Main Theorem 6.1 *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. Admittedly, in order to keep down the size of the paper, we will only prove the result for the canonical LR(1) technique. The special property of the universal LR(1) parsers constructed by the *canonical* LR(1) technique is that, if in a parsing we arrive at a configuration $(\alpha I, X\beta)$ and $a \in ACTIONS(I, X)$, then there is a sequence β' of input symbols such that the parsing can be completed successfully continuing from $(\alpha I, X\beta')$. In Section 14, we will briefly describe the canonical LR(1) technique modified to deal with derived productions.

7 Traditional parser generation

As mentioned in the introduction, only very specialized techniques [1, 8] have been presented for constructing complete linear time parsers for ambiguous grammars, and neither of these techniques provide any help in identifying the ambiguity . Consequently, today's (meaning in "The Dragon Book" of 1986 [2]) parsers for ambiguous grammars are generated unsafely, either by general heuristics, or by user supplied disambiguating rules. In this section we will discuss some of the problems caused by the lack of safety in the traditional approaches.

First we consider some of the general heuristics used for resolving the conflicts in universal LR parsers. Wharton's heuristic [28] tells us to resolve any shift/reduce or reduce/reduce conflict in favor of reducing of the production with the shortest rightside. For our universal parser $\mathcal{U}_{(4)}$ from the last section, this heuristic implies that in state $3 = GOTO(INIT(F), F))$ on input E, we will choose to reduce $E \rightarrow F$ instead of shifting, and hence the resulting parser will never terminate for an input productions like $F \rightarrow^? FF$, which is in fact a derived production.

Now, consider the heuristic from Yacc [13]. It tells us to resolve a shift/reduce conflict in favor of the shift, and to resolve a reduce/reduce conflict in favor of reducing the production listed first in the grammar. Obviously, this heuristic will not make the same mistake as Wharton's heuristic, but it will make another mistake, for in state $5 = GOTO(GOTO(INIT(F), F), F)$ on input \$, it will choose to reduce $E \rightarrow F$ instead of reduce $F \rightarrow FF$, so again the resulting parser will never terminate for the input production $F \rightarrow^? FF$.

It is easy to construct a general heuristic that works for our example (4). Take, for example, the heuristic that resolves a shift/reduce conflict in favor of the shift, and a reduce/reduce conflict in favor of reducing the production with the longest rightside. This heuristic will lead to a complete linear time parser in the case of (4), but then there are other grammars where we get problems with completeness—but not with termination. For example, this problem crops up in connection with the grammar (1) that we discussed in the in the introduction.

The problems with termination could be avoided if we restricted ourselves to grammars where no termination problems are possible. The set of these grammars is characterized constructively in [23], but why should we exclude grammars like (4)? In any case, we would still be faced with completeness problems.

Of course, for any of the above general heuristics there are grammars, for which our new techniques fail as described in Main Theorem 6.1, but for which the heuristic happens to produce a complete linear time parser. The problem is that we do not know how to test when the heuristics are lucky in this way.

We will now turn our discussion to the user supplied *disambiguating rules* facilitated, for example, by Yacc [13]. Essentially such disambiguating rules are of the form (X, a, b), meaning that in any state on input X, if there is a conflict between action a and b, then it is resolved in favor of a. Generally Yacc reports on conflicts encountered in the parser generation, but it does not report on conflicts resolved by the user supplied disambiguating rules. Below, an example will illustrate how dangerous this approach is.

First, recall the grammar in (3) with production set $\{B{\rightarrow}B{*}B, B{\rightarrow}\mathtt{0}, B{\rightarrow}\mathtt{1}\}$. Suppose we want the operator `*` to be parsed left associatively. With Yacc this is achieved by adding the high-level disambiguating rule 'left `*`' to the declaration of the grammar. This high-level disambiguating rule translates into the low-level disambiguating rule $(*, \mathrm{reduce}\, B{\rightarrow}B{*}B, \mathrm{shift})$. The universal LR(1) parser will contain exactly one such conflict, namely in state $\mathit{GOTO}($ $\mathit{GOTO}(\mathit{INIT}(B), B), B)$, and the above resolution gives us the desired left associative parsing. With our approach, left associativity is achieved by setting

$$\mathcal{E}_{in} := \left\{ \left(\begin{array}{lll} B & & \\ \mid \backslash\backslash & & \\ B & *B & \\ \mid\backslash\backslash & & \\ B{*}B & & \end{array} \;,\; \begin{array}{lll} B & & \\ \mid\backslash\backslash & & \\ B{*}B & & \\ & \mid\backslash\backslash & \\ & B{*}B & \end{array} \right) \right\}.$$

Our techniques will return $\mathcal{E}_{out} = \emptyset$ verifying that indeed $\mathcal{E}_{in}$ is generating, together with an LR(1) parser which is exactly the same as that returned by Yacc. As opposed to Yacc, our techniques guarantees completeness and linear time termination for the generated parser, but are those guarantees necessary? Everybody knows that associativity is easy. Or is it?

