18

Annotated Bibliography

The purpose of this annotated bibliography is to supply the reader with more material and with more detail than was possible in the preceding chapters, rather than to just list the works referenced in the text. The annotations cover a considerable number of subjects that have not been treated in the rest of the book.

The printed version of this book includes only those literature references and their summaries that are actually referred to in it. The full literature list with summaries as far as available can be found on the web site of this book; it includes its own authors index and subject index.

This annotated bibliography differs in several respects from the habitual literature list.

- The annotated bibliography consists of four sections:
  - Main parsing material — papers about the main parsing techniques.
  - Further parsing material — papers about extensions of and refinements to the main parsing techniques, non-Chomsky systems, error recovery, etc.
  - Parser writing and application — both in computer science and in natural languages.
  - Support material — books and papers useful to the study of parsers.
- The entries in each section have been grouped into more detailed categories; for example, the main section contains categories for general CF parsing, LR parsing, precedence parsing, etc. For details see the Table of Contents at the beginning of this book.

  Most publications in parsing can easily be assigned a single category. Some that span two categories have been placed in one, with a reference in the other.
- The majority of the entries are annotated. This annotation is not a copy of the abstract provided with the paper (which generally says something about the results obtained) but is rather the result of an attempt to summarize the technical content in terms of what has been explained elsewhere in this book.
- The entries are ordered chronologically rather than alphabetically. This arrangement has the advantage that it is much more meaningful than a single alphabetic list, ordered on author names. Each section can be read as the history of research

on that particular aspect of parsing, related material is found closely together and recent material is easily separated from older publications. A disadvantage is that it is now difficult to locate entries by author; to remedy this, an author index (starting on page 651) has been supplied.

18.1 Major Parsing Subjects

18.1.1 Unrestricted PS and CS Grammars

1. **Tanaka**, Eiichi and Fu, King-Sun. Error-correcting parsers for formal languages. *IEEE Trans. Comput.*, C-27(7):605–616, July 1978. In addition to the error correction algorithms referred to in the title (for which see [301]) a version of the CYK algorithm for context-sensitive grammars is described. It requires the grammar to be in 2-form: no rule has a right-hand size longer than 2, and no rule has a left-hand size longer than its right-hand size. This limits the number of possible rule forms to 4: $A \rightarrow a$, $A \rightarrow BC$, $AB \rightarrow CB$ (right context), and $BA \rightarrow BC$ (left context). The algorithm is largely straightforward; for example, for rule $AB \rightarrow CB$, if C and B have been recognized adjacently, an A is recognized in the position of the C. Care has to be taken, however, to avoid recognizing a context for the application of a production rule when the context is not there at the right moment; a non-trivial condition is given for this, without explanation or proof.

18.1.2 General Context-Free Parsing

2. **Irons**, E. T. A syntax-directed compiler for ALGOL 60. *Commun. ACM*, 4(1):51–55, Jan. 1961. The first to describe a full parser. It is essentially a full backtracking recursive descent left-corner parser. The published program is corrected in a Letter to the Editor by B.H. Mayoh, *Commun. ACM*, 4(6):284, June 1961.

3. **Hays**, David G. Automatic language-data processing. In H. Borko, editor, *Computer Applications in the Behavioral Sciences*, pages 394–423. Prentice-Hall, 1962. Actually about machine translation of natural language. Contains descriptions of two parsing algorithms. The first is attributed to John Cocke of IBM Research, and is actually a CYK parser. All terminals have already been reduced to sets of non-terminals. The algorithm works by combining segments of the input ("phrases") corresponding to non-terminals, according to rules $X - Y = Z$ which are supplied in a list. The program iterates on the length of the phrases, and produces a list of numbered triples, consisting of a phrase and the numbers of its two direct constituents. The list is then scanned backwards to produce all parse trees. It is suggested that the parser might be modified to handle discontinuous phrases, phrases in which X and Y are not adjacent.
The second algorithm, "Dependency-Structure Determination", seems akin to chart parsing. The input sentence is scanned repeatedly and during each scan reductions appropriate at that scan are performed: first the reductions that bind tightest, for example the nouns modified by nouns (as in "computer screen"), then such entities modified by adjectives, then the articles, etc. The precise algorithm and precedence table seem to be constructed ad hoc.

4. **Kuno**, S. and Oettinger, A. G. Multiple-path syntactic analyzer. In *Information Processing 1962*, pages 306–312, Amsterdam, 1962. North-Holland. A pool of predictions is maintained during parsing. If the next input token and a prediction allows more than one new prediction, the prediction is duplicated as often as needed, and multiple new predictions result. If a prediction fails it is discarded. This is top-down breadth-first parsing.

5. **Sakai**, Itiroo. Syntax in universal translation. In *1961 International Conference on Machine Translation of Languages and Applied Language Analysis*, pages 593–608, London, 1962. Her Majesty's Stationery Office. Using a formalism that seems equivalent

to a CF grammar in Chomsky Normal Form and a parser that is essentially a CYK parser, the author describes a translation mechanism in which the source language sentence is transformed into a binary tree (by the CYK parser). Each production rule carries a mark telling if the order of the two constituents should be reversed in the target language. The target language sentence is then produced by following this new order and by replacing words. A simple Japanese-to-English example is provided.

6. **Greibach**, S. A. *Inverses of Phrase Structure Generators*. PhD thesis, Technical Report NSF-11, Harvard U., Cambridge, Mass., 1963.

7. **Greibach**, Sheila A. Formal parsing systems. *Commun. ACM*, 7(8):499–504, Aug. 1964. "A formal parsing system $G = (V, \mu, T, R)$ consists of two finite disjoint vocabularies, V and T, a many-to-many map, μ, from V onto T, and a recursive set R of strings in T called syntactic sentence classes" (verbatim). This is intended to solve an additional problem in parsing, which occurs often in natural languages: a symbol found in the input does not always uniquely identify a terminal symbol from the language (for example, *will* (verb) versus *will* (noun)). On this level, the language is given as the entire set R, but in practice it is given through a "context-free phrase structure generator", i.e. a grammar. To allow parsing, this grammar is brought into what is now known as Greibach Normal Form: each rule is of the form $Z \to aY_1 \cdots Y_m$, where a is a terminal symbol and Z and $Y_1 \cdots Ym$ are non-terminals. Now a *directed production analyser* is defined which consists of an unlimited set of pushdown stores and an input stream, the entries of which are sets of terminal symbols (in T), derived through μ from the lexical symbols (in V). For each consecutive input entry, the machine scans the stores for a top non-terminal Z for which there is a rule $Z \to aY_1 \cdots Y_m$ with a in the input set. A new store is filled with a copy of the old store and the top Z is replaced by $Y_1 \cdots Y_m$; if the resulting store is longer than the input, it is discarded. Stores will contain non-terminals only. For each store that is empty when the input is exhausted, a parsing has been found. This is in effect non-deterministic top-down parsing with a one-symbol look-ahead. This is probably the first description of a parser that will work for any CF grammar.
A large part of the paper is dedicated to undoing the damage done by converting to Greibach Normal Form.

8. **Greibach**, S. A. A new normal form theorem for context-free phrase structure grammars. *J. ACM*, 12:42–52, Jan. 1965. A CF grammar is in "Greibach Normal Form" when the right-hand sides of the rules all consist of a terminal followed by zero or more non-terminals. For such a grammar a parser can be constructed that consumes (matches) one token in each step; in fact it does a breadth-first search on stack configurations. An algorithm is given to convert any CF grammar into Greibach Normal Form. It basically develops the first non-terminal in each rule that violates the above condition, but much care has to be taken in that process.

9. **Griffiths**, T. V. and Petrick, S. R. On the relative efficiencies of context-free grammar recognizers. *Commun. ACM*, 8(5):289–300, May 1965. To achieve a unified view of the parsing techniques known at that time, the authors define a non-deterministic two-stack machine whose only type of instruction is the replacement of two given strings on the tops of both stacks by two other strings; the machine is started with the input on one stack and the start symbol on the other and it "recognizes" the input if both stacks get empty simultaneously. For each parsing technique considered, a simple mapping from the grammar to the machine instructions is given; the techniques covered are top-down (called top-down), left-corner (called bottom-up) and bottom-up (called direct-substitution). Next, look-ahead techniques are incorporated to attempt to make the machine deterministic. The authors identify left recursion as a trouble-spot. All grammars are required to be ε-free. The procedures for the three parsing methods are given in a Letter to the Editor, *Commun. ACM*, 8(10):594, Oct 1965.

10. **Younger**, Daniel H. Recognition and parsing of context-free languages in time n^3. *Inform. Control*, 10(2):189–208, Feb. 1967. A Boolean recognition matrix R is constructed in a bottom-up fashion, in which $R[i,l,p]$ indicates that the segment of the input string starting at position i with length l is a production of non-terminal p. This matrix can be filled in $O(n^3)$ actions, where n is the length of the input string. If $R[0,n,0]$ is set, the whole string is a production of non-terminal 0. Many of the bits in the matrix can never be used in any actual parsing; these can

be removed by doing a top-down scan starting from $R[0,n,0]$ and removing all bits not reached this way. If the matrix contains integer rather than Boolean elements, it is easy to fill it with the number of ways a given segment can be produced by a given non-terminal; this yields the ambiguity rate.

11. **Dömölki**, Bálint. A universal compiler system based on production rules. *BIT*, 8(4):262–275, Oct. 1968. The heart of the compiler system described here is a production system consisting of an ordered set of production rules, which are the inverses of the grammar rules; note that the notions "left-hand side" (lhs) and "right-hand side" (rhs) are reversed from their normal meanings in this abstract. The system attempts to derive the start symbol, by always applying the first applicable production rule (first in two respects: from the left in the string processed, and in the ordered set of production rules). This resolves shift/reduce conflicts in favor of reduce, and reduce/reduce conflicts by length and by the order of the production rules. When a reduction is found, the lhs of the reducing rule is offered for semantic processing and the rhs is pushed back into the input stream, to be reread. Since the length of the rhs is not restricted, the method can handle non-CF grammars.
The so-called "Syntactic Filter" uses a bitvector technique to determine if, and if so which, production rule is applicable: for every symbol i in the alphabet, there is a bitvector $B[i]$, with one bit for each of the positions in each lhs; this bit set to 1 if this position contains symbol i. There is also a bitvector U marking the first symbol of each lhs, and a bitvector V marking the last symbol of each lhs. Now, a stack of bitvectors Q_t is maintained, with $Q_0 = 0$ and $Q_t = ((Q_{t-1} >> 1) \vee U) \wedge B[i_t]$, where i_t is the t-th input symbol. Q_t contains the answer to the question whether the last j symbols received are the first j symbols of some lhs, for any lhs and j. A 1 "walks" through an lhs part of the Q vector, as this lhs is recognized. An occurrence of an lhs is found if $Q_t \wedge V \neq 0$. After doing a replacement, t is set back k places, where k is the length of the applied lhs, so a stack of Q_t-s must be maintained. If some $Q_t = 0$, we have an error. An interesting implementation of the Dömölki algorithm is given by Hext and Roberts [15].

12. **Unger**, S. H. A global parser for context-free phrase structure grammars. *Commun. ACM*, 11(4):240–247, April 1968. The Unger parser (as described in Section 4.1) is extended with a series of tests to avoid partitionings that could never lead to success. For example, a section of the input is never matched against a non-terminal if it begins with a token no production of the non-terminal could begin with. Several such tests are described and ways are given to statically derive the necessary information (FIRST sets, LAST sets, EXCLUDE sets) from the grammar. Although none of this changes the exponential character of the algorithm, the tests do result in a considerable speed-up in practice. (An essential correction to one of the flowcharts is given in *Commun. ACM*, 11(6):427, June 1968.)

13. **Kasami**, T. and Torii, K. A syntax-analysis procedure for unambiguous context-free grammars. *J. ACM*, 16(3):423–431, July 1969. A rather complicated presentation of a variant of the CYK algorithm, including the derivation of a $O(n^2 log\ n)$ time bound for unambiguous Chomsky Normal Form grammars.

14. **Earley**, J. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, Feb. 1970. This famous paper gives an informal description of the Earley algorithm. The algorithm is compared both theoretically and experimentally with some general search techniques and with the CYK algorithm. It easily beats the general search techniques. Although the CYK algorithm has the same worst-case efficiency as Earley's, it requires $O(n^3)$ on any grammar, whereas Earley's requires $O(n^2)$ on unambiguous grammars and $O(n)$ on bounded-state grammars. The algorithm is easily modified to handle Extended CF grammars. Tomita [161] has pointed out that the parse tree representation is incorrect: it combines rules rather than non-terminals (see Section 3.7.3.1).

15. **Hext**, J. B. and Roberts, P. S. Syntax analysis by Dömölki's algorithm. *Computer J.*, 13(3):263–271, Aug. 1970. Dömölki's algorithm [11] is a bottom-up parser in which the item sets are represented as bitvectors. A backtracking version is presented which can handle any grammar. To reduce the need for backtracking a 1-character look-ahead is introduced and an algorithm for determining the actions on the look-ahead is given. Since the internal state is recomputed by vector operations for each input character, the parse table is much smaller than usual and its entries are one bit each. This, and the fact that it is all bitvector operations, makes the algorithm suitable for implementation in hardware.

16. **Kay**, M. The MIND system. In R. Rustin, editor, *Natural Language Processing*, pages 155–188. Algorithmic Press, New York, 1973. The MIND system consists of the following components: morphological analyser, syntactic processor, disambiguator, semantic processor, and output component. The information placed in the labels of the arcs of the chart is used to pass on information from one component to another.

17. **Bouckaert**, M., Pirotte, A., and Snelling, M. Efficient parsing algorithms for general context-free parsers. *Inform. Sci.*, 8(1):1–26, Jan. 1975. The authors observe that the Predictor in an Earley parser will often predict items that start with symbols that can never match the first few symbols of the present input; such items will never bear fruit and could as well be left out. To accomplish this, they extend the k-symbol reduction look-ahead Earley parser with a t-symbol prediction mechanism; this results in very general M_k^t parsing machines, the properties of which are studied, in much formal detail. Three important conclusions can be drawn. Values of k or t larger than one lose much more on processing than they will normally gain on better prediction and sharper reduction; such parsers are better only for asymptotically long input strings. The Earley parser without look-ahead (M_0^0) performs better than the parser with 1 symbol look-ahead; Earley's recommendation to use always 1 symbol look-ahead is unsound. The best parser is M_0^1; i.e. use a one symbol predictive look-ahead and no reduction look-ahead.

18. **Valiant**, Leslie G. General context-free recognition in less than cubic time. *J. Comput. Syst. Sci.*, 10:308–315, 1975. Reduces CF recognition to bit matrix multiplication in three steps, as follows. For an input string of length n, an $n \times n$ matrix is constructed, the elements of which are sets of non-terminals from a grammar G in Chomsky Normal form; the diagonal just next to the main diagonal is prefilled based on the input string. Applying transitive closure to this matrix is equivalent to the CYK algorithm, but, just like transitive closure, that is $O(n^3)$. Next, the author shows how transitive closure can be reduced by divide-and-conquer to a sequence of matrix multiplications that can together be done in a time that is not more than a constant factor larger than required for one matrix multiplication. The third step involves decomposing the matrices of sets into sets of h Boolean matrices, where h is the number of non-terminals in G. To multiply two matrices, each of their h Boolean counterparts must be multiplied with all h others, requiring $h \times h$ matrix multiplications. The fourth step, doing these matrix multiplications in time $O(n^{2.81})$ by applying Strassen's[1] algorithm, is not described in the paper.

19. **Graham**, Susan L. and Harrison, Michael A. Parsing of general context-free languages. In *Advances in Computing, Vol. 14*, pages 77–185, New York, 1976. Academic Press. The 109 page article describes three algorithms in a more or less unified manner: CYK, Earley's, and Valiant's. The main body of the paper is concerned with bounds for time and space requirements. Sharper bounds than usual are derived for special grammars, for example, for linear grammars.

20. **Sheil**, B. Observations on context-free parsing. *Statistical Methods in Linguistics*, pages 71–109, 1976. The author proves that any CF backtracking parser will have a polynomial time dependency if provided with a "well-formed substring table" (WFST), which holds the well-formed substrings recognized so far and which is consulted before each attempt to recognize a substring. The time requirements of the parser is $O(n^{c+1})$ where c is the maximum number of non-terminals in any right-hand side. A *2-form grammar* is a CF grammar such that no production rule in the grammar has more than two non-terminals on the right-hand side; nearly all practical grammars are already 2-form. 2-form grammars, of which Chomsky Normal Form grammars are a subset, can be parsed in $O(n^3)$. An algorithm for a dividing top-down parser with a WFST is supplied. Required reading for anybody who wants to write or use a general CF grammar. Many practical hints and opinions (controversial and otherwise) are given.

---

[1] Volker Strassen, "Gaussian elimination is not optimal", *Numerische Mathematik*, 13:354-356, 1969. Shows how to multiply two 2×2 matrices using 7 multiplications rather than 8 and extends the principle to larger matrices.

21. **Deussen**, P. A unified approach to the generation and acception of formal languages. *Acta Inform.*, 9(4):377–390, 1978. Generation and recognition of formal languages are seen as special cases of Semi-Thue rewriting systems, which essentially rewrite a string u to a string v. By filling in the start symbol for u or v one obtains generation and recognition. To control the movements of the rewriting system, states are introduced, combined with left- or right-preference and length restrictions. This immediately leads to a 4×4 table of generators and recognizers. The rest of the paper examines and proves properties of these.

22. **Deussen**, Peter. One abstract accepting algorithm for all kinds of parsers. In Hermann A. Maurer, editor, *Automata, Languages and Programming*, volume 71 of *Lecture Notes in Computer Science*, pages 203–217. Springer-Verlag, Berlin, 1979. CF parsing is viewed as an abstract search problem, for which a high-level algorithm is given. The selection predicate involved is narrowed down to give known linear parsing methods.

23. **Graham**, S. L., Harrison, M. A., and Ruzzo, W. L. An improved context-free recognizer. *ACM Trans. Prog. Lang. Syst.*, 2(3):415–462, July 1980. The well-formed substring table of the CYK parser is filled with dotted items as in an LR parser rather than with the usual non-terminals. This allows the number of objects in each table entry to be reduced considerably. Special operators are defined to handle ε- and unit rules.
The authors do not employ any look-ahead in their parser; they claim that constructing the recognition table is pretty fast already and that probably more time will be spent in enumerating and analysing the resulting parse trees. They speed up the latter process by removing all useless entries before starting to generate parse trees. To this end, a top-down sweep through the triangle is performed, similar to the scheme to find all parse trees, which just marks all reachable entries without following up any of them twice. The non-marked entries are then removed (p. 443).
Much attention is paid to efficient implementation, using ingenious data structures.

24. **Kilbury**, James. Chart parsing and the Earley algorithm. In Ursula Klenk, editor, *Kontextfreie Syntaxen und verwandte Systeme*, volume 155 of *Linguistische Arbeiten*, pages 76–89. Max Niemeyer Verlag, Tübingen, Oct. 1984. The paper concentrates on the various forms items in parsing may assume. The items as proposed by Earley [14] and Shieber [379] derive part of their meaning from the sets they are found in. In traditional chart, Earley and LR parsing these sets are placed between the tokens of the input. The author inserts nodes between the tokens of the input instead, and then introduces a more general, position-independent item, (i, j, A, α, β), with the meaning that the sequence of categories (linguistic term for non-terminals) α spans (=generates) the tokens between nodes i and j, and that if a sequence of categories β is recognized between j and some node k, a category A has been recognized between i and k. An item with $\beta = \varepsilon$ is called "inactive" in this paper; the terms "passive" and "complete" are used elsewhere. These Kilbury items can be interpreted both as edges in chart parsing and as items in Earley parsing. The effectiveness of these items is then demonstrated by giving a very elegant formulation of the Earley algorithm.
The various versions of chart parsing and Earley parsing differ in their inference rules only. Traditional chart parsing generates far too many items, due to the absence of a top-down selection mechanism which restricts the items to those that can lead back to the start symbol. The paper shows that Earley parsing also generates too many items, since its (top-down) predictor generates many items that can never match the input. The author then proposes a new predictor, which operates bottom-up, and predicts only items that can start with the next token in the input or with a non-terminal that has just resulted from a reduction. The algorithm is restricted to ε-free grammars only, so the completer and predictor need not be repeated. Consequently, the non-terminals introduced by the predictor do not enter the predictor again, and so the predictor predicts the non-terminal of the next-higher level only. Basically it refrains from generating items that would be rejected by the next input token anyway. This reduces the number of generated items considerably (but now we are missing the top-down restriction). Again a very elegant algorithm results.

25. **Kay**, Martin. Algorithm schemata and data structures in syntactic processing. In B.J. Grosz, K. Sparck Jones, and B.L. Webber, editors, *Readings in Natural Language Processing*, pages 35–70. Morgan Kaufmann, 1986. In this reprint of 1980 Xerox PARC

Technical Report CSL-80-12, the author develops a general CF text generation and parsing theory for linguistics, based (implicitly) on unfinished parse trees in which there is an "upper symbol" (top category, predicted non-terminal) α and a "lower symbol" β, the first text symbol corresponding to α; and (explicitly) on a rule selection table S in which the entry $S_{\alpha,\beta}$ contains (some) rules $A \rightarrow \gamma$ such that $A \in FIRST_{ALL}(\alpha)$ and $\beta \in FIRST(\gamma)$, i.e., the rules that can connect α to β. The table can be used in production and parsing; top-down, bottom-up and middle-out; and with or without look-ahead (called "directed" and "undirected" in this paper). By pruning rules from this table, specific parsing techniques can be selected.
To avoid duplication of effort, the parsing process is implemented using charts (Kay [16]). The actions on the chart can be performed in any order consistent with available data, and are managed in a queue called the "agenda". Breadth-first and depth-first processing orders are considered.

26. **Cohen**, Jacques and Hickey, Timothy J. Parsing and compiling using Prolog. *ACM Trans. Prog. Lang. Syst.*, 9(2):125–164, April 1987. Several methods are given to convert grammar rules into Prolog clauses. In the bottom-up method, a rule `E→E+T` corresponds to a clause `reduce([n(t),t(+),n(e)|X],[n(e)|X])` where the parameters represent the stack before and after the reduction. In the top-down method, a rule `T'→×FT'` corresponds to a clause `rule(n(t'),[t(*),n(f),n(t')]).` A recursive descent parser is obtained by representing a rule $S \rightarrow aSb$ by the clause `s(ASB) :- append(A,SB,ASB), append(S,B,SB), a(A), s(S), b(B).` which attempts to cut the input list ASB into three pieces A, S and B, which can each be recognized as an a, an s and a b, respectively. A fourth type of parser results if ranges in the input list are used as parameters: $s(X1,X4) :- link(X1,a,X2), s(X2,X3), link(X3,b,X4)$ in which $link(P,x,Q)$ describes that the input contains the token x between positions P and Q. For each of these methods, ways are given to limit non-determinism and backtracking, resulting among others in LL(1) parsers.
By supplying additional parameters to clauses, context conditions can be constructed and carried around, much as in a VW grammar (although this term is not used). It should be noted that the resulting Prolog programs are actually not parsers at all: they are just logic systems that connect input strings to parsings. Consequently they can be driven both ways: supply a string and it will produce the parsing; supply a parsing and it will produce the string; supply nothing and it will produce all strings with their parsings in the language.
See also same paper [341].

27. **Wirén**, Mats. A comparison of rule-invocation strategies in context-free chart parsing. In *Third Conference of the European Chapter of the Association for Computational Linguistics*, pages 226–233, April 1987. Eight chart parsing predictors are discussed and their effects measured and analysed, 2 top-down predictors and 6 bottom-up (actually left-corner) ones. The general top-down predictor acts when an active edge for a non-terminal A is added at a certain node; it then adds empty active edges for all first non-terminals in the right-hand sides of A, avoiding duplicates. The general left-corner predictor acts when an inactive (completed) edge for a non-terminal A is added at a certain node; it then adds empty active edges for non-terminals that have A as their first non-terminal in their right-hand sides.
Both can be improved by 1. making sure that the added edge has a chance of leading to completion (selectivity); 2. incorporating immediately the non-terminal just recognized (Kilbury); 3. filtering in top-down information. In all tests the selective top-down-filtered Kilbury left-corner predictor clearly outperformed the others.

28. **Rus**, Teodor. Parsing languages by pattern matching. *IEEE Trans. Softw. Eng.*, 14(4):498–511, April 1988. Considers "algebraic grammars" only: there is at least one terminal between each pair of non-terminals in any right-hand side. The rules of the grammar are ordered in "layers", each layer containing only rules whose right-hand sides contain only non-terminals defined in the same or lower layers. On the basis of these layers, very simple contexts are computed for each right-hand side, resulting in an ordered set of patterns. The input is then parsed by repeated application of the patterns in each layer, starting with the bottom one, using fast string matching.
All this is embedded in a system that simultaneously manages abstract semantics. Difficult to read due to unusual terminology.

29. **Kruseman Aretz**, F. E. J. A new approach to Earley's parsing algorithm. *Sci. Comput. Progr.*, 12(2):105–121, July 1989. Starting point is a CYK table filled with Earley items, i.e., a tabular implementation of the Earley algorithm. Rather than implementing the table, two arrays are used, both indexed with the position i in the input. The elements of the first array, D_i, are the mappings from a non-terminal X to the set of all Earley items that have the dot in front of X at position i. The elements of the second array, E_i, are the sets of all reduce items at i. Considerable math is used to derive recurrence relations between the two, leading to a very efficient evaluation order. The data structures are then extended to produce parse trees. Full implementations are given.

30. **Voisin**, Frédéric. and Raoult, Jean-Claude. A new, bottom-up, general parsing algorithm. *BIGRE Bulletin*, 70:221–235, Sept. 1990. Modifies Earley's algorithm by 1. maintaining items of the form α rather than $A \rightarrow \alpha B \bullet \beta$, which eliminates the top-down component and thus the predictive power, and 2. restoring some of that predictive power by predicting items α for each rule in the grammar $A \rightarrow \alpha\beta$ for which the input token is in FIRST(A). This is useful for the special application the authors have, a parser for a language with extensive user-definable operators.

31. **Lang**, Bernard. Towards a uniform formal framework for parsing. In Masaru Tomita, editor, *Current Issues in Parsing Technology*, pages 153–171. Kluwer Academic Publ., Boston, 1991. The paper consists of two disjoint papers. The first concerns the equivalence of grammars and parse forests; the second presents a Logical PushDown Automaton.
In tree-sharing parsers, parsing an (ambiguous) sentence S yields a parse forest. If we label each node in this forest with a unique name, we can consider each node to be a rule in a CF grammar. The node labeled N describes one alternative for the non-terminal N and if p outgoing arrows leave the node, N has p alternatives. This grammar produces exactly one sentence, S. If S contains wild-card tokens, the grammar will produce all sentences in the original grammar that match S. In fact, if we parse the sentence Σ^* in which Σ matches any token, we get a parse forest that is equivalent to the original grammar. Parsing of unambiguous, ambiguous and incomplete sentences alike is viewed as constructing a grammar that produces exactly the singleton, multi-set and infinite set of derivations that produce the members of the input set. No such parsing algorithm is given, but the reader of the paper is referred to Billot and Lang [164], and to Lang [210].
Prolog-like programs can be seen as grammars the non-terminals of which are predicates with arguments, i.e. Horn clauses. Such programs are written as Definite Clause programs. To operate these programs as solution-producing grammars, a Logical PushDown Automaton LPDA is introduced, which uses Earley deduction in a technique similar to that of Pereira and Warren [368]. In this way, a deduction mechanism is obtained that is shown to terminate on a Definite Clause Grammar on which simple depth-first resolution would loop. The conversion from DC program to a set of Floyd-like productions for the LPDA is described in full, and so is the LPDA itself.

32. **Leo**, Joop M. I. M. A general context-free parsing algorithm running in linear time on every LR(k) grammar without using lookahead. *Theoret. Comput. Sci.*, 82:165–176, 1991. Earley parsing of right-recursive LR(k) grammars will need time and space of $O(n^2)$, for the build-up of the final reductions. This build-up is prevented through the introduction of "transitive items", which store right-recursion information. A proof is given that the resulting algorithm is linear in time and space for every LR-regular grammar. The algorithm also defends itself against hidden right recursion.

33. **Nederhof**, M.-J. An optimal tabular parsing algorithm. In *32nd Annual Meeting of the Association for Computational Linguistics*, pages 117–124, June 1994. Like chart parsing, the various LR parsing methods can be characterized by the way they infer new items from old ones. In this paper, four such characterizations are given: for LC parsing, for predictive LR, for regular right part grammars, called "extended LR (ELR) grammars here, and for "Common-Prefix" parsing. For Common-Prefix see Voisin and Raoult [30]. Each of these is then expressed as a tabular parsing algorithm, and their properties are compared. ELR appears the best compromise for power and efficiency.

34. **Rytter**, W. Context-free recognition via shortest-path computation: A version of Valiant's algorithm. *Theoret. Comput. Sci.*, 143(2):343–352, 1995. The multiplication

and addition in the formula for distance in a weighted graph are redefined so that a shortest distance through a specific weighted lattice derived from the grammar and the input corresponds to a parsing. This allows advances in shortest-path computation (e.g. parallelization) to be exploited for parsing. The paper includes a proof that the algorithm is formally equivalent to Valiant's.

35. **McLean**, Philippe and Horspool, R. Nigel. A faster Earley parser. In Tibor Gyimóthy, editor, *Compiler Construction: 6th International Conference, CC'96*, volume 1060 of *Lecture Notes in Computer Science*, pages 281–293, New York, 1996. Springer-Verlag. The items in an Earley set can be grouped into subsets, such that each subset corresponds to an LR state. This is utilized to speed up the Earley algorithm. Speed-up factors of 10 to 15 are obtained, and memory usage is reduced by half.

36. **Johnstone**, Adrian and Scott, Elizabeth. Generalized recursive-descent parsing and follow-determinism. In Kai Koskimies, editor, *Compiler Construction (CC'98)*, volume 1383 of *Lecture Notes in Computer Science*, pages 16–30, Lisbon, 1998. Springer. The routine generated for a non-terminal A returns a set of lengths of input segments starting at the current position and matching A, rather than just a Boolean saying match or no match. This gives a parser that is efficient on LL(1) and non-left-recursive LR(1). Next it is made more efficient by using FIRST sets. This parser is implemented under the name *GRDP*, for "Generalised Recursive Descent Parser". It yields all parses; but can be asked to act deterministically. It then uses *FOLLOW-determinism*, in which the length is chosen whose segment is followed by a token from $\text{FOLLOW}_1(A)$; the grammar must be such that only one length qualifies.

37. **Aycock**, John and Horspool, R. Nigel. Directly-executable Earley parsing. In *Compiler Construction*, volume 2027 of *Lecture Notes in Computer Science*, pages 229–243, 2001. Code segments are generated for all actions possible on each possible Earley item, and these segments are linked together into an Earley parser using a threaded-code technique, but parent pointer manipulations are cumbersome. To remedy this, the items are grouped in the states of a split LR(0) automaton, in which each traditional LR(0) state is split in two states, one containing the items in which the dot is at the beginning (the "non-kernel" state), and one which contains the rest. The parent pointers of the non-kernel states are all equal, which simplifies implementation. The resulting parser is 2 to 3 times faster than a standard implementation of the Earley parser.

38. **Aycock**, John and Horspool, R. Nigel. Practical Earley parsing. *Computer J.*, 45(6):620–630, 2002. Empty productions are the reason for the Predictor/Completer loop in an Earley parser, but the loop can be avoided by having the Predictor also predict items of the form $A \rightarrow \alpha B \bullet \beta$ for $\bullet B$ if B is nullable. Effectively the ε is propagated by the Predictor. The nullable non-terminals are found by preprocessing the grammar. The Earley item sets are collected in an "LR(0)ε" automaton. The states of this automaton are then split as described by Aycock and Horspool [37]. A third transformation is required to allow convenient reconstruction of the parse trees.

39. **Lee**, Lillian. Fast context-free grammar parsing requires fast Boolean matrix multiplication. *J. ACM*, 49(1):1–15, 2002. The author proves that if we can do parsing in $O(n^{3-\varepsilon})$, we can do Boolean matrix multiplication in $O(n^{3-\varepsilon/3})$. To convert a given matrix multiplication of A and B into a parsing problem we start with a string w of a length that depends only on the size of the matrices; all tokens in w are different. For each non-zero element of A we create a new non-terminal $A_{i,j} \rightarrow w_p W w_q$, where w_p and w_q are judiciously chosen tokens from w and W produces any non-empty string of tokens from w; likewise for B. The resulting matrix C is grammaticalized by rules $C_{p,q} \rightarrow A_{p,r} B_{r,q}$, which implements the (Boolean) multiplication. Occurrences of $C_{p,q}$ are now attempted to be recognized in w by having start symbol rules $S \rightarrow W C_{p,q} W$. The resulting grammar is highly ambiguous, and when we parse w with it, we obtain a parse forest. If the node for $C_{p,q}$ is present in it, the bit $C_{p,q}$ in the resulting matrix is on.

