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 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.

This homemade reprent is typeset using the LaTeX document style `acmtrans`**. It shows how the published version was intended: the page references are still intact, figures and text are with the same fonts, and theoretically important material has not been deferred to an electronic appendix.**
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; e-mail: `mthorup@diku.dk`.

Ambiguous grammars have long been considered relevant in connection with programming languages, for as noticed in Aho, Johnson, and Ullman [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, Johnson, and Ullman [1975], 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. As the grammar is ambiguous the resulting parser will be non-deterministic due to conflicts in the action table. A deterministic LL or LR parser is then obtained by resolving these conflicts, possibly in a semi-automatic way using some disambiguating rules. As stated in Aho, Johnson, and Ullman [1975], 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

$$(1)\quad L\to LL,\ L\to \mathtt{1},\ L\to \mathtt{111}.$$

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 non-deterministic 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 $L\to\mathtt{1}$, s 4	r $L\to\mathtt{1}$	÷
4	s 5	÷	÷
5	r $L\to\mathtt{111}$	r $L\to\mathtt{111}$	÷

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 `11`, 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 paper we present some very general techniques for resolving the conflicts in a non-deterministic 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 $\mathcal{P}_{(1)}$ 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 non-deterministic 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 $\mathcal{P}_{(1)}$.

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 non-deterministic 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 $\mathcal{P}_{(1)}$, our techniques will construct the following set of rewriting rules:

$$\mathcal{E}_{(1)} : \left\{ \left(\begin{array}{l} L \\ |\backslash \\ LL \\ |\;|\backslash \\ \mathtt{l}LL \\ \;\;|\;| \\ \;\;\mathtt{l}\,\mathtt{l} \end{array} , \begin{array}{l} L \\ |\backslash\backslash \\ \mathtt{lll} \end{array} \right) , \left(\begin{array}{l} L \\ |\backslash \\ LL \\ \;\;|\backslash \\ \;\;LL \end{array} , \begin{array}{l} L \\ |\backslash \\ L\;L \\ |\backslash \\ LL \end{array} \right) \right\} .$$

Using these rewriting rules, the techniques will prove that $\mathcal{P}_{(1)}$ 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. In Aho, Johnson, and Ullman [1975] is 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 in 1965 presented the canonical LR technique [Knuth 1965], 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 the current paper where only the construction of deterministic parsers is considered. However, in the technical report Thorup [1992], based on the results in this paper, a theory is developed for the construction of non-deterministic 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. Firstly, 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. Secondly, 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 $\mathcal{E}_{(1)}$ we can read both that $\mathcal{P}_{(1)}$ is right associative and that it does not use $L \rightarrow \texttt{111}$.

Our techniques allow the user to state his "priorities" by supplying his own rewriting rules, thereby controlling the parser generation towards 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, Sethi, and Ullman 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 $\mathcal{E}_{(1)}$ 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, Hendriks, Klint, and Rekers 1990]. There ambiguity resolution is achieved by discarding parse trees that are non-minimal 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 doesn't 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 paper 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 by hand to identify the ambiguity and to 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, Goguen, Jouannaud, and Meseguer 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.

In order to keep the presentation short the paper will be focussed 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 paper 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 formalizes 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 non-deterministic 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 paper. These appendices contain the details of the generalization of the canonical LR(1) technique.

The paper is self-contained, but the preliminary definitions are rather dense, so the reader is referred to Aho, Sethi, and Ullman [1986] or Sippu [1988] and Sippu and Soisalon-Soininen [1990] for a standard text on grammars and parsing. The paper is a shortened version of Thorup [1993, 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 Thorup [1990] and Blikle, Tarlecki, and Thorup [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 concepts 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 $X{\rightarrow}\alpha$. We refer to X as the *leftside* and to α the *rightside* 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 leftside 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, Sethi, and Ullman [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 $X_1 \cdots X_p = \varepsilon$.

The following is a simple example of a grammar:

$$(2)\ \ S{\rightarrow}A,\ A{\rightarrow}B,\ A{\rightarrow}BC,\ B{\rightarrow}\mathtt{b},\ C{\rightarrow}\mathtt{c},\ C{\rightarrow}\varepsilon$$

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 leftside of the first production, and the terminals are the grammar symbols in `typewriter` font. Thus in the grammar (2) we have nonterminals S, A, B, C with S the start symbol, and we have terminals $\mathtt{b}, \mathtt{c}$.

Definition 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 $X{\rightarrow}^*\alpha$. The set of derived productions is defined recursively as follows:

—If X is a grammar symbol then $X{\rightarrow}^*X$ is a derived production.

—If $X{\rightarrow}^*\alpha Y\gamma$ is a derived production and $Y{\rightarrow}\beta$ is a production then $X{\rightarrow}^*\alpha\beta\gamma$ is a derived production.

For example, we have that $A{\rightarrow}^*\mathtt{b}C$ is a derived production for grammar (2). Notice that for derived productions, it is not required that the leftside is a nonterminal (as for productions, the leftside and the rightside refers to the first and the second coordinate, respectively). This dropping of the distinction between terminals and non-terminals 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, Tarlecki, and Thorup [1991], the concept of derived productions has been used in Ballance, Butcher, and Graham [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 $X{\rightarrow}\alpha$, we have that $X{\rightarrow}^*\alpha$ 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 $S{\rightarrow}^*\alpha$ 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 paper. 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 non-leaf 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 $X{\rightarrow}X_1 \cdots X_p$ 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:

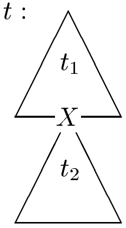
$$
t_{(2)} : \begin{array}{c} A \\ |\backslash \\ BC \\ | \\ \varepsilon \end{array} \ , \ u_{(2)} : \begin{array}{c} A \\ | \\ B \end{array}
$$

Clearly, $X \to^* \alpha$ 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* $X \to^* \alpha$, or that $X \to^* \alpha$ is the *derived production generated by* t. For example, $t_{(2)}$ and $u_{(2)}$ both generate $A \to^* B$. 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 paper, 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 *sub-parse 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 sub-tree of t excluding all nodes strictly descending from n, and let t_2 be the sub-tree of t containing all nodes descending from n including n itself. Then t_1 and t_2 are sub-parse trees of t. The construction is illustrated below with X being the label of the common node n of t_1 and t_2.

t :



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 sub-parse tree relation, in contrast to the normal sub-tree relation, respects the production structure of parse trees. For example, $u_{(2)}$

is a sub-tree but not a sub-parse tree of $t_{(2)}$. According to the definition of the sub-parse tree relation, we say a parse tree t is *minimal* in a set T of parse trees if t, but no proper sub-parse 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 $\mathrm{p\underline{t}}$ for parse trees. If X is a grammar symbol, by $\mathrm{p\underline{t}}(X)$ we denote the trivial parse tree whose single node is labeled by X. Now, let $X \rightarrow X_1 \cdots X_p$ be a production and $t_1, \ldots, t_p$ be parse trees with root symbols $X_1, \ldots, X_p$. Then by $\mathrm{p\underline{t}}(X; t_1, \ldots, t_p)$ we denote the parse tree with root symbol X and with $t_1, \ldots, t_p$ being the sub-parse trees descending from the sons of the root. Thus, continuing our example with grammar (2), we have $t_{(2)} = \mathrm{p\underline{t}}(A; \mathrm{p\underline{t}}(B), \mathrm{p\underline{t}}(C; \varepsilon))$ and $u_{(2)} = \mathrm{p\underline{t}}(A; \mathrm{p\underline{t}}(B))$.

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 paper. Thus *parsing* is the process of recognizing derived productions (not just sentences) and selecting parse trees that generate them.

A *parser* is an automaton which takes as input an *input production* meaning a pair (X, α), sometimes written $X \rightarrow^? \alpha$, 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 return some parse tree generating $X \rightarrow^* \alpha$.
(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 a 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 $\mathcal{P}$, we will refer to the parse trees it can return successfully as $\mathcal{P}$*-canonical* parse trees, possibly dropping the '$\mathcal{P}$' if some specific parser is understood. Thus completeness of $\mathcal{P}$ means that any derived production is generated by some $\mathcal{P}$-canonical parse tree.

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

An LR(1) parser $\mathcal{P}$ 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

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

Notice, that relative to traditional LR(1) parsers for sentences we have introduced the table *INIT* to take care of the leftside 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 non-terminals.

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

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

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 $X \xrightarrow{?} \alpha$ be the input production. Then the initial configuration or configuration 0 is $(\boxed{INIT(X)}, \alpha\$)$. 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 we have that the current configuration is of the form $(\boxed{INIT(X)} X \boxed{GOTO(INIT(X), X)}, \ \$)$. 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 $(\alpha \boxed{I}, \ X\beta)$, where X is a grammar symbol and $GOTO(I, X)$ is defined. The next configuration becomes $(\alpha \boxed{I} X \boxed{GOTO(I, X)} \beta)$.

—*Reduce* $X \rightarrow X_1 \cdots X_p$ is only valid if the current configuration is of the form $(\alpha \boxed{I} X_1 \boxed{I_1} \cdots X_p \boxed{I_p}, \ \beta)$ with $GOTO(I, X)$ defined. Then the subsequent configuration becomes $(\alpha \boxed{I} X \boxed{GOTO(I, X)}, \ \beta)$.

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

The parser $\mathcal{P}$ 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 leftside 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 $\text{p}\underline{\text{t}}(X)$ with X. The interesting case is when the parser reduces some production $X \rightarrow X_1 \cdots X_p$. 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 $\mathrm{p\underline{t}}(X; t_1, \ldots, t_p)$ 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

$$(3)\quad B{\to}B{*}B,\ B{\to}\mathtt{0},\ B{\to}\mathtt{1}.$$

Consider the following LR(1) parser for the grammar in (3): (`a`, `s`, and `r` stands 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 $B{\to}B{*}B$	÷	3	÷	÷
	5	÷	r $B{\to}\mathtt{0}$	÷	÷	r $B{\to}\mathtt{0}$	÷	÷	÷	÷
	6	÷	r $B{\to}\mathtt{1}$	÷	÷	r $B{\to}\mathtt{1}$	÷	÷	÷	÷
$*$	7	÷	s	÷	÷	÷	÷	8	÷	÷
	8	÷	÷	÷	÷	a	÷	÷	÷	÷
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

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

STACK	INP-BUF	ACTION
$\boxed{1}$	$\mathtt{1}{*}B{*}B\$$	s
$\boxed{1}\ \mathtt{1}_{\mathrm{p\underline{t}}(\mathtt{1})}\ \boxed{6}$	${*}B{*}B\$$	r $B{\to}\mathtt{1}$
$\boxed{1}\ B_{\mathrm{p\underline{t}}(B;\mathrm{p\underline{t}}(\mathtt{1}))}\ \boxed{2}$	${*}B{*}B\$$	s
$\boxed{1}\ B_{\mathrm{p\underline{t}}(B;\mathrm{p\underline{t}}(\mathtt{1}))}\ \boxed{2}\ {*}_{\mathrm{p\underline{t}}(*)}\ \boxed{3}$	$B{*}B\$$	s
$\boxed{1}\ B_{\mathrm{p\underline{t}}(B;\mathrm{p\underline{t}}(\mathtt{1}))}\ \boxed{2}\ {*}_{\mathrm{p\underline{t}}(*)}\ \boxed{3}\ B_{\mathrm{p\underline{t}}(B)}\ \boxed{4}$	${*}B\$$	s
$\boxed{1}\ B_{\mathrm{p\underline{t}}(B;\mathrm{p\underline{t}}(\mathtt{1}))}\ \boxed{2}\ {*}_{\mathrm{p\underline{t}}(*)}\ \boxed{3}\ B_{\mathrm{p\underline{t}}(B)}\ \boxed{4}\ {*}_{\mathrm{p\underline{t}}(*)}\ \boxed{3}$	$B\$$	s
$\boxed{1}\ B_{\mathrm{p\underline{t}}(B;\mathrm{p\underline{t}}(\mathtt{1}))}\ \boxed{2}\ {*}_{\mathrm{p\underline{t}}(*)}\ \boxed{3}\ B_{\mathrm{p\underline{t}}(B)}\ \boxed{4}\ {*}_{\mathrm{p\underline{t}}(*)}\ \boxed{3}\ B_{\mathrm{p\underline{t}}(B)}\ \boxed{4}$	$\$$	r $B{\to}B{*}B$
$\boxed{1}\ B_{\mathrm{p\underline{t}}(B;\mathrm{p\underline{t}}(\mathtt{1}))}\ \boxed{2}\ {*}_{\mathrm{p\underline{t}}(*)}\ \boxed{3}\ B_{\mathrm{p\underline{t}}(B;\mathrm{p\underline{t}}(B),\mathrm{p\underline{t}}(*),\mathrm{p\underline{t}}(B))}\ \boxed{4}$	$\$$	r $B{\to}B{*}B$
$\boxed{1}\ B_{\mathrm{p\underline{t}}(B;\mathrm{p\underline{t}}(B;\mathrm{p\underline{t}}(\mathtt{1})),\mathrm{p\underline{t}}(*),\mathrm{p\underline{t}}(B;\mathrm{p\underline{t}}(B),\mathrm{p\underline{t}}(*),\mathrm{p\underline{t}}(B)))}\ \boxed{2}$	$\$$	a

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

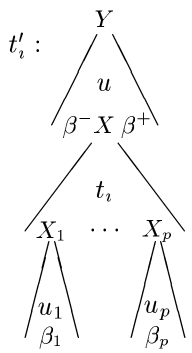
$$\mathrm{p\underline{t}}(B;\mathrm{p\underline{t}}(B;\mathrm{p\underline{t}}(\mathtt{1})),\mathrm{p\underline{t}}(*),\mathrm{p\underline{t}}(B;\mathrm{p\underline{t}}(B),\mathrm{p\underline{t}}(*),\mathrm{p\underline{t}}(B))) = \begin{array}{l} \ B \\ \ |\backslash\backslash \\ B{*}B \\ |\ \ \ |\backslash\backslash \\ \mathtt{1}\ \ B{*}B \end{array}$$

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, in his seminal paper [Knuth 1965] Knuth 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 prelimary 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. Loosely speaking, 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 speaking 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 $\imath = 0, 1$, the parse tree $t'_\imath$ results from $t_\imath$ by first for $i = 1, \ldots, p$, rooting u_i in the ith non-ε leaf of $t_\imath$, 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 $\mathcal{E}$ of parse tree pairs, by $\mathcal{E}^\prec$ we denote the closure of $\mathcal{E}$ under substitution and transitivity.

Definition 2. A set $\mathcal{E}$ of parse tree pairs is *prioritizing* if $\mathcal{E}^\prec$ is a well-founded (no infinite descending sequence, i.e. no infinite sequence $t_0 \cdots t_k \cdots$ of parse trees with $(t_{i+1}, t_i) \in \mathcal{E}^\prec$ for all i) strict partial ordering, and no two $\mathcal{E}^\prec$-minimal parse trees are equivalent. If, moreover, $\mathcal{P}$ is a parser and the $\mathcal{P}$-canonical parse trees coincide with the $\mathcal{E}^\prec$-minimal parse trees, then $\mathcal{E}$ *prioritizes* $\mathcal{P}$.

Thus, if we say that a parse tree t has higher priority than a parse tree u whenever $(t, u) \in \mathcal{E}^\prec$, then the $\mathcal{P}$-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 $\mathcal{E}$, thereby showing his priorities for the parser $\mathcal{P}$.

