The Structure of Shared Forests in Ambiguous Parsing

Sylvie Billot[†*] Bernard Lang[*]
INRIA
[†]and Université d'Orléans
billot@inria.inria.fr lang@inria.inria.fr

Abstract

The Context-Free backbone of some natural language analyzers produces all possible CF parses as some kind of shared forest, from which a single tree is to be chosen by a disambiguation process that may be based on the finer features of the language. We study the structure of these forests with respect to optimality of sharing, and in relation with the parsing schema used to produce them. In addition to a theoretical and experimental framework for studying these issues, the main results presented are:

– sophistication in chart parsing schemata (e.g. use of look-ahead) may reduce time and space efficiency instead of improving it,

– there is a shared forest structure with at most cubic size for any CF grammar,

– when $\mathcal{O}(n^3)$ complexity is required, the shape of a shared forest is dependent on the parsing schema used.

Though analyzed on CF grammars for simplicity, these results extend to more complex formalisms such as unification based grammars.

Key words: Context-Free Parsing, Ambiguity, Dynamic Programming, Earley Parsing, Chart Parsing, Parsing Strategies, Parsing Schemata, Parse Tree, Parse Forest.

1 Introduction

Several natural language parser start with a pure *Context-Free (CF)* backbone that makes a first sketch of the structure of the analyzed sentence, before it is handed to a more elaborate analyzer (possibly a coroutine), that takes into account the finer grammatical structure to filter out undesirable parses (see for example [24, 28]). In [28], Shieber surveys existing variants to this approach before giving his own tunable approach based on restrictions that " split up the infinite nonterminal domain into a finite set of equivalence classes that can be used for parsing". The basic motivation for this approach is to benefit from the CF parsing technology whose development over 30 years has lead to powerful and efficient parsers [1, 7].

A parser that takes into account only an approximation of the grammatical features will often find ambiguities it cannot resolve in the analyzed sentences[1]. A natural solution is then to produce all possible parses, according to the CF backbone, and then select among them on the basis of the complete features information. One hitch is that the number of parses may be exponential in the size of the input sentence, or even infinite for cyclic grammars or incomplete sentences [16]. However chart parsing techniques have been developed that produce an encoding of all possible parses as a data structure with a size polynomial in the length of the input sentence. These techniques are all based on a dynamic programming paradigm.

The kind of structure they produce to represent all parses of the analyzed sentence is an essential characteristic of these algorithms. Some of the published algorithms produce only a chart as described by Kay in [14], which only associates nonterminal categories to segments of the analyzed sentence [11, 39, 13, 3, 9], and which thus still requires non-trivial processing to extract parse-trees [26]. The worst size complexity of such a chart is only a square function of the size of the input[2].

However, practical parsing algorithms will often produce a more complex structure that explicitly relates the instances of nonterminals associated with sentence fragments to their constituents, possibly in several ways in case of ambiguity, with a sharing of some common subtrees between the distinct ambiguous parses [7, 4, 24, 31, 25][3]

One advantage of this structure is that the chart retains only these constituents that can actually participate in a parse. Furthermore it makes the extraction of parse-trees a trivial matter. A drawback is that this structure may be cubic in the length of the parsed sentence, and more generally polynomial[4] for some proposed algorithms [31]. However, these algorithms are rather well behaved in practice, and this complexity is not a problem.

*Address: INRIA, B.P. 105, 78153 Le Chesnay, France.
The work reported here was partially supported by the Eureka Software Factory project.

Proceedings of the 27th Annual Meeting of the Association for Computational Linguistics (ACL), pages 143-151, Vancouver (British Columbia), 26-29 June 1989. - Also available (slightly modified) as Rapport INRIA N° RR-1038 (mai 1989).

[1]Ambiguity may also have a semantical origin.

[2]We do not consider CF recognizers that have asymptotically the lowest complexity, but are only of theoretical interest here [35, 5].

[3]There are several other published implementation of chart parsers [23, 20, 33], but they often do not give much detail on the output of the parsing process, or even side-step the problem altogether [33]. We do not consider here the *well formed substring tables* of Sheil [26] which falls somewhere in between in our classification. They do not use pointers and parse-trees are only "indirectly" visible, but may be extracted rather simply in linear time. The table may contain useless constituents.

[4]Space cubic algorithms often require the language grammar to be in Chomsky Normal Form, and some authors have incorrectly conjectured that cubic complexity cannot be obtained otherwise.

In this paper we shall call *shared forests* such data structures used to represent simultaneously all parse trees for a given sentence.

Several questions may be asked in relation with shared forests:

- How to construct them during the parsing process?
- Can the cubic complexity be attained without modifying the grammar (e.g. into Chomsky Normal Form)?
- What is the appropriate data structure to improve sharing and reduce time and space complexity?
- How good is the sharing of tree fragments between ambiguous parses, and how can it be improved?
- Is there a relation between the coding of parse-trees in the shared forest and the parsing schema used?
- How well formalized is their definition and construction?

These questions are of importance in practical systems because the answers impact both the performance and the implementation techniques. For example good sharing may allow a better factorization of the computation that filters parse trees with the secondary features of the language. The representation needed for good sharing or low space complexity may be incompatible with the needs of other components of the system. These components may also make assumptions about this representation that are incompatible with some parsing schemata. The issue of formalization is of course related to the formal tractability of correctness proof for algorithms using shared forests.

In section 2 we describe a uniform theoretical framework in which various parsing strategies are expressed and compared with respect to the above questions. This approach has been implemented into a system intended for the experimental study and comparison of parsing strategies. This system is described in section 3. Section 4 contains a detailed example produced with our implementation which illustrates both the working of the system and the underlying theory.

