DETERMINISTIC TOP-DOWN AND BOTTOM-UP PARSING:
HISTORICAL NOTES AND BIBLIOGRAPHIES

ANTON NIJHOLT

MATHEMATICAL CENTRE 1983
Printed at the Mathematical Centre, 413 Kruislaan, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).
DETERMINISTIC TOP-DOWN AND BOTTOM-UP PARSING:
HISTORICAL NOTES AND BIBLIOGRAPHIES

ANTON NIJHOLT

MATHEMATISCH CENTRUM
AMSTERDAM 1983
CONTENTS

PREFACE

1. INTRODUCTION ........................................ 1

2. TOP-DOWN PARSING: A BIBLIOGRAPHY .............. 9

3. LR-GRAMMARS AND PARSING: A BIBLIOGRAPHY ....... 45

4. PRECEDENCE PARSING: A BIBLIOGRAPHY ........... 95
PREFACE

The theory of parsing was initiated in the sixties by the pioneering works of Lewis and Stearns, Knuth, Floyd, and Wirth and Weber. In the seventies a vast body of knowledge has been obtained in the three subfields to which the three bibliographies in this monograph are devoted, viz. top-down parsing, LR-grammars and parsing, and precedence parsing.

The total number of entries in these bibliographies exceeds thousand references, dealing primarily with theoretical problems, but also with more application oriented issues such as compiler construction techniques.

Anton Nijholt has performed an admirable job in collecting the material in these bibliographies. Based on his extensive research in parsing theory, he is eminently qualified for this work, and we expect that the further development of the theory of parsing will benefit substantially from his efforts.

Amsterdam, September 1982

Jaco de Bakker.
I think that I shall never see
A string as lovely as a tree.
For strings are made by fools like me,
And only parsers make a tree.

A string just lies there, plain and flat,
As boring as a welcome mat--
A rather dreary, flabby sight,
Of symbols stretched from left to right.

....

Some parsers gaily go to town,
Working from the root on down.
While others go from top to bottom,
Assigning forms to strings that's got em.

....

And there are parsers that are able
To drive constructions from a table,
Keeping guesses good and warm
With hints in Backus-Naur form.

....

(Parts from a poem on Trees by Peter Kugel,
SIGACT News, April 1975, page 19)
1. INTRODUCTION

1.1. General

In this Monograph three bibliographies on parsing are presented. These bibliographies are on top-down parsing LR parsing and precedence parsing, respectively. Although these bibliographies are mainly concerned with the theoretical issues related with parsing for (deterministic) context-free languages, several other aspects, e.g. compiler construction, have found their place in these bibliographies.

Languages are defined by grammars which give the syntactic definition of the language. Following Chomsky, in computer science grammars are generating systems. The task of parsing is to determine whether a given string of symbols is element of the language and, if so, to make the syntactic definition with respect to the grammar explicit. The latter can be done by showing how the given string can be generated by the syntactic rules of the grammar.

After the discovery of the correspondence between the definitions which were introduced by Chomsky[4] and the notation of the programming language ALGOL 60 (BNF, which stands for Backus Naur Form[1]) an enormous number of publications appeared in which properties of languages, grammars and parsing methods are discussed. Floyd[5] showed that, in general, programming languages are not context-free languages. However, considerable parts of programming languages can be described with context-free grammars. Therefore research concentrated on this class of grammars and languages and more general formalisms sometimes were obtained as generalizations which were based on the context-free grammars. The availability of a mathematical formalism led to a situation where theory was being developed which did not necessarily yield results which were applicable in computer science.

The need for a parsing method and a parser becomes clear whenever the use of a language is formalized in order to be able to communicate with hard- or software. For such an application it is not only necessary to investigate whether a given string of symbols is syntactically correct, but also the meaning which is associated with the string should be understood by the hard- or software. Therefore, in the case of a programming language, the parser is only part of a more complicated piece of software, the compiler, which translates the sentences (commands, statements, programs) of the user to another language which can be understood by the hard- or software.
In 1956 the first FORTRAN compiler was built. This compiler took about 18 man years to develop. Nowadays the building of a compiler for a programming language will take considerably less time. One of the reasons is that it has become customary to modularize the process of compilation into several phases such as e.g. lexical analysis, syntax analysis (parsing), code generation, error handling and code optimization. Because of this modularization several research areas have become prosperous and tools have been developed which can be used in the construction of each of these modules. Especially the automatic generation of lexical analyzers and parsers from the syntax of a language is no longer considered a big problem. The construction of the other parts of a compiler still asks for further formalization of the available methods. The attempts to automate the production of those parts of a compiler which deal with semantics (code generation) are based on certain enrichments of context-free grammars. Formalisms which are used are, among others, Van Wijngaarden grammars, affix grammars, attribute grammars and denotational semantics. Van Wijngaarden’s formalism has been used in the definition of ALGOL 68 [12]. Affix grammars and attribute grammars were introduced by Koster[8] and Knuth[7], respectively. Presently, the use of these types of grammars in compilers and compiler writing systems is receiving considerable attention. See e.g. the bibliography on attribute grammars compiled by Reiha[10]. In the course of the years compilers and compiler writing systems have been used for the translation and the development of programming languages, languages for editors and text processing, languages for file handling, relational data base managers, picture generation etc.

1.2. Parsing Theory

Parsing theory is part of formal language theory. Mathematical linguistics and the design of programming languages such as ALGOL have been the two main sources from which formal language theory has been developed. As stated by Greibach[6], until 1964 formal language theory still could be considered part of mathematical linguistics. After 1964 formal language theory developed as a branch of theoretical computer science.

From the point of view of parsing theory it is possible to distinguish the following three periods:

- Some theory and ideas are available from mathematical linguistics. However, compiler builders start (almost) from scratch and develop parsers as part of compilers for specific languages and computers. Ideas which were used were not always written down, partly because parsing was not always distinguished from the other parts of a compiler. First the interest focused on parsing and compilation of arithmetical expressions (formulae). Later it was recognized that this was only part of a more general problem of compiling programming languages (see Bauer[2]).
- Formal language theory is being developed. Models of compilers and theories concerning parsing are developed. Ideas which had been used by compiler builders were formalized into parsing methods. The theory starts to raise its own questions. It becomes possible to distinguish between engineering activities and scientific progress. In this period we can find the birth of the three parsing methods, LL(k), LR(k) and precedence parsing, which are covered by this Monograph. The first compiler writing systems are built. Top-down (recursive descent) and precedence oriented systems can be recognized. At the end of this period the first LR-based system is constructed.