Suppose we add a production like $B{\rightarrow}B{*}B{*}$ to our grammar. In the generated universal LR(1) parser, again the only conflict will be in state $\mathit{GOTO}(\mathit{GOTO}(\mathit{INIT}(B), B), B)$ between reduce $B{\rightarrow}B{*}B$ and shift on input $*$. Thus, again our disambiguating rule 'left `*`' resolves all the conflicts in the universal parser completely. So Yacc will not warn us about any new conflicts, but this time the generated parser is incomplete, incapable of accepting input productions like $B\overset{?}{\rightarrow}B{*}B{*}$. Admittedly, the production $B{\rightarrow}B{*}B{*}$ is quite stupid, but the lesson is that the use of disambiguating rules is unsafe, giving no protection against hidden unintended ambiguities that do not show up as new conflicts in the universal parsers. We will return with a more realistic, but more involved, example in Section 10

If instead we use our new approach with $\mathcal{E}_{in}$ as above, we will be told that no extension of $\mathcal{E}_{in}$ prioritizes a complete linear time determinization. If instead, we tried setting $\mathcal{E}_{in} := \emptyset$, we would get

$$\mathcal{E}_{out} := \left\{ \left(\begin{array}{l} B \\ |\backslash\backslash \\ B{*}B \\ \quad\ |\backslash\backslash \\ \quad\ B{*}B \end{array} , \begin{array}{l} B \\ |\ \backslash\backslash \\ B \quad {*}B \\ |\backslash\backslash \\ B{*}B \end{array} \right) , \left(\begin{array}{l} B \\ |\backslash\backslash \\ B{*}B \\ \quad\ |\backslash\backslash\backslash \\ \quad\ B{*}B{*} \end{array} , \begin{array}{l} B \\ |\ \backslash\backslash\backslash \\ B \quad {*}B{*} \\ |\backslash\backslash \\ B{*}B \end{array} \right) \right\}.$$

Together with a complete linear time LR(1) parser prioritized by $\mathcal{E}_{out}$. Properly, a wicked production like $B{\rightarrow}B{*}B{*}$ is mistaken, but this is exactly the strength of our approach: the ability to catch unintended ambiguity for ambiguous grammars in the same way as the traditional LL and LR techniques can do it for intended unambiguous grammars.

8 Optimization with ambiguity

With our new approach to ambiguity it is easy for a user to reify grammars introducing ambiguity in order to get some "better" parse trees. The point is illustrated by some examples.

First, by a *prioritized syntax-directed definition* we mean a syntax-directed definition together with a prioritizing set of parse tree pairs, where, moreover, we assume an implicit linear time parser prioritized by the set of parse tree pairs. In other words, we assume that our techniques have been involved in the construction. The classic example of grammar reification is the introduction of *special case productions*, i.e. productions that are already derived productions. Consider, for example, the following prioritized syntax-directed definition:

(5)

Production	Denotational Function	Priority
$B{\rightarrow}B_1$	$\lambda x.\, x$	$\left(\begin{array}{c} B \\ \vert \\ B_2 \end{array} , \begin{array}{c} B \\ \vert \\ B_1 \\ \vert \\ B_2 \end{array} \right)$
$B_1{\rightarrow}B_2$	$\lambda x.\, x$	
$B{\rightarrow}B_2$	$\lambda x.\, x$	

Nonterminal	Carrier
B, B_1, B_2	"any"

Notice that (5) generates no language. Still (5) is relevant, for in the next section it will be used in a larger syntax-directed definition. Trivially, the design of (5) is semantically unambiguous. The production $B{\rightarrow}B_2$ is a special case production, as it is derivable from the other productions in the grammar. However with this production we never need to use more than one production when jumping between precedence levels. Our priorities ensure us that our special case production will be used whenever possible.

In [2, p. 252] is an example concerned with the special case production $E \rightarrow E\ \texttt{sub}\ E\ \texttt{sup}\ E$ in the grammar with productions $\{E \rightarrow E\ \texttt{sub}\ E, E \rightarrow E\ \texttt{sup}\ E, E \rightarrow E\ \texttt{sub}\ E\ \texttt{sup}\ E\}$. Also this grammar can be dealt with easily using priorities, but the number of priorities in any prioritizing set found by our techniques will be rather large.

We will now turn to a more subtle example of grammar reification using ambiguity. Suppose that we want a grammar for disjunctive expressions, but that we have to do the implementation on a computer that does not directly facilitate disjunction, but instead facilitates conjunction and negation. A first approach is the following prioritized syntax-directed definition:

(6)

Production	Denotational Function	Priority
$B \rightarrow B\texttt{+}B$	$\lambda xy.\neg(\neg x \wedge \neg y)$	$\left(\begin{array}{l} B \\ \vert\backslash\backslash \\ B+B \\ \quad\ \vert\backslash\backslash \\ \quad\ B+B \end{array} , \begin{array}{l} B \\ \vert\ \backslash\backslash \\ B \quad +B \\ \vert\backslash\backslash \\ B+B \end{array}\right)$
$B \rightarrow \texttt{0} \mid \texttt{1}$	false | true	

Nonterminal	Carrier
B	**B**

Trivially (6) gives a correct specification of disjunctive expressions. Unfortunately, it gives rise to a poor implementation with many superfluous negations in connection with long expressions, e.g. for the derived production $B \rightarrow^* B\texttt{+}B\texttt{+}B$ the canonical parse tree

$$\begin{array}{l} B \\ \vert\backslash\backslash \\ B+B \\ \quad\ \vert\backslash\backslash \\ \quad\ B+B \end{array}$$

translates into the functional expression

$$\lambda x_1 y_1 y_2.\neg(\neg x_1 \wedge \neg(\neg(\neg y_1 \wedge \neg y_2))).$$

In order to avoid the superfluous negations, we construct a new prioritized syntax-directed definition.

