PRINCIPLES AND IMPLEMENTATION OF DEDUCTIVE PARSING

STUART M. SHIEBER, YVES SCHABES, * AND FERNANDO C. N. PEREIRA†

▷ We present a system for generating parsers based directly on the metaphor of parsing as deduction. Parsing algorithms can be represented directly as deduction systems, and a single deduction engine can interpret such deduction systems so as to implement the corresponding parser. The method generalizes easily to parsers for augmented phrase structure formalisms, such as definite-clause grammars and other logic grammar formalisms, and has been used for rapid prototyping of parsing algorithms for a variety of formalisms including variants of tree-adjoining grammars, categorial grammars, and lexicalized context-free grammars. ◁

1. INTRODUCTION

Parsing can be viewed as a deductive process that seeks to prove claims about the grammatical status of a string from assumptions describing the grammatical properties of the string's elements and the linear order between them. Lambek's syntactic calculi [15] comprise an early formalization of this idea, which more recently was explored in relation to grammar formalisms based on definite clauses [7, 23, 24] and on feature logics [35, 27, 6].

The view of parsing as deduction adds two main new sources of insights and techniques to the study of grammar formalisms and parsing:

Address correspondence to Stuart M. Shieber, Division of Applied Sciences, Harvard University, Cambridge, MA 02138.
*Mitsubishi Electric Research Laboratories, Cambridge, MA 02139.
†AT&T Bell Laboratories, Murray Hill, NJ 07974.

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010 0743-1066/93/$3.50

1. Existing logics can be used as a basis for new grammar formalisms with desirable representational or computational properties.
2. The modular separation of parsing into a logic of grammaticality claims and a proof search procedure allows the investigation of a wide range of parsing algorithms for existing grammar formalisms by selecting specific classes of grammaticality claims and specific search procedures.

While most of the work on deductive parsing has been concerned with (1), we will in this paper investigate (2), more specifically how to synthesize parsing algorithms by combining specific logics of grammaticality claims with a fixed search procedure. In this way, deduction can provide a metaphor for parsing that encompasses a wide range of parsing algorithms for an assortment of grammatical formalisms. We flesh out this metaphor by presenting a series of parsing algorithms literally as inference rules, and by providing a uniform deduction engine, parameterized by such rules, that can be used to parse according to any of the associated algorithms. The inference rules for each logic will be represented as unit clauses and the fixed deduction procedure, which we provide a Prolog implementation of, will be a version of the usual bottom-up consequence closure operator for definite clauses. As we will show, this method directly yields dynamic-programming versions of standard top-down, bottom-up, and mixed-direction (Earley) parsing procedures. In this, our method has similarities with the use of pure bottom-up deduction to encode dynamic-programming versions of definite-clause proof procedures in deductive databases [3, 19].

The program that we develop is especially useful for rapid prototyping of and experimentation with new parsing algorithms, and was in fact developed for that purpose. We have used it, for instance, in the development of algorithms for parsing with tree-adjoining grammars, categorial grammars, and lexicalized context-free grammars.

Many of the ideas that we present are not new. Some have been presented before; others form part of the folk wisdom of the logic programming community. However, the present work is to our knowledge the first to make the ideas available explicitly in a single notation and with a clean implementation. In addition, certain observations regarding efficient implementation may be novel to this work.

The paper is organized as follows: After reviewing some basic logical and grammatical notions and applying them to a simple example (Section 2), we describe how the structure of a variety of parsing algorithms for context-free grammars can be expressed as inference rules in specialized logics (Section 3). Then, we extend the method for stating and implementing parsing algorithms for formalisms other than context-free grammars (Section 4). Finally, we discuss how deduction should proceed for such logics, developing an agenda-based deduction procedure implemented in Prolog that manifests the presented ideas (Section 5).

2. BASIC NOTIONS

As introduced in Section 1, we see parsing as a deductive process in which rules of inference are used to derive statements about the grammatical status of strings from other such statements. Statements are represented by formulas in a suitable formal language. The general form of a rule of inference is

$$\frac{A_1 \quad \cdots \quad A_k}{B} \quad \langle \text{side conditions on } A_1, \ldots, A_k, B \rangle \qquad .$$

The *antecedents* $A_1, \ldots, A_k$ and the *consequent* B of the inference rule are formula schemata, that is, they may contain syntactic metavariables to be instantiated by appropriate terms when the rule is used. A *grammatical deduction system* is defined by a set of rules of inference and a set of *axioms* given by appropriate formula schemata.

Given a grammatical deduction system, a *derivation* of a formula B from assumptions $A_1, \ldots, A_m$ is, as usual, a sequence of formulas $S_1, \ldots, S_n$ such that $B = S_n$, and for each S_i, either s_i is one of the A_j, or S_i is an instance of an axiom, or there is a rule of inference R and formulas $S_{i_1}, \ldots, S_{i_k}$ with $i_1, \ldots, i_k < i$ such that for appropriate substitutions of terms for the metavariables in R, $S_{i_1}, \ldots, S_{i_k}$ match the antecedents of the rule, S_i matches the consequent, and the rule's side conditions are satisfied. We write $A_1, \ldots, A_m \vdash B$ and say that B is a *consequence* of $A_1, \ldots, A_m$ if such a derivation exists. If B is a consequence of the empty set of assumptions, it is said to be *derivable*, in symbols $\vdash B$.

In our applications of this model, rules and axiom schemata may refer in their side conditions to the rules of a particular grammar, and formulas may refer to string positions in the fixed string to be parsed $w = w_1 \cdots w_n$. With respect to the given string, *goal formulas* state that the string is grammatical according to the given grammar. Then parsing the string corresponds to finding a derivation witnessing a goal formula.

We will use standard notation for metavariables ranging over the objects under discussion: n for the length of the object language string to be parsed; $A, B, C \ldots$ for arbitrary formulas or symbols such as grammar nonterminals; $a, b, c, \ldots$ for arbitrary terminal symbols; $i, j, k, \ldots$ for indices into various strings, especially the string w; $\alpha, \beta, \gamma, \ldots$ for strings or terminal and nonterminal symbols. We will often use such notations leaving the type of the object implicit in the notation chosen for it. Substrings will be notated elliptically as, e.g., $w_i \cdots w_j$ for the i-th through j-th elements of w, inclusive. As is usual, we take $w_i \cdots w_j$ to be the empty string if $i > j$.

2.1. A First Example: CYK Parsing

As a simple example, the basic mechanism of the Cocke-Younger-Kasami (CYK) context-free parsing algorithm [12, 38] for a context-free grammar in Chomsky normal form can be easily represented as a grammatical deduction system.

We assume that we are given a string $w = w_1 \cdots w_n$ to be parsed and a context-free grammar $G = \langle N, \Sigma, P, S \rangle$, where N is the set of nonterminals including the start symbol S, Σ is the set of terminal symbols, ($V = N \cup \Sigma$ is the vocabulary of the grammar,) and P is the set of productions, each of the form $A \rightarrow \alpha$ for $A \in N$ and $\alpha \in V^*$. We will use the symbol $\Rightarrow$ for immediate derivation and $\stackrel{*}{\Rightarrow}$ for its reflexive, transitive closure, the derivation relation. In the case of a Chomsky-normal-form grammar, all productions are of the form $A \rightarrow B\ C$ or $A \rightarrow a$.

The *items* of the logic (as we will call parsing logic formulas from now on) are of the form $[A, i, j]$, and state that the nonterminal A derives the substring between

Item form:	$[A, i, j]$
Axioms:	$[A, i, i+1] \quad A \rightarrow w_{i+1}$
Goals:	$[S, 0, n]$
Inference rules:	$\dfrac{[B, i, j] \qquad [C, j, k]}{[A, i, k]} \quad A \rightarrow B\ C$

Figure 1. The CYK deductive parsing system.

indices i and j in the string, that is, $A \stackrel{*}{\Rightarrow} w_{i+1} \cdots w_j$. Sound axioms, then, are grounded in the lexical items that occur in the string. For each word w_{i+1} in the string and each rule $A \rightarrow w_{i+1}$, it is clear that the item $[A, i, i+1]$ makes a true claim, so that such items can be taken as axiomatic. Then whenever we know that $B \stackrel{*}{\Rightarrow} w_{i+1} \cdots w_j$ and $C \stackrel{*}{\Rightarrow} w_{j+1} \cdots w_k$ — as asserted by items of the form $[B, i, j]$ and $[C, j, k]$ — where $A \rightarrow B\ C$ is a production in the grammar, it is sound to conclude that $A \stackrel{*}{\Rightarrow} w_{i+1} \cdots w_k$, and therefore, the item $[A, i, k]$ should be inferable. This argument can be codified in a rule of inference:

$$\frac{[B, i, j] \qquad [C, j, k]}{[A, i, k]} \quad A \rightarrow B\ C$$

Using this rule of inference with the axioms, we can conclude that the string is admitted by the grammar if an item of the form $[S, 0, n]$ is deducible, since such an item asserts that $S \stackrel{*}{\Rightarrow} w_1 \cdots w_n = w$. We think of this item as the *goal item* to be proved.

In summary, the CYK deduction system (and all the deductive parsing systems we will define) can be specified with four components: a class of items; a set of axioms; a set of inference rules; and a subclass of items, the goal items. These are given in summary form in Figure 1.

This deduction system can be encoded straightforwardly by the following logic program:

```
nt(A, I1, I) :-
    word(I, W),
    (A ---> [W]),
    I1 is I - 1.
nt(A, I, K) :-
    nt(B, I, J),
    nt(C, J, K),
    (A ---> [B, C]).
```

where A `--->` `[`X_1`,`$\ldots$`,`X_m`]` is the encoding of a production $A \rightarrow X_1 \cdots X_n$ in the grammar and `word(`i`,`w_i`)` holds for each input word w_i in the string to be parsed. A suitable bottom-up execution of this program, for example using the *semi-naïve* bottom-up procedure [19] will behave similarly to the CYK algorithm on the given grammar.

2.2. Proofs of Correctness

Rather than implement each deductive system like the CYK one as a separate logic program, we will describe in Section 5 a meta-interpreter for logic programs obtained from grammatical deduction systems. The meta-interpreter is just a variant of the semi-naïve procedure specialized to programs implementing grammatical deduction systems. We will show in Section 5 that our procedure generates only items derivable from the axioms (*soundness*) and will enumerate all the derivable items (*completeness*). Therefore, to show that a particular parsing algorithm is correctly simulated by our meta-interpreter, we basically need to show that the corresponding grammatical deduction system is also sound and complete with respect to the intended interpretation of grammaticality items. By sound here we mean that every derivable item represents a true grammatical statement under the intended interpretation, and by complete we mean that the item encoding every true grammatical statement is derivable. (We also need to show that the grammatical deduction system is faithfully represented by the corresponding logic program, but in general this will be obvious by inspection.)

3. DEDUCTIVE PARSING OF CONTEXT-FREE GRAMMARS

We begin the presentation of parsing methods stated as deduction systems with several standard methods for parsing context-free grammars. In what follows, we assume that we are given a string $w = w_1 \cdots w_n$ to be parsed along with a context-free grammar $G = \langle N, \Sigma, P, S \rangle$.

3.1. Pure Top-Down Parsing (Recursive Descent)

The first full parsing algorithm for arbitrary context-free grammars that we present from this logical perspective is recursive-descent parsing. Given a context-free grammar $G = \langle N, \Sigma, P, S \rangle$, and a string $w = w_1 \cdots w_n$ to be parsed, we will consider a logic with items of the form $[\,\bullet\, \beta, j]$ where $0 \leq j \leq n$. Such an item asserts that the substring of the string w up to and including the j-th element, when followed by the string of symbols β, forms a sentential form of the language, that is, that $S \stackrel{*}{\Rightarrow} w_1 \cdots w_j \beta$. Note that the dot in the item is positioned just at the break point in the sentential form between the portion that has been recognized (up through index j) and the part that has not (β).

