89],
[Sikkel90]. For related approaches towards parallel
parsing, see [Fischer75] (parallel LR, merging con-
secutive parts of parse stacks); [Sijstems86] (an
efficient implementation of Earley’s algorithm on a
matrix of processors); [Yonezawa89] (an object-
oriented parallel Earley parser); [odAkker91] (an
annotated bibliography on recent literature);
[Nijholt91] (a general overview of parallel parsing).

In section 2 we describe a generalized LR(0)
parser, assuming some basic knowledge of LR pars-
ing techniques. In 3 and 4 the bottom-up paralleliza-
tion is discussed. The construction of a shared parse
forest is dealt with in section 5.

2. LR(0) TOMITA

Tomita’s parsing algorithm is a generalization of the
well-known LR parsing technique. An LR parser
scans a string from left to right while constructing a
rightmost derivation of the parse tree. Traditional LR
parsing requires at every step that the next action be
uniquely determined. In generalized LR parsing,
parse table entries may contain multiple actions. If
such a situation arises, both actions are carried out on
separate copies of the parse stack. In highly ambigu-
ous sentences, the number of different parse stacks
can become rather large. For efficiency, the different
parse stacks are merged into one graph-structured
stack. When different parses share common sub-
parts, the corresponding actions have to be carried
out only once.

Throughout this article, we will use the following
simple grammar G:

(1) \( S \rightarrow NP VP \)
(2) \( S \rightarrow PP \)
(3) \( *det \rightarrow \epsilon \)
(4) \( NP \rightarrow *det *noun \)
(5) \( NP \rightarrow NP PP \)
(6) \( PP \rightarrow *prep NP \)
(7) \( VP \rightarrow *verb NP \)

Note that production (3) usually reads \( NP \rightarrow *noun \).
That is, a noun phrase can have the form \( *det *noun \)
or \( *noun \). In our grammar, a noun phrase always has
the form \( *det *noun \) (followed by zero or more \( PP \)'s),
but the \( *det \) may be omitted. Both grammars obvi-
ously produce the same sentences, but our grammar is
not \( \epsilon \)-free and therefore slightly more awkward to
parse. That makes it suitable for the examples to fol-
low. Note that \( *det \) is used both as a (pre)terminal
and a proper nonterminal. As there won’t be any confusion, we use this sloppy but convenient notation.

An LR(0) parse table for grammar $G$ is shown in Figure 1. Note that (in contrast to [Tomita85], where an SLR(1) table [Aho77] is used) there is only one column containing actions. Without look-ahead the next action — or set of possible actions — is determined solely by the state of the parser. If more than one action is given in the action column, the stack forks. An example of the maintenance of a graph structured stack will be given shortly. The GOTO table has entries for terminals and nonterminals: the next state is determined by the current state and the symbol that is shifted/reduced. The parse table in Figure 1 is called annotated because it contains the sets of LR(0) items that have been used for its construction. For each symbol that follows upon a “.” in an LR(0) item of a particular state, a successor state must be present. This successor state contains the closure [Aho77] of the applicable LR(0) items. Note that $G$ is extended with a production $S’ \rightarrow S \$, with $\$ $ the end-of-sentence marker. The end-of-sentence marker must be shifted explicitly in order to determine that the input has really finished.

We will exemplify the maintenance of the graph structured stack by looking at a few instances during the parsing of the sentence John saw a lion at the zoo.