- The framework is available and is used to embed new and old questions with both practical and theoretical considerations. Techniques are further formalized and generalized. New and sometimes better proofs appear for old results. Developments in formal language theory are reflected in parsing theory. For example, parallel rewriting systems, parsing in parallel environments and especially complexity considerations are topics which acquire attention. While in the second period we see that methods for parser construction replace ideas which appeared in the construction of ALGOL 60-like programming languages, we now can distinguish, after having seen the construction of some ALGOL 68 compilers, the growth of methods for the parsing of two-level grammars and certain classes of grammars which can be considered as subclasses of the two-level grammars (e.g. the affix and attribute grammars).

Wegner[11] distinguishes three phases of programming language development, corresponding roughly to the 1950's, 1960's and 1970's. These phases are discovery and description of concepts, elaboration and analysis of concepts and software technology, and are characterized by an empirical, mathematical and an engineering approach, respectively. These phases can also be recognized in the area of parsing and its applications. During the 1970's many parser generators were constructed and used as a tool in programming language development. For example, LL(1) and LR(1) based parser generators have been developed for use on the well-known UNIX-system. Moreover, this engineering approach has stimulated research in the parsing of ambiguous grammars and attribute grammars.
1.3. Contents of the Bibliographies

LL(k), LR(k) and simple precedence parsing methods are the most widely used methods for parsers and parser generating systems. Mostly these methods are used in a slightly restricted or a slightly generalized form. LL(k) parsing is a top-down parsing method, the others are bottom-up parsing methods. Both top-down and bottom-up methods have been used in the early years of compiler (writing systems) technology. During a rather long period in the 1970's LR-parsing has received considerable attention, both in theory and in practice. However, the last few years have also shown the development of some LL(1) based systems. Although there are interesting exceptions (e.g. in Japan, USSR and West-Germany) precedence oriented systems seem to have lost the interest of those who develop compilers and compiler writing systems. Nevertheless, like the LL- and LR-methods the precedence method is still one of the fundamentals of parsing theory. Therefore these three methods are dealt with in the bibliographies of this Monograph. Parsers in working compilers have been constructed using production-language. Except for a few references this implementation method has not been considered in the bibliographies. Obviously, there are many other parsing methods. When such a method resembles deterministic top-down parsing references can be found in the top-down bibliography. Similarly, some other deterministic bottom-up methods can be found in the LR- or precedence bibliographies. It follows that some references can be found in more than one of the bibliographies.

When you're looking for a reference and you don’t know which bibliography to consult then first try the LR bibliography, next the top-down bibliography and, finally, the precedence bibliography. The area of general context-free parsing is not considered in this Monograph. However, some references to classical papers (Earley, Kasami, Valiant and Younger) and a more recent paper by Graham, Harrison and Ruzzo can be found in the LR-Bibliography.

There exist many non-context-free parsing methods. Some of these, especially those which have found practical application and which are related to the three methods treated here, are mentioned in the bibliographies. There exist various non-conventional parsing techniques. The obvious example is parsing in parallel computational environments. For natural languages (speech) special parsing methods, e.g. the use of augmented transition networks, have been developed. Another formalism which should be mentioned and which does not appear in the bibliographies is an extension of the context-free grammars, the so-called definite clause grammars (Pereira and Warren[9]). These grammars are a special case of a formalism introduced by Colmerauer. The definite clause grammar of a language can be considered as a program in the programming language PROLOG. When this program is executed it behaves as a top-down parser. Writing parsers in PROLOG has been made simple since it is possible to use a so-called grammar rule notation.
Each rule can be provided with arguments which make it possible to add
non-context-free and semantic information in the parser.

Attempts have been made to compare the LL, LR and precedence parsing
methods. A grammar for a programming language can be designed for especially one
of these methods. In that case the other methods are not necessarily suitable for this
grammar. Observations on the parsing method and resulting small manual changes in
the grammar can already lead to an enormous reduction of time and space
requirements of the parser. On the other hand, when a parsing method is selected for
inclusion in a compiler writing system it is necessary to base the choice on general
properties of the parsing methods and on the programming language grammars to be
expected. Among others, when parsers are to be generated automatically, it is
necessary to consider the theoretical generality of the method, the possibility to work
for natural grammars for programming languages, the space and time requirements of
the resulting parsers, the possibility to optimize the parsers, the possibility to include,
automatically, error detection, correction and recovery methods in the parser, and
the possibility to handle the semantics which is associated with the grammar rules.

Among others, the following topics are covered by the references.

a. equivalence properties of language classes, language hierarchies, intercalation
   theorems, properties of parse trees, grammatical covers, grammar forms,
   grammatical inference

b. relationships between grammar classes, transformations between grammar
   classes, generalizations of LL- and LR-grammars for other than context-free
   languages, deterministic translation grammars, LL- and LR- attributed gram-
   mars, decidability and complexity results for deciding whether a grammar belongs
to a certain family of grammars

c. relationships between parsing methods, unifying approaches of parsing methods,
   sizes of parsers, parser optimization, time and space complexity of parsing
   methods, parsing methods used in compilers and compiler writing systems, error
   handling (detection, correction and recovery), the design of parsing grammars for
   programming languages, incremental parsing, parsing in parallel environments,
parsing of two-level grammars

The bibliographies in this Monograph contain more than 1000 different references
and the names of more than 600 authors.
ACKNOWLEDGEMENTS

"Car enfin non omnia possumus omnes. & un homme seul ne peut parcourir tous les Pays & toutes les Bibliothéques, ny lire tous les Livres, ny puiser par consequent dans toutes les sources qui luy pourroient faciliter ce travail."