40. **Nederhof**, Mark-Jan and Satta, Giorgio. Tabular parsing. In *Formal Languages and Applications*, volume 148 of *Studies in Fuzziness and Soft Computing*, pages 529–549. Springer, April 2004. Tutorial on tabular parsing for push-down automata, Earley parsers, CYK and non-deterministic LR parsing. The construction of parse trees is also covered.

18.1.3 LL Parsing

41. **Lucas**, P. Die Strukturanalyse von Formelnübersetzern / analysis of the structure of formula translators. *Elektronische Rechenanlagen*, 3(11.4):159–167, 1961, (in German).
Carefully reasoned derivation of a parser and translator from the corresponding grammar. Two types of "syntactic variable definitions" (= grammar rules) are distinguished: "enumerating definitions", of the form $N \rightarrow A_1|A_2|\cdots$, and "concatenating definitions", of the form $N \rightarrow A_1A_2\cdots$; here N is a non-terminal and the A_i are all terminals or non-terminals. Additionally "combined definitions" are allowed, but they are tacitly decomposed into enumerating and concatenating definitions when the need arises. Each "syntactic variable" (= non-terminal) can be "explicitly defined", in which case the definition does not refer to itself, or "recursively defined", in which case it does.
For each non-terminal N two pieces of code are created: an identification routine and a translation routine. The identification routine tests if the next input token can start a terminal production of N, and the translation routine parses and translates it. The identification routines are produced by inspecting the grammar; the translation routines are created from templates, as follows. The translation routine for an enumerating definition is

```
if can.start.A1 then translate.A1 else
if can.start.A2 then translate.A2 else
... else report.error
```

The translation routine for a concatenating definition is

```
if can.start.A1 then translate.A1 else report.error;
if can.start.A2 then translate.A2 else report.error;
...
```

Each translation routine can have local variables and produces an output parameter, which can be used for code generation by the caller; all these variables are allocated statically (as global memory locations). These routines are given by flowcharts, although the author recognizes that they could be expressed in ALGOL 60.
The flowcharts are connected into one big flow chart, except that a translation routine for a recursive non-terminal starts with code that stores its previous return address and local variables on a stack, and ends with code that restores them. Since the number of local variables vary from routine to routine, the stack entries are of unequal size; such stack entries are called "drawers". No hint is given that the recursion of ALGOL 60 could be used for these purposes. Special flowchart templates are given for directly left- and right-recursive non-terminals, which transform the recursion into iteration.
For the system to work the grammar must obey requirements that are similar to LL(1), although the special treatment of direct left recursion alleviates them somewhat. These requirements are mentioned but not analysed. Nothing is said about nullable non-terminals.
The method is compared to the PDA technique of Samelson and Bauer, [112]. The two methods are recognized as equivalent, but the method presented here lends itself better for hand optimization and code insertion.
This is probably the first description of recursive descent parsing. The author states that four papers explaining similar techniques appeared after the paper was written but before it was printed. That depends on the exact definition of recursive descent: of the four only Grau [332] shows how to generate code for the routines, i.e., to use compiled recursive descent. The others interpret the grammar.

42. **Kurki-Suonio**, R. Notes on top-down languages. *BIT*, 9:225–238, 1969. Gives several variants of the LL(k) condition. Also demonstrates the existence of an LL(k) language which is not LL($k-1$).

43. **Knuth**, Donald E. Top-down syntax analysis. *Acta Inform.*, 1:79–110, 1971. A *Parsing Machine* (PM) is defined, which is effectively a set of mutually recursive Boolean functions which absorb input if they succeed and absorb nothing if they fail. Properties of the languages accepted by PMs are examined. This leads to CF grammars, dependency graphs, the null string problem, back-up, LL(k), follow function, LL(1), s-languages and a comparison of top-down versus bottom-up parsing. The author is one of the few scientists who provides insight in their thinking process.

44. **Jarzabek**, S. and Krawczyk, T. LL-regular grammars. *Inform. Process. Lett.*, 4(2):31–37, 1975. Introduces LL-regular (*LLR*) grammars: for every rule $A \rightarrow \alpha_1|\cdots|\alpha_n$, a partition $(R_1,\ldots,R_n)$ of disjoint regular sets must be given such that the rest of the input sentence is a member of exactly one of these sets. A parser can then be constructed by creating finite-state automata for these sets, and letting these finite state automata determine the next prediction.

45. **Nijholt**, A. On the parsing of LL-regular grammars. In A. Mazurkiewicz, editor, *Mathematical Foundations of Computer Science*, volume 45 of *Lecture Notes in Computer Science*, pages 446–452. Springer-Verlag, Berlin, 1976. Derives a parsing algorithm for LL-regular grammars with a regular pre-scan from right to left that leaves markers, and a subsequent scan which consists of an LL(1)-like parser.

46. **Lewi**, J., Vlaminck, K. de, Huens, J., and Huybrechts, M. The ELL(1) parser generator and the error-recovery mechanism. *Acta Inform.*, 10:209–228, 1978. See same paper [298].

47. **Poplawski**, D. A. On LL-regular grammars. *J. Comput. Syst. Sci.*, 18:218–227, 1979. Presents proof that, given a regular partition, it is decidable whether a grammar is LL-regular with respect to this partition; it is undecidable whether or not such a regular partition exists. The paper then discusses a two-pass parser; the first pass works from right to left, marking each terminal with an indication of the partition that the rest of the sentence belongs to. The second pass then uses these indications for its predictions.

48. **Nijholt**, A. LL-regular grammars. *Intern. J. Comput. Math.*, A8:303–318, 1980. This paper discusses strong-LL-regular grammars, which are a subset of the LL-regular grammars, exactly as the strong-LL(k) grammars are a subset of the LL(k) grammars, and derives some properties.

49. **Heckmann**, Reinhold. An efficient ELL(1)-parser generator. *Acta Inform.*, 23:127–148, 1986. The problem of parsing with an ELL(1) grammar is reduced to finding various FIRST and FOLLOW sets. Theorems about these sets are derived and very efficient algorithms for their computation are supplied.

50. **Barnard**, David T. and Cordy, James R. Automatically generating SL parsers from LL(1) grammars. *Comput. Lang.*, 14(2):93–98, 1989. For SL see Barnard and Cordy [265]. SL seems ideally suited for implementing LL(1) parsers, were it not that the choice action absorbs the input token on which the choice is made. This effectively prevents look-ahead, and means that when a routine for a non-terminal A is called, its first token has already been absorbed. A scheme is suggested that will replace the routine for parsing A by a routine for parsing A minus its first token. So the technique converts the grammar to simple LL(1).

51. **Parr**, Terence J. and Quong, Russell W. LL and LR translators need $k > 1$ lookahead. *ACM SIGPLAN Notices*, 31(2):27–34, Feb. 1996. Gives realistic examples of frequent programming language constructs in which $k = 1$ fails. Since $k > 1$ is very expensive, the authors introduce linear-approximate LL(k) with $k > 1$, in which for each look-ahead situation S separate values FIRST(S), SECOND(S), ..., are computed, which needs $t \times k$ space for t equal to the number of different tokens, rather than $\text{FIRST}_k(S)$, which requires t^k. This may weaken the parser since originally differing look-ahead sets like ab, cd and ad, cb both collapse to $[ac][bd]$, but usually works out OK.

18.1.4 LR Parsing

52. **Knuth**, D. E. On the translation of languages from left to right. *Inform. Control*, 8:607–639, 1965. This is the original paper on LR(k). It defines the notion as an abstract property of a grammar and gives two tests for LR(k). The first works by constructing for the grammar a regular grammar which generates all possible already reduced parts (= stacks) plus their look-aheads; if this grammar has the property that none of its words is a prefix to another of its words, the original grammar was LR(k). The second consists of implicitly constructing all possible item sets (= states)

and testing for conflicts. Since none of this is intended to yield a reasonable parsing algorithm, notation and terminology differs from that in later papers on the subject. Several theorems concerning LR(k) grammars are given and proved.

53. **Korenjak**, A. J. A practical method for constructing LR(k) processors. *Commun. ACM*, 12(11):613–623, Nov. 1969. The huge LR(1) parsing table is partitioned as follows. A nonterminal Z is chosen judiciously from the grammar, and two grammars are constructed, G_0, in which Z is considered to be a terminal symbol, and G_1, which is the grammar for Z (i.e. which has Z as the start symbol). If both grammars are LR(1) and moreover a master LR(1) parser can be constructed that controls the switching back and forth between G_0 and G_1, the parser construction succeeds (and the original grammar was LR(1) too). The three resulting tables together are much smaller than the LR(1) table for the original grammar. It is also possible to chose a set of nonterminals $Z_1 \cdots Z_n$ and apply a variant of the above technique.

54. **DeRemer**, Franklin L. Simple LR(k) grammars. *Commun. ACM*, 14(7):453–460, July 1971. SLR(k) explained by its inventor. Several suggestions are made on how to modify the method; use a possibly different k for each state; use possibly different lengths for each look-ahead string. The relation to Korenjak's approach [53] is also discussed.

55. **Anderson**, T. *Syntactic Analysis of LR(k) Languages*. PhD thesis, Technical report, University of Newcastle upon Tyne, Newcastle upon Tyne, 1972. [Note: This is one of the few papers we have not been able to access; the following is the author's abstract.] A method of syntactic analysis, termed LA(m)LR(k), is discussed theoretically. Knuth's LR(k) algorithm [52] is included as the special case $m = k$. A simpler variant, SLA(m)LR(k), is also described, which in the case SLA(k)LR(0) is equivalent to the SLR(k) algorithm as defined by DeRemer [54]. Both variants have the LR(k) property of immediate detection of syntactic errors.
The case $m = 1, k = 0$ is examined in detail, when the methods provide a practical parsing technique of greater generality than precedence methods in current use. A formal comparison is made with the weak precedence algorithm.
The implementation of an SLA(1)LR(0) parser (SLR) is described, involving numerous space and time optimizations. Of importance is a technique for bypassing unnecessary steps in a syntactic derivation. Direct comparisons are made, primarily with the simple precedence parser of the highly efficient Stanford ALGOL W compiler, and confirm the practical feasibility of the SLR parser.

56. **Anderson**, T., Eve, J., and Horning, J. J. Efficient LR(1) parsers. *Acta Inform.*, 2:12–39, 1973. Coherent explanation of SLR(1), LALR(1), elimination of unit rules and table compression, with good advice.

57. **Čulik**, II, Karel and Cohen, Rina. LR-regular grammars: An extension of LR(k) grammars. *J. Comput. Syst. Sci.*, 7:66–96, 1973. The input is scanned from right to left by a FS automaton which records its state at each position. Next this sequence of states is parsed from left to right using an LR(0) parser. If such a FS automaton and LR(0) parser exist, the grammar is LR-regular. The authors conjecture, however, that it is unsolvable to construct this automaton and parser. Examples are given of cases in which the problem can be solved.

58. **LaLonde**, Wilf R. Regular right part grammars and their parsers. *Commun. ACM*, 20(10):731–741, Oct. 1977. The notion of regular right part grammars and its advantages are described in detail. A parser is proposed that does LR(k) parsing to find the right end of the handle and then, using different parts of the same table, scans the stack backwards using a look-ahead (to the left!) of m symbols to find the left end; this is called *LR(m,k)*. The corresponding parse table construction algorithm is given by LaLonde [59].

59. **LaLonde**, W. R. Constructing LR parsers for regular right part grammars. *Acta Inform.*, 11:177–193, 1979. Describes the algorithms for the regular right part parsing technique explained by LaLonde [58]. The back scan is performed using so-called *read-back tables*. Compression techniques for these tables are given.

60. **Baker**, Theodore P. Extending look-ahead for LR parsers. *J. Comput. Syst. Sci.*, 22(2):243–259, 1981. A FS automaton is derived from the LR automaton as follows: upon a

reduce to A the automaton moves to all states that have an incoming arc marked A. This automaton is used for analysing the look-ahead as in an LR-regular parser (Čulik, II and Cohen [57]).

61. **Kristensen**, Bent Bruun and Madsen, Ole Lehrmann. Methods for computing LALR(k) lookahead. *ACM Trans. Prog. Lang. Syst.*, 3(1):60–82, Jan. 1981. The LALR(k) look-ahead sets are seen as the solution to a set of equations, which are solved by recursive traversal of the LR(0) automaton. Full algorithms plus proofs are given.

62. **LaLonde**, Wilf R. The construction of stack-controlling LR parsers for regular right part grammars. *ACM Trans. Prog. Lang. Syst.*, 3(2):168–206, April 1981. Traditional LR parsers shift each input token onto the stack; often, this shift could be replaced by a state transition, indicating that the shift has taken place. Such a parser is called a *stack-controlling LR parser*, and will do finite-state recognition without stack manipulation whenever possible. Algorithms for the construction of stack-controlling LR parse tables are given. The paper is complicated by the fact that the new feature is introduced not in a traditional LR parser, but in an LR parser for regular right parts (for which see LaLonde [58]).

63. **DeRemer**, Frank L. and Pennello, Thomas J. Efficient computation of LALR(1) look-ahead sets. *ACM Trans. Prog. Lang. Syst.*, 4(4):615–649, Oct. 1982. Rather than starting from an LR(1) automaton and collapsing it to obtain an LALR(1) automaton, the authors start from an LR(0) automaton and compute the LALR(1) look-aheads from there, taking into account that look-aheads are meaningful for reduce items only. For each reduce item $A \rightarrow \alpha\bullet$ we search back in the LR(0) automaton to find all places P where it could originate from, for each of these places we find the places Q that can be reached by a shift over A from P, and from each of these places we look forward in the LR(0) automaton to determine what the next token in the input could be. The set of all these tokens is the LALR(1) look-ahead set of the original reduce item. The process is complicated by the presence of ε-productions.
The computation is performed by four linear sweeps over the LR(0) automaton, set up so that they can be implemented by transitive closure algorithms based on strongly connected components, which are very efficient.
Care must be taken to perform the above computations in the right order; otherwise look-ahead sets may be combined too early resulting in "Not Quite LALR(1)", NQLALR(1), which is shown to be inadequate.
The debugging of non-LALR(1) grammars is also treated.

64. **Heilbrunner**, S. Truly prefix-correct chain-free LR(1) parsers. *Acta Inform.*, 22(5):499–536, 1985. A unit-free LR(1) parser generator algorithm, rigorously proven correct.

65. **Park**, Joseph C. H., Choe, K.-M., and Chang, C.-H. A new analysis of LALR formalisms. *ACM Trans. Prog. Lang. Syst.*, 7(1):159–175, Jan. 1985. The recursive closure operator *CLOSURE* of Kristensen and Madsen [61] is abstracted to an iterative δ-operator such that $CLOSURE \equiv \delta^*$. This operator allows the formal derivation of four algorithms for the construction of LALR look-ahead sets, including an improved version of the relations algorithm of DeRemer and Pennello [63]. See Park and Choe [73] for an update.

66. **Ukkonen**, Esko. Upper bounds on the size of LR(k) parsers. *Inform. Process. Lett.*, 20(2):99–105, Feb. 1985. Upper bounds for the number of states of an LR(k) parser are given for several types of grammars.

67. **Al-Hussaini**, A. M. M. and Stone, R. G. Yet another storage technique for LR parsing tables. *Softw. Pract. Exper.*, 16(4):389–401, 1986. Excellent introduction to LR table compression in general. The *submatrix technique* introduced in this paper partitions the rows into a number of submatrices, the rows of each of which are similar enough to allow drastic compressing. The access cost is $O(1)$. A heuristic partitioning algorithm is given.

68. **Ives**, Fred. Unifying view of recent LALR(1) lookahead set algorithms. *ACM SIGPLAN Notices*, 21(7):131–135, July 1986. A common formalism is given in which the LALR(1) look-ahead set construction algorithms of DeRemer and Pennello [63], Park, Choe and Chang [65] and the author can be expressed. See also Park and Choe [73].

69. **Nakata**, Ikuo and Sassa, Masataka. Generation of efficient LALR parsers for regular right part grammars. *Acta Inform.*, 23:149–162, 1986. The stack of an LALR(1) parser is augmented with a set of special markers that indicate the start of a right-hand side; adding such a marker during the shift is called a *stack shift*. Consequently there can now be a shift/stack-shift conflict, abbreviated to *stacking conflict*. The stack-shift is given preference and any superfluous markers are eliminated during the reduction. Full algorithms are given.

70. **Pennello**, Thomas J. Very fast LR parsing. *ACM SIGPLAN Notices*, 21(7):145–151, July 1986. The tables and driver of a traditional LALR(1) parser are replaced by assembler code performing linear search for small fan-out, binary search for medium and a computed jump for large fan-out. This modification gained a factor of 6 in speed at the expense of a factor 2 in size.

71. **Chapman**, Nigel P. *LR Parsing: Theory and Practice*. Cambridge University Press, New York, NY, 1987. Detailed treatment of the title subject. Highly recommended for anybody who wants to acquire in-depth knowledge about LR parsing. Good on size of parse tables and attribute grammars.

72. **Ives**, Fred. Response to remarks on recent algorithms for LALR lookahead sets. *ACM SIGPLAN Notices*, 22(8):99–104, Aug. 1987. Remarks by Park and Choe [73] are refuted and a new algorithm is presented that is significantly better than that of Park, Choe and Chang [65] and that previously presented by Ives [68].

73. **Park**, Joseph C. H. and Choe, Kwang-Moo. Remarks on recent algorithms for LALR lookahead sets. *ACM SIGPLAN Notices*, 22(4):30–32, April 1987. Careful analysis of the differences between the algorithms of Park, Choe and Chang [65] and Ives [68]. See also Ives [72].

74. **Sassa**, Masataka and Nakata, Ikuo. A simple realization of LR-parsers for regular right part grammars. *Inform. Process. Lett.*, 24(2):113–120, Jan. 1987. For each item in each state on the parse stack of an LR parser, a counter is kept indicating how many preceding symbols on the stack are covered by the recognized part in the item. Upon reduction, the counter of the reducing item tells us how many symbols to unstack. The manipulation rules for the counters are simple. The counters are stored in short arrays, one array for each state on the stack.

75. **Bermudez**, Manuel E. and Schimpf, Karl M. On the (non-)relationship between SLR(1) and NQLALR(1) grammars. *ACM Trans. Prog. Lang. Syst.*, 10(2):338–342, April 1988. Shows a grammar that is SLR(1) but not NQLALR(1).

76. **Bermudez**, Manuel E. and Schimpf, Karl M. A general model for fixed look-ahead LR parsers. *Intern. J. Comput. Math.*, 24(3+4):237–271, 1988. Extends the DeRemer and Pennello [63] algorithm to LALR(k), NQLALR(k) and SLR(k). Also defines NQSLR(k), Not-Quite SLR, in which a too simple definition of FOLLOW_k is used. The difference only shows up for $k \geq 2$, and is similar to the difference in look-ahead between full-LL(k) and strong-LL(k). Suppose for $k = 2$ we have the grammar **S→ApBq**, **S→Br**, **A→a**, **B→**ε, and we compute FOLLOW_2(**A**). Then the NQSLR algorithm computes it as FIRST_1(**B**) plus FOLLOW_1(**B**) if **B** produces ε. Since FOLLOW_1(**B**)={**q**,**r**}, this yields the set {**pq**,**pr**}; but the sequence **pr** cannot occur in any input. The authors give an example where such an unjustified look-ahead prevents parser construction.

77. **Kruseman Aretz**, F. E. J. On a recursive ascent parser. *Inform. Process. Lett.*, 29(4):201–206, Nov. 1988. Each state in an LR automaton is implemented as a subroutine. A shift calls that subroutine. A reduce to X is effected as follows. X and its length n are stored in global variables; all subroutines are rigged to decrement n and return as long as $n > 0$, and to call the proper GOTO state of X when n hits 0. This avoids the explicit stack manipulation of Roberts [78].

78. **Roberts**, George H. Recursive ascent: An LR analog to recursive descent. *ACM SIGPLAN Notices*, 23(8):23–29, Aug. 1988. Each LR state is represented by a subroutine. The shift is implemented as a subroutine call; the reduction is followed by a subroutine return possibly preceded by a return stack adjustment. The latter prevents the generation of genuine subroutines

since it requires explicit return stack manipulation. A small and more or less readable LR(0) parser is shown, in which conflicts are resolved by means of the order in which certain tests are done, like in a recursive descent parser.

79. **Bermudez**, Manuel E. and Logothetis, George. Simple computation of LALR(1) look-ahead sets. *Inform. Process. Lett.*, 31(5):233–238, 1989. The original LALR(1) grammar is replaced by a not much bigger grammar that has been made to incorporate the necessary state splitting through a simple transformation. The SLR(1) automaton of this grammar is the LALR(1) automaton of the original grammar.

80. **Horspool**, R. Nigel. ILALR: An incremental generator of LALR(1) parsers. In D. Hammer, editor, *Compiler Compilers and High-Speed Compilation*, volume 371 of *Lecture Notes in Computer Science*, pages 128–136. Springer-Verlag, Berlin, 1989. Grammar rules are checked as they are typed in. To this end, LALR(1) parse tables are kept and continually updated. When the user interactively adds a new rule, the sets FIRST and NULLABLE are recomputed and algorithms are given to distribute the consequences of possible changes over the LR(0) and look-ahead sets. Some serious problems are reported and practical solutions are given.

81. **Roberts**, George H. Another note on recursive ascent. *Inform. Process. Lett.*, 32(5):263–266, 1989. The fast parsing methods of Pennello [70], Kruseman Aretz [77] and Roberts [78] are compared. A special-purpose optimizing compiler can select the appropriate technique for each state.

82. **Bermudez**, Manuel E. and Schimpf, Karl M. Practical arbitrary lookahead LR parsing. *J. Comput. Syst. Sci.*, 41(2):230–250, Oct. 1990. Refines the extended-LR parser of Baker [60] by constructing a FS automaton for each conflict state q as follows. Starting from q and looking backwards in the LR(0) automaton, all top-of-stack segments of length m are constructed that have q on the top. These segments define a regular language R which is a superset of the possible continuations of the input (which are determined by the entire stack). Also each decision made by the LR(0) automaton to resolve the conflict in q defines a regular language, each a subset of R. If these languages are disjunct, we can decide which decision to take by scanning ahead. Scanning ahead is done using an automaton derived from q and the LR(0) automaton. Grammars for which parsers can be constructed by this technique are called LAR(m). The technique can handle some non-LR(k) grammars.

83. **Heering**, J., Klint, P., and Rekers, J. Incremental generation of parsers. *IEEE Trans. Softw. Eng.*, 16(12):1344–1351, 1990. In a very unconventional approach to parser generation, the initial information for an LR(0) parser consists of the grammar only. As parsing progresses, more and more entries of the LR(0) table (actually a graph) become required and are constructed on the fly. LR(0) inadequacies are resolved using GLR parsing. All this greatly facilitates handling (dynamic) changes to the grammar.

84. **Horspool**, R. N. Incremental generation of LR parsers. *Comput. Lang.*, 15(4):205–233, 1990. The principles and usefulness of incremental parser generation are argued. The LR parse table construction process is started with initial items $S_S \rightarrow \bullet\#_N N\#_N$ for each non-terminal N in the grammar, where the $\#_N$s are N-specific delimiters. The result is a parse table in which any non-terminal can be the start symbol. When N is modified, the item $S_S \rightarrow \bullet\#_N N\#_N$ is followed throughout the parser, updates are made, and the table is cleaned of unreachable items. Deletion is handled similarly. The algorithms are outlined.
A proof is given that modifying a single rule in a grammar of n rules can cause $O(2^n)$ items to be modified, so the process is fundamentally exponential. In practice, it turns out that incremental recomputing is in the order of 10 times cheaper than complete recomputing.

85. **Horspool**, R. Nigel and Whitney, Michael. Even faster LR parsing. *Softw. Pract. Exper.*, 20(6):515–535, June 1990. Generates directly executable code. Starts from naive code and then applies a series of optimizations: 1. Push non-terminals only; this causes some trouble in deciding what to pop upon and reduce and it may even introduce a pop-count conflict. 2. Combine the reduce

and the subsequent goto into one optimized piece of code. 3. Turn right recursion into left recursion to avoid stacking and generate code to undo the damage. 4. Eliminate unit rules.

86. **McKenzie**, B. J. LR parsing of CFGs with restrictions. *Softw. Pract. Exper.*, 20(8):823–832, Aug. 1990. It is often useful to specify semantic restrictions at certain points in the right-hand sides of grammar rules; these restrictions can serve to check the semantic correctness of the input and/or to disambiguate the grammar. Conceptually these restrictions are associated with marker non-terminals, which produce ε and upon reduction test the restriction. This causes lots of conflicts in the LR parser; rather than have the LR parser generator solve them in the usual fashion, they are solved at parse time by calling the restriction-testing routines. If no test routine succeeds, there is an error in the input; if one succeeds, the parser knows what to do; and if more than one succeeds, there is a grammar error, which can be dealt with by having a default (use the textually first restriction, for example), or by giving an error message. Many examples, no explicit code. It would seem the system can also be used to implement dynamic conflict resolvers.

87. **Roberts**, George H. From recursive ascent to recursive descent: via compiler optimizations. *ACM SIGPLAN Notices*, 25(4):83–89, April 1990. Shows a number of code transformations that will turn an LR(1) recursive ascent parser (see Roberts [78, 81]) for an LL(1) grammar into a recursive descent parser.

88. **Charles**, Phillippe. *A Practical Method for Constructing Efficient LALR(k) Parsers with Automatic Error Recovery*. PhD thesis, Technical report, NYU, Feb. 1991. Addresses various issues in LALR parsing: 1. Gives an in-depth overview of LALR parsing algorithms. 2. Modifies DeRemer and Pennello's algorithm [63] to adapt the length of the lookaheads to the needs of the states. 3. Gives an improved version of Burke and Fisher' automatic LR error recovery mechanism [317], for which see [319]. 4. Existing table compression methods are tuned to LALR tables. Explicit algorithms are given.

89. **Fortes Gálvez**, José. Generating LR(1) parsers of small size. In U. Kastens and P. Pfahler, editors, *Compiler Construction, 4th International Conference, CC'92*, volume 641 of *Lecture Notes in Computer Science*, pages 16–29. Springer-Verlag, Oct. 1992. Actually, reverse LR(1) parsers are constructed, as follows. The stack is the same as for the normal LR(1) parser, except that no states are recorded, so the stack consists of non-terminals and terminals only. When faced with the problem of whether to shift or to reduce, the stack is analysed from the top downward, rather than from the bottom upward. Since the top region of the stack contains more immediately relevant information than the bottom region, the above analysis will usually come up with an answer pretty quickly.
The analysis can be done using an FSA, starting with the look-ahead token. An algorithm to construct this FSA is described informally, and a proof is given that it has the full LR(1) parsing power. The resulting automaton is about 1/3 the size of the *yacc* automaton, so it is even smaller than the LALR(1) automaton.

90. **Shin**, Heung-Chul and Choe, Kwang-Moo. An improved LALR(*k*) parser generation for regular right part grammars. *Inform. Process. Lett.*, 47(3):123–129, 1993. Improves the algorithm of Nakata and Sassa [69] by restricting the algorithm to kernel items only.

91. **Fortes Gálvez**, José. A note on a proposed LALR parser for extended context-free grammars. *Inform. Process. Lett.*, 50(6):303–305, June 1994. Shows that the algorithm of Shin and Choe [90] is incorrect by giving a counterexample.

92. **Fortes Gálvez**, José. A practical small LR parser with action decision through minimal stack suffix scanning. In *Developments in Language Theory II*, pages 460–465, Singapore, 1995. World Scientific. Theory of and explicit algorithms for DR parsing.

93. **Seyfarth**, Benjamin R. and Bermudez, Manuel E. Suffix languages in LR parsing. *Intern. J. Comput. Math.*, A-55(3-4):135–154, 1995. An in-depth analysis of the set of strings that can follow a state in a non-deterministic LR(0) automation (= an item in the deter-

ministic one) is given and used to derive all known LR parsing algorithms. Based on first author's thesis.

94. **Nederhof**, Mark-Jan and Sarbo, Janos J. Increasing the applicability of LR parsing. In Harry Bunt and Masaru Tomita, editors, *Recent Advances in Parsing Technology*, pages 35–58. Kluwer Academic Publishers, Dordrecht, 1996. ε-reductions are incorporated in the LR items, resulting in ε-LR parsing. Now the stack contains only non-terminals that correspond to non-empty segments of the input; it may be necessary to examine the stack to find out exactly which reduction to do. ε-LR parsing has two advantages: more grammars are ε-LR than LR; and non-deterministic ε-LR tables will never make the original Tomita algorithm [162] loop, thus providing an alternative way to do GLR parsing on arbitrary CF grammars, in addition to Nozohoor-Farshi's method [167].

95. **Fortes Gálvez**, José. *A Discriminating-Reverse Approach To LR(k) Parsing*. PhD thesis, Technical report, Univesité de Nice-Sophia Antipolis, Nice, France, 1998. Existing parsing techniques are explained and evaluated for convenience and memory use. Several available implementations are also discussed. The convenience of full LR(1), LR(2), etc. parsing with minimal memory use is obtained with DR parsing. The DR(0) and DR(1) versions are discussed in detail, and measurements are provided; theory of DR(k) is given. Algorithms for ambiguous grammars are also presented.

96. **Bertsch**, Eberhard and Nederhof, Mark-Jan. Regular closure of deterministic languages. *SIAM J. Computing*, 29(1):81–102, 1999. A *meta-deterministic language* is a language expressed by a regular expression the elements of which are LR(0) languages. Every LR(k) language is meta-deterministic, i.e., can be formed as a regular sequence of LR(0) languages. Using a refined form of the technique of Bertsch [215], in which the above regular expression plays the role of the root set grammar, the authors show that meta-deterministic languages can be recognized and parsed in linear time. Many proofs, much theory.

97. **Morimoto**, Shin-Ichi and Sassa, Masataka. Yet another generation of LALR parsers for regular right part grammars. *Acta Inform.*, 37(9):671–697, 2000. To allow determining the extent of the handle of a reduce, markers are pushed on the stack whenever a production could start. For most LALR(1) grammars these allow unique identification of the handle segment at reduce time. For other LALR(1) grammars counters are included in the stack. Complicated theory, but extensive examples given.

98. **Farré**, Jacques and Fortes Gálvez, José. A bounded graph-connect construction for LR-regular parsers. In *Compiler Construction: 10th International Conference, CC 2001*, volume 2027 of *Lecture Notes in Computer Science*, pages 244–258. Springer-Verlag, 2001. Detailed description of the constructing of a practical LR-regular parser, consisting of both algorithms and heuristic rules for the development of the look-ahead automata. As an example, such a parser is constructed for a difficult subset of HTML.

99. **Kannapinn**, Sönke. *Eine Rekonstruktion der LR-Theorie zur Elimination von Redundanzen mit Anwendung auf den Bau von ELR-Parsern*. PhD thesis, Technical report, Technische Universität Berlin, Berlin, July 2001, (in German). The thesis consists of two fairly disjunct parts; the first part (100 pages) concerns redundancy in LR parsers, the second (60 pages) designs an LR parser for EBNF, after finding errors in existing publications.
The states in an LR parser hold a lot of redundancy: for example, the top state on the stack is not at all independent of the rest of the stack. This is good for time efficiency but bad for space efficiency. The states in an LR parser serve to answer three questions: 1. whether to shift or to reduce; if to reduce, 2. what rule to use; 3. to what new state to go to after the reduce. In each of these a look-ahead can be taken into account. Any scheme that provides these answers works.
The author proposes various ways to reduce the amount of information carried in the dotted items, and the LR, LALR and SLR states. In each of these cases, the ability to determine the reduce rule suffers, and further stack examination is required to answer question 2 above; this stack examination must be of bounded size, or else the parser is no longer linear-time. Under some of the

modifications, the original power remains, but other classes of grammars also appear: *algemeines LR*, translated by Joachim Durchholz by *Compact LR*, since the literal translation "general LR" is too much like "generalized LR", and ILALR.
In Compact LR, an item $A \rightarrow \alpha \bullet X \beta, t$ in a state of the LR(1) automaton is reduced to $X|u$ where X can be terminal or non-terminal, and u is the immediate look-ahead of X, i.e. the first token of β if it exists, or t if β is absent. The resulting *CLR(1)* automaton collapses considerably; for example, all reduce-only states are now empty (since X is absent) and can be combined. This automaton has the same shift behavior as the LR(1) automaton, but when a reduce is called for, no further information is available from the automaton, and stack examination is required. If the stack examination is of bounded size, the grammar was CLR(1).
The design of the LR state automata is given in great detail, with examples, but the stack examination algorithms are not given explicitly, and no examples are provided. No complete parsing example is given.
Should have been in English.