(7)

Production	Denotational Function	Priority
$B{\rightarrow}C$ $C{\rightarrow}B$	$\lambda x.\,\neg x$ $\lambda x.\,\neg x$	$\left(B\ ,\ \begin{array}{c}B\\ \mid\\ C\\ \mid\\ B\end{array}\right),\ \left(C\ ,\ \begin{array}{c}C\\ \mid\\ B\\ \mid\\ C\end{array}\right)$
$C{\rightarrow}C\texttt{+}C$	$\lambda xy.\,x\wedge y$	$\left(\begin{array}{l}C\\ \mid\backslash\backslash\\ C{+}C\\ \quad\ \mid\backslash\backslash\\ \quad\ C{+}C\end{array}\ ,\ \begin{array}{l}C\\ \mid\ \backslash\backslash\\ C\quad {+}C\\ \mid\backslash\backslash\\ C{+}C\end{array}\right)$
$B{\rightarrow}\texttt{0}\mid\texttt{1}$	false $\mid$ true	

Nonterminal	Carrier
B, C	$\mathbf{B}$

The syntax-directed design of (7) is semantically unambiguous since

$$\forall x\in\mathbf{B} : x=\neg(\neg x)$$
$$\forall x,y,z\in\mathbf{B} : x\wedge(y\wedge z)=(x\wedge y)\wedge z.$$

Back to our motivation for constructing (7), the two first priorities ensure that we will never encounter any superfluous negations, e.g. the derived production $B{\rightarrow}^{*}B\texttt{+}B\texttt{+}B$ has the canonical parse tree

$$\begin{array}{l}B\\ \mid\\ C\\ \mid\backslash\backslash\\ C{+}C\\ \mid\quad\ \mid\backslash\backslash\\ B\quad C{+}C\\ \qquad\ \mid\quad\ \mid\\ \qquad\ B\quad B\end{array}$$

that translates to the functional expression

$$\lambda x_1y_1y_2.\,\neg(\neg x_1\wedge(\neg y_1\wedge\neg y_2))$$

which, as desired, contains no superfluous negations.

The above examples should clearly illustrate how easy it is, with our new approach to ambiguity, to use ambiguity for optimizing the implementation of context-free languages.

9 Modularity

Our approach to ambiguity is inherently modular. This is illustrated by the following prioritized syntax-directed definition for a simple language of boolean expressions:

(8)

Production	Denotational Function	Priority
$B \rightarrow C$ $C \rightarrow B$	$\lambda x.\,\neg x$ $\lambda x.\,\neg x$	$\left(B\ ,\ \begin{array}{c}B\\ \mid\\ C\\ \mid\\ B\end{array}\right)\ ,\ \left(C\ ,\ \begin{array}{c}C\\ \mid\\ B\\ \mid\\ C\end{array}\right)$

$C \rightarrow C$+C	$\lambda xy.\, x \wedge y$	(C
		|\\
		C+C
		    |\\
		    C+C , C
		| \\
		C +C
		|\\
		C+C)

$B_1 \rightarrow B_1 * B_1$	$\lambda xy.\, x \wedge y$	(B
		|\\
		B*B
		    |\\
		    B*B , B
		| \\
		B *B
		|\\
		B*B)

$B \rightarrow B_1$ $B_1 \rightarrow B_2$ $B \rightarrow B_2$	$\lambda x.\, x$ $\lambda x.\, x$ $\lambda x.\, x$	$\left(\begin{array}{c}B\\ \mid\\ B_2\end{array}\ ,\ \begin{array}{c}B\\ \mid\\ B_1\\ \mid\\ B_2\end{array}\right)$
$B_2 \rightarrow$-B_2 $B_2 \rightarrow$0 | 1 $B_2 \rightarrow$(B)	$\lambda x.\,\neg x$ false | true $\lambda x.\, x$	

Nonterminal	Carrier
B, B_1, B_2, C	**B**

The above lines show how (8) is pieced together from smaller prioritized syntax-directed definitions that can be dealt with independently. Indeed, except for the naming of the symbols, in the last section we dealt with all but the last of the smaller definitions, and this last definition is trivial with regards to ambiguity.