Taking the set of such items to be the formulas of the logic, and taking the informal statement concluding the previous paragraph to provide a denotation for the sentences,[1] we can explore a proof theory for the logic. We start with an axiom

$$[\,\bullet\, S, 0] \qquad ,$$

[1] A more formal statement of the semantics could be given, e.g., as

$$[\![[\,\bullet\, \beta, j]]\!] = \begin{cases} \mathit{truth} & \text{if } S \stackrel{*}{\Rightarrow} w_1 \cdots w_j \beta \\ \mathit{falsity} & \text{otherwise} \end{cases} .$$

Item form: $[\bullet\, \beta, j]$

Axioms: $[\bullet\, S, 0]$

Goals: $[\bullet\, , n]$

Inference rules:

Scanning $\dfrac{[\bullet\, w_{j+1}\beta, j]}{[\bullet\, \beta, j+1]}$

Prediction $\dfrac{[\bullet\, B\beta, j]}{[\bullet\, \gamma\beta, j]} \quad B \to \gamma$

Figure 2. The top-down recursive-descent deductive parsing system.

which is sound because $S \stackrel{*}{\Rightarrow} S$ trivially.

Note that two items of the form $[\bullet\, w_{j+1}\beta, j]$ and $[\bullet\, \beta, j+1]$ make the same claim, namely that $S \stackrel{*}{\Rightarrow} w_1 \cdots w_j w_{j+1}\beta$. Thus, it is clearly sound to conclude the latter from the former, yielding the inference rule:

$$\frac{[\bullet\, w_{j+1}\beta, j]}{[\bullet\, \beta, j+1]} \quad ,$$

which we will call the *scanning* rule.

A similar argument shows the soundness of the *prediction* rule:

$$\frac{[\bullet\, B\beta, j]}{[\bullet\, \gamma\beta, j]} \quad B \to \gamma \quad .$$

Finally, the item $[\bullet\, , n]$ makes the claim that $S \stackrel{*}{\Rightarrow} w_1 \cdots w_n$, that is, that the string w is admitted by the grammar. Thus, if this goal item can be proved from the axiom by the inference rules, then the string must be in the grammar. Such a proof process would constitute a sound recognition algorithm. As it turns out, the recognition algorithm that this logic of items specifies is a pure top-down left-to-right regime, a recursive-descent algorithm. The four components of the deduction system for top-down parsing — class of items, axioms, inference rules, and goal items — are summarized in Figure 2.

To illustrate the operation of these inference rules for context-free parsing, we will use the toy grammar of Figure 3. Given that grammar and the string

$$w_1 w_2 w_3 = \text{a lindy swings} \tag{1}$$

S	$\rightarrow$	$NP\ VP$	Det	$\rightarrow$	a
NP	$\rightarrow$	$Det\ N\ OptRel$	N	$\rightarrow$	lindy
NP	$\rightarrow$	PN	PN	$\rightarrow$	Trip
VP	$\rightarrow$	$TV\ NP$	IV	$\rightarrow$	swings
VP	$\rightarrow$	IV	TV	$\rightarrow$	dances
$OptRel$	$\rightarrow$	$RelPro\ VP$	$RelPro$	$\rightarrow$	that
$OptRel$	$\rightarrow$	ϵ			

Figure 3. An example context-free grammar.

we can construct the following derivation using the rules just given:

1	$[\bullet S, 0]$	AXIOM
2	$[\bullet NP\ VP, 0]$	PREDICT from 1
3	$[\bullet Det\ N\ OptRel\ VP, 0]$	PREDICT from 2
4	$[\bullet \text{a}\ N\ OptRel\ VP, 0]$	PREDICT from 3
5	$[\bullet N\ OptRel\ VP, 1]$	SCAN from 4
6	$[\bullet \text{lindy}\ OptRel\ VP, 1]$	PREDICT from 5
7	$[\bullet OptRel\ VP, 2]$	SCAN from 6
8	$[\bullet VP, 2]$	PREDICT from 7
9	$[\bullet IV, 2]$	PREDICT from 8
10	$[\bullet \text{swings}, 2]$	PREDICT from 9
11	$[\bullet , 3]$	SCAN from 10

The last item is a goal item, showing that the given sentence is accepted by the grammar of Figure 3.

The above derivation, as all the others we will show, contains just those items that are strictly necessary to derive a goal item from the axiom. In general, a complete search procedure, such as the one we describe in Section 5, generates items that are either dead-ends or redundant for a proof of grammaticality. Furthermore, with an ambiguous grammar there will be several essentially different proofs of grammaticality, each corresponding to a different analysis of the input string.

3.1.1. Proof of Completeness We have shown informally above that the inference rules for top-down parsing are sound, but for any such system we also need the guarantee of *completeness*: if a string is admitted by the grammar, then for that string there is a derivation of a goal item from the initial item.

In order to prove completeness, we prove the following lemma: If $S \stackrel{*}{\Rightarrow} w_1 \cdots w_j \gamma$ is a leftmost derivation (where $\gamma \in V^*$), then the item $[\bullet \gamma, j]$ is generated. We must prove all possible instances of this lemma. Any specific instance can be characterized by specifying the string γ and the integer j, since S and $w_1 \cdots w_j$ are fixed. We shall denote such an instance by $\langle \gamma, j \rangle$. The proof will turn on ranking the various instances and proving the result by induction on the rank. The rank of the instance $\langle \gamma, j \rangle$ is computed as the sum of j and the length of a shortest leftmost derivation of $S \stackrel{*}{\Rightarrow} w_1 \cdots w_j \gamma$.

If the rank is zero, then $j = 0$ and $\gamma = S$. Then, we need to show that $[\bullet S, 0]$ is generated, which is the case since it is an axiom of the top-down deduction system.

For the inductive step, let $\langle \gamma, j \rangle$ be an instance of the lemma of some rank $r > 0$, and assume that the lemma is true for all instances of smaller rank. Two cases arise.

Case 1: $S \stackrel{*}{\Rightarrow} w_1 \cdots w_j \gamma$ in one step. Therefore, $S \rightarrow w_1 \cdots w_j \gamma$ is a rule of the grammar. However, since $[\bullet S, 0]$ is an axiom, by one application of the prediction rule (predicting the rule $S \rightarrow w_1 \cdots w_j \gamma$) and j applications of the scanning rule, the item $[\bullet \gamma, j]$ will be generated.

Case 2: $S \stackrel{*}{\Rightarrow} w_1 \cdots w_j \gamma$ in more than one step. Let us assume therefore that $S \stackrel{*}{\Rightarrow} w_1 \cdots w_{j-k} B \gamma' \Rightarrow w_1 \cdots w_j \beta \gamma'$ where $\gamma = \beta \gamma'$ and $B \rightarrow w_{j-k+1} \cdots w_j \beta$. The instance $\langle B\gamma', j-k \rangle$ has a strictly smaller rank than $\langle \gamma, j \rangle$. Therefore, by the induction hypothesis, the item $[\bullet B\gamma', j-k]$ will be generated. But then, by prediction, the item $[\bullet w_{j-k+1} \cdots w_j \beta, j-k]$ will be generated and by k applications of the scanning rule, the item $[\bullet B, j]$ will be generated.

This concludes the proof of the lemma. Completeness of the parser follows as a corollary of the lemma since if $S \stackrel{*}{\Rightarrow} w_1 \cdots w_n$, then by the lemma the item $[\bullet, n]$ will be generated.

Completeness proofs for the remaining parsing logics discussed in this paper could be provided in a similar way by relating an appropriate notion of normal-form derivation for the grammar formalism under consideration to the item invariants.

3.2. Pure Bottom-Up Parsing (Shift-Reduce)

A pure bottom-up algorithm can be specified by such a deduction system as well. Here, the items will have the form $[\alpha \bullet, j]$. Such an item asserts the dual of the assertion made by the top-down items, that $\alpha w_{j+1} \cdots w_n \stackrel{*}{\Rightarrow} w_1 \cdots w_n$ (or, equivalently but less transparently dual, that $\alpha \stackrel{*}{\Rightarrow} w_1 \cdots w_j$). The algorithm is then characterized by the deduction system shown in Figure 4. The algorithm mimics the operation of a nondeterministic shift-reduce parsing mechanism, where the string of symbols preceding the dot corresponds to the current parse stack, and the substring starting at the index j corresponds to the as yet unread input.

The soundness of the inference rules in Figure 4 is easy to see. The antecedent of the shift rule claims that $\alpha w_{j+1} \cdots w_n \stackrel{*}{\Rightarrow} w_1 \cdots w_n$, but that is also what the consequent claims. For the reduce rule, if $\alpha \gamma w_{j+1} \cdots w_n \stackrel{*}{\Rightarrow} w_1 \cdots w_n$ and $B \rightarrow \gamma$, then by definition of $\stackrel{*}{\Rightarrow}$ we also have $\alpha B w_{j+1} \cdots w_n \stackrel{*}{\Rightarrow} w_1 \cdots w_n$. As for completeness, it can be proved by induction on the steps of a reversed rightmost context-free derivation in a way very similar to the completeness proof of the last section.

The following derivation shows the operation of the bottom-up rules on example

Item form: $[\alpha \bullet, j]$

Axioms: $[\bullet, 0]$

Goals: $[S \bullet, n]$

Inference Rules:

Shift $$\frac{[\alpha \bullet, j]}{[\alpha w_{j+1} \bullet, j+1]}$$

Reduce $$\frac{[\alpha\gamma \bullet, j]}{[\alpha B \bullet, j]} \quad B \to \gamma$$

Figure 4. The bottom-up shift-reduce deductive parsing system.

sentence (1):

1	$[\bullet, 0]$	AXIOM
2	[a $\bullet$, 1]	SHIFT from 1
3	$[Det \bullet, 1]$	REDUCE from 2
4	[Det lindy $\bullet$, 2]	SHIFT from 3
5	$[Det\ N \bullet, 2]$	REDUCE from 4
6	$[Det\ N\ OptRel \bullet, 2]$	REDUCE from 5
7	$[NP \bullet, 2]$	REDUCE from 6
8	[NP swings $\bullet$, 3]	SHIFT from 7
9	$[NP\ IV \bullet, 3]$	REDUCE from 8
10	$[NP\ VP \bullet, 3]$	REDUCE from 9
11	$[S \bullet, 3]$	REDUCE from 10

The last item is a goal item, which shows that the sentence is parsable according to the grammar.

3.3. Earley's Algorithm

Stating the algorithms in this way points up the duality of recursive-descent and shift-reduce parsing in a way that traditional presentations do not. The summary presentation in Figure 5 may further illuminate the various interrelationships. As we will see, Earley's algorithm [8] can then be seen as the natural combination of those two algorithms.

