Controlled Grammatic Ambiguity

MIKKEL THORUP
University of Copenhagen

A new approach to ambiguity of context-free grammars is presented, and within this approach the LL and LR techniques are generalized to solve the following problems for large classes of ambiguous grammars:

—Construction of a parser that accepts all sentences generated by the grammar, and which always terminates in linear time.
—Identification of the structural ambiguity: a finite set of pairs of partial parse trees is constructed; if for each pair the two partial parse trees are semantically equivalent, the ambiguity of the grammar is semantically irrelevant.

The user may control the parser generation so as to get a parser which finds some specific parse trees for the sentences. The generalized LL and LR techniques will still guarantee that the resulting parser accepts all sentences and terminates in linear time on all input.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications; D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.4 [**Programming Languages**]: Processors; F.3.2 [**Logics and Meanings of Programs**]. Semantics of Programming Languages; F.4.2 [**Mathematical Logic and Formal Languages**]· Grammars and Other Rewriting Systems

General Terms: Algorithms, Design, Languages, Theory, Verification

Additional Key Words and Phrases: Grammatic ambiguity, semantic unambiguity

1. INTRODUCTION

For unambiguous grammars we have the powerful LL and LR techniques that for large classes can verify their unambiguity and construct complete linear-time parsers, i.e., parsers that accept the full languages and terminate in linear time on all inputs. With a new approach to ambiguity we generalize

Most of this work was done while the author was at Oxford University and DIMACS supported, mainly, by a NATO grant from the Danish Research Council, partly, by the Danish Research Academy.
Author's address: University of Copenhagen, Department of Computer Science, Universitetsparken 1, 2100 København Ø, Denmark; email: mthorup@diku.dk.

the LL and LR techniques to deal with large classes of ambiguous grammars as well, thus characterizing the ambiguity and constructing complete linear-time parsers.

Ambiguous grammars have long been considered relevant in connection with programming languages, for as noticed by Aho et al. [1975] ambiguous grammars are often simpler and more natural than their unambiguous counterparts. Moreover, they can often be parsed more efficiently due to their having smaller parse trees. In Aho et al., and independently in Earley [1975], the now standard approach was developed for construction of linear-time parsers for ambiguous grammars. The approach starts by applying an LL or LR technique. Since the grammar is ambiguous the resulting parser will be nondeterministic due to conflicts in the action table. A deterministic LL or LR parser is then obtained by resolving these conflicts, possibly in a semiautomatic way using some disambiguating rules. As stated in Aho et al., the problem with this approach is that the resolved parser might not be complete. In fact, as noticed in Soisalon-Soininen and Tarhio [1988], it might not even terminate on some input.

A simple example of a problematic grammar is

$$L \to LL,\ L \to \mathtt{1},\ L \to \mathtt{111}. \tag{1}$$

The idea behind the last production is to reduce the number of productions in large parse trees roughly by a factor of five. Applying any LR(1) technique to the above grammar we get the following nondeterministic LR(1) parser:

STATE	*ACTION*		*GOTO*
	1	$	*L*
0	s 3	÷	1
1	s 3	a	2
2	s 3, r $L \to LL$	r $L \to LL$	2
3	r $L \to$ 1, s 4	r $L \to$ 1	÷
4	s 5	÷	÷
5	r $L \to$ 111	r $L \to$ 111	÷

Suppose we resolve the conflicts using the heuristic from the classic parser generator Yacc [Johnson 1975] of choosing shifts over reductions. When presented with a sentence like 111, the resolved parser will make two shifts arriving at state 4 with input $, and then it is stuck. Thus the resulting parser is not complete.

In this article we present some very general techniques for resolving the conflicts in a nondeterministic LL or LR parser so that the resulting deterministic parser is guaranteed to be complete and work in linear time. Even for our problematic grammar (1), our techniques will find such a "good" LR(1) parser. For example this could be the parser $\mathscr{P}_{(1)}$ that for each entry selects the first action. An alternative good resolution is if instead of shifting in state 2 on input `1`, we choose to reduce $L \to LL$. Despite the fact that the grammar (1) is left-recursive, it is possible for us to find a good LL parser for it, but a lookahead of length 2 is needed. Given the nondeterministic parser constructed by the canonical LL(2) technique for the grammar (1), our techniques can find a good resolution which, in fact, selects exactly the same parse tree as our LR(1) parser $\mathscr{P}_{(1)}$.

The basic idea for testing if a resolved parser is good is to see if there is a set of simple parse tree rewriting rules that rewrite *any* parse tree into a parse tree for the same sentence but found by the parser. Trivially, the existence of such rewriting rules implies that the resolved LL or LR parser is complete. Moreover it turns out that in this case the resolved parser will always terminate in linear time. The simple rewriting rules are pairs (u, v) of partial parse trees where u and v have the same root and the same frontier. A parse tree t is rewritten by replacing an occurrence of v in t with u; if v does not occur in t then (u, v) does not rewrite t. Loosely speaking, we will show that it is decidable if there exists a finite set of rewriting rules whose embedding in the original nondeterministic parser implies that they can rewrite any parse tree into a parse tree for the same sentence found by the parser. In this case the parser is not only complete, it always terminates in linear time.

Continuing our example (1), in connection with the resolved parser $\mathscr{P}_{(1)}$, our techniques will construct the following set of rewriting rules:

$$\mathcal{E}_{(1)}: \left\{ \left(\begin{array}{ll} L & \\ |\backslash & \\ LL & L \\ |\,|\backslash & |\backslash\backslash \\ 1LL & 111 \\ \;|\,| & \\ \;11 & \end{array} \right), \left(\begin{array}{ll} L & L \\ |\backslash & |\,\backslash \\ LL & L\;L \\ \;|\backslash & \;|\backslash \\ \;LL & \;LL \end{array} \right) \right\}.$$

Using these rewriting rules, the techniques will prove that $\mathscr{P}_{(1)}$ is complete and works in linear time.

It should be mentioned that other more specialized techniques have been presented for recognition of resolved LL or LR parsers that are complete and work in linear time. Aho et al. [1975] found a simple combinatorial test which

works for classes of LL parsers, and Demers [1974] contains a test based on merging nonterminals and removing trivial productions. Neither of these techniques apply to our grammar (1).

By our test for good resolutions we have generalized the LL and LR techniques to construct not only complete linear-time parsers for large classes of ambiguous grammars, but also finite sets of rewriting rules characterizing the structural ambiguity. When Knuth [1965] presented the canonical LR technique, he described it as the most general-known technique for testing unambiguity of grammars. Similarly, our techniques are the most general known for characterizing finitely generated ambiguity.

An obvious application is as follows. Suppose we want to use an ambiguous grammar in the semantic definition of a formal language, as, for example, a programming language or a specification language. If we do not want to refer to some specific parser, we need to ensure that the ambiguity of the grammar is semantically irrelevant, i.e., that all parse trees for the same sentence are semantically equivalent. For example, we need to avoid ambiguity based on a dangling else while ambiguity based on algebraic identities like associativity is perfectly okay. So far, proving that the ambiguity of an ambiguous grammar is semantically irrelevant has been done by hand [Blikle 1989]. However, with the above rewriting rules, we only need to verify for each of them that the two partial parse trees are semantically equivalent. Then the rewriting preserves semantics, and hence all the parse trees for any sentence must be semantically equivalent. The advantages of semantically irrelevant ambiguity are not only theoretical. Semantically irrelevant ambiguity exhibits real freedom in parsing. This freedom is not exploited in this article, where only the construction of deterministic parsers is considered. However, in the technical report Thorup [1992], based on the results in this article, a theory is developed for the construction of nondeterministic complete linear-time parsers. The parsing freedom of these parsers can be used, for example, in connection with incremental parsing and evaluation.

The set of rewriting rules provides an exact description of the "canonical" parse trees selected by the parser for the sentences. First, the canonical parse trees are exactly the parse trees that cannot be rewritten, i.e., the set of parse trees that for no rewriting rule (u, v) contains an occurrences of v. Second, and more constructively, given any parse tree for a sentence, the corresponding canonical parse tree can be found by repeated rewriting with the rewriting rules. Thus from $\mathscr{E}_{(1)}$ we can read both that $\mathscr{P}_{(1)}$ is right-associative and that it does not use $L \to \texttt{111}$.

Our techniques allow the user to state his "priorities" by supplying his own rewriting rules, thereby controlling the parser generation toward any desired set of canonical parse trees. As always, our techniques guarantee that the generated parser is complete and works in linear time.

When comparing our ambiguity resolution based on priorities with the traditional disambiguating rules facilitated by Yacc [Aho et al. 1986; Johnson 1975], one difference is, of course, the already mentioned guarantees of

completeness and termination given by our techniques. On the other hand, there are grammars for which our techniques report failure despite the existence of good resolutions. This would never happen to Yacc, for Yacc takes *no* responsibility for the generated parsers. Another difference is that our rewriting rules are independent of the actual parsing technique. For example, our set $\mathscr{E}_{(1)}$ can be used both with the LR(1) and the LL(2) techniques. This independence makes our rewriting rules resemble more ambiguity resolutions like those given in the syntactic metalanguage SDF [Heering et al. 1990]. There ambiguity resolution is achieved by discarding parse trees that are nonminimal with respect to a fixed priority ordering on parse trees parameterized, not by rewriting rules, but by a user-supplied partial ordering of the productions. However, SDF does not guarantee that there is a unique minimal parse. Also, besides the ordering of the productions, there is a mechanism in SDF that allows certain parses to be disallowed outright ("priority conflicts"). Consequently, the disambiguated "parser" may be both incomplete and nondeterministic. Moreover, there are only exponential bounds on its complexity.