100. **Scott**, Elizabeth and Johnstone, Adrian. Reducing non-determinism in reduction-modified LR(1) parsers. Technical Report CSD-TR-02-04, Royal Holloway, University of London, Jan. 2002. Theory of the reduction-modified LR(1) parser used in GRMLR parsing (Scott [177]), plus some improvements.

18.1.5 Left-Corner Parsing

This section also covers a number of related top-down non-canonical techniques: production chain, LLP(k), PLR(k), etc. The bottom-up non-canonical techniques are collected in (Web)Section 18.2.2.

101. **Rosenkrantz**, D. J. and Lewis, II, P. M. Deterministic left-corner parsing. In *IEEE Conference Record 11th Annual Symposium on Switching and Automata Theory*, volume 11, pages 139–152, 1970. An LC(k) parser decides the applicability of a rule when it has seen the initial non-terminal of the rule if it has one, plus a look-ahead of k symbols. Identifying the initial non-terminal is done by bottom-up parsing, the rest of the rule is recognized top-down. A canonical LC pushdown machine can be constructed in which the essential entries on the pushdown stack are pairs of non-terminals, one telling what non-terminal has been recognized bottom-up and the other what non-terminal is predicted top-down. As with LL, there is a difference between LC and strong-LC. There is a simple algorithm to convert an LC(k) grammar into LL(k) form; the resulting grammar may be large, though.

102. **Lomet**, David B. Automatic generation of multiple exit parsing subroutines. In J. Loeckx, editor, *Automata, Languages and Programming*, volume 14 of *Lecture Notes in Computer Science*, pages 214–231. Springer-Verlag, Berlin, 1974. A *production chain* is a chain of production steps $X_0 \rightarrow X_1\alpha_1$, $X_1 \rightarrow X_2\alpha_2$, ..., $X_{n-1} \rightarrow \mathtt{t}\alpha_n$, with $X_0 \ldots X_n - 1$ non-terminals and `t` a terminal. If the input is known to derive from X_0 and starts with `t`, each production chain from X_0 to `t` is a possible explanation of how `t` was produced. The set of all production chains connecting X_0 to `t` is called a *production expression*. An efficient algorithm for the construction and compression of production expressions is given. Each production expression is then implemented as a subroutine which contains the production expression as a FS automaton.

103. **Demers**, Alan J. Generalized left corner parsing. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 170–182, New York, 1977. ACM. The right-hand side of each rule is required to contain a marker. The part on the left of the marker is the left corner; it is recognized by SLR(1) techniques, the rest by LL(1) techniques. An algorithm is given to determine the first admissible position in each right-hand side for the marker. Note that this is unrelated to the Generalized Left-Corner Parsing of Nederhof [172].

104. **Soisalon-Soininen**, Eljas and Ukkonen, Esko. A method for transforming grammars into LL(k) form. *Acta Inform.*, 12:339–369, 1979. Introduces a subclass of the LR(k) grammars called predictive LR(k) (PLR(k)). The deterministic LC(k) grammars are strictly included in this class, and a grammatical transformation is presented to transform a PLR(k) into an LL(k) grammar. PLR(k) grammars can therefore be parsed with the LL(k) parser of the transformed grammar. A consequence is that the classes of LL(k), LC(k), and PLR(k) languages are identical.

105. **Nederhof**, M.-J. A new top-down parsing algorithm for left-recursive DCGs. In *5th International Symposium on Programming Language Implementation and Logic Programming*, volume 714 of *Lecture Notes in Computer Science*, pages 108–122. Springer-Verlag, Aug. 1993. "Cancellation parsing" predicts alternatives for leftmost non-terminals, just as any top-down parser does, but keeps a set of non-terminals that have already been predicted as left corners, and when a duplicate turns up, the process stops. This basically parses the largest left-corner tree with all different non-terminals on the left spine. The original prediction then has to be restarted to see if there is a still larger tree.
It is shown that this is the minimal extension of top-down parsing that can handle left recursion. The parser can be made deterministic by using look-ahead and three increasingly demanding definitions are given, leading to $C(k)$, strong $C(k)$ and severe $C(k)$. It is shown that $LL(k) \subset C(k) \subset LC(k)$ and likewise for the strong variant. Cancellation parsing cannot handle hidden left recursion.
The non-deterministic case is presented as definition-clause grammars, and an algorithm is given to use attributes to aid in handling hidden left recursion. The generation of a non-deterministic cancellation parser requires no analysis of the grammar: each rule can be translated in isolation.
See also Chapter 5 of Nederhof's thesis [156].

106. **Žemlička**, Michal and Král, Jaroslav. Run-time extensible deterministic top-down parsing. *Grammars*, 2(3):283–293, 1999. Easy introduction to "kind" grammars. Basically a grammar is *kind* if it is LL(1) after left-factoring and eliminating left recursion. The paper explains how to perform these processes automatically during parser generation, which results in traditional-looking and easily modifiable recursive descent parsers. The corresponding pushdown automaton is also described.

107. **Žemlička**, Michal. Parsing with oracle. In *Text, Speech and Dialogue*, volume 4188 of *Lecture Notes in Computer Science*, pages 309–316. Springer, 2006. Summary of the definitions of the oracle-enhanced parsing automata from [108]; no examples, no applications. "Oracle" is not a language, as the title suggests, but just "an oracle".

108. **Žemlička**, Michal. *Principles of Kind Parsing*. PhD thesis, Technical report, Charles University, Prague, June 2006. Theory, practice, applications, and a parser for kind grammars [106], extensively explained. The parser is based on an oracle-enhanced PDA. To this end the notion of look-ahead is extended to that of an oracle, which allows great freedom of adaptation and modification. The automated construction of oracles for complicated look-ahead sets is discussed and examples are given.

18.1.6 Precedence and Bounded-Right-Context Parsing

Papers on bounded-context (BC) and bounded-context parsable (BCP), which are non-canonical, can be found in (Web)Section 18.2.2.

109. **Adams**, Eldridge S. and Schlesinger, Stewart I. Simple automatic coding systems. *Commun. ACM*, 1(7):5–9, July 1958. Describes a simple parser for arithmetic expressions: read the entire expression, start at the end, find the first open parenthesis, from there find the first closing parenthesis to the right, translate the isolated parentheses-free expression, replace by result, and repeat until all parentheses are gone. A parentheses-free expression is parsed by distinguishing between one-factor terms and more-than-one-factor terms, but the algorithm is not made explicit.

110. **Wolpe**, Harold. Algorithm for analyzing logical statements to produce a truth function table. *Commun. ACM*, 1(3):4–13, March 1958. The paper describes an algorithm to convert a Boolean expression into a decision table. The expression is first fully parenthesized through a number of substitution rules that represent the priorities of the operators. Parsing is then done by counting parentheses. Further steps construct a decision table.

111. **Sheridan**, P. B. The arithmetic translator-compiler of the IBM FORTRAN automatic coding system. *Commun. ACM*, 2:9–21, Feb. 1959. Amazingly compact description of a optimizing Fortran compiler; this digest covers only the translation of arithmetic expressions.
The expression is first turned into a fully parenthesized one, through a precedence-like scheme (**+** is turned into **))) + (((**, etc.). This leads to a list of triples (node number, operator, operand). This list is then reduced in several sweeps to eliminate copy operations and common subexpressions; these optimizations are machine-independent. Next several machine-dependent (for the IBM 704) optimizations are performed.

112. **Samelson**, K. and Bauer, F. L. Sequential formula translation. *Commun. ACM*, 3(2):76–83, Feb. 1960. (Parsing part only.) When translating a dyadic formula from left to right, the translation of an operator often has to be postponed because a later operator has a higher precedence. It is convenient to put such operators aside in a pushdown *cellar* (which later became known as a "stack"); the same applies to operands, for which an "address cellar" is introduced.
All parsing decisions can then be based on the most recent operator ξ in the cellar and the next input symbol α (sometimes called χ in the paper). If α is an operand, it is stacked on the address cellar and a new input symbol is read; otherwise a matrix is indexed with ξ and α, resulting in an action to be performed. This leads to a variant of operator precedence parsing.
The matrix (given in Table 1) was produced by hand from a non-existing grammar. It contains 5 different actions, two of which (1 and 3) are shifts (there is a separate shift to fill the empty stack). Action 5 is the general reduction for a dyadic operator, popping both the operator cellar and the address cellar. Action 4 handles one parentheses pair by discarding both ξ **(** **(** and α **(** **)**. Action 2 is a specialized dyadic reduction, which incorporates the subsequent shift; it is used when such a shift is guaranteed, as in two successive operators of the same precedence, and works by overwriting the top element in the cellar.

113. **Floyd**, Robert W. A descriptive language for symbol manipulation. *J. ACM*, 8:579–584, Oct. 1961. Original paper describing Floyd productions. See Section 9.3.2.

114. **Paul**, M. A general processor for certain formal languages. In *Symposium on Symbolic Languages in Data Processing*, pages 65–74, New York, 1962. Gordon and Breach. Early paper about the BRC(2,1) parser explained further in Eickel et al. [115]. Gives precise criteria under which the BRC(2,1) parser is deterministic, without explaining the parser itself.

115. **Eickel**, J., Paul, M., Bauer, F. L., and Samelson, K. A syntax-controlled generator of formal language processors. *Commun. ACM*, 6(8):451–455, Aug. 1963. In this paper, the authors develop and describe the BRC(2,1) parser already introduced by Paul [114]. The reduction rules in the grammar must have the form $U \leftarrow V$ or $R \leftarrow ST$. A set of 5 intuitively reasonable parse table construction rules are given, which assign to each combination $X_{n-1}X_n, t_k$ one of the actions $U \leftarrow X_n$, $R \leftarrow X_{n-1}X_n$, *shift* or *report error*. Here X_n is the top element of the stack and X_{n-1} the one just below it; t_k is the next input token.
An example of such a parse table construction rule is: if X_n can be reduced to a U such that $X_{n-1}U$ can be reduced to an R such that R can be followed by token t_k, then the table entry for $X_{n-1}X_n, t_k$, should contain $U \leftarrow \cdots \leftarrow X_n$. Note that chains of unit reductions are performed in one operation. The table is required to have no multiple entries. The terminology in the paper differs considerably from today's.

116. **Floyd**, Robert W. Syntactic analysis and operator precedence. *J. ACM*, 10(3):316–333, July 1963. Operator-precedence explained and applied to an ALGOL 60 compiler.

117. **Floyd**, Robert W. Bounded context syntax analysis. *Commun. ACM*, 7(2):62–67, Feb. 1964. For each right-hand side of a rule $A \rightarrow \alpha$ in the grammar, enough left and/or right context

is constructed (by hand) so that when α is found obeying that context in a sentential form in a left-to-right scan in a bottom-up parser, it can safely be assumed to be the handle. If you succeed, the grammar is *bounded-context*; if in addition the right hand contexts do not contain non-terminals, the grammar is *bounded-right-context*; analogously for bounded-left-context. A detailed example is given; it is BRC(2,1). The paper ends with a report of the discussion that ensued after the presentation of the paper.

118. **Wirth**, Niklaus and Weber, Helmut. EULER: A generalization of ALGOL and its formal definition, Part 1/2. *Commun. ACM*, 9(1/2):13–25/89–99, Jan. 1966. Detailed description of simple and extended precedence. A table generation algorithm is given. Part 2 contains the complete precedence table plus functions for the language EULER.

119. **Colmerauer**, A. *Précedence, analyse syntactique et languages de programmation*. PhD thesis, Technical report, Université de Grenoble, Grenoble, 1967, (in French). Defines two precedence schemes: total precedence, which is non-canonical, and left-to-right precedence, which is like normal precedence, except that some non-terminals are treated as if they were terminals. Some other variants are also covered, and an inclusion graph of the language types they define is shown, which includes some terra incognita.

120. **Bell**, James R. A new method for determining linear precedence functions for precedence grammars. *Commun. ACM*, 12(10):567–569, Oct. 1969. The precedence relations are used to set up a connectivity matrix. Take the transitive closure and count 1s in each row. Check for correctness of the result.

121. **Ichbiah**, J. and Morse, S. A technique for generating almost optimal Floyd-Evans productions of precedence grammars. *Commun. ACM*, 13(8):501–508, Aug. 1970. The notion of "weak precedence" is defined in the introduction. The body of the article is concerned with efficiently producing good Floyd-Evans productions from a given weak precedence grammar. The algorithm leads to production set sizes that are within 20% of the theoretical minimum.

122. **Loeckx**, Jacques. An algorithm for the construction of bounded-context parsers. *Commun. ACM*, 13(5):297–307, May 1970. The algorithm systematically generates all bounded-right-context (BRC) states the parser may encounter. Since BRCness is undecidable, the parser generator loops if the grammar is not BRC(m,n) for any value of m and n.

123. **McKeeman**, William M., Horning, James J., and Wortman, David B. *A Compiler Generator*. Prentice Hall, Englewood Cliffs, N.J., 1970. Good explanation of precedence and mixed-strategy parsing. Full application to the XPL compiler.

124. **Gray**, James N. and Harrison, Michael A. Canonical precedence schemes. *J. ACM*, 20(2):214–234, April 1973. The theory behind precedence parsing, unifying the schemes of Floyd [116], Wirth and Weber [118], and the canonical parser from Colmerauer [190]. Basically extends simple precedence by appointing some non-terminals as honorary terminals, the *strong operator set*; different strong operator sets lead to different parsers, and even to relationships with LR(k). Lots of math, lots of information. The paper emphasizes the importance of parse tree nodes being created in a clear and predictable order, in short "canonical".

125. **Levy**, M. R. Complete operator precedence. *Inform. Process. Lett.*, 4(2):38–40, Nov. 1975. Establishes conditions under which operator-precedence works properly.

126. **Henderson**, D. S. and Levy, M. R. An extended operator precedence parsing algorithm. *Computer J.*, 19(3):229–233, 1976. The relation $\lessdot$ is split into $\lessdot_1$ and $\lessdot_2$. $a \lessdot_1 b$ means that a may occur next to b, $a \lessdot_2 b$ means that a non-terminal has to occur between them. Likewise for $\doteq$ and $\gtrdot$. This is *extended operator-precedence*.

127. **Bertsch**, Eberhard. The storage requirement in precedence parsing. *Commun. ACM*, 20(3):192–194, March 1977. Suppose for a given grammar there exists a precedence matrix but the precedence functions f and g do not exist. There always exist sets of precedence functions f_i and g_j such that for two symbols a and b, comparison of $f_{c(b)}(a)$ and $g_{d(a)}(b)$ yields the precedence

relation between a and b, where c and d are selection functions which select the f_i and g_j to be compared. An algorithm is given to construct such a system of functions.

128. **Williams**, M. H. Complete operator precedence conditions. *Inform. Process. Lett.*, 6(2):60–62, April 1977. Revision of the criteria of Levy [125].

129. **Williams**, M. H. Conditions for extended operator precedence parsing. *Computer J.*, 22(2):164–168, 1979. Tighter analysis of extended operator-precedence than Henderson and Levy [126].

130. **Gonser**, Peter. *Behandlung syntaktischer Fehler unter Verwendung kurzer, fehlereinschließender Intervalle*. PhD thesis, Technical report, Technische Universität München, München, July 21 1981, (in German). The author's investigations on error treatment (see [308]) show that the precedence parsing algorithm has good error reporting properties because it allows the interval of the error to be securely determined. Since the existing precedence techniques are too weak, several new precedence grammars are proposed, often using existing terms and symbols ($\lesssim$, etc.) with new meanings.
1. An operator precedence grammar in which, for example, $a \lessdot b$ means that b can be the beginning of a non-terminal that can follow a, and $a \lesssim b$ means that b can be the first terminal in a right-hand side of a non-terminal that can follow a.
2. An extended operator precedence grammar in which two stack symbols, which must be a terminals, and a non-terminal form a precedence relation with the next input token.
3. An indexed operator precedence grammar, a grammar in which all terminal symbols are different. This virtually assures all kinds of good precedence properties; but hardly any grammar is indexed-operator. Starting from an LR(0) grammar it is, however, possible to construct a parsing algorithm that can disambiguate tokens on the fly during parsing, just in time for the precedence algorithm, by attaching LR state numbers to them. This distinguishes for example the `(` in function call `f(3)` from the `(` in the expression `x×(y+z)`. The proof of this theorem takes 19 pages; the algorithm itself another 5.
Each of these techniques comes with a set of rules for error correction.

131. **Williams**, M. H. A systematic test for extended operator precedence. *Inform. Process. Lett.*, 13(4-5):187–190, 1981. The criteria of Williams [129] in algorithmic form.

132. **Peyton Jones**, Simon L. Parsing distfix operators. *Commun. ACM*, 29(2):118–122, Feb. 1986. A distfix operator is an operator which is distributed over its operands; examples are `if . then . else . fi` and `rewrite . as . using . end`. It is useful to allow users to declare such operators, especially in functional languages.
Such distfix operators are introduced in a functional language using two devices. First the keywords of a distfix operator are given different representations depending on their positions: prefix keywords are written with a trailing dot, infix ones with a leading and a trailing dot, and postfix ones with a leading dot; so the user is required to write `rewrite. x .as. y .using. z .end`. These forms are recognized by the lexical analyzer, and given the token classes `PRE.TOKEN`, `IN.TOKEN`, and `END.TOKEN`. Second, generic rules are written in *yacc* to parse such structures.

133. **Aasa**, Annika. Precedences in specifications and implementations of programming languages. *Theoret. Comput. Sci.*, 142(1):3–26, May 1995. Fairly complicated but clearly explained algorithm for parsing expressions containing infix, prefix, postfix and distfix operators with externally given precedences. Even finding a sensible definition of the "correct" parsing is already difficult with those possibilities.

18.1.7 Finite-State Automata

134. **Mealy**, George H. A method for synthesizing sequential circuits. *Bell System Technical J.*, 34(5):1045–1079, 1955. Very readable paper on "sequential circuits" aka finite-state automata, except that the automata are built from relays connected by wires. The circuits consist

of AND, OR, and NOT gates, and delay units; the latter allow delayed feedback of signals from somewhere in the circuit to somewhere else. Starting from the circuit diagram, a set of input, output, excitation and state variables are defined, where the excitation variables describe the input to the delay units and the states their output. The delay units provide the finite-state memory. Since only the output variables are observable in response to the inputs, this leads naturally to attaching semantics to the transitions rather than to the (unobservable) states.
The relationships between the variables are recorded in "truth tables". These are shown to be equivalent to Moore's sequential machines. Moore's minimization procedure is then transformed so as to be applicable to truth tables. These then lead to minimal-size sequential circuits.
The rest of the paper dwells on the difficulties of asynchronous circuits, in which unknown delays may cause race conditions. The truth table method is reasonably good at handling them.

135. **Kleene**, S. C. Representation of events in nerve nets and finite automata. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–42. Princeton University Press, Princeton, N.J., 1956. Introduces the Kleene star, but its meaning differs from the present one. An *event* is a $k \times l$ matrix, defining the k stimuli to k neurons over a time span of length l; a stimulus has the value 0 or 1. Events can be concatenated by just writing them one after another: EF means first there was an event E and then an event F; the final event F is in the present, and the train can then be applied to the set of neurons. Events can be repeated: EF, EEF, $EEEEF$, ... E^nF. Increasing the n introduces more and more events E in a more and more remote past, and since we do not usually know exactly what happened a long time ago, we are interested in the set E^0F, E^1F, E^2F, E^3F, ... E^nF for $n \to \infty$. This set is written as $E*F$, with a binary operator $*$ (not raised), and means "An occurrence of F preceded by any number of Es". The unary raised star does not occur in the paper, so its origin must be elsewhere.

136. **Moore**, E. F. Gedanken-experiments on sequential machines. In *Automata Studies*, number 34 in Annals of Mathematics Studies, pages 129–153. Princeton University Press, Princeton, NJ, 1956. A finite-state automaton is endowed with an output function, to allow experiments with the machine; the machine is considered a black box. The output at a given moment is equal to the state of the FSA at that moment. Many, sometimes philosophical, conclusions are drawn from this model, culminating in the theorem that there is a sequence of at most $n^{nm+2}p^n/n!$ input symbols that distinguishes an FSA with n states, m different input symbols, and p different output symbols from any other such FSA.

137. **McNaughton**, R. and Yamada, H. Regular expressions and state graphs for automata. *IRE Transactions Computers*, EC-9(1):39–47, March 1960. Sets of sequences of input-output transitions are described by regular expressions, which are like regular expressions in CS except that intersection and negation are allowed. The output is generated the moment the automaton enters a state. A subset-like algorithm for converting regular expressions without intersection, negation, and ε-rules into FSAs is rigorously derived. The trouble-makers are introduced by repeatedly converting the innermost one into a well-behaved regular expression, using one of three conversion theorems. Note that the authors use ϕ for the empty sequence (string) and Λ for the empty set of strings (language).

138. **Brzozowski**, J. A. Canonical regular expressions and minimal state graphs for definite events. In *Symp. on Math. Theory of Automata*, pages 529–561, Brooklyn. N.Y., 1963. Brooklyn Politechnic. Getting unique minimal regular expressions from FSAs is difficult. The author defines a *definite event* as a regular set described by an expression of the form $E|\Sigma^*F$, where E and F are finite sets of finite-length strings. Using Brzozowski derivatives, the author gives an algorithm that will construct a definite event expression for any FSA that allows it.

139. **Brzozowski**, Janusz A. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964. The author starts from regular expressions over [0,1] that use concatenation and Kleene star only, and then adds union, intersection, complement and exclusive-or. Next the *derivative* $D_s(R)$ of the regular language R with respect to s is defined as anything that can follow a prefix s in a sequence in R. Many theorems about these derivatives are proved, for example: "A sequence s is in R if and only if $D_s(R) = \varepsilon$. More importantly, it is shown that there are only a finite number of

different derivatives of a given R; these correspond to the states in the DFA. This is exploited to construct that DFA for regular expressions featuring the extended set of operations. Many examples. For an application of Brzozowski derivatives to XML validation see Sperberg-McQueen [359].

140. **Thompson**, Ken. Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968. The regular expression is turned into a transition diagram, which is then interpreted in parallel. Remarkably, each step generates (IBM 7094) machine code to execute the next step.

141. **Aho**, Alfred V. and Corasick, Margaret J. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, June 1975. A given string embedded in a longer text is found by a very efficient FS automaton derived from that string.

142. **Krzemień**, Roman and Łukasiewicz, Andrzej. Automatic generation of lexical analyzers in a compiler-compiler. *Inform. Process. Lett.*, 4(6):165–168, March 1976. A grammar is *quasi-regular* if it features left or right recursion only; such grammars generate regular languages. A straightforward bottom-up algorithm is given to identify all quasi-regular subgrammars in a CF grammar, thus identifying its "lexical part", the part that can be handled by a lexical analyser in a compiler.

143. **Boyer**, Robert S. and Moore, J. Strother. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977. We want to find a string S of length l in a text T and start by positioning $S[1]$ at $T[1]$. Now suppose that $T[l]$ does not occur in S; then we can shift S to $T[l+1]$ without missing a match, and thus increase the speed of the search process. This principle can be extended to blocks of more characters.

144. **Ostrand**, Thomas J., Paull, Marvin C., and Weyuker, Elaine J. Parsing regular grammars with finite lookahead. *Acta Inform.*, 16:125–138, 1981. Every regular (Type 3) language can be recognized by a finite-state automaton without look-ahead, but such a device is not sufficient to do parsing. For parsing, look-ahead is needed; if a regular grammar needs a look-ahead of k tokens, it is called *FL(k)*. FS grammars are either FL(k), FL(∞) or ambiguous; a decision algorithm is described, which also determines the value of k, if appropriate.
A simple parsing algorithm is a FS automaton governed by a look-up table for each state, mapping look-aheads to new states. A second algorithm avoids these large tables by constructing the relevant look-ahead sets on the fly.

145. **Karp**, Richard M. and Rabin, Michael O. Efficient randomized pattern-matching algorithms. *IBM J. Research and Development*, 31(2):249–260, 1987. We want to find a string S of length l in a text T. First we choose a hash function H that assigns a large integer to any string of length l and compute $H(S)$ and $H(T[1\cdots l])$. If they are equal, we compare S and $T[1\cdots l]$. If either fails we compute $H(T[2\cdots l+1])$ and repeat the process. The trick is to choose H so that $H(T[p+1\cdots p+l])$ can be computed cheaply from $H(T[p\cdots p+l-1])$. Note that this is not a FS algorithm but achieves a similar result.

146. **Jones**, Douglas W. How (not) to code a finite-state machine. *ACM SIGPLAN Notices*, 23(8):19–22, Aug. 1988. Small, well-structured and efficient code can be generated for a FS machine by deriving a single deterministic regular expression from the FS machine and implementing this expression directly using **while** and **repeat** constructions.

147. **Aho**, A. V. Algorithms for finding patterns in strings. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science - Algorithms and Complexity, Vol. A*, pages 255–300. Elsevier, Amsterdam, The Netherlands, 1990. Chapter 5 of the handbook. Encyclopedic article on the subject, covering the state of the art in:
    single string matching:
        brute-force
        Karp-Rabin, caterpillar hash function
        Knuth-Morris-Pratt, automaton, forward
        Boyer-Moore, backward
    multiple string matching:

Aho-Corasick
Commentz-Walter, best description around
regular expression matching:
Thompson NFSA construction
regular expression matching with variables:
proved NP-complete
longest common substring location:
longest path in matrix
McCreight suffix trees
giving a very readable account of each of them, often with proof and complexity analysis. Draws amazing conclusions from Cook's Theorem: "Every 2-way deterministic pushdown automaton (2DPDA) language can be recognized in linear time on a random-access machine".
The paper ends with 139 literature references.

148. **Roche**, Emmanuel. Factorization of finite-state transducers. Technical Report TR-95-2, Mitsubishi Electric Research Laboratories, Cambridge, MA., Feb. 1995. A non-deterministic FSA F is decomposed into two deterministic ones by constructing a new graph on the states of F, in which arcs are present between each pair of states that can be reached by the same input string. This graph is then colored and the colors are considered new states. Two new automata are constructed, one which leads from the states of F to colors and one which leads from colors to states of F; they are constructed in such a way that they are deterministic. The concatenation C of these automata is equivalent to F. Often C is smaller than the traditional minimized deterministic equivalent of F, but of course it takes twice the time to do a transition.

149. **Watson**, Bruce W. A new regular grammar pattern matching algorithm. In Josep Díaz and Maria Serna, editors, *Algorithms: ESA '96, Fourth Annual European Symposium*, volume 1136 of *Lecture Notes in Computer Science*, pages 364–377, Barcelona, Spain, Sept. 1996. Springer. Careful derivation of an algorithm, which applies the Boyer-Moore token-skipping technique [143] to regular expression matching.

150. **Brüggemann-Klein**, Anne and Wood, Derick. The validation of SGML content models. *Math. Comp. Modelling*, 25(4):73–84, 1997. The checking of an SGML file requires the construction of a FS automaton based on the document grammar. The paper gives criteria such that the automaton can be constructed in linear time.

151. **Laurikari**, Ville. Efficient submatch addressing for regular expressions. Master's thesis, Helsinki University of Technology, Helsinki, Nov. 2001. Gives a linear-time algorithm for unambiguous substring parsing with a regular grammar, i.e., the algorithm returns a structured match for a regular expression matching a segment of the input. Unambiguity is enforced by three rules: longest possible match; longest possible subexpression match; and last possible match, in this order. Each transition in the NFA is augmented with a "tag", a variable which is set to the current input position when the transition is taken. A series of increasingly efficient but complicated algorithms for simulating tagged NFAs is given. Next it is shown how the gathered information can be used for creating a parse tree or to do approximate regular expression matching.
Chapters 4 and 5 report on the conversion of the tagged NFA to a tagged DFA, and on speed and memory usage tests, in which the tagged DFA performs between reasonably and spectacularly well. Excellent description and analysis of previous papers on finite-state parsing.

18.1.8 General Books and Papers on Parsing

152. **Aho**, Alfred V. and Ullman, Jeffrey D. *The Theory of Parsing, Translation and Compiling: Volume I: Parsing*. Prentice Hall, Englewood Cliffs, N.J., 1972. The book describes the parts of formal languages and automata theory relevant to parsing in a strict mathematical fashion. Since a considerable part of the pertinent theory of parsing had already been developed in 1972, the book is still reasonably up to date and is a veritable trove of definitions, theorems, lemmata and

proofs.
The required mathematical apparatus is first introduced, followed by a survey of compiler construction and by properties of formal languages. The rest of the book confines itself to CF and regular languages.
General parsing methods are treated in full: backtracking top-down and bottom-up, CYK and Earley. Directional non-backtracking methods are explained in detail, including general LL(k), LC(k) and LR(k), precedence parsing and various other approaches. A last chapter treats several non-grammatical methods for language specification and parsing.
Many practical matters concerning parser construction are treated in volume II, where the theoretical aspects of practical parser construction are covered; recursive descent is not mentioned, though.

153. **Backhouse**, Roland C. *Syntax of Programming Languages*. Prentice Hall, London, 1979. Grammars are considered in depth, as far as they are relevant to programming languages. FS automata and the parsing techniques LL and LR are treated in detail, and supported by lots of well-explained math. Often complete and efficient algorithms are given in Pascal. Much attention is paid to error recovery and repair, especially to least-cost repairs and locally optimal repairs. Definitely recommended for further reading.

154. **Nijholt**, Anton. Parsing strategies: A concise survey. In J. Gruska and M. Chytil, editors, *Mathematical Foundations of Computer Science*, volume 118 of *Lecture Notes in Computer Science*, pages 103–120. Springer-Verlag, Berlin, 1981. The context-free parser and language field is surveyed in terse prose. Highly informative to the connoisseur.

155. **Leermakers**, R. *The Functional Treatment of Parsing*. Kluwer Academic Publishers, 1993. Parsing for the mathematically inclined, based on a formalism of the author's own creation. In fact the author proposes what seems to be a calculus for parsers: basic parsing problems are cast in the formalism, computations are performed on these formulas, and we arrive at new formulas that translate back into actual parsers, for example Earley or recursive ascent LR. These parsers have the form of functional programs.
The book contains a two-chapter introduction to the formalism, followed by chapters on applications to recursive descent, recursive ascent, parse forests, LR parsing, grammar transformations and attribute grammars. Some philosophical notes on these and other subjects end the book. The text is written in a deceptively simple but very clear prose, interleaved with considerable stretches of formulas.
The formalism has a high threshold, and requires considerable mathematical sophistication (Lambek types, etc.); but it has the clear and redeeming advantage that it is functional (excuse the pun): it allows actual computations to be performed and is not just an exposition aid.
For a review, see Schabes [157]. For an implementation see Sperber and Thiemann [356].

156. **Nederhof**, M.-J. *Linguistic Parsing and Program Transformation*. PhD thesis, Technical report, Katholieke Universiteit Nijmegen, Nijmegen, 1994. Contains in coherent chapter form versions of the following papers: "Generalized left-corner parsing" [172], "An Optimal Tabular Parsing Algorithm" [33], "Increasing the Applicability of LR Parsing" [94], and "Top-Down Parsing for Left-Recursive Grammars" [105], preceded by an introduction to parsing, and followed by a chapter on attribute propagation, and one on a grammar workbench.

157. **Schabes**, Yves. The functional treatment of parsing: Book review. *Computational Linguistics*, 21(1):112–115, 1995. Review of Leermakers [155]. Praises the approach and the courage. Criticizes the unusual formalism and some of the complexity analysis.

158. **Sikkel**, K. *Parsing Schemata*. Springer Verlag, 1996. Describes the primordial soup algorithm: the soup initially contains all grammar rules and all rules of the form $A \rightarrow t_i$ for all t_i in the input; both are in effect parse tree fragments. During stewing fragments are combined according to obvious rules, until all possible combinations have been formed. Then the complete parse trees float to the surface.
The rules of this algorithm, which is actually a transitive closure algorithm, are then formalized into sets of inference rules geared to parsing, called parsing schemata. These are then specialized to form many existing parsing methods and some new ones, including predictive head-corner parsing

[204]) and a parallel bottom-up GLR parser [233]. All this is supported by great mathematical rigor, but enough diagrams and examples are given to keep it readable.

18.2 Advanced Parsing Subjects

18.2.1 Generalized Deterministic Parsing

159. **Lang**, Bernard. Deterministic techniques for efficient non-deterministic parsers. In J. Loeckx, editor, *Automata, Languages and Programming*, volume 14 of *Lecture Notes in Computer Science*, pages 255–269. Springer-Verlag, Berlin, 1974. Explores the theoretical properties of doing breadth-first search to resolve the non-determinism in a bottom-up automaton with conflicts. See Tomita [160, 161, 162] for a practical realization.