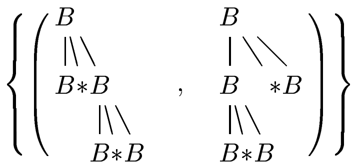
PROPOSITION 4.1. *If a set $\mathcal{E}$ of parse tree pairs prioritizes a parser $\mathcal{P}$, then $\mathcal{P}$ is complete.*

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

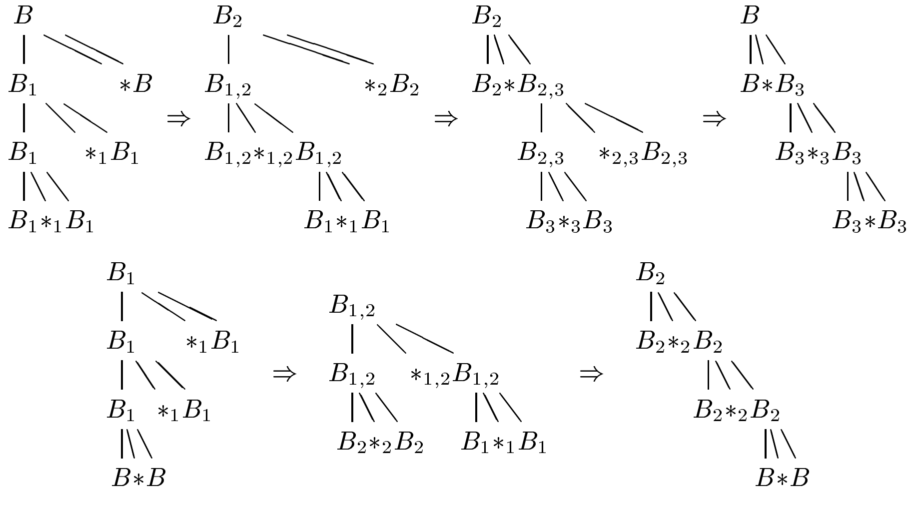
Sometimes, we will think about the definition of prioritized parsers in terms of rewriting. We say that the parse tree pair (t,u) *rewrites* the parse tree u' to the parse tree t' if (t',u') follows from (t,u) by substitution. Notice, that u' is not uniquely defined in terms of t, u, and t', for u might occur at different positions in u'. Now, let $\mathcal{E}$ be a set of parse tree pairs prioritizing a parser $\mathcal{P}$. Then, the definition of prioritizing says that given any parse tree, if, as long as possible, we rewrite with the parse tree pairs from $\mathcal{E}$, then eventually we will arrive at a $\mathcal{P}$-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 paper where, typically, we will be interested in expansions of $\mathcal{E}^{\prec}$ 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), page 10, they could return

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



which prioritizes the parser $\mathcal{P}_{(3)}$ that we found at page 10. Below are shown two different maximal rewriting sequences with $\mathcal{E}_{(3)}$, starting in the same parse tree. (The indices indicate the sub-parse 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 $\mathcal{E}$ is given, it is sufficient to check for each parse tree pair in $\mathcal{E}$, 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 $\mathcal{P}$-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 prioriting set $\mathcal{E}_{(3)}$, so its ambiguity is irrelevant with respect to some semantics if and only if these semantics associate an associative operation with the production $E \rightarrow E * E$. 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 $u = \mathrm{p}\underline{\mathrm{t}}(X)$, and local ambiguity corresponds to the case where $u_1, \cdots, u_p$ 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 whilst our prioritizing sets once and for all characterizes the ambiguity for the whole grammar. On the other hand, our techniques only works for restricted classes of grammars, for which finite prioritzing sets can be found.

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

```
B
| \ \
B   *B
|\\
B*B
```

as a sub-parse 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 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 $\mathcal{E}$ of parse tree pairs and a parser $\mathcal{P}$ such that $\mathcal{E}$ prioritizes $\mathcal{P}$. 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 prioritizing 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 paper 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

In the following we assume that the LL and LR techniques have been modified to work with derived productions instead of sentences. We will be focussing 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, 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 productions) will try to construct a complete linear time LL or LR parser. Unfortunately they can never be successful if the grammar is ambiguous. The techniques will always succeed in generating a generally non-deterministic 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 non-deterministic 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 non-deterministic 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

$$(4)\quad E{\to}F,\ F{\to}E,\ F{\to}FF.$$

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 non-deterministic 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 $\mathcal{U}$ 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 $a, b \in \mathcal{U}_e$. Now, the triple (e, a, b) is called the *projection* on $\mathcal{U}$ 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):

$$a_{(4)} : \begin{pmatrix} \begin{array}{l} F \\ \mid\diagdown \\ F\ \ F \\ \mid\!\diagdown\ \ \mid \\ FFE \\ \quad\ \ \mid \\ \quad\ \ F \end{array} , & \begin{array}{l} \quad F \\ \quad\mid\!\diagdown \\ \quad FF \\ \quad\mid\ \mid\!\diagdown \\ EFF \\ \mid \\ F \end{array} \end{pmatrix}.$$

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

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

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

An *ordering* $\mathcal{O}$ of a universal LL or LR parser $\mathcal{U}$ is represented by associating with each entry e in the action table a strict partial ordering $\mathcal{O}_e$ of $\mathcal{U}_e$. Thereby $\mathcal{O}$ 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 $\mathcal{U}$. Then $(t, u) \in \mathcal{O}$ if and only if $(a, b) \in \mathcal{O}_e$. Notice that because it is required that all the $\mathcal{O}_e$s are strict partial orderings, the definition guarantees that in fact $\mathcal{O}$ is a strict partial ordering. Also notice that if $\mathcal{E}$ is a set of parse tree pairs and there is an ordering of $\mathcal{U}$ that contains $\mathcal{E}$, then there is a unique least such ordering $\mathcal{O}$; namely, the "entry-wise" transitive closure of the projection of $\mathcal{E}$ in $\mathcal{U}$. In this case we say that $\mathcal{O}$ is the universal parser ordering *spanned* by $\mathcal{E}$; otherwise $\mathcal{E}$ does not span a universal parser ordering.

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

Definition 3. Let $\mathcal{E}$ be a set of parse tree pairs, $\mathcal{U}$ be a universal LL or LR parser, and $\mathcal{P}$ be a determinization of $\mathcal{U}$. Then $\mathcal{E}$ *$\mathcal{U}$-prioritizes* $\mathcal{P}$ if $\mathcal{E}$ prioritizes $\mathcal{P}$ and $\mathcal{E}^{\prec}$ spans an ordering of $\mathcal{U}$ in which $\mathcal{P}$ is minimal.

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

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

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

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

The role of $\mathcal{E}_{in}$ is to allow the user to impose his own priorities on the parser construction. Continuing our example with grammar (4), if we set $\mathcal{E}_{in}$ to be the set of the first two parse tree pairs in $\mathcal{E}_{(4)}$, we specify that the generated parser should be "cycle-free", but leave it open whether it should be left or right associative.

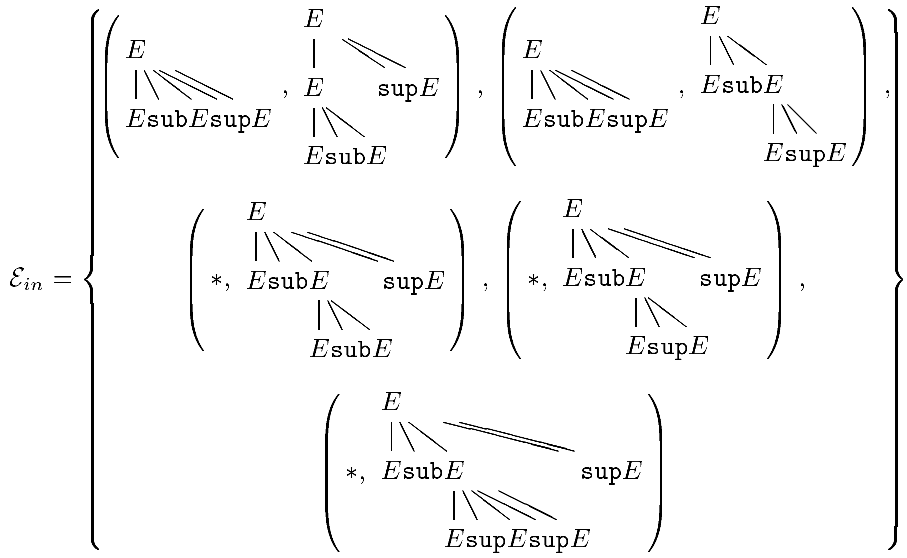
Admittedly, in order to keep down the size of the presentation, we will only prove Main Theorem 5.1 in detail for the CLR(1) technique. This is done constructively in Section 8 together with the appendices. In the following, when we talk unspecified about *our technique*, it is understood that we are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

6. DISAMBIGUATING WITH PRIORITIES

In this section we will discuss in some more detail how the priorities from $\mathcal{E}_{in}$ in Main Theorem 5.1 can be used like traditional rewriting rules. The discussion will be based on an example stemming from Aho, Sethi, and Ullman [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 grammar

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

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



The 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 $\mathcal{E}_{in}$ (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 lead 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 $E \to E\,\texttt{sub}\,E\,\texttt{sup}\,E$ rather than with $E \to E\,\texttt{sup}\,E$. Technically this is told to Yacc, by listing the production $E \to E\,\texttt{sub}\,E\,\texttt{sup}\,E$ first in the declaration of the grammar—as we already did.

It seems that one needs to be quite familar with Yacc and the LR(1) techniques, in order to feel confortable 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 $E{\rightarrow}E\,\mathtt{sub}\,E\,\mathtt{sup}\,E$ 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 beof the same precedence. Technically this is told to Yacc via the declaration `%left 'sub' 'sup'`. Such a declaration overrules the heuristic about prefering 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 $E{\rightarrow}E\,\mathtt{sub}\,E\,\mathtt{sup}\,E$ rather than with $E{\rightarrow}E\,\mathtt{sup}\,E$, but it implies that we will never get to a state from which it is possible to reduce $E{\rightarrow}E\,\mathtt{sub}\,E\,\mathtt{sup}\,E$. 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:

$$(6)\quad S{\rightarrow}\mathtt{t}\,S\,\mathtt{e}\,S,\ S{\rightarrow}\mathtt{t}S,\ S{\rightarrow}\mathtt{s}.$$

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:

$$\mathcal{E}_{(6)}: \left\{ \left(\begin{array}{l} S \\ |\backslash \\ \mathtt{t}S \\ \quad |\backslash\backslash\backslash \\ \quad \mathtt{t}S\mathtt{s}S \end{array} ,\ \begin{array}{l} S \\ |\backslash\backslash\backslash \\ \mathtt{t}S\ \mathtt{e}S \\ \ \ |\backslash \\ \ \ \mathtt{t}S \end{array} \right), \ldots, \left(\begin{array}{l} S \\ |\backslash \\ \mathtt{t}S \\ \quad |\backslash\backslash\backslash \\ \quad \mathtt{t}S\mathtt{e}S \\ \qquad |\backslash\backslash\backslash \\ \qquad \mathtt{t}S\mathtt{e}S \\ \qquad\quad |\backslash\backslash\backslash \\ \qquad\quad \mathtt{t}S\mathtt{e}S \end{array} ,\ \begin{array}{l} S \\ |\backslash \quad \backslash\backslash \\ \mathtt{t}S \qquad\quad \mathtt{e}S \\ \quad |\backslash\backslash\backslash \\ \quad \mathtt{t}S\mathtt{e}S \\ \qquad |\backslash\backslash\backslash \\ \qquad \mathtt{t}S\mathtt{e}S \\ \qquad\quad |\backslash \\ \qquad\quad \mathtt{t}S \end{array} \right), \ldots \right\}$$

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

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

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 Main 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 $\mathcal{G}$, some universal LL or LR parser $\mathcal{U}$ for $\mathcal{G}$, and a finite set $\mathcal{E}_{in}$ of parse tree pairs. Our goal is to find a determinization $\mathcal{P}$ of $\mathcal{U}$, and a finite set $\mathcal{E}_{out}$ of parse tree pairs such that $\mathcal{P}$ and $\mathcal{E}_{out}$ *match* each other in the sense that $(\mathcal{E}_{in} \cup \mathcal{E}_{out})$ $\mathcal{U}$-prioritizes $\mathcal{P}$. If such a matching pair exists we want $\mathcal{E}_{out}$ to be minimal given $\mathcal{P}$, and $\mathcal{P}$ to be a linear time parser. The latter condition will be shown always to be satisfied when $\mathcal{P}$ 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 $\mathcal{R}$ of parse tree pairs defined as follows: The second coordinates in $\mathcal{R}$ are the minimal parse trees amongst the non-trivial parse trees having the root symbol as a single symbol in the frontier. Let t be any second coordinate in $\mathcal{R}$, and let $X \to^* X$ be the derived production generated by t, then the corresponding first coordinate is $\mathrm{p\underline{t}}(X)$. For grammar (3), pare 10, the set $\mathcal{R}$ is empty, but for grammar (4), page 14, we have $\mathcal{R}$ equal to

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

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

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

—*All $\mathcal{E}^{\prec}$-minimal parse trees are $\mathcal{P}$-canonical.*

—*$(\mathcal{E} \cup \mathcal{R})^{\prec}$ spans an ordering of $\mathcal{U}$ in which $\mathcal{P}$ is minimal.*

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

(i) All $(\mathcal{E}_{in} \cup \mathcal{E}_{out})^{\prec}$-minimal parse trees are $\mathcal{P}$-canonical.

(ii) $(\mathcal{E}_{in} \cup \mathcal{E}_{out} \cup \mathcal{R})^{\prec}$ spans an ordering of $\mathcal{U}$ in which $\mathcal{P}$ is minimal.

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

Trivially (ii) can only be satisfied if $\mathcal{P}$ is minimal in an ordering of $\mathcal{U}$ spanned by $(\mathcal{E}_{in} \cup \mathcal{R})^{\prec}$. 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 little new ambiguity in each step, then the set $\mathcal{E}_{in}$ will largely determine the parsing, keeping the number of relevant determinizations small. Thus we check that $(\mathcal{E}_{in} \cup \mathcal{R})^{\prec}$ spans an ordering $\mathcal{O}$ of $\mathcal{U}$. If not, we report that there cannot be a matching pair.

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

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

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

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

The above outlines an algorithm for the computation described in Main 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 non-canonical action in a given second coordinate.

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

1) Given a finite set $\mathcal{E}$ of parse tree pairs how do we check that $\mathcal{E}^{\prec}$ spans an

ordering of $\mathcal{U}$? If it does, how do we construct this ordering? The problem is to construct the projections in $\mathcal{U}$ of the possibly infinite set of parse tree pairs in $\mathcal{E}^{\prec}$.

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

3) Given a determinization $\mathcal{P}$ and a parse tree t, how do we check that there is a $\mathcal{P}$-canonical equivalent to t? The problem is that $\mathcal{P}$ 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 non-terminating.

All these problems are sorted out in the appendices relative to the CLR(1) technique (Knuth's canonical LR(1) technique [Knuth 1965] 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 $\mathcal{P}$ 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 set up, Appendix D solves problem 1, Appendix E addresses problem 2, Appendix F addresses problem 3, and Appendix G deals with problem 4. After having addressed all the above problems, Appendix H puts everything together in an exact algorithm for the computation described in Main 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 paper 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 paper 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 me with the presentation of this work. Many others have been helpful with suggestions, comments, and encouragement during the years I have been working 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.

References

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

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

BALLANCE, R., BUTCHER, J., AND GRAHAM, S. 1988. Grammatical abstraction and incremental syntax analysis in a language-based editor. *SIGPLAN Notices 23,* 7, 185–198.

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

BLIKLE, A. 1989. Denotational engineering. *Science of Computer Programming 12,* 3, 207–253.

BLIKLE, A., TARLECKI, A., AND THORUP, M. 1991. On conservative extensions of syntax in system development. In *Images of Programming, Dedicated to the memory of A.P. Erchov*, North Holland, Amsterdam, pp. 209–235.

BLIKLE, A. AND THORUP, M. 1990. On conservative extensions of syntax in the process of system development. In *Proc. VDM'90, VDM and Z—Formal Methods in Software Development*, LNCS 428, Springer, Berlin, pp. 504–525.

DEMERS, A. 1974. Skeletal LR parsing. In *IEEE Conf. Rec. of the 15th Ann. Symp. on Switching and Automata Theory*, IEEE, New York, pp. 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 *Proc. 12th ACM Symposium on Principles of Programming Languages*, ACM, New York, pp. 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 Tans. on Software Engineering 16,* 12, 1344–1351.

JOHNSON, S. 1975. Yacc—yet another compiler compiler. Technical Report CS-32, AT&T Bell Laboratories, Murray Hill, NJ.

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

KNUTH, D. 1965. On translation of languages from left to right. *Information and Control 8,* 6, 607–639.

KRUSKAL, J. 1960. Well-quasi-ordering, the tree theorem, and vazsonyi's conjecture. *Trans. Amer. 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 *Proc. 5th ACM Symposium on Principles of Programming Languages*, ACM, New York, pp. 241–254.

REKERS, J. 1991. Generalized LR parsing for general context-free grammars. Technical Report Technical Report 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 15.* Berlin, London: Springer-Verlag.

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

SOISALON-SOININEN, E. AND TARHIO, J. 1988. Looping LR parsers. *Information Processing Letters 26,* 5, 251–253.

THORUP, M. 1992. Ambiguity for incremental parsing and evaluation. Technical Report PRG-TR-24-92, Programming Research Group of Oxford University.

THORUP, M. 1993. *Topics in Computation.* Ph. D. thesis, Programming Research Group of Oxford University.

THORUP, M. 1994. Disambiguating grammars by exclusion of sub-parse trees. Technical Report DIKU-94/11, Dept. of Computer Science, University of Copenhagen.

TOMITA, M. 1986. *Efficient parsing for natural language: a fast algorithm for practical systems.* Kluwer Academic Publishers.

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, Berlin, New York.

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

Appendix

An appendix to this paper 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.