<table>
<thead>
<tr>
<th>state</th>
<th>LR(0) items</th>
<th>action</th>
<th>*det</th>
<th>*noun</th>
<th>*verb</th>
<th>*prep</th>
<th>NP</th>
<th>PP</th>
<th>VP</th>
<th>S</th>
<th>$</th>
</tr>
</thead>
</table>
| 0     | $S' \rightarrow S S$  
S \rightarrow NP VP  
S \rightarrow S PP  
NP \rightarrow *det *noun  
NP \rightarrow NP PP  
*det -> . | re3 | 3 | -   | -   | 2   | -   | 1  | -  |
| 1     | $S' \rightarrow S S$  
S \rightarrow S PP  
PP \rightarrow *prep NP | sh | -   | -   | -   | 6   | -   | 5   | -   | 4 |
| 2     | $S \rightarrow NP \cdot VP  
NP \rightarrow NP \cdot PP  
VP \rightarrow *verb NP  
PP \rightarrow *prep NP | sh | -   | -   | 7   | 6   | -   | 9   | 8   | - |
| 3     | NP \rightarrow *det *noun | sh | -   | 10  | -   | -   | -   | -   | -   | - |
| 4     | $S' \rightarrow S S$  
S \rightarrow S PP | acc | -   | -   | -   | -   | -   | -   | -   | - |
| 5     | S \rightarrow S PP | re2 | -   | -   | -   | -   | -   | -   | -   | - |
| 6     | PP \rightarrow *prep NP  
NP \rightarrow *det *noun  
NP \rightarrow NP PP  
*det -> . | re3 | 3 | - | - | 11 | - | - | - | - |
| 7     | VP \rightarrow *verb NP  
NP \rightarrow *det *noun  
NP \rightarrow NP PP  
*det -> . | re3 | 3 | - | - | 12 | - | - | - | - |
| 8     | S \rightarrow NP VP | re1 | - | - | - | - | - | - | - | - |
| 9     | NP \rightarrow NP PP | re5 | - | - | - | - | - | - | - | - |
| 10    | NP \rightarrow *det *noun | re4 | - | - | - | - | - | - | - | - |
| 11    | PP \rightarrow *prep NP  
NP \rightarrow NP \cdot PP  
PP \rightarrow *prep NP | re6 | - | - | 6 | - | 9 | - | - | - |
| 12    | VP \rightarrow *verb NP  
NP \rightarrow NP \cdot PP  
PP \rightarrow *prep NP | re7 | - | - | 6 | - | 9 | - | - | - |

**Figure 1:** An annotated LR(0) parse table
After having processed *John saw a lion*, we have a stack as shown in Figure 2(a). The next action could be reduce (7) or shift. As a matter of policy, all possible reduce actions are carried out before the next shift is done. Hence the reduction VP→*verb NP is carried out, yielding 2(b). After reducing S→NP PP both branches do a shift. The next symbol *prep* yields state 6 for either branch, in which case they can be joined again. This is shown in Figure 2(c).

When a reduction is carried out, it is not necessary to physically remove the part of the stack that is being reduced (and indeed [Tomita85] doesn’t do it). Alternatively, it can remain as an inactive branch on the stack. In this way the LR parser delivers a graph that contains a node for every recognized constituent. Let $a_1 \cdots a_n$ be the sentence to be parsed. Provided that the grammar is acyclic (which is a usual condition for LR parsers) it is easy to show that a (generalized) LR(0) parser recognizes symbol $A$ for every $i,j$ satisfying both conditions

$$A \Rightarrow^* a_{i+1} \cdots a_j \quad (1)$$

$$S \Rightarrow^* a_1 \cdots a_i A \gamma \text{ for some } \gamma \in V^* \quad (2)$$

3. PARALLEL BOTTOM-UP TOMITA:
A FIRST APPROACH

The condition (2) above is sometimes called a top-down filter. As in some versions of Earley’s algorithm [Graham80, Nijholt89], we can remove the top-down filter and construct a parser that recognizes $A$ for every $i,j$ such that $A \Rightarrow^* a_{i+1} \cdots a_j$, removing the left-to-right restriction imposed by top-down filtering. We will construct a pipeline of processors $P_0 \cdots P_n$, each one starting at a different position in the sentence. Processor $P_i$ creates a partial parse stack, covering all partial parses $A \Rightarrow^* a_{i+1} \cdots a_j$ for $i \leq j \leq n$. Eventually, occurrences of $S$ such that $S \Rightarrow^* a_1 \cdots a_n$ will be delivered by $P_0$. While building such a parse stack, $P_i$ uses other partial parses that have been delivered by $P_{i-1} \cdots P_0$. This is schematically shown in Figure 3. Furthermore, each $P_i$ receives word $a_{i+1}$ from input and delivers information of partial parses to an output. How this is organized in detail is beyond the scope of this article.