Our techniques can verify that it is safe to unite all the smaller prioritized syntax-directed definitions in (8), for if we apply them to the union of the smaller grammars together with the union of the smaller prioritizing sets, the result is a parser together with an empty set of parse tree pairs. The empty set tells us that by uniting the smaller grammars we introduce no new ambiguity, and that our original priorities are sufficient for prioritizing the new parser. Thus, the united syntax-directed definition is semantically unambiguous because of the semantic unambiguity of each of the smaller syntax-directed definitions. Moreover, since we introduced no new ambiguity and since the smaller grammars share no productions, we may conclude that our new parser is a direct extension of the parsers we had prioritized for the smaller syntax-directed definitions, i.e. given a production derived from one of the smaller grammars, the parse tree constructed by the new parser will be the same as the one constructed by the original parser for that grammar.

A real life situation could be that we want to construct a grammar for a programming language with both imperative and applicative features, and that we already have constructed a prioritized syntax-directed definition for an imperative and for an applicative programming language. A tempting approach is to start copying from our old syntax-directed definitions, and copy the old parse tree pairs as well, for we have already checked the semantic irrelevance and the priorities of these parse tree pairs. Applying our techniques to the resulting grammar and the resulting set of parse tree pairs, if they return an empty set of parse tree pairs, then we are done. Our merging has introduced no new ambiguity. Testing this by hand is tedious if the grammars involved are large. Otherwise we only have to consider semantics and priorities of the new parse tree pairs found by the system.

10 Too many cooks ...

In the last section we advocated the idea of having different people working independently on different parts of a syntax directed definition. This framework gives rise to some of the most convincing examples of the need for the safety provided by our approach to ambiguity.

First, continuing our example (8), suppose someone wants to add the production $B_2 \rightarrow *$ with semantics $\perp$ (= undefined). Our techniques can verify that this is not a problem, but what if somebody else wants to add an operator `**` for some kind of exponentiation? The correct solution is to add the production $B_1 \rightarrow B_2 ** B_2$. Seen in isolation, this does not cause any semantic problems, though we do need the generalization of the LR(2) technique in order to deal with the enriched grammar. Now, what happens if both additions are done at the same time?

We will now turn to a more realistic example which approaches a formal proof that the safety provided by our techniques is necessary. The idea is that we have two language designers working independently on a syntax-directed definition, each doing perfectly sensible things, but when the results are merged, we get problems and yet no warnings since no new conflicts are introduced in the action table. The common starting point of the two language designers is the following syntax-directed definition:

(9)

Production	Denotational Function
$I \rightarrow I * I$	$\lambda xy.\, xy$
$I \rightarrow \texttt{if}\ B\ \texttt{then}\ I\ \texttt{else}\ I$	$\lambda xyz.\, \text{if}\ x\ \text{then}\ y\ \text{else}\ z$
$B \rightarrow I$	$\lambda x.\, x \bmod 2$

Nonterminal	Carrier
I	$\mathbf{N}_0$
B	$\mathbf{B} = \{0, 1\}$

In this example, we let our techniques decide about the associativity of $*$. The first of the language designers makes the clever observation that it is cheaper to multiply modulo 2 than to do regular multiplication. Indeed this is true in practice if our regular integers use long representations. He therefore enriches (9) with

Production	Denotational Function
$B \rightarrow B * B$	$\lambda xy.\, xy \bmod 2\ (= x \wedge y)$

together with the priority

$$\left(\begin{array}{ccc} B & & \\ |\backslash\backslash & & \\ B & * & B \\ \vert & & \vert \\ I & & I \end{array} \quad , \quad \begin{array}{ccc} & B & \\ & \vert & \\ & I & \\ & |\backslash\backslash & \\ I & * & I \end{array} \right)$$

The two parse trees are semantically equivalent since

$$\forall xy \in \mathbf{N}_0 : (x \bmod 2) \wedge (y \bmod 2) = xy \bmod 2$$

and running our techniques ensures that so far everything is OK indepedently of the associativity of $*$. With a system like Yacc the same optimization could be achieved by resolving all conflicts between reduce $B \rightarrow I$ and shift on input $*$ in favor of reduce $B \rightarrow I$, i.e. by adding the disambiguating rule $(*, \text{reduce}\ B \rightarrow I, \text{shift})$ (cf. page 22).

The other language designer wants to extend the language with exponentiation

Production	Denotational Function
$I \rightarrow I ** I$	$\lambda xy.\, x^y$

Adding this directly to (9) does not cause any problems, but the two contributions together do cause a problem for our new techniques. They will report that there is no extension of the first designers priority to a prioritizing set for the combined grammar.

The problem discovered by our techniques is real, for suppose we apply Yacc to the combined result of the two designers, i.e. to the combined grammar together with the disambiguating rule supplied by the first designer. All conflicts in the resulting universal parser will be resolved by the disambiguating rule, so Yacc will not produce any warnings. However, the resulting parser will not accept the derived production $B \rightarrow^* I ** I$, for in state $GOTO(INIT(B), I)$ on input $*$ it will reduce $B \rightarrow I$, thereby preventing itself from completing the parsing successfully. Thus Yacc produces an incomplete parser without any warnings.

11 Bad grammars

In this section we will discuss some typical examples of grammars that force our techniques to report a failure. In Part II, Section 21, we will describe exactly how our techniques deal with the problems in these grammars.