THIS DOCUMENT IS THE APPENDIX TO THE FOLLOWING PAPER:

Controlled Grammatic Ambiguity
MIKKEL THORUP
University of Copenhagen

1267

APPENDIX A. THE PROPERTIES OF $\mathcal{R}$

This appendix is devoted to proving Theorem 8.1, on which hings the correctness of the techniques presented in this paper. Recall the statement of Theorem 8.1:

> *Let $\mathcal{P}$ be a determinization of a universal LL or LR parser $\mathcal{U}$, and let $\mathcal{E}$ be a set of parse tree pairs. Then $\mathcal{E}$ $\mathcal{U}$-prioritizes $\mathcal{P}$ if and only if the following conditions are satisfied:*
> *(i) All $\mathcal{E}^{\prec}$-minimal parse trees are $\mathcal{P}$-canonical.*
> *(ii) $(\mathcal{E} \cup \mathcal{R})^{\prec}$ spans an ordering of $\mathcal{U}$ in which $\mathcal{P}$ is minimal.*

Our first step in the proof is to show the following simple result:

OBSERVATION A.1. *If a set $\mathcal{E}$ of parse tree pairs prioritizes a parser $\mathcal{P}$, then $\mathcal{R} \subseteq \mathcal{E}^{\prec}$.*

PROOF. Assume that $\mathcal{E}$ prioritizes $\mathcal{P}$. Then, in particular, $\mathcal{P}$ is complete, and the set of $\mathcal{P}$-canonical parse trees is stable (i.e. closed under the sub-parse trees operation). Consider any parse tree pair $(\mathrm{p}\underline{\mathrm{t}}(X), t) \in \mathcal{R}$. Since $\mathcal{E}$ prioritizes $\mathcal{P}$ and since $\mathrm{p}\underline{\mathrm{t}}(X)$ has no proper sub-parse trees, if $\mathrm{p}\underline{\mathrm{t}}(X)$ is not $\mathcal{P}$-canonical, it must be a second coordinate in $\mathcal{E}$. Hence no parse tree containing $\mathrm{p}\underline{\mathrm{t}}(X)$ can be canonical, so, in particular, there cannot be a canonical parse tree for $X \rightarrow^* X$. This contradicts that $\mathcal{P}$ is complete, so we may conclude that $\mathrm{p}\underline{\mathrm{t}}(X)$ is a $\mathcal{P}$-canonical parse tree. Thus, since $\mathcal{E}$ prioritizes $\mathcal{P}$, we have $(\mathrm{p}\underline{\mathrm{t}}(X), t) \in \mathcal{E}^{\prec}$, as desired. □

The main step in establishing Theorem 8.1 is to prove the following:

KEY LEMMA A.2. *Let $\mathcal{E}$ be a set of parse tree pairs. If $(\mathcal{E} \cup \mathcal{R})^{\prec}$ is irreflexive, then is $\mathcal{E}^{\prec}$ is well-founded.*

Before proving the key lemma, first we will see how it implies Theorem 8.1.

PROOF THAT KEY LEMMA A.2 IMPLIES THEOREM 8.1.

$\underline{\Leftarrow:}$ Assume (i) and (ii). Clearly, the two together imply that the $\mathcal{P}$-canonical and the $\mathcal{E}^{\prec}$-mimimal parse trees coincide. Moreover, since all universal parser orderings are strict, by (ii) we have that $(\mathcal{E} \cup \mathcal{R})^{\prec}$ is irreflexive, and hence, by Key Lemma A.2, that $\mathcal{E}^{\prec}$ is well-founded. Thus, by definition, $\mathcal{E}$ prioritizes $\mathcal{P}$, which, together with (ii), implies that $\mathcal{E}$ $\mathcal{U}$-prioritizes $\mathcal{P}$, as desired.

$\underline{\Rightarrow:}$ Assume that $\mathcal{E}$ $\mathcal{U}$-prioritizes $\mathcal{P}$. Then, in particular, $\mathcal{E}$ prioritizes $\mathcal{P}$, so by Observation A.1, we have $\mathcal{R} \subseteq \mathcal{E}^{\prec}$. Thus $\mathcal{E}^{\prec} = (\mathcal{E} \cup \mathcal{R})^{\prec}$, so (i) and (ii) follows directly from the definition of $\mathcal{U}$-prioritizing.

□

The rest of this appendix is devoted to proving Key Lemma A.2. First some technical definitions. Let t be a non-trivial parse tree with $(n_1, \ldots, n_p)$ being the sequence of sons of the root. Then for $i = 1, \ldots, p$, by $t(i)$ we denote the sub-parse tree descending from n_i, i.e. $t = \mathrm{p\underline{t}}(X; t(1), \ldots, t(p))$ for some X. An auxiliary tree embedding relation $\sqsubseteq$ is defined as the reflexive partial ordering generated by the following conditions:

—If t is a non-trivial parse tree with the root production $X \to X_1 \cdots X_p$, then for $i = 1, \ldots, p$, we have $t(i) \sqsubseteq t$.

—If t, u are non-trivial parse trees with the common root production $X \to X_1 \cdots X_p$, and for $i = 1, \ldots, p$, we have $t(i) \sqsubseteq u(i)$, then $t \sqsubseteq u$.

Now, the following lemma can be seen as a variant for parse trees of Kruskal's Tree Theorem [Kruskal 1960].

Lemma A.3. *Given an infinite sequence $t_0 \cdots t_k \cdots$ of parse trees there exist two numbers i, j such that $i < j$ and $t_i \sqsubseteq t_j$.*

Proof. (essentially that of Nash-Williams [Nash-Williams 1963]) In the proof we identify a set I of numbers (≥ 0) with the sequence consisting of the numbers in ascending order, and then by $I[j]$ we denote the $(j+1)$th number in the sequence.

Suppose there exists a counterexample to the lemma. Let $\boldsymbol{t}$ be a minimal counterexample in the lexicographic ordering induced by $\sqsubseteq$: starting with the empty sequence, suppose we have obtained the sequence $t_0 \cdots t_{k-1}$ of parse trees; then let t_k be a $\sqsubseteq$-minimal parse tree such that $t_0 \cdots t_k$ is a prefix of a counterexample.

There are only finitely many grammar symbols, so $\boldsymbol{t}$ will contain infinitely many non-trivial parse trees. Moreover, there are only finitely many grammar productions, so there is a grammar production $X \to X_1 \cdots X_p$ which is root production of infinitely many of the parse trees in $\boldsymbol{t}$. Denote by I the infinite set of indices of parse trees with $X \to X_1 \cdots X_p$ as root production.

Fix $l \in \{1, \ldots, p\}$ and let J be any infinite subset I. We will now find an infinite subset L of J so that the infinite subsequence $t_{L[0]}(l) \cdots t_{L[k]}(l) \cdots$ of $\boldsymbol{t}$ is $\sqsubseteq$-ascending.

Denote by K the subset of indices k from J such that for some parse tree t which is $\sqsubseteq$-minimal in $\{t_j(l) | j \in J\}$, we have that k is the smallest index satisfying $t_k(l) = t$. We want to prove that K is finite, so suppose, to the contrary, that K is infinite, and consider the infinite sequence

$$\boldsymbol{u} = u_0 \cdots u_k \cdots = t_0 \cdots t_{K[0]-1} t_{K[0]}(l) \cdots t_{K[k']}(l) \cdots$$

By the definition of $\sqsubseteq$, we have that $t_{K[0]}(l) \sqsubseteq t_{K[0]}$, so $\boldsymbol{u}$ is less than $\boldsymbol{t}$ in the $\sqsubseteq$-lexicographic ordering, but $\boldsymbol{t}$ was a minimal counterexample to the lemma, so $\boldsymbol{u}$ is not a counterexample to the lemma. Thus, for some i, j where $i < j$, we have $u_i \sqsubseteq u_j$.

$\underline{i, j < K[0]}$: Then $u_i = t_i$ and $u_j = t_j$, so $t_i \sqsubseteq t_j$ follows from $u_i \sqsubseteq u_j$, contradicting that $\boldsymbol{t}$ is a counterexample to the lemma.

$\underline{i < K[0] \le j}$: Set $j' := K[j - K[0]]$. Then $t_i = u_i \sqsubseteq u_j = t_{j'}(l)$. By the definition of $\sqsubseteq$ we have $t_{j'}(l) \sqsubseteq t_{j'}$ so by transitivity of $\sqsubseteq$, we get $t_i \sqsubseteq t_{j'}$. But $j' \ge j > i$ so $t_i \sqsubseteq t_{j'}$ contradicts that $\boldsymbol{t}$ is a counterexample to the lemma.

$\underline{K[0] \le i, j}$: Set $i' := K[i - K[0]]$ and $j' := K[j - K[0]]$. Then $t_{i'}(l) = u_i \sqsubseteq u_j = t_{j'}(l)$, so by the definition of K, since $i' < j'$, we have that j' cannot be in K.

Thus K is finite.

Since K is finite and J infinite, we can select a $k \in K$ such that the set $J^k = \{j \in J | k < j, t_k(l) \sqsubseteq t_j(l)\}$ is infinite. Now, let k be the first element of L, i.e. set $L[0] = k$. We construct the rest of L by applying the above procedure recursively with with the infinite subset J^k of J in place of J.

We have now seen that for all $l \le p$, if J is an infinite subset of I, there is an infinite subset L of J such that the infinite sequence $t_{L[0]}(l) \cdots t_{L[k]}(l) \cdots$ of parse trees is $\sqsubseteq$-ascending. In particular, this implies that there are infinite sets $L_1, \ldots, L_p$ of indices such that $I \supseteq L_1 \supseteq \cdots \supseteq L_p$, and such that for $l = 1, \ldots, p$, the sequence $t_{L_l[0]}(l) \cdots t_{L_l[k]}(l) \cdots$ is $\sqsubseteq$-ascending.

From the construction of L_p it follows that for $l = 0, \ldots, n$, the infinite sequence $t_{L_p[0]}(l) \cdots t_{L_p[k]}(l) \cdots$ is $\sqsubseteq$-ascending. Hence, by the definition of $\sqsubseteq$, we have that $t_{L_p[0]} \cdots t_{L_p[k]} \cdots$ is $\sqsubseteq$-ascending. Thus, for example, $t_{L_p[0]} \sqsubseteq t_{L_p[1]}$ contradicting that $\boldsymbol{t}$ is a counterexample to the lemma. $\square$

Denote by $\mathcal{R}^{\preceq}$ the reflexive closure of $\mathcal{R}^{\prec}$. Thus $t\,\mathcal{R}^{\preceq}\,u$ means that t results from u by contraction of sub-parse trees with the root symbol as the only symbol in the frontier.

LEMMA A.4. *If two parse trees t and u are equivalent and $t \sqsubseteq u$, then $t\,\mathcal{R}^{\preceq}\,u$*

PROOF. First we observe an alternative but equivalent definition of $\sqsubseteq$, that for all parse trees t, u, we have $t \sqsubseteq u$ if and only if there is a sequence $t = t_0, \ldots, t_k = u$ of parse trees satisfying the following conditions:

(a) t_1 consists of t_0 rooted in some parse tree v_0 which might be trivial.

(b) For $i = 1, \ldots, k-1$, there are three parse trees u_i, v_i, w_i where v_i and w_i have common root symbol X_i, and where, moreover, X_i is the label of some leaf k_i of u_i and of some leaf l_i of v_i, and such that

—t_i consists of w_i rooted in k_i in u_i.

—t_{i+1} consists of w_i rooted in l_i in v_i which in turn is rooted in k_i in u_i.

Thus t_i is obtained from t_{i+1} by contraction of v_i.

Assume that $t_0 = t$ and $t_k = u$ are equivalent. We want to show that all the t_is are equivalent. This will imply the lemma, because then, for $i = 0, \ldots, k-1$, the root symbol of v_i would be the one and only symbol in the frontier of v_i, and hence we would have $t_i\,\mathcal{R}^{\preceq}\,t_{i+1}$. Thus, $t_0\,\mathcal{R}^{\preceq}\,t_k$ would follow by transitivity.

For $i = 1, \ldots, k-1$. the parse trees t_i and t_{i+1} have the same root; namely the root of u_i. Moreover, t_0 and t_k have the same root since they are equivalent. Hence all of the t_is have the same root. For $i = 0, \ldots, k-1$, all the symbols in the frontier of t_i occur in in the frontier of t_{i+1} in the same order. Thus, since t_0 and t_k have the same frontier, we may conclude that all the t_is have the same frontier, and hence that they are equivalent. $\square$

PROOF OF KEY LEMMA A.2. Putting Lemma A.3 and Lemma A.4 together, immediately we get the following result:

Given an infinite sequence $t_0 \cdots t_k \cdots$ of equivalent parse trees, there exist two numbers i, j such that $i < j$ and $t_i \, \mathcal{R}^{\preceq} \, t_j$.

Now, let $\mathcal{E}$ be a set of parse tree pairs, and assume that $(\mathcal{E} \cup \mathcal{R})^{\prec}$ is irreflexive. The latter gives that $\mathcal{E}^{\prec}$ is a strict partial ordering. In order to establish the key lemma, we have to prove that $\mathcal{E}^{\prec}$ is well-founded.

Suppose that $\mathcal{E}^{\prec}$ is not well-founded. Then, by definition, there is then an infinite $\mathcal{E}^{\prec}$-descending sequence $t_0 \cdots t_k \cdots$ of parse trees. By the above result, for some numbers i, j, we have $i < j$ and $t_i \, \mathcal{R}^{\preceq} \, t_j$. Thus, either $t_i \, \mathcal{R}^{\prec} \, t_j$, or $t_i = t_j$; but by definition of the sequence we have $t_j \, \mathcal{E}^{\prec} \, t_i$, so both cases contradict that $(\mathcal{E} \cup \mathcal{R})^{\prec}$ is irreflexive. Thus $\mathcal{E}^{\prec}$ is well-founded, as desired. □

APPENDIX B. UNIVERSAL CLR(1) PARSERS

Recall that by *the universal CLR(1) parser of a grammar*, we refer to the generally non-deterministic LR(1) parser returned by the canonical LR(1) technique [Knuth 1965] when applied to the grammar. As usual, we assume that the technique has been modified to deal with derived productions. Essentially this parser $\mathcal{U}$ is defined by the following properties:

universality. There corresponds a unique parsing to any parse tree.

completability. Suppose we in a parsing arrive at a configuration of the form $(\alpha I, X\beta)$ and $a \in \mathit{ACTIONS}(I, X)$. Then for some β', from configuration $(\alpha I, X\beta')$, the parsing can be completed successfully starting with the action a.

no junk. For all states I there is an input symbol X such that $\mathit{ACTIONS}(I, X) \neq \emptyset$, and for all such X, there is a parsing encountering a configuration of the form $(\alpha I, X\beta)$.

minimality. It is the least general LR(1) parser satisfying the above conditions, i.e. if $\mathcal{U}'$ is another such parser, there is a surjection μ from the states in $\mathcal{U}'$ to the states in $\mathcal{U}$ such that for all states in I' in $\mathcal{U}'$ and for all input symbols X, if $I = \mu(I')$, we have $\mathit{ACTIONS}_{\mathcal{U}'}(I', X) = \mathit{ACTIONS}_{\mathcal{U}}(I, X)$ and $\mathit{GOTO}_{\mathcal{U}'}(I', X) = \mathit{GOTO}_{\mathcal{U}}(I, X)$.

We will now, in a rather condensed manner, recall the construction of the universal CLR(1) parser from Knuth [1965].

An *accept production* is an auxiliary production on the form $X' \rightarrow X$ where X is any grammar symbol. An *item* is a grammar production or an accept production with a '$\bullet$' somewhere in the rightside. Let I be any set of pairs of items and input symbols. By $\mathit{CLOSURE}(I)$ we denote the closure of I under the condition that if $(X_1 \rightarrow \beta \bullet X\gamma, Y_1) \in \mathit{CLOSURE}(I)$ and either $Y \in \mathit{FIRST}(\gamma) \setminus \$$, or $Y = Y_1$ and $\$ \in \mathit{FIRST}(\gamma)$, then for all grammar productions $X \rightarrow \alpha$, we have $(X \rightarrow \bullet\, \alpha, Y) \in \mathit{CLOSURE}(I)$. Here

$$\mathit{FIRST}(\varepsilon) = \$$$
$$\mathit{FIRST}(X\alpha) = \{X\} \cup \bigcup \{\mathit{FIRST}(\beta\alpha) | X \rightarrow \beta \text{ is a production}\}.$$

Now, the *goto graph* of a grammar is a directed graph where the vertices are sets of pairs of items and input symbols, and where the arcs are labeled with grammar symbols. It is generated by the following conditions:

—For each grammar symbol X, there is a vertex $\mathit{CLOSURE}(\{X' \rightarrow \bullet\, X, \$\})$.

—If I is a vertex and $(X \rightarrow \alpha \bullet Y\beta, Z) \in I$, there is a vertex

$$J = CLOSURE(\{(X \rightarrow \alpha Y \bullet \beta, Z) | (X \rightarrow \alpha \bullet Y\beta, Z) \in I\}),$$

and an arc from I to J labeled Y.

The universal CLR(1) parser is constructed directly from the goto graph:

—The states are the set of vertices.
—For all grammar symbols X, the state $INIT(X)$ equals $CLOSURE(\{X' \rightarrow \bullet X, \$\})$.
—For all states I and grammar symbols X, we have $GOTO(I, X)$ defined if there is an arc labeled X from the state I to some state J, in which case $GOTO(I, X) = J$.
—Let I be a state and X an input symbol; then in $ACTIONS(I, X)$ we have the action
accept if $X = \$$ and for some Y we have $(Y' \rightarrow Y \bullet, \$) \in I$.
shift if for some Y, α, β, Z, we have $(Y \rightarrow \alpha \bullet X\beta, Z) \in I$.
reduce $Y \rightarrow \alpha$ if for some Y, α, we have $(Y \rightarrow \alpha \bullet, X) \in I$.