This was written by S. de Brossard in the introduction of the third part of the second edition of Dictionnaire de Musique which appeared in 1705. In this introduction of Catalogue des Auteurs Brossard explains that he wanted to give a detailed bibliography containing work and names of authors "qui ont écrit touchant la Musique". However: "Mais il faut que je l’avoue, malgré tout mon travail; mes Memoires ne suffisent pas pour executer avec l’exactitude que je souhaiterois, un projet de cette nature."

Therefore Brossard confines himself to the listing of three categories of names of authors without giving further references. These categories, summarizing ten years of work and containing about 230, 100 and 600 names, respectively, are composed as follows.

"Dans la premiere on trouvera les Noms des Auteurs que j’ay vus, lus & examinez moy-meme.

Dans la 2e, on trouvera les Noms de ceux que je n’ay pas encore eu le temps, ny l’occasion de lire & d’examiner, mais qui sont aisez a trouver, & que j’espere de lire avec le temps.

Dans la 3e, enfin on trouvera les Noms de ceux que je n’ay point lus, ny vus, & que je connois que par les yeux, & sur la foy d’autrui."

After spending only a few months on the compilation of this Monograph, with the help of review and abstract journals and with the help of modern tools which make it possible to store, change and format extensive lists of publications it is almost impudent to do so, but I wish to dedicate this Monograph to S. de Brossard.

Several approaches to the parsing problem fall outside the scope of the present Monograph. Hopefully I will have the opportunity to edit a more complete version of this Monograph in the future. A number of people have contributed to this Monograph in various ways. Peter Kugel gave me permission to use his poem "Trees". Without Adri Breukink Brossard would have been unknown to me. Many years ago Leo Verbeek introduced me to the parsing problem. Jaco de Bakker gave me the opportunity to spend some years on research in topics related with parsing. Michael Harrison and Derick Wood were among those who gave me feedback at several occasions. I gratefully acknowledge the help of those who contributed to earlier versions of parts of these bibliographies. Hans Meijer and Rinus Plasmeijer gave
some help with the phototypesetter. Finally, I thank the Informatics Department of the Katholieke Universiteit Nijmegen in the Netherlands for providing me with the necessary opportunities and facilities which made this Monograph possible. The Monograph has been completed at the Computer Science Department of Twente University of Technology in Enschede.
REFERENCES


2. TOP-DOWN PARSING: A BIBLIOGRAPHY

SURVEY OF THE LITERATURE

2.1. General and History

This bibliography contains references to reports and papers on (deterministic) top-down parsing, recursive descent and LL(k) grammars and languages. In 1978 Wood[400] compiled a bibliography on 'top-down deterministic parsing'. In this bibliography we give a more recent and more complete survey of papers and reports on top-down deterministic parsing. Many, but not all the references of [400] have been included. Some of the papers not included are mentioned in other recent bibliographies which deal with other approaches of the parsing problem for context-free grammars and languages. These other bibliographies are on:

- LR-grammars and parsing (see Section 3 of this Monograph and Burgess and James[60])
- precedence relations (see Section 4 of this Monograph and Nijholt[264])
- error handling (Ciesinger[69])
- translator writing tools (Meijer and Nijholt[237])

LL(k) grammars were introduced by Lewis and Stearns[217,218]. However, in [217] they used the name TD(k) grammar (Top Down grammar). The name (parsing from Left to right using Leftmost derivations and k symbols of lookahead) was introduced by Knuth[181], and most of their formal properties were first discussed by Rosenkrantz and Stearns[306]. Independently, Wood[396], building on earlier work in Great Britain, and Thompson and Booth[357] introduced similar classes of grammars.

Long before the formal introduction of LL(k) grammars parsing methods suitable for LL-type grammars have been used in compilers and compiler writing systems. One of these methods has become known as recursive descent, a method which can also be considered as parsing with restricted transition diagrams (cf. Conway[77]) and restricted parsing machines (cf. Knuth[181]). Recursive descent parsing is sometimes attributed to Lucas[227]. In that case the parser consists of a set of recursive procedures which recognize the input. LL(k) parsing is also known as predictive
parsing. In that case the implementation of the parser uses a table and a pushdown stack. In fact, the formal and explicit introduction of the pushdown stack in mathematical linguistics was motivated by this predictive parsing method. In 1961 Oettinger[272] explains that the method grew out of reflections on a technique used by I. Rhodes in automatic translation of natural languages. The following explanation is from Oettinger's paper. "The Rhodes method of "predictive" syntactic analysis is based on the observation that in scanning through a Russian sentence from left to right it is possible, on the one hand, to make predictions about the syntactic structures to be met further to the right and, on the other hand, to determine the syntactic role of the word currently being scanned by testing what previously made predictions it fulfills. The predictions are stored in a linear array called the "prediction pool" which behaves approximately as a pushdown store. Before a new sentence is scanned, a set of initial predictions is entered in the pool. The first word of the sentence is then admitted, and a test is made to see if the topmost prediction in the pool will accept it. If so, the successful prediction is erased from the prediction pool, and new predictions based both on lexical data about the word from a dictionary and on syntactic rules embodied in the predictive analysis system are entered into the prediction pool on top of whatever earlier predictions may have remained there. The system is then ready to process the next word in the sentence. " In Bauer[28] other remarks can be found on parsing methods in the early years of compiler construction.

2.2. Properties of Grammars and Languages

Programming languages are not necessarily LL(k) languages. In fact, some simple constructs in wellknown programming languages can not be described by any LL(k) grammar. However, in the practice of compiler writing such a problem can almost always easily be handled. It is desirable whether, for a fixed k, a context-free grammar is LL(k). Moreover, there exist methods to investigate whether a given language is not LL(k) (cf. Beatty[30,32]). For any two LL(k) grammars it is decidable whether they are equivalent, i.e., whether they generate the same language. This was first shown by Rosenkrantz and Stearns[308] using automata theory. This pushdown automata approach has been the start of extensive research activities to solve the equivalence problem for deterministic languages. Korenjak and Hopcroft[187] have used a more direct scheme (based on grammars) to solve the problem for a simple subclass of the LL grammars. See also Wood[399] and Butzbach[63]. Attempts have been made to generalize this direct scheme such that it can be used for the equivalence problem of LL(k) grammars. See Harrison, Havel and Yehudai[133], Olshansky and Pnueli[270] and Tomita[360,361].