Consider, for example, the prepositional phrase in the zoo. We will denote the corresponding grammar symbols as $\langle 4, * \text{prep}, 5 \rangle$, $\langle 5, * \text{det}, 6 \rangle$, $\langle 6, * \text{noun}, 7 \rangle$. Place markers are necessary to identify the part of the sentence that corresponds with a recognized grammar symbol. Processor $P_4$ reads the marked terminal $\langle 4, * \text{prep}, 5 \rangle$ from input. Then it simply waits until $P_5$ passes a noun phrase $\langle 5, NP, 7 \rangle$. $P_4$ extends its stack with the noun phrase by a shift operation. Subsequently, $\langle 4, * \text{prep}, 5, NP, 7 \rangle$ is reduced to $\langle 4, PP, 7 \rangle$, which is also passed downstream to $P_5$.

Each processor has a copy of an adapted parse table. The construction of this table differs in two respects from the construction of Tomita’s table:

- State 0 contains LR(0) items $A \rightarrow \alpha$ for every $A \rightarrow \alpha \in P \cup \{S \rightarrow S\}$.
- When a new state $s'$ is constructed that is reachable via an entry $(s, X)$ in the GOTO table, do not take the closure of the appropriate LR(0) items,
only the appropriate items with "$\cdot X$" replaced by "X*".

The parallel annotated LR(0) table for our example grammar is shown in Figure 4. The entries $(0, VP)$ and $(0, PP)$ show a next state "C". This means that the partial parse is completed, and nothing can be added to the particular branch of the partial parse stack. State C has no actions and is not shown in the table. Furthermore, there are the following differences in the construction of the partial parse stacks:

- On a reduce it is not allowed to prune the branch of the stack that is reduced. It may be still be needed at some moment in future. Synchronize on shift is no longer possible, as each processor starts at a different part of the sentence.
- The shift operation is not restricted to terminals, but may also be applied to appropriate nonterminal symbols that are passed down the pipeline.
- Two place markers are tagged onto each symbol. Without these place markers it would not be possible to decide at which position in the stack a new symbol is to be added.

A completed partial parse stack for processor $P_1$ is shown in Figure 5.

It is important that symbols are passed along the pipeline in an appropriate order. When a processor is to decide whether a next symbol $(i, X, j)$ can be added to the stack, it should have received all applicable symbols $(k, X, l)$ with $l \leq i$, and should have incorporated the useful ones into its parse stack. In particular, symbols $(j, X, j)$ should not be processed until all symbols $(k, X, l)$ with $k \leq l \leq j$ have been dealt with; likewise, symbols $(j, X, j)$ should precede all symbols $(k, X, l)$ with $j \leq k < l$.

The ordering requirements are satisfied if processor $P_i$ receives its input from $P_{i+1}$, for appropriate symbols $X$, in the following order:

$(i+1, X, i+1), (i+1, X, i+2), (i+2, X, i+2), \cdots (n-1, X, n), \cdots (i+1, X, n), (n, X, n), (n, \$, n+1).$

$P_i$ can insert its own products, yielding an output stream

$(i, X, i), (i, X, i+1), (i+1, X, i+1), \cdots (n-1, X, n), \cdots, (i, X, n), (n, X, n), (n, \$, n+1)$

for appropriate symbols $X$. Consequently, the ordering requirements are also satisfied for processor $P_{i-1}$. The end-of-sentence marker $(n, \$, n+1)$ serves also indicates the end of the stream of symbols that is passed along the pipeline.

If there are different symbols $(i, X, j)$ and $(i, Y, j)$ for some specific values of $i$ and $j$ with $i < j$, it does not matter in which order they are processed. For different symbols $(j, X, j)$ and $(j, Y, j)$, however, the ordering requirements state that they should precede each other. That is, they have to be dealt with simultaneously. We will not worry about that here, and take it for granted that a processor notices when a sequence of symbols $(j, X, j), (j, Y, j) \cdots$ is passing and takes appropriate action. Such sequences of non-terminals require special handling only if there is a production with multiple nullable symbols in its
right-hand side.

