Proc. 12th Int. Conf. on Computational Linguistics - COLING'88, (Dénes Vargha ed.), Budapest (Hungary), 22-27 August 1988, vol.1, pp. 365-371.

Parsing Incomplete Sentences

Bernard LANG
INRIA
B.P. 105, 78153 Le Chesnay, France
lang@inria.inria.fr

Abstract

An efficient context-free parsing algorithm is presented that can parse sentences with unknown parts *of unknown length*. It produces in finite form all possible parses (often infinite in number) that could account for the missing parts. The algorithm is a variation on the construction due to Earley. However, its presentation is such that it can readily be adapted to any chart parsing schema (top-down, bottom-up, etc...).

1 Introduction

It is often necessary in practical situations to attempt parsing an incorrect or incomplete input. This may take many forms: e.g. missing or spurious words, misspelled or misunderstood or otherwise unknown words [28], missing or unidentified word boundaries [22, 27]. Specific techniques may be developed to deal with these situations according to the requirements of the application area (e.g. natural language processing, programming language parsing, real-time or off-line processing).

The *context-free (CF)* parsing of a sentence with unknown words has been considered by other authors [28]. Very simply, an unknown word may be considered as a *"special multi-part-of-speech word whose part of speech can be anything"*. This multi-part-of-speech word need not be introduced in the CF grammar of the language, but only implicitly in the construction of its parser. This works very well with Earley-like (chart) parsers that can simulate all possible parsing paths that could lead to a correct parse.

In this paper, we deal with the more complex problem of parsing a sentence for which one or several subparts *of unknown length* are missing. Again we can use a chart parser to try all possible parses *on all possible inputs*. However the fact that the length of the missing subsequence is unknown raises an additional difficulty. Many published chart parsers [24, 28, 23, 21] are constructed with the assumption that the CF grammar of the language has no cyclic rules. This hypothesis is reasonable for the syntax of natural (or programming) languages. However the resulting simplification of the parser construction does not allow its extension to parsing sentences with unknown subsequences of words.

If the length (in words) of the missing subsequence were known, we could simply replace it with as many unknown words, a problem we know how to handle. When this length is not known, the algorithm has to simulate the parsing of an arbitrary number of words, and thus may have to go several times through reduction by the same rules of the grammar[1] without ever touching the stack present before scanning the unknown sequence, and without reading the input beyond that sequence. If we consider the unknown sequence as a special input word, we are in a situation that is analogous to that created by cyclic grammars, i.e. grammars where a nonterminal may derive onto itself without producing any terminal. This explains why techniques limited to non-cyclic grammars cannot deal with this problem.

It may be noted that the problem is different from that of parsing in a word lattice [22, 27] since all possible path in the lattice have a known bounded length, even when the lattice contains separated unknown words. However the technique presented here combines well with word lattice parsing.

The ability to parse unknown subsequences may be useful to parse badly transmitted sentences, and sentences that are interrupted (e.g. in a discussion) or otherwise left unfinished (e.g. because the rest may be inferred from the context). It may also be used in programming languages: for example the programming language SETL [9] allows some statements to be left unfinished in some contexts.

The next section contains an introduction to all-paths parsing. In section 3 we give a more detailed account of our basic algorithm and point at the features that allow the handling of cyclic grammars. Section 4 contains the modifications that make this algorithm capable of parsing incomplete sentences. The full algorithm is given in appendix C, while two examples are given in appendices A and B.

[1] This grammar oriented view of the computation of the automaton is only meant as a support for intuition.

2 All-Paths Parsing

Since Earley's first paper [10], many adaptations or improvements of his algorithm have been published [6, 5, 24, 28]. They are usually variations following some chart parsing schema [16]. In a previous paper [18], the author attempted to unify all these results by proposing an Earley-like construction for all-paths interpretation of (non-deterministic) *Push-Down-Transducers (PDT)*. The idea was that left-to-right parsing schemata may usually be expressed as a construction technique for building a recognizing *Push-Down-Automaton (PDA)* from the CF grammar of the language. This is quite apparent when comparing the PDA constructions in [12] to the chart schemata of [16] which are now a widely accepted reference. Thus a construction proposed for general PDTs is de facto applicable to most left-to-right parsing schemata, and allows in particular the use of well established PDT construction techniques (e.g. precedence, LL(k), LR(k) [8, 14, 2]) for general CF parsing.

In this earlier paper, our basic algorithm is proved correct, and its complexity is shown to be $O(n^3)$, i.e. as good as the best general parsing algorithms[2]. As is usual with Earley's construction[3], the theoretical complexity bound is rarely attained, and the algorithm behaves linearly most of the time. Further optimizations are proposed in [18] that improve this behavior.

Most published variants of Earley's algorithm, including Earley's own, may be viewed as (a sometimes weaker form of) our construction applied to some specific PDA or PDT. This is the explicit strategy of Tomita [28] in the special case of LALR(1) PDT construction technique. A notable exception is the very general approach of Sheil [25], though it is very similar to a Datalog extension [19] of the algorithm presented here.

An essential feature of all-paths parsing algorithms is to be able to produce all possible parses in a concise form, with as much sharing as possible of the common subparses. This is realized in many systems [6, 24, 28] by producing some kind of *shared-forest* which is a representation of all parse-trees with various sharings of common subparts. In the case of our algorithm, a parse is represented by the sequence of rules to be used in a left-to-right reduction of the input sentence to the initial nonterminal of the grammar. Sharing between all possible parses is achieved by producing, instead of an extensionally given set of possible parse sequences, a new CF grammar that generates all possible parse sequences (possibly an infinite number if the grammar of the input language is cyclic, and if the parsed sentence is infinitely ambiguous). With appropriate care, it is also possible to read this output grammar as a shared-forest (see appendix A). However its meaningful interpretation as a shared-forest is dependent on the parsing schema (cf. [12, 16]) used in constructing the PDT that produces it as output. Good definition and understanding of shared forests is essential to properly define and handle the extra processing needed to disambiguate a sentence, in the usual case when the ambiguous CF grammar is used only as a parsing backbone [24, 26]. The structure of shared forests is discussed in [4].

Before and while following the next section, we suggest that the reader looks at Appendix A which contains a detailed example showing an output grammar and the corresponding shared forest for a slightly ambiguous input sentence.