In recursive-descent parsing, we keep a partial sentential form for the material yet to be parsed, using the dot at the beginning of the string of symbols to remind us that these symbols come after the point that we have reached in the recognition process. In shift-reduce parsing, we keep a partial sentential form for the material that has already been parsed, placing a dot at the end of the string to remind us that these symbols come before the point that we have reached in the recognition process. In Earley's algorithm we keep both of these partial sentential forms, with the dot marking the point somewhere in the middle where recognition has reached. The dot thus changes from a mnemonic to a necessary role. In addition, Earley's algorithm

Algorithm	*Bottom-Up*	*Top-Down*	*Earley's*
Item form	$[\alpha \bullet, j]$	$[\bullet \beta, j]$	$[i, A \rightarrow \alpha \bullet \beta, j]$
Invariant	$\alpha w_{j+1} \cdots w_n \overset{*}{\Rightarrow} w_1 \cdots w_n$	$S \overset{*}{\Rightarrow} w_1 \cdots w_j \beta$	$S \overset{*}{\Rightarrow} w_1 \cdots w_i A \gamma$ $\alpha w_{j+1} \cdots w_n \overset{*}{\Rightarrow} w_{i+1} \cdots w_n$
Axioms	$[\bullet, 0]$	$[\bullet S, 0]$	$[0, S' \rightarrow \bullet S, 0]$
Goals	$[S \bullet, n]$	$[\bullet, n]$	$[0, S' \rightarrow S \bullet, n]$
Scanning	$\dfrac{[\alpha \bullet, j]}{[\alpha w_{j+1} \bullet, j+1]}$	$\dfrac{[\bullet w_{j+1}\beta, j]}{[\bullet \beta, j+1]}$	$\dfrac{[i, A \rightarrow \alpha \bullet w_{j+1}\beta, j]}{[i, A \rightarrow \alpha w_{j+1} \bullet \beta, j+1]}$
Prediction		$\dfrac{[\bullet B\beta, j]}{[\bullet \gamma\beta, j]} \quad B \rightarrow \gamma$	$\dfrac{[i, A \rightarrow \alpha \bullet B\beta, j]}{[j, B \rightarrow \bullet \gamma, j]} \quad B \rightarrow \gamma$
Completion	$\dfrac{[\alpha\gamma \bullet, j]}{[\alpha B \bullet, j]} \quad B \rightarrow \gamma$		$\dfrac{[i, A \rightarrow \alpha \bullet B\beta, k] \quad [k, B \rightarrow \gamma \bullet, j]}{[i, A \rightarrow \alpha B \bullet \beta, j]}$

Figure 5. Summary of parsing algorithms presented as deductive parsing systems. (In the axioms and goal items of Earley's algorithm, S' serves as a new nonterminal not in N.)

localizes the piece of sentential form that is being tracked to that introduced by a single production. (Because the first two parsers do not limit the information stored in an item to only local information, they are not practical algorithms as stated. Rather, some scheme for sharing information among items would be necessary to make them tractable [16, 4].)

The items of Earley's algorithm are thus of the form $[i, A \rightarrow \alpha \bullet \beta, j]$ where α and β are strings in V^* and $A \rightarrow \alpha\beta$ is a production of the grammar. As was the case for the previous two algorithms, the j index provides the position in the string that recognition has reached, and the dot position marks that point in the partial sentential form. In these items, however, an extra index i marks the starting position of the partial sentential form, as we have localized attention to a single production. In summary, an item of the form $[i, A \rightarrow \alpha \bullet \beta, j]$ makes the top-down claim that $S \stackrel{*}{\Rightarrow} w_1 \cdots w_i A\gamma$, and the bottom-up claim that $\alpha w_{j+1} \cdots w_n \stackrel{*}{\Rightarrow} w_{i+1} \cdots w_n$. The two claims are connected by the fact that $A \rightarrow \alpha\beta$ is a production in the grammar.

The algorithm itself is captured by the specification found in Figure 5. Proofs of soundness and completeness are somewhat more complex than those for the pure top-down and bottom-up cases shown above, and are directly related to the corresponding proofs for Earley's original algorithm [8].

The following derivation, again for sentence (1), illustrates the operation of the Earley inference rules:

1	$[0, S' \rightarrow \bullet\, S, 0]$	AXIOM
2	$[0, S \rightarrow \bullet\, NP\ VP, 0]$	PREDICT from 1
3	$[0, NP \rightarrow \bullet\, Det\ N\ OptRel, 0]$	PREDICT from 2
4	$[0, Det \rightarrow \bullet\, \text{a}, 0]$	PREDICT from 3
5	$[0, Det \rightarrow \text{a} \bullet, 1]$	SCAN from 4
6	$[0, NP \rightarrow Det \bullet N\ OptRel, 1]$	COMPLETE from 3 and 5
7	$[1, N \rightarrow \bullet\, \text{lindy}, 1]$	PREDICT from 6
8	$[1, N \rightarrow \text{lindy} \bullet, 2]$	SCAN from 7
9	$[0, NP \rightarrow Det\ N \bullet OptRel, 2]$	COMPLETE from 6 and 8
10	$[2, OptRel \rightarrow \bullet, 2]$	PREDICT from 9
11	$[0, NP \rightarrow Det\ N\ OptRel \bullet, 2]$	COMPLETE from 9 and 10
12	$[0, S \rightarrow NP \bullet VP, 2]$	COMPLETE from 2 and 11
13	$[2, VP \rightarrow \bullet\, IV, 2]$	PREDICT from 12
14	$[2, IV \rightarrow \bullet\, \text{swings}, 2]$	PREDICT from 13
15	$[2, IV \rightarrow \text{swings} \bullet, 3]$	SCAN from 14
16	$[2, VP \rightarrow IV \bullet, 3]$	COMPLETE from 13 and 15
17	$[0, S \rightarrow NP\ VP \bullet, 3]$	COMPLETE from 12 and 16
18	$[0, S' \rightarrow S \bullet, 3]$	COMPLETE from 1 and 17

The last item is again a goal item, so we have an Earley derivation of the grammaticality of the given sentence.

4. DEDUCTIVE PARSING FOR OTHER FORMALISMS

The methods (and implementation) that we developed have also been used for rapid prototyping and experimentation with parsing algorithms for grammatical frameworks other than context-free grammars. They can be naturally extended to handle augmented phrase-structure formalisms such as logic grammar and constraint-based formalisms. They have been used in the development and testing of algorithms for parsing categorial grammars, tree-adjoining grammars, and lexicalized context-free grammars. In this section, we discuss these and other extensions.

4.1. Augmented Phrase-Structure Formalisms

It is straightforward to see that the three deduction systems just presented can be extended to constraint-based grammar formalisms with a context-free backbone. The basis for this extension goes back to metamorphosis grammars [7] and definite-clause grammars (DCG) [23]. In those formalisms, grammar symbols are first-order terms, which can be understood as abbreviations for the sets of all their ground instances. Then an inference rule can also be seen as an abbreviation for all of its ground instances, with the metagrammatical variables in the rule consistently instantiated to ground terms. Computationally, however, such instances are generated lazily by accumulating the consistency requirements for the instantiation of inference rules as a conjunction of equality constraints and maintaining that conjunction in normal form — sets of variable substitutions — by unification. (This is directly related to the use of unification to avoid "guessing" instances in the rules of existential introduction and universal elimination in a natural-deduction presentation of first-order logic).

We can move beyond first-order terms to general constraint-based grammar formalisms [35, 6] by taking the above constraint interpretation of inference rules as basic. More explicitly, a rule such as Earley completion

$$\frac{[i, A \to \alpha \bullet B\beta, k] \quad [k, B \to \gamma \bullet, j]}{[i, A \to \alpha B \bullet \beta, j]}$$

is interpreted as shorthand for the constrained rule:

$$\frac{[i, A \to \alpha \bullet B\beta, k] \quad [k, B' \to \gamma \bullet, j]}{[i, A' \to \alpha B'' \bullet \beta, j]} \qquad A \doteq A' \text{ and } B \doteq B' \text{ and } B \doteq B''$$

where '$\doteq$' is the term equality predicate for the constraint-based grammar formalism being interpreted [35].

When such a rule is applied, the three constraints it depends on are conjoined with the constraints for the current derivation. In the particular case of first-order terms and antecedent-to-consequent rule application, completion can be given more explicitly as

$$\frac{[i, A \to \alpha \bullet B\beta, k] \quad [k, B' \to \gamma \bullet, j]}{[i, \sigma(A \to \alpha B \bullet \beta), j]} \qquad \sigma = \mathrm{mgu}(B, B') \qquad .$$

where $mgu(B, B')$ is the most general unifier of the terms B and B'. This is the interpretation implemented by the deduction procedure described in the next section.

The move to constraint-based formalisms raises termination problems in proof construction that did not arise in the context-free case. In the general case, this is inevitable, because a formalism like DCG [23] or PATR-II [33] has Turing-machine power. However, even if constraints are imposed on the context-free backbone of the grammar productions to guarantee decidability, such as *offline parsability* [5, 24, 35], the prediction rules for the top-down and Earley systems are problematic. The difficulty is that prediction can feed on its own results to build unboundedly large items. For example, consider the DCG

$$
\begin{aligned}
&s \rightarrow r(0, N) \\
&r(X, N) \rightarrow r(s(X), N)\ b \\
&r(N, N) \rightarrow a
\end{aligned}
$$

It is clear that this grammar accepts strings of the form ab^n with the variable N being instantiated to the unary (successor) representation of n. It is also clear that the bottom-up inference rules will have no difficulty in deriving the analysis of any input string. However, Earley prediction from the item $[0, s \rightarrow\ \bullet\, r(0, N), 0]$ will generate an infinite succession of items:

$$
\begin{aligned}
&[0, s \rightarrow\ \bullet\, r(0, N), 0] \\
&[0, r(0, N) \rightarrow\ \bullet\, r(s(0), N)\ b, 0] \\
&[0, r(s(0), N) \rightarrow\ \bullet\, r(s(s(0)), N)\ b, 0] \\
&[0, r(s(s(0)), N) \rightarrow\ \bullet\, r(s(s(s(0))), N)\ b, 0] \\
&\ldots
\end{aligned}
$$

This problem can be solved in the case of the Earley inference rules by observing that prediction is just used to narrow the number of items to be considered by scanning and completion, by maintaining the top-down invariant $S \stackrel{*}{\Rightarrow} w_1 \cdots w_i A\gamma$. But this invariant is not required for soundness or completeness, since the bottom-up invariant is sufficient to guarantee that items represent well-formed substrings of the input. The only purpose of the top-down invariant is to minimize the number of completions that are actually attempted. Thus the only indispensable role of prediction is to make available appropriate instances of the grammar productions. Therefore, any relaxation of prediction that makes available items of which all the items predicted by the original prediction rule are instances will not affect soundness or completeness of the rules. More precisely, it must be the case that any item $[i, B \rightarrow\ \bullet\, \gamma, i]$ that the original prediction rule would create is an instance of some item $[i, B' \rightarrow\ \bullet\, \gamma', i]$ created by the relaxed prediction rule. A relaxed prediction rule will create no more items than the original predictor, and in fact may create far fewer. In particular, repeated prediction may terminate in cases like the one described above. For example, if the prediction rule applied to $[i, A \rightarrow \alpha \bullet B'\beta, j]$ yields $[i, \sigma(B \rightarrow\ \bullet\, \gamma), i]$ where $\sigma = \mathrm{mgu}(B, B')$, a relaxed prediction rule might yield $[i, \sigma'(B \rightarrow\ \bullet\, \gamma), i]$, where σ' is a less specific substitution than σ chosen so that only a finite number of instances of $[i, B \rightarrow \bullet\gamma, i]$ are ever generated. A similar notion for general constraint grammars is called restriction [34, 35], and a related technique has been used in partial evaluation of logic programs [28].

The problem with the DCG above can be seen as following from the computation of derivation-specific information in the arguments to the nonterminals. However, applications frequently require construction of the derivation for a string (or similar information), perhaps for the purpose of further processing. It is simple enough to augment the inference rules to include with each item a derivation. For the Earley deduction system, the items would include a fourth component representing

Item form:	$[i, A\alpha \bullet \beta, j, D]$
Axioms:	$[0, S' \rightarrow \bullet S, 0, \langle\rangle]$
Goals:	$[0, S' \rightarrow S \bullet, n, D]$

Inference rules:

Scanning $$\frac{[i, A \rightarrow \alpha \bullet w_{j+1}\beta, j, D]}{[i, A \rightarrow \alpha w_{j+1} \bullet \beta, j+1, D]}$$

Prediction $$\frac{[i, A \rightarrow \alpha \bullet B\beta, j, D]}{[j, B \rightarrow \bullet \gamma, j, \langle\rangle]} \quad B \rightarrow \gamma$$

Completion $$\frac{[i, A \rightarrow \alpha \bullet B\beta, k, D_1] \quad [k, B \rightarrow \gamma \bullet, j, D_2]}{[i, A \rightarrow \alpha B \bullet \beta, j, D_1 \cdot tree(B \rightarrow \gamma, D_2)]}$$

Figure 6. The Earley deductive parsing system modified to generate derivation trees.

a sequence of derivation trees, one for each element of the right-hand side of the item before the dot. Each derivation tree has nodes labeled by productions of the grammar. The inference rules would be modified as shown in Figure 6. In the completion rule, we use the following notations: $tree(l, D)$ denotes the tree whose root is labeled by the node label (grammar production) l and whose children are the trees in the sequence D in order; and $S \cdot s$ denotes the appending of the element s at the end of the sequence S.