This article is basically theoretical. The classes of grammars we can deal with will be defined mathematically, but only future experience can tell how well they cover practical applications. The examples are all very simple, just showing that our techniques have enriched the class of grammars that can be handled automatically with several nice ambiguous constructs. For each of these examples it would be easy to identify by hand the ambiguity and construct a complete linear-time parser, and, in fact, the same might be the case for any single naturally occurring ambiguity. The problem in dealing with ambiguous grammars does not lie in the deliberate use of ambiguous constructs by the designer of the grammar. Rather, the problem is to ensure that there is no hidden ambiguity. In real life, the problem could occur if, say, a grammar was so big that it had to be written in parts by several different authors. Even if the compiler experts claim that they can deal with all grammars occurring in practice, the problem is still relevant for more dynamic systems like OBJ [Futatsugi et al. 1985] where compiler tyros may define their own languages. The problem is solved by the LL and LR techniques for large classes of unambiguous grammars, and with our generalization it is solved for large classes of ambiguous grammars as well.

For conciseness, the article will focus around the canonical LR(1) technique [Knuth 1965] which is theoretically the most beautiful of the LL and LR techniques. It is, however, not difficult to translate our results to any of the other techniques. Moreover, we will only touch the semantic aspects peripherally.

A key to our handling of ambiguity is to generalize the focus from sentences to derived productions (if X is a grammar symbol deriving a string α of grammar symbols, then $X \rightarrow^* \alpha$ is a derived production). Now, the article is divided as follows: Section 2 describes the basic notion of grammars in terms of derived productions, and Section 3 reviews parsing, modified to deal with derived productions. After these two preliminary sections, Section 4 formal-

izes the idea of rewriting rules rewriting all parse trees into the canonical parse trees found by a parser. Section 5 contains the exact statement of the main result. This includes a precise definition of what is meant by embedding rewriting into the finite nondeterministic parsers constructed by the LL and LR techniques. Section 6 discusses the relation between traditional disambiguating rules and priorities, and Section 7 discusses the problems with a dangling else. Section 8 gives a general outline of the generalizations of the LL and LR techniques. Moreover, it gives an overview of the appendices of this article. These appendices contain the details of the generalization of the canonical LR(1) technique.

The article is self-contained, but the preliminary definitions are rather dense, so the reader is referred to Aho et al. [1986] or Sippu [1988] and Sippu and Soisalon-Soininen [1990] for a standard text on grammars and parsing. The article is a shortened version of Thorup [1994, Part I], to which the reader is referred for more examples and discussions.

2. GRAMMARS IN TERMS OF DERIVED PRODUCTIONS

As was first noticed algebraically in Blikle and Torup [1990] and Blikle et al. [1991] a key to working formally with the ambiguity of grammars is to focus on "derived productions" which are natural generalizations of sentential forms (which in turn are generalizations of sentences). Below, we will review the basic concept of context-free grammars, but in terms of derived productions.

A *grammar* $\mathcal{G}$ consists of a finite set V of symbols together with a finite set P of pairs (X, α) where X belongs to V and α is a, possibly empty, string of symbols from V. We refer to the symbols in V as *grammar symbols* and to the pairs in P as *productions*. A production (X, α) is written $X \to \alpha$. We refer to X as the *left side* and to α the *right side* of the production. The grammar symbols are divided into *terminals* and *nonterminals*. One of the latter is distinguished as the *start symbol*. A grammar is well formed only if all the left-side symbols of the productions are nonterminals.

We will always understand an underlying grammar when we talk about grammar symbols, productions, etc. Moreover, we adopt the following notational convention from Aho et al. [1986]: A, B, C stand for nonterminals, X, Y, Z for grammar symbols, and α, β, γ for possibly empty strings of grammar symbols. The symbol ϵ is generally reserved to denote an empty string. Thus, for $p = 0$ we have $X_1 \ldots X_p = \epsilon$.

The following is a simple example of a grammar:

$$S \to A,\ A \to B,\ A \to BC,\ B \to \mathtt{b},\ C \to \mathtt{c},\ C \to \epsilon \tag{2}$$

As in all later examples, only the productions of a grammar are given explicitly. The grammar symbols are understood to be all symbols used in the productions; the start symbol is the left side of the first production; and the terminals are the grammar symbols in `typewriter` font. Thus in the gram-

mar (2) we have nonterminals S, A, B, C with S the start symbol, and we have terminals b, c.

Definition 2.1. *Derived productions* are pairs (X, α) where X is a grammar symbol, and α is a string of grammar symbols. A derived production (X, α) is written $X \to^* \alpha$. The set of derived productions is defined recursively as follows:

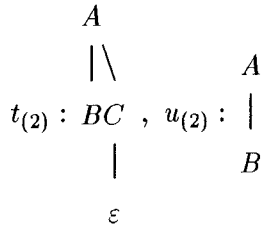
—If X is a grammar symbol then $X \to^* X$ is a derived production.
—If $X \to^* \alpha Y \gamma$ is a derived production and $Y \to \beta$ is a production then $X \to^* \alpha\beta\gamma$ is a derived production.

For example, we have that $A \to^* \mathrm{b}C$ is a derived production for grammar (2). Notice that for derived productions, it is not required that the left side is a nonterminal (as for productions, the left side and the right side refer to the first and the second coordinate, respectively). This dropping of the distinction between terminals and nonterminals turns out to be very convenient for the following definitions, algorithms, and proofs. Besides having been used algebraically in Blikle and Thorup [1990] and Blikle et al. [1991], the concept of derived productions has been used in Ballance et al. [1988] in connection with "grammatical abstraction." Often we will ignore the notational difference between derived productions and productions when writing statements like "all productions are derived productions" where formally we should have written something like "for all productions $X \to \alpha$, we have that $X \to^* \alpha$ is a derived production."

In terms of derived productions, a *sentential form* is a string α of grammar symbols such that with S denoting the start symbol, we have that $S \to^* \alpha$ is a derived production. A *sentence* is then a sentential form consisting of terminals only. Thus, as claimed, derived productions are more general than both sentential forms and sentences.

As with sentences, we are interested in the way a derived production is generated from its recursive definition. This is done in terms of *parse trees*. Parse trees for derived productions form the base for all reasoning in the remainder of this article. They are therefore introduced with more care and terminology than usual. A parse tree is an ordered rooted tree where each node is labeled either with a grammar symbol or with ϵ. Since our parse tree is ordered, the sons of a nonleaf node n are given as an ordered sequence. Nodes labeled ϵ play a special role, allowing us to have internal nodes with an empty sequences of sons. More precisely, only a leaf may be labeled ϵ, and only if it is the single son of some node. If n has a single son labeled ϵ, we interpret this as its sequence of sons being empty. Concerning the general labeling, it is required that if an internal node labeled X has its sons labeled $X_1, \ldots, X_p$ in this order, then $X \to X_1 \ldots X_p$ is a production. By the *root symbol* of a parse tree we refer to the label of the root, and by the *frontier* we refer to the sequence of labels of the non-ϵ leaves. Parse trees are depicted with the root in the top and the sons ordered from the left to the right. Thus, for grammar (2) we have the following parse trees with root symbol A and

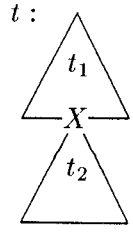
frontier B:



Clearly, $X \rightarrow^* \alpha$ is a derived production if and only if there is a parse tree t with root symbol X and frontier α. In this case, we say that t is a *parse tree for* $X \rightarrow^* \alpha$, or that $X \rightarrow^* \alpha$ is the *derived production generated by* t. For example, $t_{(2)}$ and $u_{(2)}$ both generate $A \rightarrow^* B$. Correspondingly, parse trees for sentential forms are parse trees where the root symbol is the start symbol, and parse trees for sentences are parse trees where, moreover, the frontier consists of terminals.

If we have two parse trees, like $t_{(2)}$ and $u_{(2)}$, generating the same derived production, we say that they are *equivalent*, and then the underlying grammar is said to be *ambiguous*. Thus grammar (2) is ambiguous. Traditionally, one only says that a grammar is ambiguous if the grammar has two parse trees for the same sentence. Clearly, our grammar (2) is ambiguous also with respect to this notion of ambiguity. Notice, however, that the traditional notion of ambiguity is strictly weaker, for it is possible for a grammar to be ambiguous with respect to derived productions but not with respect to sentences if there are grammar symbols that are not used in a parse tree for any sentence.

The study of parse trees plays a major role in this article, so we need some precise definitions of a few more parse tree-related concepts. A parse tree is said to be *trivial* if it contains only one node. A *subparse tree* relation on parse trees is defined to be the least partial ordering, i.e., the least reflexive and transitive relation, with the following property: let t be a parse tree and n be a non-ϵ node of t. Moreover, let t_1 be the subtree of t excluding all nodes strictly descending from n, and let t_2 be the subtree of t containing all nodes descending from n including n itself. Then t_1 and t_2 are subparse trees of t. The construction is illustrated below with X being the label of the common node n of t_1 and t_2.



Notice, that if n is a leaf of t, then t_2 is trivial; and then $t_1 = t$. Similarly $t_2 = t$ if n is the root of t. With t, n, t_1, and t_2 as above, we say that t is obtained by *rooting t_2 in the leaf n of t_1*.

It is important to notice that the subparse tree relation, in contrast to the normal subtree relation, respects the production structure of parse trees. For example, $u_{(2)}$ is a subtree but not a subparse tree of $t_{(2)}$. According to the definition of the subparse tree relation, we say a parse tree t is *minimal* in a set T of parse trees if t, but no proper subparse tree of t, is in T. Hence, for example, if T is the set of all parse trees, then the minimal parse trees in T are the trivial parse trees.