3 The Basic Algorithm

A formal definition of the extended algorithm for possibly incomplete sentences is given in appendix C. The formal aspect of our presentation of the algorithm is justified by the fact that it allows specialization of the given constructions to specific parsing schema without loss of the correctness and complexity properties, as well as the specialization of the optimization techniques (see [18]) established in the general case. The examples presented later were obtained with an adaptation of this general algorithm to bottom-up LALR(1) parsers [8].

Our aim is to parse sentences in the language $\mathcal{L}(\mathbf{G})$ generated by a CF phrase structure grammar $\mathbf{G} = (\mathbf{V}, \mathbf{\Sigma}, \mathbf{\Pi}, \mathring{\mathrm{N}})$ according to its syntax. The notation used is $\mathbf{V}$ for the set of nonterminal, $\mathbf{\Sigma}$ for the set of terminals, $\mathbf{\Pi}$ for the rules, and $\mathring{\mathrm{N}}$ for the initial nonterminal.

We assume that, by some appropriate parser construction technique (e.g. [14, 8, 2, 1]) we mechanically produce from the grammar $\mathbf{G}$ a parser for the language $\mathcal{L}(\mathbf{G})$ in the form of a (possibly non-deterministic) *push-down transducer (PDT)* $\mathcal{T}_{\mathbf{G}}$. The output of each possible computation of the parser is a sequence of rules in $\mathbf{\Pi}$[4] to be used in a left-to-right reduction of the input sentence (this is obviously equivalent to producing a parse-tree).

We assume for the PDT $\mathcal{T}_{\mathbf{G}}$ a very general formal definition that can fit most usual PDT construction techniques. It is defined as an 8-tuple $\mathcal{T}_{\mathbf{G}} = (\mathbf{Q}, \mathbf{\Sigma}, \mathbf{\Delta}, \mathbf{\Pi}, \delta, \mathring{\mathrm{q}}, \mathring{\$}, \mathbf{F})$ where: $\mathbf{Q}$ is the set of states, $\mathbf{\Sigma}$ is the set of input word symbols, $\mathbf{\Delta}$ is the set of stack symbols, $\mathbf{\Pi}$ is the set of output symbols (*i.e. rules of* $\mathbf{G}$), $\mathring{\mathrm{q}}$ is the initial state, $\mathring{\$}$ is the initial stack symbol, $\mathbf{F}$ is the set of final states, δ is a finite set of transitions of the form: $(\mathrm{p\ A\ a} \mapsto \mathrm{q\ B}\ u)$ with $\mathrm{p, q} \in \mathbf{Q}$, $\mathrm{A,B} \in \mathbf{\Delta} \cup \{\epsilon_{\mathbf{\Delta}}\}$, $\mathrm{a} \in \mathbf{\Sigma} \cup \{\epsilon_{\mathbf{\Sigma}}\}$, and $u \in \mathbf{\Pi}^{\star}$.

[2] Theoretically faster algorithms [29, 7] can achieve $O(n^{2.496})$ but with an unacceptable constant factor. Note also that we do not require the grammar to be in Chomsky Normal Form.

[3] And unlike tabular algorithms such as Cocke-Younger-Kasami's [13, 15, 30, 11].

[4] Implementations usually denote these rules by their index in the set $\mathbf{\Pi}$.

Let the PDT be in a configuration $\rho = (\mathrm{p}\ \mathrm{A}\alpha\ \mathrm{a}x\ u)$ where p is the current state, $\mathrm{A}\alpha$ is the stack contents with A on the top, $\mathrm{a}x$ is the remaining input where the symbol a is the next to be shifted and $x \in \mathbf{\Sigma}^\star$, and u is the already produced output. The application of a transition $\tau = (\mathrm{p}\ \mathrm{A}\ \mathrm{a} \mapsto \mathrm{q}\ \mathrm{B}\ v)$ results in a new configuration $\rho' = (\mathrm{q}\ \mathrm{B}\alpha\ x\ uv)$ where the terminal symbol a has been *scanned* (i.e. *shifted*), A has been popped and B has been pushed, and v has been concatenated to the existing output u. If the terminal symbol a is replaced by $\epsilon_{\mathbf{\Sigma}}$ in the transition, no input symbol is scanned. If A (resp. B) is replaced by $\epsilon_{\mathbf{\Delta}}$ then no stack symbol is popped from (resp. pushed on) the stack.

Our algorithm consists in an Earley-like[5] simulation of the PDT $\mathcal{T}_{\mathbf{G}}$. Using the terminology of [2], the algorithm builds an *item set* $\mathcal{S}_i$ successively for each word symbol x_i holding position i in the input sentence x. An item is constituted of two *modes* of the form $(\mathrm{p}\ \mathrm{A}\ i)$ where p is a PDT state, A is a stack symbol, and i is the index of an input symbol. The item set $\mathcal{S}_i$ contains *items* of the form $((\mathrm{p}\ \mathrm{A}\ i)\ (\mathrm{q}\ \mathrm{B}\ j))$. These items are used as nonterminals of a grammar $\mathcal{G} = (\mathcal{S}, \mathbf{\Pi}, \mathcal{P}, \mathrm{U_f})$, where $\mathcal{S}$ is the set of all items (i.e. the union of $\mathcal{S}_i$), and the rules in $\mathcal{P}$ are constructed together with their left-hand-side item by the algorithm. The initial nonterminal $\mathrm{U_f}$ of $\mathcal{G}$ derives on the last items produced by a successful computation.

The meaning of an item $\mathrm{U} = ((\mathrm{p}\ \mathrm{A}\ i)\ (\mathrm{q}\ \mathrm{B}\ j))$ is the following:

- there are computations of the PDT on the given input sentence that reach a configuration ρ' where the state is p, the stack top is A and the last symbol scanned is x_i;
- the next stack symbol is then B and, for all these computations, it was last on top in a configuration ρ where the state was q and the last symbol scanned was x_j;
- the rule sequences in $\mathbf{\Pi}^\star$ derivable from U in the grammar $\mathcal{G}$ are exactly those sequences output by the above defined computations of the PDT between configurations ρ and ρ'.

In simpler words, an item may be understood as a set of distinguished fragments of the possible PDT computations, that are independent of the initial content of the stack, except for its top element. Item structures are used to share these fragments between all PDT computations that can use them, so as to avoid duplication of work. In the output grammar an item is a nonterminal that may derive on the outputs produced by the corresponding computation fragments.