Of course, use of such rules makes the caching of lemmas essentially useless, as lemmas derived in different ways are never identical. Appropriate methods of implementation that circumvent this problem are discussed in Section 5.4.

4.2. Combinatory Categorial Grammars

A combinatory categorial grammar [1] consists of two parts: (1) a lexicon that maps words to sets of categories; (2) rules for combining categories into other categories.

Categories are built from atomic categories and two binary operators: forward slash (/) and backward slash (\). Informally speaking, words having categories of the form X/Y, $X\backslash Y$, $(W/X)/Y$ etc. are to be thought of as functions over Y's. Thus the category $S\backslash NP$ of intransitive verbs should be interpreted as a function from noun phrases (NP) to sentences (S). In addition, the direction of the slash (forward as in X/Y or backward as in $X\backslash Y$) specifies where the argument must be found, immediately to the right for / or immediately to the left for \.

For example, a CCG lexicon may assign the category $S\backslash NP$ to an intransitive verb (as the word *sleeps*). $S\backslash NP$ identifies the word (*sleeps*) as combining with a (subject) noun phrase (NP) to yield a sentence (S). The back slash (\) indicates that the subject must be found immediately to the left of the verb. The forward slash / would have indicated that the argument must be found immediately to the right of the verb.

Word	*Category*
Trip	NP
merengue	NP
likes	$(S\backslash NP)/NP$
certainly	$(S\backslash NP)/(S\backslash NP)$

Figure 7. An example CCG lexicon.

More formally, categories are defined inductively as follows:[2] Given a set of basic categories,

- Basic categories are categories.
- If c_1 and c_2 are categories, then (c_1/c_2) and $(c_1\backslash c_2)$ are categories.

The lexicon is defined as a mapping f from words to finite sets of categories. Figure 7 is an example of a CCG lexicon. In this lexicon, *likes* is encoded as a transitive verb $(S\backslash NP)/NP$, yielding a sentence (S) when a noun phrase (NP) object is found to its right and when a noun phrase subject (NP) is then found to its left.

Categories can be combined by a finite set of rules that fall in two classes: application and composition.

Application allows the simple combination of a function with an argument to its right (forward application) or to its left (backward application). For example, the sequence $(S\backslash NP)/NP \quad NP$ can be reduced to $S\backslash NP$ by applying the forward application rule. Similarly, the sequence $NP \quad S\backslash NP$ can be reduced to S by applying the backward application rule.

Composition allows to combine two categories in a similar fashion as functional composition. For example, forward composition combines two categories of the form $X/Y \; Y/Z$ to another category X/Z. The rule gives the appearance of "canceling" Y, as if the two categories were numerical fractions undergoing multiplication. This rule corresponds to the fundamental operation of "composing" the two functions, the function X/Y from Y to X and the function Y/Z from Z to Y.

The rules of composition can be specified formally as productions, but unlike the productions of a CFG, these productions are universal over all CCGs. In order to reduce the number of cases, we will use a vertical bar $|$ as an instance of a forward or backward slash, $/$ or $\backslash$. Instances of $|$ in left- and right-hand sides of a single production should be interpreted as representing slashes of the same direction. The symbols X, Y and Z are to be read as variables which match any category.

Forward application:	X	$\rightarrow$	$X/Y \quad Y$
Backward application:	X	$\rightarrow$	$Y \quad X\backslash Y$
Forward composition:	$X\|Z$	$\rightarrow$	$X/Y \quad Y\|Z$
Backward composition:	$X\|Z$	$\rightarrow$	$Y\|Z \quad X\backslash Y$

[2] The notation for backward slash used in this paper is consistent with one defined by Ades and Steedman [1]: $X\backslash Y$ is interpreted as a function from Ys to Xs. Although this notation has been adopted by the majority of combinatory categorial grammarians, other frameworks [15] have adopted the opposite interpretation for $X\backslash Y$: a function from Xs to Ys.

Item form:	$[X, i, j]$
Axioms:	$[X, i, i+1] \qquad \text{where } X \in f(w_{i+1})$
Goals:	$[S, 0, n]$
Inference rules:	
Forward Application	$\dfrac{[X/Y, i, j] \quad [Y, j, k]}{[X, i, k]}$
Backward Application	$\dfrac{[Y, i, j] \quad [X\backslash Y, j, k]}{[X, i, k]}$
Forward Composition 1	$\dfrac{[X/Y, i, j] \quad [Y/Z, j, k]}{[X/Z, i, k]}$
Forward Composition 2	$\dfrac{[X/Y, i, j] \quad [Y\backslash Z, j, k]}{[X\backslash Z, i, k]}$
Backward Composition 1	$\dfrac{[Y/Z, i, j] \quad [X\backslash Y, j, k]}{[X/Z, i, k]}$
Backward Composition 2	$\dfrac{[Y\backslash Z, i, j] \quad [X\backslash Y, j, k]}{[X\backslash Z, i, k]}$

Figure 8. The CCG deductive parsing system.

A string of words is accepted by a CCG, if a specified category (usually S) derives a string of categories that is an image of the string of words under the mapping f.

A bottom-up algorithm — essentially the CYK algorithm instantiated for these productions — can be easily specified for CCGs. Given a CCG, and a string $w = w_1 \cdots w_n$ to be parsed, we will consider a logic with items of the form $[X, i, j]$ where X is a category and i and j are integers ranging from 0 to n. Such an item, asserts that the substring of the string w from the $i+1$-th element up to the j-th element can be reduced to the category X. The required proof rules for this logic are given in Figure 8.

With the lexicon in Figure 7, the string

$$\text{Trip certainly likes merengue} \qquad (2)$$

can be recognized as follows:

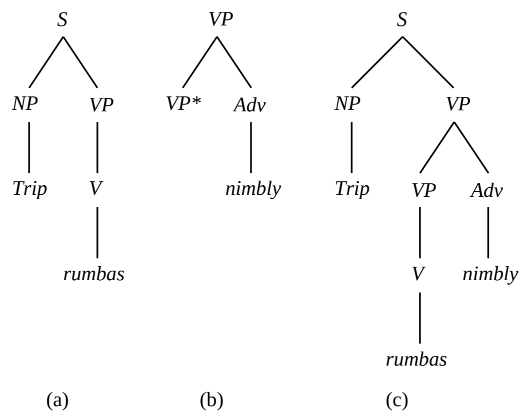

Figure 9. An example tree-adjoining grammar consisting of one initial tree (a), and one auxiliary tree (b). These trees can be used to form the derived tree (c) for the sentence "Trip rumbas nimbly." (In an actual English grammar, the tree depicted in (a) would not be an elementary tree, but itself derived from two trees, one for each lexical item, by a substitution operation.)

1	$[NP, 0, 1]$	AXIOM
2	$[(S\backslash NP)/(S\backslash NP), 1, 2]$	AXIOM
3	$[(S\backslash NP)/NP, 2, 3]$	AXIOM
4	$[(S\backslash NP)/NP, 1, 3]$	FORWARD COMPOSITION from 2 and 3
5	$[NP, 3, 4]$	AXIOM
6	$[(S\backslash NP), 1, 4]$	FORWARD APPLICATION from 4 and 5
7	$[S, 0, 4]$	BACKWARD APPLICATION from 1 and 6

Other extensions of CCG (such as generalized composition and coordination) can be easily implemented using such deduction parsing methods.

4.3. Tree-Adjoining Grammars and Related Formalisms

The formalism of tree-adjoining grammars (TAG) [11, 10] is a tree-generating system in which trees are combined by an operation of adjunction rather than the substitution operation of context-free grammars.[3] The increased expressive power of adjunction allows important natural-language phenomena such as long-distance dependencies to be expressed *locally* in the grammar, that is, within the relevant lexical entries, rather than by many specialized context-free rules [14].

[3] Most practical variants of TAG include both adjunction and substitution, but for purposes of exposition we restrict our attention to adjunction alone, since substitution is formally dispensable and its implementation in parsing systems such as we describe is very much like the context-free operation. Similarly, we do not address other issues such as adjoining constraints and extended derivations. Discussion of those can be found elsewhere [29, 30].

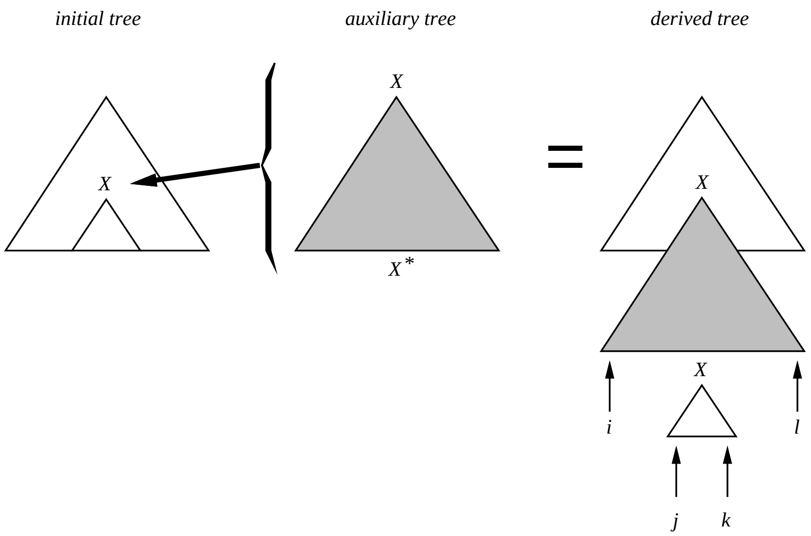

Figure 10. The operation of adjunction. The auxiliary tree is spliced into the initial tree to yield the derived tree at right.

A tree-adjoining grammar consists of a set of *elementary trees* of two types: *initial trees* and *auxiliary trees*. An initial tree is complete in the sense that its frontier includes only terminal symbols. An example is given in Figure 9(a). An auxiliary tree is incomplete; it has a single node on the frontier, the *foot node*, labeled by the same nonterminal as the root. Figure 9(b) provides an example. (By convention, foot nodes are redundantly marked by a diacritic asterisk ($*$) as in the figure.)

Although auxiliary trees do not themselves constitute complete grammatical structures, they participate in the construction of complete trees through the adjunction operation. Adjunction of an auxiliary tree into an initial tree is depicted in Figure 10. The operation inserts a copy of an auxiliary tree into another tree in place of an interior node that has the same label as the root and foot nodes of the auxiliary tree. The subtree that was previously connected to the interior node is reconnected to the foot node of the copy of the auxiliary tree. For example, the auxiliary tree in Figure 9(b) can be adjoined at the *VP* node of the initial tree in Figure 9(a) to form the derived tree in Figure 9(c). Adjunction in effect supports a form of string wrapping and is therefore more powerful than the substitution operation of context-free grammars.

A tree-adjoining grammar can be specified as a quintuple $G = \langle N, \Sigma, I, A, S \rangle$, where N is the set of nonterminals including the start symbol S, Σ is the disjoint set of terminal symbols, I is the set of initial trees, and A is the set of auxiliary trees.

To describe adjunction and TAG derivations, we need notation to refer to tree nodes, their labels, and the subtrees they define. Every node in a tree α can be specified by its *address*, a sequence of positive integers defined inductively as follows:

the empty sequence ϵ is the address of the root node, and $p \cdot k$ is the address of the k-th child of the node at address p. $\mathit{Foot}(\alpha)$ is defined as the address of the foot node of the tree α if there is one; otherwise $\mathit{Foot}(\alpha)$ is undefined.