Communication can be asynchronous, as in a UNIX™ pipeline, or synchronized as in a transputer system. In either case, the parse suffers from a severe communication bottleneck when all recognized symbols are passed down the pipeline. This is illustrated by a (synchronized) communication trace of *John saw a lion in the zoo* in Figure 6. With some effort, communication can be reduced a great deal.

4. REDUCING THE COMMUNICATION

There are several ways to detect that a symbol cannot be used by another processor further down the pipeline. Firstly, consider symbols \( X \in V \) such that in the right-hand side of a production \( X \) appears only as the first symbol. In such a case, a marked symbol \( \langle i, X, j \rangle \) can be used by processor \( P_i \), but not by any other processor. For example, *det* appears in a right-hand side of a production only in \( NP \rightarrow *det *noun \). If \( P_i \) hits upon a determiner \( \langle i, *det, i \rangle \) or \( \langle i, *det, i+1 \rangle \), there is no way in which processors \( P_0 \cdots P_{i-1} \) can do anything useful with it. This can be generalized into
<table>
<thead>
<tr>
<th>t</th>
<th>0 ← 1</th>
<th>1 ← 2</th>
<th>2 ← 3</th>
<th>3 ← 4</th>
<th>4 ← 5</th>
<th>5 ← 6</th>
<th>6 ← 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>(2, *d, 2)</td>
<td>(3, *d, 3)</td>
<td>(3, NP, 4)</td>
<td>(5, *d, 5)</td>
<td>(6, *d, 6)</td>
<td>(6, NP, 7)</td>
<td>(7, *d, 7)</td>
</tr>
<tr>
<td>5</td>
<td>(3, *d, 3)</td>
<td>(3, NP, 4)</td>
<td>(4, *p, 5)</td>
<td>(6, *d, 6)</td>
<td>(6, NP, 7)</td>
<td>(7, *d, 7)</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>(3, *n, 4)</td>
<td>(2, NP, 4)</td>
<td>(5, *d, 5)</td>
<td>(6, *n, 7)</td>
<td>(5, NP, 7)</td>
<td>(7, *d, 7)</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>(3, NP, 4)</td>
<td>(4, *d, 4)</td>
<td>(5, *d, 6)</td>
<td>(6, NP, 7)</td>
<td>(7, *d, 7)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>(2, NP, 4)</td>
<td>(4, *p, 5)</td>
<td>(6, *d, 6)</td>
<td>(5, NP, 7)</td>
<td>(7, *d, 7)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>(1, VP, 4)</td>
<td>(5, *d, 5)</td>
<td>(6, *n, 7)</td>
<td>(4, PP, 7)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>(4, *d, 5)</td>
<td>(5, *d, 6)</td>
<td>(6, NP, 7)</td>
<td>(7, *d, 7)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>(5, *d, 5)</td>
<td>(6, *n, 7)</td>
<td>(4, PP, 7)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>(5, *d, 6)</td>
<td>(6, NP, 7)</td>
<td>(3, NP, 7)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>(6, *d, 6)</td>
<td>(5, NP, 7)</td>
<td>(7, *d, 7)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>(6, *n, 7)</td>
<td>(4, PP, 7)</td>
<td>(7, *d, 7)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>(6, NP, 7)</td>
<td>(3, NP, 7)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>(5, NP, 7)</td>
<td>(2, NP, 7)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>(4, PP, 7)</td>
<td>(7, *d, 7)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>(3, NP, 7)</td>
<td>(7, *d, 7)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>(2, NP, 7)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>(1, VP, 7)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>(7, *d, 7)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>(7, *d, 7)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6: Synchronized communication, no filtering

communication savings rule I:
Processor $P_i$ passes a marked symbol $(i,X,j)$ for arbitrary $j$ only if there are $A\in N$, $a\in V^*$, $\beta\in V^*$ such that $A\rightarrow\alpha X\beta \in P$.

rule I for $G$ is shown in tabular form in Figure 7.