160. **Tomita**, Masaru. LR parsers for natural languages. In *10th International Conference on Computational Linguistics*, pages 354–357. ACL, 1984. Two detailed examples of GLR parsing, on two English sentences. The parser features equal state combination, but no equal stack combination.

161. **Tomita**, Masaru. An efficient context-free parsing algorithm for natural languages. In *International Joint Conference on Artificial Intelligence*, pages 756–764, 1985. Explains GLR parsing in three steps: using stack lists, in which each concurrent LR parser has its own private stack; using tree-structured stacks, in which equal top states are combined yielding a forest of trees; and using the full graph-structured stacks. Also points out the defect in Earley's parse forest representation (Earley [14]), and shows that repairing it causes the algorithm to require more than $O(n^3)$ space on highly ambiguous grammars.

162. **Tomita**, Masaru. *Efficient Parsing for Natural Language*. Kluwer Academic Publishers, Boston, 1986. Tomita describes an efficient parsing algorithm to be used in a "natural-language setting": input strings of some tens of words and considerable but not pathological ambiguity. The algorithm is essentially LR, starting parallel parses when an ambiguity is found in the LR-table. Full examples are given of handling ambiguities, lexical elements with multiple meanings and unknown lexical elements.
The algorithm is compared extensively to Earley's algorithm by measurement and it is found to be consistently five to ten times faster than the latter, in the domain for which it is intended. Earley's algorithm is better in pathological cases; Tomita's fails on unbounded ambiguity. No time bounds are given explicitly, but graphs show a behavior better than $O(n^3)$. Bouckaert, Pirotte and Snelling's algorithm [17]) is shown to be between Earley's and Tomita's in speed.
MacLisp programs of the various algorithms are given and the application in the Nishida and Doshita Machine Translation System is described.
For a review see Bańko [163].

163. **Bańko**, Mirosław. Efficient parsing for natural language: Book review. *Computational Linguistics*, 14(2):80–81, 1988. Two-column summary of Tomita's book [162].

164. **Billot**, Sylvie and Lang, Bernard. The structure of shared forests in ambiguous parsing. In *27th Annual Meeting of the Association for Computational Linguistics*, pages 143–151, June 1989. A parse forest resulting from parsing can be represented very conveniently by a grammar; subtrees are shared because they are represented by the same non-terminal. If the grammar is in 2-form (Sheil [20]), its size is $O(n^3)$, which is satisfactory.
To investigate this representation with various parsing schemes, the PDT interpreter of Lang [210] is implemented and LR(0), LR(1), LALR(1), LALR(2), weak precedence, and LL(0) transducers for some simple grammars are compared using it. A general observation is that parsers with great resolution power perform worse than weak precedence, because the overspecificness of the context prevents useful sharing of subtrees.

165. **Kipps**, James R. GLR parsing in time $O(n^3)$. In M. Tomita, editor, *Generalized LR Parsing*, pages 43–59. Kluwer Academic Publishers, Boston, 1991. Proves that the original GLR algorithm costs $O(n^{k+1})$ for grammars with rules of maximum length k. Identifies as cause of this complexity the searching of the graph-structured stack(GSS) during reduces. This process can take $O(i^l)$ actions at position i for a reduce of length l; worst case it has to be done for each input position, hence the $O(n^{k+1})$. The paper describes a memoization technique that stores for each node in the GSS and each distance $1 \leq p \leq k$ all nodes at distance p in an ancestors table ; this makes reduction $O(1)$, and when done cleverly the ancestors table can fully replace the GSS. Building the ancestors table costs $O(i^2)$ regardless of the grammar, hence the overall $O(n^3)$. For almost all grammars, however, the original algorithm is faster.
Contains explicit code for the original and the improved GLR algorithm.

166. **Lankhorst**, Marc. An empirical comparison of generalized LR tables. In R. Heemels, A. Nijholt, and K. Sikkel, editors, *Tomita's Algorithm: Extensions and Applications (TWLT1)*, number 91-68 in Memoranda Informatica in Twente Workshops on Language Technology, pages 87–93, Enschede, the Netherlands, 1991. Unversity of Twente. Lots of bar graphs, showing that as far as speed is concerned, LALR(1) wins by perhaps 5-10% over LR(0) and SLR(1), but that LR(1) is definitely worse. The reason is the large number of states, which reduces the number of common stack suffixes to be combined. In the end, the much simpler LR(0) is only a few percent sub-optimal.

167. **Nozohoor-Farshi**, R. GLR parsing for ε-grammars. In M. Tomita, editor, *Generalized LR Parsing*, pages 61–75. Kluwer Academic Publishers, Boston, 1991. Shows that Tomita's algorithm [162] loops on grammars with hidden left recursion where the left recursion can be hidden by unbounded many εs. Remedies this by constructing and pushing on the stack an FSA representing the unbounded string of εs, with its proper syntactic structure. This also happens to make the parser impervious to loops in the grammar, thus achieving full coverage of the CF grammars.

168. **Piastra**, Marco and Bolognesi, Roberto. An efficient context-free parsing algorithm with semantic actions. In *Trends in Artificial Intelligence*, volume 549 of *Lecture Notes in Artificial Intelligence*, pages 271–280. Springer-Verlag, Oct. 1991. A simple condition is imposed on the unit and ε-rules of a grammar, which controls the reductions in a reduce/reduce conflict in a GLR parser. The result is that the reductions can be done so that multiple values result for locally ambiguous segments of the input, and common stack suffixes can still be combined as usual.

169. **Rekers**, J. Generalized LR parsing for general context-free grammars. Technical Report CS-R9153, CWI, Amsterdam, 1991. Extensive pictorial explanation of the GLR algorithm, including parse forest construction, with full algorithms in a clear pseudo-code. The GLR parser in compiled LeLisp is 3 times slower than *yacc* on Pascal programs; the Earley parser drowned in garbage collection. On the other hand, Earley wins over GLR on highly ambiguous grammars.

170. **Deudekom**, A. van and Kooiman, P. Top-down non-correcting error recovery in LLgen. Technical Report IR 338, Vrije Universiteit, Faculteit Wiskunde en Informatica, Amsterdam, Oct. 1993. Describes the implementation of a Richter-style [313] error recovery mechanism in *LLgen*, an LL(1) parser generator, using a Generalized LL parser. The parser uses a reversed tree with loops as the data structure to store the predictions.
The error-recovering parser is an add-on feature and is activated only when an error has been found. It has to work with a grammar for suffixes of the original language, for which the LL(1) parser generator has no parse tables. So the parser uses the FIRST and FOLLOW sets only. Full algorithms are described.
A specialized garbage collector for the particular data structure was designed by Wattel and is described in the report. Its activation costs about 10% computing time, but saves large amounts of memory. Efficiency measurements are provided.
See [320] for the error-handling part.

171. **Merrill**, G. H. Parsing non-LR(k) grammars with yacc. *Softw. Pract. Exper.*, 23(8):829–850, 1993. This is generalized LR by *depth-first* rather than breadth-first search. LR conflicts in the Berkeley LALR(1) parser *byacc* are solved by recursively starting a subparser for each possibility. These parsers run in "trial mode", which means that all semantic actions except those specifically marked for trial are suppressed. Once the right path has been found, normal parsing continues along it. The design process and the required modifications to *byacc*, the lexical analyser, and the input grammar are described in detail.

172. **Nederhof**, Mark-Jan. Generalized left-corner parsing. In *Sixth Conference of the European Chapter of the Association for Computational Linguistics*, pages 305–314, April 1993. A non-deterministic LC parser is extended to generalized parsing. This requires three problems to be solved to avoid non-termination: cycles, hidden left recursion, and ε-subtrees, subtrees that just produce ε. The hidden left recursion problem is solved by performing LC actions for any rule $A \rightarrow \mu B\beta$ when $\mu \stackrel{*}{\rightarrow} \varepsilon$; cycles are handled by creating loops in the parse tree under construction; and all empty subtrees are computed in advance. A special packing of the parse forest brings down the time and space complexity from $O(n^{p+1})$ where p is the length of the longest RHS to $O(n^3)$. Note that this technique is unrelated to the Generalized Left-Corner Parsing of Demers [103]. See also Chapter 2 of Nederhof's thesis [156].

173. **Lavie**, Aaron and Tomita, Masaru. GLR*: An efficient noise-skipping parsing algorithm for context-free grammars. In Harry Bunt and Masaru Tomita, editors, *Recent Advances in Parsing Technology*, pages 183–200. Kluwer Academic Publishers, Dordrecht, 1996. The GLR* parser finds the longest subsequence of the input that is in the language; it does supersequence parsing. At each input token shifts are performed from all states that allow it; this implements skipping arbitrary segments of the input. A grading function is then used to weed out unwanted parsings. The algorithm has exponential complexity; to counteract this, the number of skipping shifts per token can be limited; a limit of 5 to 10 gives good results.

174. **Nederhof**, M.-J. and Satta, G. Efficient tabular LR parsing. In *34th Annual Meeting of the Association for Computational Linguistics*, pages 239–246. Association for Computational Linguistics, 1996. Replaces the graph-structured stack of GLR parsing by the triangular table of CYK parsing, thus gaining efficiency and simplicity. The algorithm requires the grammar to be in "binary" form, which is Chomsky Normal form plus ε-rules. Explains how the very simple PDA used, among others, by Lang [210] can be obtained from the LR(0) table.

175. **Alonso Pardo**, M. A., Cabrero Souto, D., and Vilares Ferro, M. Construction of efficient generalized LR parsers. In Derick Wood and Sheng Yu, editors, *Second International Workshop on Implementing Automata*, volume 1436 of *Lecture Notes in Computer Science*, pages 7–24, Berlin, 1998. Springer-Verlag. Systematic derivation of an $O(n^3)$ GLR parsing algorithm from the Earley parser. First the Earley parser is rewritten as a dynamic programming algorithm. Next the Earley sets are compiled into sets of LR(0) states. Then look-ahead is introduced leading to LR(1) states, which are then combined into LALR(1) states. And finally implicit binarization is used to achieve the $O(n^3)$ complexity. The resulting parser consists of a considerable number of set definitions. It is about 5 times faster than the GLR parser from Rekers [169].

176. **Aycock**, John and Horspool, R. Nigel. Faster generalized LR parsing. In *Compiler Construction: 8th International Conference, CC'99*, volume 1575 of *Lecture Notes in Computer Science*, pages 32–46, Berlin, 1999. Springer Verlag. The stack is needed only for non-left recursion in an LR parser; everything else can be done by a DFA on the top of the stack. Recursion points (called "limit points" in the paper) are identified in the grammar using a heuristic form of the feedback arc set (FAS) algorithm. The grammar is broken at those points; this yields a regular grammar for which a DFA is constructed. Only when the DFA reaches a limit point, stack actions are initiated. The resulting very fast LR parser is used as a basis for a GLR parser. See also Aycock et al. [178].

177. **Scott**, Elizabeth, Johnstone, Adrian, and Hussain, Shamsa Sadaf. Tomita-style generalised LR parsers. Technical report, Royal Holloway, University of London, London, Dec. 2000. GLR parsers are bothered by nullable non-terminals at the beginning and end of a rule; those at the beginning cause errors when they hide left recursion; those at the end cause gross inefficiencies. The Generalized Reduction-Modified LR parser GRMLR solves the first problem by using an improved version of Nozohoor-Farshi's solution [167]. It solves the second problem by using an item $A \rightarrow \alpha\bullet\beta$ as a reduce item when β is nullable; a rule $A \rightarrow \alpha\beta$ with β nullable is called *right-nullable*.
For grammars that exhibit these problems a gain of roughly 30% is obtained. Full algorithm and correctness proof given.

178. **Aycock**, John, Horspool, R. Nigel, Janoušek, Jan, and Melichar, Bořivoj. Even faster generalized LR parsing. *Acta Inform.*, 37(9):633–651, 2001. Actions on the graph-structured stack are the most expensive items in generalized LR parsing and the fewer are required the better. For grammars without right recursion or hidden left recursion the stack actions between two shifts can be combined into two batches, a pop sequence and a push sequence. The optimization saves between 70% (for an unambiguous grammar) and 90% (for a highly ambiguous grammar) on processing time.

179. **Fortes Gálvez**, José, Farré, Jacques, and Aguiar, Miguel Ángel Pérez. Practical nondeterministic DR(k) parsing on graph-structured stack. In *Computational Linguistics and Intelligent Text Processing*, volume 2004 of *Lecture Notes in Computer Science*, pages 411–422. Springer Verlag, 2001. Generalized DR parsing. Applying the LR-to-DR table conversion of [95] does not work if the LR table has multiple entries, so a direct DR table construction algorithm is presented, which is capable of producing a non-deterministic DR table. A GSS algorithm using this table is described. Explicit algorithms are given.

180. **Johnstone**, Adrian and Scott, Elizabeth. Generalised reduction modified LR parsing for domain specific language prototyping. In *35th Hawaii International Conference on System Sciences*, page 282. IEEE, 2002. Summary of Scott et al. [177].

181. **Johnstone**, Adrian and Scott, Elizabeth. Generalised regular parsers. In *Compiler Construction: 12th International Conference, CC'03*, volume 2622 of *Lecture Notes in Computer Science*, pages 232–246. Springer Verlag, 2003. The grammar is decomposed into a regular grammar and a set of recursive grammars as follows. All derivations of the form $A \xrightarrow{*} \alpha A \beta$ with α and β not empty are blocked by replacing the A in the right-hand side of a rule involved in this derivation by a special symbol $A^{\perp}$. This yields the regular grammar; it is transformed into an NFA whose arcs are labeled with terminals, left- or right-recursive rule numbers $\mathcal{R}n$, or ε; this is a *Reduction Incorporated Automaton* (RIA). Next a *Recursive Call Automaton* (RCA) is constructed for each thus suppressed A. Each such automaton is then connected to the NFA by transitions marked with $push(A)$ and *pop*, in a way similar to that of ATNs. Finally the εs are removed using the subset algorithm; any other non-determinism remains. The resulting automaton is grafted on a graph-structured stack in GLR fashion. When the automaton meets a $push(A)$ transition, return info is stacked and the automaton proceeds to recognize an A; upon *pop* it returns.
The resulting parser operates with a minimum of stack operations, and with zero stack operations for almost all CF grammars that define a regular language. For proofs, etc. see Scott and Johnstone [183].

182. **Scott**, E., Johnstone, A., and Economopoulos, G. R. BRN-table based GLR parsers. Technical Report CSD-TR-03-06, CS Dept., Royal Holloway, University of London, London, July 2003. After a detailed informal and formal description of the GRMLR parser [177], called "RNGLR" for "right-nullable GLR" here, the notion "binary right-nullable", or BRN, is introduced, for the purpose of making the GLR parser run in $O(n^3)$ on all grammars. In BRN the LR(1) table is modified so that each reduction grabs at most 2 stack elements. This makes the GLR parser react as if the longest right-hand side is at most 2 long, and since GLR parsing is $O(n^{k+1})$, where k is the length of the longest right-hand side, $O(n^3)$ complexity results.

Many examples, many pictures, much explicit code, many proofs, extensive complexity results, many of them in closed formula forms, etc. With so many goodies it lacks an index.

183. **Scott**, Elizabeth and Johnstone, Adrian. Table based parsers with reduced stack activity. Technical Report CSD-TR-02-08, CS Dept., Royal Holloway, University of London, London, May 2003. Proofs, examples and background information for Johnstone and Scott [181].

184. **Johnstone**, Adrian, Scott, Elizabeth, and Economopoulos, Giorgios R. Generalised parsing: Some costs. In *Compiler Construction: 13th International Conf. CC'2004*, volume 2985 of *Lecture Notes in Computer Science*, pages 89–103, Berlin, 2004. Springer-Verlag. Several GLR techniques are compared experimentally and the effects found are discussed. The answers depend on many factors, including available memory size; for present-day grammars and machines RNGLR is a good choice.

185. **Johnstone**, Adrian and Scott, Elizabeth. Recursion engineering for reduction incorporated parsers. *Electr. Notes Theor. Comput. Sci.*, 141(4):143–160, 2005. Reduction-incorporated parsers require the grammar to be split in a regular part and a set of recursive non-terminals, where we want the regular part to be large and the recursive part to be small. We can make the regular part larger and larger by substituting out more and more non-terminals. The tables that correspond to optimum parsing speed can be enormous, and trade-offs have to be made. Heuristics, profiling, and manual intervention are considered, the latter based on the visualization tool *VCG*.

186. **Scott**, Elizabeth and Johnstone, Adrian. Generalised bottom up parsers with reduced stack activity. *Computer J.*, 48(5):565–587, 2005. The Reduction Incorporated (RI) technique from Johnstone and Scott [181] and Scott and Johnstone [183] is incorporated in a table-driven bottom-up parser, yielding a "shared packed parse forest" (SPPF). Run-time data structures can be an order of magnitude or more smaller than those of a GSS implementation. Extensive implementation code, proofs of correctness, efficiency analyses.

187. **Johnstone**, Adrian, Scott, Elizabeth, and Economopoulos, Giorgios R. Evaluating GLR parsing algorithms. *Sci. Comput. Progr.*, 61(3):228–244, 2006. A clear exposition of two improvements of Nozohoor-Farshi's modification [167] to Tomita's algorithm, the Right Nulled GLR (RNGLR) algorithm [182], and the Binary Right Nulled GLR (BRNGLR) algorithm [182] is followed by an extensive comparison of these methods, using LR(0), SLR(1) and LR(1) tables for grammars for C, Pascal and Cobol. The conclusion is that Right Nulled GLR (RNGLR) with an SLR(1) table performs adequately except in bizarre cases.

188. **Scott**, Elizabeth and Johnstone, Adrian. Right nulled GLR parsers. *ACM Trans. Prog. Lang. Syst.*, 28(4):577–618, 2006. After a 9 page(!) history of parsing since the time that the parsing problem was considered solved (mid-1970s), the principles of GLR parsing and right-nulled LR(1) (RN) parsing (Scott [100, 177]) are explained and combined in the RNGLR algorithm. The resulting recognizer is then extended to produce parse trees. Depending on the nature of the grammar, using right-nulled LR(1) can help considerably: on one grammar RNGLR visits only 25% of the edges visited by the standard GLR algorithm. Extensive implementation code, proofs of correctness, efficiency analyses.

18.2.2 Non-Canonical Parsing

This section covers the bottom-up non-canonical methods; the top-down ones (LC, etc.) are collected in (Web)Section 18.1.5.

189. **Floyd**, Robert W. Bounded context syntax analysis. *Commun. ACM*, 7(2):62–67, Feb. 1964. For each right-hand side of a rule $A \rightarrow \alpha$ in the grammar, enough left and/or right context

is constructed (by hand) so that when α is found obeying that context in a sentential form in a left-to-right scan in a bottom-up parser, it can safely be assumed to be the handle. If you succeed, the grammar is *bounded-context*. A complicated set of rules is given to check if you have succeeded. See [117] for the bounded-right-context part.

190. **Colmerauer**, Alain. *Précedence, analyse syntactique et languages de programmation*. PhD thesis, Technical report, Université de Grenoble, Grenoble, 1967, (in French). Defines total precedence and left-to-right precedence. See [119].

191. **Colmerauer**, Alain. Total precedence relations. *J. ACM*, 17(1):14–30, Jan. 1970. The non-terminal resulting from a reduction is not put on the stack but pushed back into the input stream; this leaves room for more reductions on the stack. This causes precedence relations that differ considerably from simple precedence.

192. **Szymanski**, T. G. *Generalized Bottom-up Parsing*. PhD thesis, Technical Report TR73-168, Cornell University, Ithaca, N.Y., 1973. For convenience derivation trees are linearized by, for each node, writing down its linearized children followed by a token $\texttt{]}_{\texttt{n}}$, where $\texttt{n}$ is the number of the production rule. For a given grammar G all its sentential forms form a language in this notation: G's *description language*. Define a "phrase" as a node that has only leaves as children. Now suppose we delete from derivation trees all nodes that are not phrases, and linearize these. This results in the *phrase language* of G. The point is that phrases can be reduced immediately, and consequently the phrase language contains all possibilities for immediate reduces.
Phrase languages are a very general model for bottom-up parsing. Consider a phrase P in a phrase language. We can then compute the left and right contexts of P, which turn out to be CF languages. The construct consisting of the left context of P, P, and right context of P is a *parsing pattern* for P. A complete set of mutually exclusive parsing patterns G is a *parsing scheme* for G. It is undecidable if there is a parsing scheme for a given grammar.
The problem can be made manageable by putting restrictions on the parsing patterns. Known specializations are bounded-right-context (Floyd [117]), LR(k) (Knuth [52]), LR-regular (Čulik, II and Cohen [57]), and bounded-context parsable (Williams [193]). New specializations discussed in this thesis are *FPFAP(k)*, where regular left and right contexts are maintained and used in a left-to-right scan with a k-token look-ahead; LR(k,∞) and LR(k,t), in which the left context is restricted to that constructed by LR parsing; and *RPP*, *Regular Pattern Parsable*, which is basically FPFAP(∞).
The rest (two-thirds) of the thesis explores these new methods in detail. LR(k) and SLR(k) are derived as representations of inexact-context parsing. A section on the comparison of these methods as to grammars and languages and a section on open problems conclude the thesis.

193. **Williams**, John H. Bounded-context parsable grammars. *Inform. Control*, 28(4):314–334, Aug. 1975. The bounded-context parser without restrictions on left and right context, hinted at by Floyd [189], is worked out in detail; grammars allowing it are called bounded-context parsable, often abbreviated to BCP. All LR languages are BCP languages but not all LR grammars are BCP grammars. BCP grammars allow, among others, the parsing in linear time of some non-deterministic languages. Although a parser could be constructed, it would not be practical.

194. **Szymanski**, Thomas G. and Williams, John H. Non-canonical extensions of bottom-up parsing techniques. *SIAM J. Computing*, 5(2):231–250, June 1976. Theory of non-canonical versions of several bottom-up parsing techniques, with good informal introduction. Condensation of Szymanski's thesis [192].

195. **Friede**, Dietmar. Transition diagrams and strict deterministic grammars. In Klaus Weihrauch, editor, *4th GI-Conference*, volume 67 of *Lecture Notes in Computer Science*, pages 113–123, Berlin, 1978. Springer-Verlag. Explores the possibilities to parse strict deterministic grammars (a large subset of LR(0)) using transition diagrams, which are top-down. This leads to PLL(k) grammars, which are further described in Friede [196].

196. **Friede**, Dietmar. Partitioned LL(k) grammars. In H.A. Maurer, editor, *Automata, Languages and Programming*, volume 71 of *Lecture Notes in Computer Science*, pages 245–255. Springer-Verlag, Berlin, 1979. The left factorization, usually performed by hand, which

turns a rule like $A \rightarrow PQ|PR$ into $A \rightarrow PZ$; $Z \rightarrow Q|R$, is incorporated into the parsing algorithm in a very general and recursive way. This results in the PLL(k) grammars and their languages. The resulting grammars are more like LC and LR grammars than like LL grammars. Many theorems, some surprising, about these grammars and languages are proved; examples are: 1. the PLL(1) grammars include the LL(1) grammars. 2. the PLL(0) grammars are exactly the strict deterministic grammars. 3. the classes of PLL(k) languages are all equal for $k > 0$. 4. the PLL(0) languages form a proper subset of the PLL(1) languages. Theorems (2), (3) and (4) also hold for LR, but a PLL parser is much simpler to construct.

197. **Tai**, Kuo-Chung. Noncanonical SLR(1) grammars. *ACM Trans. Prog. Lang. Syst.*, 1(2):295–320, Oct. 1979. An attempt is made to solve reduce/reduce conflicts by postponing the decision, as follows. Suppose two reduce items $A \rightarrow \alpha\bullet$ and $B \rightarrow \beta\bullet$ with overlapping look-aheads in an item set I. The look-ahead for the A item is replaced by LM_FOLLOW(A), the set of non-terminals that can follow A in any *leftmost* derivation, same for the look-ahead of B, and all initial items for these non-terminals are added to I. Now I will continue to try to recognize the above non-terminals, which, once found can be used as look-ahead non-terminals to resolve the original reduce/reduce conflict. This leads to two non-canonical parsing methods LSLR(1) and NSLR(1), which differ in details.

198. **Proudian**, Derek and Pollard, Carl J. Parsing head-driven phrase structure grammar. In *23rd Annual Meeting of the Association for Computational Linguistics*, pages 167–171, 1985. The desirability of starting analysis with the "head" of a phrase is argued on linguistic grounds. Passing of features between parents and children is automatic, allowing a large part of English to be represented by 16 rules only. Parsing is chart parsing, in which the order in which edges are added is not left-to-right, but rather controlled by head information and the unification of features of children.

199. **Kay**, Martin. Head-driven parsing. In *International Workshop on Parsing Technologies*, pages 52–62, 1989. Since the complements of a non-terminal (= the structures it governs in linguistic terms) are often more important than textual adjacency, it is logical and profitable to parse first the section that supplies the most information. This is realized by appointing one of the members in each RHS as the "head". Parsing then starts by finding the head of the head etc. of the start symbol; usually it is a verb form which then gives information about its subject, object(s), etc. Finding the head is awkward, since it may be anywhere in the sentence. A non-directional chart parser is extended with three new types of arcs, pending, current and seek, which assist in the search. Also, a Prolog implementation of an Unger parser is given which works on a grammar in 2-form: if the head is in the first member of an alternative, searching starts from the left, otherwise from the right. The advantages of head-driven parsing are conceptual; the author expects no speed-up.

200. **Salomon**, Daniel J. and Cormack, Gordon V. Scannerless NSLR(1) parsing of programming languages. *ACM SIGPLAN Notices*, 24(7):170–178, July 1989. The traditional CF syntax is extended with two rule types: an exclusion rule $A \nrightarrow B$, which means that any sentential form in which A generates a terminal production of B (with B regular) is illegal; and an adjacency restriction $A \not\leftarrow B$ which means that any sentential form in which terminal productions of A and B are adjacent, is illegal. The authors show that the addition of these two types of rules allows one to incorporate the lexical phase of a compiler into the parser. The system uses a non-canonical SLR(1) parser.

201. **Satta**, Giorgio and Stock, Oliviero. Head-driven bidirectional parsing: A tabular method. In *International Workshop on Parsing Technologies*, pages 43–51, 1989. The Earley algorithm is adapted to head grammars, as follows. A second dot is placed in each Earley item for a section $w_{m,n}$ of the input, not coinciding with the first dot, with the meaning that the part between the dots produces $w_{m,n}$. Parsing no longer proceeds from left to right but according to an action pool, which is prefilled upon initialization, and which is processed until empty.
The initialization creates all items that describe terminal symbols in the input that are heads in any production rule. Processing takes one item from the action pool, and tries to perform five actions on it, in arbitrary order: extend the left dot in an uncompleted item to the left, likewise to the right,

use the result of the completed item to extend a left dot to the left, likewise to the right, and use a completed item to identify a new head in some production rule.
The presented algorithm contains an optimization to prevent subtrees to be recognized twice, by extending left then right, and by extending right then left.

202. **Hutton**, Michael D. Noncanonical extensions of LR parsing methods. Technical report, University of Waterloo, Waterloo, Aug. 1990. After a good survey of existing non-canonical methods, the author sets out to create a non-canonical LALR(k) (*NLALR(k)*) parser, analogous to Tai's NSLR(1) from SLR(1), but finds that it is undecidable if a grammar is NLALR(k). The problem is solved by restricting the number of postponements to a fixed number t, resulting in *NLALR(k,t)*, also called *LALR(k,t)*.

203. **Nederhof**, M.-J. and Satta, G. An extended theory of head-driven parsing. In *32nd Annual Meeting of the Association for Computational Linguistics*, pages 210–217, June 1994. The traditional Earley item $[A \to \alpha\bullet\beta, i]$ in column j is replaced by a position-independent double-dotted item $[i,k,A \to \alpha\bullet\gamma\bullet\beta, m, p]$ with the meaning that a parsing of the string $a_{i+1}\cdots a_p$ by $A \to \alpha\gamma\beta$ is sought, where γ already produces $a_{k+1}\cdots a_m$. This collapses into Earley by setting $\alpha = \varepsilon$, $k = i$, $p = n$ where n is the length of the input string, and putting the item in column m; the end of the string sought in Earley parsing is not known, so $p = n$ can be omitted.
Using these double-dotted items, Earley-like algorithms are produced basing the predictors on top-down parsing, head-corner parsing (derived from left-corner), predictive head-infix (HI) parsing (derived from predictive LR), extended HI parsing (derived from extended LR), and HI parsing (extended from LR), all in a very compact but still understandable style. Since head parsing is by nature partially bottom-up, ε-rules are a problem, and the presented algorithms do not allow them.
Next, head grammars are generalized by requiring that the left and right parts around the head again have sub-heads, and so on recursively. A parenthesized notation is given: $S \to ((c)A(b))s$, in which s is the head, A the sub-head of the left part, etc. The above parsing algorithm is extended to these generalized head grammars. Correctness proofs are sketched.

204. **Sikkel**, K. and Akker, R. op den. Predictive head-corner chart parsing. In Harry Bunt and Masaru Tomita, editors, *Recent Advances in Parsing Technology*, pages 113–132. Kluwer Academic Publishers, Dordrecht, 1996. Starting from the start symbol, the heads are followed down the grammar until a terminal t is reached; this results in a "head spine". This terminal is then looked up in the input, and head spines are constructed to each position p_i at which t occurs. Left and right arcs are then predicted from each spine to p_i, and the process is repeated recursively for the head of the left arc over the segment $1..p_i - 1$ and for the head of the right arc over the segment $p_i + 1..n$.

205. **Noord**, Gertjan van. An efficient implementation of the head-corner parser. *Computational Linguistics*, 23(3):425–456, 1997. Very carefully reasoned and detailed account of the construction of a head-corner parser in Prolog, ultimately intended for speech recognition. Shows data from real-world experiments. The author points out that memoization is efficient for large chunks of input only.

206. **Madhavan**, Maya, Shankar, Priti, Rai, Siddharta, and Ramakrishna, U. Extending Graham-Glanville techniques for optimal code generation. *ACM Trans. Prog. Lang. Syst.*, 22(6):973–1001, Nov. 2000. (Parsing part only.) Classical Graham–Glanville code generation is riddled by ambiguities that have to be resolved too early, resulting in sub-optimal code. This paper describes a parsing method which the authors do not seem to name, and which allows ambiguities in an LR-like parser to remain unresolved arbitrarily long.
The method is applicable to grammars that have the following property; no technical name for such grammars is given. All rules are either "unit rules" in which the right-hand side consists of exactly one non-terminal, or "operator rules" in which the right-hand side consists of N (≥ 0) non-terminals followed by a terminal, the "operator". As usual with operators, the operator has an arity, which has to be equal to N.
In such a grammar, each shift of a terminal is immediately followed by a reduce, and the arity of the terminal shifted determines the number of items on the stack that are replaced by one non-terminal.

This allows us to do the reduction, even if multiple reductions are possible, without keeping multiple stacks as is done in a GLR parser: all reduces take away the same number of stack items. Note that all the items on the stack are non-terminals. Such a reduction results in a set of non-terminals to be pushed on the stack, each with a different, possibly ambiguous parse tree attached to them. This set may then be extended by other non-terminals, introduced by unit reductions using unit rules; only when no further reduces are possible is the next terminal (= operator) shifted in.
A new automaton is constructed from the existing LR(0) automaton, based on the above parsing algorithm; the unit reductions have been incorporated completely into the automaton. The algorithm to do so is described extensively.
The parser is used to parse the intermediate code stream in a compiler and to isolate in it operator trees that correspond to machine instructions, the grammar rules. A cost is attached to each rule, and the costs are used to disambiguate the parse tree and so decide on the machine code to be generated.

207. **Farré**, Jacques and Fortes Gálvez, José. A basis for looping extensions to discriminating-reverse parsing. In *5th Internat. Conf. Implementation and Applications of Automata, CIAA 2000*, volume 2088 of *Lecture Notes in Computer Science*, pages 122–134, 2001. Since DR parsers require only a small top segment of the stack, they can easily build up enough left context after a DR conflict to do non-canonical DR (NDR). When a conflict occurs, a state-specific marker is pushed on the stack, and input symbols are shifted until enough context is assembled. Then DR parsing can resume normally on the segment above the marker. The shift strategy is guided by a mirror image of the original DR graph. This requires serious needlework to the graphs, but complete algorithms are given. This technique shows especially clearly that non-canonical parsing is actually doing a CF look-ahead.