LL(k) grammars are properly included in the class of LR(k) grammars. The LL(k) languages are properly included in the family of deterministic or LR(1) languages. There exists an infinite hierarchy of LL(k) languages based on the look-ahead k (Kurki-Suonio[206]).
From the point of view of parsing LL(1) grammars are very attractive. Software has been developed to transform grammars into LL(1) type grammars. A successful example is SID (Syntax Improving Device) (see Foster[99]), but also in more recently constructed compiler writing systems transformations are performed to obtain a grammar which is more suitable for the system. Such transformations include left factoring, elimination of left recursion, transformations to Greibach normal form and some others. These transformations should be done in such a way that the original semantics of the language can be preserved. A survey of research in this area can be found in Nijholt[255]. For existing programming languages (e.g. FORTRAN, ALGOL 60, ALGOL 68, PASCAL and ADA) grammars have been designed which can be used by top-down parsing methods.

2.3. Parsing and Error-Handling

LL(k) parsing can be done in linear time. In Hunt, Szymanski and Ullman[149] a lower bound for LL(k) testing is indirectly obtained from testing the LR(k) property. More recently, Sippu and Soisalon-Soininen[332] have obtained a faster algorithm for LL(k) testing. There exist algorithms which perform the test for subclasses of the LL(k) grammars. E.g., for k = 1 (Johnson and Sethi[168,169]) and for strong LL(k) grammars (Hunt, Szymanski and Ullman[155]).

Apart from the references to papers and reports with theoretical results on LL(k) grammars, parsing and languages, references can be found in publications which deal with error handling and with the construction of parsers (or compilers) and parser generating systems (or compiler writing systems) in which essential use is made of properties of LL(k) grammars and parsing. Irons[164] describes one of the first top-down error recovery mechanisms. Since then, interest in formal methods which can be used in the automatic generation of error correcting parsers has rapidly increased. In Burgess[59] it is shown, among others, how top-down parsers for Wood's left factored grammars can recover from errors. Wirth[391] discusses error handling in a top-down PL/0 parser. Lewi et al[215] and Milton, Kirchhoff and Rowland[240] describe LL parser generators with error recovery and error correcting mechanisms. Fischer, Milton and Quiring[96] have introduced an insertion-only LL error-corrector which can be used for those LL(1) grammars for which errors can be repaired by a suitable insertion of a terminal string. LL(k) parsers can be constructed such that they have the property that each erroneous input symbol will be immediately detected. This is not the case for strong LL parsers. However, each LL(1) grammar can be modified such that its strong LL parser has this Immediate Error Detection property (cf. Fischer, Tai and Milton[94]). Other correction techniques for LL parsers can be found in, among others, Amman[9], Seeger[321], Pai[273], Pai and Kieburz[275] and Roehrich[302]. In Backhouse[18] a simple recovery scheme is implemented in a recursive descent parser for PL/0.
2.4. More General Than Context-Free

By allowing backtracking top-down parsing methods can be made to work for every context-free language. However, such a method will be exponential in time complexity. Whenever a grammar is not LL(1) then there is still the possibility to use an LL(1) parsing method. For example, methods in Aho, Johnson and Ullman[3] and Wharton[386] make it possible to parse ambiguous grammars. Moreover, in a compiler writing system attempts can be made to modify the input grammar.

It is also possible to consider more general classes of grammars and languages and use a generalization of the LL parsing method. This has been done for certain subclasses of the context-sensitive grammars. See e.g. Barth[27] and Szabo[349]. Special methods have been developed for indexed grammars (Sebesta and Jones [320], Parchmann, Duske and Specht[277, 278] and Weiss[382]) and for macro grammars (Heydthausen and Mehilhorn[135] and Mehilhorn[235]). Parsing attribute grammars can be done in such a way that evaluating the attributes can influence the parsing. See Lewis, Rosenkrantz and Stearns[220] for the general theory of deterministic attributed translation. LL parsing and evaluation of attribute grammars is discussed in Madsen[229], Madsen and Jones[228], Milton[239], Milton and Fischer[241], Raiha and Ukkonen[296], Rowland[311] and Wilhelm[389]. Top-down parsing of two-level (van Wijngaarden) grammars has been discussed in, e.g., Barringer[24], Hunter et al[161] (see also Meijer[236]) and Meertens and van Vliet[234].

Instead of using a fixed amount k of look-ahead it is also possible to perform LL parsing with regular look-ahead. In this way LL-regular grammars have been obtained (Jarzabek and Krawczyk[167] and Nijholt[256]). In Poplawski[287] this class is further generalized in order to include ambiguous grammars which can be parsed using disambiguating rules. The equivalence problem for LL-regular grammars is decidable (Nijholt[260]).

2.5. Compilers and Compiler Writing Systems

Lewis and Rosenkrantz[219] first used LL theory and automata theory to develop an ALGOL compiler. See also the compiler described in Wirth[390] and the generating systems in Rosen[303], Ward[376], Lewi et al[214], Hunter et al[161], Milton et al[240], Pyster[293], Shyamasundar[328], Wilhelm et al[388] and Aronson and Laforce[15].
REFERENCES


15. M. Aronson and D.S. Laforce. COG User’s guide. TR 81-8, Computer Science Department, SUNY, Albany, 1981.


88. A.G. Duncan and J.S. Hutchison. Using attribute grammars to test designs and implementations. 5th Int. Conf. on *Software Engineering*, 1981 (San Diego), 170-178.


115. R. Giegerich and R. Wilhelm. Implementierbarkeit attributierter grammatica- 

116. A.E. Glennie. On the syntax machine and the construction of a universal 

Programmirovanie 11 (1975), Nr.6, 880-890 (1976).

118. V.N. Glushkova. Towards a simplification of LL(k) grammars. Program. 

Software 5 (1979), 166-172.