Finally, it is convenient to introduce a constructor $\underline{\text{pt}}$ for parse trees. If X is a grammar symbol, by $\underline{\text{pt}}(X)$ we denote the trivial parse tree whose single node is labeled by X. Now, let $X \to X_1 \cdots X_p$ be a production and $t_1, \ldots, t_p$ be parse trees with root symbols $X_1, \ldots, X_p$. Then by $\underline{\text{pt}}(X; t_1, \ldots, t_p)$ we denote the parse tree with root symbol X and with $t_1, \ldots, t_p$ being the subparse trees descending from the sons of the root. Thus, continuing our example with grammar (2), we have $t_{(2)} = \underline{\text{pt}}(A; \underline{\text{pt}}(B), \underline{\text{pt}}(C; \epsilon))$ and $u_{(2)} = \underline{\text{pt}}(A; \underline{\text{pt}}(B))$.

3. PARSERS

In this section we review the concepts of parsing, but modified to deal with derived productions. The modification itself is straightforward, using ideas from incremental parsing and error recovery [Ghezzi and Mandrioli 1979; Pennello and DeRemer 1979]. However, knowing the exact way the modification is done is crucial to the understanding of the rest of the article. Thus *parsing* is the process of recognizing derived productions (not just sentences) and selecting parse trees that generate them.

A *parser* is an automaton with takes as input an *input production* meaning a pair (X, α), sometimes written $X \to^{\circ} \alpha$, where X is a grammar symbol and α is a sequence of grammar symbols. There are now three possibilities:

(1) The parser successfully accepts the input production as a derived production, and returns some parse tree generating $X \to^* \alpha$.
(2) The parser terminates unsuccessfully. This might be the case even if the input production is a derived production.
(3) The parser does not even terminate.

A parser is a *linear-time* parser if it terminates for all inputs in time linear in the length of the input production. Moreover, a parser is *complete* if it successfully finds a canonical parse tree for each derived production. We always want parsers to be complete linear-time parsers, and we refer to such parsers as *implementations* of a grammar.

Given some parser $\mathscr{P}$, we will refer to the parse trees it can return successfully as *$\mathscr{P}$-canonical* parse trees, possibly dropping the "$\mathscr{P}$" if some

specific parser is understood. Thus completeness of $\mathscr{P}$ means that any derived production is generated by some $\mathscr{P}$-canonical parse tree.

The traditional LL and LR parsers can easily be modified to parse derived productions instead of just sentences. In this article we focus on LR(1) parsers. Below, we review the definition of LR(1) parsers modified for derived productions.

An LR(1) parser $\mathscr{P}$ for a grammar is characterized by the following components:

—A finite set of states.

—A table *INIT* from grammar symbols to states.

—A partial table *GOTO* from states and grammar symbols to states.

—An action table *ACTION* from states and input symbols to actions.

Here, by the *input symbols* we mean all grammar symbols plus an extra symbol \$, and by *actions* we mean members of the set

$$\{\text{accept}, \text{shift}, \text{error}\} \cup \{\text{reduce } X \to \alpha \mid X \to \alpha \text{ is a production}\}.$$

Notice, that relative to traditional LR(1) parsers for sentences we have introduced the table *INIT* to take care of the left side which is no longer fixed to some specific start symbol. Moreover, the tables *GOTO* and *ACTION* are now defined for all grammar symbols regardless of whether they are terminals or nonterminals.

A parsing with $\mathscr{P}$ proceeds through a series of configurations of the form

$$\left(\boxed{I_0} X_1 \boxed{I_1} \cdots X_m \boxed{I_m}, X_{m+1} \cdots X_n\right)$$

where $X_1, \ldots, X_{n-1}$ are grammar symbols, where $X_n = \$$, and where $I_0, \ldots, I_m$ are states. The first component of the configuration is called the *stack*, and the second component is called the *input buffer*. The last state, I_m, in the stack is referred to as *the state*, and first input symbol, X_{m+1}, in the input buffer is referred to as *the input symbol*. By the *action* we refer to $ACTION(I_m, X_{m+1})$, i.e., the entry in the action table given by the state and the input symbol.

Let $X \to^? \alpha$ be the input production. Then the initial configuration or configuration 0 is

$$\left(\boxed{INIT(X)}, \alpha\$\right).$$

The next configurations are determined by the action of the current configuration. The effects of the different actions are as follows:

—*Accept* is only valid if the current configuration is of the form

$$\left(\boxed{INIT(X)} X \boxed{GOTO(INIT(X), X)}, \$\right).$$

The parsing is then completed successfully, accepting the input production as a derived production. The parse tree for the input production constructed by the parsing is described later.

—*Shift* is only valid if the current configuration is of the form

$$\left(\alpha \boxed{I}, X\beta\right),$$

where X is a grammar symbol and $GOTO(I, X)$ is defined. The next configuration becomes $\left(\alpha \boxed{I} X \boxed{GOTO(I, X)} \beta\right)$.

—Reduce $X \to X_1 \cdots X_p$ is only valid if the current configuration is of the form

$$\left(\alpha \boxed{I} X_1 \boxed{I_1} \cdots X_p \boxed{I_p}, \beta\right)$$

with $GOTO(I, X)$ defined. Then the subsequent configuration becomes $\left(\alpha \boxed{I} X \boxed{GOTO(I, X)}, \beta\right)$.

—*Error* simply means that we terminate unsuccessfully, giving up the input production.

The parser $\mathscr{P}$ is only well formed if there is no input production for which it will ever try an invalid action. Notice that if in the parsing of an input production with left side X, we encounter a configuration with state I and the sequence $X_1, \ldots, X_m$ of grammar symbols in the stack, then $I = GOTO(\cdots GOTO(INIT(X), X_1) \cdots X_m)$.

In order to define the parse tree constructed by an accepting parsing, we associate a parse tree with each grammar symbol in the stack. If the grammar symbol is X, the parse tree will always have X as root symbol. Recall that in the initial configuration we have no grammar symbols in the stack. When the parsing shifts a symbol X from the input buffer to the stack, we associate $\underline{pt}(X)$ with X. The interesting case is when the parser reduces some production $X \to X_1 \cdots X_p$. Before the reduction, we have $X_1, \ldots, X_p$ as a final segment of the sequence of grammar symbols in the stack, and the reduction will replace them by X. If $t_1, \ldots, t_p$ are the parse trees associated with $X_1, \ldots, X_p$ before the reduction, then after the reduction the parse tree $\underline{pt}(X; t_1, \ldots, t_p)$ is associated with X. When a parsing accepts, it returns the parse tree associated with the single grammar symbol on the stack.

The definition of LR(1) parsers for derived productions is illustrated by an example based on the grammar

$$B \to B * B,\ B \to 0,\ B \to 1. \tag{3}$$

Consider the following LR(1) parser for the grammar in (3): (a, s, and r stand for accept, shift, and reduce, respectively).

$\mathcal{P}_{(3)}$:

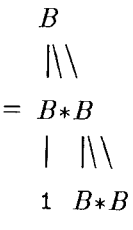
INIT	STATE	*ACTION*					*GOTO*			
		B	$*$	0	1	\$	B	$*$	0	1
B	1	s	÷	s	s	÷	2	÷	5	6
	2	÷	s	÷	÷	a	÷	3	÷	÷
	3	s	÷	s	s	÷	4	÷	5	6
	4	÷	s	÷	÷	r $B\rightarrow B*B$	÷	3	÷	÷
	5	÷	r $B\rightarrow$0	÷	÷	r $B\rightarrow$0	÷	÷	÷	÷
	6	÷	r $B\rightarrow$1	÷	÷	r $B\rightarrow$1	÷	÷	÷	÷
$*$	7	÷	s	÷	÷	÷	÷	8	÷	÷
	8	÷	÷	÷	÷	a	÷	÷	÷	÷
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Given the input production $B \rightarrow^{?} 1*B*B$ it will carry out the following parsing:

STACK	INP-BUF	ACTION
[1]	$1*B*B\$$	s
[1] $1_{\underline{\text{pt}}(1)}$ [6]	$*B*B\$$	r $B\rightarrow$1
[1] $B_{\underline{\text{pt}}(B;\underline{\text{pt}}(1))}$ [2]	$*B*B\$$	s
[1] $B_{\underline{\text{pt}}(B;\underline{\text{pt}}(1))}$ [2] $*_{\underline{\text{pt}}(*)}$ [3]	$B*B\$$	s
[1] $B_{\underline{\text{pt}}(B;\underline{\text{pt}}(1))}$ [2] $*_{\underline{\text{pt}}(*)}$ [3] $B_{\underline{\text{pt}}(B)}$ [4]	$*B\$$	s
[1] $B_{\underline{\text{pt}}(B;\underline{\text{pt}}(1))}$ [2] $*_{\underline{\text{pt}}(*)}$ [3] $B_{\underline{\text{pt}}(B)}$ [4] $*_{\underline{\text{pt}}(*)}$ [3]	$B\$$	s
[1] $B_{\underline{\text{pt}}(B;\underline{\text{pt}}(1))}$ [2] $*_{\underline{\text{pt}}(*)}$ [3] $B_{\underline{\text{pt}}(B)}$ [4] $*_{\underline{\text{pt}}(*)}$ [3] $B_{\underline{\text{pt}}(B)}$ [4]	\$	r $B\rightarrow B*B$
[1] $B_{\underline{\text{pt}}(B;\underline{\text{pt}}(1))}$ [2] $*_{\underline{\text{pt}}(*)}$ [3] $B_{\underline{\text{pt}}(B;\underline{\text{pt}}(B),\underline{\text{pt}}(*),\underline{\text{pt}}(B))}$ [4]	\$	r $B\rightarrow B*B$
[1] $B_{\underline{\text{pt}}(B;\underline{\text{pt}}(B;\underline{\text{pt}}(1)),\underline{\text{pt}}(*),\underline{\text{pt}}(B;\underline{\text{pt}}(B),\underline{\text{pt}}(*),\underline{\text{pt}}(B)))}$ [2]	\$	a

Thus the following parse tree is the $\mathcal{P}_{(3)}$-canonical parse tree for $B \rightarrow^{*} 1*B*B$:

$$\underline{\text{pt}}(B;\underline{\text{pt}}(B;\underline{\text{pt}}(1)),\underline{\text{pt}}(*),\underline{\text{pt}}(B;\underline{\text{pt}}(B),\underline{\text{pt}}(*),\underline{\text{pt}}(B))) =$$

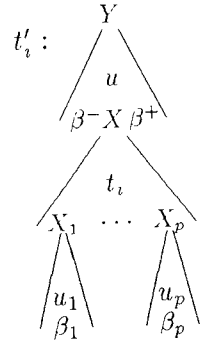


The traditional techniques for construction of LL and LR parsers only work for classes of unambiguous grammars. However, in Section 8 and the appendices they will be generalized so that they construct complete linear-time LL and LR parsers for large classes of ambiguous grammars as well.