We denote by $\alpha@p$ the node of α at address p, and by α/p the subtree of α rooted at $\alpha@p$. The grammar symbol that labels node ν is denoted by $\mathit{Label}(\nu)$. Given an elementary tree node ν, $\mathit{Adj}(\nu)$ is defined as the set of auxiliary trees that can be adjoined at node ν.[4]

Finally, we denote by $\alpha[\beta_1 \mapsto p_1, \ldots, \beta_k \mapsto p_k]$ the result of adjoining the trees $\beta_1, \ldots, \beta_k$ at distinct addresses $p_1, \ldots, p_k$ in the tree α.

The set of trees $D(G)$ derived by a TAG G can be defined inductively. $D(G)$ is the smallest set of trees such that

1. $I \cup A \subseteq D(G)$, that is, all elementary trees are derivable, and
2. Define $D(\alpha, G)$ to be the set of all trees derivable as $\alpha[\beta_1 \mapsto p_1, \ldots, \beta_k \mapsto p_k]$ where $\beta_1, \ldots, \beta_k \in D(G)$ and $p_1, \ldots, p_k$ are distinct addresses in α. Then, for all elementary trees $\alpha \in I \cup A$, $D(\alpha, G) \subseteq D(G)$. Obviously, if α is an initial tree, the tree thus derived will have no foot node, and if α is an auxiliary tree, the derived tree will have a foot node.

The valid derivations in a TAG are the trees in $D(\alpha_S, G)$ where α_S is an initial tree whose root is labeled with the start symbol S.

Parsers for TAG can be described just as those for CFG, as deduction systems. The parser we present here is a variant of the CYK algorithm extended for TAGs, similar, though not identical, to that of Vijay-Shanker [36]. We chose it for expository reasons: it is by far the simplest TAG parsing algorithm, in part because it is restricted to TAGs in which elementary trees are at most binary branching, but primarily because it is purely a bottom-up system; no prediction is performed. Despite its simplicity, the algorithm must handle the increased generative capacity of TAGs over that of context-free grammars. Consequently, the worst case complexity for the parser we present is worse than for CFGs — $O(n^6)$ time for a sentence of length n.

The present algorithm uses a *dotted tree* to track the progress of parsing. A dotted tree is an elementary tree of the grammar with a dot adjacent to one of the nodes in the tree. The dot itself may be in one of two positions relative to the specified node: above or below. A dotted tree is thus specified as an elementary tree α, an address p in that tree, and a marker to specify the position of the dot relative to the node. We will use the notation $\nu^\bullet$ and $\nu_\bullet$ for dotted trees with the dot above and below node ν, respectively.[5]

In order to track the portion of the string covered by the production up to the dot position, the CYK algorithm makes use of two indices. In a dotted tree, however,

[4] For TAGs with no constraints on adjunction (for instance, as defined here), $\mathit{Adj}(\nu)$ is just the set of elementary auxiliary trees whose root node is labeled by $\mathit{Label}(\nu)$. When other adjoining constraints are allowed, as is standard, they can be incorporated through a revised definition of Adj.

[5] Although both this algorithm and Earley's use a dot in items to distinguish the progress of a parse, they are used in quite distinct ways. The dot of Earley's algorithm tracks the left-to-right progress of the parse among siblings. The dot of the CYK TAG parser tracks the pre-/post-adjunction status of a single node. For this reason, when generalizing Earley's algorithm to TAG parsing [29], four dot positions are used to simultaneously track pre-/post-adjunction and before/after node left-to-right progress.

there is a further complication in that the elementary tree may contain a foot node so that the string covered by the elementary tree proper has a gap where the foot node occurs. Thus, in general, four indices must be maintained: two (i and l in Figure 10) to specify the left edge of the auxiliary tree and the right edge of the parsed portion (up to the dot position) of the auxiliary tree, and two more (j and k) to specify the substring dominated by the foot node.

The parser therefore consists of inference rules over items of the following forms: $[\nu^{\bullet}, i, j, k, l]$ and $[\nu_{\bullet}, i, j, k, l]$, where

- ν is a node in an elementary tree,
- i, j, k, l are indices of positions in the input string $w_1 \cdots w_n$ ranging over $\{0, \cdots, n\} \cup \{\_\}$, where $\_$ indicates that the corresponding index is not used in that particular item.

An item of the form $[\alpha@p^{\bullet}, i, \_, \_, l]$ specifies that there is a tree $T \in D(\alpha/p, G)$, with no foot node, such that the fringe of T is the string $w_{i+1} \cdots w_l$. An item of the form $[\alpha@p^{\bullet}, i, j, k, l]$ specifies that there is a tree $T \in D(\alpha/p, G)$, with a foot node, such that the fringe of T is the string $w_{i+1} \cdots w_j \; \mathit{Label}(\mathit{Foot}(T)) \; w_{k+1} \cdots w_l$. The invariants for $[\alpha@p_{\bullet}, i, \_, \_, l]$ and $[\alpha@p_{\bullet}, i, j, k, l]$ are similar, except that the derivation of T must not involve adjunction at node $\alpha@p$.

The algorithm preserves this invariant while traversing the derived tree from bottom to top, starting with items corresponding to the string symbols themselves, which follow from the axioms

$$[\nu^{\bullet}, i, \_, \_, i+1] \qquad \mathit{Label}(\nu) = w_{i+1}$$

combining completed subtrees into larger ones, and combining subtrees before adjunction (with dot below) and derived auxiliary trees to form subtrees after adjunction (with dot above). Figure 11 depicts the movement of the dot from bottom to top as parsing proceeds. In Figure 11(a), the basic rules of dot movement not involving adjunction are shown, including the axiom for terminal symbols, the combination of two subchildren of a binary tree and one child of a unary subtree, and the movement corresponding to the absence of an adjunction at a node. These are exactly the rules that would be used in parsing within a single elementary tree. Figure 11(b) displays the rules involved in parsing an adjunction of one tree into another.

These dot movement rules are exactly the inference rules of the TAG CYK deductive parsing system, presented in full in Figure 12. In order to reduce the number of cases, we define the notation $i \cup j$ for two indices i and j as follows:

$$i \cup j = \begin{cases} i & j = \_ \\ j & i = \_ \\ i & i = j \\ \mathit{undefined} & \mathit{otherwise} \end{cases}$$

Although this parser works in time $O(n^6)$ — the Adjoin rule with its six independent indices is the step that accounts for this complexity — and its average behavior may be better, it is in practice too inefficient for practical use for two reasons. First, an attempt is made to parse all auxiliary trees starting bottom-up

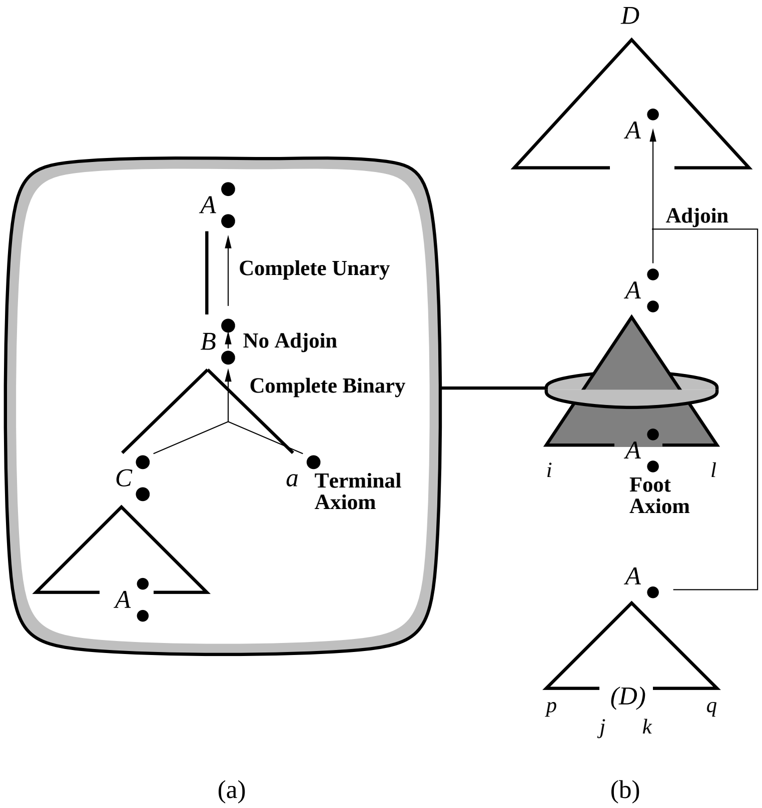

Figure 11. Examples of dot movement in the CYK tree traversal implicit in the TAG parsing algorithm.

Item form: $[\nu^{\bullet}, i, j, k, l]$
$[\nu_{\bullet}, i, j, k, l]$

Axioms:

Terminal Axiom $[\nu^{\bullet}, i, \_, \_, i+1]$ $\quad Label(\nu) = w_{i+1}$

Empty String Axiom $[\nu^{\bullet}, i, \_, \_, i]$ $\quad Label(\nu) = \epsilon$

Foot Axiom $[\beta@Foot(\beta)_{\bullet}, p, p, q, q]$ $\quad \beta \in A$

Goals: $[\alpha@\epsilon^{\bullet}, 0, \_, \_, n]$ $\quad \alpha \in I$ and $Label(\alpha@\epsilon) = S$

Inference Rules:

Complete Unary $$\frac{[\alpha@(p \cdot 1)^{\bullet}, i, j, k, l]}{[\alpha@p_{\bullet}, i, j, k, l]} \qquad \alpha@(p \cdot 2) \text{ undefined}$$

Complete Binary $$\frac{[\alpha@(p \cdot 1)^{\bullet}, i, j, k, l] \quad [\alpha@(p \cdot 2)^{\bullet}, l, j', k', m]}{[\alpha@p_{\bullet}, i, j \cup j', k \cup k', m]}$$

No Adjoin $$\frac{[\nu_{\bullet}, i, j, k, l]}{[\nu^{\bullet}, i, j, k, l]}$$

Adjoin $$\frac{[\beta@\epsilon^{\bullet}, i, p, q, l] \quad [\nu_{\bullet}, p, j, k, q]}{[\nu^{\bullet}, i, j, k, l]} \qquad \beta \in Adj(\nu)$$

Figure 12. The CYK deductive parsing system for tree-adjoining grammars.

from the foot node, regardless of whether the substring between the foot indices actually can be parsed in an appropriate manner. This problem can be alleviated, as suggested by Vijay-Shanker and Weir [37], by replacing the Foot Axiom with a Complete Foot rule that generates the item $[\beta@Foot(\beta)_\bullet, p, p, q, q]$ only if there is an item $[\nu_\bullet, p, j, k, q]$ where $\beta \in Adj(\nu)$, i.e.,

$$\textbf{Complete Foot} \quad \frac{[\nu_\bullet, p, j, k, q]}{[\beta@Foot(\beta)_\bullet, p, p, q, q]} \qquad \beta \in Adj(\nu)$$

This complicates the invariant considerably, but makes auxiliary tree parsing much more goal-directed. Second, because of the lack of top-down prediction, attempts are made to parse elementary trees that are not consistent with the left context. Predictive parsers for TAG can be, and have been, described as deductive systems. For instance, Schabes [29] provides a detailed explanation for a predictive left-to-right parser for TAG inspired by the techniques of Earley's algorithm. Its worst-case complexity is $O(n^6)$ as well, but its average complexity on English grammar is well superior to its worst case, and also to the CYK TAG parser. A parsing system based on this algorithm is currently being used in the development of a large English tree-adjoining grammar at the University of Pennsylvania [21].