120. N.G. Grafeeva. A class of grammars which allow parsing with backtracking by 
86-90.


122. S.A. Greibach. A new normal-form theorem for context-free phrase structure 

123. S.A. Greibach. A simple proof of the standard form theorem for context-free 

124. S.A. Greibach. Comments on the roots of theorems and languages, both easy 

125. M. Griffiths. Analyse deterministe et compileurs. Thesis for Docteur d'etat, 
University of Grenoble, 1969.

126. M. Griffiths and M. Peltier. Grammar transformation as an aid to compiler 
production. Manuscript, undated.

127. T.V. Griffiths and S.R. Petrick. On the relative efficiencies of context-free 

128. T.V. Griffiths and S.R. Petrick. Top-down versus bottom-up analysis. IFIP 
Congress 68, Software 1, Booklet B, 80-85.


211. J. van Leeuwen. An elementary proof that a certain context-free language is not LL(k) and a generalization. Notes, 6 October 1972.


257. A. Nijholt. A framework for classes of grammars between the LL(k) and LR(k) grammars. CSTR No. 80-CS-25, McMaster University, Hamilton, Ontario, 1980.


293. A. Pyster. ZUSE User's manual. TRCS 81-04, Dept. of Computer Science, University of California, Santa Barbara, 1981.


321. H. Seeger. Fehlerbehandlung in LL-Parcern. Diplomarbeit, Technische

322. S. Sekimoto. A characterization of look-ahead stateless DPDA’s of a
No.6, 490-491, 1980.

323. V.W. Setzer. Non recursive top-down syntax analysis. Software - Practice
and Experience 9 (1979), 237-245.

324. V.V. Sevcenko. The parametrization of LL(k) grammars. Software of

325. V.V. Sevcenko. On the question of deterministic methods for top-down

326. V.V. Sevcenko. A certain approach to the problem of syntactical analysis
(Russian). Kibernetika (Kiev) 1974, No.4, 30-38. [MR 53#12112, 1977].

327. B.A. Sheil. Observations on context-free parsing. Harvard University, Center
for Research in Computing Technology, TR-12-76, 1976. Also: SMIL
(Stockholm), 1976.


329. B.A. Silverberg. Using a grammatical formalism as a programming language.

330. B.P. Sindeev. Transformation algorithm to produce syntax in a form
convenient for single-pass compiling. Programmirovanie 1975, No.4, July-
Aug., 36-44.

331. S. Sippu and E. Soisalon-Soininen. On constructing LL(k) parsers. In:
Automata, Languages and Programming, H.A. Maurer (ed.), Lect. Notes in

332. S. Sippu and E. Soisalon-Soininen. Characterizations of the LL(k) property.
In: Automata, Languages and Programming, J.W. de Bakker and J. van
596-608.


3. LR-Grammars and Parsing: A Bibliography

Survey of the Literature

3.1. General and History

This bibliography contains references to papers and reports on LR-grammars and parsing. Other recently compiled bibliographies on parsing deal with:

- top-down parsing (see Section 2 of this Monograph and Wood[550])
- precedence relations (see Section 4 of this Monograph and Nijholt[364])
- error handling (Ciesinger[77])
- translator writing tools (Meijer and Nijholt[335])

According to LaLonde[295] 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." According to Aho[16] "LR parsing is the single most promising parsing technique known at this time (1976)."

All these nice properties were not immediately recognized when this class of grammars was introduced. LR(k) grammars (parsing from left to right using Rightmost reductions and k symbols of lookahead) were introduced by Knuth[264] in 1965. Many of the formal properties of LR(k) grammars have already been discussed in this original paper. Later papers have been concerned with both more practical approaches of the parsing problem for LR(k) grammars and more formal treatment of the properties of LR(k) grammars and parsing. Classical theoretical problems are e.g. the relationships with other classes of grammars, decidability and equivalence problems and relationships with (deterministic) pushdown automata. LR(k) grammars can be considered as the largest class of grammars which can be parsed from left to right using a pushdown stack and a deterministic finite control. The class of LR(1) languages coincides with the class of deterministic languages (Ginsburg and Greibach[163]). There exist direct transformations from LR(k) to LR(1) grammars (cf. Mickunas[341] and Mickunas, Lancaster and Schneider[342]).
3.2. Properties of Grammars and Languages

LR(k) grammars are usually defined by introducing conditions on the rightmost derivations of a context-free grammar. Many, mostly slightly different, definitions of this type exist. See Geller and Harrison[154] for a survey and some basic consequences of the differences. Other definitions are possible, e.g. it is possible to put conditions on parsers which can be constructed for general context-free grammars (see e.g. Heilbrunner[205]). Definitions range from very formal (e.g. in terms of category theory, Hotz and Claus[215]) to operational. For practical reasons (parser size) various restrictions have been defined on the LR(k) definition. Often this is done in terms of properties of the state sets of the LR parser. In this way SLR(k) and LALR(k) grammars have been introduced.

Many classes of grammars have been defined between the LL(k) and LR(k) grammars. Deussen[108]. Demers[91] and Brosigol[63] give unifying approaches for top-down and bottom-up parsing methods for these grammar classes. Another attempt to include such grammar classes in a general framework is in Nijholt[360]. LR(k) grammars can be transformed to more restricted classes of grammars. This is one of the theoretical areas where strict deterministic grammars (Harrison and Havel[196]) play an important role (cf. Moura[349]). LR(k) grammars can be transformed to certain types of bounded (right) context and precedence grammars. See Graham[173,175] and Gray and Harrison[181]. Some subclasses of the LR(k) grammars can be transformed to LL(k) grammars (cf. Soisalon-Soininen and Ukkonen[481] and Hammer[189]). The proof that each LL(k) grammar is an LR(k) grammar has developed from ‘intuitively clear’ (Knuth[265]) to formal. In [363] the many different proofs are compared. DeRemer[98] has first shown that the class of simple precedence grammars is included in the class of SLR(1) grammars.