208. **Farré**, Jacques and Fortes Gálvez, José. Bounded-graph construction for noncanonical discriminating-reverse parsers. In *6th Internat. Conf. Implementation and Applications of Automata, CIAA 2001*, volume 2494 of *Lecture Notes in Computer Science*, pages 101–114. Springer Verlag, 2002. Improvements to the graphs construction algorithm in [207].

209. **Farré**, Jacques and Fortes Gálvez, J. Bounded-connect noncanonical discriminating-reverse parsers. *Theoret. Comput. Sci.*, 313(1):73–91, 2004. Improvements to [208]. More theory of non-canonical DR parsers, defining BC(h)DR(0).

18.2.3 Substring Parsing

210. **Lang**, Bernard. Parsing incomplete sentences. In D. Vargha, editor, *12th International Conf. on Comput. Linguistics COLING'88*, pages 365–371. Association for Computational Linguistics, 1988. An incomplete sentence is a sentence containing one or more unknown symbols (represented by `?`) and/or unknown symbol sequences (represented by `*`). General left-to-right CF parsers can handle these inputs as follows. Upon seeing `?` make transitions on all possible input symbols while moving to the next position; upon seeing `*` make transitions on all possible input symbols while staying at the same position. The latter process requires transitive closure.
These features are incorporated into an all-paths non-deterministic interpreter of pushdown transducers. This PDT interpreter accepts transitions of the form $(p\ A\ a\ \rightarrow\ q\ B\ u)$, where p and q are states, A and B stack symbols, a is an input token, and u is an output token, usually a number of a production rule. A, B, a and/or u may be missing, and the input may contain wild cards. Note that these transitions can push and pop only one stack symbol at a time; transitions pushing or popping more than one symbol have to be decomposed. The interpretation is performed by constructing sets of Earley-like items between successive input tokens; these items then form the non-terminals of the output grammar. Given the form of the allowed transitions, the output grammar is automatically in 2-form, but may contain useless and unreachable non-terminals. The grammar produces the input string as many times as there are ambiguities, interlaced with output tokens which tell how the preceding symbols must be reduced, thus creating a genuine parse tree.

Note that the "variation of Earley's algorithm" from the paper is not closely related to Earley, but is rather a formalization of generalized LR parsing. Likewise, the items in the paper are only remotely related to Earley items. The above transitions on `?` and `*` are, however, applicable independent of this.

211. **Cormack**, Gordon V. An LR substring parser for noncorrecting syntax error recovery. *ACM SIGPLAN Notices*, 24(7):161–169, July 1989. The LR(1) parser generation method is modified to include suffix items of the form $A \rightarrow \cdots \bullet \beta$, which mean that there exists a production rule $A \rightarrow \alpha\beta$ in the grammar and that it can be the handle, provided we now recognize β. The parser generation starts from a state containing all possible suffix items, and proceeds in LR fashion from there, using fairly obvious shift and reduce rules. If this yields a deterministic parser, the grammar was BC-LR(1,1); it does so for any bounded-context(1,1) grammar, thereby confirming Richter's [313] conjecture that linear-time suffix parsers are possible for BC grammars. The resulting parser is about twice as large as an ordinary LR parser. A computationally simpler BC-SLR(1,1) variant is also explained. For the error recovery aspects see the same paper [318].

212. **Rekers**, Jan and Koorn, Wilco. Substring parsing for arbitrary context-free grammars. *ACM SIGPLAN Notices*, 26(5):59–66, May 1991. A GLR parser is modified to parse substrings, as follows. The parser is started in all LR states that result from shifting over the first input symbol. Shifts are handled as usual, and so are reduces that find all their children on the stack. A reduce to $A \rightarrow \alpha$, where A contains more symbols than the stack can provide, adds all states that can be reached by a shift over A. A technique is given to produce trees for the completion of the substring, to be used, for example, in an incremental editor.

213. **Rekers**, J. *Parser Generation for Interactive Environments*. PhD thesis, Technical report, Leiden University, Leiden, 1992. Same as [347]. Chapter 4 discusses the substring parser from [212].

214. **Bates**, Joseph and Lavie, Alon. Recognizing substrings of LR(k) languages in linear time. *ACM Trans. Prog. Lang. Syst.*, 16(3):1051–1077, 1994. Reporting on work done in the late 1970s, the authors show how a GLR parser can be modified to run in linear time when using a conflict-free LR table. Basically, the algorithm starts with a GLR stack configuration consisting of all possible states, and maintains as large a right hand chunk of the GLR stack configuration as possible. This results in a forest of GLR stack configurations, each with a different state at the root; each path from the root is a top segment of a possible LR stack, with the root as top of stack. For each token, a number of reduces is performed on all trees in the forest, followed by a shift, if possible. Then all trees with equal root states are merged. If a reduce $A \rightarrow \alpha$ reduces more stack than is available, new trees result, each consisting of a state that allows a shift on A. When two paths are merged, the shorter path wins, since the absence of the rest of a path implies all possible paths, which subsumes the longer path. Explicit algorithm and proofs are given. See Goeman [218] for an improved version.

215. **Bertsch**, Eberhard. An asymptotically optimal algorithm for non-correcting LL(1) error recovery. Technical Report 176, Ruhr-Universität Bochum, Bochun, Germany, April 1994. First a suffix grammar G_S is created from the LL(1) grammar G. Next G_S is turned into a left-regular grammar L by assuming its CF non-terminals to be terminals; this regular grammar generates the "root set" of G. Then a linear method is shown to fill in the recognition table in linear time, by doing tabular LL(1) parsing using grammar G. Now all recognizable non-terminals in the substring are effectively terminals, but of varying size. Next a second tabular parser is explained to parse the non-terminals according to the left-recursive grammar L; it is again linear. Finally the resulting suffix parser is used to do non-correcting error recovery.

216. **Nederhof**, Mark-Jan and Bertsch, Eberhard. Linear-time suffix parsing for deterministic languages. *J. ACM*, 43(3):524–554, May 1996. Shows that an Earley parser working with a conflict-free LL(1) parse table runs in linear time. Next extends this result to suffix parsing with an Earley parser. The LR case is more complicated. The language is assumed to be described by a very restricted pushdown automaton, rather than by a CF grammar. Using this automaton in suffix parsing with an Earley parser rather than an LL(1) parse table results in an $O(n^2)$ algorithm. To

avoid this the automaton is refined so it consumes a token on every move. The Earley suffix parser using this automaton is then proven to be linear. Several extensions and implementation ideas are discussed. See Section 12.3.3.2.

217. **Ruckert**, Martin. Generating efficient substring parsers for BRC grammars. Technical Report 98-105, State University of New York at New Paltz, New Paltz, NY 12561, July 1998. All BRC(m,n) parsing patterns are generated and subjected to Floyd's [117] tests; BRC(m,n) (bounded-right-context) is BCP(m,n) with the right context restricted to terminals. If a complete set remains, then for every correct sentential form there is at least one pattern which identifies a handle; this handle is not necessarily the leftmost one, so the parser is non-canonical – but it is linear. Since this setup can start parsing anew wherever it wants to, it identifies correct substrings in a natural way, if the sentential form is not correct. Heuristics are given to improve the set of parsing patterns. The paper is written in a context of error recovery.

218. **Goeman**, Heiko. On parsing and condensing substrings of LR languages in linear time. *Theoret. Comput. Sci.*, 267(1-2):61–82, 2001. Tidied-up version of Bates and Lavie's algorithm [214], with better code and better proofs. The algorithm is extended with memoization, which condenses the input string as it is being parsed, thus increasing reparsing speed.

18.2.4 Parsing as Intersection

219. **Bar-Hillel**, Y., Perles, M., and Shamir, E. On formal properties of simple phrase structure grammars. *Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung*, 14:143–172, 1961. The intersection of a CF grammar and a FS automaton is constructed in a time $O(n^d+1)$, where n is the number of states in the automaton, and d is the maximum length of the RHSs in the grammar. For more aspects of the paper see [386].

220. **Lang**, Bernard. A generative view of ill-formed input processing. In *ATR Symposium on Basic Research for Telephone Interpretation*, Dec. 1989. Proposes weighted grammars (à la Lyon [294]) for the treatment of various ill or problematically formed input, among which word lattices. A word lattice is a restricted form of FSA, but even general FSAs may appear as input when sequences of words are missing or partially identified. The author notes in passing that "the parsing of an FSA A according to a CF grammar G can produce a new grammar T for the intersection of the languages $\mathcal{L}(A)$ and $\mathcal{L}(G)$, giving to all sentences in that intersection the same structure as the original grammar G".

221. **Noord**, Gertjan van. The intersection of Finite State Automata and Definite Clause Grammars. In *33rd Annual Meeting of the Association for Computational Linguistics*, pages 159–165, June 1995. Mainly about DCGs, but contains a short but useful introduction to parsing as intersection.

222. **Albro**, Daniel M. Taking primitive optimality theory beyond the finite state. Technical report, Linguistics Department UCLA, 2000. Primitive optimality theory concerns the creation of a set of acceptable surface representations from an infinite source of underlying representations; acceptability is defined by a series of constraints. The set of representations is implemented as an FS machine, the constraints as weighted FS machines. The representation M generated by the infinite source is passed through each of the constraint machines and the weights are the penalties it incurs. After each constraint machine, all non-optimal paths are removed from M. All this can be done very efficiently, since FS machines can be intersected easily.
The paper proposes to increase the power of this system by allowing CF and multiple context-free grammars (Seki et al. [272]) as the representation; intersection with the constraining FS machines is still possible. It is trivial to extend an Earley parser to do this intersection job, but it just yields a set of sets of items, and a 50-lines algorithm to retrieve the new intersection grammar from these data is required and given in the paper. Further extensions of the intersecting Earley parser include the handling of the weights and the adaptation to MCF grammars. Fairly simple techniques suffice in all three cases.

18.2.5 Parallel Parsing Techniques

223. **Fischer**, Charles N. On parsing context-free languages in parallel environments. Technical Report 75-237, Cornell University, Ithaca, New York, April 1975. Introduces the parallel parsing technique discussed in Section 14.2. Similar techniques are applied for LR parsing and precedence parsing, with much theoretical detail.

224. **Brent**, R. P. and Goldschlager, L. M. A parallel algorithm for context-free parsing. *Australian Comput. Sci. Commun.*, 6(7):7.1–7.10, 1984. Is almost exactly the algorithm by Rytter [226], except that its recognize phase also does a plain CYK combine step, and their propose step is a bit more complex. Also proves $^2 \log n$ efficiency on $O(n^6)$ nodes. Suggests that it can be done on $O(n^4.9911)$ nodes, depending on Boolean matrix multiplication of matrices of size $O(n^2) \times O(n^2)$.

225. **Bar-On**, Ilan and Vishkin, Uzi. Optimal parallel generation of a computation tree form. *ACM Trans. Prog. Lang. Syst.*, 7(2):348–357, April 1985. Describes an optimal parallel algorithm to find a computation tree form from a general arithmetic expression. The heart of this algorithm consists of a parentheses-matching phase which solves this problem in time $O(\log\ n)$ using $n/\log\ n$ processors, where n is the number of symbols in the expression. First, the expression is split up into $n/\log\ n$ successive segments of length $\log\ n$, and each segment is assigned to a processor. Each processor then finds the pairs of matching parentheses in its own segment using a stack. This takes $O(\log\ n)$ time. Next, a binary tree is used to compute the nesting level of the left-over parentheses, and this tree is used to quickly find matching parentheses.

226. **Rytter**, Wojciech. On the recognition of context-free languages. In Andrzej Skowron, editor, *Computation Theory*, volume 208 of *Lecture Notes in Computer Science*, pages 318–325. Springer Verlag, Berlin, 1985. Describes the Rytter chevrons, which are represented as a pair of parse trees: the pair $((A,i,j),(B,k,l))$ is "realizable" if $A \stackrel{*}{\rightarrow} w[i\ldots k]Bw[l\ldots j]$, where $w[1\ldots n]$ is the input. The author then shows that using these chevrons, one can do CFL recognition in $O(\log^2 n)$ time on certain parallel machines using $O(n^6)$ processors; the dependence on the grammar size is not indicated. The paper also shows that the algorithm can be simulated on a multihead 2-way deterministic pushdown automaton in polynomial time.

227. **Rytter**, Wojciech. Parallel time $O(\log n)$ recognition of unambiguous CFLs. In *Fundamentals of Computation Theory*, volume 199 of *Lecture Notes in Computer Science*, pages 380–389. Springer Verlag, Berlin, 1985. Uses the Rytter chevrons as also described in [226] and shows that the resulting recognition algorithm can be executed in $O(\log n)$ time on a parallel W-RAM, which is a parallel random access machine which allows simultaneous reads and also simultaneous writes provided that the same value is written.

228. **Chang**, J. H., Ibarra, O. H., and Palis, M. A. Parallel parsing on a one-way array of finite-state machines. *IEEE Trans. Comput.*, 36:64–75, 1987. Presents a very detailed description of an implementation of the CYK algorithm on a one-way two-dimensional array of finite-state machines, or rather a 2-DSM, in linear time.

229. **Yonezawa**, Akinori and Ohsawa, Ichiro. Object-oriented parallel parsing for context-free grammars. In *COLING-88: 12th International Conference on Computational Linguistics*, pages 773–778, Aug. 1988. The algorithm is distributed bottom-up. For each rule $A \rightarrow BC$, there is an agent (object) which receives messages containing parse trees for Bs and Cs which have just been discovered, and, if the right end of B and the left end of C are adjacent, constructs a parse tree for A and sends it to every agent who manages a rule that has A as its RHS. ε-rules and circularities are forbidden.

230. **Srikant**, Y. N. Parallel parsing of arithmetic expressions. *IEEE Trans. Comput.*, 39(1):130–132, 1990. This short paper presents a parallel parsing algorithm for arithmetic expressions and analyzes its performance on different types of models of parallel computation. The parsing algorithm works in 4 steps:

1. Parenthesize the given expression fully.
2. Delete redundant parameters.
3. Separate the sub-expressions at each level of parenthesis nesting and determine the root of the tree form of each sub-expression in parallel.
4. Separate the sub-expressions at each level of parenthesis nesting and determine the children of each operator in the tree form of each sub-expression in parallel.

The algorithm takes $O(\sqrt{n})$ on a mesh-connected computer, and $O(\log^2 n)$ on other computation models.

231. **Alblas**, Henk, Nijholt, Anton, Akker, Rieks op den, Oude Luttighuis, Paul, and Sikkel, Klaas. An annotated bibliography on parallel parsing. Technical Report INF 92-84, University of Twente, Enschede, The Netherlands, Dec. 1992. Introduction to parallel parsing covering: lexical analysis, parsing, grammar decomposition, string decomposition, bracket matching, miscellaneous methods, natural languages, complexity, and parallel compilation; followed by an annotated bibliography of about 200 entries.

232. **Janssen**, W., Poel, M., Sikkel, K., and Zwiers, J. The primordial soup algorithm: A systematic approach to the specification of parallel parsers. In *Fifteenth International Conference on Computational Linguistics*, pages 373–379, Aug. 1992. Presents a general framework for specifying parallel parsers. The soup consists of partial parse trees that can be arbitrarily combined. Parsing algorithms can be described by specifying constraints in the way trees can be combined. The paper describes the mechanism for a.o. CYK and bottom-up Earley (BUE), which is Earley parsing without the top-down filter. Leaving out the top-down filter allows for parallel bottom-up, rather than left-to-right processing. The mechanism allows the specification of parsing algorithms without specifying flow control or data structures, which gives an abstract, compact, and elegant basis for the design of a parallel implementation.

233. **Sikkel**, Klaas and Lankhorst, Marc. A parallel bottom-up Tomita parser. In Günther Görz, editor, *1. Konferenz "Verarbeitung Natürlicher Sprache" - KONVENS'92*, Informatik Aktuell, pages 238–247. Springer-Verlag, Oct. 1992. Presents a parallel bottom-up GLR parser that can handle any CF grammar. Removes the left-to-right restriction and introduces processes that parse the sentence, starting at every position in the input, in parallel. Each process yields the parts that start with its own word. The processes are organized in a pipeline. Each process sends the completed parts that it finds and the parts that it receives from his right neighbor to his left neighbor, who combines the parts that it receives with the parts that it already found to create new parts. It uses a simple pre-computed parsing table and a graph-structured stack (actually tree-structured) in which (partially) recognized parts are stored. Empirical results indicate that parallelization pays off for sufficiently long sentences, where "sufficiently long" depends on the grammar. A sequential Tomita parser is faster for short sentences. The algorithm is discussed in Section 14.3.1.

234. **Alblas**, Henk, Akker, Rieks op den, Oude Luttighuis, Paul, and Sikkel, Klaas. A bibliography on parallel parsing. *ACM SIGPLAN Notices*, 29(1):54–65, 1994. A modified and compacted version of the bibliography by Alblas et al. [231].

235. **Hendrickson**, Kenneth J. A new parallel LR parsing algorithm. In *ACM Symposium on Applied Computing*, pages 277–281. ACM, 1995. Discusses the use of a *marker-passing* computational paradigm for LR parsing. Each state in the LR parsing table is modeled as a node with links to other nodes, where the links represent state transitions. All words in the input sentences are broadcast to all nodes in the graph, acting as *activation* markers. In addition, each node has a *data* marker specifying which inputs are legal for shifting a token and/or which reduction to use. The parsing process is then started by placing an initial *prediction* marker for each sentence on the start node. When a prediction marker arrives at a node, it will collide with the activation markers at that node, provided they are at the same position in the same sentence. The result of such a collision is determined by the data marker at that node which may specify reductions and/or shifts, which are handled sequentially, resulting in new prediction markers which are sent to their destination node.

236. **Ra**, Dong-Yul and Kim, Jong-Hyun. A parallel parsing algorithm for arbitrary context-free grammars. *Inform. Process. Lett.*, 58(2):87–96, 1996. A parallel parsing algorithm based on Earley's algorithm is proposed. The Earley items construction phase is parallelized by assigning a processor to each position in the input string. Each processor i then performs n stages: stage k consists of the computation of all Earley items of "length" k which start at position i. After each stage, the processors are synchronized, and items are transferred. It turns out that this only requires data transfer from processor $i+1$ to processor i. Any items that processor i needs from processor $i+m$ are obtained by processor $i+1$ at stage $m-1$. When not enough processors are available ($p < n$), a stage is divided into $\lceil n/p \rceil$ phases, such that processor i computes all items starting at positions i, $i+p$, $i+2p$, et cetera. If in the end processor 0 found an item $\texttt{S} \rightarrow \alpha \bullet \texttt{,0,}n$, the input string is recognized.
To find a parse, each processor processes requests to find a parse for completed items (i.e. the dot is at the end of the right hand side) that it created. In processing such a request, the processor generates requests to other processors. Now processor 0 is asked $\texttt{S} \rightarrow \alpha \bullet \texttt{,0,}n$.
A very detailed performance analysis is given, which shows that the worst case performance of the algorithm is $O(n^3/p)$ on p processors.

18.2.6 Non-Chomsky Systems

237. **Koster**, Cornelis H. A. and Meertens, Lambert G. L. T. Basic English, a generative grammar for a part of English. Technical report, Euratom Seminar "Machine en Talen" of E.W. Beth, University of Amsterdam, 1962. [2]

238. **McClure**, R. M. TMG: A syntax-directed compiler. In *20th National Conference*, pages 262–274. ACM, 1965. A transformational grammar system in which the syntax is described by a sequence of parsing routines, which can succeed, and then may absorb input and produce output, or fail, and then nothing has happened; this requires backtracking. Each routine consists of a list of possibly labeled calls to other parsing routines of the form <*routine_name/failure_label*>. If the called routine succeeds, the next call in the list is performed; if it fails, control continues at the *failure_label*. An idiom for handling left recursion is given. This allows concise formulation of many types of input. Rumor has it that TMG stands for "transmogrify", but "transformational grammar" is equally probable.

239. **Gilbert**, Philip. On the syntax of algorithmic languages. *J. ACM*, 13(1):90–107, Jan. 1966. Unlike Chomsky grammars, which are production devices, an "analytic grammar" is a recognition device: membership of the language is decided by an algorithm based on the analytic grammar. An analytic grammar is a set of reduction rules, which are Chomsky Type 1 production rules in reverse, plus a scan function S. An example of a reduction rule is $\texttt{abcde} \rightarrow \texttt{aGe}$, which reduces $\texttt{bcd}$ to $\texttt{G}$ in the context $\texttt{a} \cdots \texttt{e}$.
A string belongs to the language if it can be reduced to the start symbol by applying reduction rules, such that the position of each reduction in the sentential form is allowed by the scan function. Reduction can never increase the length of the sentential form, so if we avoid duplicate sentential forms, this process always terminates. So an analytic grammar recognizes a recursive set. The author also proves that for every recursive set there is analytic grammar which recognizes it; this may require complicated scan functions.
Two examples are given: Hollerith constants, and declaration and use of identifiers. There seem to be no further publications on analytic grammars.

240. **Hotz**, G. Erzeugung formaler Sprachen durch gekoppelte Ersetzungen. In F.L. Bauer and K. Samelson, editors, *Kolloquium über Automatentheorie und formale Sprachen*, pages 62–73. TU Munich, 1967, (in German). Terse and cryptic paper in which the components of Chomsky's grammars and its mechanism are generalized into an X-category, consisting of

[2] It is to be feared that this paper is lost. Any information to the contrary would be most welcome.

an infinite set of sentential forms, a set of functions that perform substitutions, a set of sources (left-hand sides of any non-zero length), a set of targets (right-hand sides of any length), an inference operator (for linking two substitutions), and a concatenation operator. By choosing special forms for the functions and the operators and introducing a number of homomorphisms this mechanism is used to define coupled production. Using theorems about X-categories it is easy to prove that the resulting languages are closed under union, intersection and negation. Many terms not explained; no examples given.

241. **Sintzoff**, M. Existence of a van Wijngaarden syntax for every recursively enumerable set. *Annales de la Société Scientifique de Bruxelles*, 81(II):115–118, 1967. A relatively simple proof of the theorem that for every semi-Thue system we can construct a VW grammar that produces the same set.

242. **Friš**, Ivan. Grammars with partial ordering of the rules. *Inform. Control*, 12:415–425, 1968. The CF grammars are extended with restrictions on the production rules that may be applied in a given production step.
One restriction is to have a partial order on the production rules and disallow the application of a production rule if a smaller (under the partial ordering) rule could also be applied. This yields a language class in between CF and CS which includes $a^n b^n c^n$, but the author uses 26(!) rules and 15 orderings to pull this off.
Another restriction is to require the control word of the derivation to belong to a given regular language. This yields exactly the CS languages. A formal proof is given, but no example.
For errata see "Inform. Control", 15(5):452-453, Nov. 1969.
(The *control word* of a derivation *D* is the sequence of the numbers of the production rules used in *D*, in the order of their application. The term is not used in this paper and is by Salomaa.)

243. **Knuth**, Donald E. Semantics of context-free languages. *Math. Syst. Theory*, 2(2):127–145, 1968. Introduces inherited attributes after acknowledging that synthesized attributes were already used by Irons in 1961. Shows how inherited attributes may simplify language description, mainly by localizing global effects. Gives a formal definition of attribute grammars and shows that they can express any expressible computation on the parse tree, by carrying around an attribute that represents the entire tree.
With having both synthesized and inherited attributes comes the danger of circularity of the attribute rules. An algorithm is given to determine that situation statically (corrected by the author in *Math. Syst. Theory*, **5**, 1, 1971, pp. 95-96.)
Next a simple but non-trivial language for programming a Turing machine called Turingol is defined using an attribute grammar. The full definition fits on one printed page. A comparison with other systems (Vienna Definition Language, etc.) concludes the paper.

244. **Wijngaarden**, A. van et al. Report on the algorithmic language ALGOL 68. *Numer. Math.*, 14:79–218, 1969. VW grammars found their widest application to date in the definition of ALGOL 68. Section 1.1.3 of the ALGOL 68 Revised Report contains a very carefully worded description of the two-level mechanism. The report contains many interesting applications.
See also [251].

245. **Koster**, C. H. A. Affix grammars. In J.E.L. Peck, editor, *ALGOL 68 Implementation*, pages 95–109. North-Holland Publ. Co., Amsterdam, 1971. Where attribute grammars have attributes, affix grammars have affixes, and where attribute grammars have evaluation functions affix grammars have them too, but the checks in an affix grammar are part of the grammar rather than of the evaluation rules. They take the form of primitive predicates, pseudo-non-terminals with affixes similar to the `where...` predicates in a VW grammar, which produce ε when they succeed, but block the production process when they fail. Unlike attribute grammars, affix grammars are production systems. If the affix grammar is "well-formed", a parser for it can be constructed.

246. **Birman**, Alexander and Ullman, Jeffrey D. Parsing algorithms with backtrack. *Inform. Control*, 23(1):1–34, 1973. Whereas a Chomsky grammar is a mechanisms for generating languages, which can, with considerable difficulty, be transformed into a parsing mechanism, a *TS* (*TMG recognition Scheme*), (McClure [238]) is a top-down parsing technique, which can, with far

less difficulty, be transformed into a language generation mechanism. Strings that are accepted by a given TS belong to the language of that TS.
A TS is a set of recursive routines, each of which has the same structure: *A* **`= if`** *recognize B* **`andif`** *recognize C* **`then`** *succeed* **`else`** *recognize D* **`fi`**, where each routine does backtracking when it returns failure; this models backtracking top-down parsing. This routine corresponds to the TS rule $A \rightarrow BC/D$.
The paper also introduces *generalized TS* (*gTS*), which has rules of the form $A \rightarrow B(C,D)$, meaning *A* **`= if`** *recognize B* **`then`** *recognize C* **`else`** *recognize D* **`fi`**. This formalism allows negation: **`return if`** *recognize* A_i **`then`** *fail* **`else`** *succeed* **`fi`**.
TS and gTS input strings can be recognized in one way only, since the parsing algorithm is just a deterministic program. TS and gTS languages can be recognized in linear time, as follows. There are $|V|$ routines, and each can be called in $n+1$ positions, where V is the set of non-terminals and n is the length of the input string. Since the results of the recognition routines depend only on the position at which they are started, their results can be precomputed and stored in a $|V| \times n$ matrix. A technique is shown by which this matrix can be computed from the last column to the first.
Since CF languages probably cannot be parsed in linear time, there are probably CF languages which are not TS or gTS, but none are known. [g]TS languages are closed under intersection (recognize by one TS, fail, and then recognize by the other TS), so there are non-CF languages which are [g]TS; $a^n b^n c^n$ is an example. Many more such properties are derived and proved in a heavy formalism.

247. **Lepistö**, Timo. On ordered context-free grammars. *Inform. Control*, 22(1):56–68, Feb. 1973. More properties of ordered context-free grammars (see Friš [242]) are given.

248. **Schuler**, P. F. Weakly context-sensitive languages as model for programming languages. *Acta Inform.*, 3(2):155–170, 1974. *Weakly context-sensitive languages* are defined in two steps. First some CF languages are defined traditionally. Second a formula is given involving the CF sets, Boolean operators, quantifiers, and substitutions; this formula defines the words in the WCS language. An example is the language $\mathcal{L}_0 = \mathbf{a}^n\mathbf{b}^n\mathbf{a}^n$. We define the CF languages $S_1 = \mathbf{a}^n\mathbf{b}^n$ and $S_2 = \mathbf{a}^k$. Then $\mathcal{L}_0 = \{w | \exists x \in S_1 \exists y \in S_2 | xy = w \wedge \exists z | y\mathbf{b}z = x\}$. It is shown that this is stronger than CF but weaker than CS. WCS languages are closed under union, intersection, complementation and concatenation, but not under unbounded concatenation (Kleene star). A Turing machine parser is sketched, which recognizes strings in $O(n^k)$ where k depends on the complexity of the formula. A WCS grammar is given, which checks definition and application of variables and labels in ALGOL 60. The unusual formalism and obscure text make the paper a difficult read.

249. **Wijngaarden**, A. van. The generative power of two-level grammars. In J. Loeckx, editor, *Automata, Languages and Programming*, volume 14 of *Lecture Notes in Computer Science*, pages 9–16. Springer-Verlag, Berlin, 1974. The generative power of VW grammars is illustrated by creating a VW grammar that simulates a Turing machine; the VW grammar uses only one metanotion, thus proving that one metanotion suffices.

250. **Joshi**, Aravind K., Levy, Leon S., and Takahashi, Masako. Tree adjunct grammars. *J. Comput. Syst. Sci.*, 10(1):136–163, 1975. See Section 15.4.
The authors start by giving a very unintuitive and difficult definition of trees and tree grammars, which fortunately is not used in the rest of the paper. A hierarchy of tree adjunct grammars is constructed, initially based on the maximum depth of the adjunct trees. This hierarchy does not coincide with Chomsky's:
$$L(TA(1)) \subset L(CF) \subset L(TA(2)) \subset L(TA(3)) \subset \cdots \subset L(CS)$$
A "simultaneous tree adjunct grammar" (STA grammar) also consists of a set of elementary trees and a set of adjunct trees, but the adjunct trees are divided into a number of groups. In each adjunction step one group is selected, and all adjunct trees in a group must be applied simultaneously. It is shown that:
$$L(CF) \subset L(TA(n)) \subset L(STA) \subset L(CS)$$

251. **Wijngaarden**, A. van et al. Revised report on the algorithmic language ALGOL 68. *Acta Inform.*, 5:1–236, 1975. See van Wijngaarden et al. [244].

252. **Cleaveland**, J. Craig and Uzgalis, Robert C. *Grammars for Programming Languages*. Elsevier, New York, 1977. In spite of its title, the book is a highly readable explanation of two-level grammars, also known as *van Wijngaarden grammars* or VW grammars. After an introductory treatment of formal languages, the Chomsky hierarchy and parse trees, it is shown to what extent CF languages can be used to define a programming language. These are shown to fail to define a language completely and the inadequacy of CS grammars is demonstrated. VW grammars are then explained and the remainder of the book consists of increasingly complex and impressive examples of what a VW grammar can do. These examples include keeping a name list, doing type checking and handling block structure in the definition of a programming language. Recommended reading.

253. **Meersman**, R. and Rozenberg, G. Two-level meta-controlled substitution grammars. *Acta Inform.*, 10:323–339, 1978. The authors prove that the uniform substitution rule is essential for two-level grammars; without it, they would just generate the CF languages. This highly technical paper examines a number of variants of the mechanisms involved.

254. **Dembiński**, Piotr and Małuszyński, Jan. Two-level grammars: CF grammars with equation schemes. In Hermann A. Maurer, editor, *Automata, Languages and Programming*, volume 71 of *Lecture Notes in Computer Science*, pages 171–187. Springer-Verlag, Berlin, 1979. The authors address a restricted form of VW grammars in which each metanotion produces a regular language and no metanotion occurs more than once in a hypernotion; such grammars still have full Type 0 power. A context-free skeleton grammar is derived from such a grammar by brute force: each hypernotion in the grammar is supposed to produce each other hypernotion, through added renaming hyperrules. Now the context-free structure of the input is handled by the skeleton grammar whereas the context conditions show up as equations derived trivially from the renaming rules.
The equations are string equations with variables with regular domains. To solve these equations, first all variables are expressed in a number of new variables, each with the domain Σ^*. Then each original variable is restricted to its domain. Algorithms in broad terms are given for both phases.
Any general context-free parser is used to produce all parse trees and for each parse tree we try to solve the set of equations corresponding to it. If the attempt succeeds, we have a parsing. This process will terminate if the skeleton grammar identifies a finite number of parse trees, but in the general case the skeleton grammar is infinitely ambiguous and we have no algorithm.

255. **Kastens**, Uwe. Ordered attribute grammars. *Acta Inform.*, 13(3):229–256, 1980. A *visit* to a node is a sequence of instructions of two forms: evaluate attribute m of child n or of the parent, and perform visit k of child n. A node may require more than one visit, hence the "visit k". If a sequence of visits exists for all nodes so that all attributes are evaluated properly, which is almost always the case, the attribute grammar is *ordered*.

256. **Wegner**, Lutz Michael. On parsing two-level grammars. *Acta Inform.*, 14:175–193, 1980. The article starts by defining a number of properties a VW grammar may exhibit; among these are "left-bound", "right-bound", "free of hidden empty notions", "uniquely assignable" and "locally unambiguous". Most of these properties are undecidable, but sub-optimal tests can be devised. For each VW grammar G_{VW}, a CF *skeleton grammar* G_{SK} is defined by considering all hypernotions in the VW grammar as non-terminals of G_{SK} and adding the cross-references of the VW grammar as production rules to G_{SK}. G_{SK} generates a superset of G_{VW}. The cross-reference problem for VW grammars is unsolvable but again any sub-optimal algorithm (or manual intervention) will do. Parsing is now done by parsing with G_{SK} and then reconstructing and testing the metanotions. A long list of conditions necessary for the above to work are given; these conditions are in terms of the properties defined at the beginning.