2 A Uniform Framework

To discuss the above issues in a uniform way, we need a general framework that encompasses all forms of chart parsing and shared forest building in a unique formalism. We shall take as a basis a formalism developed by the second author in previous papers [15, 16]. The idea of this approach is to separate the dynamic programming constructs needed for efficient chart parsing from the chosen parsing schema. Comparison between the classifications of Kay [14] and Griffith & Petrick [10] shows that a parsing schema (or parsing strategy) may be expressed in the construction of a *Push-Down Transducer (PDT)*, a well studied formalization of left-to-right CF parsers[5]. These PDTs are usually *non-deterministic* and cannot be used as produced for actual parsing. Their backtrack simulation does not always terminate, and is often time-exponential when it does, while breadth-first simulation is usually exponential for both time and space. However, by extending Earley's dynamic programming construction to PDTs, Lang provided in[15] a way of simulating all possible computations of any PDT in cubic time and space complexity. This approach may thus be used as a uniform framework for comparing chart parsers[6].

[5]Griffith & Petrick actually use Turing machines for pedagogical reasons.

[6]The original intent of [15] was to show how one can generate efficient general CF chart parsers, by first producing the PDT with the efficient techniques for deterministic parsing developed for the compiler technology [6, 12, 1]. This idea was later successfully used by Tomita [31] who applied it to LR(1) parsers [6, 1], and later to other pushdown based parsers [32].

2.1 The algorithm

The following is a formal overview of parsing by dynamic programming interpretation of PDTs.

Our aim is to parse sentences in the language $\mathcal{L}(\mathbf{G})$ generated by a CF phrase structure grammar $\mathbf{G} = (\mathbf{V}, \mathbf{\Sigma}, \mathbf{\Pi}, \overset{\circ}{\mathrm{N}})$ 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, $\overset{\circ}{\mathrm{N}}$ for the initial nonterminal, and ϵ for the empty string.

We assume that, by some appropriate parser construction technique (e.g. [12, 6, 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 $\mathbf{\Pi}$[7] 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 $\mathcal{T}_{\mathbf{G}} = (\mathbf{Q}, \mathbf{\Sigma}, \mathbf{\Delta}, \mathbf{\Pi}, \delta, \overset{\circ}{\mathrm{q}}, \overset{\circ}{\$}, \mathbf{F})$ 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[8] (*i.e. rules of* $\mathbf{G}$), $\overset{\circ}{\mathrm{q}}$ is the initial state, $\overset{\circ}{\$}$ is the initial stack symbol, $\mathbf{F}$ is the set of final states, δ is a finite set of transitions of the form: $(\mathrm{p}\ \mathrm{A}\ \mathrm{a} \ \mapsto\ \mathrm{q}\ \mathrm{B}\ u)$ with $\mathrm{p}, \mathrm{q} \in \mathbf{Q}$, $\mathrm{A}, \mathrm{B} \in \mathbf{\Delta} \cup \{\epsilon\}$, $\mathrm{a} \in \mathbf{\Sigma} \cup \{\epsilon\}$, and $u \in \mathbf{\Pi}^{\star}$.

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 ϵ in the transition, no input symbol is scanned. If A (resp. B) is replaced by ϵ then no stack symbol is popped from (resp. pushed on) the stack.

Our algorithm consists in an Earley-like[9] simulation of the PDT $\mathcal{T}_{\mathbf{G}}$. Using the terminology of [1], 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 an *output 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.

Appendix A gives the details of the construction of items and rules in $\mathcal{G}$ by interpretation of the transitions of the PDT.

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

[8]Actual implementations use output symbols from $\mathbf{\Pi} \cup \mathbf{\Sigma}$, since rules alone do not distinguish words in the same lexical category.

[9]We assume the reader to be familiar with some variation of Earley's algorithm. Earley's original paper uses the word *state* (from dynamic programming terminology) instead of *item*.

More details may be found in [15, 16].

2.2 The shared forest

An apparently major difference between the above algorithm and other parsers is that it represents a parse as the string of the grammar rules used in a leftmost reduction of the parsed sentence, rather than as a parse tree (cf. section 4). When the sentence has several distinct parses, the set of all possible parse strings is represented in finite shared form by a CF grammar that generates that possibly infinite set. Other published algorithms produce instead a graph structure representing all parse-trees with sharing of common subparts, which corresponds well to the intuitive notion of a shared forest.

This difference is only appearance. We show here in section 4 that *the CF grammar of all leftmost parses is just a theoretical formalization of the shared-forest graph.* Context-Free grammars can be represented by AND-OR graphs that are closely related to the syntax diagrams often used to describe the syntax of programming languages [37], and to the transition networks of Woods [22]. In the case of our grammar of leftmost parses, this AND-OR graph (which is acyclic when there is only finite ambiguity) is precisely the shared-forest graph. In this graph, AND-nodes correspond to the usual parse-tree nodes, while OR-nodes correspond to ambiguities, i.e. distinct possible subtrees occurring in the same context. Sharing of subtrees in represented by nodes accessed by more than one other node.

The grammar viewpoint is the following (cf. the example in section 4). Non-terminal (resp. terminal) symbols correspond to nodes with (resp. without) outgoing arcs. AND-nodes correspond to right-hand sides of grammar rules, and OR-nodes (i.e. ambiguities) correspond to non-terminals defined by several rules. Subtree sharing is represented by several uses of the same symbol in rule right-hand sides.