Many other formalisms related to tree-adjoining grammars have been proposed, and the deductive parsing approach is applicable to these as well. For instance, as part of an investigation of the precise definition of TAG derivation, Schabes and Shieber describe a compilation of tree-adjoining grammars to linear indexed grammars, together with an efficient algorithm, stated as deduction system for recognition and parsing according to the compiled grammar [30]. A prototype of this parser has been implemented using the deduction engine described here. (In fact, it was as an aid to testing this algorithm, with its eight inference rules each with as many as three antecedent items, that the deductive parsing meta-interpreter was first built.)

Schabes and Waters [31, 32] suggest the use of a restricted form of TAG in which the foot node of an auxiliary tree can occur only at the left or right edge of the tree. Since the portion of string dominated by an auxiliary tree is contiguous under this constraint, only two indices are required to track the parsing of an auxiliary tree adjunction. Consequently, the formalism can generate only context-free languages and can be parsed in cubic time. The resulting system, called tree insertion grammar (TIG), is a compromise between the parsing efficiency of context-free grammar and the elegance and lexical sensitivity of tree-adjoining grammar. TIG has also been used to parse CFGs more quickly by using a construction that converts a context-free grammar into a lexicalized tree insertion grammar (LTIG) that preserves the trees produced. The deductive parsing meta-interpreter has also been used for rapid prototyping of an Earley-style parser for TIG [32].

4.4. Inadequacy for Sequent Calculi

All the parsing logics discussed here have been presented in a natural-deduction format that can be implemented directly by bottom-up execution. However, important parsing logics, in particular the Lambek calculus [15, 18], are better presented in a sequent-calculus format. The main reason for this is that those systems use nonatomic formulas that represent concurrent or hypothetical analyses. For instance, if for arbitrary u with category B we conclude that vu has category A, then in the Lambek calculus we can conclude that v has category A/B.

The main difficulty with applying our techniques to sequent systems is that computationally such systems are designed to be used in a top-down direction. For instance, the rule used for the hypothetical analysis above has the form:

$$\frac{\Gamma B \vdash A}{\Gamma \vdash A/B} \tag{3}$$

It is reasonable to use this rule in a goal-directed fashion (consequent to antecedent) to show $\Gamma \vdash A/B$, but using it in a forward direction is impractical, because B must be arbitrarily assumed before knowing whether the rule is applicable.

More generally, in sequent formulations of syntactic calculi the goal sequent for showing the grammaticality of a string w_i has the form

$$W_1 \cdots W_n \vdash S$$

where W_i gives the grammatical category of w_i and S is the category of a sentence. Proof search proceeds by matching current sequents to the consequents of rules and trying to prove the corresponding antecedents, or by recognizing a sequent as an axiom instance $A \vdash A$. The corresponding natural deduction proof would start from the assumptions $W_1, \ldots, W_n$ and try to prove S, which is just the proof format that we have used here. However, sequent rules like (3) above correspond to the introduction of an additional assumption (not one of the W_i) at some point in the proof and its later discharge, as in the natural-deduction detachment rule for propositional logic. But such undirected introduction of assumptions just in case they may yield consequences that will be needed later is computationally very costly.[6] Systems that make full use of the sequent formulation therefore seem to require top-down proof search. It is of course possible to encode top-down search in a bottom-up system by using more complex encodings of search state, as is done in Earley's algorithm or in the magic sets/magic templates compilation method for deductive databases [3, 25]. Pentus [22], for instance, presents a compilation of Lambek calculus to a CFG, which can then be processed by any of the standard methods. However, it is not clear yet that such techniques can be applied effectively to grammatical sequent calculi so that they can be implemented by the method described here.

5. CONTROL AND IMPLEMENTATION

The specification of inference rules, as carried out in the previous two sections, only partially characterizes a parsing algorithm, in that it provides for what items are to be computed, but not in what order. This further control information is provided by choosing a deduction procedure to operate over the inference rules. If the deduction procedure is complete, it actually makes little difference in what order the items

[6] There is more than a passing similarity between this problem and the problem of pure bottom-up parsing with grammars with gaps. In fact, a natural logical formulation of gaps is as assumptions discharged by the *wh*-phrase they stand for [20, 9].

are enumerated, with one crucial exception: We do not want to enumerate an item more than once. To prevent this possibility, it is standard to maintain a cache of lemmas, adding to the cache only those items that have not been seen so far. The cache plays the same role as the *chart* in chart-parsing algorithms [13], the *well-formed substring table* in CYK parsing [12, 38], and the *state sets* in Earley's algorithm [8]. In this section, we develop a forward-chaining deduction procedure that achieves this elimination of redundancy by keeping a chart.

Items should be added to the chart as they are proved. However, each new item may itself generate new consequences. The issue as to when to compute the consequences of a new item is subtle. A standard solution is to keep a separate *agenda* of items that have been proved but whose consequences have not been computed. When an item is removed from the agenda and added to the chart, its consequences are computed and themselves added to the agenda for later consideration.

Thus, the general form of an agenda-driven, chart-based deduction procedure is as follows:

1. Initialize the chart to the empty set of items and the agenda to the axioms of the deduction system.
2. Repeat the following steps until the agenda is exhausted:
   (a) Select an item from the agenda, called the *trigger item*, and remove it.
   (b) Add the trigger item to the chart, if necessary.
   (c) If the trigger item was added to the chart, generate all items that are new immediate consequences of the trigger item together with all items in the chart, and add these generated items to the agenda.
3. If a goal item is in the chart, the goal is proved (and the string recognized); otherwise it is not.

There are several issues that must be determined in making this general procedure concrete, which we describe under the general topics of *eliminating redundancy* and *providing efficient access*. At this point, however, we will show that, under reasonable assumptions, the general procedure is sound and complete.

In the arguments that follow, we will assume that items are always ground and thus derivations are as defined in Section 2. A proof for the more general case, in which items denote sets of possible grammaticality judgments, would require more intricate definitions for items and inference rules, without changing the essence of the argument.

Soundness We need to show that if the above procedure places item I in the chart when the agenda has been initialized in step (1) with items $A_1, \ldots, A_k$, then $A_1, \ldots, A_k \vdash I$. Since any item in the chart must have been in the agenda, and been placed in the chart by step (2b), it is sufficient to show that $A_1, \ldots, A_k \vdash I$ for any I in the agenda. We show this by induction on the *stage* $\sharp(I)$ of I, the number of the iteration of step (2) at which I has been added to the agenda, or 0 if I has been placed in the agenda at step (1). Note that since several items may be added to the agenda in any given iteration, many items may have the same stage number.

If $\sharp(I) = 0$, I must be an axiom, and thus the trivial derivation consisting of I alone is a derivation of I from $A_1, \ldots, A_k$.

Assume that $A_1, \ldots, A_k \vdash J$ for $\sharp(J) < n$ and that $\sharp(I) = n$. Then I must have been added to the agenda by step (2c), and thus there are items $J_1, \ldots, J_m$ in the chart and a rule instance such that

$$\frac{J_1 \quad \cdots \quad J_m}{I} \quad \langle \text{side conditions on } J_1, \ldots, J_m, I \rangle$$

where the side conditions are satisfied. Since $J_1, \ldots, J_m$ are in the chart, they must have been added to the agenda at the latest at the beginning of iteration n of step (2), that is, $\sharp(J_i) < n$. By the induction hypothesis, each J_i must have a derivation Δ_i from $A_1, \ldots, A_k$. But then, by definition of derivation, the concatenation of the derivations $\Delta_1, \ldots, \Delta_m$ followed by I is a derivation of I from $A_1, \ldots, A_k$.

Completeness We want to show that if $A_1, \ldots, A_k \vdash I$, then I is in the chart at step (3). Actually, we can prove something stronger, namely that I is eventually added to the chart, if we assume some form of *fairness* for the agenda. Then we will have covered cases in which the full iteration of step (2) does not terminate but step (3) can be interleaved with step (2) to recognize the goal as soon as it is generated. The form of fairness we will assume is that if $\sharp(I) < \sharp(J)$ then item I is removed from the agenda by step (2a) before item J. The agenda mechanism described in Section 5.3 below satisfies this fairness assumption.

We show completeness by induction on the length of any derivation $D_1, \ldots, D_n$ of I from $A_1, \ldots, A_k$. (Thus we show implicitly that the procedure generates every derivation, although in general it may share steps among derivations.)

For $n = 1$, $I = D_1 = A_i$ for some i. It will thus be placed in the agenda at step (1), that is $\sharp(I) = 0$. Thus by the fairness assumption I will be removed from the agenda in at most k iterations of step (2). When it is, it is either added to the chart as required, or the chart already contains the same item. (See discussion of the "if necessary" proviso of step (2b) in Section 5.1 below.)

Assume now that the result holds for derivations of length less than n. Consider a derivation $D_1, \ldots, D_n = I$. Either I is an axiom, in which case we have just shown it will have been placed in the chart by iteration k, or, by definition of derivation, there are $i_1, \ldots, i_m < n$ such that there is a rule instance

$$\frac{D_{i_1} \quad \cdots \quad D_{i_m}}{I} \quad \langle \text{side conditions on } D_{i_1}, \ldots D_{i_m}, I \rangle \tag{4}$$

with side conditions satisfied. By definition of derivation, each prefix $D_1, \ldots, D_{i_j}$ of $D_1, \ldots, D_n$ is a derivation of D_{i_j} from $A_1, \ldots, A_k$. Then each D_{i_j} is in the chart, by the induction hypothesis. Therefore, for each D_{i_j} there must have been an identical item I_j in the agenda that was added to the chart at step (2b). Let I_p be the item in question that was the last to be added to the chart. Immediately after that addition, all of the I_j (that is, all of the D_{i_j}) are in the chart, and $I_p = D_{i_p}$ is the trigger item for rule application (4). Thus I is placed in the agenda. Since step (2c) can only add a finite number of items to the agenda, by the fairness assumption item I will eventually be considered at steps (2a) and (2b), and added to the chart if not already there.

5.1. Eliminating Redundancy

Redundancy in the chart. The deduction procedure requires the ability to generate new consequences of the trigger item and the items in the chart. The key word in this requirement is "new". Indeed, the entire point of a chart-based system is to allow caching of proved lemmas so that previously proved (old) lemmas are not further pursued. It is therefore crucial that no item be added to the chart that already exists in the chart, and it is for this reason that step (2b) above specifies addition to the chart only "if necessary".

Definition of "redundant item". The point of the chart is to serve as a cache of previously proved items, so that an item already proved is not pursued. What does it mean for an item to be redundant, that is, occurring already in the agenda or chart? In the case of ground items, the appropriate notion of occurrence in the chart is the existence of an identical chart item. If items can be non-ground (for instance, when parsing relative to definite-clause grammars rather than context-free grammars) a more subtle notion of occurrence in the chart is necessary. As mentioned above, a non-ground item stands for all of its ground instances, so that a non-ground item occurs in the chart if all its ground instances are covered by chart items, that is, if it is a specialization of some chart item. (This test suffices because of the *strong compactness* of sets of terms defined by equations: if the instances of a term A are a subset of the union of the instances of B and C, then the instances of A must be a subset of the instances of either B or C [17].) Thus, the appropriate test is whether an item in the chart subsumes the item to be added.[7]

Redundancy in the agenda. We pointed out that redundancy checking in the chart is necessary. The issue of redundancy in the agenda is, however, a distinct one. Should an item be added to the agenda that already exists there?

Finding the rule that matches a trigger item, triggering the generation of new immediate consequences, and checking that consequences are new are expensive operations to perform. The existence of duplicate items in the agenda therefore generates a spurious overhead of computation especially in pathological cases where exponentially many duplicate items can be created in the agenda, each one creating an avalanche of spurious overhead.

For these reasons, it is also important to check for redundancy in the agenda, that is, the notion of "new immediate consequences" in step (2c) should be interpreted as consequent items that do not already occur in the chart *or agenda*. If redundancy checking occurs at the point items are about to be added to the agenda, it is not required when they are about to be added to the chart; the "if necessary" condition in step (2b) will in this case be vacuous, since always true.