A second, somewhat more involved communication savings scheme is the following. Each processor $P_i$ has "its own" terminal symbol $(i,a_{i+1},i+1)$. Is it possible, knowing the marked terminal symbol $(i,a_{i+1},i+1)$, to discard some symbols $(i+1,X,j)$ that are passed by $P_{i+1}$? It is simple to compute a set of symbols $FOLLOW(a)$ that logically can follow $a$ in an arbitrary valid sentence. For example, $P_2$ "owns" the terminal $(2,*det,3)$. It receives a noun phrase $(3,NP,4)$ from $P_3$ (following rule I). It is obvious, however, that $(3,NP,4)$ cannot be concatenated with $(2,*det,3)$, hence it should be discarded by $P_2$.

$P_i$ should discard any symbol $(i+1,X,j)$ for which $X\notin FOLLOW(a_{i+1})$. But a more subtle filtering is possible. As an example, consider the marked symbol $(3,*noun,4)$ that is received by $P_2$. As $(3,*noun,4)\in FOLLOW(*det)$, it is clearly a useful symbol, that is, useful to $P_2$. But one can argue that it cannot be useful to $P_1$; $P_0$ and hence need not be passed on. A close inspection of the parse table in Figure 4 shows that a *noun can be used only if a processor has an immediately preceding *det on its stack. As $(2,*det,3)$ is not sent to $P_1$; $P_0$ (rule I), there is no way in which one of these processors could add $(3,*noun,4)$ to its stack. In general, if $P_i$ owns a terminal $(i,a_{i+1})$, a symbol $(i+1,X,j)$ needs to be passed on if the combination $aX$ appears somewhere not at the beginning of the right-hand side of a production, and also if a combination $AX$ appears in the right-hand side of a production, and $A$ produces a string ending with $a$. More formally,

Communication savings rule II:
processor $P_i$, owning a terminal symbol $(i,a_i,i+1)$, passes a marked symbol $(i+1,X,j)$ for arbitrary $j$ only if one of the following cases applies:
\[
\begin{array}{cccccccc}
\text{*det} & *\text{noun} & *\text{verb} & *\text{prep} & \text{NP} & \text{PP} & \text{VP} & \text{S} \\
- & + & - & - & + & + & + & - \\
\end{array}
\]

+ in entry \(X\) means: \(P_i\) passes symbols \((i,X,j)\) to \(P_{i-1}\)

**Figure 7:** Communication savings table I

\[
\begin{array}{cccccccc}
a \setminus X & *\text{det} & *\text{noun} & *\text{verb} & *\text{prep} & \text{NP} & \text{PP} & \text{VP} & \text{S} \\
*\text{det} & - & - & - & - & - & - & - & - \\
*\text{noun} & - & - & - & - & - & + & + & - \\
*\text{verb} & - & - & - & - & - & - & - & - \\
*\text{prep} & - & - & - & - & - & - & - & - \\
\end{array}
\]

+ in entry \((a,X)\) means: if \(a_i+1 = a\), \(P_i\) passes a symbol \((i+1,X,j)\) to \(P_{i-1}\)