Notice that our generalization of parsers to work for derived productions rather than just for sentences rightly can be viewed as a simplification. For example, Knuth [1965] showed that if a grammar is unambiguous and all grammar symbols are used in parse trees for sentences, then a complete linear-time LR(1) parser for sentences can be constructed if it exists. In the context of derived production, the statement simplifies to: if a grammar is unambiguous, then a complete linear-time LR(1) parser for derived productions can be constructed if it exists. In other words we get rid of the "no-junk" requirement in the original formulation.

4. PRIORITIZED PARSING

We are now done with the preliminary sections, ready to introduce "prioritized parsing" which is a key concept for our automatic verification of the completeness of parsers. It formalizes the rewritings discussed in the introduction. A *parse tree pair* is an ordered pair of equivalent parse trees, i.e., parse trees with the same root and the same frontier. In general, a parse tree pair (t_0', t_1') follows by *substitution* from a parse tree pair (t_0, t_1) if t_0' results from t_1' by replacing with t_0 an occurrence of t_1 in t_1'. Formally, this is the case if and only if there exists numbers k, p, and parse trees $u, u_1, \ldots, u_p$ such that for $i = 0, 1$, the parse tree t_i' results from t_i by first for $i = 1, \ldots, p$, rooting u_i in the ith non-ϵ leaf of t_i, and subsequently rooting the obtained parse tree in the kth non-ϵ leaf of u. The figure below illustrates the construction. In the figure, $X \rightarrow^* X_1 \cdots X_p$ is the derived production generated by t_0 and t_1, and $Y \rightarrow^* \beta^- \beta_1 \cdots \beta_p \beta^+$ is the derived production generated by t_0' and t_1'. Given a set $\mathscr{E}$ of parse tree pairs, by $\mathscr{E}^\prec$ we denote the closure of $\mathscr{E}$ under substitution and transitivity.



Definition 4.1. A set $\mathscr{E}$ of parse tree pairs is *prioritizing* if $\mathscr{E}^\prec$ is a well-founded (no infinite descending sequence, i.e., no infinite sequence $t_0 \cdots$

$t_k \cdots$ of parse trees with $(t_{i+1}, t_i) \in \mathscr{E}^{\prec}$ for all i) strict partial ordering, and no two $\mathscr{E}^{\prec}$-minimal parse trees are equivalent. If, moreover, $\mathscr{P}$ is a parser and the $\mathscr{P}$-canonical parse trees coincide with the $\mathscr{E}^{\prec}$-minimal parse trees, then $\mathscr{E}$ *prioritizes* $\mathscr{P}$.

Thus, if we say that a parse tree t has higher priority than a parse tree u whenever $(t, u) \in \mathscr{E}^{\prec}$, then the $\mathscr{P}$-canonical parse trees are exactly those with the highest priority. The term is chosen because later the user will be allowed to manipulate the set $\mathscr{E}$, thereby showing his priorities for the parser $\mathscr{P}$.

PROPOSITION 4.1. *If a set $\mathscr{E}$ of parse tree pairs prioritizes a parser $\mathscr{P}$, then $\mathscr{P}$ is complete.*

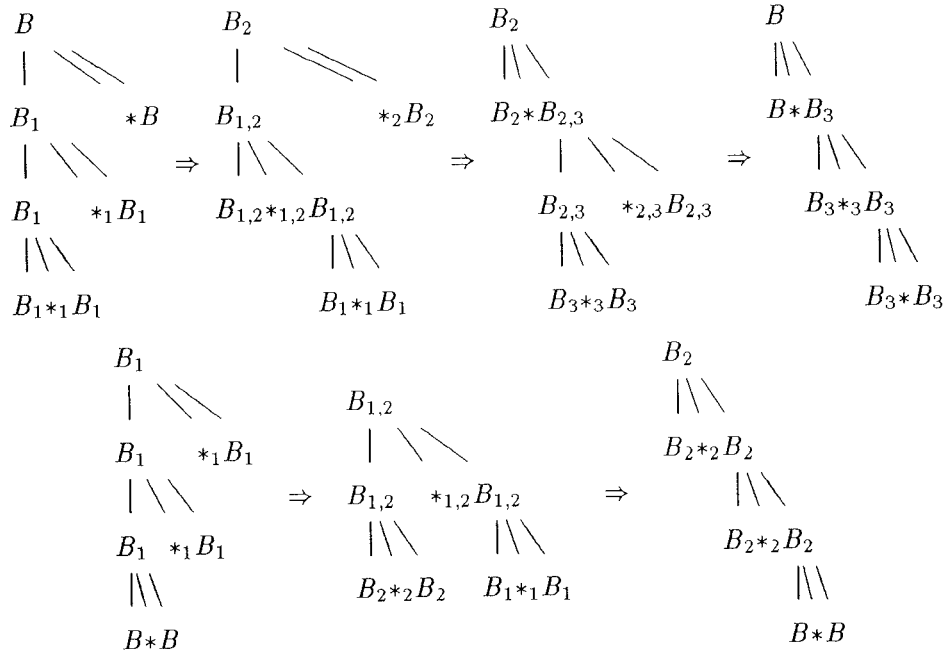
PROOF. Let $X \to^* \alpha$ be a derived production. By definition, $X \to^* \alpha$ is generated by at least one parse tree u. Since $\mathscr{E}^{\prec}$ is well founded, there is an $\mathscr{E}^{\prec}$-minimal, hence $\mathscr{P}$-canonical, parse tree t with $(t, u) \in \mathscr{E}^{\prec}$. But $\mathscr{E}^{\prec}$ only relates equivalent parse trees, so t is a $\mathscr{P}$-canonical parse tree for $X \to {}^*\alpha$. □

Sometimes, we will think about the definition of prioritized parsers in terms of rewriting. We say that the parse tree pair (t, u) *rewrites* the parse tree u' to the parse tree t' if (t', u') follows from (t, u) by substitution. Notice, that u' is not uniquely defined in terms of t, u, and t', for u might occur at different positions in u'. Now, let $\mathscr{E}$ be a set of parse tree pairs prioritizing a parser $\mathscr{P}$. Then, the definition of prioritizing says that given any parse tree, if, as long as possible, we rewrite with the parse tree pairs from $\mathscr{E}$, then eventually we will arrive at a $\mathscr{P}$-canonical equivalent to t. At the moment it might seem most natural to think of "prioritizing" in this way. However, our more algebraic description in terms of orderings forms a better base for the rest of the article where, typically, we will be interested in expansions of $\mathscr{E}^{\prec}$ to orderings that are not expressible in terms of rewriting.

The techniques that will be presented in Section 8 and the appendices can find prioritizing sets for large classes of grammars. For the grammar (3), they could return

$$\mathcal{E}_{(3)} : \left\{ \left(\begin{array}{ccc} B & & B \\ /\backslash & & /\backslash \\ B*B & , & B \quad *B \\ /\backslash & & /\backslash \\ B*B & & B*B \end{array} \right) \right\}$$

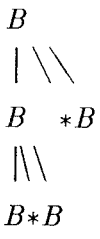
which prioritizes the parser $\mathscr{P}_{(3)}$ that we found in Section 3. Below are shown two different maximal rewriting sequences with $\mathscr{E}_{(3)}$, starting in the same parse tree (the indices indicate the subparse trees being replaced).



Notice that the number of rewritings needed to transform a parse tree into its canonical equivalent is not unique.

Observe that any prioritizing set characterizes the ambiguity completely. Suppose, for example, that we want to show that the ambiguity of a grammar is irrelevant with respect to some semantics. If a prioritizing set $\mathscr{E}$ is given, it is sufficient to check for each parse tree pair in $\mathscr{E}$ that the two parse trees are semantically equivalent, i.e., that rewriting with the pair preserves semantics. This follows, for consider any two equivalent parse trees t and u. By the definition of prioritizing, they can both be rewritten into their common $\mathscr{P}$-canonical equivalent v. Hence they are both semantically equivalent to v, and hence to each other. In connection with our grammar (3), we have the prioritizing set $\mathscr{E}_{(3)}$, so its ambiguity is irrelevant with respect to some semantics if and only if these semantics associate an associative operation with the production $E \to E * E$. This characterization of the ambiguity of a grammar is somewhat related to the work in Tomita [1986], Rekers [1991], and Billot and Lang [1989]. There they operate with the concepts of "shared forests" and "local ambiguity," both of which are subsumed by the concept of substitution. With $u, u_1, \ldots, u_p$ as in the definition of substitution, shared forests corresponds to the case where u is trivial, i.e., where $u = \underline{\mathrm{pt}}(X)$, and local ambiguity corresponds to the case where $u_1, \ldots, u_p$ are trivial. In Tomita [1986], Rekers [1991], and Billot and Lang [1989] these concepts are used for a compact (cubic) representation of the different parse trees for a given sentence while our prioritizing sets once and for all characterize the ambiguity for the whole grammar. On the other hand, our techniques only work for restricted classes of grammars, for which finite prioritizing sets can be found.