To our knowledge, this representation of parse-forests as grammars is the simplest and most tractable theoretical formalization proposed so far, and the parser presented here is the only one for which the correctness of the output grammar — i.e. of the shared-forest — has ever been proved. Though in the examples we use graph(ical) representations for intuitive understanding (grammars are also sometimes represented as graphs [37]), they are not the proper formal tool for manipulating shared forests, and developing formalized (proved) algorithms that use them. Graph formalization is considerably more complex and awkward to manipulate than the well understood, specialized and few concepts of CF grammars. Furthermore, unlike graphs, this grammar formalization of the shared forest may be tractably extended to other grammatical formalisms (cf. section 5).

More importantly, our work on the parsing of incomplete sentences [16] has exhibited the fundamental character of our grammatical view of shared forests: when parsing the completely unknown sentence, the shared forest obtained is precisely the complete grammar of the analyzed language. This also leads to connections with the work on partial evaluation [8].

2.3 The shape of the forest

For our shared-forest, a cubic space complexity (in the worst case — space complexity is often linear in practice) is achieved, without requiring that the language grammar be in Chomsky Normal Form, by producing a grammar of parses that has at most two symbols on the right-hand side of its rules. This amounts to representing the list of sons of a parse tree node as a Lisp-like list built with binary nodes (see figures 1 & 2), and it allows partial sharing of the sons [10].

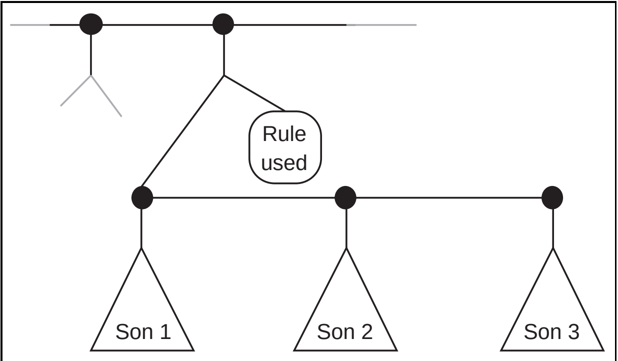

Figure 1: Bottom-up parse-tree

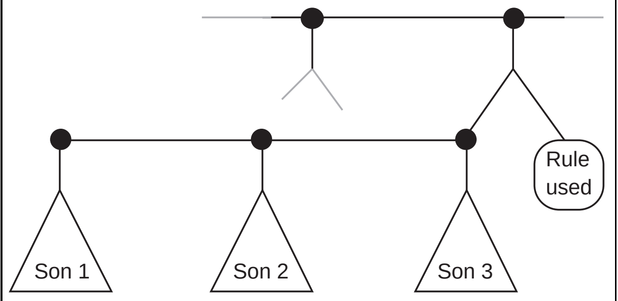

Figure 2: Top-down parse-tree

The structure of the parse grammar, i.e. *the shape of the parse forest, is tightly related to the parsing schema used*, hence to the structure of the possible computation of the non-deterministic PDT from which the parser is constructed. First we need a precise characterization of parsing strategies, whose distinction is often blurred by superimposed optimizations. We call bottom-up a strategy in which the PDT decides on the nature of a constituent (i.e. on the grammar rule that structures it), after having made this decision first on its subconstituents. It corresponds to a postfix left-to-right walk of the parse tree. Top-Down parsing recognizes a constituent before recognition of its subconstituents, and corresponds to a prefix walk. Intermediate strategies are also possible.

The sequence of operations of a bottom-up parser is basically of the following form (up to possible simplifying optimizations): To parse a constituent A, the parser first parses and pushes on the stack each sub-constituent B_i; at some point, it decides that it has all the constituents of A on the stack and it pops them all, and then it pushes A and outputs the (rule number π of the) recognized rule $\pi: \ A \rightarrow B_1 \ldots B_n$,. Dynamic programming interpretation of such a sequence results in a shared forest containing parse-trees with the shape described in figure 1, i.e. where each node of the forest points to the beginning of the list of its sons.

A top-down PDT uses a different sequence of operations, detailed in appendix B, resulting in the shape of figure 2 where a forest node points to the end of the list of sons, which

[10]This was noted by Sheil [26] and is implicit in his use of "2-form" grammars.

is itself chained backward. These two figures are only simple examples. Many variations on the shape of parse trees and forests may be obtained by changing the parsing schema.

Sharing in the shared forest may correspond to sharing of a complete subtree, but also to sharing of a tail of a list of sons: *this is what allows the cubic complexity*. Thus bottom-up parsing may share only the rightmost subconstituents of a constituent, while top-down parsing may share only the leftmost subconstituents. *This relation between parsing schema and shape of the shared forest (and type of sharing) is a consequence of intrinsic properties of chart parsing*, and not of our specific implementation.

It is for example to be expected that the bidirectional nature of island parsing leads to irregular structure in shared forests, when optimal sharing is sought for.

3 Implementation and Experimental Results

The ideas presented above have been implemented in an experimental system called *Tin* (after the woodman of OZ). The intent is to provide a uniform framework for the construction and experimentation of chart parsers, somewhat as systems like MCHART [29], but with a more systematic theoretical foundation. The kernel of the system is a virtual parsing machine with a stack and a set of primitive commands corresponding essentially to the operation of a practical Push-Down Transducer. These commands include for example: `push` (resp. `pop`) to push a symbol on the stack (resp. pop one), `checkwindow` to compare the look-ahead symbol(s) to some given symbol, `checkstack` to branch depending on the top of the stack, `scan` to read an input word, `output` to output a rule number (or a terminal symbol), `goto` for unconditional jumps, and a few others. However *these commands are never used directly to program parsers*. They are used as machine instructions for compilers that compile grammatical definitions into Tin code according to some parsing schema.

A characteristic of these commands is that they may all be marked as non-deterministic. The intuitive interpretation is that there is a non-deterministic choice between a command thus marked and another command whose address in the virtual machine code is then specified. However execution of the virtual machine code is done by an *all-paths* interpreter that follows the dynamic programming strategy described in section 2.1 and appendix A.