257. **Wijngaarden**, A. van. Languageless programming. In *IFIP/TC2/WG2.1 Working Conference on the Relations Between Numerical Computation and Programming Languages*, pages 361–371. North-Holland Publ. Comp., 1981. Forbidding-looking paper which presents an interpreter for a stack machine expressed in a VW grammar. The paper is more accessible than it would seem: the interpreter "reads" — if the term applies — as a cross between Forth and assembler language. A simple but non-trivial program, actually one hyperrule, is given,

which computes the n-th prime, subtracts 25 and "outputs" the answer in decimal notation. The interpreter and the program run correctly on Grune's interpreter [260].

258. **Gerevich**, László. A parsing method based on van Wijngaarden grammars. *Computational Linguistics and Computer Languages*, 15:133–156, 1982. In consistent substitution, a metanotion is replaced consistently by one of its terminal productions; in *extended consistent substitution*, a metanotion is replaced consistently by one of the sentential forms it can produce. The author proves that VW grammars with extended consistent substitution are equivalent to those with just consistent substitution; this allows "lazy" evaluation of the metanotions during parsing. Next an example of a top-down parser using the lazy metanotions as logic variables (called here "grammar-type variables") is shown and demonstrated extensively. The third part is a reasonably intuitive list of conditions under which this parser type works, presented without proof. The fourth part shows how little the VW grammar for a small ALGOL 68-like language needs to be changed to obey these conditions.

259. **Watt**, D. A. and Madsen, O. L. Extended attribute grammars. *Computer J.*, 26(2):142–149, 1983. The assignment rules $A_i := f_i(A_j,\ldots,A_k)$ of Knuth's [243] are incorporated into the grammar by substituting $f_i(A_j,\ldots,A_k)$ for A_i. This allows the grammar to be used as a production device: production fails if any call is undefined. The grammar is then extended with a transduction component; this restores the semantics expressing capability of attribute grammars. Several examples from compiler construction given.

260. **Grune**, Dick. How to produce all sentences from a two-level grammar. *Inform. Process. Lett.*, 19:181–185, Nov. 1984. All terminal productions are derived systematically in breadth-first order. The author identifies pitfalls in this process and describes remedies. A parser is used to identify the hyperrules involved in a given sentential form. This parser is a general CF recursive descent parser to which a consistency check for the metanotions has been added; it is not described in detail.

261. **Małuszyński**, J. Towards a programming language based on the notion of two-level grammar. *Theoret. Comput. Sci.*, 28:13–43, 1984. In order to use VW grammars as a programming language, the cross-reference problem is made solvable by requiring the hypernotions to have a tree structure rather than be a linear sequence of elements. It turns out that the hyperrules are then a generalization of the Horn clauses, thus providing a link with DCGs.

262. **Edupuganty**, Balanjaninath and Bryant, Barrett R. Two-level grammars for automatic interpretation. In *1985 ACM Annual Conference*, pages 417–423. ACM, 1985. First the program is parsed without regard to the predicate hyperrules; this yields both instantiated and uninstantiated metanotions. Using unification-like techniques, these metanotions are then checked in the predicates and a set of interpreting hyperrules is used to construct the output metanotion. All this is similar to attribute evaluation. No exact criteria are given for the validity of this procedure, but a substantial example is given.
The terminal symbols are not identified separately but figure in the hypernotions as protonotions; this is not fundamental but does make the two-level grammar more readable.

263. **Fisher**, A. J. Practical LL(1)-based parsing of van Wijngaarden grammars. *Acta Inform.*, 21:559–584, 1985. Fisher's parser is based on the idea that the input string was generated using only a small, finite, part of the infinite *strict grammar* that can be generated from the VW grammar. The parser tries to reconstruct this part of the strict grammar on the fly while parsing the input. The actual parsing is done by a top-down interpretative LL(1) parser, called the *terminal parser*. It is driven by a fragment of the strict grammar and any time the definition of a non-terminal is found missing by the terminal parser, it asks another module, the *strict syntax generator*, to try to construct it from the VW grammar. For this technique to work, the VW grammar has to satisfy three conditions: the defining CF grammar of each hyperrule is unambiguous, there are no free metanotions, and the skeleton grammar (as defined by Wegner [256]) is LL(1). The parser system is organized as a set of concurrent processes (written in occam), with both parsers, all hyperrule matchers and several other modules as separate processes. The author claims that "this concurrent organization ... is strictly a property of the algorithm, not of the implementation", but

a sequential, albeit slower, implementation seems quite possible. The paper gives heuristics for the automatic generation of the cross-reference needed for the skeleton grammar; gives a method to handle *general hyperrules*, hyperrules that fit all hypernotions, efficiently; and pays much attention to the use of angle brackets in VW grammars.

264. **Vijay-Shankar**, K. and Joshi, Aravind K. Some computational properties of tree adjoining grammars. In *23rd Annual Meeting of the ACL*, pages 82–93, University of Chicago, Chicago, IL, July 1985. Parsing: the CYK algorithm is extended to TAGs as follows. Rather than having a two-dimensional array $A_{i,j}$ the elements of which contain non-terminals that span $t_{i..j}$ where t is the input string, we have a four-dimensional array $A_{i,j,k,l}$ the elements of which contain tree nodes X that span $t_{i..j}..t_{k..l}$, where the gap $t_{j+1,k-1}$ is spanned by the tree hanging from the foot node of X. The time complexity is $O(n^6)$ for TAGs that are in "two form" .
Properties: informal proofs are given that TAGs are closed under union, concatenation, Kleene star, and intersection with regular languages.

265. **Barnard**, D. T. and Cordy, J. R. SL parses the LR languages. *Comput. Lang.*, 13(2):65–74, July 1988. *SL* (*Syntax Language*) is a special-purpose language for specifying recursive input-output transducers. An SL program consists of a set of recursive parameterless routines. The code of a routine can call other routines, check the presence of an input token, produce an output token, and perform an n-way switch on the next input token, which gets absorbed in the process. Blocks can be repeated until an exit statement is switched to. The input and output streams are implicit and are the only variables.

266. **Schabes**, Yves and Vijay-Shankar, K. Deterministic left to right parsing of tree adjoining languages. In *28th Meeting of the Association for Computational Linguistics*, pages 276–283. Association for Computational Linguistics, 1990. Since production using a TAG can be based on a stack of stacks (see Vijay-Shankar and Joshi [264]), the same model is used to graft LR parsing on. Basically, the stacks on the stack represent the reductions of the portion left of the foot in each adjoined tree; the stack itself represents the spine of the entire tree recognized so far. Dotted trees replace the usual dotted items; stack manipulation during the "Resume Right" operation, basically a shift over a reduced tree root, is very complicated. See Nederhof [281].

267. **Heilbrunner**, S. and Schmitz, L. An efficient recognizer for the Boolean closure of context-free languages. *Theoret. Comput. Sci.*, 80:53–75, 1991. The CF grammars are extended with two operators: negation (anything not produced by A) and intersection (anything produced by both A and B). The non-terminals in the grammar have to obey a hierarchical order, to prevent paradoxes: $A \rightarrow A$ would define an A which produces anything not produced by A. An Earley parser in CYK formulation (Graham et al. [23]) is extended with inference (dot-movement) rules for these operators, and a special computation order for the sets is introduced. This leads to a "naive" (well...) algorithm, to which various optimizations are applied, resulting in an efficient $O(n^3)$ algorithm. A 10-page formal proof concludes the paper.

268. **Koster**, C. H. A. Affix grammars for natural languages. In Henk Alblas and Bořivoj Melichar, editors, *Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 469–484, New York, 1991. Springer Verlag. The domains of the affixes are restricted to finite lattices, on the grounds that this is convenient in linguistics; lattices are explained in the text. This formally reduces the grammar to a CF one, but the size can be spectacularly smaller. Inheritance and subsetting of the affixes is discussed, as are parsing and left recursion. An example for English is given. Highly amusing account of the interaction between linguist and computer scientist.

269. **Koster**, C. H. A. Affix grammars for programming languages. In Henk Alblas and Bořivoj Melichar, editors, *Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 358–373. Springer-Verlag, New York, 1991. After a historical introduction, the three formalisms VW Grammar, Extended Attribute/Affix Grammar, and Attribute Grammar are compared by implementing a very simple language consisting of declarations and assignments in them. The comparison includes Prolog. The conclusion finds

far more similarities than differences; VW Grammars are the most descriptional, Attribute Grammars the most operational, with EAGs in between.

270. **Krulee**, Gilbert K. *Computer Processing of Natural Language*. Prentice-Hall, 1991. Concentrates on those classes of grammars and features that are as applicable to English as to Pascal (paraphrase of page 1). No overly soft linguistic talk, no overly harsh formalisms.
The book is based strongly on two-level grammars, but these are not the Van Wijngaarden type in that no new non-terminals are produced. The metanotions produce strings of terminals and non-terminals of the CF grammar rather than segments of names of non-terminals. When this is done, the applied occurrences of metanotions in the CF grammar must be substituted using some uniform substitution rule. An Earley parser for this kind of two-level grammars is sketched. Parsers for ATN systems are also covered.

271. **Schabes**, Yves and Joshi, Aravind K. Parsing with lexicalized tree adjoining grammars. In Masaru Tomita, editor, *Current Issues in Parsing Technology*, pages 25–47. Kluwer Academic Publ., Boston, 1991. A grammar is "lexicalized" if each right-hand side in it contains at least one terminal, called its "anchor". Such grammars cannot be infinitely ambiguous. In parsing a sentence from a lexicalized grammar, one can first select the rules that can play a role in parsing, based on the terminals they contain, and restrict the parser to these. In very large grammars this helps.
Various parser variants for this structure are described: CYK, top-down, Earley and even LR.
Feature-based tree adjoining grammars are tree adjoining grammars with attributes and unification rules attached to each node. Although recognition for feature-based tree adjoining grammars is undecidable, an adapted Earley algorithm is given that will parse a restricted set of feature-based lexicalized tree adjoining grammars.

272. **Seki**, Hiroyuki, Matsumura, Takashi, Fuji, Mamoru, and Kasami, Tadao. On multiple context-free grammars. *Theoret. Comput. Sci.*, 88:191–229, 1991. Each non-terminal in a *multiple context-free grammar* (MCFG) produces a fixed number of strings rather than just one string; so it has a fixed number of right-hand sides. Each right-hand side is composed of terminals and components of other non-terminals, under the condition that if a component of a non-terminal A occurs in the right-hand side of a non-terminal B all components of A must be used. Several varieties are covered in the paper, each with slightly different restrictions.
MCFGs are stronger than CFGs: for example, $S \rightarrow (aS_1, bS_2, cS_3) | (\varepsilon, \varepsilon, \varepsilon)$, where S_1, S_2, and S_3 are the components of S, produces the language $a^n b^n c^n$. But even the strongest variety is weaker than CS.
Properties of this type of grammars are derived and proved; the grammars themselves are written in a mathematical notation. An $O(n^e)$ recognition algorithm is given, where e is a grammar-dependent constant. The algorithm is a variant of CYK, in that it constructs bottom-up sets of components of increasing length, until that length is equal to the length of the input. Parsing (the recovery of the derivation tree) is not discussed.

273. **Fisher**, Anthony J. A "yo-yo" parsing algorithm for a large class of van Wijngaarden grammars. *Acta Inform.*, 29(5):461–481, 1992. High-content paper describing a top-down parser which tries to reconstruct the production process that led to the input string, using an Earley-style parser to construct metanotions bottom-up where needed; it does not involve a skeleton grammar. It can handle a class of VW grammars characterized roughly by the following conditions: the cross-reference problem must be solvable by LL(1) parsing of the hypernotions; certain mixes of "left-bound" and "right-bound" (see Wegner [256]) do not occur; and the VW grammar is not left-recursive. The time requirement is $O(n^3 f^3(n))$, where $f(n)$ depends on the growth rate of the fully developed hypernotions ("protonotions") as a function of the length of the input. For "decent" grammars, $f(n) = n$, and the time complexity is $O(n^6)$.

274. **Grune**, Dick. Two-level grammars are more expressive than Type 0 grammars — or are they?. *ACM SIGPLAN Notices*, 28(8):43–45, Aug. 1993. VW grammars can construct names of non-terminals, but they can equally easily construct names of terminals, thus allowing the grammar to create new terminal symbols. This feat cannot be imitated by Type 0 grammars, so in a

sense VW grammars are more powerful. The paper gives two views of this situation, one in which the statement in the title is true, and one in which it is undefined.

275. **Pitsch**, Gisela. *LL(k)* parsing of coupled context-free grammars. *Computational Intelligence*, 10(4):563–578, 1994. LL parsing requires the prediction of a production $A \rightarrow A_1, A_2, \ldots, A_n$, based on look-ahead, and in CCFG we need the look-ahead at n positions in the input. Although we know which position in the input corresponds to A_1, we do not know which positions match $A_2, \ldots, A_n$, and we cannot obtain the required look-aheads. We therefore restrict ourselves to strong LL, based on the FIRST set of A_1 and the FOLLOW sets of $A_1, \ldots, A_n$. Producing the parse tables is complex, but parsing itself is simple, and linear-time.

276. **Satta**, Giorgio. Tree adjoining grammar parsing and boolean matrix multiplication. *Computational Linguistics*, 20(2):173–192, 1994. Proves that if we can do tree parsing in $O(n^p)$, we can do Boolean matrix multiplication in $O(n^{2+p/6})$, which for $p = 6$ amounts to the standard complexities for both processes. Since Boolean matrix multiplication under $O(n^3)$ is very difficult, it is probable that tree parsing under $O(n^6)$ is also very difficult.

277. **Pitsch**, Gisela. LR(*k*)-coupled-context-free grammars. *Inform. Process. Lett.*, 55(6):349–358, Sept. 1995. The coupling between the components of the coupled non-terminals is implemented by adding information about the reduction of a component X_1 to a list called "future", which runs parallel to the reduction stack. This list is used to control the LR automaton so that only proper reduces of the further components X_n of X will occur.

278. **Hotz**, Günter and Pitsch, Gisela. On parsing coupled-context-free languages. *Theoret. Comput. Sci.*, 161(1-2):205–233, 1996. General parsing with CCF grammars, mainly based on the CYK algorithm. Full algorithms, extensive examples.

279. **Kulkarni**, Sulekha R. and Shankar, Priti. Linear time parsers for classes of non context free languages. *Theoret. Comput. Sci.*, 165(2):355–390, 1996. The non-context-free languages are generated by two-level grammars as follows. The rules of the base grammar are numbered and one member of each RHS is marked as distinguished; the start symbol is unmarked. So from each unmarked non-terminal in the parse tree one can follow a path downward by following marked non-terminals, until one reaches a terminal symbol. A parse tree is acceptable only if the sequence of numbers of the rules on each such path is generated by the control grammar.
LL(1) and LR(1) parsers for such grammars, using stacks of stacks, are described extensively.

280. **Rußmann**, A. Dynamic LL(*k*) parsing. *Acta Inform.*, 34(4):267–290, 1997. Theory of LL(1) parsing of dynamic grammars.

281. **Nederhof**, Mark-Jan. An alternative LR algorithm for TAGs. In *36th Annual Meeting of the Association for Computational Linguistics*, pages 946–952. ACL, 1998. The traditional LR parsing algorithm is extended in a fairly straightforward way to parsing TAGs. It uses the traditional LR stack containing states and symbols alternately, although the symbols are sometimes more complicated. The author shows that Schabes and Vijay-Shankar's algorithm [266] is incorrect, and recognizes incorrect strings.
Upon implementation, it turned out that the LR transition tables were "prohibitively large" (46MB) for a reasonable TAG for English. But the author represents the table as a set of Prolog clauses (!) and does not consider table compression.

282. **Prolo**, Carlos A. An efficient LR parser generator for tree adjoining grammars. In *6th Int. Workshop on Parsing Technologies (IWPT 2000)*, pages 207–218, 2000. Well-argued exposition of the problems inherent in LR parsing of TAGs. Presents an LR parser generator which produces tables that are one or two orders of magnitude smaller than Nederhof's [281], making LR parsing of tree adjoining grammars more feasible.

283. **Okhotin**, Alexander. Conjunctive grammars. *J. Automata, Languages and Combinatorics*, 6(4):519–535, 2001. A conjunctive grammar is a CF grammar with an additional intersection operation. Many properties of conjunctive grammars are shown and proven,

and many examples are provided. For example, the conjunctive grammars are stronger than the intersection of a finite number of CF languages. They lead to parse dags. Tabular parsing is possible in time $O(n^3)$.

284. **Ford**, Bryan. Packrat parsing: Simple, powerful, lazy, linear time. *ACM SIGPLAN Notices*, 37(9):36–47, Sept. 2002. A straightforward backtracking top-down parser in Haskell is supplied with memoization (see Section 17.3.4), which removes the need for repeated backtracking and achieves unbounded look-ahead. Linear-time parsing is achieved by always matching the largest possible segment; this makes the result of a recognition unique, and the parsing unambiguous. Left recursion has to be removed by the user, but code is supplied to produce the correct parse tree nevertheless. Since the memoized functions remember only one result and then stick to that, Packrat parsing cannot handle all CF languages; a delineation of the set of suitable languages is not given. See, however, Ford [286]. Implementation of the parser using monads is discussed.

285. **Jackson**, Quinn Tyler. Efficient formalism-only parsing of XML/HTML using the §-calculus. *ACM SIGPLAN Notices*, 38(2):29–35, Feb. 2003. The §-calculus is a CF grammar in which new values can be dynamically assigned to non-terminals in the grammar during parsing. Such values can be the value of a generic terminal (identifiers, etc.) found in the input or a new CF production rule, somewhat similar to the Prolog assert feature. This allows context-sensitive restrictions to be incorporated in the grammar. This system is used to write a concise grammar capable of handling both XML and HTML documents. It is then run on Meta-S, a backtracking LL(k) recursive descent parser for the §-calculus.

286. **Ford**, Bryan. Parsing expression grammars: A recognition-based syntactic foundation. In *31st ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 111–122. ACM, Jan. 2004. A *PEG* (*Parsing Expression Grammar*) describes a language by being a recognition algorithm. It is basically an EBNF grammar whose meaning is determined by a top-down interpreter, similar to those described by Birman and Ullman [246]. The interpreter works left-to-right top-to-bottom and always consumes the longest possible input: an expression $e_1 e_2 \cdots / e_3 \cdots$ means **if** e_1 **andif** e_2 **andif** $\cdots$ **then** *succeed* **else** e_3 **andif** $\cdots$ **fi**. If an expression succeeds it consumes what it has recognized; if an expression fails, it consumes nothing, even if subsections of it have recognized some input. This requires backtracking. PEGs have two additional operators, &A, which tests for the presence of an A but consumes nothing, and !A, which tests for the absence of an A and consumes nothing. PEGs have to be "well-formed", which basically means "not left-recursive".
PEGs have several advantages over CF grammars: PEGs are unambiguous; PEG languages are closed under intersection and negation; PEGs can recognize some non-CF languages; and parsing with PEGs can be done in linear time.
These and several other properties — static analysis, well-formedness, algebraic equalities, relation to Birman and Ullman's TS and gTS — are proved in the paper, with short common-sense proofs.

287. **Grimm**, Robert. Practical packrat parsing. Technical Report TR2004-854, Dept. of Computer Science, New York University, New York, March 2004. Describes an object-oriented implementation of Packrat parsing in Java, called Rats"!. It allows the attachment of semantics to rules.

288. **Okhotin**, Alexander. Boolean grammars. *Information and Computation*, 194(1):19–48, 2004. Boolean grammars are CF grammars extended with intersection and negation. The languages they define are not described by a substitution mechanism, but in one of two ways: as the solution of a set of equations, and as the partial fixed point of a function. It is not necessary for both of them to exist, but it is shown that *if* both exist, they define the same language. If neither solution exists, the grammar is not well-formed. Many properties of Boolean grammars are shown and proven; a binary form is defined; and the corresponding CYK algorithm is presented, yielding a parse dag. This allows parsing in $O(n^3)$. Remarkably they can be recognized in $O(n)$ space, but that takes some doing.
Linear Boolean grammars are Boolean grammars in which each conjunct contains at most one non-terminal. They are proven to be equivalent to trellis automata. Useful tables of comparisons of grammars and languages complete the paper.

289. **Okhotin**, Alexander. LR parsing for boolean grammars. In *International Conference on Developments in Language Theory (DLT)*, volume 9, pages 362–373, 2005. GLR parsing is extended with two operations, a "conjunctive reduce", which is almost the same as the traditional reduce, except that for $X \to ABC\&DEF$ it reduces only if both ABC and DEF are present, and an *invalidate*, which removes clusters of branches from the GSS in response to finding an X where the grammar calls for $\neg X$. Complete algorithms are given. The time complexity is $O(n^4)$, which can be reduced to $O(n^3)$ by further memoization. A short sketch of an LL(1) parser for Boolean grammars is also given.

290. **Okhotin**, Alexander. On the existence of a Boolean grammar for a simple procedural language. In *11th International Conference on Automata and Formal Languages: AFL'05*, 2005. A paradigm for using Boolean grammars for the formal specification of programming languages is being developed. The method involves a sublanguage $C = I\Sigma^*I$, where both occurrences of I represent the same identifier and Σ^* can be anything as long as it sets itself off against the two identifiers. The CF part of the Boolean grammar is then used to assure CF compliance of the program text, and repeated intersection with C is used to insure that all identifiers are declared and intersection with $\neg C$ to catch multiple declarations. Once C has been defined the rest of the Boolean grammar is quite readable; it completely specifies and checks all context conditions. Experiments show that the time complexity is about $O(n^2)$. A critical analysis closes the paper.

291. **Jackson**, Quinn Tyler. *Adapting to Babel: Adaptivity and Context-Sensitivity in Parsing*. In Press, 2006. The §-calculus (pronounced "meta-ess calculus") (Jackson [285]) is extended with a notation A-BNF, "Adaptive BNF", which is BNF extended with several grammar and set manipulation functions, including intersection with a set generated by a subgrammar. This allows full Turing power. A very simple example is a §-grammar (A-BNF) for palindromes: `S ::= $x('[a-zA-Z]') [S] x`; this means: to accept an `S`, accept one token from the input if it intersects with the set of letters and assign it to the variable `x`, optionally accept an `S`, and finally accept the token in variable `x`.
The implementation uses a pushdown automaton augmented with name-indexed tries (PDA-T) reminiscent of a nested stack automaton, and zillions of optimizations. The time complexity is unknown; in practice it is almost always less than $O(n^2)$ and always less than $O(n^3)$. Although §-grammars may be seen as generating devices, the author makes a strong point for seeing them as recognition devices.
All facets of the system are described extensively, with many examples.

18.2.7 Error Handling

292. **Aho**, A. V. and Peterson, T. G. A minimum-distance error-correcting parser for context-free languages. *SIAM J. Computing*, 1(4):305–312, 1972. A CF grammar is extended with error productions so that it will produce Σ^*; this is effected by replacing each occurrence of a terminal in a rule by a non-terminal that produces said terminal "with 0 errors" and any amount of garbage, including ε, "with 1 or more errors". The items in an Earley parser are extended with a count, indicating how many errors were needed to create the item. An item with error count k is added only if no similar item with a lower error count is present already.

293. **Conway**, R. W. and Wilcox, T. R. Design and implementation of a diagnostic compiler for PL/I. *Commun. ACM*, 16(3):169–179, 1973. Describes a diagnostic PL/C compiler, using a systematic method for finding places where repair is required, but the repair strategy for each of these places is chosen by the implementor. The parser uses a separable transition diagram technique (see Conway [333]). The error messages detail the error found and the repair chosen.

294. **Lyon**, G. Syntax-directed least-errors analysis for context-free languages: a practical approach. *Commun. ACM*, 17(1):3–14, Jan. 1974. Discusses a least-error analyser, based on Earley's parser without look-ahead. The Earley items are extended with an error count, and the parser is started with items for the start of each rule, in each state set. Earley's scanner is extended

as follows: for all items with the dot in front of a terminal, the item is added to the same state set with an incremented error count and the dot after the terminal (this represents an insertion of the terminal); if the terminal is not equal to the input symbol associated with the state set, add the item to the next state set with an incremented error count and the dot after the terminal (this represents a replacement); add the item as it is to the next state set, with an incremented error count (this represents a deletion). The completer does its work as in the Earley parser, but also updates error counts. Items with the lowest error counts are processed first, and when a state set contains an item, the same item is only added if it has a lower error count.

295. **Graham**, Susan L. and Rhodes, Steven P. Practical syntactic error recovery. *Commun. ACM*, 18(11):639–650, Nov. 1975. See Section 16.5 for a discussion of this error recovery method.

296. **Horning**, James J. What the compiler should tell the user. In Friedrich L. Bauer and Jürgen Eickel, editors, *Compiler Construction, An Advanced Course, 2nd ed*, volume 21 of *Lecture Notes in Computer Science*, pages 525–548. Springer, 1976. Lots of good advice on the subject, in narrative form. Covers the entire process, from lexical to run-time errors, considering detection, reporting and possible correction. No implementation hints.

297. **Hartmann**, Alfred C. *A Concurrent Pascal Compiler for Minicomputers*, volume 50 of *Lecture Notes in Computer Science*. Springer, 1977. [Parsing / error recovery part only:] Each grammar rule is represented as a small graph; each graph is converted into a subroutine doing top-down recursive descent. To aid error recovery, a set of "key" tokens is passed on, consisting of the union of the FIRST sets (called "handles" in the text) of the symbols on the prediction stack, the intuition being that each of these tokens could, in principle, start a prediction if all the previous ones failed. This set is constructed and updated during parsing. Before predicting the alternative for a non-terminal *A*, all input tokens not in the key set at this place are skipped, if any. If that does not bring up a token from *A*'s FIRST set — and thus allow an alternative to be chosen — *A* is discarded and the next prediction is tried.

298. **Lewi**, J., Vlaminck, K. de, Huens, J., and Huybrechts, M. The ELL(1) parser generator and the error-recovery mechanism. *Acta Inform.*, 10:209–228, 1978. Presents a detailed recursive descent parser generation scheme for ELL(1) grammars, and also presents an error recovery method based on so-called *synchronization triplets* (*a*,*b*,*A*). *a* is a terminal from FIRST(*A*), *b* is a terminal from LAST(*A*). The parser operates either in parsing mode or in error mode. It starts in parsing mode, and proceeds until an error occurs. Then, in error mode, symbols are skipped until either an end marker *b* is found where *a* is the last encountered corresponding begin-marker, in which case parsing mode resumes, or a begin-marker *a* is found, in which case *A* is invoked in parsing mode. As soon as *A* is accepted, error-mode is resumed. The success of the method depends on careful selection of synchronization triplets.

299. **Mickunas**, M. Dennis and Modry, John A. Automatic error recovery for LR parsers. *Commun. ACM*, 21(6):459–465, June 1978. When an error is encountered, a set of provisional parsings of the beginning of the rest of the input (so-called *condensations*) are constructed: for each state a parsing is attempted and those that survive according to certain criteria are accepted. This yields a set of target states. Now the stack is "frayed" by partly or completely undoing any reduces; this yields a set of source states. Attempts are made to connect a source state to a target state by inserting or deleting tokens. Careful rules are given.

300. **Pennello**, Thomas J. and DeRemer, Frank L. A forward move algorithm for LR error recovery. In *Fifth ACM Symposium on Principles of Programming Languages*, pages 241–254, Jan. 1978. Refer to Graham and Rhodes [295]. Backward moves are found to be detrimental to error recovery. The extent of the forward move is determined as follows. At the error, an LALR(1) parser is started in a state including *all* possible items. The thus extended automaton is run until it wants to reduce past the error detection point. The resulting right context is used in error correction. An algorithm for the construction of a reasonably sized extended LALR(1) table is given.

301. **Tanaka**, Eiichi and Fu, King-Sun. Error-correcting parsers for formal languages. *IEEE Trans. Comput.*, C-27(7):605–616, July 1978. Starts from a CF CYK parser based on a 2-form grammar. The entry for a recognized symbol A in the matrix contains 0, 1 or 2 pointers to its children, plus an error weight; the entry with the lowest error weight is retained. Next, the same error-correction mechanism is introduced in a context-sensitive CYK parser, for which see [1]. Full algorithms are given. Finally some theorems are proven concerning these parsers, the main one being that the error-correcting properties under these algorithms depend on the language only, not on the grammar used. High-threshold, notationally heavy paper, with extensive examples though.

302. **Fischer**, C. N., Tai, K.-C., and Milton, D. R. Immediate error detection in strong LL(1) parsers. *Inform. Process. Lett.*, 8(5):261–266, June 1979. A strong-LL(1) parser will sometimes perform some incorrect parsing actions, connected with ε-matches, when confronted with an erroneous input symbol, before signalling an error; this impedes subsequent error correction. A subset of the LL(1) grammars is defined, the *nullable LL(1) grammars*, in which rules can only produce ε directly, not indirectly. A special routine, called before an ε-match is done, hunts down the stack to see if the input symbol will be matched or predicted by something deeper on the stack; if not, an error is signaled immediately. An algorithm to convert any strong-LL(1) grammar into a non-nullable strong-LL(1) grammar is given. (See also Mauney and Fischer [309]).

303. **Fischer**, C. N., Milton, D. R., and Quiring, S. B. Efficient LL(1) error correction and recovery using only insertions. *Acta Inform.*, 13(2):141–154, 1980. See Section 16.6.4 for a discussion of this error recovery method.

304. **Pemberton**, Steven. Comments on an error-recovery scheme by Hartmann. *Softw. Pract. Exper.*, 10(3):231–240, 1980. Extension of Hartmann's error recovery scheme [297]. Error recovery in a recursive descent parser is done by passing to each parsing routine a set of "acceptable" symbols. Upon encountering an error, the parsing routine will insert any directly required terminals and then skip input until an acceptable symbol is found. Rules are given and refined on what should be in the acceptable set for certain constructs in the grammar.

305. **Röhrich**, Johannes. Methods for the automatic construction of error correcting parsers. *Acta Inform.*, 13(2):115–139, Feb. 1980. See Section 16.6.3 for a discussion of this error recovery method. The paper also discusses implementation of this method in LL(k) and LR(k) parsers, using so-called *deterministic continuable stack automata*.

306. **Anderson**, Stuart O. and Backhouse, Roland C. Locally least-cost error recovery in Earley's algorithm. *ACM Trans. Prog. Lang. Syst.*, 3(3):318–347, July 1981. Parsing and error recovery are unified so that error-free parsing is zero-cost error recovery. The information already present in the Earley items is utilized cleverly to determine possible continuations. From these and from the input, the locally least-cost error recovery can be computed, albeit at considerable expense. Detailed algorithms are given.

307. **Dwyer**, Barry. A user-friendly algorithm. *Commun. ACM*, 24(9):556–561, Sept. 1981. Skinner's theory of operant conditioning applied to man/machine interaction: tell the user not what is wrong but help him how to do better. In syntax errors this means showing what the parser understood and what the pertinent syntax rules are.

308. **Gonser**, Peter. *Behandlung syntaktischer Fehler unter Verwendung kurzer, fehlereinschließender Intervalle*. PhD thesis, Technical report, Technische Universität München, München, July 21 1981, (in German). Defines a syntax error as a minimal substring of the input that cannot be a substring of any correct input; if there are n such substrings, there are (at least) n errors. Finding such substrings is too expensive, but if we are doing simple precedence parsing and have a stack configuration $b \lessdot A_1 \odot A_2 \odot \cdots \odot A_k$ and a next input token c, where $\odot$ is either $\lessdot$ or $\doteq$ and there is no precedence relation between A_k and c, then the substring from which $bA_1A_2 \cdots A_kc$ was reduced must contain at least one error. The reason is that precedence information does not travel over terminals; only non-terminals can transmit information from left to right through the stack, by the choice of the non-terminal. So if the c cannot be understood, the cause cannot lie to the left of the b. This gives us an interval that is guaranteed to contain an error.

Several rules are given on how to turn the substring into an acceptable one; doing this successively for all error intervals turns the input into a syntactically correct one. Since hardly any grammar is simple precedence, several other precedence-like grammar forms are developed which are stronger and in the end cover the deterministic languages. See [130] for these.

309. **Mauney**, Jon and Fischer, Charles N. An improvement to immediate error detection in strong LL(1) parsers. *Inform. Process. Lett.*, 12(5):211–212, 1981. The technique of Fischer, Tai and Milton [302] is extended to all LL(1) grammars by having the special routine which is called before an ε-match is done do conversion to non-nullable on the fly. Linear time dependency is preserved by setting a flag when the test succeeds, clearing it when a symbol is matched and by not performing the test if the flag is set: this way the test will be done at most once for each symbol.