If a set of parse trees pairs prioritizes a parser, it implies that the set of canonical parse trees is exactly the set of parse trees that does not have subparse trees among the second coordinates in the prioritizing set. Thus, from the fact that $\mathscr{E}_{(3)}$ prioritizes $\mathscr{P}_{(3)}$, it follows that the set of $\mathscr{P}_{(3)}$-canonical parse trees is exactly the set of parse trees not containing



as a subparse tree. Notice that such a concise characterization of right-associative parsing has not appeared previously in literature! However, a somewhat related characterization has been proposed independently by W. Maddox (personal communication, 1993). Our characterization is only possible due to our change in focus from sentences to derived productions.

By *prioritizing the parsing* of a grammar we mean: finding a finite set $\mathscr{E}$ of parse tree pairs and a parser $\mathscr{P}$ such that $\mathscr{E}$ prioritizes $\mathscr{P}$. Thus by prioritizing the parsing of a grammar we both identify the ambiguity and characterize a complete parser. In the next section we will state precisely to what extent the techniques to be presented in Section 8 and the appendices can help us prioritize the parsing of grammars. In Section 7, we shall see that there are grammars, like those containing a dangling else, which have no finite prioritizing set. It is not claimed that our techniques can deal successfully with all natural grammars. The LL and LR techniques deal with limited but important classes of unambiguous grammars, i.e., grammars with empty prioritizing sets of parse tree pairs. The result of this article is that these techniques can be generalized to deal safely with correspondingly limited but important classes of ambiguous grammars with finite prioritizing sets of parse tree pairs. The relevance of these new classes is indicated simply by showing some concrete natural ambiguous grammars, like the one in (3), with which they can deal.

5. THE MAIN RESULT

We assume that the LL and LR techniques have been modified to work with derived productions instead of sentences. We will focus on the modified version of the canonical LR(1) technique [Knuth 1965], henceforth referred to as the *CLR*(1) technique (we change name from "canonical LR" to "CLR" not as much for brevity, but as to avoid confusion stemming from overloading of the term "canonical"). The CLR(1) technique is formally described in Appendix B.

Given a grammar, the traditional LL and LR techniques (modified to deal with derived production) will try to construct a complete linear-time LL or LR parser. Unfortunately they can never be successful if the grammar is ambigu-

ous. The techniques will always succeed in generating a generally nondeterministic LL or LR parser, and they are successful exactly when this parser is deterministic. Thus, failure shows up as conflicts in the generated action table, i.e., as entries containing more than one action. The generated nondeterministic parser always has the property that, with an appropriate parsing, it can construct any parse tree for any sentence. To emphasize this property we will refer to the nondeterministic parsers generated by the LL and LR techniques as *universal parsers*. Notice that universality implies completeness in the case where the universal parser is deterministic.

Consider, for example, the grammar

$$E \to F,\ F \to E,\ F \to FF. \tag{4}$$

Such a grammar could be relevant in connection with change in representation, i.e., if we wanted to use equations like $x \times y = \exp(\ln x + \ln y)$ or $x \wedge y = \neg(\neg x \vee \neg y)$. However, it is chosen here because of its richness on ambiguity. Applying the CLR(1) technique, we get the following nondeterministic LR(1) parser:

$\mathcal{U}_{(4)}$:

INIT	STATE	*ACTIONS*			*GOTO*	
		E	*F*	$	*E*	*F*
E	1	s	s	÷	2	3
	2	r $F \to E$	r $F \to E$	a, r $F \to E$	÷	÷
	3	s, r $E \to F$	s, r $E \to F$	r $E \to F$	4	5
	4	r $F \to E$	r $F \to E$	r $F \to E$	÷	÷
	5	s, r $F \to FF$, r $E \to F$	s, r $F \to FF$, r $E \to F$	r $F \to FF$, r $E \to F$	4	5
F	6	s	s	÷	4	7
	7	s, r $E \to F$	s, r $E \to F$	a, r $E \to F$	4	5

By a *determinization* of a universal LL or LR parser $\mathcal{U}$ we mean a deterministic LL or LR parser $\mathcal{P}$ obtained by resolving the conflicts in the action table, i.e., for each entry e in the action table of $\mathcal{U}$, we select a *canonical* action $\mathcal{P}_e \in \mathcal{U}_e$, where $\mathcal{U}_e$ denotes the set of actions in the entry e of the action table of $\mathcal{U}$. For our example (4), we let $\mathcal{P}_{(4)}$ denote the determinization of $\mathcal{U}_{(4)}$ derived by selecting the first action in each entry. Then $\mathcal{P}_{(4)}$ is a complete linear-time parser. The universal parsers generated by most of the LL and LR techniques, including the LL(k), the LALR(k), and the CLR(k) technique, satisfy the no-junk property that all actions in each entry of the action table are used in the parsing corresponding to some parse tree. Hence, whenever we resolve a conflict in the action table, we exclude some parse trees. Thus, if

for an unambiguous grammar, we get conflicts in the action table, there is no hope of finding a complete determinization.

Let $\mathscr{U}$ be a universal LL or LR parser. Moreover, let (t, u) be a parse tree pair with $t \neq u$. Consider the unique parsings constructing t and u. Let c be the first configuration from which the two parsings carry out different actions a and b, respectively. Let e be the entry in the action table corresponding to c; then $a, b \in \mathscr{U}_e$. Now, the triple (e, a, b) is called the *projection* on $\mathscr{U}$ of the parse tree pair (t, u). For parse tree pairs with two identical parse trees, the projection is undefined, denoted $\perp$. Take for example, the following parse tree pair for grammar (4):

$$a_{(4)}: \left(\begin{array}{ccc} F & & F \\ |\diagdown & & |\diagdown \\ F & F & FF \\ |\diagdown & | & , \quad | \; |\diagdown \\ FFE & & EFF \\ & | & | \\ & F & F \end{array} \right).$$

The parse trees in $a_{(4)}$ are constructed by following parsings with $\mathscr{U}_{(4)}$:

Stack	Buffer	Action	Stack	Buffer	Action
$\boxed{6}$	$FFF\$$	s	$\boxed{6}$	$FFF\$$	s
$\boxed{6}F\boxed{7}$	$FF\$$	s	$\boxed{6}F\boxed{7}$	$FF\$$	r $E \rightarrow F$
$\boxed{6}F\boxed{7}F\boxed{5}$	$F\$$	r $F \rightarrow FF$	$\boxed{6}E\boxed{4}$	$FF\$$	r $F \rightarrow E$
$\boxed{6}F\boxed{7}$	$F\$$	s	$\boxed{6}F\boxed{7}$	$FF\$$	s
$\boxed{6}F\boxed{7}F\boxed{5}$	$\$$	r $E \rightarrow F$	$\boxed{6}F\boxed{7}F\boxed{5}$	$F\$$	s
$\boxed{6}F\boxed{7}E\boxed{4}$	$\$$	r $F \rightarrow E$	$\boxed{6}F\boxed{7}F\boxed{5}F\boxed{5}$	$\$$	r $F \rightarrow FF$
$\boxed{6}F\boxed{7}F\boxed{5}$	$\$$	r $F \rightarrow FF$	$\boxed{6}F\boxed{7}F\boxed{5}$	$\$$	r $F \rightarrow FF$
$\boxed{6}F\boxed{7}$	$\$$	a	$\boxed{6}F\boxed{7}$	$\$$	a

Thus, the projection of $a_{(4)}$ in $\mathscr{U}_{(4)}$ is $((7, F)$, shift, reduce $E \rightarrow F))$.

An ordering $\mathscr{O}$ of a universal LL or LR parser $\mathscr{U}$ is represented by associating with each entry e in the action table a strict partial ordering $\mathscr{O}_e$ of $\mathscr{U}_e$. Thereby $\mathscr{O}$ defines a strict partial ordering on parse trees. Fix two equivalent but different parse trees t and u, and let (e, a, b) be the projection of (t, u) in $\mathscr{U}$. Then $(t, u) \in \mathscr{O}$ if and only if $(a, b) \in \mathscr{O}_e$. Notice that because it is required that all the $\mathscr{O}_e$s are strict partial orderings, the definition guarantees that in fact $\mathscr{O}$ is a strict partial ordering. Also notice that if $\mathscr{E}$ is a set of parse tree pairs and there is an ordering of $\mathscr{U}$ that contains $\mathscr{E}$, then there is a unique least such ordering $\mathscr{O}$, namely, the "entry-wise" transitive closure of

the projection of $\mathscr{E}$ in $\mathscr{U}$. In this case we say that $\mathscr{O}$ is the universal parser ordering *spanned* by $\mathscr{E}$; otherwise $\mathscr{E}$ does not span a universal parser ordering.

Given a universal LL or LR parser $\mathscr{U}$ with an ordering $\mathscr{O}$ and a determination $\mathscr{P}$, we say that $\mathscr{P}$ is *minimal* in $\mathscr{O}$ if for each entry e we have that $\mathscr{P}_e$ is minimal in $\mathscr{O}_e$. Thus, if $\mathscr{P}$ is minimal in $\mathscr{O}$, then all $\mathscr{P}$-canonical parse trees are $\mathscr{O}$-minimal.