The class of LR(1) languages coincides with the class of deterministic languages. Just as the proofs that each LL(k) grammar is an LR(k) grammar the proofs of this property illustrate the development and the use of mathematical proof techniques in formal language theory. Arguments and proofs yielding the result can be found in Knuth[264], Lehman[305], Hopcroft and Ullman[210,211], Aho and Ullman[13] and Harrison and Havel[197]. Especially in the latter paper the result is obtained in a rigorous way. The solution for the equivalence problem for deterministic languages has been approached bottom-up. That is, first the equivalence problem for regular languages was solved, then the equivalence problem for simple deterministic and LL(k) languages was shown to be decidable. Valiant[525] introduced new techniques which started new activities in this area. At this moment the results obtained by Oyamaguchi, Honda and Inagaki[376] and by Ukkonen[517] are the most general available. In this bibliography this research area is not covered.
3.3. Parsing and Parser Optimization

LR(k) grammars can be parsed in linear time. The number of states of an LR(k) parser can be exponential in the size of the grammar. Testing whether for a fixed value of k an arbitrary grammar is LR(k) can be done in deterministic polynomial time. In Hunt, Szymanski and Ullman[221] lower time bounds on the complexity of LR(k) and SLR(k) testing can be found. It is not known whether testing for the LALR(k) property can be done in polynomial time. Testing can be done by constructing the LALR(k) parser. However, this may take exponential time and space. In Ukkonen and Soisalon-Soininen[518] it is shown that LALR(k) testing is PSPACE-complete. Lower bounds on the sizes of LR-parsers are discussed in Pittil[403] and in Ukkonen[520].

Before the introduction of LR(k) grammars parsing methods have been introduced and used in the construction of compilers and compiler writing systems which resemble (restricted versions of) the LR parsing method. Moreover, some of these methods are suitable for grammars which can generate all deterministic languages. See e.g. Floyd[141] (1964) and Eickel et al[124] (1962; see also Schkolnick[450]). LR parsing is an example of a shift-reduce parsing method. The origins of such a method can be found in Floyd[140]. In Bauer[45] other remarks can be found on parsing methods in the early years of compiler construction. Since 1965 LR(k) (or SLR(k), LALR(k) etc.) grammars have been given for programming languages ranging from FORTRAN (Rauhauser[420], Russel[441]) to ADA (Persch et al[397], Wetherell [538] and others).

Apart from the references to papers and reports with theoretical results on LR(k) grammars, languages and parsing, references can be found to publications which deal with the construction of parsers (or compilers) and parser generating systems (or compiler writing systems) in which essential use is made of properties of LR(k) grammars and parsing. Some papers which have had much impact on the practical construction of LR parsers are by Korenjak[271], DeRemer[102], LaLonde, Lee and Horning[291], Aho and Johnson[14], Anderson, Eve and Horning[28] and Pager [386]. In some of these papers, and, moreover, in e.g. Purdom[408,410] and Horning and LaLonde[213] empirical results on the size and the speed of LR parsers are reported.

Parser efficiency can be increased by eliminating semantically irrelevant reductions from an LR parser. Moreover, it is possible to eliminate "error" and "don't care" entries from the tables which are constructed for an LR parser. One way to compress tables is by using lists. Such a method is used in the Yacc compiler writing system[241]. Among others, the following papers are concerned with parser optimization: Aho and Ullman[6,10], Anderson, Eve and Horning[28], Backhouse[39], Demers[90], Goos[169], Joliat[243,246], LaLonde[294], Makinouchi[326], Pager[385], Sekimo et al[457] and Soisalon-Soininen[482,484]. In Dencker et al[95] the algorithmic complexity of six compression methods for parser tables are
compared and empirical results of their implementations for several programming languages are given.

3.4. More General Than Context-Free

Whenever a grammar is not LR(1), this does not necessarily mean that an LR(1) parsing method can not be used. Information which is available in the symbol table, rules concerning the precedence of symbols (operators) and other semantics can be embedded in the LR parser to resolve possible parsing conflicts. Aho, Johnson and Ullman[15], Tarhio[501] and Terry[505] discuss LR parsing for ambiguous grammars. Demers[89], Earley[123], Ruzicka[443] and Wharton[539] give some theoretical results in this area. Purdom and Brown[412], Kron et al[289] and Gillett and Leach[163] are concerned with the same problem. In the Yacc compiler writing system[241] the user has the possibility to add disambiguating rules to the syntax of the grammar for which an LR parser has to be constructed.

Less ad hoc generalizations of an LR parser are obtained if the parsing problem for more powerful classes of grammars is considered. LR-type grammars and parsing methods have been introduced for indexed grammars (Sebesta and Jones[455]; see also Parchmann et al[389]) and for context-sensitive grammars (cf. Walters[529]). Crowe[85] and Watt[530,531] consider the parsing problem for affix grammars. Just as in the case of context-free grammars, parsing methods for two-level grammars (van Wijngaarden grammars, W-grammars) have developed from the application of ad hoc ideas in the construction of a compiler (in this case for ALGOL 68) to formal parsing methods. See Birrell[52], Nadrchal et al[351] and Boom[56]. In Hunter et al[232] LL versus LR parsing is discussed. Meijer[334] comments on this paper. LR-type parsing of two-level grammars is treated in Ambler[22], Bowlden[62] and Wegner[355]. In Friede[150] restrictions on the definition of two-level grammars are introduced in order to obtain one-pass parsable grammars.

The problem of parsing and translation for Knuth's attribute grammars has become one of the more important topics in parsing theory. Among the papers dealing with this topic are those of Lewis, Rosenkrantz and Stearns[313], Jones and Madsen[247], Madsen[320], Milton[344], Raita and Ukkonen[417], Rowland[439], Tarhio[500], Watt[532] and Wilhelm[541].

3.5. Automatic Parser Construction and Error-Handling

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, when we give an LR(0) parser the possibility of a look-ahead of one symbol then most conflicts can be resolved. Therefore automatic construction consists of first computing the LR(0) parser and then a method is used to incorporate a look-ahead of one symbol in this parser. This look-ahead computation
has lead to the definitions of SLR(1), *comprehensive* LR(1), *not quite* LALR(1) and
LALR(1) grammars. In Kristensen and Madsen[282] various methods (up to 1979) to
compute LALR(k) look-ahead are compared. In DeRemer and Pennello[104] and in
Fischer[136] other methods for computing look-ahead are presented.
Leinieus[306], Peterson[398] and James[238] are concerned with error-correcting
techniques which can be used in LR parsers. The techniques of Graham and
Rhodes[175] have been used for SLR(k) grammars (see Druseikis and Ripley[116])
and for LALR(k) and LR(k) grammars (cf. Pennello and DeRemer[396] and
Mickunas and Modrey[343]). Other recent papers on error correction for LR parsers
are e.g. Tai[494], Dion and Fischer[112,113] Graham, Haley and Joy[178], Sippu and
Soisalon-Soininen[470-472] and Roehrich[433]. Other references can be found in the
bibliography and in Ciesinger[77]. A discussion on some error correcting techniques
can be found in Gries[185]. An important issue in error correction theory is whether
parsing is slowed down by the used technique.