The Tin interpreter is used in two different ways:

1. to study the effectiveness for chart parsing of known parsing schemata designed for deterministic parsing. We have only considered formally defined parsing schemata, corresponding to established PDA construction techniques that we use to mechanically translate CF grammars into Tin code. (e.g. LALR(1) and LALR(2) [6], weak precedence [12], LL(0) top-down (recursive descent), LR(0), LR(1) [1] ...).

2. to study the computational behavior of the generated code, and the optimization techniques that could be used on the Tin code — and more generally chart parser code — with respect to code size, execution speed and better sharing in the parse forest.

Experimenting with several compilation schemata has shown that *sophistication may have a negative effect on the efficiency of all-path parsing*[11]. Sophisticated PDT construction techniques tend to multiply the number of special cases, thereby increasing the code size of the chart parser. Sometimes it also prevents sharing of locally identical subcomputations because of differences in context analysis. This in turn may result in lesser sharing in the parse forest and sometimes longer computation, as in example SBBL in appendix C, but of course it does not change the set of parse-trees encoded in the forest[12]. Experimentally, weak precedence gives slightly better sharing than LALR(1) parsing. The latter is often viewed as more efficient, whereas it only has a larger deterministic domain.

One essential guideline to achieve better sharing (and often also reduced computation time) is to try to recognize every grammar rule in only one place of the generated chart parser code, even at the cost of increasing non-determinism.

Thus simpler schemata such as precedence, LL(0) (and probably LR(0)[13]) produce the best sharing. However, since they correspond to a smaller deterministic domain within the CF grammar realm, they may sometimes be computationally less efficient because they produce a larger number of useless items (i.e. edges) that correspond to dead-end computational paths.

Slight sophistication (e.g. LALR(1) used by Tomita in [31], or LR(1)) may slightly improve computational performance by detecting earlier dead-end computations. This may however be at the expense of the forest sharing quality.

More sophistication (say LR(2)) is usually losing on both accounts as explained earlier. The duplication of computational paths due to distinct context analysis overweights the benefits of early elimination of dead-end paths. But there can be no absolute rule: if a grammar is "close" to the LR(2) domain, an LR(2) schema is likely to give the best result for most parsed sentences.

Sophisticated schemata correspond also to larger parsers, which may be critical in some natural language applications with very large grammars.

The choice of a parsing schema depends in fine on the grammar used, on the corpus (or kind) of sentences to be an-

[11] We mean here the sophistication of the CF parser construction technique rather than the sophistication of the language features chosen to be used by this parser.

[12] This negative behavior of some techniques originally intended to preserve determinism had been remarked and analyzed in a special case by Bouckaert, Pirotte and Snelling [3]. However we believe their result to be weaker than ours, since it seems to rely on the fact that they directly interpret grammars rather than first compile them. Hence each interpretive step include in some sense compilation steps, which are more expensive when look-ahead is increased. Their paper presents several examples that run less efficiently when look-ahead is increased. For all these examples, this behavior disappears in our compiled setting. However the grammar SBBL in appendix C shows a loss of efficiency with increased look-ahead that is due exclusively to loss of sharing caused by irrelevant contextual distinctions. This effect is particularly visible when parsing incomplete sentences [16].

Efficiency loss with increased look-ahead is mainly due to state splitting [6]. This should favor LALR techniques over LR ones.

[13] Our results do not take into account a newly found optimization of PDT interpretation that applies to all and only to bottom-up PDTs. This should make simple bottom-up schemes competitive for sharing quality, and even increase their computational efficiency. However it should not change qualitatively the relative performances of bottom-up parsers, and may emphasize even more the phenomenon that reduces efficiency when look-ahead increases.

alyzed, and on a balance between computational and sharing efficiency. It is best decided on an experimental basis with a system such as ours. Furthermore, we do not believe that any firm conclusion limited to CF grammars would be of real practical usefulness. The real purpose of the work presented is to get a qualitative insight in phenomena which are best exhibited in the simpler framework of CF parsing. This insight should help us with more complex formalisms (cf. section 5) for which the phenomena might be less easily evidenced.

Note that the evidence gained contradicts the common belief that parsing schemata with a large deterministic domain (see for example the remarks on LR parsing in [31]) are more effective than simpler ones. Most experiments in this area were based on incomparable implementations, while our uniform framework gives us a common theoretical yardstick.

4 A Simple Bottom-Up Example

The following is a simple example based on a bottom-up PDT generated by our LALR(1) compiler from the following grammar taken from [31]:

```
(0) '$ax ::= $ 's $          (4) 'np ::= det n
(1) 's ::= 'np 'vp           (5) 'np ::= 'np 'pp
(2) 's ::= 's 'pp            (6) 'pp ::= prep 'np
(3) 'np ::= n                (7) 'vp ::= v 'np
```

Nonterminals are prefixed with a quote symbol. The first rule is used for initialization and handling of the delimiter symbol `$`. The `$` delimiters are implicit in the actual input sentence.

The sample input is "`(n v det n prep n)`". It figures (for example) the sentence: "`I see a man at home`".

4.1 Output grammar produced by the parser

The grammar of parses of the input sentence is given in figure 3.

The initial nonterminal is the left-hand side of the first rule. For readability, the nonterminals have been given computer generated names of the form nt$x$, where x is an integer. All other symbols are terminal. Integer terminals correspond to rule numbers of the input language grammar given above, and the other terminals are symbols of the parsed language, except for the special terminal "`nil`" which indicates the end of the list of subconstituents of a sentence constituent, and may also be read as the empty string ϵ. Note the ambiguity for nonterminal `nt4`.