Triggering the generation of new immediate consequences. With regard to step (2c), in which we generate "all items that are new immediate consequences of the trigger item together with all other items in the chart", we would like, if at all possible, to refrain from generating redundant items, rather than generating, checking for, and disposing of the redundant ones. Clearly, any item that is an immediate consequence of the other chart items only (that is, without the trigger item) is not a new consequence of the full chart. (It would have been generated when the last of

[7] This subsumption check can be implemented in several ways in Prolog. The code made available with this paper presents two of the options.

the antecedents was itself added to the chart.) Thus, the inference rules generating new consequences must have at least one of their antecedent items being the trigger item, and the search for new immediate consequences can be limited to just those in which at least one of the antecedents is the trigger item. The search can therefore be carried out by looking at all antecedent items of all inference rules that match the trigger item, and for each, checking that the other antecedent items are in the chart. If so, the consequent of that rule is generated as a potential new immediate consequence of the trigger items plus other chart items. (Of course, it must be checked for prior existence in the agenda and chart as outlined above.)

5.2. Providing Efficient Access

Items should be stored in the agenda and chart in such a way that they can be efficiently accessed. Stored items are accessed at two points: when checking a new item for redundancy and when checking a (non-trigger) antecedent item for existence in the chart. For efficient access, it is desirable to be able to directly index into the stored items appropriately, but appropriate indexing may be different for the two access paths. We discuss the two types of indexing separately, and then turn to the issue of variable renaming.

Indexing for redundancy checking. Consider, for instance, the Earley deduction system. All items that potentially subsume an item $[i, A \rightarrow \alpha \bullet \beta, j]$ have a whole set of attributes in common with the item, for instance, the indices i and j, the production from which the item was constructed, and the position of the dot (i.e., the length of α). Any or all of these might be appropriate for indexing into the set of stored items.

Indexing for antecedent lookup. The information available for indexing when looking items up as potential matches for antecedents can be quite different. In looking up items that match the second antecedent of the completion rule $[k, B \rightarrow \gamma \bullet, j]$, as triggered by an item of the form $[i, A \rightarrow \alpha \bullet B\beta, k]$, the index k will be known, but j will not be. Similarly, information about B will be available from the trigger item, but no information about γ. Thus, an appropriate index for the second antecedent of the completion rule might include its first index k and the main functor of the left-hand-side B. For the first antecedent item, a similar argument calls for indexing by its second index k and the main functor of the nonterminal B following the dot. The two cases can be distinguished by the sequence after the dot: empty in the former case, non-empty in the latter.

Variable renaming. A final consideration in access is the renaming of variables. As non-ground items stored in the chart or agenda are matched against inference rules, they become further instantiated. This instantiation should not affect the items as they are stored and used in proving other consequences, so that care must be taken to ensure that variables in agenda and chart items are renamed consistently before they are used. Prolog provides various techniques for achieving this renaming implicitly.

5.3. Prolog Implementation of Deductive Parsing

In light of the considerations presented above, we turn now to our method of implementing an agenda-based deduction engine in Prolog. We take advantage of certain features that have become standard in Prolog implementations, such as clause indexing. The code described below is consistent with Quintus Prolog.

5.3.1. Implementation of Agenda and Chart Since redundancy checking is to be done in both agenda and chart, we need the entire set of items in both agenda and chart to be stored together. For efficient access, we store them in the Prolog database under the predicate `stored/2`. The agenda and chart are therefore comprised of a series of unit clauses, e.g.,

```
stored(1, item(...)).        ⟵ beginning of chart
stored(2, item(...)).
stored(3, item(...)).
...
stored(i-1, item(...)).      ⟵ end of chart
stored(i, item(...)).        ⟵ head of agenda
stored(i+1, item(...)).
...
stored(k-1, item(...)).
stored(k, item(...)).        ⟵ tail of agenda
```

The first argument of `stored/2` is a unique identifying index that corresponds to the position of the item in the storage sequence of chart and agenda items. (This information is redundantly provided by the clause ordering as well, for reasons that will become clear shortly.) The index therefore allows (through Quintus's indexing of the clauses for a predicate by their first head argument) direct access to any stored item.

Since items are added to the sequence at the end, all items in the chart precede all items in the agenda. The agenda items can therefore be characterized by two indices, corresponding to the first (*head*) and last (*tail*) items in the agenda. A data structure packaging these two "pointers" therefore serves as the proxy for the agenda in the code. An item is moved from the agenda to the chart merely by incrementing the head pointer. Items are added to the agenda by storing the corresponding item in the database and incrementing the tail pointer.

To provide efficient access to the stored items, auxiliary indexing tables can be maintained. Each such indexing table, is implemented as a set of unit clauses that map access keys to the indexes of items that match them. In the present implementation, a single indexing table (under the predicate `key_index/2`) is maintained that is used for accessing items both for redundancy checking and for antecedent lookup. (This is possible because only the item attributes available in both types of access are made use of in the keys, leading to less than optimal indexing for redundancy checking, but use of multiple indexing tables leads to much more database manipulation, which is quite costly.)

In looking up items for redundancy checking, all stored items should be considered, but for antecedent lookup, only chart items are pertinent. The distinction between agenda and chart items is, under this implementation, implicit. The chart items are those whose index is less than the head index of the agenda. This test must be made whenever chart items are looked up. However, since clauses are stored sequentially by index, as soon as an item is found that fails the test (that is, is in the agenda) the search for other chart items can be cut off.

5.3.2. Implementation of the Deduction Engine Given the design decisions described above, the general agenda-driven, chart-based deduction procedure presented in Section 5 can be implemented in Prolog as follows:

```
parse(Value) :-
    % (1) Initialize the chart and agenda
    init_chart,
    init_agenda(Agenda),
    % (2) Remove items from the agenda and process
    %     until the agenda is empty
    exhaust(Agenda),
    % (3) Try to find a goal item in the chart
    goal_item_in_chart(Goal).
```

To exhaust the agenda, trigger items are repeatedly processed until the agenda is empty:

```
exhaust(Empty) :-
    % (2) If the agenda is empty, we're done
    is_empty_agenda(Empty).

exhaust(Agenda0) :-
    % (2a) Otherwise get the next item index from the agenda
    pop_agenda(Agenda0, Index, Agenda1),
    % (2b) Add it to the chart
    add_item_to_chart(Index),
    % (2c) Add its consequences to the agenda
    add_consequences_to_agenda(Index, Agenda1, Agenda),
    % (2) Continue processing the agenda until empty
    exhaust(Agenda).
```

For each item, all consequences are generated and added to the agenda:

```
add_consequences_to_agenda(Index, Agenda0, Agenda) :-
    findall(Consequence,
            consequence(Index, Consequence),
            Consequences),
    add_items_to_agenda(Consequences, Agenda0, Agenda).
```

The predicate `add_items_to_agenda/3` adds the items under appropriate indices as stored items and updates the head and tail indices in `Agenda0` to form the new agenda `Agenda`.

A trigger item has a consequence if it matches an antecedent of some rule, perhaps with some other antecedent items and side conditions, and the other antecedent items have been previously proved (thus in the chart) and the side conditions hold:

```
consequence(Index, Consequent) :-
    index_to_item(Index, Trigger),
    matching_rule(Trigger,
```

```
                    RuleName, Others, Consequent, SideConds),
    items_in_chart(Others, Index),
    hold(SideConds).
```

Note that the indices of items, rather than the items themselves are stored in the agenda, so that the index of the trigger item must first be mapped to the actual item (with `index_to_item/2`) before matching it against a rule antecedent. The `items_in_chart/2` predicate needs to know both the items to look for (`Others`) and the index of the current item (`Index`) as the latter distinguishes the items in the chart (before this index) from those in the agenda (after this index).

We assume that the inference rules are stored as unit clauses under the predicate `inference(RuleName, Antecedents, Consequent, SideConds)` where `RuleName` is some mnemonic name for the rule (such as `predict` or `scan`), `Antecedents` is a list of the antecedent items of the rule, `Consequent` is the single consequent item, and `Sideconds` is a list of encoded Prolog literals to execute as side conditions. To match a trigger item against an antecedent of an inference rule, then, we merely select a rule encoded in this manner, and split up the antecedents into one that matches the trigger and the remaining unmatched antecedents (to be checked for in the chart).

```
matching_rule(Trigger,
              RuleName, Others, Consequent, SideConds) :-
    inference(RuleName, Antecedents, Consequent, SideConds),
    split(Trigger, Antecedents, Others).
```

5.3.3. Implementation of Other Aspects A full implementation of the deduction-parsing system — complete with encodings of several deduction systems and sample grammars — is available from the first author and from the Computation and Language E-Print Archive (`cmp-lg@xxx.lanl.gov`) as part of paper cmp-lg/9404008. The distributed code covers the following aspects of the implementation that are not elsewhere described.

1. Input and encoding of the string to be parsed.
2. Implementation of the deduction engine driver including generation of consequences.
3. Encoding of the storage of items including the implementation of the chart and agenda.
4. Encoding of deduction systems.
5. Implementation of subsumption checking.

All Prolog code distributed has been tested under the Quintus Prolog system.

5.4. Alternative Implementations

This implementation of agenda and chart provides a compromise in terms of efficiency, simplicity, and generality. Other possibilities will occur to the reader that may have advantages under certain conditions. Some of the alternatives are described in this section.

Separate agenda and chart in database. Storage of the agenda and the chart under separate predicates in the Prolog database allows for marginally more efficient lookup of chart items; an extraneous arithmetic comparison of indices is eliminated. However, this method requires an extra retraction and assertion when moving an index from agenda to chart, and makes redundancy checking more complex in that two separate searches must be engaged in.

Passing agenda as argument. Rather than storing the agenda in the database, the list of agenda items might be passed as an argument. (The implementation of queues in Prolog is straightforward, and would be the natural structure to use for the agenda argument.) This method again has the marginal advantage in antecedent lookup, but it becomes almost impossible to perform efficient redundancy checking relative to items in the agenda.

Efficient bottom-up interpretation. The algorithm just presented can be thought of as a pure bottom-up evaluator for inference rules given as definite clauses, where the head of the clause is the consequent of the rule and the body is the antecedent. However, given appropriate inference rules, the bottom-up procedure will simulate non-bottom-up parsing strategies, such as the top-down and Earley strategies described in Section 3. Researchers in deductive databases have extensively investigated variants of that idea: how to take advantage of the tabulation of results in the pure bottom-up procedure while keeping track of goal-directed constraints on possible answers. As part of these investigations, efficient bottom-up evaluators for logic programs have been designed, for instance CORAL [26]. Clearly, one could use such a system directly as a deduction parser.

Construction of derivations. The direct use of the inference rules for building derivations, as presented in Section 4.1, is computationally inefficient, since it eliminates structure-sharing in the chart. All ways of deriving the same string will yield distinct items, so that sharing of computation of subderivations is no longer possible.

A preferable method is to compute the derivations offline by traversing the chart after parsing is finished. The deduction engine can be easily modified to do so, using a technique reminiscent of that used in the Core Language Engine [2]. First, we make use of two versions of each inference rule, an online version such as the Earley system given in Figure 5, with no computation of derivations, and an offline version like the one in Figure 6 that does generate derivation information. We will presume that these two versions are stored, respectively, under the predicates `inference/4` (as before) and `inference_offline/4`, with the names of the rules specifying the correspondence between related rules. Similarly, the online `initial_item/1` specification should have a corresponding `initial_item_offline/1` version.

The deduction engine parses a string using the online version of the rules, but also stores, along with the chart, information about the ways in which each chart item can be constructed, with unit clauses of the form

```
stored_history(Consequent, Rule, Antecedents).     ,
```

which specify that the item whose index is given by `Consequent` can be generated by the inference rule whose name is `Rule` from the antecedent items given in the sequence `Antecedents`. For each application of `Rule` that generates `Consequent` from the antecedent items `Antecedent`, a clause of this form is asserted to record that possible derivation. Note that only in the first such derivation of `Consequent` will the consequent itself be added to the agenda, but each redundant derivation of `Consequent` must still be recorded to ensure that all possible derivations are represented. (If an item is generated as an initial item, its history would mark the fact by a unit clause using the constant `initial` for the `Rule` argument.)