310. **Anderson**, S. O. and Backhouse, R. C. An alternative implementation of an insertion-only recovery technique. *Acta Inform.*, 18:289–298, 1982. Argues that the FMQ error corrector of Fischer, Milton and Quiring [303] does not have to compute a complete insertion. It is sufficient to compute the first symbol. If $w = w_1 w_2 \cdots w_n$ is an optimal insertion for the error a following prefix u, then $w_2 \cdots w_n$ is an optimal insertion for the error a following prefix uw_1. Also, immediate error detection is not necessary. Instead, the error corrector is called for every symbol, and returns an empty insertion if the symbol is correct.

311. **Anderson**, S. O., Backhouse, R. C., Bugge, E. H., and Stirling, C. P. An assessment of locally least-cost error recovery. *Computer J.*, 26(1):15–24, 1983. Locally least-cost error recovery consists of a mechanism for editing the next input symbol at least cost, where the cost of each edit operation is determined by the parser developer. The method is compared to Wirth's followset method (see Stirling [314]) and compares favorably.

312. **Brown**, P. J. Error messages: The neglected area of the man/machine interface?. *Commun. ACM*, 26(4):246–249, 1983. After showing some appalling examples of error messages, the author suggests several improvements: 1. the use of windows to display the program text, mark the error, and show the pertinent manual page; 2. the use of a syntax-directed editor to write the program; 3. have the parser suggest corrections, rather than just error messages. Unfortunately 1 and 3 seem to require information of a quality that parsers that produce appalling error messages just cannot provide.

313. **Richter**, Helmut. Noncorrecting syntax error recovery. *ACM Trans. Prog. Lang. Syst.*, 7(3):478–489, July 1985. Extends Gonser's method [308] by using suffix grammars and a reverse scan, which yields provable properties of the error interval. See Section 16.7 for a discussion of this method. Bounded-context grammars are conjectured to yield deterministic suffix grammars.

314. **Stirling**, Colin P. Follow set error recovery. *Softw. Pract. Exper.*, 15(3):239–257, March 1985. Describes the followset technique for error recovery: at all times there is a set of symbols that depends on the parse stack and that will not be skipped, called the *followset*. When an error occurs, symbols are skipped until one is found that is a member of this set. Then, symbols are inserted and/or the parser state is adapted until this symbol is legal. In fact there is a family of error recovery (correction) methods that differ in the way the followset is determined. The paper compares several of these methods.

315. **Choe**, Kwang-Moo and Chang, Chun-Hyon. Efficient computation of the locally least-cost insertion string for the LR error repair. *Inform. Process. Lett.*, 23(6):311–316, 1986. Refer to Anderson et al. [311] for locally least-cost error correction. The paper presents an efficient implementation in LR parsers, using a formalism described by Park, Choe and Chang [65].

316. **Kantorowitz**, E. and Laor, H. Automatic generation of useful syntax error messages. *Softw. Pract. Exper.*, 16(7):627–640, July 1986. Rules for useful syntax error messages: 1. Indicate a correction only if it is the only possibility. 2. Otherwise show the full list of legal tokens in the error position. 3. Mark skipped text.
To implement this the grammar is required to be LL(1) and each rule is represented internally by a syntax diagram. In case 1 the recovery is easy: perform the correction. Case 2 relies on an

"acceptable set", computed in two steps. First all paths in the present syntax diagram starting from the error point are searched for terminals that do not occur in the FIRST sets of non-terminals in the same syntax diagram. If that set is not empty it is the acceptable set. Otherwise the FOLLOW set is constructed by consulting the stack, and used as the acceptable set. Explicit algorithms given.

317. **Burke**, Michael G. and Fisher, Gerald A. A practical method for LL and LR syntactic error diagnosis and recovery. *ACM Trans. Prog. Lang. Syst.*, 9(2):164–197, April 1987. Traditional error recovery assumes that all tokens up to the error symbol are correct. The article investigates the option of allowing earlier tokens to be modified. To this end, parsing is done with two parsers, one of which is a number of tokens ahead of the other. The first parser does no actions and keeps enough administration to be rolled back, and the second performs the semantic actions; the first parser will modify the input stream or stack so that the second parser will never see an error. This device is combined with three error repair strategies: single token recovery, scope recovery and secondary recovery. In single token recovery, the parser is rolled back and single tokens are deleted, inserted or replaced by tokens specified by the parser writer. In scope recovery, closers as specified by the parser writer are inserted before the error symbol. In secondary recovery, sequences of tokens around the error symbol are discarded. In each case, a recovery is accepted if it allows the parser to advance a specified number of tokens beyond the error symbol. It is reported that this technique corrects three quarters of the normal errors in Pascal programs in the same way a knowledgeable human would. The effects of fine-tuning are discussed.

318. **Cormack**, Gordon V. An LR substring parser for noncorrecting syntax error recovery. *ACM SIGPLAN Notices*, 24(7):161–169, June 1989. Using the BC-SLR(1,1) substring parser from the same paper ([211]) the author gives examples of interval analysis on incorrect Pascal programs.

319. **Charles**, Philippe. An LR(k) error diagnosis and recovery method. In *Second International Workshop on Parsing Technologies*, pages 89–99, Feb. 1991. Massive approach to syntax error recovery, extending the work of Burke and Fisher [317], in four steps. 1. No information is lost in illegal reductions, as follows. During each reduction sequence, the reduce actions are stored temporarily, and actually applied only when a successful shift action follows. Otherwise the original stack is passed to the recovery module. 2. Primary (local) recovery: include merging the error token with its successor; deleting the error token; inserting an appropriate terminal in front of the error token; replacing the error token by a suitable terminal; inserting an appropriate non-terminal in front of the error token; replacing the error token by a suitable non-terminal. All this is controlled by weights, penalties and number of tokens that can be accepted after the modification. 3. Secondary (phrase-level) recovery: for a sequence of "important non-terminals" the unfinished phrase is removed from the stack and a synchronization is made, until a good one is found. Criteria for "important non-terminals" are given. 4. Scope recovery, in which nesting errors are repaired: For each self-embedding rule A, nesting information is precomputed, in the form of a scope prefix, a scope suffix, a look-ahead token, and a set of states. Upon error, these scopes are tested to bridge a possible gap over missing closing elements. The system provided excellent error recovery in a very large part of the cases tried.
Complete algorithms are given.

320. **Deudekom**, A. van and Kooiman, P. Top-down non-correcting error recovery in LLgen. Technical Report IR 338, Vrije Universiteit, Faculteit Wiskunde en Informatica, Amsterdam, Oct. 1993. Describes the addition of a Richter-style [313] error recovery mechanism to *LLgen*, an LL(1) parser generator, using a Generalized LL parser. The suffix grammar used by the mechanism is generated on the fly, and pitfalls concerning left recursion (a general problem in LL parsing), right recursion (a specific problem in error recovery), and ε-rules are pointed out and solved.
LLgen allows liberties with the LL(1) concept; these may interfere with automated error recovery. The conflict resolvers turned out to be no problem, but LLgen allows subparsers to be called from semantic actions, thus extending the accepted language, and syntax error messages to be given from semantic actions, thus restricting the accepted language. The error recovery grammar, however, has to represent the accepted language precisely; this necessitated two new parser generator directives.

Examples of error recovery and efficiency measurements are provided.
See also [170] for the Generalized LL parsing part.

321. **McKenzie**, Bruce J., Yeatman, Corey, and De Vere, Lorraine. Error repair in shift-reduce parsers. *ACM Trans. Prog. Lang. Syst.*, 17(4):672–689, July 1995. The two-stage technique described uses breadth-first search to obtain a series of feasible repairs, each of which is then validated. The first feasible validated repair is accepted.
To obtain feasible repairs, a priority queue of parser states each containing a stack, a representation of the rest of the input, a string of insert tokens, a string of deleted tokens and a cost is created in breadth-first fashion, ordered by cost. The top parser state in the queue is considered, a new state is created for each possible shift, with its implied inserted token, and a new state for the deletion of one token from the input, each of them with its cost. If one of these new states allows the parser to continue, it is deemed feasible and examined for validity.
The repair is valid if it allows the parser to accept the next N input tokens. If it is invalid, more parser states are created in the priority queue. If the queue gets exhausted, no error recovery is possible.
The paper contains much sound advice about implementing such a scheme. To reduce the number of parser states that have to be examined, a very effective pruning heuristic is given, which reduces the number by two or three orders of magnitude. In rare cases, however, the heuristic causes some cheaper repairs to be missed. See also Bertsch and Nederhof [323].

322. **Ruckert**, Martin. Generating efficient substring parsers for BRC grammars. Technical Report 98-105, State University of New York at New Paltz, New Paltz, NY 12561, July 1998. Error reporting and recovery using a BRC-based substring parser. For the parser see [217].

323. **Bertsch**, Eberhard and Nederhof, Mark-Jan. On failure of the pruning technique in "error repair in shift-reduce parsers". *ACM Trans. Prog. Lang. Syst.*, 21(1):1–10, Jan. 1999. The authors analyse the pruning heuristic presented in McKenzie et al. [321], and show that it can even cause the repair process to fail. A safe pruning heuristic is given, but it is so weak, and the failing cases are so rare, that the authors recommend to use the original but slightly faulty heuristic anyway.

324. **Ruckert**, Martin. Continuous grammars. In *26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 303–310. ACM, 1999. Gives an example of a situation in which an error in the first token of the input can only be detected almost at the end of the input, invalidating almost all parsing done so far. To avoid such disasters, the author defines "continuous grammars", in which changing one token in the input can effect only limited changes in the parse tree: the mapping from string to parse tree is "continuous" rather than "discontinuous". This goal is achieved by imposing a metric for the distance between two nodes on a BCP grammar, and requiring that this distance is bounded by a constant for any single-token change in the input. It turns out that all bounded-context grammars are continuous; those bounded-context parsable are not, but can often be doctored.

325. **Cerecke**, Carl. Repairing syntax errors in LR-based parsers. *New Zealand J. Computing*, 8(3):3–13, June 2001. Improves McKenzie et al.'s algorithm [321] by limiting the lengths of circular search paths in the LR automaton. Left-recursive rules do not create cycles; right-recursive rules create cycles that have to be followed only once; and self-embedding rules create cycles that have to be followed until l symbols have been inserted, where l is the verification length. The improved parser solved 25% of the errors not solved by the original algorithm.

326. **Kim**, I.-S. and Choe, K.-M. Error repair with validation in LR-based parsing. *ACM Trans. Prog. Lang. Syst.*, 23(4):451–471, 2001. The combinations explored dynamically in McKenzie et al.'s algorithm [321] are computed statically during LR table generation, using a shortest-path search through the right-context graph.

327. **Corchuelo**, Rafael, Pérez, José A., Ruiz, Antonio, and Toro, Miguel. Repairing syntax errors in LR parsers. *ACM Trans. Prog. Lang. Syst.*, 24(6):698–710, Nov. 2002. The four LR parse action shift, reduce, accept, and reject are formalized as operators on a pair (stack, rest

of input). The error repair actions of an LR parser, insert, delete and forward move are described in the same formalism. ("Forward move" performs a limited number of LR parse actions, to see if there is another error ahead.)
The three error repair operators generate a search space, which is bounded by the depth of the forward move (N), the number of input tokens considered (N_t), and the maximum number of insertions (N_i) and deletions (N_d). The search space is searched breadth-first, with or without an error cost function; if the search space is found not to contain a solution, the system reverts to panic mode. The breadth-first search is implemented by a queue.
The system produces quite good but not superb error repair, is fast, and can easily be added to existing parsers, since it does not require additional tables and uses existing parsing actions only. With $N = 3$, $N_t = 10$, $N_i = 4$, and $N_d = 3$, the system almost always finds a solution; the solution is acceptable in about 85% of the cases. These results are compared to an extensive array of other error repair techniques.

328. **Jeffery**, Clinton L. Generating LR syntax error messages from examples. *ACM Trans. Prog. Lang. Syst.*, 25(5):631–640, Sept. 2003. The parser generator is provided with a list of error situations (pieces of incorrect code) with their desired error messages. The system then generates a provisional LR parser, runs it on each of the error situations, records in which state the parser ends up on which input token, and notes the triple (LR state, error token, error message) in a list. This list is then incorporated in the definitive parser, which will produce the proper error message belonging to the state and the input token, when it detects an error.

18.2.8 Incremental Parsing

329. **Lindstrom**, Gary. The design of parsers for incremental language processors. In *Second Annual ACM Symposium on Theory of Computing*, pages 81–91. ACM, 1970. The input is conceptually divided into "fragments" (substrings) by appointing by hand a set C of terminals that act as fragment terminators. Good candidates are separators like **`end`**, **`else`**, and **`;`**. Now for each non-terminal A in the grammar we create three new non-terminals: $<\!A$, which produces all prefixes of $\mathcal{L}(A)$ that end in a token in C, $A\!>$ for all suffixes, and $<\!A\!>$ for all infixes; rules for these are constructed. The input is then parsed by an LR parsing using these rules. The resulting fragments are saved and reused when the input is modified.
The parser does not know its starting state, and works essentially like the substring parser of Bates and Lavie [214], but the paper does not discuss the time complexity.

330. **Degano**, Pierpaolo, Mannucci, Stefano, and Mojana, Bruno. Efficient incremental LR parsing for syntax-directed editors. *ACM Trans. Prog. Lang. Syst.*, 10(3):345–373, July 1988. The non-terminals of a grammar are partitioned by hand into sets of "incrementally compatible" non-terminals, meaning that replacement of one non-terminal by an incrementally compatible one is considered a minor structural change. Like in Korenjak's method [53], for a partitioning in n sets $n+1$ parse tables are constructed, one for each set and one for the grammar that represents the connection between the sets. The parser user is allowed interactively to move or copy the string produced by a given non-terminal to a position where an incrementally compatible one is required. This approach keeps the text (i.e. the program text) reasonably correct most of the time and uses rather small tables.

331. **Vilares Ferro**, M. and Dion, B. A. Efficient incremental parsing for context-free languages. In *1994 International Conference on Computer Languages*, pages 241–252. IEEE Computer Society Press, May 1994. Suppose the GSS of a GLR parsing for a string w is available, and a substring $w_{i\cdots j}$ is replaced by a string u, possibly of different length. Two algorithms are supplied to update the GSS. In "total recovery" the smallest position $k \geq j$ is found such that all arcs (pops) from k reach back over i; the section $i \cdots k$ is then reparsed. Much technical detail is needed to make this work. "Partial recovery" preserves only those arcs that are completely to the right of the affected region. Extensive examples are given and many experimental results reported.

18.3 Parsers and Applications

18.3.1 Parser Writing

332. **Grau**, A. A. Recursive processes and ALGOL translation. *Commun. ACM*, 4(1):10–15, Jan. 1961. Describes the principles of a compiler for ALGOL 60, in which each entity in the language corresponds to a subroutine. Since ALGOL 60 is recursive in that blocks may contain blocks, etc., the compiler routines must be recursive (called "self-enslaving" in the paper); but the author has no compiler that supports recursive subroutines, so code segments for its implementation (routine entry, exit, stack manipulation, etc.) are provided. Which routine is called when is determined by the combination of the next input symbol and a state which is maintained by the parser. This suggests that the method is a variant of recursive ascent rather than of recursive descent.
The technique is demonstrated for a representative subset of ALGOL. In this demo version there are 13 states, determined by hand, and 17 token classes. The complete 13×17 matrix is provided; the contents of each entry is designed by considering exactly what must be done in that particular case.

333. **Conway**, Melvin E. Design of a separable transition-diagram compiler. *Commun. ACM*, 6(7):396–408, July 1963. The first to introduce coroutines and to apply them to structure a compiler. The parser is Irons' [2], made deterministic by a No-Loop Condition and a No-Backup Condition. It follows transition diagrams rather than grammar rules.

334. **Tarjan**, R. E. Depth first search and linear graph algorithms. *SIAM J. Computing*, 1(2):146–160, 1972. The power of depth-first search is demonstrated by two linear graph algorithms: a biconnectivity test and finding strongly connected components.
An undirected graph is *biconnected* if for any three nodes p, q, and r, you can go from p to q while avoiding r. The depth-first search on the undirected graph imposes a numbering on the nodes, which gives rise to beautiful palm trees.
A *strongly connected component* is a subset of the nodes of a directed graph such that for any three nodes p, q, and r in that subset, you can go from p to q while going through r.

335. **Aho**, A. V., Johnson, S. C., and Ullman, J. D. Deterministic parsing of ambiguous grammars. *Commun. ACM*, 18(8):441–452, 1975. Demonstrates how LL and LR parsers can be constructed for certain classes of ambiguous grammars, using simple disambiguating rules, such as operator-precedence.

336. **Glanville**, R. Steven and Graham, Susan L. A new method for compiler code generation (extended abstract). In *Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 231–240, 1978. SLR(1) parsing is used to structure the intermediate code instruction stream originating from a compiler front end. The templates of the target machine instructions form the grammar for the structuring; this grammar is almost always ambiguous and certainly not SLR(1). The parser actions are accompanied by actions that record semantic restrictions and costs. SLR(1) conflicts are resolved in 2 ways: upon shift/reduce conflicts the parser shifts; upon reduce/reduce conflicts the reduction with the longest reduce with the lowest cost which is compatible with the semantic restrictions is used. The parser cannot get stuck provided the grammar is "uniform". Conditions for a uniform grammar are given and full algorithms are supplied.

337. **Milton**, D. R., Kirchhoff, L. W., and Rowland, B. R. An ALL(1) compiler generator. *ACM SIGPLAN Notices*, 14(8):152–157, Aug. 1979. Presents an LL(1) parser generator and attribute evaluator which allows LL(1) conflicts to be solved by examining attribute values; the generated parsers use the error correction algorithm of Fischer, Milton and Quiring [303].

338. **Dencker**, Peter, Dürre, Karl, and Heuft, Johannes. Optimization of parser tables for portable compilers. *ACM Trans. Prog. Lang. Syst.*, 6(4):546–572, Oct. 1984. Given an $n \times m$ parser table, an $n \times m$ bit table is used to indicate which entries are error entries; this table is significantly smaller than the original table and the remaining table is now sparse (typically 90-98% don't-care entries). The remaining table is compressed row-wise (column-wise) by setting up

an interference graph in which each node corresponds to a row (column) and in which there is an edge between any two nodes the rows (columns) of which occupy an element in the same position. A (pseudo-)optimal partitioning is found by a minimal graph-coloring heuristic.

339. **Waite**, W. M. and Carter, L. R. The cost of a generated parser. *Softw. Pract. Exper.*, 15(3):221–237, 1985. Supports with measurements the common belief that compilers employing table-driven parsers suffer performance degradation with respect to hand-written recursive descent compilers. Reasons: interpretation of parse tables versus direct execution, attribute storage allocation and the mechanism to determine which action(s) to perform. Then, a parser interface is proposed that simplifies integration of the parser; implementation of this interface in assembly language results in generated parsers that cost the same as recursive descent ones. The paper does not consider generated recursive descent parsers.

340. **Aho**, A. V., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, Mass., 1986. The "Red Dragon Book". Excellent, UNIX-oriented treatment of compiler construction. Even treatment of the various aspects.

341. **Cohen**, Jacques and Hickey, Timothy J. Parsing and compiling using Prolog. *ACM Trans. Prog. Lang. Syst.*, 9(2):125–164, April 1987. See same paper [26] for parsing techniques in Prolog. Shows that Prolog is an effective language to do grammar manipulation in: computation of FIRST and FOLLOW sets, etc.

342. **Koskimies**, Kai. Lazy recursive descent parsing for modular language implementation. *Softw. Pract. Exper.*, 20(8):749–772, Aug. 1990. Actually, it is lazy *predictive* recursive descent parsing for LL(1) grammars done such that each grammar rule translates into an independent module which knows nothing of the other rules. But prediction requires tables and tables are not modular. So the module for a rule A provides a routine STA(A) for creating at parse time the "start tree" of A; this is a tree with A at the top and the tokens in FIRST(A) as leaves (but of course FIRST(A) is unknown). STA(A) may call STA routines for other non-terminals to complete the tree, but in an LL(1) grammar this process will terminate; special actions are required if any of these non-terminals produces ε.
When during parsing A is predicted and a is the input token, a is looked up in the leaves of the start tree of A, and the path from that leaf to the top is used to expand A (and possibly its children) to produce A. This technique is in between non-predictive recursive descent and LL(1). Full code and several optimization are given.

343. **Norvig**, P. Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics*, 17(1):91–98, March 1991. Shows a general top-down parser in Common Lisp, which is based on a function which accepts a non-terminal N and a sequence of tokens I as inputs and produces a list of the suffixes of I that remain after prefixes that are produced by N have been removed. The resulting parser has exponential complexity, and the author shows that by memoizing the function (and some others) the normal $O(n^3)$ complexity can be achieved, supplying working examples. But the generation process loops on left-recursive grammar rules.

344. **Frost**, Richard A. Constructing programs as executable attribute grammars. *Computer J.*, 35(4):376–389, 1992. Introduces 4 combinators for parsing and processing of input described by an attribute grammar. Emphasis is on attribute evaluation rather than on parsing.

345. **Hutton**, Graham. Higher-order functions for parsing. *J. Functional Programming*, 2(3):323–343, 1992. By having the concatenation (invisible) and the alternation (vertical bar) from the standard grammar notation as higher-order functions, parsers can be written that are very close to the original grammar. Such higher-order functions — functions that take functions as parameters — are called combinators. The paper explains in detail how to define and use them, with many examples The resulting parser does breadth-first recursive descent CF parsing, provided the grammar is not left-recursive. The semantics of a recognized node is passed on as an additional parameter.
The ideas are then used to implement a simple pocket calculator language. The tiny system consists

of a layout analyser, a lexical analyser, a scanner, and a syntax analyser, each only a few lines long; these are then combined into a parser in one line. Methods to restrict the search are discussed.

346. **Leermakers**, René, Augusteijn, Lex, and Kruseman Aretz, Frans E. J. A functional LR parser. *Theoret. Comput. Sci.*, 104:313–323, 1992. An efficient formulation of an LR parser in the functional paradigm is given, with proof of correctness. It can do LR(0), LALR(1) and GLR.

347. **Rekers**, J. *Parser Generation for Interactive Environments*. PhD thesis, Technical report, Leiden University, Leiden, 1992. Discusses several aspects of incremental parser generation, GLR parsing, grammar modularity, substring parsing, and SDF. Algorithms in Lisp provided.

348. **Bod**, R. Using an annotated language corpus as a virtual stochastic grammar. In *Proceedings of the 11th National Conference on Artificial Intelligence*, pages 778–783, Washington, DC, 1993. AAAI Press. A CF language is specified by a (large) set of annotated parse trees rather than by a CF grammar; this is realistic in many situations, including natural language learning. Probabilities are then derived from the set of trees, and parsing of new input strings is performed by weighted tree matching.

349. **Nederhof**, M.-J. and Sarbo, J. J. Efficient decoration of parse forests. In H. Trost, editor, *Feature Formalisms and Linguistic Ambiguity*, pages 53–78. Ellis Horwood, 1993. Concerns affix computation in AGFLs, of which the authors give a solid formal definition. Any CF method is used to obtain a parse forest. Each node in the forest gets a set of tuples, each tuple corresponding with one possible value set for its affixes. Expanding these sets of tuples would generate huge parse forests, so we keep the original parse forest and set up propagation equations. Sections ("cells") of the parse forest are isolated somewhat similar to basic blocks. Inside these cells, the equations are equalities; between the cells they are inclusions, somewhat similar to the dataflow equations between basic blocks. Additional user information may be needed to achieve uniqueness. Efficient implementations of the data structures are given.

350. **Frost**, R. A. Using memoization to achieve polynomial complexity of purely functional executable specifications of non-deterministic top-down parsers. *ACM SIGPLAN Notices*, 29(4):23–30, April 1994. The idea of obtaining a polynomial-time parser by memoizing a general one (see Norvig [343]) is combined with a technique to memoize functional-language functions, to obtain a polynomial-time parser in a functional language. A full example of the technique is given.

351. **Johnson**, Mark. Memoization in top-down parsing. *Computational Linguistics*, 21(3):405–418, 1995. Avoids the problem of non-termination of the creation of a list of suffixes in Norvig [343] by replacing the list by a function (a "continuation") which will produce the list when the times comes. Next the memoization is extended to the effect that a memo entry is prepared *before* the computation is made rather than after it. The author shows that in this setup left recursion is no longer a problem. Provides very clear code examples in Scheme.

352. **Kurapova**, E. W. and Ryabko, B. Y. Using formal grammars for source coding. *Problems of Information Transmission*, 31(1):28–32, 1995, (in Russian). The input to be compressed is parsed using a hand-written grammar and codes indicating the positions visited in the grammar are output; this stream is then compressed using Huffman coding. The process is reversed for decompression. Application to texts of known and unknown statistics is described, and the compression of a library of Basic programs using an LL(10) (!) character-level grammar is reported. The achieved results show a 10-30% improvement over existing systems. No explicit algorithms.

353. **Frost**, Richard A. and Szydlowski, Barbara. Memoizing purely functional top-down backtracking language processors. *Sci. Comput. Progr.*, 27(3):263–288, Nov. 1996. Using Hutton's combinators [345] yields a parser with exponential time requirements. This can be remedied by using memoization, bringing back the time requirement to the usual $O(n^3)$.

354. **Bhamidipaty**, A. and Proebsting, T. A. Very fast YACC-compatible parsers (for very little effort). *Softw. Pract. Exper.*, 28(2):181–190, 1998. Generate straightforward ANSI C code for each state of the LALR(1) parse table using **`switch`** statements, and let the C compiler worry over optimizations. The result is a *yacc*-compatible parser that is at most 30% larger, and about 4 times faster.

355. **Clark**, C. Build a tree — save a parse. *ACM SIGPLAN Notices*, 34(4):19–24, April 1999. Explains the difference between processing the nodes recognized during parsing on the fly and storing them as a tree. Obvious, but experience has shown that this has to be explained repeatedly.

356. **Sperber**, Michael and Thiemann, Peter. Generation of LR parsers by partial evaluation. *ACM Trans. Prog. Lang. Syst.*, 22(2):224–264, 2000. The techniques of Leermakers [155] are used to implement a recursive-ascent LR parser in Scheme. Constant propagation on the program text is then used to obtain a partial evaluation, yielding efficiencies that are comparable to those of *bison*.

357. **Metsker**, Steven John. *Building Parsers with Java*. Addison Wesley, 2001. Actually on how to implement "little languages" by using the toolkit package **`sjm.parse`**, supplied by the author. The terminology is quite different from that used in parsing circles. Grammars and non-terminals are hardly mentioned, but terminals are important. Each non-terminal corresponds to a parsing object, called a "parser", which is constructed from objects of class **`Repetition`**, **`Sequence`**, **`Alternation`** and **`Word`**; these classes (and many more) are supplied in the toolkit package. They represent the rule types $A \rightarrow B^*$, $A \rightarrow BC\cdots$, $A \rightarrow B|C|\cdots$, and $A \rightarrow t$, resp. Since each of these is implemented by calling the constructor of its components, B, C, ... cannot call A or a "parser class loop" would ensue; a special construction is required to avoid this problem (p. 105-106). But most little languages are not self-embedding anyway, except for the expression part, which is covered in depth.
The match method of a parser for A accepts a set of objects of class **`Assembly`**. An "assembly" contains a configuration (input string, position, and stack), plus an object representing the semantics of the part already processed. The match method of A produces another set of assemblies, those that appear after A has been matched and its semantics processed; the classes in the toolkit package just serve to lead these sets from one parser to the next. Assemblies that cannot be matched drop out; if there are FIRST/FIRST conflicts or FIRST/FOLLOW conflicts, assemblies are duplicated for each possibility. If at the end more than one assembly remains an error message is given; if none remains another error message is given. This implements top-down breadth-first parsing. It is interesting to see that this is an implementation of the 1962 "Multiple-Path Syntactic Analyzer" of Kuno and Oettinger [4].
The embedding in a programming language allows the match methods to have parameters, so very sophisticated context-sensitive matches can be programmed.
Chapters 1-9 explain how to build and test a parser; chapter 10 discusses some of the internal workings of the supplied classes; chapters 11-16 give detailed examples of implemented little languages, including a Prolog-like one, complete with unification; and chapter 17 gives further directions.
Tons of practical advice at a very manageable pace, allowing the user to quickly construct flexible parsers for little languages.

358. **Ljunglöf**, Peter. *Pure Functional Parsing: An Advanced Tutorial*. PhD thesis, Technical Report 6L, Chalmers University of Technology, Göteborg, April 2002. Consulted for its description of the Kilbury Chart Parser in Haskell. Assume the grammar to be in Chomsky Normal Form. Kilbury (chart) parsing proceeds from left to right, building up arcs marked with zero or more non-terminals A, which mean that A can produce the substring under the arc, and zero or more non-terminal pairs $B|C$, which mean that if this arc is connected on the right to an arc spanning C, both arcs together span a terminal production of B. For each token t, three actions are performed: Scan, Predict and Combine. Scan adds an arc spanning t, marked with all non-terminals that produce t. For each arc ending at and including t and marked A, Predict adds a mark $B|C$ to that arc for each rule $B \rightarrow AC$ in the grammar. For each arc ending at and including t, starting at position p, and marked A, Combine checks if there is an arc ending at p and marked $B|A$, and if so, adds an arc marked B, spanning both arcs.

The technique can be extended for arbitrary CF grammars. Basically, the markers are items, with the item $A \rightarrow \alpha\bullet\beta$ corresponding to the marker $A|\beta$.

359. **Sperberg-McQueen**, C. M. Applications of Brzozowski derivatives to XML schema processing. In *Extreme Markup Languages 2005*, page 26, Internet, 2005. IDEAlliance. Document descriptions in XML are based on "content models," which are very similar to regular expressions. It is important to find out if a content model C_1 "subsumes" a content model C_2, i.e., if there is a mapping such that the language of C_2 is included in the language of C_1. The paper shows how Brzozowski derivatives [138] can be used profitably for answering this and related questions.

18.3.2 Parser-Generating Systems

360. **Lesk**, M. E. and Schmidt, E. Lex: A Lexical Analyzer Generator. In *UNIX Manuals*, page 13. Bell Laboratories, Murray Hill, New Jersey, 1975. The regular grammar is specified as a list of regular expressions, each associated with a semantic action, which can access the segment of the input that matches the expression. Substantial look-ahead is performed if necessary. *lex* is a well-known and often-used lexical-analyser generator.

361. **Johnson**, Stephen C. YACC: Yet Another Compiler-Compiler. Technical report, Bell Laboratories, Murray Hill, New Jersey 07974, 1978. In spite of its title, *yacc* is one of the most widely used parser generators. It generates LALR(1) parsers from a grammar with embedded semantic actions and features a number of disambiguating and conflict-resolving mechanisms.

362. **Grune**, Dick and Jacobs, Ceriel J. H. A programmer-friendly LL(1) parser generator. *Softw. Pract. Exper.*, 18(1):29–38, Jan. 1988. Presents a practical ELL(1) parser generator, called *LLgen*, which generates fast error correcting recursive descent parsers. In addition to the error correction, *LLgen* features static as well as dynamic conflict resolvers and a separate compilation facility. The grammar can be viewed as a program, allowing for a natural positioning of semantic actions.

363. **Johnstone**, Adrian and Scott, Elizabeth. rdp: An iterator-based recursive descent parser generator with tree promotion operators. *ACM SIGPLAN Notices*, 33(9):87–94, Sept. 1998. Recursive descent parser generator with many add-ons: 1. A generalized BNF grammar structure `(` *expression* `)` *low* `@` *high separator*, which produces minimally *low* and maximally *high* productions of *expression*, separated by *separator*s. 2. Inlined extended ANSI-C code demarcated by `[*` and `*]`. 3. Inherited attributes as input parameters to grammar rules, and 1 synthetic attribute per grammar rule. This requires a rule to return two values: the Boolean success or failure value, and the synthetic attribute. An extended-code statement is provided for this. 4. Libraries for symbol tables, graph handling, scanning, etc. 5. Parse tree constructors, which allow the result of a subparse action to be attached to the parse tree in various places.
The parser is generalized recursive descent, for which see Johnstone and Scott [36].

18.3.3 Applications

364. **Kernighan**, B. W. and Cherry, L. L. A system for typesetting mathematics. *Commun. ACM*, 18(3):151–157, March 1975. A good example of the use of an ambiguous grammar to specify the preferred analysis of special cases.

365. **Share**, Michael. Resolving ambiguities in the parsing of translation grammars. *ACM SIGPLAN Notices*, 23(8):103–109, Aug. 1988. The UNIX LALR parser generator *yacc* is extended to accept LALR conflicts and to produce a parser that requests an interactive user decision when a conflict occurs while parsing. The system is used in document conversion.