It is possible to simplify this grammar to 7 rules without losing the sharing of common subparses. However it would no longer exhibit the structure that makes it readable as a shared-forest (though this structure could be retrieved).

The two parses of the input sentence defined by this grammar are:

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

Here again the two `$` symbols must be read as delimiters. The "`nil`" symbols, no longer useful, have been omitted in these two parses.

```
nt0 ::= nt1 0               nt19 ::= nt20 nil
nt1 ::= nt2 nt3             nt20 ::= n
nt2 ::= $                   nt21 ::= nt22 nil
nt3 ::= nt4 nt37            nt22 ::= nt23 6
nt4 ::= nt5 2               nt23 ::= nt24 nt25
nt4 ::= nt29 1              nt24 ::= prep
nt5 ::= nt6 nt21            nt25 ::= nt26 nil
nt6 ::= nt7 1               nt26 ::= nt27 3
nt7 ::= nt8 nt11            nt27 ::= nt28 nil
nt8 ::= nt9 3               nt28 ::= n
nt9 ::= nt10 nil            nt29 ::= nt8 nt30
nt10 ::= n                  nt30 ::= nt31 nil
nt11 ::= nt12 nil           nt31 ::= nt32 7
nt12 ::= nt13 7             nt32 ::= nt14 nt33
nt13 ::= nt14 nt15          nt33 ::= nt34 nil
nt14 ::= v                  nt34 ::= nt35 5
nt15 ::= nt16 nil           nt35 ::= nt16 nt36
nt16 ::= nt17 4             nt36 ::= nt22 nil
nt17 ::= nt18 nt19          nt37 ::= nt38 nil
nt18 ::= det                nt38 ::= $
```

Figure 3: Grammar of parses of the input sentence

4.2 Parse shared-forest constructed from that grammar

To explain the structure of the shared forest, we first build a graph from the grammar, as shown in figure 4. Each node corresponds to one terminal or nonterminal of the grammar in figure 3, and is labelled by it. The labels at the right of small dashes are rule numbers from the parsed language grammar (see beginning of section 4). The basic structure is that of figure 1.

From this first graph, we can trivially derive the more traditional shared forest given in figure 5. Note that this simplified representation is not always adequate since it does not allow partial sharing of their sons between two nodes. Each node includes a label which is a non-terminal of the parsed language grammar, and for each possible derivation (several in case of ambiguity) there is the number of the grammar rule used for that derivation. Though this simplified version is more readable, the representation of figure 5 is not adequate to represent partial sharing of the subconstituents of a constituent.

Of course, the "constructions" given in this section are purely virtual. In an implementation, the data-structure representing the grammar of figure 3 may be directly interpreted and used as a shared-forest.

A similar construction for top-down parsing is sketched in appendix B.

5 Extensions

As indicated earlier, our intent is mostly to understand phenomena that would be harder to evidence in more complex grammatical formalisms.

This statement implies that our approach can be extended. This is indeed the case. It is known that many simple parsing schemata can be expressed with stack based machines [32].