It follows from the analysis in Knuth [1965], that, indeed, this construction is always finite, and that it satisfies all the properties that was mentioned in the beginning of this appendix.

For grammar (4) page 14, we get the goto graph in Figure 1. The numbers at the bottom right corner of each vertex indicate the corresponding state numbers from $\mathcal{U}_{(4)}$ on page 14.

APPENDIX C. PARTIAL PARSING AND OTHER USEFUL NOTIONS

In this appendix we will introduce some notation and some concepts in connection with universal CLR(1) parsers that will facilitate more involved reasoning.

A first important change is that we consider the symbol shifted by the action shift as being part of the action. Thus the action shift on input X becomes shift X.

For notational convinience, by shift $X_1 \cdots X_n$ we denote the action sequence (shift $X_1 \cdots$ shift X_n), and by $GOTO(I, X_1 \cdots X_n)$ we denote $GOTO(\cdots GOTO(I, X_1) \cdots X_n)$.

Let t be a parse tree and i a number such that $i \leq |t|$, i.e. i is at most the number of non-ε nodes in t. Then state(t, i), input(t, i), and action(t, i) denote the state of, the input of, and the action from configuration i in the parsing corresponding to t. Notice that the first action is action$(t, 0)$. Also notice that we always have action$(t, |t|)$ = accept, for the $non - \varepsilon$ leaves of t correspond to shifts and the internal nodes of t correspond to the leftsides of reductions.

Now, the completability and no junk properties of the universal CLR(1) parser can be stated together as follows. Let I be a state, X an input symbol and a an action. Then $a \in ACTIONS(I, X)$ if and only if for some parse tree t and some $i \leq |t|$, we have I = state(t, i), X = input(t, i), and a = action(t, i). Actually, this does not quite rule out superfluous states, so in order to get the full no junk property, we need to add that there is no state I such that $ACTIONS(I, X) = \emptyset$ for all input symbols X.

We need to be able to talk about *partial parsings*. A partial configuration consists of a suffix of a stack starting with a state. An initial partial configuration is a state only. The effect of the actions on the partial configurations is as the effect on the

Fig. 1: Goto graph ($X \to \alpha \bullet \beta,\ X_0, / \cdots / X_n$ is a shorthand for $(X \to \alpha \bullet \beta, X_0), \ldots, (X \to \alpha \bullet \beta, X_n)$)

$$
\begin{array}{cccc}
 & \left\{ \begin{array}{l} E' \to \bullet E,\ \$ \\ E \to \bullet F,\ \$/E/F \\ F \to \bullet E,\ \$/E/F \\ F \to \bullet FF,\ \$/E/F \end{array} \right\}_1 & & \\
 & \downarrow F & \searrow E & \\
 & \left\{ \begin{array}{l} E \to F \bullet,\ \$/E/F \\ F \to F \bullet F,\ \$/E/F \\ F \to \bullet E,\ \$/E/F \\ F \to \bullet FF,\ \$/E/F \\ E \to \bullet F,\ \$/E/F \end{array} \right\}_3 & & \left\{ \begin{array}{l} E' \to E \bullet,\ \$ \\ F \to E \bullet,\ \$/E/F \end{array} \right\}_2 \\
 & \downarrow F & \searrow E & \\
\stackrel{F}{\hookrightarrow} & \left\{ \begin{array}{l} F \to FF \bullet,\ \$/E/F \\ F \to F \bullet F,\ \$/E/F \\ E \to F \bullet,\ \$/E/F \\ F \to \bullet E,\ \$/E/F \\ F \to \bullet FF,\ \$/E/F \\ E \to \bullet F,\ \$/E/F \end{array} \right\}_5 & \stackrel{E}{\longrightarrow} & \left\{ F \to E \bullet,\ \$/E/F \right\}_4 \\
 & \uparrow F & \nearrow E & \uparrow E \\
 & \left\{ \begin{array}{l} F' \to F \bullet,\ \$ \\ F \to F \bullet F,\ \$/E/F \\ E \to F \bullet,\ \$/E/F \\ F \to \bullet E,\ \$/E/F \\ F \to \bullet FF,\ \$/E/F \\ E \to \bullet F,\ \$/E/F \end{array} \right\}_7 & \stackrel{F}{\longleftarrow} & \left\{ \begin{array}{l} F' \to \bullet F,\ \$ \\ F \to \bullet E,\ \$/E/F \\ F \to \bullet FF,\ \$/E/F \\ E \to \bullet F,\ \$/E/F \end{array} \right\}_6
\end{array}
$$

stacks of the normal configurations. If we are in a partial configuration of the form $(st\boxed{I})$, then an action a is possible if for some input symbol X we have

$$a \in \mathit{ACTIONS}(I, X) \setminus \{\text{accept}\}$$

and, moreover, in the case of a reduction with a production $X \to \alpha$, the stack contains all of α.

Since we changed the action shift to include the symbol shifted, a partial parsing is uniquely characterized by the state of the initial partial configuration and the sequence of actions.

We say that *from state I it is possible to carry out the sequence of actions $\boldsymbol{a}$ arriving at state J* if there is a partial parsing with initial configuration $\boxed{I}$, sequence $\boldsymbol{a}$ of actions, and state J in the final configuration. In this case we denote J by $I^{\boldsymbol{a}}$.

If I is a state, $\boldsymbol{a}$ and $\boldsymbol{b}$ are action sequences, and $I^{\mathbf{ab}}$ is defined, then trivially $I^{\boldsymbol{a}}$ is defined, but $(I^{\boldsymbol{a}})^{\boldsymbol{b}}$ might be undefined, e.g. if $\mathbf{a} = \text{shift}\,\alpha$ and $\boldsymbol{b} = \text{reduce}\,X \to \alpha$.

Partial parsings have some important qualities. Suppose from state I that it is possible to carry out the sequence of actions $\boldsymbol{a}$ arriving at state J, and moreover that the action a is possible in state J on input X. From the completability and no junk properties of universal CLR(1) parsers, it follows that there is a successful parsing that at some stage arrives at a configuration with state I, from there carries out the sequence $\boldsymbol{a}$ of actions arriving at state J with input X, and from there carries out the action a. Thus, for universal CLR(1) parsers, any partial parsing can be extended to a successful parsing. Notice that if our partial parsing starts in an initial state, then the extension can be chosen with the partial parsing as a prefix.

For the next appendix it is important to observe that if the action sequence $\boldsymbol{a}$ is a proper prefix of the action sequence for some parse tree with root symbol X, and, moreover, for the state I we have $I^{\text{shift}X}$ defined, then $I^{\boldsymbol{a}}$ is defined.

APPENDIX D. SPANNING A UNIVERSAL PARSER ORDERING

The whole appendix is devoted to proving the following result solving problem 1, page 20:

THEOREM D.1. *Given a universal CLR(1) parser $\mathcal{U}$ and a finite set $\mathcal{E}$ of parse tree pairs, we can decide if $\mathcal{E}^{\prec}$ spans an ordering of $\mathcal{U}$, and if so, construct this ordering.*

Denote by $\mathcal{E}^S$ the closure under substitution of $\mathcal{E}$. Then $\mathcal{E}^{\prec}$ is the transitive closure of $\mathcal{E}^S$, and all universal orderings are transitive, so $\mathcal{E}^{\prec}$ and $\mathcal{E}^S$ span the same ordering of $\mathcal{U}$, if any. Now, for all parse tree pairs (t,u), let $P(t,u)$ denote the finite set of projections (cf. page 15) into $\mathcal{U}$ of the parse tree pairs in the typically infinite set $\{(t,u)\}^S$. Assuming that $P(t,u)$ is computable, we have the following trivial algorithm for the computation described in Theorem D.1

Algorithm A: Given $\mathcal{U}$ and $\mathcal{E}$, and assuming that $P(t,u)$ can be computed for all $(t,u) \in \mathcal{E}$, the algorithm constructs the ordering of $\mathcal{U}$ spanned by $\mathcal{E}$ if it is defined; otherwise it reports that no such ordering is spanned.

A.1.Set $P := \bigcup\{P(t,u) | (t,u) \in \mathcal{E}\}$.

A.2.If $\perp \in P$ then report "$\mathcal{E}$ does not span an ordering of $\mathcal{U}$".

A.3.For all states I and input symbols X:
A.3.1. Set $\mathcal{O}_{I,X} := \{(a,b) | ((I,X),a,b) \in P\}$.
A.3.2. Set $\mathcal{O}_{I,X} :=$ the transitive closure of $\mathcal{O}_{I,X}$.
A.3.3. If $\mathcal{O}_{I,X}$ is not irreflexive then report "$\mathcal{E}$ does not span an ordering of $\mathcal{U}$".

A.4.Return $\mathcal{O}$.

Thus, Theorem D.1 follows if we can compute $P(t,u)$ for all $(t,u) \in \mathcal{E}$. An algorithm doing this is stated formally below. We use the notation that $\overline{0} = 1$ and $\overline{1} = 0$. Also we use the following auxiliary function from pairs of sequences of grammar symbols and sets of input symbols to sets of input symbols:

$$\mathit{FirstL}(\alpha, L) = \begin{cases} \mathit{FIRST}(\alpha) & \text{if } \$ \notin \mathit{FIRST}(\alpha) \\ (\mathit{FIRST}(\alpha) \setminus \{\$\}) \cup L & \text{otherwise.} \end{cases}$$

Here *FIRST* is as defined on page A–4. The correctness of the algorithm will be argued after its formal statement.

Algorithm B: Given $\mathcal{U}$, the algorithm computes $P(t_0,t_1)$ for any parse tree pair (t_0,t_1), i.e. it computes the set of projections in $\mathcal{U}$ of all the parse tree pairs that follows from (t_0,t_1) by substitution.

B.1.If $t_0 = t_1$ then return $\{\perp\}$.

B.2.Set $P := \emptyset$.

B.3.Let $X \rightarrow^* X_1, \ldots, X_p$ be the derived production generated by t_0 and t_1.

B.4.Let the sequences of actions $\boldsymbol{a}^-, \boldsymbol{a}_0^+, \boldsymbol{a}_1^+$, and the actions a_0, a_1, where $a_0 \neq a_1$, be defined such that for $\imath = 0, 1$, the action sequence $\boldsymbol{a}^- a_\imath \boldsymbol{a}_\imath^+$(accept) is the action sequence of $t_\imath$. Thus $\boldsymbol{a}^-$ is the largest common prefix of the action sequences of t_0 and t_1.

B.5.Define k such that $X_1, \ldots, X_{k-1}$ are the symbols shifted by $\boldsymbol{a}^-$.

B.6.For all states I_0 such that shift$X$ is defined:

B.6.1. Set $I_1 := I_0^{\boldsymbol{a}^-}$.

B.6.2. Denote by L the set of possible inputs in state $I_0^{\text{shift}X}$.

B.6.3. If both a_0 and a_1 are reductions then:

B.6.3.1. For all input symbols $Y \in \mathit{FirstL}(X_k \cdots X_p, L)$, set $P := P \cup \{((I_1, Y), a_0, a_1)\}$.

B.6.4. If $a_\imath = \text{shift}X_k$, where either $\imath = 0$, or $\imath = 1$, then:

B.6.4.1. Denote by $\boldsymbol{c}$ the largest prefix of $a_{\bar\imath}\boldsymbol{a}_{\bar\imath}^+$ that does not contain any shifts.

B.6.4.2. For all non-empty prefix $\boldsymbol{d}$ of $\boldsymbol{c}$ such that in state $\mathit{INIT}(X_k)^{\boldsymbol{d}}$ it is possible to accept:

B.6.4.2.1. Denote by v the parse tree with action sequence $\boldsymbol{d}$(accept).

B.6.4.2.2. For $\imath = 0, 1$, denote by $t_\imath^v$ the result of rooting v in the kth non-ε leaf in $t_\imath$.

B.6.4.2.3. Set $P := P \cup P(t_0^v, t_1^v)$.

B.6.4.3. Construct the set S which is the closure of $\{(I_1, \mathit{INIT}(X_k))\}$ under the condition that if (I, J) belongs to S and $J^{\boldsymbol{c}}$ is defined then $(I^{\boldsymbol{c}}, J^{\boldsymbol{c}})$ belongs to S.

B.6.4.4. For all $I, J, \boldsymbol{d}$ such that $(I, J) \in S$ and $\boldsymbol{d}$ is a proper prefix of $\boldsymbol{c}$ with $J^{\boldsymbol{d}}$ defined:

B.6.4.4.1. Denote by a the action in $\boldsymbol{c}$ following $\boldsymbol{d}$.

B.6.4.4.2. For all Y, b such that $Y \neq \$$, $b \notin \{a, \text{accept}\}$, and in state $J^{\boldsymbol{d}}$ on input Y the action b is possible:

B.6.4.4.2.1. Set $a'_\imath := b$, and $a'_{\bar\imath} := a$.

B.6.4.4.2.2. Set $P := P \cup \{((I^{\boldsymbol{d}}, Y), a'_0, a'_1)\}$.

B.6.4.4.3. For all Y, b such that $Y \in \mathit{FirstL}(X_{k+1} \cdots X_p, L)$, $b \notin \{a, \text{accept}\}$, and in state $J^{\boldsymbol{d}}$ on input \$ the action b is possible:

B.6.4.4.3.1. Set $a'_\imath := b$, and $a'_{\bar\imath} := a$.

B.6.4.4.3.2. Set $P := P \cup \{((I^{\boldsymbol{d}}, Y), a'_0, a'_1)\}$.

B.6.5. If $a_\imath = \text{accept}$, where either $\imath = 0$, or $\imath = 1$, then:

B.6.5.1. Denote by $\boldsymbol{c}$ the prefix of $a_{\bar\imath}\boldsymbol{a}_{\bar\imath}^+$ without the accept.

B.6.5.2. For all $\boldsymbol{d}$ such that $\boldsymbol{d}$ is a proper prefix of $\boldsymbol{c}$:

B.6.5.2.1. Denote by a the action in $\boldsymbol{c}$ following $\boldsymbol{d}$.

B.6.5.2.2. For all Y, b such that $Y \in L$, $b \neq a$, and b is possible in state $I_0^{(\text{shift}X)\boldsymbol{d}}$ on input Y:

B.6.5.2.2.1. Set $a'_\imath := b$, and $a'_{\bar\imath} := a$.

B.6.5.2.2.2. Set $P := P \cup \{((I_0^{(\text{shift}X)\boldsymbol{d}}, Y), a'_0, a'_1)\}$.

B.7.Return P.

We will now settle Theorem D.1 by showing the correctness of Algorithm B.

LEMMA D.2. *Algorithm B always terminate.*

PROOF. All the iterations are easily bounded, so the only critical step is the recursion at step B.6.4.2.3. By definition in step B.6.4.1 the action sequence $\boldsymbol{c}$ contains no shifts, and this property is inherited by $\mathbf{d}$ in step B.6.4.2. Hence, by definition, in step B.6.4.2.1, the parse tree v has an empty frontier, implying that the length of the frontier of t_0' and t_1' is one less than that of t_0 and t_1. Thus, the total depth of the recursion caused by step B.6.4.2.3 cannot be bigger than the length of the frontier of t_0 and t_1. We may therefore conclude that Algorithm B always terminates. □

LEMMA D.3. *Suppose the parse tree pair (t_0', t_1') follows by substitution from the parse tree pair (t_0, t_1). Then the projection of (t_0', t_1') is included in the set of projections returned by Algorithm B.*

PROOF. The result is trivial if $t_0 = t_1$, for then also $t_0' = t_1'$, so the projection of (t_0', t_1') is $\perp$, which is returned in step B.1 Thus we may assume that $t_0 \neq t_1$. Notice that this does not in general imply $t_0' \neq t_1'$. However, if $t_0' \neq t_1'$ and $\boldsymbol{a}'^-$ is the longest common prefix of the action sequences of t_0' and t_1', then $\text{state}(t_0', |\boldsymbol{a}'^-|) = \text{state}(t_1', |\boldsymbol{a}'^-|)$ and $\text{input}(t_0', |\boldsymbol{a}'^-|) = \text{input}(t_1', |\boldsymbol{a}'^-|)$, and then

$$((\text{state}(t_0', |\boldsymbol{a}'^-|), \text{input}(t_0', |\boldsymbol{a}'^-|)), \text{action}(t_0', |\boldsymbol{a}'^-|), \text{action}(t_1', |\boldsymbol{a}'^-|))$$

is the projection of (t_0', t_1').