The items may also be read as an encoding of the possible configurations that could be attained by the PDT on the given input, with sharing of common stack fragments (the same fragment may be reused several times for the same stack in the case of cyclic grammars, or incomplete sentences). In figure 1 we represent a partial collection of items. Each item is represented by its two modes as $(K_h\ K_{h'})$ without giving the internal structure of modes as a triples (PDT-state $\times$ stack-symbol $\times$ input-index). Each mode K_h actually stands for the triple $(\mathrm{p}_h\ \mathrm{A}_h\ i_h)$. We have added arrows from the second component of every item $(K_h\ K_{h'})$ to the first component of any item $(K_{h'}\ K_{h''})$. This chaining indicates in reverse the order in which the corresponding modes are encountered during a possible computation of the PDT. In particular, the sequence of stack symbols of the first modes of the items in any such chain is a possible stack content. Ignoring the output, an item $(K_h\ K_{h'})$ represent the set of PDT configurations where the current state is p_h, the next input symbol to be read has the index $i_h{+}1$, and the stack content is formed of all the stack symbols to be found in the first mode of all items of any chain of items beginning with $(K_h\ K_{h'})$. Hence, if the collection of items of figure 1 is produced by a dynamic programming computation, it means that a standard non-deterministic computation of the PDT could have reached state p_1, having last read the input symbol of index i_1, and having built any of the following stack configurations (among others), with the stack top on the left hand side: $\mathrm{A}_1\mathrm{A}_2\mathrm{A}_9\ldots$, $\mathrm{A}_1\mathrm{A}_2\mathrm{A}_3\mathrm{A}_7\ldots$, $\mathrm{A}_1\mathrm{A}_2\mathrm{A}_3\mathrm{A}_5\mathrm{A}_6\ldots$, $\mathrm{A}_1\mathrm{A}_2\mathrm{A}_3\mathrm{A}_5\mathrm{A}_8\ldots$, $\mathrm{A}_1\mathrm{A}_2\mathrm{A}_4\mathrm{A}_3\mathrm{A}_5\mathrm{A}_8\ldots$, $\mathrm{A}_1\mathrm{A}_2\mathrm{A}_4\mathrm{A}_5\mathrm{A}_8\ldots$, and so on.

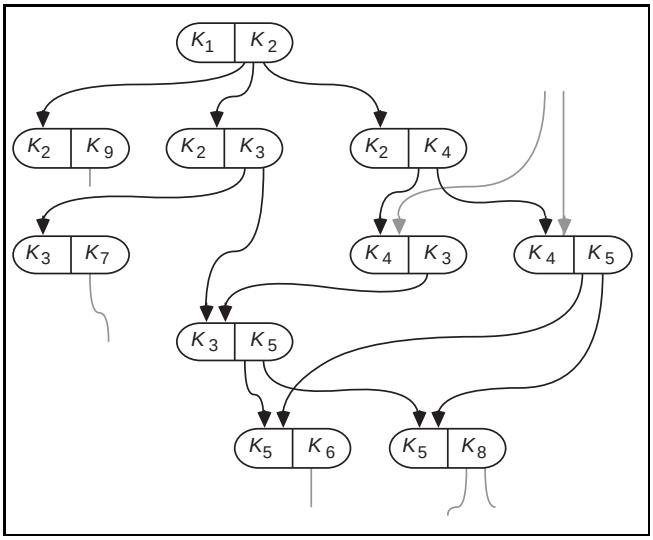

Figure 1: Items as shared representations of stack configurations

The transitions of the PDT are interpreted to produce new items, and new associated rules in $\mathcal{P}$ for the output grammar $\mathcal{G}$, as described in appendix C. When the same item is produced several times, only one copy is kept in the item set, but a new rule is produced each time. This merging of identical items accounts for the sharing of identical subcomputations. The corresponding rules with same left-hand-side (i.e. the multiply pro-

[5] We assume the reader to be familiar with some variation of Earley's algorithm. Earley's original paper uses the word *state* instead of *item*.

duced item) account for some of the sharing in the output (cf. appendices A & B). Sharing in the output also appears in the use of the same item in the right hand side of several different output rules. This directly results from the non-determinism of the PDT computation, i.e. the ambiguity of the input sentence.

The critical feature of the algorithm for handling cyclic rules (i.e. infinite ambiguity) is to be found in the handling of popping transitions[6]. When applying a popping transition $\tau = (\mathrm{p\ A}\ \epsilon_{\mathbf{\Sigma}} \mapsto \mathrm{r}\ \epsilon_{\mathbf{\Delta}}\ z)$ to the item $\mathrm{U} = ((\mathrm{p\ A}\ i)\ (\mathrm{q\ B}\ j))$ the algorithm must find all items $\mathrm{Y} = ((\mathrm{q\ B}\ j)\ (\mathrm{s\ D}\ k))$, i.e. all items with first mode $(\mathrm{q\ B}\ j)$, produced and build for each of them a new item $\mathrm{V} = ((\mathrm{r\ B}\ i)\ (\mathrm{s\ D}\ k))$ together with the output rule $(\mathrm{V} \rightarrow \mathrm{U}z)$ to be added to $\mathcal{P}$. The subtle point is that the Y-items must be all items with $(\mathrm{q\ B}\ j)$ as first mode, *including those that, when $j = i$, may be built later in the computation* (e.g. because their existence depends on some other V-item built in that step).

4 Parsing Incomplete Sentences

In order to handle incomplete sentences, we extend the input vocabulary with 2 symbols: "?" standing for one unknown word symbol, and "$*$" standing for an unknown sequence of input word symbols[7].

Normally a *scanning* transition, say $(\mathrm{p}\ \epsilon\ \mathrm{a} \mapsto \mathrm{r}\ \epsilon\ z)$, is applicable to an item, say $\mathrm{U} = ((\mathrm{p\ A}\ i)\ (\mathrm{q\ B}\ j))$ in $\mathcal{S}_i$, only when $\mathrm{a} = x_{i+1}$, where x_{i+1} is the next input symbol to be shifted. It produces a new item in $\mathcal{S}_{i+1}$ and a new rule in $\mathcal{P}$, respectively $\mathrm{V} = ((\mathrm{r\ A}\ i{+}1)\ (\mathrm{q\ B}\ j))$ and $(\mathrm{V} \rightarrow \mathrm{U}z)$ for the above transition and item.

When the next input symbol to be shifted is $x_{i+1} = ?$ (i.e. the unknown input word symbol), then any scanning transition may be applied as above independently of the input symbol required by the transition (provided that the transition is applicable with respect to PDT state and stack symbol).