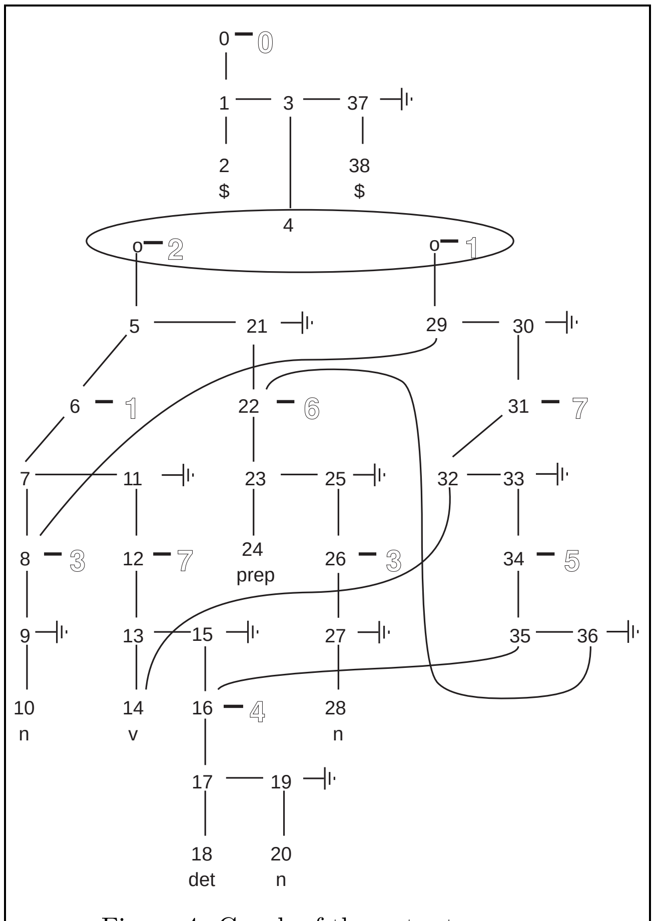

Figure 4: Graph of the output grammar

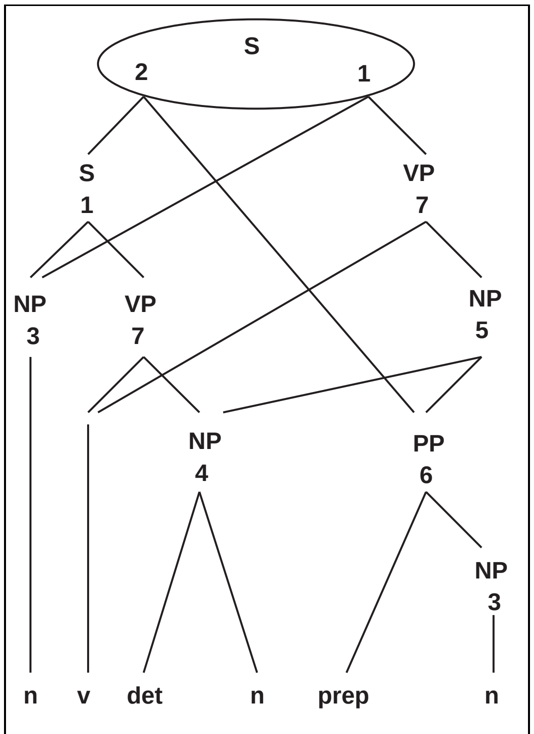

Figure 5: The shared forest

This is certainly the case for all left-to-right CF chart parsing schemata.

We have formally extended the concept of PDA into that of Logical PDA which is an operational push-down stack device for parsing unification based grammars [17, 18] or other non-CF grammars such as Tree Adjoining Grammars [19]. Hence we are reusing and developing our theoretical [18] and experimental [36] approach in this much more general setting which is more likely to be effectively usable for natural language parsing.

Furthermore, these extensions can also express, within the PDA model, non-left-to-right behavior such as is used in island parsing [38] or in Sheil's approach [26]. More generally they allow the formal analysis of agenda strategies, which we have not considered here. In these extensions, the counterpart of parse forests are proof forests of definite clause programs.

6 Conclusion

Analysis of all-path parsing schemata within a common framework exhibits in comparable terms the properties of these schemata, and gives objective criteria for chosing a given schema when implementing a language analyzer. The approach taken here supports both theoretical analysis and actual experimentation, both for the computational behavior of parsers and for the structure of the resulting shared forest. Many experiments and extensions still remain to be made: improved dynamic programming interpretation of bottom-up parsers, more extensive experimental measurements with a variety of languages and parsing schemata, or generalization of this approach to more complex situations, such as word lattice parsing [21, 30], or even handling of "secondary" language features. Early research in that latter direction is promising: our framework and the corresponding paradigm for parser construction have been extended to full first-order Horn clauses [17, 18], and are hence applicable to unification based grammatical formalisms [27]. Shared forest construction and analysis can be generalized in the same way to these more advanced formalisms.

Acknowledgements: We are grateful to Véronique Donzeau-Gouge for many fruitful discussions.

This work has been partially supported by the Eureka Software Factory (ESF) project.

References

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

[2] Billot, S. 1988 *Analyseurs Syntaxiques et Non-Déterminisme*. Thèse de Doctorat, Université d'Orléans la Source, Orléans (France).

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

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

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

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

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

[8] Futamura, Y. (ed.) 1988 Proceedings of the Workshop on Partial Evaluation and Mixed Computation. *New Generation Computing* 6(2,3).

[9] 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.

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

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

[12] 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.

[13] 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.

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

[15] Lang, B. 1974 Deterministic Techniques for Efficient Non-deterministic Parsers. *Proc. of the 2nd 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).

[16] Lang, B. 1988 Parsing Incomplete Sentences. *Proc. of the 12th Internat. Conf. on Computational Linguistics (COLING'88)* Vol. 1 :365-371, D. Vargha (ed.), Budapest (Hungary).

[17] Lang, B. 1988 Datalog Automata. *Proc. of the 3rd Internat. Conf. on Data and Knowledge Bases*, C. Beeri, J.W. Schmidt, U. Dayal (eds.), Morgan Kaufmann Pub., pp. 389-404, Jerusalem (Israel).

[18] Lang, B. 1988 *Complete Evaluation of Horn Clauses, an Automata Theoretic Approach.* INRIA Research Report 913.

[19] Lang, B. 1988 *The Systematic Construction of Earley Parsers: Application to the Production of $\mathcal{O}(n^6)$ Earley Parsers for Tree Adjoining Grammars.* In preparation.

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

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

[22] Pereira, F.C.N.; and Warren, D.H.D. 1980 Definite Clause Grammars for Language Analysis — Asurvey of the Formalism and a Comparison with Augmented Transition Networks. *Artificial Intelligence* 13: 231-278.

[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] Rekers, J. 1987 *A Parser Generator for Finitely Ambiguous Context-Free Grammars.* Report CS-R8712, Computer Science/Dpt. of Software Technology, Centrum voor Wiskunde en Informatica, Amsterdam (The Netherlands).

[26] 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).

[27] Shieber, S.M. 1984 The Design of a Computer Language for Linguistic Information. *Proc. of the 10th Internat. Conf. on Computational Linguistics — COLING'84*: 362-366, Stanford (California).

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

[29] Thompson, H. 1983 MCHART: A Flexible, Modular Chart Parsing System. Proc. of the National Conf. on Artificial Intelligence (AAAI-83), Washington (D.C.), pp. 408-410.

[30] 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.

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

[32] Tomita, M. 1988 Graph-structured Stack and Natural Language Parsing. *Proceedings of the 26th Annual Meeting of the Association for Computational Linguistics*: 249-257.

[33] Uehara, K.; Ochitani, R.; Kakusho, O.; Toyoda, J. 1984 A Bottom-Up Parser based on Predicate

Logic: A Survey of the Formalism and its Implementation Technique. *1984 Internat. Symp. on Logic Programming*, Atlantic City (New Jersey), : 220-227.

[34] U.S. Department of Defense 1983 *Reference Manual for the Ada Programming Language.* ANSI/MIL–STD–1815 A.

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

[36] Villemonte de la Clergerie, E.; and Zanchetta, A. 1988 *Evaluateur de Clauses de Horn.* Rapport de Stage d'Option, Ecole Polytechnique, Palaiseau (France).