It is impossible to start mentioning all the LR based parser generators which are in
use today. Especially at West-German universities (e.g. Karlsruhe, Berlin, Dort-
mund and Muenchen) there have been successful activities in this area. The
bibliography contains references to LR parser generators which have been developed
in Belgium, Danmark, Finland, Norway, Italy, Japan, India, Korea, Canada and the
U.S.A.
Johnson[241] describes the LALR(1) based parser generator Yacc which has been
implemented on the UNIX system. Fischer[136] has developed an *incremental* LR
parser constructor. That is, given an LR(1) parsing table corresponding to a
grammar, and given a set of productions to be inserted into or deleted from the
grammar, the parsing table is modified such that it corresponds to the new grammar.

Wetherell and Shannon[537] give an excellent account of ten years experience with
an LR based parser generator. Raiha[418] reports experiences with the HLP compiler
writing system. Obviously, any reader interested in the practical construction of LR
parsers and parser generators can find a lot more examples in this bibliography.
REFERENCES


76. Chi Corporation. LIS language implementation system. Chi Corporation, Cleveland, Ohio, undated report.


100. F.L. DeRemer. Extended LR(k) grammars and their parsers. Manuscript, University of California at Santa Cruz, 1970.


114. F.C. Druseikis. SLR(1) parser generator. Dept. of Computer Science, University of Arizona, 1975.


229. H. Hunt and D.J. Rosenkrantz. Efficient algorithms for structural similarity of grammars. Conf. Record of Seventh ACM Symp. on *Principles of Program-


259. R. Kemp. LR(0) grammars generated by a LR(0) parser. Report A79/18, Universitaet des Saarlandes, Saarbruecken, 1979.


321. M. Madsen and N.D. Jones. Letting the attributes influence the parsing. Undated manuscript.


332. G.T. McGruther. An approach to automatic syntax error detection, recovery and correction for LR(k) grammars. Naval Postgraduate School, Monterey,


339. M.D. Mickunas and V.B. Schneider. On the ability to cover LR(k) grammars with LR(1), SLR(1) and (1,1) bounded context grammars. IEEE Conf. Record of 14th Annual Symp. on Switching and Automata Theory, 1973, 109-121.


342. M.D. Mickunas, R.L. Lancaster and V.B. Schneider. Transforming LR(k) grammars to LR(1), SLR(1) and (1,1) bounded-right-context grammars. J. Assoc. Comput. Mach. 23 (1976), 511-533.


359. A. Nijholt. A framework for classes of grammars between the LL(k) and LR(k) grammars. CSTR-No. 80-CS-25, McMaster University, Hamilton, Ontario, 1980.


371. M.F. O'Hare. Modification of the LR(k) parsing technique to include automatic syntactic error recovery. Thesis, University of California, Santa Cruz, 1976.


408. P. Purdom. The size of LR(0) machines. Technical Report No.2, Computer Science Department, Indiana University, undated.


425. J.P. Rethore. Automates a piles communicantes. These 3e cycle, Department of Computer Science, University of Lille, 1975.


495. K.-C. Tai. On the implementation of parsing tables. SIGPLAN Notices 14, Nr.1, January 1979, 100-101.


4. PRECEDENCE PARSING: A BIBLIOGRAPHY

SURVEY OF THE LITERATURE

4.1. General and History

This bibliography contains references to papers in which precedence relations are used. Other recently compiled bibliographies on parsing deal with:

- top-down parsing (see Section 2 of this Monograph and Wood[244])
- LR-grammars and parsing (see Section 3 of this Monograph and Burgess and James[39])
- error handling (Ciesinger[41])
- translator writing tools (Meijer and Nijholt[168])

In Floyd[75] (1963) operator precedence relations and operator precedence grammars were introduced. In Wirth and Weber[238] (1966) the Wirth-Weber precedence relations and the simple precedence grammars are defined. Independently, Pair[179] has introduced simple precedence grammars in 1964. Precedence relations have been used in parsing and compiler construction before the appearance of the formal definitions in these two papers. Although a few papers are mentioned, this bibliography does not fully cover such publications. In Bauer[32] some notes on early research in parsing can be found. In that paper it is mentioned that the origins of precedence parsing can be traced back to at least 1956.

Apart from these papers the bibliography contains papers on the use of precedence relations in formal language theory (parsing, algebraic properties, graph grammars, grammatical inference, grammar forms, EOL-systems) and in compiler construction. Papers on Domolki’s syntax analysis algorithm are also included in this bibliography.
4.2. Properties of Grammars and Languages

In Aho and Ullman[2] precedence relations have been used in order to give a formal proof of the theorem that every deterministic language can be generated with an LR(1) grammar. Properties of precedence grammars and languages were first investigated by Fischer[74]. Graham[87] has extensively studied the relationships between precedence grammars and Floyd's bounded context grammars. Among others it is shown which class of precedence grammars is able to generate exactly the deterministic languages. Gray and Harrison[92] consider combinations of precedence and LR parsing techniques and they consider transformations from LR grammars to grammars for which such combined parsing techniques can be used. Yehudai[246] discusses some aspects of the definition of simple precedence grammars. A recent discussion on the role of precedence relations in the theory of parsing can be found in Nijholt[174].

In Floyd's scheme the relations are defined between the terminal symbols. Moreover, the grammar under consideration is supposed to be in operator form. That is, no two adjacent nonterminal symbols are allowed in the righthand sides of the grammar rules. A drawback of Floyd's scheme is that the corresponding precedence parsing algorithm can accept a superset of the language of the grammar. In Ruzicka[199] an algorithm is presented which can be used to decide whether the parsing algorithm accepts exactly the language of the underlying grammar. Another approach to this problem is followed by Henderson and Levy[104] and Levy[148]. They modify the precedence relations in an attempt to overcome this discrepancy. Unfortunately their attempts are not quite satisfactory (cf. Williams[232-234]).