**Figure 8:** Communication savings table II

<table>
<thead>
<tr>
<th>(t)</th>
<th>0 ← 1</th>
<th>1 ← 2</th>
<th>2 ← 3</th>
<th>3 ← 4</th>
<th>4 ← 5</th>
<th>5 ← 6</th>
<th>6 ← 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>(\langle 3, *n, 4 \rangle)</td>
<td>-</td>
<td>-</td>
<td>(\langle 6, *n, 7 \rangle)</td>
<td>(\langle 7, S, 8 \rangle)</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>-</td>
<td>(\langle 3, NP, 4 \rangle)</td>
<td>-</td>
<td>(\langle 5, NP, 7 \rangle)</td>
<td>(\langle 6, NP, 7 \rangle)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>(\langle 2, NP, 4 \rangle)</td>
<td>-</td>
<td>(\langle 4, PP, 7 \rangle)</td>
<td>-</td>
<td>(\langle 7, S, 8 \rangle)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>(\langle 1, VP, 4 \rangle)</td>
<td>-</td>
<td>(\langle 4, PP, 7 \rangle)</td>
<td>-</td>
<td>(\langle 7, S, 8 \rangle)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>(\langle 4, PP, 7 \rangle)</td>
<td>(\langle 3, NP, 7 \rangle)</td>
<td>-</td>
<td>(\langle 7, S, 8 \rangle)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>(\langle 4, PP, 7 \rangle)</td>
<td>(\langle 2, NP, 7 \rangle)</td>
<td>(\langle 7, S, 8 \rangle)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>(\langle 1, VP, 7 \rangle)</td>
<td>(\langle 7, S, 8 \rangle)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>(\langle 7, S, 8 \rangle)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 9:** Synchronized communication with filtering

(i) there are \(B \in N, \beta \in V^*, \alpha \in V^*\) such that
\[B \rightarrow \alpha \alpha X \beta \in P;\]

(ii) there are \(A, B \in N, \alpha, \beta, \gamma \in V^*\) such that
\[A \Rightarrow *_{\gamma a} \text{ and } B \rightarrow \alpha \alpha X \beta \in P;\]

Communication savings tables II for our example grammar is shown in Figure 8. Perhaps surprisingly at first sight, in all but two cases a symbol is discarded at latest by the second processor that handles it! Only a completed propositional phrase and a completed verb phrase should be passed down the pipeline. All other symbols are processed and disposed with locally.

The (synchronized) communication trace in Figure 9 is a lot shorter now. The communication bottleneck has disappeared, most processors are idle some of the time, waiting for new useful symbols to arrive.

5. THE CONSTRUCTION OF A PARSE

Every processor can construct a partial parse forest as in Tomita's algorithm. We add a unique label to every symbol node in the stack, and we maintain a list of labels. With every *shift* and *reduce*, an entry is added to the partial parse list. This list of labels represents the parse in the following way: If a symbol is a leaf, only that symbol is added in our list. If a symbol is not a leaf, the symbol plus an ordered list of successor nodes is added. A partial parse for \(P_i\) is shown in Figure 10 (cf. the parse stack in Figure 5). We use letters \(a, b, c, \cdots\) as labels. In an ambiguous sentence, symbols can be recognized in more than one way. In that case, we add multiple ordered sets of successor labels to the parse list entry. Thus a shared forest representation of all parses is constructed. A parse list may contain *unreachable* entries: recognized symbols that are not part of any
valid parse tree. We may simply ignore them.

Our aim, however, is to construct a total parse forest, not a set of partial parse forests. To that end we change the algorithm into its final form as follows:

- Every symbol \((i,X,j)\) that is parsed by \(P_i\) gets a (global) label \(i.l\), where \(l\) is the (local) label of \((i,X,j)\). The label is integrated into the marked symbol notation: \((i,l,X,j)\). Hence the label column can be deleted from the partial parse lists.

- Within the partial parse list of each processor, we do not have to make entries for objects that have been defined elsewhere. We can simply refer to their original labels.

Thus a single parse list representing the complete shared forest is obtained. The parse list in Figure 11 contains quite a few unreachable entries, due to the choice of the rather clumsy grammar \(G\). If we had used a grammar with (3) \(NP \rightarrow *noun\), rather than (3) \(det \rightarrow e\), all determiners of zero width would be absent.

### 6. CONCLUSIONS AND FUTURE RESEARCH

A novel, efficient parallel generalized LR parser has been presented, based on a bottom-up approach that is related to conventional parallelizations of Earley’s algorithm. In the near future we intend to test the algorithm against substantial (subsets of) natural language grammars, rather than the canonical small examples.

### ACKNOWLEDGEMENTS

I am grateful to Marc Lankhorst and Anton Nijholt for constructive comments and pointing out some errors.

### REFERENCES


[Earley70] J. Earley: An Efficient Context-Free Parsing Algorithm, Communications of