Definition 5.1. Let $\mathscr{E}$ be a set of parse tree pairs, $\mathscr{U}$ be a universal LL or LR parser, and $\mathscr{P}$ be a determinization of $\mathscr{U}$. Then $\mathscr{E}$ *$\mathscr{U}$-prioritizes* $\mathscr{P}$ if $\mathscr{E}$ prioritizes $\mathscr{P}$, and $\mathscr{E}^{\prec}$ spans an ordering of $\mathscr{U}$ in which $\mathscr{P}$ is minimal.

This definition is extremely important for the following. Generally, it says that given a set $\mathscr{E}$ of parse tree pairs prioritizing a parser $\mathscr{P}$, then $\mathscr{E}$ $\mathscr{U}$-prioritizes $\mathscr{P}$ if the finite universal parser $\mathscr{U}$ can "see" that the rewritings never loop and that no canonical parse tree can be written. Continuing our example with grammar (4), set

$$\mathcal{E}_{(4)} = \left\{ \begin{pmatrix} & E \\ & | \\ E\ , & F \\ & | \\ & E \end{pmatrix} , \begin{pmatrix} & F \\ & | \\ F\ , & E \\ & | \\ & F \end{pmatrix} , \begin{pmatrix} F & F \\ |\backslash & |\backslash \\ FF & F\ F \\ |\backslash & |\backslash \\ FF & FF \end{pmatrix} \right\}.$$

Moreover, denote by $\mathscr{O}_{(4)}$ the ordering of $\mathscr{U}_{(4)}$ obtained by ordering the actions in the entries of the action table of $\mathscr{U}_{(4)}$ in the order that they are already listed. Notice, that the pair $a_{(4)}$ is in $\mathscr{O}_{(4)}$ but not in $\mathscr{E}_{(4)}^{\prec}$. The techniques from Section 8 and the appendices can ascertain that $\mathscr{E}_{(4)}$ prioritizes $\mathscr{P}_{(4)}$ and that $\mathscr{E}_{(4)}$ spans $\mathscr{O}_{(4)}$. Moreover, it is clear that $\mathscr{P}_{(4)}$ is minimal in $\mathscr{O}_{(4)}$. Thus it follows that $\mathscr{E}_{(4)}$ $\mathscr{U}_{(4)}$-prioritizes $\mathscr{P}_{(4)}$. Also the techniques can verify that $\mathscr{P}_{(4)}$ works in linear time, so $\mathscr{P}_{(4)}$ is a complete linear-time parser. Finding such a parser for grammar (4) is not completely trivial. Suppose, for example, we apply Warton's heuristic [Wharton 1976] for conflict resolution to $\mathscr{U}_{(4)}$. It prefers reductions over shifts, so in state $7 = GOTO(INIT(F), F))$ on input E, it will choose to reduce $E \to F$ instead of shifting. Hence the resulting parser will never terminate for an input production like $F \to^{?} FF$, which in fact is a derived production.

THEOREM 5.1. *Given a grammar $\mathscr{G}$, a finite set $\mathscr{E}_{in}$ of parse tree pairs, and the universal parser $\mathscr{U}$ obtained by any one of the various LL or LR techniques, we can decide if there exists a determination $\mathscr{P}$ of $\mathscr{U}$, and a finite set $\mathscr{E}_{out}$ of parse tree pairs such that $(\mathscr{E}_{in} \cup \mathscr{E}_{out})$ $\mathscr{U}$-prioritizes $\mathscr{P}$. In this case we can find such a $\mathscr{P}$ and $\mathscr{E}_{out}$, where $\mathscr{P}$ is a linear-time parser, and $\mathscr{E}_{out}$ is minimal given $\mathscr{P}$.*

The role of $\mathscr{E}_{in}$ is to allow the user to impose his own priorities on the parser construction. Continuing our example with grammar (4), if we set $\mathscr{E}_{in}$

to be the set of the first two parse tree pairs in $\mathscr{E}_{(4)}$, we specify that the generated parser should be "cycle free," but leave it open whether it should be left- or right-associative.

Admittedly, in order to keep down the size of the presentation, we will only prove Theorem 5.1 in detail for the CLR(1) technique. This is done constructively in Section 8 together with the appendices. In the following, when we talk unspecified about *our technique*, it is understood that we are talking about the generalization of the CLR(1) technique.

6. DISAMBIGUATING WITH PRIORITIES

In this section we will discuss in more detail how the priorities from $\mathscr{E}_{in}$ in Theorem 5.1 can be used like traditional rewriting rules. The discussion will be based on an example stemming from Aho et al. [1986, pp. 251–254], and which is taken from a real-world grammar for the equation-typesetting language EQN [Kernighan and Cherry 1975]. We consider the following grammar.

$$E \to E\,\mathtt{sub}\,E\,\mathtt{sup}\,E,\ E \to E\,\mathtt{sub}\,E,\ E \to E\,\mathtt{sup}\,E,\ E \to \{E\},\ E \to \mathtt{c} \qquad (5)$$

Our aim is to construct a (complete linear-time) parser that uses the "special-case" production $E \to E\,\mathtt{sub}\,E\,\mathtt{sup}\,E$ in connection with any substring `c sub c sup c` of the input, but in no other cases. This is specified by setting the following:

$$\mathcal{E}_{in} = \left\{ \left(\begin{array}{ll} E & E \\ E & \\ E\,\mathtt{sub}\,E\,\mathtt{sup}\,E & E \quad \mathtt{sup}\,E \\ & E\,\mathtt{sub}\,E \end{array} \right), \left(\begin{array}{ll} E & E \\ E & \\ E\,\mathtt{sub}\,E\,\mathtt{sup}\,E & E\,\mathtt{sub}\,E \\ & E\,\mathtt{sup}\,E \end{array} \right), \right.$$

$$\left(\begin{array}{ll} E & \\ *,\ E\,\mathtt{sub}\,E & \mathtt{sup}\,E \\ E\,\mathtt{sub}\,E & \end{array} \right), \left(\begin{array}{ll} E & \\ *,\ E\,\mathtt{sub}\,E & \mathtt{sup}\,E \\ E\,\mathtt{sup}\,E & \end{array} \right),$$

$$\left. \left(\begin{array}{ll} E & \\ *,\ E\,\mathtt{sub}\,E & \mathtt{sup}\,E \\ E\,\mathtt{sup}\,E\,\mathtt{sup}\,E & \end{array} \right) \right\}$$

The last three pairs are $*$-*priorities*. The idea behind them is to set the technique free in choosing the first coordinates. Hence the effect of a $*$-priority is to exclude its second coordinates from the canonical parse trees. Our technique will successfully find a complete linear-time parser prioritized by some superset of $\mathscr{E}_{ln}$ (with some adequate first coordinates in place of the $*$s). By looking at the priorities returned together with the parser, we can see exactly how the technique resolved the various associativity and precedence questions.

With Yacc [Johnson 1975], which is based on the LALR(1) technique, it is in some sense easier to construct a parser for grammar (5) satisfying the specification. Yacc has a general heuristic which prefers shifts over reductions—the very same heuristic that led to an incomplete parser for grammar (1) in the introduction. Thus, all we need to get a correct parser is to add a disambiguating rule saying that we prefer to reduce with $E \to E\,\mathtt{sub}\,E\,\mathtt{sup}\,E$ rather than with $E \to E\,\mathtt{sup}\,E$. Technically this is told to Yacc, by listing the production $E \to E\,\mathtt{sub}\,E\,\mathtt{sup}\,E$ first in the declaration of the grammar—as we already did.

It seems that one needs to be quite familiar with Yacc and the LR(1) techniques, in order to feel comfortable with the above conflict resolution. A more serious complaint is if there is a risk that somebody else could have worked on the grammar, adding some disambiguating rules. Suppose, for example, that somebody had worked on the grammar before the special-case production $E \to E\,\mathtt{sub}\,E\,\mathtt{sup}\,E$ was introduced. It would have made perfect sense for her/him to decide that both `sub` and `sup` should be left-associative and that they should be of the same precedence. Technically this is told to Yacc via the declaration `%left 'sub' 'sup'`. Such a declaration overrules the heuristic about preferring shifts over reductions. Without going into details, when applied to the full grammar (5), this declaration implies that any conflict between shifting and reducing will be settled in favor of reducing. This does not contradict our preference of reducing with $E \to E\,\mathtt{sub}\,E\,\mathtt{sup}\,E$ rather than with $E \to E\,\mathtt{sup}\,E$, but it implies that we will never get to a state from which it is possible to reduce $E \to E\,\mathtt{sub}\,E\,\mathtt{sup}\,E$. Consequently, our special-case production will never be used.