When the next input symbol to be shifted is $x_{i+1} = *$ (i.e. the unknown input subsequence), then the algorithm proceeds as for the unknown word, except that the new item V is created in item set $\mathcal{S}_i$ instead of $\mathcal{S}_{i+1}$, i.e. $\mathrm{V} = ((\mathrm{r\ A}\ i)\ (\mathrm{q\ B}\ j))$ in the case of the above example. Thus, in the presence of the unknown symbol subsequence $*$, scanning transitions may be applied any number of times to the same computation thread, without shifting the input stream[8].

Scanning transitions are also used normally on input symbol x_{i+2} so as to produce also items in $\mathcal{S}_{i+2}$, for example the item $((\mathrm{r\ A}\ i{+}2)\ (\mathrm{q\ B}\ j))$, assuming $\mathrm{a} = x_{i+2}$ in the case of the above example[9]. This is how computation proceeds beyond the unknown subsequence.

There is a remaining difficulty due to the fact that it may be hard to relate a parse sequence of rules in $\mathbf{\Pi}$ to the input sentence because of the unknown number of input symbol actually assumed for an occurrence of the unknown input subsequence. We solve this difficulty by including the input word symbols in their proper place in parse sequences, which can thus be read as postfix polish encodings of the parse tree. In such a parse sequence, the symbol $*$ is included a number of times equal to the assumed length of the corresponding unknown input subsequence(s) for that parse (cf. appendix B).

A last point concerns simplification of the resulting grammar $\mathcal{G}$, or equivalently of the corresponding shared-parse-forest. In practice an unknown subsequence may stand for an arbitrarily complex sequence of input word symbols, with a correspondingly complex parse structure. Since the subsequence is unknown anyway, its hypothetical structures can be summarized by the non-terminal symbols that dominate it (thanks to context-freeness).

Hence the output parse grammar $\mathcal{G}$ produced by our algorithm may be simplified by replacing with the unknown subsequence terminal $*$, all nonterminals (i.e. items) that derive only on (occurrences of) this symbol. However, to keep the output readable, we usually qualify these $*$ symbols with the appropriate nonterminal of the parsed language grammar $\mathbf{G}$. The substructures thus eliminated can be retrieved by arbitrary use of the original CF grammar of the parsed language, which thus complements the simplified output grammar[10]. An example is given in appendix B.

5 Conclusion

We have shown that Earley's construction, when correctly accepting cyclic grammars, may be used to parse incomplete sentences. The generality of the construction presented allows its adaptation to any of the classical parsing schemata [16], and the use of well established parser construction techniques to achieve efficiency. The formal setting we have chosen is to our

[6] Popping transitions are also the critical place to look at for ensuring $\mathrm{O}(n^3)$ worst case complexity.

[7] Several adjacent "$*$" are equivalent to a single one.

[8] Note that in such a situation, a rule $X \rightarrow aX$ of the language grammar $\mathbf{G}$ behaves as if it were a cyclic rule $X \rightarrow X$, since the parsing proceeds as if it were ignoring terminal symbols. This does not lead to an infinite computation since only a finite number (proportional to i) of distinct items can be built in $\mathcal{S}_i$.

[9] We assume, only for simplicity of exposition, that $*$ is followed by a normal input word symbol. Note also that $\mathcal{S}_{i+1}$ is not built.

[10] If the input were reduced to the unknown subsequence alone, the output grammar $\mathcal{G}$ would be equivalent to the original grammar $\mathbf{G}$ of the input language (up to simple transformation). The output parse sequences would then simplify into a single occurrence of the symbol $*$ qualified by the initial nonterminal $\overset{\circ}{\mathrm{N}}$ of the language grammar $\mathbf{G}$.

knowledge the only one that has ever been used to prove the correctness of the constructed parse forest as well as that of the recognizer itself. We believe it to be a good framework to study the structure of parse forests [4], and to develop optimization strategies.

Recent extensions of our approach to recursive queries in Datalog [19] and to Horn clauses [20] are an indication that these techniques may be applied effectively to more complex grammatical setting, including unification based grammars and logic based semantics processing. More generally, dynamic programming approaches such as the one presented here should be a privileged way of dealing with ill-formed input, since the variety of possible errors is the source of even more combinatorial problems than the natural ambiguity or non-determinism already present in many "correct" sentences.

Acknowledgements: Sylvie Billot is currently studying the implementation technology for the algorithms described here [3, 4]. The examples in appendices A & B were produced with her prototype implementation. The author gratefully acknowledges her commitment to have this implementation running in time, as well as numerous discussions with her, Véronique Donzeau-Gouge, and Anne-Marie Vercoustre.

References

[1] Aho, A.V.; Sethi, R.; and Ullman, J.D. 1986 *Compilers — Principles, Techniques and Tools*. Addison-Wesley.

[2] Aho, A.V.; and Ullman, J.D. 1972 *The Theory of Parsing, Translation and Compiling*. Prentice-Hall, Englewood Cliffs, New Jersey.

[3] Billot, S. 1986 *Analyse Syntaxique Non-Déterministe*. Rapport DEA, Université d'Orléans la Source, and INRIA, France.

[4] Billot, S.; and Lang, B. 1988 *The structure of Shared Forests in Ambiguous Parsing*. In preparation.

[5] Bouckaert, M.; Pirotte, A.; and Snelling, M. 1975 Efficient Parsing Algorithms for General Context-Free Grammars. *Information Sciences* 8(1): 1-26.

[6] Cocke, J.; and Schwartz, J.T. 1970 *Programming Languages and Their Compilers*. Courant Institute of Mathematical Sciences, New York University, New York.

[7] Coppersmith, D.; and Winograd, S. 1982 On the Asymptotic Complexity of Matrix Multiplication. *SIAM Journal on Computing*, 11(3): 472-492.

[8] DeRemer, F.L. 1971 Simple LR(k) Grammars. *Communications ACM* 14(7): 453-460.

[9] Donzeau-Gouge, V.; Dubois, C.; Facon, P.; and Jean F. 1987 Development of a Programming Environment for SETL. *ESEC'87, Proc. of the* 1^{st} *European Software Engineering Conference*, Strasbourg (France), pp. 23-34.

[10] Earley, J. 1970 An Efficient Context-Free Parsing Algorithm. *Communications ACM* 13(2): 94-102.