Let $X, X_1, \ldots, X_p, \boldsymbol{a}^-, a_0, a_1, \boldsymbol{a}_0^+, \boldsymbol{a}_1^+, k$ be as defined in steps B.3–B.5. By the definition of substitution (page 11) there are parse trees $u, u_1, \ldots, u_p$ and a number l such that for $\imath = 0, 1$, the parse tree $t_\imath'$ results from $t_\imath$ by first, for $i = 1, \ldots, p$, rooting u_i in the ith non-ε leaf of $t_\imath$, and subsequently rooting the obtained parse tree in the lth non-ε leaf of u. Then X is the label of this lth leaf of u, and then X_i is the root symbol of u_i. Notice, that $u_1, \ldots, u_{k-1}$ cannot affect the projection of (t_0', t_1'), so without loss of generality, we may assume, for $i = 0, \ldots, k-1$, that $u_i = \text{p}\underline{\text{t}}(X_i)$.

Let the action sequences $\boldsymbol{b}^-, \boldsymbol{b}^+$ be defined such that $\boldsymbol{b}^-(\text{shift}X)\boldsymbol{b}^+$ is the action sequence of u with the shift$X$ being the lth shift. Similarly, let $\beta^- X \beta^+$ be the frontier of u. For $i = k, \ldots p$, denote by $\boldsymbol{b}_i$ the action sequence of u_i without the terminating accept. Moreover, denote by β_i the frontier of u_i. Thus, for $\imath = 0, 1$, if $\boldsymbol{a}_{\imath,0}(\text{shift}X_1)\boldsymbol{a}_{\imath,1}(\text{shift}X_2)\cdots(\text{shift}X_p)\boldsymbol{a}_{\imath,p}(\text{accept})$ is the action sequence of $t_\imath$, then

$$\boldsymbol{b}^- \boldsymbol{a}_{\imath,0}(\text{shift}X_1)\boldsymbol{a}_{\imath,1}(\text{shift}X_2)\boldsymbol{a}_{\imath,2}\cdots(\text{shift}X_{k-1})\boldsymbol{a}_{\imath,k-1}\boldsymbol{b}_k\boldsymbol{a}_{\imath,k}\cdots\boldsymbol{b}_p\boldsymbol{a}_{\imath,p}\boldsymbol{b}^+$$

is the action sequence of $t_\imath'$; and, similarly, $\beta^- X_1 \cdots X_{k-1}\beta_k \cdots \beta_p \beta^+$ is the frontier of $t_\imath'$.

Observe that $I_0 := \text{state}(u, |\boldsymbol{b}^-|)$ must be one of the instantiations of I_0 in step B.6, for u forces $\text{shift}X = \text{action}(u, |\boldsymbol{b}^-|)$ to be possible in $\text{state}(u, |\boldsymbol{b}^-|)$. In the remainder of the proof, we focus on the iteration with $I_0 = \text{state}(u, |\boldsymbol{b}^-|)$, and show that the projection of (t_0', t_1') is found within this iteration.

Let I_1 and L be as defined in step B.6.1 and B.6.2. As a consequence of our choice of I_0 relative to u, we have that u forces the first symbol Z of $\beta^+\$$ to be in L. Following the algorithm, the rest of the proof divides into cases depending on a_0 and a_1.

Both a_0 and a_1 are reductions (step B.6.3): Set $\boldsymbol{a}'^- = \boldsymbol{b}^-\boldsymbol{a}^-$. Clearly $\boldsymbol{a}'^-$ is the longest common prefix of the action sequences of t'_0 and t'_1. Then state$(t'_0, |\boldsymbol{a}'^-|)$ $= I_0^{\boldsymbol{a}^-} = I_1$, and action$(t'_\imath, |\boldsymbol{a}'^-|) = a_\imath$ for $\imath = 0, 1$. Moreover, input$(t'_0, |\boldsymbol{a}'^-|)$ is the first symbol of $\beta_k \cdots \beta_p \beta^+\$$ and hence of $\beta_k \cdots \beta_p Z$, but $Z \in L$, so input$(t'_0, |\boldsymbol{a}'^-|) \in \mathit{FirstL}(X_k \cdots X_p, L)$. Thus the projection of (t'_0, t'_1) is found in step B.6.3.1.

$a_\imath = \text{shift}X_k$, where either $\imath = 0$, or $\imath = 1$ (step B.6.4): Then $a_{\bar\imath}$ must be a reduction. Let $\boldsymbol{c}$ be the prefix of $a_{\bar\imath}\mathbf{a}^+_{\bar\imath}$ defined in step B.6.4.1. Thus, shiftX_k is the action following $\boldsymbol{c}$ in $a_{\bar\imath}\boldsymbol{a}^+_{\bar\imath}$.

$\boldsymbol{b}_k$ is a prefix of $\boldsymbol{c}\,\boldsymbol{b}_k$: Then $\boldsymbol{b}_k$ is a prefix of $\boldsymbol{c}^\infty$, so there is a unique $n \geq 0$ and non-empty prefix $\boldsymbol{d}$ of $\boldsymbol{c}$ such that $\boldsymbol{b}_k = \boldsymbol{c}^n\,\boldsymbol{d}$. We claim that $n = 0$. Recall, that $\boldsymbol{b}_k$ (accept) is the action sequence of the parse tree u_k. This implies that the overall effect of $\boldsymbol{b}_k$ on the stack is to push the symbol X_k. At the same time, it implies that both $\boldsymbol{c}$ and $\boldsymbol{d}$ are prefix of the action sequence of some parse tree, but any such prefix pushes at least one symbol onto the stack. Thus we cannot have $n > 0$, so $\boldsymbol{b}_k = \boldsymbol{d}$.

Now $\boldsymbol{d}$ is a non-empty prefix of $\boldsymbol{c}$ and since $\boldsymbol{d} = \boldsymbol{b}_k$, in state $\mathit{INIT}(X_k)^{\boldsymbol{d}}$, it is possible to accept, so $\boldsymbol{d}$ has the same value as in one of the iterations in step B.6.4.2. With this instantiation of $\boldsymbol{d}$, in step B.6.4.2.1, we get $v = u_k$. Now, with $t^v_\imath$ as defined in step B.6.4.2.2, it is clear that (t'_0, t'_1) follows from (t^v_0, t^v_1) by substitution. Recall from the proof of Lemma D.2 that the frontier of $t^v_\imath$ is one shorter than that of $t_\imath$. Thus, inductively we may conclude that the projection of (t'_0, t'_1) is found in step B.6.4.2.3.

$\boldsymbol{b}_k$ is not a prefix of $\boldsymbol{c}\,\boldsymbol{b}_k$: Let the action sequence $\boldsymbol{x}$ and the actions a and b, where $a \neq b$, be defined such that $\boldsymbol{x}a$ is a prefix of $\boldsymbol{c}\,\boldsymbol{b}_k$ and $\boldsymbol{x}\,b$ is a prefix of $\boldsymbol{b}_k$. Set $\boldsymbol{a}'^- = \boldsymbol{b}^-\boldsymbol{a}^-\boldsymbol{x}$. Thus $\boldsymbol{a}'^-$ is the longest common prefix of t'_0 and t'_1, and then action$(t'_\imath, |\boldsymbol{a}'^-|) = b$ and action$(t'_{\bar\imath}, |\boldsymbol{a}'^-|) = a$.

Clearly $\boldsymbol{x}$ is a prefix of $\boldsymbol{c}^\infty$, so there is a unique $n \geq 0$ and proper prefix $\boldsymbol{d}$ of $\boldsymbol{c}$ such that $\boldsymbol{x} = \boldsymbol{c}^n\,\boldsymbol{d}$. Set

$$(I, J) = ((\cdots(I_1 \overbrace{^{\boldsymbol{c}})\cdots)^{\boldsymbol{c}}}^{n}, (\cdots(\mathit{INIT}(X_k) \overbrace{^{\boldsymbol{c}})\cdots)^{\boldsymbol{c}}}^{n})$$

This is well-defined since $\boldsymbol{c}$ (if $n > 0$) and $\boldsymbol{d}$ are prefix of the action sequence $\boldsymbol{b}_k$ of u_k. With the set S as defined in step B.6.4.3, we have $(I, J) \in S$, so all of $I, J, \boldsymbol{d}$ have values as in some iteration of step B.6.4.4. Notice that also a has the value that it gets assigned in this iteration in step B.6.4.4.1.

The frontier β_k of u_k is non-empty: Denote by Y the first symbol in β_k. Thus Y = input$(t'_0,|\boldsymbol{a}'^-|)$. Set b = action$(u_k, |\boldsymbol{x}|)$. Clearly b = action$(u_k, |\boldsymbol{x}|)$ is possible in state $J^{\boldsymbol{d}} = \mathit{INIT}(X_k)^{\boldsymbol{x}}$ = state$(u_k, |\boldsymbol{x}|)$ on input Y = input$(u_k, |\boldsymbol{x}|)$. Thus b and Y are instantiated as in some iteration of step B.6.4.4.2, and hence the projection of (t'_0, t'_1) is found in steps B.6.4.4.2.1–B.6.4.4.2.2.

The frontier β_k of u_k is empty: Denote by Y the first symbol of $\beta_{k+1} \cdots \beta_p$ $\beta^+\$$. Thus Y = input$(t'_0, |\boldsymbol{a}'^-|)$. Moreover, we then have $Y \in \mathit{FirstL}($ $X_{k+1} \cdots X_p, L)$ since L contains the first symbol Z in $\beta^+\$$. Set b = action$(u_k, |\boldsymbol{x}|)$. Clearly b is possible in state $J^{\boldsymbol{d}}$ = state$(u_k, |\boldsymbol{x}|)$ on input $\$$ = input$(u_k, |\boldsymbol{x}|)$. Thus b and Y are instantiated as in some iteration of step B.6.4.4.3, and hence the projection of (t'_0, t'_1) is found in steps B.6.4.4.3.1–B.6.4.4.3.2.

$a_\imath =$ accept, where either $\imath = 0$, or $\imath = 1$ (step B.6.5): Then $a_{\bar\imath}$ must be a reduction, and moreover, we must have $I_0^{\boldsymbol{a}^-} = I_0^{\text{shift}X}$. Let $\boldsymbol{c}$ be as defined in step B.6.5.1. Thus $\boldsymbol{c}(\text{accept}) = a_{\bar\imath}\boldsymbol{a}_{\bar\imath}^+$. Let the action sequence $\boldsymbol{x}$ and the actions a and b, where $a \neq b$, be defined such that $\boldsymbol{x}a$ is a prefix of $\boldsymbol{c}\,\boldsymbol{b}^+$ and $\boldsymbol{x}\,b$ is a prefix of $\boldsymbol{b}^+$. Set $\boldsymbol{a}'^- = \boldsymbol{b}^-\boldsymbol{a}^-\boldsymbol{x}$. Thus $\boldsymbol{a}'^-$ is the longest common prefix of t'_0 and t'_1, and then $\text{action}(t'_\imath, |\boldsymbol{a}'^-|) = b$ and $\text{action}(t'_{\bar\imath}, |\boldsymbol{a}'^-|) = a$. Clearly $\boldsymbol{x}$ is a prefix of $\boldsymbol{c}^\infty$, so there is a unique $n \geq 0$ and proper prefix $\boldsymbol{d}$ of $\boldsymbol{c}$ such that $\boldsymbol{x} = \boldsymbol{c}^n\,\boldsymbol{d}$.

Notice that $(\text{shift}X)\boldsymbol{c}(\text{accept})$ constructs a parse tree generating $X \to^* X$, and hence so does $(\text{shift}X)\boldsymbol{c}^n(\text{accept})$. Thus we have

$$\text{state}(t'_0, |\boldsymbol{a}'^-|) = I_0^{\boldsymbol{a}^-\boldsymbol{x}} = I_0^{(\text{shift}X)\boldsymbol{c}^n\boldsymbol{d}} = I_0^{(\text{shift}X)\boldsymbol{d}}.$$

Let Y be the first symbol in $\beta^+\$$. Then $\text{input}(t'_0, |\boldsymbol{a}'^-|) = Y$, and, moreover, $Y \in L$. The parse tree u forces b to be possible in state $I_0^{(\text{shift}X)\boldsymbol{d}}$ on input Y, since these are the action, the state, and the input after the initial action sequence $\boldsymbol{b}^-(\text{shift}X)\boldsymbol{x}$ has been carried out. Thus b and Y are instantiated as in some iteration of step B.6.5.2.2, so the projection of (t'_0, t'_1) is found in steps B.6.5.2.2.1–B.6.5.2.2.2.

□

LEMMA D.4. *For all p in the set P returned by Algorithm B, there is parse tree pair (t'_0, t'_1) that follows from (t_0, t_1) by substitution and which projects to p.*

PROOF. Using the completability and no-junk properties of universal CLR(1) parsers, it is straightforward, but tidious, to verify that the proof of Lemma D.3 can be reversed, so that for any projection p, we find parse trees $u, u_k, \cdots, u_p$ giving us a parse tree pair (t'_0, t'_1) which projects to p. □

PROOF OF THEOREM D.1. By Lemmas D.2, D.3, and D.4, Algorithm B always terminates finding the set of projections of all the parse tree pairs following by substitution from the input parse tree pair. Thus with Algorithm A and Algorithm B together, given a universal CLR(1) parser $\mathcal{U}$ and a finite set $\mathcal{E}$ of parse tree pairs, we can decide if $\mathcal{E}^\prec$ spans an ordering of $\mathcal{U}$, and if so, construct this ordering. □

Concerning our example with grammar (4), for the parse tree pairs in $\mathcal{E}_{(4)}$ we get

$$P\left(E\ ,\ \begin{array}{c} E \\ | \\ F \\ | \\ E \end{array} \right) = \{\ ((2,\$), \text{accept}, \text{reduce}\ F \to E)\ \},$$

$$
P\left(F\ ,\ \begin{array}{c} F \\ | \\ E \\ | \\ F \end{array} \right) = \left\{ \begin{array}{l}
((3,E), \text{shift}E, \text{reduce } E{\to}F) \\
((3,F), \text{shift}F, \text{reduce } E{\to}F) \\
((5,E), \text{shift}E, \text{reduce } E{\to}F) \\
((5,E), \text{reduce } F{\to}FF, \text{reduce } E{\to}F) \\
((5,F), \text{shift}F, \text{reduce } E{\to}F) \\
((5,F), \text{reduce } F{\to}FF, \text{reduce } E{\to}F) \\
((5,\$), \text{reduce } F{\to}FF, \text{reduce } E{\to}F) \\
((7,E), \text{shift}E, \text{reduce } E{\to}F) \\
((7,F), \text{shift}F, \text{reduce } E{\to}F) \\
((7,\$), \text{accept}, \text{reduce } E{\to}F)
\end{array} \right\} \text{ and}
$$

$$
P\left(\begin{array}{l} F \\ |\backslash \\ FF \\ \ \ |\backslash \\ \ \ FF \end{array} \ ,\ \begin{array}{l} F \\ |\backslash \\ F\ F \\ \ \ |\backslash \\ \ \ FF \end{array} \right) = \left\{ \begin{array}{l}
((5,E), \text{shift}E, \text{reduce } F{\to}FF) \\
((5,F), \text{shift}F, \text{reduce } F{\to}FF)
\end{array} \right\}.
$$

APPENDIX E. TRACING THE MINIMAL NON-CANONICAL PARSE TREES

This appendix addresses problem 2, page 21. As mentioned in Section 8, we will not solve the problem exactly as stated. However, when we come to Appendix H, where the concrete description of the algorithm generalizing the canonical LR(1) technique is given, it will be clear that the results of this appendix give a sufficient substitution for an exact solution to problem 2.

A set of parse trees is said to be *stable* if it contains all sub-parse trees of parse trees in the set. If the set of canonical parse trees of some parser is stable, we say that the parser itself is *stable*. Notice that if a set $\mathcal{E}$ of parse tree pairs prioritizes a parser $\mathcal{P}$, then the $\mathcal{P}$-canonical parse trees are exactly the parse trees that have no sub-parse tree among the second coordinates in $\mathcal{E}$. Hence any prioritized parser must be stable.

More locally, if for some canonical parse tree, all sub-parse trees are canonical, then we say that this parse tree is *stable canonical*. Thus for a stable parser all canonical parse trees are stable canonical. This simple observation is very useful when looking for prioritized and hence stable parsers.

A parser $\mathcal{P}$ is said to be *basis complete* if all trivial parse trees $\text{p}\underline{\text{t}}(X)$ are canonical. Notice that checking for basis completeness is easy. Also notice that if $\mathcal{P}$ is not basis complete, it means that there is a grammar symbol X for which the trivial parse tree $\text{p}\underline{\text{t}}(X)$ is not canonical, but this implies that there cannot be any stable canonical parse tree for $X{\to}^*X$. Thus, if a parser $\mathcal{P}$ is not basis complete, it cannot be prioritized at all.

The aim for this appendix is to show the following result which is our substitution for an exact solution to problem 2:

THEOREM E.1. *Given a basis complete determinization of the universal CLR(1) parser, we can represent a set M of parse trees satisfying the following conditions:*

—*M contains all minimal non-canonical parse trees*

—*If the determinization is stable then M is exactly the set of minimal non-canonical parse trees.*

—*It is decidable whether M is infinite.*

We are going to give a concrete construction for such a set M, given any basis complete determinization. Notice, that basis completeness means that all non-canonical parse trees are non-trivial.