[37] Wirth, N. 1971 The Programming Language Pascal. *Acta Informatica*, 1(1).

[38] Ward, W.H.; Hauptmann, A.G.; Stern, R.M.; and Chanak, T. 1988 Parsing Spoken Phrases Despite Missing Words. In *Proceedings of the 1988 International Conference on Acoustics, Speech, and Signal Processing (ICASSP 88)*, Vol. 1: 275-278.

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

A The algorithm

This is the formal description of a *minimal* dynamic programming PDT interpreter. The actual Tin interpreter has a larger instruction set. Comments are prefixed with "—".

```
— Begin parse with input sentence $x$ of length $n$

step-A: — Initialization
    $\overset{\circ}{U}$ := $((\overset{\circ}{q}\,\overset{\circ}{\$}\,0)\,(\overset{\circ}{q}\,\overset{\circ}{\$}\,0))$;     — initial item
    $\overset{\circ}{\pi}$ := $(\overset{\circ}{U} \rightarrow \epsilon)$;            — first rule of output grammar
    $\mathcal{S}_0$ := $\{\overset{\circ}{U}\}$;              — 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
while $i \leq n$ loop
  for every item $U = ((p\ A\ i)\ (q\ B\ j))$ in $\mathcal{S}_i$ do
  for every  transition $\tau$ in $\delta$ do
  — we consider four kinds of transitions, corresponding
  — to the instructions of a minimal PDT interpreter.
    if $\tau = (p\ \epsilon\ \epsilon\ \mapsto\ r\ \epsilon\ z)$ then            — OUTPUT $z$
      V := $((r\ A\ i)\ (q\ B\ j))$;
      $\mathcal{S}_i$ := $\mathcal{S}_i \cup \{V\}$;
      $\mathcal{P}$ := $\mathcal{P} \cup \{(V \rightarrow Uz)\}$;
    if $\tau = (p\ \epsilon\ \epsilon\ \mapsto\ r\ C\ \epsilon)$ then            — PUSH C
      V := $((r\ C\ i)\ (p\ A\ i))$;
      $\mathcal{S}_i$ := $\mathcal{S}_i \cup \{V\}$;
      $\mathcal{P}$ := $\mathcal{P} \cup \{(V \rightarrow \epsilon)\}$;
    if $\tau = (p\ A\ \epsilon\ \mapsto\ r\ \epsilon\ \epsilon)$ then            — POP A
      for every item $Y = ((q\ B\ j)\ (s\ D\ k))$ in $\mathcal{S}_j$ do
        V := $((r\ B\ i)\ (s\ D\ k))$;
        $\mathcal{S}_i$ := $\mathcal{S}_i \cup \{V\}$;
        $\mathcal{P}$ := $\mathcal{P} \cup \{(V \rightarrow YU)\}$;
    if $\tau = (p\ \epsilon\ a\ \mapsto\ r\ \epsilon\ \epsilon)$ then            — SHIFT a
      V := $((r\ A\ i{+}1)\ (q\ B\ j))$;
      $\mathcal{S}_{i+1}$ := $\mathcal{S}_{i+1} \cup \{V\}$;
      $\mathcal{P}$ := $\mathcal{P} \cup \{(V \rightarrow U)\}$;
  $i$ := $i + 1$;
end loop;
step-C: — Termination
 for every item U = $((f\ \overset{\circ}{\$}\ n)\ (\overset{\circ}{q}\,\overset{\circ}{\$}\,0))$  in $\mathcal{S}_n$
       such that $f \in F$ do
   $\mathcal{P}$ := $\mathcal{P} \cup (U_f \rightarrow U)$;  — $U_f$ is the initial nonterminal of $\mathcal{G}$.
— End of parse
```

B Interpretation of a top-down PDT

To illustrate the creation of the shared forest, we present here informally a simplified sequence of transitions in their order of execution by a top-down parser. We indicate the transitions as Tin instructions on the left, as defined in appendix A. On the right we indicate the item and the rule produced by execution of each instruction: the item is the left-hand-side of the rule.

The pseudo-instruction *scan* is given in italics because it does not exist, and stands for the parsing of a sub-constituent: either several transitions for a complex constituent or a single `shift` instruction for a lexical constituent. The global behavior of *scan* is the same as that of `shift`, and it may be understood as a shift on the whole sub-constituent.

Items are represented by a pair of integer. Hence we give no details about states or input, but keep just enough information to see how items are inter-related when applying a `pop` transition: it must use two items of the form `(a,b)` and `(b,c)` as indicated by the algorithm.

The symbol π stands for the rule used to recognize a constituent s, and π_i stands for the rule used to recognize its i^{th} sub-constituent s_i. The whole sequence, minus the first and the last two instructions, would be equivalent to "*scan s*".

```
...              (6,5)
push $\pi$           (7,6)   -> $\epsilon$
push $\pi_1$          (8,7)   -> $\epsilon$
scan $s_1$           (9,7)   -> (8,7) $s_1$
out $\pi_1$           (10,7)  -> (9,7) $\pi_1$
pop              (11,6)  -> (7,6) (10,7)
push $\pi_2$          (12,11) -> $\epsilon$
scan $s_2$           (13,11) -> (12,11) $s_2$
out $\pi_2$           (14,11) -> (13,11) $\pi_2$
pop              (15,6)  -> (11,6) (14,11)
push $\pi_3$          (16,15) -> $\epsilon$
scan $s_3$           (17,15) -> (16,15) $s_3$
out $\pi_3$           (18,15) -> (17,15) $\pi_3$
pop              (19,6)  -> (15,6) (18,15)
out $\pi$            (20,6)  -> (19,6) $\pi$
pop              (21,5)  -> (6,5)(20,6)
...
```

This grammar may be simplified by eliminating useless non-terminals, deriving on the empty string ϵ or on a single other non-terminal. As in section 4, the simplified grammar may then be represented as a graph which is similar, with more details (the rules used for the subconstituents), to the graph given in figure 2.