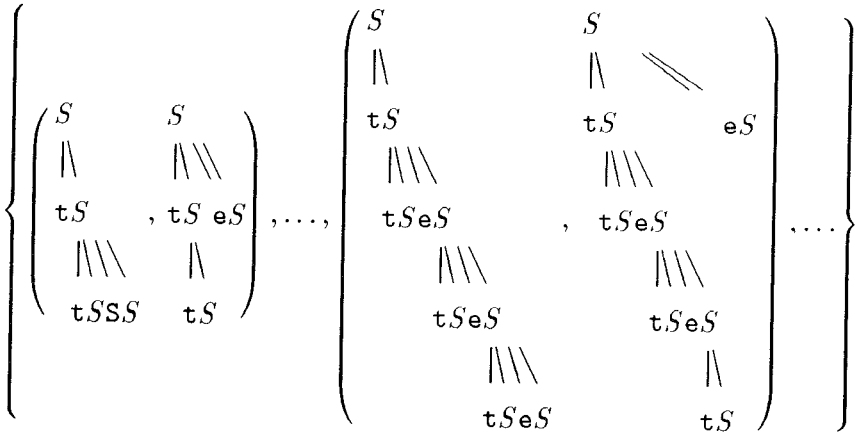
7. THE DANGLING ELSE

Unfortunately there are grammars for which our approach cannot be successful. The most prominent example is the dangling-else construction which is commonly used despite it being semantically problematic (in order to get the correct semantics of a sentence with a dangling else, one applies the ad hoc rule that the "else" belongs to the nearest preceding "then"). The following grammar represents the classic dangling-else construction:

$$S \to \mathtt{t}\,S\,\mathtt{e}\,S,\ S \to \mathtt{t}\,S,\ S \to \mathtt{s}. \tag{6}$$

Our techniques cannot handle this grammar, for it can be shown that it has no finite prioritizing set. However, our techniques can offer to start listing the

following infinite prioritizing set of parse tree pairs:



The universal CLR(1) parser for grammar (6) contains only one conflict, and it turns out that *any* resolution of this conflict gives a complete linear-time parser (choosing shifting over reducing gives the parser with the intended semantics). Unfortunately, our techniques cannot verify this completeness due the lack of a finite prioritizing set. Hence, if we want the safety of our techniques, we have to disambiguate grammar (6). Using the technique from Thorup [1994], this can be done automatically, the result being the grammar

$$S \to S_1,\ S \to S_2,\ S_1 \to \mathtt{t}\,S,\ S_1 \to \mathtt{t}\,S_2\,\mathtt{e}\,S_1,\ S_2 \to \mathtt{t}\,S_2\,\mathtt{e}\,S_2,\ S_2 \to \mathtt{s}.$$

An even better solution would be to avoid the problem completely by changing the language to something more readable, as is done in Modula-2 [Wirth 1985].

Of course, the lack of safety is not a problem in connection with a single small grammar like (6). As indicated in the last section, the problem arises if we start using grammars like (6) in a larger context, say, where several language designers are collaborating on the same grammar, or just with a large grammar where checking against hidden ambiguity and incompleteness of the generated parsers is cumbersome by hand.

In Appendix J we shall return to the possibility of generalizing our techniques to deal directly with a dangling-else construction.

8. ALGORITHMIC OUTLINE

In this section we will give a general outline of the computation described in Theorem 5.1. Also, we will give an introduction to the appendices which, relative to the CLR(1) technique, contain all the details of the computation. Initially, we assume that we have none of the $*$-priorities that we discussed in Section 6. Thus, we are given a grammar $\mathscr{G}$, some universal LL or LR parser $\mathscr{U}$ for $\mathscr{G}$, an a finite set $\mathscr{E}_{in}$ of parse tree pairs. Our goal is to find a determinization $\mathscr{P}$ of $\mathscr{U}$, and a finite set $\mathscr{E}_{out}$ of parse tree pairs such that $\mathscr{P}$ and $\mathscr{E}_{out}$ *match* each other in the sense that $(\mathscr{E}_{in} \cup \mathscr{E}_{out})$ $\mathscr{U}$-prioritizes $\mathscr{P}$. If such a matching pair exists we want $\mathscr{E}_{out}$ to be minimal given $\mathscr{P}$, and $\mathscr{P}$ to

be a linear-time parser. The latter condition will be shown always to be satisfied when $\mathscr{P}$ is matched. If there is no such matching pair, we want to be able to report this.

Our key to searching matching pairs is to construct first a catalyst set $\mathscr{R}$ of parse tree pairs defined as follows. The second coordinates in $\mathscr{R}$ are the minimal parse trees among the nontrivial parse trees having the root symbol as a single symbol in the frontier. Let t be any second coordinate in $\mathscr{R}$, and let $X \rightarrow^* X$ be the derived production generated by t, then the corresponding first coordinate is $\underset{\sim}{\text{pt}}(X)$. For grammar (3) from Section 3, the set $\mathscr{R}$ is empty, but for grammar (4) from Section 4, we have $\mathscr{R}$ equal to

$$\mathcal{R}_{(4)}: \left\{ \left(\begin{array}{ccc} & E & \\ & | & \\ E, & F & \\ & | & \\ & E & \end{array} \right), \left(\begin{array}{ccc} & F & \\ & | & \\ F, & E & \\ & | & \\ & F & \end{array} \right) \right\}.$$

It is straightforward to see that $\mathscr{R}$ is always finite, and to construct $\mathscr{R}$. The very special properties of $\mathscr{R}$ are described in the following theorem, the proof of which is deferred to the appendices:

THEOREM 8.1. *Let $\mathscr{P}$ be a determinization of a universal LL or LR parser $\mathscr{U}$, and let $\mathscr{E}$ be a set of parse tree pairs. Then $\mathscr{E}$ $\mathscr{U}$-prioritizes $\mathscr{P}$ if and only if the following conditions are satisfied*:

—*All $\mathscr{E}^{\prec}$-minimal parse trees are $\mathscr{P}$-canonical.*
—*$(\mathscr{E} \cup \mathscr{R})^{\prec}$ spans an ordering of $\mathscr{U}$ in which $\mathscr{P}$ is minimal.*

From the theorem it follows that a determinization $\mathscr{P}$ of our universal parser $\mathscr{U}$ and a finite set $\mathscr{E}_{out}$ of parse tree pairs match each other if and only if the following conditions are satisfied:

(i) All $(\mathscr{E}_{in} \cup \mathscr{E}_{out})^{\prec}$-minimal parse trees are $\mathscr{P}$-canonical.
(ii) $(\mathscr{E}_{in} \cup \mathscr{E}_{out} \cup \mathscr{R})^{\prec}$ spans an ordering of $\mathscr{U}$ in which $\mathscr{P}$ is minimal.

Notice that we have no explicit test for the well-foundedness of $(\mathscr{E}_{in} \cup \mathscr{E}_{out})^{\prec}$ which is required for $\mathscr{E}_{in} \cup \mathscr{E}_{out}$ to be prioritizing. From $(\mathscr{E}_{in} \cup \mathscr{E}_{out} \cup \mathscr{R})^{\prec}$ spanning a universal parser ordering it follows directly that $(\mathscr{E}_{in} \cup \mathscr{E}_{out} \cup \mathscr{R})^{\prec}$ is irreflexive. The strength of Theorem 8.1 is that this irreflexivety implies the desired well-foundedness of $(\mathscr{E}_{in} \cup \mathscr{E}_{out})^{\prec}$.

Trivially (ii) can only be satisfied if $\mathscr{P}$ is minimal in an ordering of $\mathscr{U}$ spanned by $(\mathscr{E}_{in} \cup \mathscr{R})^{\prec}$. This restriction on the number of relevant determinizations is crucial for the running time. Generally this number is exponential, but if the user works modularly, introducing only a slight new ambiguity in each step, then the set $\mathscr{E}_{in}$ will largely determine the parsing, keeping the number of relevant determinizations small. Thus we check that $(\mathscr{E}_{in} \cup \mathscr{R})^{\prec}$ spans an ordering $\mathscr{O}$ of $\mathscr{U}$. If not, we report that there cannot be a matching pair.

Now, choose some $\mathscr{O}$-minimal determinization $\mathscr{P}$. We are going to decide if $\mathscr{P}$ is matched by a set $\mathscr{E}_{out}$, and, if so, construct a minimal such set to be returned together with $\mathscr{P}$. If not, we will have to try another $\mathscr{O}$-minimal determinization. If no $\mathscr{O}$-minimal determinization $\mathscr{P}$ is matched, we will have to report that there is no matching pair.

Denote by M the set of minimal noncanonical parse trees. By definition, if $(\mathscr{E}_{in} \cup \mathscr{E}_{out})$ prioritizes $\mathscr{P}$, the $(\mathscr{E}_{in} \cup \mathscr{E}_{out})$-minimal parse trees coincide with the $\mathscr{P}$-canonical parse trees, but this can only be the case if all parse trees from M are second coordinates in either $\mathscr{E}_{in}$ or $\mathscr{E}_{out}$. Suppose M is infinite. Since $\mathscr{E}_{in}$ is finite, there cannot be a finite set $\mathscr{E}_{out}$ matching $\mathscr{P}$. Thus, if M is infinite, we have to try another $\mathscr{O}$-minimal determinization.

Denote by M^- the set M without the parse trees occurring as second coordinates in $\mathscr{E}_{in}$. Check that each parse tree in M^- has a $\mathscr{P}$-canonical equivalent. If not, $\mathscr{P}$ cannot be complete, so we have to try another $\mathscr{O}$-minimal determinization.

Construct $\mathscr{E}_{out}$ such that M^- coincides with the second coordinates, and such that the first coordinates are their $\mathscr{P}$-canonical equivalents. Clearly we thereby satisfy (i). Also it is clear that if (ii) is satisfied, then $\mathscr{E}_{out}$ is a minimal set matching $\mathscr{P}$, since all the second coordinates are forced. We claim that (ii) is satisfied if $\mathscr{P}$ is matched at all. Suppose that some set $\mathscr{E}'_{out}$ matches $\mathscr{P}$. Then $\mathscr{E}_{in} \cup \mathscr{E}'_{out}$ prioritizes $\mathscr{P}$, so any parse tree pair with a canonical first coordinate is contained in $(\mathscr{E}_{in} \cup \mathscr{E}'_{out})^{\prec}$. Hence, in particular, we have $\mathscr{E}_{out} \subseteq (\mathscr{E}_{in} \cup \mathscr{E}'_{out})^{\prec}$. Thus, from the fact that $\mathscr{E}'_{out}$ satisfies the condition that $(\mathscr{E}_{in} \cup \mathscr{E}'_{out} \cup \mathscr{R})^{\prec}$ spans an ordering of $\mathscr{U}$ in which $\mathscr{P}$ is minimal, we may conclude also that $\mathscr{E}_{out}$ satisfies the condition. Hence (ii) is satisfied, as claimed. Thus, our final step in the processing of $\mathscr{P}$ is to check if (ii) is satisfied. If so, we return $\mathscr{P}$ and $\mathscr{E}_{out}$ successfully; otherwise, $\mathscr{P}$ cannot be matched, so we must try another $\mathscr{O}$-minimal determinization.

The above outlines an algorithm for the computation described in Theorem 5.1. In case our input contained some $*$-priorities, all we need to do is to avoid determinizations in which some second coordinate is canonical. Then as first coordinates, in place of the $*$s, we can just use their canonical equivalents. Unfortunately, $*$-priorities do not speed up the search for an adequate determinization as much as did the normal priorities. Essentially, the problem is that we do not know in advance which is the first noncanonical action in a given second coordinate.

Our algorithmic outline assumes that we can solve the following problems:

(1) Given a finite set $\mathscr{E}$ of parse tree pairs how do we check that $\mathscr{E}^{\prec}$ spans an ordering of $\mathscr{U}$? If it does, how do we construct this ordering? The problem is to construct the projections in $\mathscr{U}$ of the possibly infinite set of parse tree pairs in $\mathscr{E}^{\prec}$.

(2) Given a determinization $\mathscr{P}$, how do we construct the set M of minimal noncanonical parse trees which constituted the necessary second coordinates in $\mathscr{E}_{in} \cup \mathscr{E}_{out}$? In particular, how do we decide if M is infinite, implying that $\mathscr{P}$ cannot be matched?

(3) Given a determinization $\mathscr{P}$ and a parse tree t, how do we check that there is a $\mathscr{P}$-canonical equivalent to t? The problem is that $\mathscr{P}$ might not be terminating for all input.
(4) How do we prove that all matched determinizations are linear-time parsers? The essential problem is that they might be nonterminating.

All these problems are sorted out in the appendices relative to the CLR(1) technique (Knuth's canonical LR(1) technique modified to deal with derived productions). Not all of them are solved exactly as stated. In connection with problems 2 and 3 we will not give an exact solution if for some other reason we can infer that $\mathscr{P}$ cannot be prioritized.

Now, the appendices are divided as follows. Appendix A proves Theorem 8.1. In Appendix B we formally describe the construction of universal CLR(1) parsers, and in Appendix C we introduce some convenient notions concerning such parsers. With the framework setup, Appendix D solves problem 1; Appendix E addresses problem 2; Appendix F addresses problem 3; and Appendix G deals with problem 4. Appendix H puts everything together in an exact algorithm for the computation described in Theorem 5.1 for the CLR(1) technique. The algorithm is followed by some examples in Appendix I. Finally, Appendix J contains some technical remarks about efficiency and possible generalizations.

9. CONCLUSION

The theoretical result of this article is that the various LL and LR techniques can be generalized to deal with classes of ambiguous grammars, characterizing their ambiguity and generating complete linear-time parsers.

The practical result is the guaranteed completeness and correctness of the generated parsers with respect to explicitly declared user intent (the input priorities). Unintentional ambiguity arising from grammar evolution or the combination of reused grammar fragments originally developed in other contexts will not slip by unnoticed. This contrasts the traditional approach based on disambiguating rules which do not in general give warnings against incompleteness or incorrectness.

Unfortunately there are interesting grammars like those based on a dangling else for which the techniques presented in this article will report failure. In that connection, it would be a significant improvement of our techniques to generalize them to deal with parsers prioritized by no finite but by infinite sets of parse tree pairs.

ACKNOWLEDGMENTS

Special thanks to William Maddox, Mark-Jan Nederhof, and Eljas Soisalon-Soininen, all for having put a very large effort into helping with the presentation of this work. Many others have been helpful with suggestions, comments, and encouragement during the years of my work on the theory and algorithms just presented. Among these are Andrew Appel, Dines Bjørner, Andrezej Blikle, Alan Demers, Martin Farach, Tony Hoare, Neil Jones, Bernard

Lang, Bill McColl, Colin McDiarmid, Hans Rischel, Søren Riis, Mary Ryan, Barbara Ryder, Vincent Sgro, Anders Thorup, and various anonymous referees.

APPENDIX

An appendix to this article is available in electronic form (Postscript™). Any of the following methods may be used to obtain it; or see the inside back cover of a current issue for up-to-date instructions.

—By anonymous ftp from acm.org, file [pubs.journals.toplas.append]p1267.ps

—Send electronic mail to mailserve@acm.org containing the line

send [anonymous.pubs.journals.toplas.append]p1267.ps

—By *Gopher* from acm.org

—By anonymous ftp from ftp.cs.princeton.edu, file pub/toplas/append/p1267.ps

—Hardcopy from *Article Express*, for a fee: phone 800-238-3458, fax 201-216-8526, or write P.O. Box 1801, Hoboken NJ 07030; and request ACM-TOPLAS-APPENDIX-1267.

REFERENCES

AHO, A., JOHNSON, S., AND ULLMAN, J. 1975. Deterministic parsing of ambiguous grammars. *Commun. ACM 18*, 8, 441–452.

AHO, A., SETHI, R., AND ULLMAN, J. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass.

BALLANCE, R., BUTCHER, J., AND GRAHAM, S. 1988. Grammatical abstraction and incremental syntax analysis in a language-based editor. *SIGPLAN Not. 23*, 7, 185–198.

BILLOT, S. AND LANG, B. 1989. The structure of shared forests in ambiguous parsing. In *Proceedings of the 27th Annual Meeting of the Association for Computational Linguistics.* ACL, 143–151.

BLIKE, A. 1989. Denotational engineering. *Sci. Comput. Program. 12*, 3, 207–253.

BLIKLE, A. AND THORUP, M. 1990. On conservative extensions of syntax in the process of system development. In *Proceedings of the VDM'90 VDM and Z—Formal Methods in Software Development.* Lecture Notes in Computer Science, vol. 428. Springer, New York, 504–525.

BLIKLE, A., TARLECKI, A., AND THORUP, M. 1991. On conservative extensions of syntax in system development. In *Images of Programming.* North Holland, Amsterdam, 209–235.

DEMERS, A. 1974. Skeletal LR parsing. In *IEEE Conference Records of the 15th Annual Symposium on Switching and Automata Theory.* IEEE, New York, 185–198.

EARLEY, J. 1975. Ambiguity and precedence in syntax description. *Acta Informatica 4*, 2, 183–192.

FUTATSUGI, K., GOGUEN, J., JOUANNAUD, J., AND MESEGUER, J. 1985. Principles of OBJ2. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages.* ACM, New York, 52–66.

GHEZZI, C. AND MANDRIOLI, D. 1979. Incremental parsing. *ACM Trans. Program. Lang. Syst. 1*, 1, 58–71.

HEERING, J., HENDRIKS, P., KLINT, P., AND REKERS, J. 1990. Incremental generation of parsers. *IEEE Trans. Softw. Eng. 16*, 12, 1344–1351.

JOHNSON, S. 1975. Yacc—yet another compiler compiler. Tech. Rep. CS-32, AT & T Bell Laboratories, Murray Hill, N.J.

KERNIGHAN, B. AND CHERRY, J. 1975. A system for typesetting mathematics. *Commun. ACM 18*, 3, 151–157.

KNUTH, D. 1965. On translation of languages from left to right. *Inf. Contr. 8*, 6, 607–639.

KRUSKAL, J. 1960. Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture. *Trans. Am. Math. Soc. 95*, 210–225.

NASH-WILLIAMS, C. 1963. On well-quasi-ordering finite trees. *Proc. Camb. Phil. Soc. 59*, 833–835.

PENNELLO, T. AND DEREMER, F. 1979. A forward move algorithm for LR error recovery. In *Proceedings of the 5th ACM Symposium on Principles of Programming Languages*. ACM, New York, 241–254.

REKERS, J. 1991. Generalized LR parsing for general context-free grammars. Tech. Rep. CS R9153, Centrum voor Wiskunde en Informatica, Mathematisch Centrum, Amsterdam.

SIPPU, S. 1988. *Parsing Theory*. Vol. 1. *Languages and Parsing*. EATCS Monographs on Theoretical Computer Science, vol. 15. Springer-Verlag, Berlin.

SIPPU, S. AND SOISALON-SOININEN, E. 1990. *Parsing Theory*. Vol 2. *LR(k) and LL(k) Parsing*. EATCS Monographs on Theoretical Computer Science, vol. 20. Springer-Verlag, Berlin.

SOISALON-SOININEN, E. AND TARHIO, J. 1988. Looping LR parsers. *Inf. Process. Lett. 26*, 5, 251–253.

THORUP, M. 1994. Disambiguating grammars by exclusion of sub-parse trees. Tech. Rep. 94/11, Dept. of Computer Science, Univ of Copenhagen, Denmark.

THORUP, M. 1993. Topics in computation. Ph.D. thesis, Programming Research Group, Oxford Univ., Oxford, U.K.

THORUP, M. 1992. Ambiguity for incremental parsing and evaluation Tech. Rep. PRG-TR-24-92, Programming Research Group, Oxford Univ., Oxford, U.K.

TOMITA, M. 1986. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic, Boston, Mass.

WHARTON, R. 1976. Resolution of ambiguity in parsing. *Acta Informatica 6*, 4, 387–395.

WIRTH, N. 1985. *Programming in Modula-2*. 3rd ed. Springer-Verlag, New York.

Received February 1992; revised July 1993, September 1993, and January 1994; accepted January 1994