366. **Evans**, William S. Compression via guided parsing. In *Data Compression Conference 1998*, pages 544–553. IEEE, 1998. To transmit text that conforms to a given grammar, the movements of the parser are sent rather than the text itself. For a top-down parser they are the rule numbers of the predicted rules; for bottom-up parsers they are the state transitions of the LR automaton. The packing problem is solved by adaptive arithmetic coding. The results are roughly 20% better than *gzip*.

367. **Evans**, William S. and Fraser, Christopher W. Bytecode compression via profiled grammar rewriting. *ACM SIGPLAN Notices*, 36(5):148–155, May 2001. The paper concerns the situation in which compressed bytecode is interpreted by on-the-fly decompression. The bytecode compression/decompression technique is based on the following observations.
1. Bytecode is repetitive and conforms to a grammar, so it can be represented advantageously as a parse tree in prefix form. Whenever the interpreter reaches a node representation, it knows the non-terminal (N) the node conforms to, exactly as with expressions in prefix form. The first byte of the node representation serves as a guiding byte and indicates which of the alternatives of the grammar rule N applies. This allows the interpreter again to know which non-terminal the next node conforms to, as required above.
2. Since non-terminals usually have few alternatives, most of the bits in the guiding bytes are wasted, and it would be better if all non-terminals had exactly 256 alternatives. One way to achieve this is to substitute some alternatives of some non-terminals in the alternatives of other non-terminals, thereby creating alternatives of alternatives, etc. This increases the number of alternatives per non-terminals and allows a more efficient representation of those subtrees of the parse tree that contain these alternatives of alternatives.
3. By choosing the substitutions so that the most frequent alternatives of alternatives are present in the grammar, a — heuristically — optimal compression can be achieved. The heuristic algorithm is simple: repeatedly substitute the most frequent non-terminal pair, unless the target non-terminal would get more than 256 alternatives in the process.
A few minor problems still have to be solved. The resulting grammar (expanded specifically for a given program) is ambiguous; an Earley parser is used to obtain the simplest — and most compact — parsing. Labels are dealt with as follows. All non-terminals that are ever a destination of a jump are made alternatives of the start non-terminal and parsing starts anew at each label. Special arrangements are made for linked-in code.
In one sample, the bytecode size was reduced from 199kB to 58kB, whereas the interpreter grew by 11kB, due to a larger grammar.

18.3.4 Parsing and Deduction

368. **Pereira**, Fernando C. N. and Warren, David H. D. Parsing as deduction. In *21st Annual Meeting of the Association for Computational Linguistics*, pages 137–144, Cambridge, Mass., 1983. The Prolog deduction mechanism is top-down depth-first. It can be exploited to do parsing, using Definite Clause grammars. Parsing can be done more efficiently with Earley's technique. The corresponding Earley deduction mechanism is derived and analysed.

369. **Vilain**, Marc. Deduction as parsing: Tractable classification in the KL-ONE framework. In *National Conf. on Artificial Intelligence (AAAI-91), Vol. 1*, pages 464–470, 1991. The terms in the frame language KL-ONE are restricted as follows. The number of possible instances of each logic variable must be finite, and the free (existential) terms must obey a partial ordering. A tabular Earley parser is then sketched, which solves the "deductive recognition" in $O(\kappa^{3\alpha})$, where κ is the number of constants in ground rules, and α is the maximum number of terms in a rule.

370. **Rosenblueth**, David A. Chart parsers as inference systems for fixed-mode logic programs. *New Generation Computing*, 14(4):429–458, 1996. Careful reasoning shows that chart parsing can be used to implement fixed-mode logic programs, logic programs in which the parameters can be divided into synthesized and inherited ones, as in attribute grammars. Good explanation of chart parsers. See also Rosenblueth [371].

371. **Rosenblueth**, David A. and Peralta, Julio C. SLR inference: an inference system for fixed-mode logic programs based on SLR parsing. *J. Logic Programming*, 34(3):227–259, 1998. Uses parsing to implement a better Prolog. When a logic language clause is written in the form of a difference list $a(X_0,X_n)$:-$b_1(X_0,X_1),b_2(X_1,X_2),\ldots,b_n(X_{n-1},X_n)$, it can be related to a grammar rule $A \rightarrow B_1B_2\cdots B_n$, and SLR(1) techniques can be used to guide the search process. Detailed explanation of how to do this, with proofs. Lots of literature references. See also Rosenblueth [370].

372. **Vilares Ferro**, Manuel and Alonso Pardo, Miguel A. An LALR extension for DCGs in dynamic programming. In Carlos Martín Vide, editor, *Mathematical and Computational Analysis of Natural Language*, volume 45 of *Studies in Functional and Structural Linguistics*, pages 267–278. John Benjamins, 1998. First a PDA is implemented in a logic notation. Next a control structure based on dynamic programming is imposed on it, resulting in a DCG implementation. The context-free backbone of this DCG is isolated, and an LALR(1) table for it is constructed. This LALR(1) automaton is made to run simultaneously with the DCG interpreter, which it helps by pruning off paths. An explanation of the possible moves of the resulting machine is provided.

373. **Morawietz**, Frank. Chart parsing and constraint programming. In *18th International Conference on Computational Linguistics: COLING 2000*, pages 551–557, Internet, 2000. ACL. The straight-forward application of constraint programming to chart parsing has the inference rules of the latter as constraints. This results in a very obviously correct parser, but is inefficient. Specific constraints for specific grammars are discussed.

374. **Erk**, Katrin and Kruijff, Geert-Jan M. A constraint-programming approach to parsing with resource-sensitive categorial grammar. In *Natural Language Understanding and Logic Programming (NLULP'02)*, pages 69–86, Roskilde, Denmark, July 2002. Computer Science Department, Roskilde University. The parsing problem is reformulated as a set of constraints over a set of trees, and an existing constraint resolver is used to effectuate the parsing.

18.3.5 Parsing Issues in Natural Language Handling

375. **Yngve**, Victor H. A model and an hypothesis for language structure. *Proceedings of the American Philosophical Society*, 104(5):444–466, Oct. 1960. To accommodate discontinuous constituents in natural languages, the Chomsky CF grammar is extended and the language generation mechanism is modified as follows. 1. Rules can have the form $A \rightarrow \alpha\cdots\beta$, where the $\cdots$ is part of the notation. 2. Derivations are restricted to leftmost only. 3. A sentential form $\phi_1 \bullet AX\phi_2$, where $\bullet$ indicates the position of the derivation front, leads to $\phi_1 A \bullet \alpha X \beta \phi_2$; in other words, the right-hand side surrounds the next symbol in the sentential form. 4. The A in 3 remains in the sentential form, to the left of the dot, so the result is a derivation tree in prefix form rather than a sentence. 5. The length of the part of the sentential form after the dot is recorded in the derivation tree with each non-terminal; it is relevant since it represents the amount of information the speaker needs to remember in order to create the sentence, the "depth" of the sentence. Linguistic properties of this new device are examined.
The hypothesis is then that languages tend to use means to keep the depths of sentence to a minimum. Several linguistic phenomena are examined and found to support this hypothesis.

376. **Dewar**, Hamish P., Bratley, Paul, and Thorne, James P. A program for the syntactic analysis of English sentences. *Commun. ACM*, 12(8):476–479, 1969. The authors argue that the English language can be described by a regular grammar: most rules are regular already and the others describe concatenations of regular sublanguages. The finite-state parser used constructs the state subsets on the fly, to avoid large tables. Features (attributes) are used to check consistency and to weed out the state subsets.

377. **Chester**, Daniel. A parsing algorithm that extends phrases. *Am. J. Computational Linguistics*, 6(2):87–96, April 1980. A variant of a backtracking left-corner parser is described that is particularly convenient for handling continuing phrases like: "the cat that caught the rat that stole the cheese".

378. **Woods**, William A. Cascaded ATN grammars. *Am. J. Computational Linguistics*, 6(1):1–12, Jan. 1980. The grammar (of a natural language) is decomposed into a number of grammars, which are then *cascaded*; that is, the parser for grammar G_n obtains as input the linearized parse tree produced by the parser for G_{n-1}. Each grammar can then represent a linguistic hypothesis. Such a system is called an "Augmented Transition Network" (ATN). An efficient implementation is given.

379. **Shieber**, Stuart M. Direct parsing of ID/LP grammars. *Linguistics and Philosophy*, 7:135–154, 1984. In this very readable paper, the Earley parsing technique is extended in a straightforward way to ID/LP grammars (Gazdar et al. [381]). The items are still of the form $A \rightarrow \alpha \bullet \beta, i$, the main difference being that the β in an item is understood as the set of LP-acceptable permutations of the elements of the β in the grammar rule. Practical algorithms are given.

380. **Blank**, Glenn D. A new kind of finite-state automaton: Register vector grammar. In *Ninth International Conference on Artificial Intelligence*, pages 749–756. UCLA, Aug. 1985. In FS grammars, emphasis is on the states: for each state it is specified which tokens it accepts and to which new state each token leads. In *Register-Vector grammars* (*RV grammars*) emphasis is on the tokens: for each token it is specified which state it maps onto which new state(s). The mapping is done through a special kind of function, as follows. The state is a (global) vector (array) of registers (features, attributes). Each register can be *on* or *off*. For each token there is a condition vector with elements which can be *on*, *off* or *mask* (= *ignore*); if the condition matches the state, the token is allowed. For each token there is a result vector with elements which can be *on*, *off* or *mask* (= *copy*); if the token is applied, the result-vector elements specify how to construct the new state. ε-moves are incorporated by having tokens (called *labels*) which have ε for their representation. Termination has to be programmed as a separate register.
RV grammars are claimed to be compact and efficient for describing the FS component of natural languages. Examples are given. Embedding is handled by having a finite number of levels inside the state.

381. **Gazdar**, Gerald, Klein, Ewan, Pullum, Geoffrey, and Sag, Ivan. *Generalized Phrase Structure Grammar*. Basil Blackwell Publisher, Ltd., Oxford, UK, 1985. The phrase structure of natural languages is more easily and compactly described using *Generalized Phrase Structure Grammars* (*GPSGs*) or *Immediate Dominance/Linear Precedence grammars* than using conventional CF grammars. Theoretical foundations of these grammars are given and the results are used extensively in linguistic syntactic theory. GPSGs are not to be confused with general phrase structure grammars, aka Chomsky Type 0 grammars, which are called "unrestricted" phrase structure grammars in this book.
The difference between GPSGs, ID/LP grammars and CF grammars is explained clearly. A GPSG is a CF grammar, the non-terminals of which are not unstructured names but sets of *features* with their values; such compound non-terminals are called *categories*. An example of a feature is **`NOUN`**, which can have the values **`+`** or **`-`**; **`<NOUN,+>`** will be a constituent of the categories "noun phrase", "noun", "noun subject", etc.
ID/LP grammars differ from GPSGs in that the right-hand sides of production rules consist of multisets of categories rather than of ordered sequences. Thus, production rules (Immediate Dominance rules) define vertical order in the production tree only. Horizontal order in each node is restricted through (but not necessarily completely defined by) Linear Precedence rules. Each LP rule is considered to apply to every node; this is called the *Exhaustive Constant Partial Ordering property*.

382. **Blank**, Glenn D. A finite and real-time processor for natural language. *Commun. ACM*, 32(10):1174–1189, Oct. 1989. Several aspects of the register-vector grammars of Blank [380] are treated and extended: notation, center-embedding (3 levels), non-determinism through boundary-backtracking, efficient implementation.

383. **Abney**, Steven P. and Johnson, Mark. Memory requirements and local ambiguities of parsing strategies. *J. Psycholing. Res.*, 20(3):233–250, 1991. Based on the fact that parse stack space in the human brain is severely limited and that left-corner parsing requires exactly 2 stack entries for left-branching constructs and exactly 3 for right-branching, the authors conclude that neither top-down nor bottom-up parsing can be involved, but left-corner can.

384. **Resnik**, Philip. Left-corner parsing and psychological plausibility. In *14th International Conference on Computational Linguistics*, pages 191–197. Association for Computational Linguistics, 1992. Argues that the moment of composition of semantics is more important than the parsing technique; also in this respect a form of left-corner parsing is compatible with human language processing.

18.4 Support Material

18.4.1 Formal Languages

385. **Chomsky**, Noam. On certain formal properties of grammars. *Inform. Control*, 2:137–167, 1959. This article discusses what later became known as the Chomsky hierarchy. Chomsky defines type 1 grammars in the "context-sensitive" way. His motivation for this is that it permits the construction of a tree as a structural description. Type 2 grammars exclude ε-rules, so in Chomsky's system, type 2 grammars are a subset of type 1 grammars.
Next, the so called *counter languages* are discussed. A counter language is a language recognized by a finite automaton, extended with a finite number of counters, each of which can assume infinitely many values. $L_1 = \{a^n b^n | n > 0\}$ is a counter language, $L_2 = \{xy | x,y \in \{a,b\}^*,\ y$ is the mirror image of $x\}$ is not, so there are type 2 languages that are not counter languages. The reverse is not investigated.
The Chomsky Normal Form is introduced, but not under that name, and a bit different: Chomsky calls a type 2 grammar *regular* if production rules have the form $A \to a$ or $A \to BC$, with $B \neq C$, and if $A \to \alpha A \beta$ and $A \to \gamma A \eta$ then $\alpha = \gamma$ and $\beta = \eta$. A grammar is self-embedding if there is a derivation $A \stackrel{*}{\to} \alpha A \beta$ with $\alpha \neq \varepsilon$ and $\beta \neq \varepsilon$. The bulk of the paper is dedicated to the theorem that the extra power of type 2 grammars over type 3 grammars lies in this self-embedding property.

386. **Bar-Hillel**, Y., Perles, M., and Shamir, E. On formal properties of simple phrase structure grammars. *Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung*, 14:143–172, 1961. (Reprinted in Y. Bar-Hillel, *Language and Information: Selected Essays on their Theory and Application*, Addison-Wesley, 1964, pp. 116-150.) Densely-packed paper on properties of context-free grammars, called *simple phrase structure grammars*, or SPGs here (this paper was written in 1961, two years after the introduction of the Chomsky hierarchy). All proofs are constructive, which makes the paper very important to implementers.
The main subjects are: any finite (one- and two-tape) automaton can be converted into a CF grammar; CF grammars are closed under reflection, union, product, and closure; CF grammars are not closed under intersection or complementation; almost any CF grammar can be made ε-free; almost any CF grammar can be made free of unit rules; it is decidable if a given CF grammar produces a given (sub)string; it is undecidable if the intersection of two CF grammars is a CF grammar; it is undecidable if the complement of a CF grammar is a CF grammar; it is undecidable if one CF grammar produces a sublanguage of another CF grammar; it is undecidable if one CF grammar produces the same language as another CF grammar; it is undecidable if a CF grammar produces a regular language; a non-self-embedding CF grammar produces a regular language; the intersection of a CF grammar and a FS automaton is a CF grammar.
Some of the "algorithms" described in this paper are impractical. For example, the decidability of parsing is proved by systematically producing all terminal productions up to the lengths of the input string, which is an exponential process. On the other hand, the intersection of a CF grammar and a

FS automaton is constructed in a time $O(n^d + 1)$, where n is the number of states in the automaton, and d is the maximum length of the RHSs in the grammar. This is the normal time complexity of general CF parsing. See also the same paper [219].

387. **Haines**, Leonard H. On free monoids partially ordered by embedding. *J. Combinatorial Theory*, 6:94–98, 1969. Proves that for any (infinite) set of words L (= subset of Σ^*) the following holds: 1. any language consisting of all subsequences of words in L is regular; 2. any language consisting of all words that contain subsequences of words in L is regular. This means that subsequence and supersequence parsing reduce to regular parsing.

388. **Cook**, Steven A. Linear time simulation of deterministic two-way pushdown automata. In *IFIP Congress (1)*, pages 75–80, 1971. Source of "Cook's Theorem": "Every 2-way deterministic pushdown automaton (2DPDA) language can be recognized in linear time on a random-access machine". A "proper arc" is a sequence of transitions of the 2DPDA for the given input that starts by pushing a stack symbol X, ends by popping the same X, and none of the in-between transitions pops the X. A "flat arc" is a single transition for the given input that neither pushes nor pops. Arcs are an efficient way to move the head over long distances without depending on or disturbing the stack underneath.
The algorithm starts by constructing all flat arcs, and from there builds all other arcs, until one connects the initial state to one of the final states. Since $|S|$ arcs can start at any point of the input, where $|S|$ is the number of transitions in the 2DPDA, and since each such arc has only one end point because the automaton is deterministic, there are only $|S|n$ arcs. The algorithm computes them so that no arc gets computed twice, so the algorithm is linear.
The theorem has many unexpected applications; see for example Aho's survey of algorithms for finding patterns in strings [147].

389. **Greibach**, Sheila A. The hardest context-free language. *SIAM J. Computing*, 2(4):304–310, Dec. 1973. The grammar is brought in Greibach Normal Form (Greibach [7]. Each rule $A \rightarrow aBCD$ is converted into a mapping $a \Rightarrow \bar{A}DCB$, which should be read as: "a can be replaced by a cancellation of prediction A, followed by the predictions D, C, and B, that is, in back-to-front order."
These mappings are used as follows. Suppose we have a grammar `S→aBC; B→b; C→c`, which yields the maps `a` $\Rightarrow$ `S̄CB`, `b` $\Rightarrow$ `B̄`, and `c` $\Rightarrow$ `C̄`. Now the input `abc` maps to `S̄CBB̄C̄`, which is prefixed with the initial prediction `S` to form `SS̄CBB̄C̄`. We see that when we view A and $\bar{A}$ as matching parentheses, we have obtained a well-balanced parenthesis string (wbps), and in fact the mapping of any correct input will be well balanced.
This makes parsing seem trivial, but in practice there will be more than one mapping for each terminal, and we have to chose the right one to get a wbps. The alternatives for each terminal are worked into the mapping by demarcating them with markers and separators, such that the mapping of any correct input maps to a conditionally well-balanced parenthesis string (cwbps), the condition being that the right segments are matched. These cwbpses form a CF language which depends on the symbols of the grammar only; the rules have been relegated to the mapping. (It is not shown that the cwbpses are a CF set.)
The dependency on the symbols of the grammar is removed by expressing them in unary notation: `B`, being the second non-terminal, is represented as `[xx[`, and `B̄` as `]x̄x̄]`, etc. With this representation, the cwbpses are not dependent on any grammar any more and any parsing problem can be transformed into them in linear time. So if we can parse cwbpses in time $O(n^x)$, we can parse any CF language in time $O(n^x)$, which makes cwbpses the *hardest context-free language*.

390. **Liu**, Leonard Y. and Weiner, Peter. An infinite hierarchy of intersections of context-free languages. *Math. Syst. Theory*, 7(2):185–192, May 1973. It is easy to see that the language $a^m b^n c^p \cdots a^m b^n c^p \cdots$ where there are k different $a, b, c \cdots$s, can be generated as the intersection of k CF languages: take for the first language $a^m b^* c^* \cdots a^m b^* c^* \cdots$, for the second language $a^* b^n c^* \cdots a^* b^n c^* \cdots$, etc. The authors then give a 6-page proof showing that the same cannot be achieved with $k-1$ languages; this proves the existence of the subject in the title.

391. **Hopcroft**, John E. and Ullman, Jeffrey D. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachussetts, 1979. No-frills account of

formal language theory and computational (im)possibilities. Covers CYK and LR parsers, but as recognizers only.

392. **Heilbrunner**, Stephan. Tests for the LR-, LL-, and LC-regular conditions. *J. Comput. Syst. Sci.*, 27(1):1–13, 1983. Careful analysis shows that the LR-regular test in Čulik, II and Cohen's paper [57] is not correct. The repair leads to item grammars, which are right-regular grammars in which items are non-terminals. This mechanism is then used for very precise tests for LR-, LL-, and LC-regular-ness. Some proofs are given, but others are referred to a technical report.

393. **Rayward-Smith**, V. J. *A First Course in Formal Languages*. Blackwell Scientific, Oxford, 1983. Very useful intermediate between Révész [394] and Hopcroft and Ullman [391]. Quite readable (the subject permitting); simple examples; broad coverage. No treatment of LALR, no bibliography.

394. **Révész**, György E. *Introduction to Formal Languages*. McGraw-Hill, Singapore, 1985. This nifty little book contains many results and elementary proofs of formal languages, without being "difficult". It gives a description of the ins and outs of the Chomsky hierarchy, automata, decidability and complexity of context-free language recognition, including the hardest context-free language. Parsing is discussed, with descriptions of the Earley, LL(k) and LR(k) algorithms, each in a few pages.

395. **Geffert**, Viliam. A representation of recursively enumerable languages by two homomorphisms and a quotient. *Theoret. Comput. Sci.*, 62:235–249, 1988. Imagine the following mechanism to generate strings. The mechanism uses two homomorphisms h_1 and h_2 (a *homomorphism* is a translation table from tokens to strings of zero or more tokens) and an alphabet Σ; the tokens in the translation tables may or may not be in Σ. Now take an arbitrary string α, and construct the two translations $h_1(\alpha)$ and $h_2(\alpha)$. If it now so happens that $h_2(\alpha) = h_1(\alpha)w$ (so $h_1(\alpha)$ is the head of $h_2(\alpha)$ and w is the tail), and w consists of tokens that all happen to be in the alphabet Σ, then we keep w; otherwise α leads nowhere.
The author shows that this mechanism is equivalent to a Chomsky Type 0 grammar, and that the grammar defines the two homomorphisms and vice versa. The details are complicated, but basically h_1 and h_2 are such that as α grows, h_2 grows faster than h_1. The consequence is that if we want to extend α by a few tokens δ, the translation of δ through h_1 must match tokens already produced long ago by $h_2(\alpha)$ or α will be rejected; so very soon our hand is forced. This effect is used to enforce the long-range relationships characteristic of general phrase-structure grammars. In fact, α is some encoding of the derivation of w.

396. **Billington**, David. Using the context-free pumping lemma. *Commun. ACM*, 36(4):21, 81, April 1993. Short note showing a somewhat sharper lemma, better suited for proving that a language is not CF.

397. **Sudkamp**, Thomas A. *Languages and Machines*. Addison-Wesley, second edition, 1997. Carefully reasoned, very readable, but sometimes dull introduction to formal languages, with serious attention to grammars and parsing. FSA minimization, grammar manipulation, proof techniques for the equivalence of a grammar and a language, same for non-existence of a grammar for a language, etc. Many fully worked out examples.
Consists of five parts: CF grammars and parsing; automata and languages; decidability and computation; computational complexity; deterministic parsing.

398. **Schmitz**, Sylvain. Conservative ambiguity detection in context-free grammars. Technical Report RR-2006-30-FR, Université de Nice, Nice, 2006. A grammar G is represented in a way similar to Figure 9.48. On the basis of this representation an infinite graph is defined in which each node represents a rightmost sentential form of G. G is ambiguous of there is more than one path from a given node to another node in this graph. The infinite graph is rendered finite by defining equivalence relations between nodes that preserve the multiple paths if they exist. Testing the finite graph for multiple paths is simple. A lattice of possible equivalence relations is presented. The time complexity is $O(|G|^2|T|^{4k})$, where $|G|$ is the size of the grammar, $|T|$ is the number of terminals, and k depends on the equivalence relation.

18.4.2 Approximation Techniques

399. **Pereira**, Fernando C. N. and Wright, Rebecca N. Finite-state approximation of phrase-structure grammars. In *29th Annual Meeting of the Association for Computational Linguistics*, pages 246–255. Association for Computational Linguistics, 1991. The idea is to "flatten" the LR(0) automaton of a grammar G into an FSA that will accept any string from G, and not very much more. But the LR(0) automaton stops at reduce states, whereas the FSA has to continue. For any state s which contains an item $A \to \alpha\bullet$, all paths are searched backwards to states t where the item started. (Cf. the **lookback** relation from Section 9.7.1.3.) Each t_i has a transition on A to a state u_i. Now ε-transitions are added from s to each u_i.
This is the version that keeps no stack at all. It can be improved by keeping finite simplifications of the stack, and several variants are examined in great detail and with full theoretical support. For all left-linear and right-linear grammars and some CF grammars the approximation is exact.

400. **Pereira**, Fernando C. N. and Wright, Rebecca N. Finite-state approximation of phrase-structure grammars. In Emmanuel Roche and Yves Schabes, editors, *Finite-State Language Processing*, pages 149–173. MIT Press, 1997. The LR(0) automaton of a grammar is "flattened" by ignoring the stack and replacing any reduction to A by an ε-transition to all states with incoming arrows marked A. This yields too coarse automata, even for regular and finite grammars. Rather than ignoring the stack, stack configurations are simulated and truncated as soon as they begin to repeat. This yields unwieldy automata. To remedy this the grammar is first decomposed into subgrammars by isolating strongly connected components in the grammar graph. Full algorithms and proofs are given. Sometimes the grammar needs to be modified (left-factored) to avoid exponential blow-up. See also [399, 404].

401. **Nederhof**, Mark-Jan. Regular approximations of CFLs: A grammatical view. In H. Bunt and A. Nijholt, editors, *Advances in Probabilistic and Other Parsing Technologies*, pages 221–241. Kluwer Academic Publishers, 2000. A regular envelope of a CF grammar is constructed by finding the self-embedding rules in it and splitting them in a left-recursive and a right-recursive persona. Many other regular approximating algorithms are discussed and compared.

402. **Nederhof**, Mark-Jan. Practical experiments with regular approximation of context-free languages. *Computational Linguistics*, 26(1):17–44, 2000. A regular envelope of a CF grammar is constructed by assigning a start state and a stop state to each non-terminal and $m+1$ intermediate states to each rule $A \to X_1 \cdots X_m$. These states are then connected by transitions, to form a transition network for the entire grammar. Properties of this approximation are investigated, and the algorithm is refined. It is compared empirically to other algorithms, where it proves effective, especially for large grammars.

403. **Yli-Jyrä**, Anssi. Regular approximations through labeled bracketing. In *Formal Grammar 2003*, pages 189–201. European Summer School in Logic Language and Information, 2003. A CF language consists of a nesting component, described by a grammar for nesting brackets (i.e. the sections of text that are forced to nest), and a regular component, which describes the shapes of these brackets. The nesting part can be decomposed in separate grammars for each nesting set of brackets. By judiciously restricting the various components, good and compact approximations to CF languages can be obtained. Properties of the various possibilities are examined.

404. **Pereira**, Fernando C. N. and Wright, Rebecca N. Finite-state approximation of phrase-structure grammars. Technical report, AT&T Reseach, Murray Hill, NJ, March 2005. Revised and extended version of Pereira and Wright [399, 400].

18.4.3 Transformations on Grammars

405. **Foster**, J. M. A syntax-improving program. *Computer J.*, 11(1):31–34, May 1968. The parser generator SID (Syntax Improving Device) attempts to remove LL(1) conflicts by eliminat-

ing left recursion, and then left-factoring, combined with inline substitution. If this succeeds, SID generates a parser in machine language.

406. **Hammer**, Michael. A new grammatical transformation into LL(k) form. In *Sixth Annual ACM Symposium on Theory of Computing*, pages 266–275, 1974. First an LR(k) automaton is constructed for the grammar. For each state that predicts only one non-terminal, say A, a new LR(k) automaton is constructed with A as start symbol, etc. This process splits up the automaton into many smaller ones, each using a separate stack, hence the name "multi-stack machine". For all LL(k) grammars this multi-stack machine is cycle-free, and for many others it can be made so, using some heuristics. In that case the multi-stack machine can be converted to an LL(k) grammar. This works for all LC(k) grammar and more. An algorithm for repairing the damage to the parse tree is given. No examples.

407. **Mickunas**, M. D., Lancaster, R. L., and Schneider, V. B. Transforming LR(k) grammars to LR(1), SLR(1) and (1,1) bounded right-context grammars. *J. ACM*, 23(3):511–533, July 1976. The required look-ahead of k tokens is reduced to $k-1$ by incorporating the first token of the look-ahead into the non-terminal; this requires considerable care. The process can be repeated until $k=1$ for all LR(k) grammars and even until $k=0$ for some grammars.

408. **Rosenkrantz**, D. J. and Hunt, H. B. Efficient algorithms for automatic construction and compactification of parsing grammars. *ACM Trans. Prog. Lang. Syst.*, 9(4):543–566, Oct. 1987. Many grammar types are defined by the absence of certain conflicts: LL(1), LR(1), operator-precedence, etc. A simple algorithm is given to modify a given grammar to avoid such conflicts. Modification is restricted to the merging of non-terminals and possibly the merging of terminals; semantic ambiguity thus introduced will have to be cleared up by later inspection. Proofs of correctness and applicability of the algorithm are given. The maximal merging of terminals while avoiding conflicts is also used to reduce grammar size.

18.4.4 Miscellaneous Literature

This section contains support material that is not directly concerned with parsers or formal languages.

409. **Warshall**, Stephen. A theorem on boolean matrices. *J. ACM*, 9(1):11–12, 1962. Describes how to obtain B^*, where B is an $n \times n$ Boolean matrix, in $O(n^3)$ actions, using a very simple 3-level loop.

410. **Michie**, D. "memo" functions and machine learning. *Nature*, 218(5136):19–22, April 6 1968. Recognizes that a computer function should behave like a mathematical function: the input determines the output, and how the calculation is done is immaterial, or at least behind the screens. This idea frees the way for alternative implementations of a given function, in this case by memoization.
A function implementation consists of a rote part and a rule part. The rote part contains the input-to-output mappings the function has already learned by rote, and the rule provides the answer if the input is new. New results are added to the rote part and the data is ordered in order of decreasing frequency by using a self-organizing list. This way the function "learns" answers by rote as it is being used. The list is fixed-size and if it overflows, the least popular element is discarded. Several examples of applications are given.

411. **Bhate**, Saroja and Kak, Subhash. Pāṇini's grammar and computer science. *Annals of the Bhandarkar Oriental Research Institute*, 72:79–94, 1993. In the Aṣṭādhyāyī, the Sanskrit scholar Pāṇini (probably c. 520-460 B.C.) gives a complete account of the Sanskrit morphology in 3,959 rules. The rules are context-free substitution rules with simple context conditions, and can be interpreted mechanically. The basic form is $A \rightarrow B(C)$, which means that A must be replaced by B if condition C applies; in the Sanskrit text, the separators $\rightarrow$, (and) are expressed through case endings. Macros are defined for many ordered sets. For example, the rule *iko yaṇ aci* means: *i*, *u*,

ṛ, and *ḷ* must be replaced by *y*, *v*, *r* and *l* respectively, when a vowel follows. All three words are macros: *ikaḥ* stands for the ordered set *iuṛḷ*; *yaṇ* stands for *yvrl*; and *ac* stands for all vowels. The rules literally means "of-*ikaḥ* [must come] *yaṇ* at-*ac*".
The rules differ from that of a CF grammar: 1. they have context conditions; 2. replacement is from a member of an ordered set to the corresponding member of the other ordered set; 3. the rules are applied in the order they appear in the grammar; 4. rule application is obligatory. Because of this Pāṇini's rules are more similar to a Unix *sed* script.
The authors explain several other features, all in computer science terms, and consider further implications for computer science and linguistics.

412. **Nuutila**, Esko. An efficient transitive closure algorithm for cyclic digraphs. *Inform. Process. Lett.*, 52(4):207–213, Nov. 1994. Very careful redesign of the top-down transitive closure algorithm using strongly connected components, named COMP_TC. Extensive experimental analysis, depicted in 3D graphs.

413. **Thompson**, Simon. *Haskell: The Craft of Functional Programming*. Addison Wesley, Harlow, England, 2nd edition, March 1999. Functional programming in Haskell and how to apply it. Section 17.5 describes a straightforward top-down parser; it is formulated as a monad on page 405.

414. **Grune**, Dick, Bal, Henri E., Jacobs, Ceriel J. H., and Langendoen, Koen G. *Modern Compiler Design*. John Wiley, Chichester, UK, 2000. Describes, among other things, LL(1), LR(0) and LR(1) parsers and attribute grammar evaluators in a compiler-design setting.

415. **Cormen**, Thomas H., Leiserson, Charles E., Rivest, Ronald L., and Stein, Clifford. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001. Extensive treatment of very many subjects in algorithms, breadth-first search, depth-first search, dynamic programming, on which it contains a large section, topological sort, etc., etc.

416. **Goodrich**, Michael T. and Tamassia, Roberto. *Algorithm Design: Foundations, Analysis, and Internet Examples*. John Wiley and Sons, 2nd edition, 2002. Low-threshold but extensive and in-depth coverage of algorithms and their efficiency, including search techniques, dynamic programming, topological sort.

417. **Sedgewick**, Robert. *Algorithms in C/C++/Java: Fundamentals, Data Structures, Sorting, Searching, and Graph Algorithms*. Addison-Wesley, Reading, Mass., 2001/2002. Comprehensive, understandable treatment of many algorithms, beautifully done. Available for C, C++ and Java.