C Experimental Comparisons

This appendix gives some of the experimental data gathered to compare compilation schemata.

For each grammar, the first table gives the size of the PDTs obtained by compiling it according to several compilation schemata. This size corresponds to the number of instructions generated for the PDT, which is roughly the number of possible PDT states.

The second table gives two figures for each schema and for several input sentences. The first figure is the number of items computed to parse that sentence with the given schema: it may be read as the number of computation steps and is thus a measure of computational efficiency. The second figure is the number of items remaining after simplification of the output grammar: it is thus an indicator of sharing quality. Sharing is better when this second figure is low.

In these tables, columns headed with LR/LALR stands for the LR(0), LR(1), LALR(1) and LALR(2) cases (which often give the same results), unless one of these cases has its own explicit column.

Tests were run on the GRE, NSE, UBDA and RR grammars of [3]: they did not exhibit the loss of efficiency with increased look-ahead that was reported for the bottom-up look-ahead of [3].

We believe the results presented here are consistent and give an accurate comparison of performances of the parsers considered, despite some implementation departure from the strict theoretical model required by performance considerations. A first version of our LL(0) compiler gave results that were inconsistent with the results of the bottom-up parsers. This was a clue to a weakness in that LL(0) compiler which was then corrected. We consider this experience to be a confirmation of the usefulness of our uniform framework.

It must be stressed that these are preliminary experiments. On the basis of their analysis, we intend a new set of experiments that will better exhibit the phenomena discussed in the paper. In particular we wish to study variants of the schemata and dynamic programming interpretation that give the best possible sharing.

C.1 Grammar UBDA

```
A ::= A A | a
```

LR(0)	LR(1)	LALR(1)	LALR(2)	preced.	LL(0)
38	60	41	41	36	46

input string	LR/LALR	preced.	LL(0)
`a`	14 - 9	15 - 9	41 - 9
`aa`	23 - 15	29 - 15	75 - 15
`aaaaaa`	249 - 156	226 - 124	391 - 112

C.2 Grammar RR

```
A ::= x A | x
```

This grammar is LALR(1) but not LR(0), which explains the lower performance of the LR(0) parser.

LR(0)	LR(1)	LALR(1)	LALR(2)	preced.	LL(0)
34	37	37	37	48	46

input string	LR(0)	LR/LALR	preced.	LL(0)
`x`	14 - 9	14 - 9	15 - 9	28 - 9
`xx`	23 - 13	20 - 13	25 - 13	43 - 13
`xxxxxx`	99 - 29	44 - 29	56 - 29	123 - 29

C.3 Picogrammar of English

```
S ::= NP VP | S PP
NP ::= n | det n | NP PP
VP ::= v NP
PP ::= prep NP
```

LR(0)	LR(1)	LALR(1)	LALR(2)	preced.	LL(0)
110	341	104	104	90	116

input string	LR/LALR	preced.	LL(0)
`n v n prep n`	71 -47	72 - 47	169 - 43
`n v n (prep n)`^{2}	146 - 97	141 - 93	260 - 77
`n v n (prep n)`^{3}	260 - 172	245 - 161	371 - 122
`n v n (prep n)`^{6}	854 - 541	775 - 491	844 - 317

C.4 Grammar of Ada expressions

This grammar, too long for inclusion here, is the grammar of expressions of the language Ada, as given in the reference manual [34]. This grammar is ambiguous.

In these examples, the use of look-ahead give approximately a 25% gain in speed efficiency over LR(0) parsing, with the same forest sharing.

However the use of look-ahead may increase the LR(1) parser size quadratically with the grammar size. Still, a better engineered LR(1) construction should not usually increase that size as dramatically as indicated by our experimental figure.

LR(0)	LR(1)	LALR(1)	preced.
587	32210	534	323

input string	LR(0)	LR(1)	LALR(1)	preced.
a*3	74 - 39	59 - 39	59 - 39	80 - 39
(a*3)+b	137 - 75	113 - 75	113 - 75	293 - 75
a*3+b**4	169 - 81	122 - 81	122 - 81	227 - 81

C.5 Grammar PB

```
E ::= a A d | a B c | b A c | b B d
A ::= e
B ::= e
```

LR(0)	LR(1)	LALR(1) & (2)	preced.	LL(0)
76	100	80	84	122

This grammar is LR(1) but is not LALR. For each compilation schema, it gives the same result on all possible inputs: `aed`, `aec`, `bec` and `bed`.

LR(0)	LR(1)	LALR(1) & (2)	preced.	LL(0)
26 - 15	23 - 15	26 - 15	29 - 15	47 - 15

C.6 Grammar SBBL

```
E ::= X A d | X B c | Y A c | Y B d
X ::= f
Y ::= f
A ::= e A | g
B ::= e A | g
```

LR(0)	LR(1)	LALR(1)	LALR(2)	preced.
159	294	158	158	104

input string	LR(0)	LR(1)	LALR(1) & (2)	preced.
`fegd`	50 - 21	57 - 37	50 - 21	84 - 36
`feeegd`	62 - 29	75 - 49	62 - 29	110 - 44

The terminal `f` may be ambiguously parsed as `X` or as `Y`. This ambiguous left context increases uselessly the complexity of the LR(1) parses during recognition of the `A` and `B` constituents. Hence LR(0) performs better in this case since it ignores the context.