Some technical definitions are needed. Given a parse tree, by a *leaf rightside* we understand a sequence of leaves corresponding to the rightside of some production. More precisely, $l_1, \ldots, l_p$ is a leaf rightside if it is the sequence of sons of some internal node and all of $l_1, \cdots, l_p$ are leaves. In particular, any ε-leaf constitutes a leaf rightside. Notice that any non-trivial parse tree has a leaf rightside, and that if we remove a leaf rightside of a parse tree, the result is a parse tree. Repeating this "pruning" process, eventually we will arrive at the trivial parse tree consisting of the root symbol.

A parse tree is said to be *good* if it is non-canonical, but removal of any leaf rightside results in a canonical parse tree. Trivially, all minimal non-canonical parse trees are good. Our first step in constructing M will be to introduce traces. Traces are special kinds of partial parsings. The successful traces construct exactly the good parse trees. Loosely speaking, the actions of a successful trace will correspond to the suffix of the actions sequence of a good parse tree starting from the first leaf rightside. All preceding actions must be shifts, so this suffix determines the good parse tree uniquely.

From the traces we will construct a finite trace graph representing M. The directed walks (possibly looping dipaths) in this graph will correspond to traces. The graph will contain a source vertex and some terminal vertices, and any directed walk from the source to a terminal vertex will correspond to a successful trace for a parse tree in M. Thus M is infinite if and only if is there a loop in a directed walk from a source vertex to some terminal vertex.

We will now go ahead with the formal definition of *traces*. All further motivation is deferred to the following lemmas and their proofs. An initial *trace configuration* is an initial partial configuration. All other trace configurations are triples (s, G, B) where s is a partial configuration, where either G is the symbol '$*$', or G is a set of input symbols, and where B is a set of input symbols. We will refer to G and B as the *goal* and *block*.

Trace actions are grammar productions with a dot somewhere in the rightside, like the items from Appendix B. To a trace action $X \rightarrow \alpha \bullet \beta$ corresponds the sequence of normal actions which first shifts β and then reduce $X \rightarrow \alpha\beta$. This sequence of normal actions is denoted $\overline{\mathrm{a}}(X \rightarrow \alpha \bullet \beta)$.

From an initial trace configuration $(\boxed{I})$, by $X \rightarrow \bullet\, \beta$ we can get to the trace configuration

$$((\boxed{I}\, X \boxed{GOTO(I, X)}), G, B)$$

if the following conditions are satisfied:

(i) From partial configuration $(\boxed{I})$, it is possible to carry out $\overline{\mathrm{a}}(X \rightarrow \bullet\, \beta)$ arriving at the partial configuration $(\boxed{I}\, X \boxed{GOTO(I, X)})$.

(ii) From state I, it is canonical to shift X.

(iii) If, by shifting β from state I, we encounter a non-canonical shift, then $G = *$ and $B = \emptyset$; otherwise, both G and B equal the set of input symbols Y such that in state $GOTO(I, \beta)$ on input Y, it is possible, but not canonical, to reduce

$X \rightarrow \beta$.

From the trace configuration (s, G, B), by $X \rightarrow \alpha \bullet \beta$ we can get to the trace configuration (s', G', B') if the following conditions are satisfied:

(iv) If $G = *$, then $\alpha \neq \varepsilon$.
(v) If $G = *$ and $\beta = Y\gamma$, then $Y \notin B$.
(vi) If $G \neq *$ and $\beta = Y\gamma$, then $Y \in G$.
(vii) If $G \neq *$ and $\alpha = \varepsilon$, then $X \notin B$.
(viii) From partial configuration s, it is canonical to shift β thus arriving at a partial configuration $(\boxed{I_0} X_1 \cdots X_n \boxed{I_n})$.
(ix) In state I_n on some input Y it is possible to reduce $X \rightarrow \alpha\beta$. Denote by C is the set of input symbols Y on which in state I it is possible but not canonical to reduce $X \rightarrow \alpha\beta$.
(x) Set $p = |\alpha\beta|$. If $n \geq p$ then $s' = (\boxed{I_0} X_1 \cdots X_{n-p} \boxed{I_{n-p}} X \boxed{GOTO(I_{n-p}, X)})$; otherwise, $s' = (\boxed{I} X \boxed{GOTO(I, X)})$, where, from state I, canonically we can shift the first $n - p$ symbols of α thus arriving at state I_0. In this case we say that $X \rightarrow \alpha \bullet \beta$ *transcends* c with these first $n - p$ symbols of α.
(xi) If $G \neq *$ and $\alpha = \varepsilon$, then, from partial configuration c, it is canonical to shift X.
(xii) If $G = *$, then $G' = *$.
(xiii) If $G \neq *$ and $\beta = \varepsilon$, then $G' = G \setminus C$.
(xiv) If $G \neq *$ and $\beta = Y\gamma$, then $G' = *$.
(xv) If $\beta = \varepsilon$, then $B' = B \cup C$; otherwise $B' = C$.

A trace configuration (c, G, B) is *accepting* if the following condition is satisfied:

(xvi) In the state of c on input \$ it is canonical to accept, and, moreover, either $\$ \in G$, or $G = *$ and $\$ \notin B$.

A *trace* is a sequence of the form $c_0\, A_0\, c_1 \cdots A_{n-1}\, c_n$ where $c_0, \ldots, c_n$ are trace configuration, where $A_0, \ldots, A_{n-1}$ are trace actions, where c_0 is initial, and where for $i = 0, \ldots, n-1$, we can get from c_i to c_{i+1} by A_i. If, moreover, c_n is accepting, we say the trace is *successful*.

We associate with a successful trace $T = c_0\, A_0 \cdots A_{n-1}\, c_n$ the action sequence

$$(\text{shift}\, \delta_{n-1} \cdots \delta_0)\, \overline{\text{a}}(A_0) \cdots \overline{\text{a}}(A_{n-1})\, (\text{accept})$$

where δ_i is the sequence of grammar symbols with which a_i transcends c_i if A_i transcends c_i (cf. condition (x)); otherwise $\delta_i = \varepsilon$. From conditions (i), (viii)–(x) it follows that this action sequence is the action sequence of a successful parsing. In fact, in this parsing, the state from which we carry out the first action in $\overline{\text{a}}(A_i)$ is exactly the state of the partial configuration in c_i. In particular, if $c_0 = (\boxed{I_0})$ and $A_{n-1} = X_{n-1} \rightarrow \alpha_{n-1} \bullet \beta_{n-1}$, then $I_0 = \mathit{INIT}(X_{n-1})^{\text{shift}\, \delta_{n-1} \cdots \delta_0}$. We say that our successful trace *constructs* the parse tree produced by this parsing.

The following statement indicates—with no clue to a finite representation—the relationship between traces and the set M from Theorem E.1:

PROPOSITION E.2. *Denote by M the set of parse trees t satisfying that t is constructed by a successful trace T and that there is no other accepting trace T' such that the sequence of trace actions of T' is a proper prefix of the sequence of trace actions of T. Then M contains all minimal non-canonical parse trees.*

If the set of canonical parse trees is stable, then M contains exactly the minimal non-canonical parse trees.

Several lemmas are needed before we can settle the proposition. The first two state that a parse tree is constructed by a successful trace if and only if it is good.

LEMMA E.3. *If a trace is successful, it constructs a good parse tree.*

PROOF. Let T be a successful trace, and let t be the parse tree it constructs. We want to show that t is good, i.e. that t is non-canonical, but that if we remove any leaf rightside from t the results is a canonical parse tree.

Write T in the form $(s_0)(X_0 \rightarrow \alpha_0 \bullet \beta_0)(s_1, G_1, B_1) \cdots (X_{n-1} \rightarrow \alpha_{n-1} \bullet \beta_{n-1})(s_n, G_n, B_n)$. Then the action sequence $\boldsymbol{a}$ of t is of the from $(\text{shift}\,\delta)\,\overline{\text{a}}(X_0 \rightarrow \alpha_0 \bullet \beta_0) \cdots \overline{\text{a}}(X_{n-1} \rightarrow \alpha_{n-1} \bullet \beta_{n-1})\,(\text{accept})$.

Notice that the trace action $X_i \rightarrow \alpha_i \bullet \beta_i$ corresponds to a leaf rightside labeled $\alpha_i \beta_i$ if and only if $\alpha_i = \varepsilon$ (recall that $\alpha_0 = \varepsilon$ by definition). Thus to show that t is good we need to show the following:

(a) $\boldsymbol{a}$ contains a non-canonical action.

(b) For any i with $\alpha_i = \varepsilon$, replacing $\overline{\text{a}}(X_i \rightarrow \bullet\, \beta_i)$ with $(\text{shift} X_i)$ in $\boldsymbol{a}$ gives a canonical action sequence.

The proof divides into cases:

shift β_0 contains a non-canonical shift: Clearly (a) is satisfied. By condition (iii) we have $G_1 = *$, and hence by condition (xii) we have $G_i = *$ for all $i > 0$. By condition (iv), this implies that $\alpha_i \neq \varepsilon$ for all $i > 0$. Thus, in order to verify (b), it satisfies to show that $\boldsymbol{a}^0 = (\text{shift}\,\delta)\,(\text{shift}\,X_0)\,\overline{\text{a}}(X_1 \rightarrow \alpha_1 \bullet \beta_1) \cdots \overline{\text{a}}(X_{n-1} \rightarrow \alpha_{n-1} \bullet \beta_{n-1})\,(\text{accept})$ is canonical.
Clearly $(\text{shift}\,\delta)$ is canonical by condition (x), and $(\text{shift}\,X_0)$ is canonical by condition (ii). Moreover, condition (viii) guarantees that all other shifts in $\boldsymbol{a}^0$ are canonical, and condition (xvi) states that the final accept is canonical. Thus the only remaining concern is if for some $i > 0$, the reduction with $X_i \rightarrow \alpha_i \beta_i$ is non-canonical. By conditions (ix) and (xv), this can only be the case if the input symbol for this reduction is contained in B_{i+1}, but this possibility is excluded by conditions (v) and (xvi). Thus we may conclude that (b) is satisfied, and hence that, indeed, t is a good parse tree.

shift β_0 does not contain a non-canonical shift: We settle (a) by showing that the reduction with $X_0 \rightarrow \beta_0$ is non-canonical. By condition (iii), this is the case if the input symbol for this reduction is contained in G_1, but this is guaranteed by conditions (vi) and (xiii). Thus (a) holds.
Our first step in verifying (b) is to show that $\boldsymbol{a}^0 = (\text{shift}\,\delta)\,(\text{shift}\,X_0)\,\overline{\text{a}}(X_1 \rightarrow \alpha_1 \bullet \beta_1) \cdots \overline{\text{a}}(X_{n-1} \rightarrow \alpha_{n-1} \bullet \beta_{n-1})\,(\text{accept})$ is canonical. As in the last case, it is easy to see that all the shifts and the final accept are canonical. Again the problem is if, for some $i > 0$, the reduction with $X_i \rightarrow \alpha_i \beta_i$ is non-canonical. As in condition (viii), denote by C the set of input symbols for which this reduction

is non-canonical. Moreover, let Y be the actual input symbol for the reduction. Thus we want to show that $Y \notin C$.

By conditions (xii)–(xv) we have either $G_{i+1} \neq *$ and $G_{i+1} \cap C = \emptyset$, or $G_{i+1} = *$ and $B_{i+1} \supseteq C$. In the former case conditions (vi) and (xiii) ensure that Y is in G_{i+1}, hence not in C, and in the latter case conditions (v) and (xv) ensure that Y is not in B_{i+1}, hence not in C. Thus Y is outside C, as desired, so $\boldsymbol{a}^0$ is indeed canonical.

It remains to be shown that if, for some $k > 0$, we have $\alpha_i = \varepsilon$, then $\boldsymbol{a}^k = (\text{shift}\,\delta)\overline{\text{a}}(X_0 \to \alpha_0 \bullet \beta_0) \cdots \overline{\text{a}}(X_{k-1} \to \alpha_{k-1} \bullet \beta_{k-1})\,(\text{shift}\,X_k)\,\overline{\text{a}}(X_{k+1} \to \alpha_{k+1} \bullet \beta_{k+1}) \cdots \overline{\text{a}}(X_{n-1} \to \alpha_{n-1} \bullet \beta_{n-1})\,(\text{accept})$ is canonical. The final accept and all shifts but shiftX_k are shown to be canonical as above. Moreover, shiftX_k is canonical by condition (xi). By condition (iv) we have $G_k \neq *$, and hence by condition (xii), for $i = 1, \ldots, k$, we have $G_i \neq *$. Thus by condition (xiv), for $i = 1, \ldots, k-1$, we have $\beta_i = \varepsilon$. So X_k is the input symbol for all the preceding reductions. Now, by conditions (iii), (ix), and (xv), the set B_{k-1} contains all input symbols for which any of the preceding reductions are non-canonical, but by condition (vii), we do not have X in B_{k-1}. Thus all the reductions preceding shiftX_k are canonical. Concerning the reductions succeeding shiftX_k, they can all be shown to be canonical using the same argument as was applied to the reductions in $\boldsymbol{a}^0$, so we may conclude that $\boldsymbol{a}^k$ is canonical. Thus (b) holds, so, indeed, t is a good parse tree.

□

LEMMA E.4. *If a parse tree is good, it is constructed by a unique successful trace.*

PROOF. Let t be a good parse tree with action sequence $\boldsymbol{a}$. Suppose that $\boldsymbol{a}$ is of the form $\boldsymbol{a}_0\,\overline{\text{a}}(X_x \to \bullet \beta_x)\,\boldsymbol{a}_1\,\overline{\text{a}}(X_y \to \bullet \beta_y)\,\boldsymbol{a}_2$. Thus t has two leaf rightsides. We claim that all shifts in $\boldsymbol{a}$ are canonical, and that $\boldsymbol{a}_1$ contains no shifts. Since t is good, $\boldsymbol{a}$ is non-canonical, but for any $i \in \{x, y\}$, if we replace $\overline{\text{a}}(X_x \to \bullet\, \beta_x)$ with shift X_x the resulting action sequence $\boldsymbol{a}^x$ becomes canonical. Clearly, all non-canonical actions in $\boldsymbol{a}$ have to be contained in $\boldsymbol{a}_0\,\overline{\text{a}}(X_x \to \bullet\, \beta_x)$, for otherwise $\boldsymbol{a}^x$ would be non-canonical like $\boldsymbol{a}$. However, $\boldsymbol{a}_0\,\overline{\text{a}}(X_x \to \bullet\, \beta_x)$ cannot contain a non-canonical shift, for then $\boldsymbol{a}^y$ would be non-canonical like $\boldsymbol{a}$. Thus, $\boldsymbol{a}_0\,\overline{\text{a}}(X_x \to \bullet\, \beta_x)$ contains a non-canonical reduction. Clearly, replacing $\overline{\text{a}}(X_y \to \bullet\, \beta_y)$ with shiftX_y can only affect the input symbol of this non-canonical reduction—making it canonical—if there are no shifts in $\boldsymbol{a}_1$. Hence, if there are shifts in $\boldsymbol{a}_1$ then $\boldsymbol{a}^y$ is non-canonical like $\boldsymbol{a}$. Thus the claim follows.

As a first consequence of the claim, we have that for any two consecutive reductions in $\boldsymbol{a}$, if β is the sequence of symbols shifted between them, the second reduction is with a production $X \to \alpha\beta$, where possibly α is empty. In other words, β cannot contain any symbols that are not reduced by the second reduction. This implies that $\boldsymbol{a}$ is of the form $(\text{shift}\,\delta)\,\overline{\text{a}}(X_0 \to \alpha_0 \bullet \beta_0) \cdots \overline{\text{a}}(X_{n-1} \to \alpha_{n-1} \bullet \beta_{n-1})\,(\text{accept})$. From the fact that $\boldsymbol{a}$ can be written in this form, it follows that if we ignore all conditions but (i), (viii)–(x), and (vi), and, moreover, substitute "possible" for "canonical" in these conditions, then there is a unique trace

$$T = (s_0)(X_0 \to \alpha_0 \bullet \beta_0)\,(s_1, G_1, B_1) \cdots (X_{n-1} \to \alpha_{n-1} \bullet \beta_{n-1})(s_n, G_n, B_n)$$

constructing t. Here s_0 is the stack consisting of the state $\mathit{INIT}(X)^{\text{shift}\,\delta}$, and for

$i > 0$, we have that s_i is a suffix of the stack of the parsing starting in $\mathit{INIT}(X_{n-1})$ after having carried out the action sequence $(\text{shift}\,\delta)\,\overline{\text{a}}(X_0 \rightarrow \alpha_0 \bullet \beta_0) \cdots \overline{\text{a}}(X_{i-1} \rightarrow \alpha_{i-1} \bullet \beta_{i-1})$. The exact extent of the suffix is determined by conditions (i),(viii), and (x). All the G_is and B_i are determined by conditions (iii), and (xii)–(xv).

In order to complete the proof of the lemma, we need to show that T satisfies all the original conditions. Basically, this just amounts to reversing the argument from the proof of Lemma E.3, so we will restrict ourselves to an example: proving that condition (iv) is satisfied. Suppose that $\boldsymbol{a}$ contains a non-canonical shift. Then, by our claim, t has only one leaf rightside. Thus $\alpha_i = \varepsilon$ for all $i > 0$, so condition (iv) is always satisfied. Suppose, conversely, that $\boldsymbol{a}$ contains no non-canonical shifts. Then by condition (iii), we have $G_i \neq *$. If (iv) is false, for some $k > 0$, we have $G_k = *$ and $\alpha_k = \varepsilon$. By conditions (xiii) and (xiv) the former implies that for some i, where $0 < i < k$, we have $\beta_i \neq \varepsilon$. Thus there is a shift between the first leaf rightside and the leaf rightside corresponding to $\overline{\text{a}}(X_k \rightarrow \bullet\, \beta_k)$, contradicting our claim. Hence condition (iv) holds. □

LEMMA E.5. *Assume that the set of canonical parse trees is stable, and let t be a good parse tree. If t is not minimal non-canonical, there is only one of the sons of the root from which a non-trivial parse tree t^r descends, and then t^r is good.*

PROOF. Assume that t is not minimal non-canonical; then t has a proper sub-parse tree t^- which is non-canonical. The parse tree t^- must contain all leaf rightsides of t, for, if t^- did not contain a leaf rightside l, then t^- would be contained in the parse tree t^l which is t without l. Since t is good, t^l is canonical, and by assumption the set of canonical parse trees is stable, so all sub-parse trees of t^l are canonical. This contradicts that the sub-parse tree t^- is non-canonical. Thus, as claimed, t^- contains all leaf rightsides of t.

Since t has a proper sub-parse tree t^- containing all leaf rightsides of t, there can only be one non-trivial parse tree t^r descending from a son of the root of t, and then t^- must be a sub-parse tree of t^r.

It remains to be shown that t^r is good. First, since t^r contains t^-, which is non-canonical, by stability, t^r is itself non-canonical. Now, let l be any leaf rightside of t^r, and denote by t^{rl} the result of removing l from t^r. Similarly, denote by t^l the result of removing the leaf rightside l from t. Since t is good, t^l is canonical, and t^{rl} is contained in t^l, so by stability, t^{rl} is canonical. Thus t^r is good. □

PROOF OF PROPOSITION E.2. As in the proposition, denote by M the set of parse trees t such that t is constructed by a successful trace T_1 and there is no other successful trace T_2 such that the sequence of trace actions of T_2 is a proper prefix of the sequence of trace actions of T_1. We want to show the following:

(a) M contains all minimal non-canonical parse trees.

(b) If the set of canonical parse trees is stable, then M contains exactly the minimal non-canonical parse trees.

Let T_1 and T_2 be successful traces constructing parse trees t_1 and t_2 respectively. It is easy to check if the sequence of trace actions of T_1 is a proper prefix of the sequence of trace actions of T_2 then t_1 is a proper sub-parse tree of t_2. Now, all good parse trees are non-canonical and all minimal non-canonical parse trees are good, so (a) follows directly from Lemma E.3 and Lemma E.4 which together say that a parse tree is good if and only if it is constructed by a successful trace.

Assume that the set of canonical parse trees is stable, and let t be a good parse tree which is not minimal non-canonical. Moreover, let t^r be the good sub-parse tree provided by Lemma E.5. Thus t^r is the only non-trivial parse tree descending from a son of the root of t. By Lemma E.4, we have unique traces T and T^r such that T constructs t and T^r constructs t^r. The sequence of trace actions of T^r is the proper prefix of the sequence of trace actions of T obtained by removing the last trace action corresponding to the root production, so t is not in M. Thus (b) holds. □

The traces have another property which is important in finding a finite representation of our set M.

LEMMA E.6. *Only finitely many different trace configurations occur in traces.*

PROOF. We prove the result by bounding the number of states in the stacks of the partial configurations in the trace configurations of an arbitrary, possibly unsuccessful trace

$$(s_0)\,(X_0\to\bullet\beta_1)\,(s_1,G_1,B_1)\,(X_1\to\alpha_1\bullet\beta_1)\cdots(X_{n-1}\to\alpha_{n-1}\bullet\beta_{n-1})a_{n-1}\,(s_n,G_n,B_n).$$

Denote by $|s|$ the number of states in the stack of the partial configuration s. Then $|s_0| = 1$, and, for $i = 1,\ldots,n$, we have $|s_i| = \max(1, |s_{i-1}| + 1 - |\alpha_i|)$. Thus the result follows if we can bound the number of indices i for which $\alpha_i = \varepsilon$.

Denote by k the last index for which $\alpha_k = \varepsilon$. Thus, we only need consider indices strictly smaller than k. If $k = 0$ we are done, so assume that $k > 0$. Then by condition (iv), we have $G_k \neq *$, so by condition (xii), for all $i \leq k$, we have $G_i \neq *$. By condition (xiv), this implies for $i < k$, that $\beta_i = \varepsilon$. Thus, if $\alpha_i = \varepsilon$ for some $i < k$, then $a_i = X_i \to \bullet\,\varepsilon$.

The desired bound on the number of indices $i < k$ for which $\alpha_i = \varepsilon$ is now achieved by showing that there cannot exist indices i, j and a grammar symbol X such that $i < j < k$, and $a_i = a_j = X \to \bullet\,\varepsilon$; for suppose that i, j, and X satisfy this condition. By conditions (ii) and (xi), from s_i on input X the canonical action is to shift X, and hence it is non-canonical to reduce $X \to \varepsilon$. Thus, by conditions (ix) and (xv), we have $X \in B_{i+1}$. Recall that $\beta_l = \varepsilon$ for all $l < k$. By condition (xv), this implies $B_i \subseteq B_j$, so, in particular, $X \in B_j$. However, by condition (xi) this contradict $a_j = X \to \bullet\,\varepsilon$. □

Now, we are ready to construct the announced trace graph giving an adequate representation of M. The *trace graph* is the directed graph G generated by the following conditions:

—The vertices are sets of trace configurations, and the arcs are labeled with trace actions.

—There is a source vertex containing all initial trace configurations.

—A vertex is terminal if it includes an accepting trace configuration.

—Let v be a vertex in G that is not terminal and which includes a trace configuration c. Moreover, let a be a trace action and c' a trace configuration such that we can get from c to c' by a. Then there is a vertex v' in G consisting of all trace configurations c'_1 such that for some trace configuration c_1 in v, we can get from c_1 to c'_1 by a.

Fig. 2. Trace graph

$$
\begin{array}{l}
\qquad\qquad F\!\to\bullet FF \nearrow
\left\{\begin{array}{l}
(\boxed{1}F\boxed{3}, \{E,F\}, \{E,F\})\\
(\boxed{3}F\boxed{5}, \{E,F\}, \{E,F\})\\
(\boxed{5}F\boxed{5}, \{E,F\}, \{E,F\})\\
(\boxed{6}F\boxed{7}, \{E,F\}, \{E,F\})\\
(\boxed{7}F\boxed{5}, \{E,F\}, \{E,F\})
\end{array}\right\}
\xrightarrow{F\to F\bullet F}
\left\{\begin{array}{l}
(\boxed{1}F\boxed{3}, *, \{E,F\})\\
(\boxed{3}F\boxed{5}, *, \{E,F\})\\
(\boxed{5}F\boxed{5}, *, \{E,F\})\\
(\boxed{6}F\boxed{7}, *, \{E,F\})\\ \hline
(\boxed{7}F\boxed{5}, *, \{E,F\})
\end{array}\right\}
\\[2ex]
\left\{\begin{array}{l}
(\boxed{1})\\ (\boxed{2})\\ (\boxed{3})\\ (\boxed{4})\\ (\boxed{5})\\ (\boxed{6})\\ (\boxed{7})
\end{array}\right\}
\xrightarrow{F\to\bullet E}
\left\{ (\boxed{1}F\boxed{3}, \{\$\}, \{\$\}) \right\}
\xrightarrow{E\to F\bullet}
\left\{ \underline{(\boxed{1}E\boxed{2}, \{\$\}, \{\$,E,F\})} \right\}
\\[2ex]
\downarrow E\!\to\bullet F
\\
\left\{\begin{array}{l}
(\boxed{1}E\boxed{2}, \{E,F\}, \{E,F\})\\
(\boxed{3}E\boxed{4}, \{E,F,\$\}, \{E,F,\$\})\\
(\boxed{5}E\boxed{4}, \{E,F,\$\}, \{E,F,\$\})\\
(\boxed{6}E\boxed{4}, \{E,F,\$\}, \{E,F,\$\})\\
(\boxed{7}E\boxed{4}, \{E,F,\$\}, \{E,F,\$\})
\end{array}\right\}
\xrightarrow{F\to E\bullet}
\left\{\begin{array}{l}
(\boxed{1}F\boxed{3}, \{E,F\}, \{E,F\})\\
(\boxed{3}F\boxed{5}, \{E,F,\$\}, \{E,F,\$\})\\
(\boxed{5}F\boxed{5}, \{E,F,\$\}, \{E,F,\$\})\\
(\boxed{6}F\boxed{7}, \{E,F,\$\}, \{E,F,\$\})\\ \hline
(\boxed{7}F\boxed{5}, \{E,F,\$\}, \{E,F,\$\})
\end{array}\right\}
\end{array}
$$

OBSERVATION E.7. *Given a sequence **A** of trace actions; then the following are equivalent:*

—***A*** *is the sequence of trace actions on a path from the source vertex to some terminal vertex.*

—***A*** *is the sequence of trace actions of some successful trace, but no proper prefix of **A** is the sequence of trace actions of an accepting trace.*

PROOF OF THEOREM E.1. Let the trace graph G be as defined above; then G is finite, for, by definition, all vertices are subsets of the set of configurations occurring in traces, and by Lemma E.6 this set is finite.

Denote by M the set of parse trees t such that t corresponds to the sequences of trace actions on a path from the source vertex to some terminal vertex. Proposition E.2 together with Observation E.7 give us firstly, that the set M contains all minimal non-canonical parse trees, and secondly, that if the set of canonical parse trees is stable, then M contains only the minimal non-canonical parse trees.

We can test if the set M is infinite simply by testing if there is a loop on a path from the source vertex to some terminal vertex. Thus G gives us the desired representation. $\square$

The trace graph for the determinization $\mathcal{P}_{(4)}$ of the universal CLR(1) parser $\mathcal{U}_{(4)}$ (cf. page 14) is depicted in Figure 2. However, as a kind of optimization, we have excluded all trace configurations with $G = \emptyset$ since these cannot lead to a successful trace. Notice, that it represents exactly the second coordinates of our prioritizing set $\mathcal{E}_{(4)}$, page 16.

APPENDIX F. LOOP FREE PARSING

This appendix addresses problem 3, page 21. By an *ε-loop* in a parsing with a universal CLR(1) parser we understand two reductions with the same production $X \rightarrow \varepsilon$ without any shifts in between. By an *ε-protected* parsing with a CLR(1) parser $\mathcal{P}$ we understand a parsing that halts unsuccessfully when encountering an ε-loop.

THEOREM F.1. *Let $\mathcal{P}$ denote an $(\mathcal{R}^{\prec})^{\mathcal{U}}$-minimal determinization of a universal CLR(1) parser $\mathcal{U}$. Consider the ε-protected parsing with $\mathcal{P}$ on input (X, α). The parsing always terminates, and if not successfully, (X, α) is not generated by any stable canonical parse tree.*

The theorem is divided into two propositions that are proved independently.

PROPOSITION F.2. *Given a determinization of a universal CLR(1) parser, the parsing corresponding to a stable canonical parse tree is always ε-loop free.*

PROOF. Consider a canonical parse tree t where in the corresponding parsing, for some grammar production $X \rightarrow \varepsilon$, there are two reductions of $X \rightarrow \varepsilon$ without any shifts in between. We want to prove that t is not stable canonical.

Denote by I the state of the configuration from which the first of the above reductions are done. Moreover, denote by t_1 and t_2 the results of removing from t the empty leaf rightsides corresponding to the first and the second, respectively, of the above reductions with $X \rightarrow \varepsilon$. Since both t_1 and t_2 are sub-parse trees of t, if t was stable canonical, both t_1 and t_2 would be stable canonical. In the parsings corresponding to t_1 in state I on input X we shift X, whilst in the parsing corresponding to t_2 in state I on input X we reduce $X \rightarrow \varepsilon$. But only one action is canonical in state I on input X, so t_1 and t_2 cannot both be canonical. Thus t is not stable canonical. $\square$

PROPOSITION F.3. *Given a universal CLR(1) parser and an $(\mathcal{R}^{\prec})^{\mathcal{U}}$-minimal determinization. The number of actions carried out in an ε-protected $\mathcal{P}$-parsing of a production $X \rightarrow \alpha$ is $O(|\alpha|)$.*

The $O(|\alpha|)$ bound is not needed here where we are only interested in termination. However, we shall use it in the next appendix to prove linearity of the generated parser.

PROOF. We prove the proposition by showing that before the nth shift of an ε-protected $\mathcal{P}$-parsing there is $O(n)$ actions. By the definition of ε-protected parsing, the number of reductions of empty strings is at most $kn \in O(n)$, where k is the number of grammar symbols. Only shifts and reductions of empty strings increase the size of the stack. As there are at most $O(n)$ of these operations, there is at most $O(n)$ operations decreasing the size of the stack. Thus, we may conclude that there is at most $O(n)$ reductions with productions $X \rightarrow X_1 \cdots X_p$ where $p \neq 1$.

We complete the proof by showing that there cannot be k consecutive reductions of single symbols. Any such sequence would contain a sub-sequence $\boldsymbol{r}$ = reduce $X_k \rightarrow X_0$, reduce $X_{k-1} \rightarrow X_k, \ldots,$ reduce $X_0 \rightarrow X_1$. Suppose, for a contradiction, that we have a canonical parsing starting with the action sequence $\boldsymbol{ar}$. By the definition of universal CLR(1) parsers, there is a completion $\boldsymbol{b}$ such that $\boldsymbol{arb}$ is the action sequence of some parse tree t. Then $\boldsymbol{ab}$ is the action sequence of an equivalent parse tree t^-. Clearly, we encounter a non-canonical action earlier in $\boldsymbol{ab}$ than in $\boldsymbol{arb}$, and $\mathcal{P}$ is $(\mathcal{R}^{\prec})^{\mathcal{U}}$-minimal, so $(t^-, t) \notin (\mathcal{R}^{\prec})^{\mathcal{U}}$. On the other hand, t^- is achieved from t by contraction of a parse tree with X_0 as both root and frontier, so by definition $(t^-, t) \in \mathcal{R}^{\prec} \subseteq (\mathcal{R}^{\prec})^{\mathcal{U}}$. This completes the proof. □

APPENDIX G. THE RESOLVED PARSER

In this appendix we solve problem 4, page 21.

THEOREM G.1. *Given a universal CLR(1) parser, let $\mathcal{P}$ be a complete stable $(\mathcal{R}^{\prec})^{\mathcal{U}}$-minimal determinization. Then $\mathcal{P}$ is a linear time parser.*

PROOF. The result follows if we can prove that any parsing with $\mathcal{P}$ is ε-loop free, for then by Proposition F.3 the parsing carries out a number of actions at most linear in the size of the leftside of the input production, and the time spent on one action is a constant depending on the original grammar. Notice that in the case of a successful parsing with $\mathcal{P}$, the corresponding parse tree is stable canonical, so the result follows directly from Proposition F.2.

Now consider an arbitrary incomplete parsing with input $X \overset{?}{\to} \alpha$. Suppose that we have carried out the actions sequence $\boldsymbol{a}$ shifting α_0, and that we have $\alpha_1\,\$$ left in the input buffer. Then $\alpha_0\alpha_1 = \alpha$. By the completability of universal CLR(1) parsers, there is a string α_1' that starts with the same symbol as α_1, and such that the parsing can be completed successfully with $\alpha_1'\,\$$ in the input buffer. Thus $X \to \alpha_0\alpha_1'$ is a derived production. Moreover, $\mathcal{P}$ is complete, so a parsing of $X \overset{?}{\to} \alpha_0\alpha_1'$ will be successful, and hence ε-loop free. But this parsing starts with the canonical action sequence $\boldsymbol{a}$, which is therefore itself ε-loop free. □

APPENDIX H. THE CLR(1) GENERALIZATION

Finally, we are in the position to describe the generalization of canonical LR(1) technique to deal with ambiguous grammars.

Algorithm C: Let a grammar $\mathcal{G}$ and a finite set $\mathcal{E}_{in}$ of parse tree pairs be given. Denote by $\mathcal{U}$ the universal CLR(1) parser for $\mathcal{G}$. The algorithm decides if there exists a *matching* pair, meaning a pair $(\mathcal{P}, \mathcal{E}_{out})$ where $\mathcal{P}$ is a determinization of $\mathcal{U}$ and where $\mathcal{E}_{out}$ is a finite set of parse tree pairs such that $(\mathcal{E}_{in} \cup \mathcal{E}_{out})$ $\mathcal{U}$-prioritizes $\mathcal{P}$. In this case, it finds such a matching pair $(\mathcal{P}, \mathcal{E}_{out})$, where $\mathcal{P}$ is a linear time parser, and $\mathcal{E}_{out}$ is minimal given $\mathcal{P}$.

C.1.Construct the set $\mathcal{R} = \mathcal{R}(\mathcal{G})$.

C.2.Check that $(\mathcal{E}_{in} \cup \mathcal{R})^{\prec}$ spans a universal parser ordering $\mathcal{O}$. If not, report failure.

C.3.Search among the basis complete $\mathcal{O}$-minimal determinizations $\mathcal{P}$ for one for which the following sub-routine successfully finds a matching set $\mathcal{E}_{out}$ of parse tree pairs; if no such $\mathcal{P}$ exists, report failure:

C.3.1. Construct a representation of a set M such that
- —M contains all minimal non-canonical parse trees.
- —If the determinization is stable then M is exactly the set of minimal non-canonical parse trees.
- —It is decidable whether M is infinite.

If M is infinite, try another determinization.

C.3.2. Let M^- be the set of parse trees in M that are not second coordinates in $\mathcal{E}_{in}$.
C.3.3. Apply ε-protected parsing with $\mathcal{P}$ to all derived productions generated by parse trees in M^-. If one of these derived productions is not accepted, try another determinization.
C.3.4. Set $\mathcal{E}_{out} := \{(t,u)|u \in M^- \text{ and } t \text{ is the canonical equivalent of } u\}$.
C.3.5. Check that $(\mathcal{E}_{in} \cup \mathcal{E}_{out} \cup \mathcal{R})^{\prec}$ spans a universal parser ordering in which $\mathcal{P}$ is minimal; if not, try another $\mathcal{O}$-minimal determinization.
C.3.6. Return successfully the matching pair $(\mathcal{P}, \mathcal{E}_{out})$.

The algorithm will be illustrated by some examples in the next appendix.

LEMMA H.1. *All steps in Algorithm C are implementable.*

PROOF. Theorem D.1 shows that step C.2 and step C.3.5 can be implemented. The loop in Step C.3 only considers basis complete determinizations, so Step C.3.1 can be implemented by Theorem E.1. Also, all the considered determinizations are minimal with respect to $\mathcal{O} = ((\mathcal{E}_{in} \cup \mathcal{R})^{\prec})^{\mathcal{U}}$ which contains $(\mathcal{R}^{\prec})^{\mathcal{U}}$, so steps C.3.3–C.3.4 are implementable by Theorem F.1. All the other steps are trivially implementable, so this completes the proof of the lemma. □

We will now turn to the proof of correctness. Recall from Theorem 8.1, that a determinization $\mathcal{P}$ and and a finite set $\mathcal{E}_{out}$ of parse tree pairs match each other if and only if

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

LEMMA H.2. *If Algorithm C successfully return $(\mathcal{P}, \mathcal{E}_{out})$ in step C.3.6, then $\mathcal{P}$ and $\mathcal{E}_{out}$ match each other, $\mathcal{P}$ is not matched by any cardinality-wise smaller set $\mathcal{E}'_{out}$, and $\mathcal{P}$ is a linear time parser.*

PROOF. By construction of $\mathcal{E}_{out}$ in steps C.3.1–C.3.2, all minimal non-canonical parse trees are second coordinates either in $\mathcal{E}_{in}$ or in $\mathcal{E}_{out}$, so no non-canonical parse trees can be $(\mathcal{E}_{in} \cup \mathcal{E}_{out})^{\prec}$-minimal. Thus (i) follows. Moreover, condition (ii) follows directly from the test in step C.3.5, so, indeed, we may conclude that $\mathcal{P}$ and $\mathcal{E}_{out}$ match each other.

Let $\mathcal{E}'_{out}$ be any set of parse tree pairs matching $\mathcal{P}$. Since $\mathcal{P}$ is prioritized, the set of canonical parse trees is stable, so the set M constructed in step C.3.1 contains exactly the set of minimal non-canonical parse trees. By condition (i) this implies that $\mathcal{E}'_{out}$ has to contain as second coordinates all second coordinates of $\mathcal{E}_{out}$, so $\mathcal{E}'_{out}$ cannot have smaller cardinality than $\mathcal{E}_{out}$.

From condition (ii), it follows that $\mathcal{P}$ is $(\mathcal{R}^{\prec})^{\mathcal{U}}$-minimal. Moreover, since $\mathcal{P}$ is prioritized, it follows that $\mathcal{P}$ is complete, and that the set of canonical parse trees is stable. Thus, by Theorem G.1, we have that $\mathcal{P}$ is a linear time parser. □

LEMMA H.3. *If Algorithm C reports failure, there is no matching pair.*

PROOF. Suppose that the algorithm reports failure, but that there exists a matching pair $(\mathcal{P}', \mathcal{E}'_{out})$. Denote by (i)′ and (ii)′ condition (i) and (ii) with $\mathcal{P}'$ and $\mathcal{E}'_{out}$ in place of $\mathcal{P}$ and $\mathcal{E}_{out}$, respectively. Thus we have that $\mathcal{E}'_{out}$ is finite, and that conditions (i)′ and (ii)′ are satisfied.

Condition (ii)$'$ implies that we pass the test in step C.2 arriving at the loop in step C.3. Thus, reporting failure implies that we have tried an iteration with all basis complete $((\mathcal{E}_{in} \cup \mathcal{R})^{\prec})^{\mathcal{U}}$-minimal determinization. Since $\mathcal{P}'$ is prioritized, it is complete, hence basis complete. Moreover, by condition (ii)$'$, $\mathcal{P}'$ is $((\mathcal{E}_{in} \cup \mathcal{R})^{\prec})^{\mathcal{U}}$-minimal, so there will be an iteration with $\mathcal{P} = \mathcal{P}'$.

Since $\mathcal{P}'$ is prioritized, the set of canonical parse trees is stable, so the set M constructed in step C.3.1 contains exactly the set of minimal non-canonical parse trees. By condition (ii)$'$ the two finite sets $\mathcal{E}_{in}$ and $\mathcal{E}'_{out}$ contain as second coordinates all parse trees in M, so M is finite, and we pass the test in step C.3.2.

Again since $\mathcal{P}'$ is prioritized, all parse trees have stable canonical equivalents. Thus, by Theorem F.1, all the ε-protected parsings are accepting, and hence we pass the test in step C.3.3.

Since all the first coordinates in the set $\mathcal{E}_{out}$ constructed in step C.3.4 are canonical, $\mathcal{E}_{out}$ is contained in $\mathcal{E}^{\prec}$, for any set $\mathcal{E}$ prioritizing $\mathcal{P}$. Thus $(\mathcal{E}_{in} \cup \mathcal{E}_{out} \cup \mathcal{R})^{\prec} \subseteq (\mathcal{E}_{in} \cup \mathcal{E}'_{out} \cup \mathcal{R})^{\prec}$, so condition (ii)$'$ implies that we pass the check in step C.3.5, but this means that the iteration will be successful, contradicting that we reported failure. □

Proof of Main Theorem 5.1 for the CLR(1) technique. The result follows directly from the correctness of Algorithm C, and this correctness is given by the above lemmas. □

We have now completed the proof of Main Theorem 5.1 for the canonical LR(1) technique. Similar constructive proofs can be given for all the other LL and LR techniques. In the next appendix we will go through some examples illustrating Algorithm C, and in Appendix J, there will be a short discussion of the efficiency of the algorithm.

APPENDIX I. EXAMPLES

We will now apply the generalized CLR(1) technique to some of our previous examples. Basically, we will just sum up what we already know about these examples, by relating them to Algorithm C. First we will consider grammar (4) that we have used as an example throughout the paper, and afterwards, we will discuss how the algorithm reports failure for the dangling else grammar from Section 7. For simplicity, in both examples we will assume that $\mathcal{E}_{in} = \emptyset$.

Recall that grammar (4) has productions $E \to F$, $F \to E$, and $F \to FF$. As we saw in Appendix B, the grammar has the following universal CLR(1) parser:

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

INIT	State	*ACTIONS*			*GOTO*	
		E	F	\$	E	F
E	1	s	s	$\div$	2	3
	2	r $F \to E$	r $F \to E$	a, r $F \to E$	$\div$	$\div$
	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 $F \to E$	$\div$	$\div$
	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	$\div$	4	7
	7	s, r $E \to F$	s, r $E \to F$	a, r $E \to F$	4	5

Now, the input for Algorithm C is ready. Step C.1 tells us to construct the set $\mathcal{R}$:

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

Step C.2 asks us to check that $\mathcal{R}^{\prec}_{(4)}$ spans an ordering of $\mathcal{U}_{(4)}$. From the example in Appendix D, it follows that $\mathcal{R}^{\prec}_{(4)}$ spans the ordering:

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

State	*ACTIONS*		
	E	F	$\$$
1	$\div$	$\div$	$\div$
2	$\div$	$\div$	a $\prec$ r $F \to E$
3	s $\prec$ r $E \to F$	s $\prec$ r $E \to F$	$\div$
4	$\div$	$\div$	$\div$
5	s $\prec$ r $E \to F$, r $F \to FF$ $\prec$ r $E \to F$	s $\prec$ r $E \to F$, r $F \to FF$ $\prec$ r $E \to F$	r $F \to FF$ $\prec$ r $E \to F$
6	$\div$	$\div$	$\div$
7	s $\prec$ r $E \to F$	s $\prec$ r $E \to F$	a $\prec$ r $E \to F$

For the loop in step C.3, we have to pick some basis complete $\mathcal{O}_{(4)}$ minimal determinization. The only freedom left in this choice is in state 5 on input E or F, where we can choose between reducing with $F \to FF$ and shifting. In both cases we we choose reduce $F \to FF$. Thus our determinization $\mathcal{P}_{(4)}$ is obtained by choosing the first action in each entry of the action table in $\mathcal{U}_{(4)}$. Concerning step C.3.1, in Appendix E, we found the following trace graph representation of the set M:

$$M_{(4)}: \begin{tikzcd} & v_1 \arrow[r, "E \to F\bullet"] & t_1 \\ s_1 \arrow[ur, "F \to \bullet E"] \arrow[r, "F \to \bullet FF"] \arrow[dr, "E \to \bullet F"'] & v_2 \arrow[r, "F \to F \bullet F"] & t_2 \\ & v_3 \arrow[r, "F \to E\bullet"] & t_3 \end{tikzcd}$$

Now, as the set $\mathcal{E}_{out}$ constructed in steps C.3.2–C.3.4, we get

$$\mathcal{E}_{(4)}: \left\{ \left(E \; , \begin{array}{c} E \\ | \\ F \\ | \\ E \end{array} \right) , \left(F \; , \begin{array}{c} F \\ | \\ E \\ | \\ F \end{array} \right) , \left(\begin{array}{l} F \\ |\backslash \\ FF \\ \;\; |\backslash \\ \;\; FF \end{array} , \begin{array}{l} F \\ | \; \backslash \\ F \;\; F \\ |\backslash \\ FF \end{array} \right) \right\}.$$

Notice that in this example, we have $\mathcal{E}_{in} \cup \mathcal{E}_{out} \cup \mathcal{R} = \mathcal{E}_{(4)}$. Thus step C.3.5 asks us to check that $\mathcal{E}^{\prec}_{(4)}$ spans a universal parser ordering in which $\mathcal{P}$ is minimal. From the example in Appendix D it follows that indeed this is the case. Thus, in step C.3.6, successfully we return the matching pair $(\mathcal{P}_{(4)}, \mathcal{E}_{(4)})$.

We will now discuss how and why the algorithm reports failure for the dangling else grammar (6) with production set $\{E \to \mathtt{a}\, E\, \mathtt{b}\, E, E \to \mathtt{a} E\}$. In the universal CLR(1) parser there will be a conflict on input $\mathtt{b}$ between shifting and reducing with $E{\rightarrow}\mathtt{a}\,E\,\mathtt{b}\,E$. If we choose to shift, we get the following trace graph representation for M:

$$M_{(6)}: \quad \begin{tikzcd} s \arrow[r, "E\rightarrow\bullet\mathtt{a}E"] & v \arrow[loop right, "E\rightarrow\mathtt{a}E\,\mathtt{b}E\bullet"] \arrow[d, "E\rightarrow\mathtt{a}E\bullet\mathtt{b}E"] \\ & t \end{tikzcd}$$

The trace graph contains a loop, so the set M is infinite, meaning that the test in step C.3.1 fails. Notice that M is exactly the second coordinates of the infinite prioritizing set $\mathcal{E}_{(6)}$, page 18. If instead we had chosen reduce $E{\rightarrow}\mathtt{a}\,E\,\mathtt{b}\,E$ over shift, we would have got a trace graph representing the first coordinates. In either case, we have to report failure.

APPENDIX J. SOME TECHNICAL REMARKS

In this appendix we will discuss efficiency and possible generalizations of our approach. So far, in the presentation of our generalization of the canonical LR(1) technique, we have not worried about efficiency, but only about computability. Clearly, our algorithm is exponential in the size of the grammar, as follows just from the fact that we use the canonical LR(1) technique as a front end. In any case, a formal complexity analysis does not make much sense, since grammars are not arbitrary, but written by humans. However, there is a real issue in avoiding trying too many determinizations in the loop of Algorithm C in step C.3.

A first approach to getting down the number of relevant determinizations was already discussed in Section 8; namely to let the user works modularly, introducing only a little new ambiguity in each step, for then set $\mathcal{E}_{in}$ together with $\mathcal{R}$ will largely determine the parsing, keeping the number of relevant determinizations small.

Of course, we cannot rely on any discipline on the user's part. Instead we can improve the algorithm itself by exploiting the fact that not all choices are independent. Take the situation from our example with grammar (4), page 14, where we had the choice in state 5 on input E or F between reducing with $F{\rightarrow}FF$ and shifting. It should be clear that we have to make the same choice for both inputs, for otherwise, there is no chance of getting a set $\mathcal{E}_{out}$ spanning a universal parser ordering. Exploiting such observations algorithmically is possible. Notice, that, if in the above choice, we had picked to shift instead of reducing with $F{\rightarrow}FF$, the iteration would still have been successful, though the returned parser would have been right associative instead of left associative. This situation is expected to be typical, i.e. that if we take dependencies between choices into consideration, then often any determinization is good, so we need only one iteration of the loop. Finally, one could introduce some heuristics that could guide the search for a good determinization.

An other possibility for improving the efficiency is to use as front end the LALR(1) techniques rather than the idealistic canonical LR(1) technique, and then apply exactly the same algorithm for the generalization. Any successfully returned pair would still satisfy all the desired properties, but sometimes, the algorithm would discard determinizations unduly. The problem is that, when applied directly to a universal LALR(1) parser, our spanning algorithm might find some extra projections which turn out to be incompatible with the true projections. The reason for this problem is that universal LALR(1) parsers do not satisfy the full completabily property of universal CLR(1) parsers. Instead they satisfy the following weaker version: suppose in a parsing we arrive at a configuration of the form (α, β). Then for some β', from configuration (α, β'), the parsing can be completed successfully. In order to recognize all good determinizations of a universal LALR(1) parser, it seems as if we need to do some simulation of the canonical LR(1) technique. Such a perfect solution for the LALR(1) technique makes no sense from a practical view point, for the LALR(1) technique is itself a clever example of sacrificing generality for speed. Thus a quick fix, just applying Algorithm C directly to the LALR(1) technique, would suit it just right.

Unfortunately there are interesting grammars like those based on a dangling else for which the techniques presented in this paper will report failure. In that connection, it would be a significant improvement of our techniques to generalize them to deal with parsers $\mathcal{U}$-prioritized by infinite sets of parse tree pairs (cf. Definition 3). Indeed that would capture the dangling else grammar (6), page 18. The universal CLR(1) parser for grammar (6) contains only one conflict, which is between shifting and reducing. Let $\mathcal{P}_{(6)}$ be the determinization obtained by the standard resolution of the conflict in favor of shifting. As pointed out by Maddox (personal communication, 1993), if e is any one of the parse tree pairs from $\mathcal{E}_{(6)}$, page 18, then setting $\mathcal{E}_{in} = \{e\}$ is sufficient for specifying $\mathcal{P}_{(6)}$ (cf. Section 6). Now, $\mathcal{P}_{(6)}$ is a complete linear time parser $\mathcal{U}$-prioritized by the infinite set $\mathcal{E}_{(6)}$, page 18, but how do we verify this automatically? At the moment we have a nice finite representation of $\mathcal{E}_{(6)}$: the second coordinates are succinctly represented by the three node trace graph $M_{(6)}$ that we found automatically at page A–25, and any one of the canonically equivelent first coordinates can be found in linear time using $\mathcal{P}_{(6)}$. The problems that remain are to verify automatically that all the infinitely many second coordinates really do have canonical equivalents, and to verify that all members of $\mathcal{E}_{(6)}$ project in the same way into the one and only conflict in $\mathcal{U}$.