Observation 11.1 *There are grammars with no finite generating set of parse tree pairs.*

An example is the following grammar based on a dangling-else:

$$(10)\quad \{E \rightarrow \mathtt{a}\, E\, \mathtt{b}\, E, E \rightarrow \mathtt{a} E\}.$$

It has no finite generating set, but our techniques can offer to start listing the following infinite prioritizing set of parse tree pairs:

$$\mathcal{E}_{(10)}: \left\{ \begin{pmatrix} \begin{array}{l} E \\ |\backslash \\ \mathtt{a}E \\ \quad |\backslash\backslash\backslash \\ \quad \mathtt{a}E\mathtt{b}E \end{array} & , & \begin{array}{l} E \\ |\backslash\backslash\backslash \\ \mathtt{a}E\ \mathtt{b}E \\ \quad |\backslash \\ \quad \mathtt{a}E \end{array} \end{pmatrix}, \ldots, \begin{pmatrix} \begin{array}{l} E \\ |\backslash \\ \mathtt{a}E \\ \quad |\backslash\backslash\backslash \\ \quad \mathtt{a}E\mathtt{b}E \\ \qquad |\backslash\backslash\backslash \\ \qquad \mathtt{a}E\mathtt{b}E \\ \qquad\quad |\backslash\backslash\backslash \\ \qquad\quad \mathtt{a}E\mathtt{b}E \end{array} & , & \begin{array}{l} E \\ |\backslash\backslash\backslash \\ \mathtt{a}E \qquad\quad \mathtt{b}E \\ \quad |\backslash\backslash\backslash \\ \quad \mathtt{a}E\mathtt{b}E \\ \qquad |\backslash\backslash\backslash \\ \qquad \mathtt{a}E\mathtt{b}E \\ \qquad\quad |\backslash \\ \qquad\quad \mathtt{a}E \end{array} \end{pmatrix}, \ldots \right\}$$

We shall return to a discussion of the dangling-else in the end of this section.

Observation 11.2 *There are grammars that have finite generating sets of parse tree pairs, but which do not have any prioritizing sets.*

An example is the grammar

$$(11)\quad \{A \rightarrow BB, B \rightarrow \varepsilon\}$$

It has the following finite generating set of parse tree pairs:

$$\mathcal{E}_{(11)}: \left\{ \begin{pmatrix} \begin{array}{l} A \\ |\backslash \\ BB \\ | \\ \varepsilon \end{array} & , & \begin{array}{r} A\ \ \\ |\backslash \\ BB \\ | \\ \varepsilon \end{array} \end{pmatrix} \right\}$$

from which follows by substitution the parse tree pair

$$\begin{pmatrix} \begin{array}{l} \boldsymbol{A} \\ |\backslash \\ \boldsymbol{BB} \\ |\ \ | \\ \boldsymbol{\varepsilon}\ \varepsilon \end{array} & , & \begin{array}{l} \boldsymbol{A} \\ |\backslash \\ \boldsymbol{BB} \\ |\ \ | \\ \varepsilon\ \boldsymbol{\varepsilon} \end{array} \end{pmatrix}.$$

Hence $\mathcal{E}^{\prec}_{(11)}$ is not irreflexive. Clearly any generating set for the grammar (11) contains either the parse tree pair in $\mathcal{E}_{(11)}$, or the reverse pair. Thus it follows that there cannot be any prioritizing set for (11)—not even an infinite one.

Observation 11.3 *No inherently ambiguous grammar has a finite prioritizing set.*

This observation stems from [26] where it is shown that if we have found a finite prioritizing set of parse tree pairs, then we can automatically disambiguate the grammar.

Observation 11.4 *There are grammars with no deterministic complete linear time LR(1) parsers.*

Recall that when restricted to unambiguous grammars, the canonical LR(1) technique fails (conflicts in the action table) exactly for grammars of this type [16]. A typical example is the one with production set

$$(12)\quad \{S{\rightarrow}\mathtt{aaa}, S{\rightarrow}E\,\mathtt{ab}, E{\rightarrow}\mathtt{a}\}$$

.

Observation 11.5 *There are grammars such that, if $\mathcal{U}$ denotes the universal parser generated by the canonical LR(1) technique, there is a complete linear time determinization $\mathcal{P}$ which is prioritized, but not $\mathcal{U}$-prioritized, by some finite set $\mathcal{E}$ of parse tree pairs.*

The simplest example is the grammar with productions set

$$(13)\quad \{L{\rightarrow}LL, L{\rightarrow}\varepsilon\}$$

The problem, which was actually found using the techniques presented in Part II is the following: it can be shown that for any prioritizing set $\mathcal{E}$ of parse tree pairs, we have in the set

$$\mathcal{R}_{(13)} : \left\{ \left(L \;,\; \begin{array}{c} L \\ |\backslash \\ LL \\ | \;\; \\ \varepsilon \;\; \end{array} \right), \left(L \;,\; \begin{array}{c} L \\ |\backslash \\ LL \\ \;\; | \\ \;\; \varepsilon \end{array} \right) \right\} \subseteq \mathcal{E}^{\prec}$$