[11] Graham, S.L.; Harrison, M.A.; and Ruzzo W.L. 1980 An Improved Context-Free Recognizer. *ACM Transactions on Programming Languages and Systems* 2(3): 415-462.

[12] Griffiths, I.; and Petrick, S. 1965 On the Relative Efficiencies of Context-Free Grammar Recognizers. *Communications ACM* 8(5): 289-300.

[13] Hays, D.G. 1962 Automatic Language-Data Processing. In *Computer Applications in the Behavioral Sciences*, (H. Borko ed.), Prentice-Hall, pp. 394-423.

[14] Ichbiah, J.D.; and Morse, S.P. 1970 A Technique for Generating Almost Optimal Floyd-Evans Productions for Precedence Grammars. *Communications ACM* 13(8): 501-508.

[15] Kasami, J. 1965 *An Efficient Recognition and Syntax Analysis Algorithm for Context-Free Languages*. Report of Univ. of Hawaii, also AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford (Massachusetts), also 1966, University of Illinois Coordinated Science Lab. Report, No. R-257.

[16] Kay, M. 1980 Algorithm Schemata and Data Structures in Syntactic Processing. *Proceedings of the Nobel Symposium on Text Processing*, Gothenburg.

[17] Knuth, D.E. 1965 On the Translation of Languages from Left to Right. *Information and Control*, 8: 607-639.

[18] Lang, B. 1974 Deterministic Techniques for Efficient Non-deterministic Parsers. *Proc. of the* 2^{nd} *Colloquium on Automata, Languages and Programming*, J. Loeckx (ed.), Saarbrücken, Springer Lecture Notes in Computer Science 14: 255-269.
Also: Rapport de Recherche 72, IRIA-Laboria, Rocquencourt (France).

[19] Lang, B. 1988 *Datalog Automata*. To appear in *Proc. of the* 3^{rd} *Internat. Conf. on Data and Knowledge Bases*, Jerusalem (Israel).

[20] Lang, B. 1988 *Complete Evaluation of Horn Clauses, an Automata Theoretic Approach*. In preparation.

[21] Li, T.; and Chun, H.W. 1987 A Massively Parallel Network-Based Natural Language Parsing System. *Proc. of* 2^{nd} *Int. Conf. on Computers and Applications* Beijing (Peking), : 401-408.

[22] Nakagawa, S. 1987 Spoken Sentence Recognition by Time-Synchronous Parsing Algorithm of Context-Free Grammar. *Proc. ICASSP 87*, Dallas (Texas), Vol. 2 : 829-832.

[23] Phillips, J.D. 1986 A Simple Efficient Parser for Phrase-Structure Grammars. *Quarterly Newsletter of the Soc. for the Study of Artificial Intelligence (AISBQ)* 59: 14-19.

[24] Pratt, V.R. 1975 LINGOL — A Progress Report. In *Proceedings of the 4th IJCAI*: 422-428.

[25] Sheil, B.A. 1976 Observations on Context Free Parsing. in *Statistical Methods in Linguistics*: 71-109, Stockholm (Sweden), Proc. of Internat. Conf. on Computational Linguistics (COLING-76), Ottawa (Canada).
Also: Technical Report TR 12-76, Center for Research in Computing Technology, Aiken Computation Laboratory, Harvard Univ., Cambridge (Massachusetts).

[26] Shieber, S.M. 1985 Using Restriction to Extend Parsing Algorithms for Complex-Feature-Based Formalisms. *Proceedings of the* 23^{rd} *Annual Meeting of the Association for Computational Linguistics*: 145-152.

[27] Tomita, M. 1986 An Efficient Word Lattice Parsing Algorithm for Continuous Speech Recognition. In *Proceedings of IEEE-IECE-ASJ International Conference on Acoustics, Speech, and Signal Processing (ICASSP 86)*, Vol. 3: 1569-1572.

[28] Tomita, M. 1987 An Efficient Augmented-Context-Free Parsing Algorithm. *Computational Linguistics* 13(1-2): 31-46.

[29] Valiant, L.G. 1975 General Context-Free Recognition in Less than Cubic Time. *Journal of Computer and System Sciences*, 10: 308-315.

[30] Younger, D.H. 1967 Recognition and Parsing of Context-Free Languages in Time n^3. *Information and Control*, 10(2): 189-208

A Simple example without unknown input subsequence

This first simple example, without unknown input, is intended to familiarize the reader with our constructions.

A.1 Grammar of the analyzed language

This grammar is taken from [28].

Nonterminals are in capital letters, and terminals are in lower case. The first rule is used for initialization and handling of the delimiter symbol `$`. The `$` delimiters are implicit in the actual input sentence.

```
(0)     $AX ::= $    S  $
(1)     S   ::= NP   VP
(2)     S   ::= S    PP
(3)     NP  ::= n
(4)     NP  ::= det  n
(5)     NP  ::= NP   PP
(6)     PP  ::= prep NP
(7)     VP  ::= v    NP
```

A.2 Input sentence

This input corresponds (for example) to the sentence:

"`I saw a man with a mirror`"

```
ANALYSIS OF: (n v det n prep det n)
```

A.3 Output grammar produced by the parser

The grammar output by the parser is given in figure 2. The initial nonterminal is the left-hand side of the first rule. For readability, the nonterminal/items have been given computer generated names of the form `nt`x, where x is an integer. At this point we have forgotten the internal structure of the items corresponding to their role in the parsing process. All other symbols are terminal. Integer terminals correspond to rule numbers of the input language grammar **G** (see section A.1 above), and the other terminals are symbols of the parsed language, i.e. symbols in $\mathbf{\Sigma}$. Note the ambiguity for nonterminal `nt3`.

nt0 ::= nt1 nt2	nt14 ::= det
nt1 ::= $	nt15 ::= n
nt2 ::= nt3 nt28	nt16 ::= nt17 6
nt3 ::= nt4 2	nt17 ::= nt18 nt19
nt3 ::= nt23 1	nt18 ::= prep
nt4 ::= nt5 nt16	nt19 ::= nt20 4
nt5 ::= nt6 1	nt20 ::= nt21 nt22
nt6 ::= nt7 nt9	nt21 ::= det
nt7 ::= nt8 3	nt22 ::= n
nt8 ::= n	nt23 ::= nt7 nt24
nt9 ::= nt10 7	nt24 ::= nt25 7
nt10 ::= nt11 nt12	nt25 ::= nt11 nt26
nt11 ::= v	nt26 ::= nt27 5
nt12 ::= nt13 4	nt27 ::= nt12 nt16
nt13 ::= nt14 nt15	nt28 ::= $

Figure 2: The output grammar.