Wirth and Weber generalize Floyd's scheme by defining precedence relations on the entire vocabulary. Moreover, the restriction to operator form is relaxed. In this way the simple precedence grammars are obtained. Other authors have further generalized this scheme and have defined combinations with bounded context techniques. In this way extended precedence, weak precedence, soft precedence, strong precedence, left local precedence, strict precedence, total precedence, parallel total precedence, right precedence, mixed strategy precedence, extended operator precedence and weak operator precedence grammars have been defined. This list is not complete. Among them the weak precedence grammars, introduced by Ichbiah and Morse[118], and the mixed strategy precedence grammars, introduced by McKeeman et al[164], have turned out to be practically useful. Colmerauer[48] and Gray and Harrison[93] present general theories of precedence analysis.
4.3. Precedence Relations and Functions

The precedence relations can be stored in a matrix, the so-called precedence matrix. This matrix is then used in the implementation of the parsing method. Research in the area of precedence parsing has centered on the possibility to reduce the storage requirements of this matrix, without affecting the parsing algorithm, and on methods to transform non-precedence grammars to precedence grammars. Reducing the storage requirements has been done by compressing the precedence matrix to linear precedence functions. If $n$ is the size of the vocabulary of the grammar, then $n \times n$ is the size of the matrix of precedence relations. However, for the precedence functions it is only necessary to store $2n$ entries.

Algorithms to compute the precedence relations have been given by Floyd[75], Martin[156], Wirth and Weber[238] and Hunt, Szymanski and Ullman[116]. In the latter paper it is shown that this computation can be done in $O(n^2)$ steps, where $n$ is the size of the grammar. Computation of linear precedence functions has been considered by Floyd[75], Wirth and Weber[238], Wirth[239], Aho and Ullman[3], Bell[34,35], Martin[157], Bertsch[36] and Duong-Kien et al[65]. See Aoe et al[12], Er[69] and Hunt, Szymanski and Ullman[116] for more recent results. In the latter paper it is shown that this computation can be done in $O(n)$ steps. Unfortunately, not for all precedence grammars there exist precedence functions. Moreover, the reduction of storage requirements can cause weaker error detection and recovery properties of the parser.

Algorithms to transform grammars to precedence grammars can be found in Fischer[74], Haynes[102], Lim[149], Learner and Lim[143], McAfee and Presser [161], Presser[185] and Presser and Melkanoff[186].

4.4. Parsing and Error-Handling

Precedence relations have played an important role in the theory and practise of building parsers, compilers and compiler writing systems. Cf. Hoare[111], McClure[162] and Aho and Ullman[2]. Compilers have been constructed with a parsing module based on precedence relations. Cf. Bauer, Becker and Graham[33] for an ALGOL compiler, Dewar and McCann[59] for a SNOBOL compiler and Natt and Simon[174] for an EULER compiler. Lindsey and Turner[150,151] and Meertens and van Vliet[167] have used precedence methods in the parsing and error recovery for ALGOL 68. The original version of the XPL compiler writing system (McKeeman, Horning and Wortman[165]) was based on precedence analysis (later replaced by an algorithm utilizing LR(k) techniques). LeCarme and Bochmann[144], Ichbiah and Morse[118], White and Presser[231] (the UCSB translator writing system) and Zimmer[250] describe other wellknown examples of parser generating systems based on precedence relations. See also Cook[49] and Schwenkel[203,204].
Precedence parsers for various programming languages have been compared empirically with other types of parsers. Both time and space requirements for precedence parsers are, sometimes considerably, larger than for SLR(1) and LALR(1) parsers (Horning and LaLonde[114]). However, good comparisons are difficult since, among others, the generality of the parsing method, the possibility to work for "natural" programming language grammars, and error detection and recovery possibilities should be taken into account. Nevertheless, the existing comparisons are biased in favor of non-precedence methods. Feldman and Gries[71] remark that a grammar for an average programming language has to be manipulated considerably before it is a precedence grammar. This can be contrasted with what has been called DeRemer's Thesis: "If a designer sets out to design an unambiguous CF grammar to specify the structural properties of a language, his result will be an LR(k) grammar." DeRemer's Thesis has been confirmed, and in fact refined to LALR(1) grammars, in various papers which are included in the LR-Bibliography.

After 1972 LR(1) and LL(1) parsing methods have become the main tools for building parsers and the parser generating part of translator writing systems (cf. Meijer and Nijholt[168]). However, there exist some interesting exceptions.

Precedence relations can be defined for various classes of grammars which are more general than the context-free grammars. They have been used for OL systems (Hladky[110]), graph grammars (Franck[77] and Kaul[131]) and context-sensitive and general phrase structure grammars (Khabbaz[132], Babinov[25] and Haskell[100]). In the latter paper a transformation from general phrase structure grammars to precedence form is given. Moreover, precedence relations appear in the definitions of subclasses of the indexed grammars (cf. Ordjan and Dzulakjan[178]) and in the area of parallel parsing methods (Fischer[72] and Pronina[191]).

Error-correcting techniques for simple precedence grammars have been given by Wirth[236], Leinius[146] and Rhodes[193] (see also Graham and Rhodes[89]). Some of the earlier methods are language-dependent. Hence, automatic generation of error-correcting parsers is not always possible with these methods. Certain parsing methods can announce errors as soon as the part of the input which has been seen thus far does no longer form a prefix of a sentence in the language. None of the methods in the above mentioned papers has this Correct Prefix Property. In Geller, Graham and Harrison[82] and in Moll[171] classes of grammars are defined for which parsers can have this property. Recently, various reports and theses have been published at West-German universities in which error correcting methods for precedence parsers are introduced (cf., among others, Buechler[38], Hauger[101], Gonser[85] and Richter[194]). Error recovery for precedence parsers has also been considered by Ripley[196].
REFERENCES


TITLES IN THIS SERIES


A. REZUS, A bibliography of lambda-calculi, combinatory logics and related topics, 1982. ISBN 90 6196 234 X.

ANTON NIJHOLT, Deterministic top-down and bottom-up parsing: Historical notes and bibliographies, 1983. ISBN 90 6196 245 5.