Now, by substitution we get the following pairs:

$$\left(\begin{array}{l} \boldsymbol{L} \\ |\backslash \\ LL \\ |\;| \\ \varepsilon\,\varepsilon \end{array} \;,\; \begin{array}{l} \boldsymbol{L} \\ |\backslash \\ \boldsymbol{LL} \\ |\;|\backslash \\ \boldsymbol{\varepsilon}\,LL \\ \;\;\;|\;| \\ \;\;\;\varepsilon\,\varepsilon \end{array} \right) \text{ and } \left(\begin{array}{l} L \\ |\backslash \\ \boldsymbol{L}L \\ \;\;|\backslash \\ \;\;LL \\ \;\;|\;| \\ \;\;\varepsilon\,\varepsilon \end{array} \;,\; \begin{array}{l} L \\ |\;\diagdown \\ \boldsymbol{L}\;\;L \\ |\backslash\;|\backslash \\ \boldsymbol{LL}LL \\ \;\;|\;|\;| \\ \;\;\boldsymbol{\varepsilon}\,\varepsilon\,\varepsilon \end{array} \right)$$

Given any universal LR(1) parser, these pairs will project to

$((GOTO(GOTO(INIT(L), L), L), \$), \text{reduce } L{\rightarrow}LL, \text{reduce } L{\rightarrow}\varepsilon)$ and
$((GOTO(GOTO(INIT(L), L), L), \$), \text{reduce } L{\rightarrow}\varepsilon, \text{reduce } L{\rightarrow}LL)$,

respectively. Thus no prioritizing set for (13) can span an ordering of a universal LR(1) parser.

Recall that our bad example (10) was based on a dangling else. Basically, we saw that a grammar with a dangling else has no finite prioritizing set, and hence it cannot be dealt with successfully by our techniques. However, our techniques can start listing an infinite set of parse tree pairs, and seen from a semantic viewpoint, this is fully sufficient since already the first pair will reveal the semantic problem in the grammar.

Now, how do we react to the semantic problem in the dangling else? One approach is to find a more complicated but semantically unambiguous grammar for the same language. Another approach is to say that the dangling else signals a real problem in the language which should hence be changed. Wirth had a dangling else in his programming language Pascal [29], but changed the construction for his later programming language Modula-2 [30]. Below we compare two equally expressive subsets of Pascal and Modula-2 (In order to avoid a discussion of the semantics of imperative languages, the semantics is just given in terms of syntactically unambiguous segments of pascal programs).

Pascal Production	Denotational Function	Modula-2 Production
$S \rightarrow S;S$	$\lambda xy.\,\texttt{begin}\,x\,;\,y\,\texttt{end}$	$S \rightarrow S;S$
$S \rightarrow \varepsilon$	ε	$S \rightarrow \varepsilon$
$S \rightarrow S_1$	$\lambda x.\,x$	
$S_1 \rightarrow B := E$	$\lambda xy.\,x := y$	$S_1 \rightarrow B := E$
$S_1 \rightarrow \texttt{begin}\,S\,\texttt{end}$	$\lambda x.\,\texttt{begin}\,x\,\texttt{end}$	
$S_1 \rightarrow \texttt{while}\,B\,\texttt{do}\,S_1$	$\lambda xy.\,\texttt{while}\,x\,\texttt{do}\,y$	$S_1 \rightarrow \texttt{WHILE}\,B\,\texttt{DO}\,S_1\,\texttt{END}$
$S_1 \rightarrow \texttt{if}\,B\,\texttt{then}\,S_1$	$\lambda xy.\,\texttt{if}\;x\;\texttt{then}\,y$	$S_1 \rightarrow \texttt{IF}\,B\,\texttt{THEN}\,S_1\,\texttt{END}$
$S_1 \rightarrow \texttt{if}\,B\,\texttt{then}\,S_1\,\texttt{else}\,S_2$	$\lambda xyz.\,\texttt{if}\,x\,\texttt{then}\,y\,\texttt{else}\,z$	$S_1 \rightarrow \texttt{IF}\,B\,\texttt{THEN}\,S_1\,\texttt{ELSE}\,S_2\,\texttt{END}$
$S_1 \rightarrow S_2$	$\lambda x.\,x$	

Thus, if we take the size of the least semantically unambiguous grammar for a language as a measure for how readable it is for humans, then it is clear that Modula-2 is more readable than Pascal. Here readability refers to the difficulty in reading *other* peoples programs. It would be interesting to find out if the concepts of finite versus infinite generating sets have any significance for the theory of natural and artificial languages.

Notice that many of the bad grammars are based on ε-productions. However, ε-productions are not inherently problematic. For example, the use of ε-productions in our grammars for Pascal and Modula-2 does not cause any problems for our techniques. In conclusion, there are many types of grammars that our techniques report failure for, but often the problems for our techniques are based on real problems in the grammar. In any case, we recall that when it comes to ambiguous grammars, basically, there are no safe alternatives to our approach.