A.4 Simplified output grammar

This is a simplified form of the grammar in which some of the structure that makes it readable as a shared-forest has been lost (though it could be retrieved). However it preserves all sharing of common subparses. This is the justification for having so many rules, while only 2 parse sequences may be generated by that grammar.

```
nt0 ::=  $ nt3 $
nt3 ::=  nt7 nt11 nt12 7 1 nt16 2
nt3 ::=  nt7 nt11 nt12 nt16 5 7 1
nt7 ::=  n 3
nt11 ::=  v
nt12 ::=  det n 4
nt16 ::=  prep det n 4 6
```

The 2 parses of the input, which are defined by this grammar, are:

```
$ n 3 v det n 4 7 1 prep det n 4 6 2 $
$ n 3 v det n 4 prep det n 4 6 5 7 1 $
```

Here again the 2 symbols $ must be read as delimiters.

A.5 Parse forest built from that grammar

To explain the construction of the shared forest, we first build in figure 3 a graph from the grammar of section A.3. Here the graph is acyclic, but with an incomplete input, it could have cycles. Each node corresponds to one terminal or nonterminal of the grammar in section A.3, and is labeled by it. The labels at the right of small dashes are input grammar rule numbers (cf. section A.1). Note the ambiguity of node `nt3` represented by an ellipse joining the two possible parses.

From the graph of figure 3, we can trivially derive the shared-forest given in figure 4.

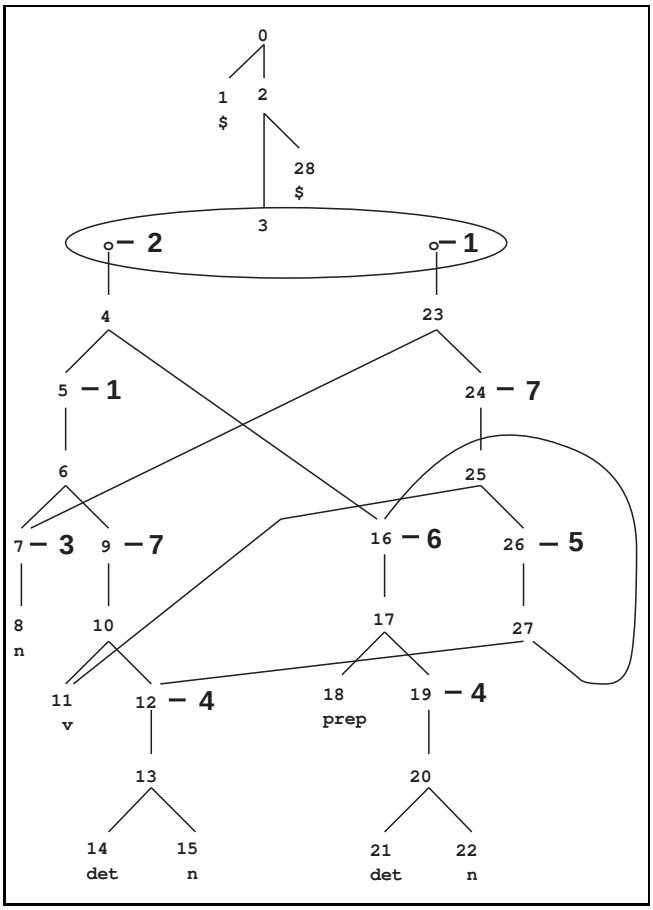

Figure 3: Graph of the output grammar.

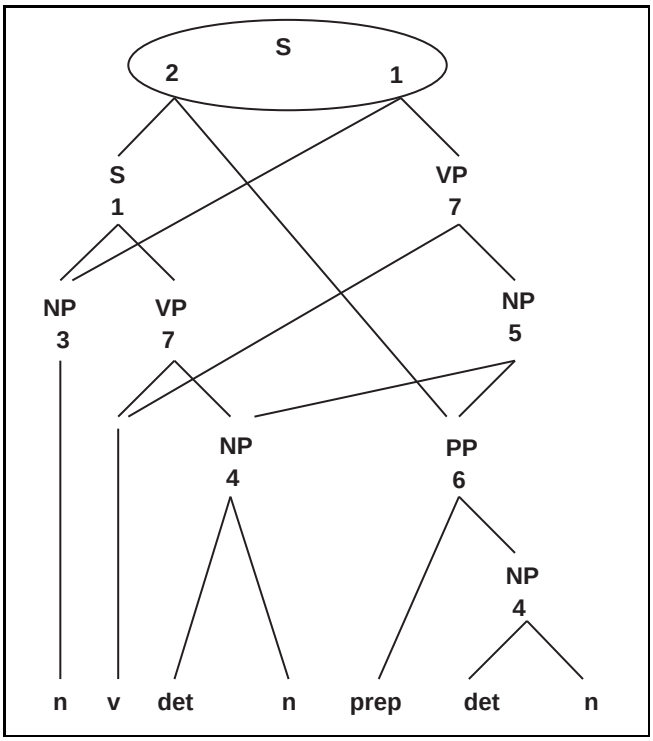

Figure 4: The shared parse forest

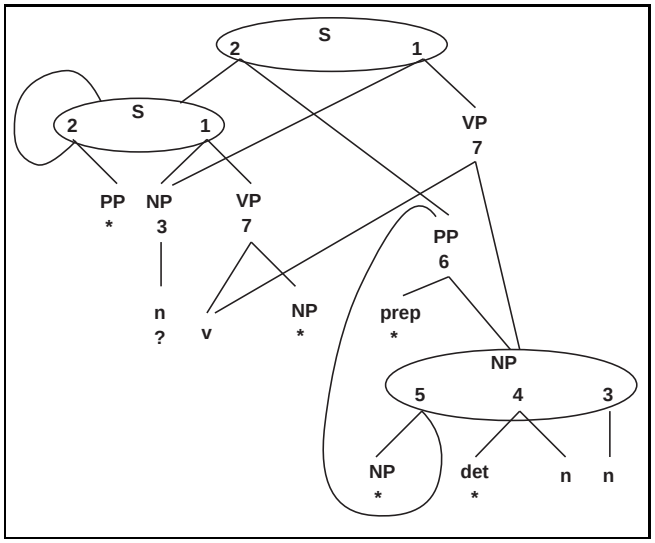

Figure 5: Shared-forest for an incomplete sentence.

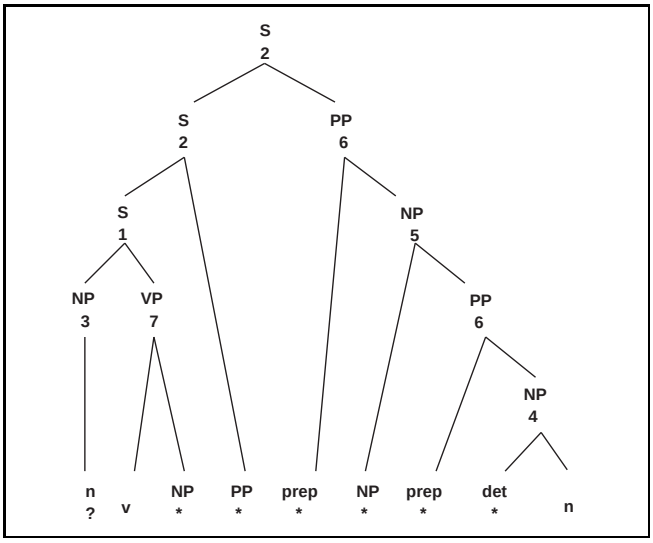

Figure 6: A parse tree extracted from the forest.

For readability, we present this shared-forest in a simplified form. Actually the sons of a node need sometimes to be represented as a binary Lisp-like list, so as to allow proper sharing of some of the sons. Each node includes a label which is a non-terminal of the grammar **G**, and for each possible derivation (several in case of ambiguity, e.g. the top node of figure 4) there is the number of the grammar rule used for that derivation.

The constructions in this section are purely virtual, and are not actually necessary in an implementation. The data-structure representing the grammar of section A.3 may be directly interpreted and used as a shared-forest.

B Example with an unknown input subsequence

B.1 Grammar of the analyzed language

The grammar is the same as in appendix A.

B.2 Input sentence

This input corresponds (for example) to the sentence:

"... saw ... mirror"

where the first "..." are known to be one word, and the last "..." may be any number of words, i.e.:

```
ANALYSIS OF: (?  v * n)
```

B.3 Output grammar produced by the parser

Note that the nodes that derive on (several) symbol(s) $*$ have been replaced by $*$ for simplification as indicated at the end of section 4. This explains the gaps in the numbering of nonterminals.

```
nt0 ::=  nt1 nt2
nt1 ::=  $
nt2 ::=  nt3 nt38
nt3 ::=  nt4 2
nt3 ::=  nt33 1
nt4 ::=  nt5 nt25
nt5 ::=  nt6 2
nt5 ::=  nt17 1
nt6 ::=  nt5 *
nt17 ::=  nt18 nt20
nt18 ::=  nt19 3
nt19 ::=  ?
nt20 ::=  nt21 7
nt21 ::=  nt22 *
nt22 ::=  v
nt25 ::=  nt26 6
nt26 ::=  * nt27
nt27 ::=  nt28
nt27 ::=  nt32 5
nt28 ::=  nt29 4
nt28 ::=  nt31 3
nt29 ::=  * nt30
nt30 ::=  n
nt31 ::=  n
nt32 ::=  * nt25
nt33 ::=  nt18 nt34
nt34 ::=  nt35 7
nt35 ::=  nt22 nt36
nt36 ::=  nt28
nt36 ::=  nt37 5
nt37 ::=  * nt25
nt38 ::=  $
```

B.4 Simplified output grammar

```
nt0 ::=  $ nt3 $
nt3 ::=  nt5 nt25 2
nt3 ::=  nt18 nt22 nt36 7 1
nt5 ::=  nt5 * 2
nt5 ::=  nt18 nt22 * 7 1
nt27 ::=  nt28
nt27 ::=  * nt25 5
nt28 ::=  * n 4
nt28 ::=  n 3
nt25 ::=  * nt27 6
nt18 ::=  ? 3
nt22 ::=  v
nt36 ::=  nt28
nt36 ::=  * nt25 5
```

A parse of the input, chosen in the infinite set of possible parses defined by this grammar, is the following (see figure 6):

```
$ ?  3 v * 7 1 * 2 * * * * n 4 6 5 6 2 $
```

This is not really a complete parse since, due to the first simplification of the grammar, some $*$ symbols stand for a missing nonterminal, i.e. for any parse of a string derived from this nonterminal. For example the first $*$ stand for the nonterminal `NP` and could be replaced by "`* 3`" or by "`* * 4 * * 3 6 5`".

B.5 Parse shared-forest built from that grammar

The output grammars given above are not optimal with respect to sharing. Mainly the nonterminals `nt27` and `nt36` should be the same (they do generate the same parse fragments). Also the terminal `n` should appear only once. We give in figure 5 a shared-forest corresponding to this grammar, build as in the previous example of appendix A, were we have improved the sharing by merging `nt27` and `nt36` so as to improve readability. We do not give the intermediate graph representing the output grammar as we did in appendix A.

Our implementation is currently being improved to directly achieve better sharing.

In figure 6 we give one parse-tree extracted from the shared-forest of figure 5. It corresponds to the parse sequence given as example in section B.4 above. Note that, like the corresponding parse sequence, this is not a complete parse tree, since it has nonterminals labeling its leaves. A complete parse tree may be obtained by completing arbitrarily these leaves according to the original grammar of the language as defined in section A.1.

C The algorithm

The length of this algorithm is due to its generality. Fewer types of transitions are usually needed with specific implementations, typically only one for scanning transitions.

Comments are prefixed with "—".