When parsing has completed, a separate process is applied to each goal item, which traverses these stored histories using the second (offline) version of the inference rules rather than the first, building derivation information in the process. The following Prolog code serves the purpose. It defines `offline_item(Index, Item)`, a predicate that computes the offline item `Item` (presumably including derivation information) corresponding to the online item with index given by `Index`, using the second version of the inference rules, by following the derivations stored in the chart history.

```
offline_item(Index, Item) :-
    stored_history(Index, initial, _NoAntecedents),
    initial_item_offline(Item).
offline_item(Index, Item) :-
    stored_history(Index, Rule, Antecedents),
    offline_items(Antecedents, AntecedentItems)
    inference_offline(Rule, AntecedentItems, Item, SideConds),
    hold(SideConds).

offline_items([], []).
offline_items([Index|Indexes], [Item|Items]) :-
    offline_item(Index, Item),
    offline_items(Indexes, Items).
```

The offline version of the inference rules need not merely compute a derivation. It might perform some other computation dependent on derivation, such as semantic interpretation. Abstractly, this technique allows for staging the parsing into two phases, the second comprising a more fine-grained version of the first. Any staged processing of this sort can be implemented using this technique.

Finer control of execution order For certain applications, it may be necessary to obtain even finer control over the order in which the antecedent items and side conditions are checked when an inference rule is triggered. Given that the predicates `items_in_chart/2` and `holds/1` perform a simple left-to-right checking of the items and side conditions, the implementation of `matching_rule/5` above leads to the remaining antecedent items and side conditions being checked in left-to-right order as they appear in the encoded inference rules, and the side conditions being checked after the antecedent items. However, it may be preferable to interleave the checks for antecedents and for side conditions, perhaps in different orders depending on which antecedent triggered the rule.

For instance, the side condition $A \doteq A'$ in the second inference rule of Section 4.1 must be handled before checking for the nontrigger antecedent of that rule, in order to minimize nondeterminism. If the first antecedent is the trigger, we want to check the side conditions and then look for the second antecedent, and correspondingly for triggering the second antecedent. The implementation above disallows this possibility, as side conditions are always handled after the antecedent items. Merely swapping the order of handling side conditions and antecedent items, although perhaps sufficient for this example, does not provide a general solution to this problem.

Various alternatives are possible to implement a finer level of control. We present an especially brutish solution here, although more elegant solutions are possible. Rather than encoding an inference rule as a single unit clause, we encode it with one clause per trigger element under the predicate

```
inference(RuleName, Antecedents, Consequent)
```

where `Rulename` and `Consequent` are as before, but `Antecedents` is now a list of all the antecedent items and side conditions of the rule, with the trigger item first. (To distinguish antecedent items from side conditions, a disambiguating prefix operator can be used, e.g., `@item(...)` versus `?side_condition(...)`.) Matching an item against a rule then proceeds by looking for the item as the first element of this antecedent list.

```
matching_rule(Trigger, RuleName, Others, Consequent) :-
    inference(RuleName, [Trigger|Others], Consequent),
```

The `consequence/2` predicate is modified to use this new `matching_rule/4` predicate, and to check that all of the antecedent items and side conditions hold.

```
consequence(Index, Consequent) :-
    index_to_item(Index, Trigger),
    matching_rule(Trigger, RuleName, Others, Consequent),
    hold(Others, Index).
```

The antecedent items and side conditions are then checked in the order in which they occur in the encoding of the inference rule.

```
hold([], _Index).
hold([Antecedent|Antecedents], Index) :-
    holds(Antecedent, Index),
    hold(Antecedents, Index).

holds(@Item,      Index) :- item_in_chart(Item, Index).
holds(?SideCond, _Index) :- call(SideCond).
```

6. CONCLUSION

The view of parsing as deduction presented in this paper, which generalizes that of previous work in the area, makes possible a simple method of describing a variety of parsing algorithms — top-down, bottom-up, and mixed — in a way that highlights the relationships among them and abstracts away from incidental differences of control. The method generalizes easily to parsers for augmented phrase structure formalisms, such as definite-clause grammars and other logic grammar formalisms. Although the deduction systems do not specify detailed control structure, the control information needed to turn them into full-fledged parsers is uniform, and can therefore be given by a single deduction engine that performs sound and complete bottom-up interpretation of the rules of inference. The implemented deduction engine that we described has proved useful for rapid prototyping of parsing algorithms for a variety of formalisms including variants of tree-adjoining grammars, categorial grammars, and lexicalized context-free grammars.

ACKNOWLEDGEMENTS

This material is based in part upon work supported by the National Science Foundation under Grant No. IRI-9350192 to SMS and by an associated Xerox Corporation grant. The authors would like to thank the anonymous reviewers for their helpful comments on an earlier draft.

REFERENCES

1. Anthony E. Ades and Mark J. Steedman. On the order of words. *Linguistics and Philosophy*, 4(4):517–558, 1982.
2. Hiyan Alshawi, editor. *The Core Language Engine*. ACL-MIT Press Series in Natural Language Processing. MIT Press, Cambridge, Massachusetts, 1992.
3. François Bancilhon and Raghu Ramakrishnan. An amateur's introduction to recursive query processing strategies. In Michael Stonebraker, editor, *Readings in Database Systems*, section 8.2, pages 507–555. Morgan Kaufmann, San Mateo, California, 1988.
4. Sylvie Billot and Bernard Lang. The structure of shared forests in ambiguous parsing. In *Proceedings of the 27th Annual Meeting of the Association for Computational Linguistics*, pages 143–151, Vancouver, British Columbia, 26-29 June 1989.
5. Joan Bresnan and Ron Kaplan. Lexical-functional grammar: A formal system for grammatical representation. In J. Bresnan, editor, *The Mental Representation of Grammatical Relations*, pages 173–281. MIT Press, 1982.
6. Bob Carpenter. *The Logic of Typed Feature Structures*. Number 32 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, England, 1992.
7. Alain Colmerauer. Metamorphosis grammars. In Leonard Bolc, editor, *Natural Language Communication with Computers*, pages 133–187. Springer-Verlag, 1978. First appeared as "Les Grammaires de Metamorphose", Groupe d'Intelligence Artificielle, Université de Marseille II, November 1975.
8. Jay C. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.
9. Joshua S. Hodas. Specifying filler-gap dependency parsers in a linear-logic programming language. In Krzysztof Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 622–636, Washington, DC, 1992.
10. Aravind K. Joshi. How much context-sensitivity is necessary for characterizing structural descriptions—Tree adjoining grammars. In D. Dowty, L. Karttunen, and A. Zwicky, editors, *Natural Language Processing—Theoretical, Computational and Psychological Perspectives*. Cambridge University Press, New York, 1985.
11. Aravind K. Joshi, L. S. Levy, and M. Takahashi. Tree adjunct grammars. *Journal of Computer and System Sciences*, 10(1):136–163, 1975.

12. T. Kasami. An efficient recognition and syntax algorithm for context-free languages. Technical Report AF-CRL-65-758, Air Force Cambridge Research Laboratory, Bedford, Massachusetts, 1965.
13. Martin Kay. Algorithm schemata and data structures in syntactic processing. In Barbara J. Grosz, Karen Sparck Jones, and Bonnie Lynn Webber, editors, *Readings in Natural Language Processing*, chapter I. 4, pages 35–70. Morgan Kaufmann, Los Altos, California, 1986. Originally published as a Xerox PARC technical report, 1980.
14. Anthony Kroch and Aravind K. Joshi. Linguistic relevance of tree adjoining grammars. Technical Report MS-CIS-85-18, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, Pennsylvania, April 1985.
15. Joachim Lambek. The mathematics of sentence structure. *American Mathematical Monthly*, 65:154–170, 1958.
16. Bernard Lang. Deterministic techniques for efficient non-deterministic parsers. In J. Loeckx, editor, *Proceedings of the 2nd Colloquium on Automata, Languages and Programming*, pages 255–269, Saarbrücken, Germany, 1974. Springer-Verlag.
17. Jean-Louis Lassez, Michael J. Maher, and Kim G. Marriot. Unification revisited. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625, San Mateo, California, 1988. Morgan Kaufmann.
18. Michael Moortgat. *Categorial Investigations: Logical and Linguistic Aspects of the Lambek Calculus*. PhD thesis, University of Amsterdam, Amsterdam, The Netherlands, October 1988.
19. Jeffrey F. Naughton and Raghu Ramakrishnan. Bottom-up evaluation of logic programs. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, chapter 20, pages 641–700. MIT Press, Cambridge, Massachusetts, 1991.
20. Remo Pareschi and Dale A. Miller. Extending definite clause grammars with scoping constructs. In David H. D. Warren and Peter Szeredi, editors, *Seventh International Conference on Logic Programming*, Jerusalem, Israel, 1990. MIT Press.
21. Patrick Paroubek, Yves Schabes, and Aravind K. Joshi. XTAG — a graphical workbench for developing tree-adjoining grammars. In *Proceedings of the Third Conference on Applied Natural Language Processing*, pages 216–223, Trento, Italy, 1992.
22. M. Pentus. Lambek grammars are context free. In *Proceedings of the Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 429–433, Montreal, Canada, 19–23 June 1993. IEEE Computer Society Press.
23. Fernando C. N. Pereira and David H. D. Warren. Definite clause grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13:231–278, 1980.
24. Fernando C. N. Pereira and David H. D. Warren. Parsing as deduction. In *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics*, pages 137–144, Cambridge, Massachusetts, June 15–17 1983.
25. Raghu Ramakrishnan. Magic templates: A spellbinding approach to logic programs. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, pages 140–159, Seattle, Washington, 1988. MIT Press.
26. Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. CORAL: Control, Relations and Logic. In *Procs. of the International Conf. on Very Large Databases*, 1992.
27. William C. Rounds and Alexis Manaster-Ramer. A logical version of functional grammar. In *Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics*, pages 89–96, Stanford, California, 1987. Stanford University.
28. Taisuke Sato and Hisao Tamaki. Enumeration of success patterns in logic programs.

Theoretical Computer Science, 34:227–240, 1984.

29. Yves Schabes. Left to right parsing of lexicalized tree-adjoining grammars. *Computational Intelligence*, 1994. To appear.
30. Yves Schabes and Stuart Shieber. An alternative conception of tree-adjoining derivation. In *Proceedings of the 20th Annual Meeting of the Association for Computational Linguistics*, pages 167–176, 1992.
31. Yves Schabes and Richard C. Waters. Lexicalized context-free grammars. In *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics*, pages 121–129, Columbus, Ohio, June 1993.
32. Yves Schabes and Richard C. Waters. Tree insertion grammar: A cubic-time parsable formalism that strongly lexicalizes context-free grammar. Technical Report 94-13, Mitsubishi Electric Research Laboratories, Cambridge, Massachusetts, 1994.
33. Stuart M. Shieber. Criteria for designing computer facilities for linguistic analysis. *Linguistics*, 23:189–211, 1985.
34. Stuart M. Shieber. Using restriction to extend parsing algorithms for complex-feature-based formalisms. In *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics*, pages 145–152, Chicago, Illinois, 1985. University of Chicago.
35. Stuart M. Shieber. *Constraint-Based Grammar Formalisms*. MIT Press, Cambridge, Massachusetts, 1992.
36. K. Vijay-Shanker. *A Study of Tree Adjoining Grammars*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, Pennsylvania, 1987.
37. K. Vijay-Shanker and David J. Weir. Parsing some constrained grammar formalisms. *Computational Linguistics*, 19(4):591–636, December 1993.
38. D. H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208, 1967.