Part II
Proofs and Algorithms

12 An introductory outline

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 $X \rightarrow^* X$ be the derived production generated by t, then the corresponding first coordinate is $\mathrm{p}\underline{\mathrm{t}}(X)$. For our example (4), page 30, we have $\mathcal{R}$ equal to

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

Except for a renaming of the grammar symbols, the set $\mathcal{R}_{(4)}$ is also the set $\mathcal{R}$ for the examples (7) and (8). For our example (13), the set $\mathcal{R}$ equals the set $\mathcal{R}_{(13)}$ that we found on page 30. For all the other examples, the set $\mathcal{R}$ is empty.

It will be shown that $\mathcal{R}$ is always finite and computable. Moreover, it will be shown that $\mathcal{R}$ has the important property that a determinization $\mathcal{P}$ of $\mathcal{U}$ 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.

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, as advocated in Section 9, 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 $\mathcal{E}'_{out}$ satisfy 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 witnessing Main Theorem 6.1. However, the outline assumes that we can solve the following problems:

1) How do we prove the claims concerning $\mathcal{R}$? We need to prove

   1a) $\mathcal{R}$ is finite and computable, and

   1b) $\mathcal{P}$ and $\mathcal{E}_{out}$ match each other if and only if (i) and (ii) are satisfied.

2) 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 $\mathcal{E}^{\prec}$.

3) 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?

4) Given a determinization $\mathcal{P}$ and a parse tree t, how do we check that there is a $\mathcal{P}$-canonical equivalent to t for $X \rightarrow \alpha$? The problem is that $\mathcal{P}$ might not be terminating for all input.

5) How do we prove that all matched determinizations are linear time parsers? The essential problem is that they might be non-terminating.

Basically, these problems give the motivation for the remaining sections. However, we are not going to solve them all exactly as stated. In connection with problem 3 and problem 4 we will not give an exact solution if for some other reason we can infer that $\mathcal{P}$ cannot be prioritized. Section 13 solves problem 1. In Section 14 we review the construction of the universal CLR(1) parser from [16], modified to deal with derived productions, and then in Section 15 we introduce some convenient notions concerning such parsers. With the framework set up, Section 16 solves problem 2, Section 17 addresses problem 3, Section 18 addresses 4, and Section 19 deals with 5. After having addressed all the above problems, Section 20 puts everything together in an exact algorithm witnessing Theorem 6.1 for the CLR(1) technique. The algorithm is followed by some examples in Section 21. The whole part is completed by some technical remarks in Section 22.

13 The properties of $\mathcal{R}$

In this section we are going to solve problem 1 from the last section, establishing the properties 1a and 1b of the set catalyst set $\mathcal{R}$ defined on page 32. First we establish property 1a.

Proposition 13.1 *The set $\mathcal{R}$ is finite and computable.*

Proof: The first coordinates in $\mathcal{R}$ are trivial parse trees, i.e. parse trees of the form $\underline{\mathrm{pt}}(X)$. Hence, the result follows if we can show that the depth of any second coordinate in $\mathcal{R}$ is at most $2|A|$ where A denotes the finite set of grammar symbols. Let t be any second coordinate in $\mathcal{R}$. Thus t is a minimal parse tree under the condition that it is non-trivial and has the root symbol as the only symbol in the frontier. Consider the path $(n_0, \ldots, n_k)$ starting at the the root of t and finishing at the only non-ε leaf of t. Let $X_0, \ldots, X_k$ be the labels of $n_0, \ldots, n_k$; then $X_0 = X_k$.

We claim that no grammar symbol can occur twice amongst $X_0, \ldots, X_{k-1}$; for suppose that we has $X_i = X_j$ with $0 \leq i < j < k$. Denote by s the sub-parse tree of t containing all nodes descending from n_i but those strictly descending from n_j. By definition of t all leaves of s except for n_j are labeled ε, so $X_i = X_j$ is both the root symbol of s, and the only symbol in the frontier of s. But s is a proper¡ sub-parse tree of t, so this contradicts the minimality of t. Thus $k \leq |A|$.

Now, consider any path $(m_0, \ldots, m_l)$ in t such that m_0 is a son of one of $n_0, \ldots, n_{k-1}$ but different from either, and such that m_l is a leaf of t; then m_l is labeled ε. The same argument as above shows that no grammar symbol can label more than one of $m_0, \ldots, m_{l-1}$, and hence $l \leq |A|$. Thus the length of the longest path in t from the root to a leaf is at most $2|A|$, which is the desired bound on the depth of t. ∎

The rest of this section concerns property 1b. We start with a very simple result.

Observation 13.2 *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 $(\underline{\mathrm{pt}}(X), t) \in \mathcal{R}$. Since $\mathcal{E}$ prioritizes $\mathcal{P}$ and since $\underline{\mathrm{pt}}(X)$ has no proper sub-parse trees, if $\underline{\mathrm{pt}}(X)$ is not $\mathcal{P}$-canonical, it must be a second coordinate in $\mathcal{E}$. Hence no parse tree containing $\underline{\mathrm{pt}}(X)$ 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 $\underline{\mathrm{pt}}(X)$ is a $\mathcal{P}$-canonical parse tree. Thus, since $\mathcal{E}$ prioritizes $\mathcal{P}$, we have $(\underline{\mathrm{pt}}(X), t) \in \mathcal{E}^{\prec}$, as desired.
■

The hard[1] part of establishing property 1b is to prove the following result:

Theorem 13.3 *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 theorem, first we will see how property 1b follows (slightly paraphrased) as a corollary.