— *Begin parse with input sequence x of length n*

step-A: — *Initialization*

$\overset{\circ}{\mathrm{U}} := ((\overset{\circ}{\mathrm{q}}\ \overset{\circ}{\$}\ 0)\ (\overset{\circ}{\mathrm{q}}\ \overset{\circ}{\$}\ 0))$; — *initial item*
$\overset{\circ}{\pi} := (\overset{\circ}{\mathrm{U}} \to \epsilon)$; — *first rule of output grammar*
$\mathcal{S}_0 := \{\overset{\circ}{\mathrm{U}}\}$; — *initialize item-set $\mathcal{S}_0$*
$\mathcal{P} := \{\overset{\circ}{\pi}\}$; — *rules of output grammar*
$i := 0$; — *input-scanner index is set*
— *before the first input symbol*

step-B: — *Iteration*

`loop` — *while $i \leq n$ (cf. exit in step-B.2)*

`if` $x_{i+1} \neq *$

step-B.1:
— *Normal completion of item-set $\mathcal{S}_i$*
— *with non-scanning transitions.*
`for every item` $\mathrm{U} = ((\mathrm{p\ A}\ i)\ (\mathrm{q\ B}\ j))$ `in` $\mathcal{S}_i$ `do`
`for every non-scanning transition` τ `in` δ
`do`
— *we distinguish five cases, according to τ:*

case-B.1.1: — *stack-free transition*
`if` $\tau = (\mathrm{p}\ \epsilon\ \epsilon \mapsto \mathrm{r}\ \epsilon\ z)$
`then`
$\mathrm{V} := ((\mathrm{r\ A}\ i)\ (\mathrm{q\ B}\ j))$;
$\mathcal{S}_i := \mathcal{S}_i \cup \{\mathrm{V}\}$;
$\mathcal{P} := \mathcal{P} \cup \{(\mathrm{V} \to \mathrm{U}z)\}$;
case-B.1.2: — *push transition*
`if` $\tau = (\mathrm{p}\ \epsilon\ \epsilon \mapsto \mathrm{r\ C}\ z)$
`then`
$\mathrm{V} := ((\mathrm{r\ C}\ i)\ (\mathrm{p\ A}\ i))$;
$\mathcal{S}_i := \mathcal{S}_i \cup \{\mathrm{V}\}$;
$\mathcal{P} := \mathcal{P} \cup \{(\mathrm{V} \to z)\}$;
case-B.1.3: — *pop transition*
`if` $\tau = (\mathrm{p\ A}\ \epsilon \mapsto \mathrm{r}\ \epsilon\ z)$
`then`
`for every item` $\mathrm{Y} = ((\mathrm{q\ B}\ j)\ (\mathrm{s\ D}\ k))$
`in` $\mathcal{S}_j$ `do`
$\mathrm{V} := ((\mathrm{r\ B}\ i)\ (\mathrm{s\ D}\ k))$;
$\mathcal{S}_i := \mathcal{S}_i \cup \{\mathrm{V}\}$;
$\mathcal{P} := \mathcal{P} \cup \{(\mathrm{V} \to \mathrm{Y}\mathrm{U}z)\}$;
case-B.1.4: — *pop-push transition*
`if` $\tau = (\mathrm{p\ A}\ \epsilon \mapsto \mathrm{r\ C}\ z)$
`then`
$\mathrm{V} := ((\mathrm{r\ C}\ i)\ (\mathrm{q\ B}\ j))$;
$\mathcal{S}_i := \mathcal{S}_i \cup \{\mathrm{V}\}$;
$\mathcal{P} := \mathcal{P} \cup \{(\mathrm{V} \to \mathrm{U}z)\}$;
case-B.1.5:
— *Other non-scanning transitions are ignored*

`else`
— $\boxed{x_{i+1} = *}$
— *i.e. the next input symbol*
— *is the unknown subsequence:*

step-B∗.1:
— *Completion of item-set $\mathcal{S}_i$*
— *with non-scanning transitions*
— *and with dummy scanning transitions.*
— *This step is similar to step-B.1,*
— *but considering all transitions as non-scanning.*
`for every item` $\mathrm{U} = ((\mathrm{p\ A}\ i)\ (\mathrm{q\ B}\ j))$ `in` $\mathcal{S}_i$ `do`
`for every transition` τ `in` δ `do`
— *we distinguish five cases, according to τ:*

case-B∗.1.1:
`if` $\tau = (\mathrm{p}\ \epsilon\ \epsilon \mapsto \mathrm{r}\ \epsilon\ z)$ `or`
$\tau = (\mathrm{p}\ \epsilon\ \mathrm{a} \mapsto \mathrm{r}\ \epsilon\ z)$
`then`
$\mathrm{V} := ((\mathrm{r\ A}\ i)\ (\mathrm{q\ B}\ j))$;
$\mathcal{S}_i := \mathcal{S}_i \cup \{\mathrm{V}\}$;
$\mathcal{P} := \mathcal{P} \cup \{(\mathrm{V} \to \mathrm{U}z)\}$;
— *and so on as in step-B.1*

...

step-B.2: — *Exit for main loop*
`if` $i = n$ `then exit loop;` — *go to step-C*
$h := i + 1$;
`while` $x_h = *$ `do` $h := h + 1$;

step-B.3: — *Initialization of item-set* $\mathcal{S}_h$
$\mathcal{S}_h := \emptyset$;
for every item $\mathrm{U} = ((\mathrm{p}\ \mathrm{A}\ i)\ (\mathrm{q}\ \mathrm{B}\ j))$ **in** $\mathcal{S}_i$ **do**
for every scanning transition τ **in** δ **do**
...
— *Proceed by cases as in step-B.1,*
— *but with scanning transitions, and*
— *adding the new items to* $\mathcal{S}_h$ *instead of* $\mathcal{S}_i$.
— *See for example the following case:*
case-B.3.2:
if $\tau = (\mathrm{p} \in \mathrm{a} \mapsto \mathrm{r}\ \mathrm{C}\ z)$ **with** $x_h = \mathrm{a}$ **or** $x_h = ?$
then
$\mathrm{V} := ((\mathrm{r}\ \mathrm{C}\ h)\ (\mathrm{p}\ \mathrm{A}\ i))$;
$\mathcal{S}_h := \mathcal{S}_h \cup \{\mathrm{V}\}$;
$\mathcal{P} := \mathcal{P} \cup \{(\mathrm{V} \to z)\}$;
...
step-B.4: — *Incrementation of scanning index* i
$i := h$;
end loop;
step-C: — *Termination*
for every item $\mathtt{U} = ((\mathrm{f} \overset{\circ}{\$} n)\ (\overset{\circ}{\mathrm{q}} \overset{\circ}{\$} 0))$ **in** $\mathcal{S}_n$
such that $\mathrm{f} \in \mathrm{F}$ **do**
$\mathcal{P} := \mathcal{P} \cup (\mathrm{U_f} \to \mathrm{U})$;
— $\mathrm{U_f}$ *is the initial nonterminal of* $\mathcal{G}$.
— *End of parse*