Corollary 13.4 *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 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.

Proof:

$\underline{\Leftarrow}$: 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 Theorem 13.3, 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 13.2, 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 section is devoted to proving Theorem 13.3. 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. $t = \underline{\mathrm{pt}}(X; t(1), \ldots, t(p))$ for some X. An auxiliary tree embedding relation $\sqsubseteq$ is defined as the reflexive partial ordering generated by the following conditions:

[1] Using methods from [22], it can be shown that any proof of Theorem 13.3 is inherently transfinite. More precisely, the theorem is independent of quite strong sub-systems of second order arithmetic. The problem is that there are concrete grammars with concrete prioritizing sets of parse tree pairs such that we cannot constructively bind the number of rewritings that may be used to transform a parse tree into its canonical equivalent. Fortunately, our techniques are never going to carry out these rewritings. In order to prove the correctness of our techniques, we only need know that they will terminate in principle.

- If t is a non-trivial parse tree with the root production $X \rightarrow X_1 \cdots X_p$, 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 $X \rightarrow X_1 \cdots X_p$, 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 [17].

Lemma 13.5 *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 [18]): 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[2].

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 $X \rightarrow X_1 \cdots X_p$ 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 $X \rightarrow X_1 \cdots X_p$ 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.

$\underline{i < K[0] \leq j}$: 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 $j' \geq j > i$ so $t_i \sqsubseteq t_{j'}$ contradicts that $\boldsymbol{t}$ is a counterexample to the lemma.

[2]It is here we set up the transfinite induction needed to prove Theorem 13.3; namely over the uncountably many (countably) infinite sequences of parse trees.

$\underline{K[0] \le i, j}$: 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. ■

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 13.6 *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. ■

Proof of Theorem 13.3: Putting Lemma 13.5 and Lemma 13.6 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 theorem, 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. ■

14 Universal CLR(1) parsers

By *the universal CLR(1) parser of a grammar*, we refer to the general (meaning possibly non-deterministic) LR(1) parser returned by the canonical LR(1) technique [16] 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 [16].

An *accept production* is an auxiliary production on the form $X' \to X$ where X is any grammar symbol. An *item* is a grammar production or an accept production with a '•' 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 $(X_1 \to \beta \bullet X\gamma, Y_1) \in$

CLOSURE(I) and either $Y \in FIRST(\gamma) \setminus \$$, or $Y = Y_1$ and $\$ \in FIRST(\gamma)$, then for all grammar productions $X \rightarrow \alpha$, we have $(X \rightarrow \bullet\, \alpha, Y) \in CLOSURE(I)$. Here

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

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 $CLOSURE(\{X' \rightarrow \bullet\, X, \$\})$.
- 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 [16], that, indeed, this construction is always finite, and that it satisfies all the properties that was mentioned in the beginning of this section.

For our example (4) page 18, we get the goto graph below. The numbers at the bottom right corner of each vertex indicate the corresponding state numbers from $\mathcal{U}_{(4)}$ on page 19.

$(X \rightarrow \alpha \bullet \beta,\ X_0, / \cdots / X_n$ is a shorthand for $(X \rightarrow \alpha \bullet \beta, X_0), \ldots, (X \rightarrow \alpha \bullet \beta, X_n))$

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

15 Partial parsing and other useful notions

In this section 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 we always have $\text{action}(t, |t|) = \text{accept}$.

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 $I = \text{state}(t, i)$, $X = \text{input}(t, i)$, and $a = \text{action}(t, i)$. 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 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 $X \rightarrow \alpha$, 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 $\boldsymbol{b} = \text{reduce}\,X \rightarrow \alpha$.

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 section 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.

16 Spanning a universal parser ordering

The whole section is devoted to proving the following result solving problem 2, page 33:

Theorem 16.1 *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. Thus, the result follows if for any parse tree pair (t,u), we can compute the finite set $P(t,u)$ of projections (cf. page 19) into $\mathcal{U}$ of the parse tree pairs in $\{(t,u)\}^S$; for then we have the following trivial algorithm witnessing Theorem 16.1

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 16.1 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:

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

Here *FIRST* is as defined on page 39. 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. Let $k-1$ be the number of shifts in $\boldsymbol{a}^-$.

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 $I_0^{\mathrm{shift}X}$.

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 $a_\imath = \mathrm{shift}X_k$, 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 $Y \neq \$$, $b \notin \{a, \mathrm{accept}\}$, 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 $Y \in \mathit{FirstL}(X_{k+1} \cdots X_p, L)$, $b \notin \{a, \mathrm{accept}\}$, 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 $a_\imath = \mathrm{accept}$, 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 $I_0^{(\mathrm{shift}X)\boldsymbol{d}}$ on input Y: