DIMACS Technical Report 93-61
August 1993

Controlled Grammatic Ambiguity

by

Mikkel Thorup[1,2,3]
Programming Research Group
of Oxford University
8–11 Keble Road
Oxford OX1 3QD
United Kingdom

e-mail: `thorup@prg.ox.ac.uk`

[1]DIMACS visitor 92/93.
[2]Supported by the Danish Technical Research Council and the Danish Research Academy.
[3]Moving to (10/1/93): University of Copenhagen, Department of Computer Science, Universitetsparken 1, 2100 København Ø, Denmark; e-mail: `mthorup@diku.dk`.

DIMACS is a cooperative project of Rutgers University, Princeton University, AT&T Bell Laboratories and Bellcore.

DIMACS is an NSF Science and Technology Center, funded under contract STC–91–19999; and also receives support from the New Jersey Commission on Science and Technology.

ABSTRACT

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.

Introduction

Our concern in this paper is parser construction for and semantic unambiguity of ambiguous context-free grammars. For unambiguous grammars we have the powerful LL and LR techniques that for large classes can verify their (semantic) unambiguity and construct complete linear time parsers, i.e. parsers that accept the full languages and terminate in linear time on all inputs. With our 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.

Ambiguous grammars have long been considered relevant in connection with programming languages, for as noticed in 1975 by Aho, Johnson, and Ullman [1] 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 [1], and independently in [9], 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 [1], the problem with this approach is that the resolved parser might not be complete. In fact, as noticed in [23], it might not even terminate on some input.

A simple example of a problematic grammar is the one with production set

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

The idea behind the last production is to reduce the number of productions in large parse trees by roughly 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*
	`l`	$	L
0	s 3	÷	1
1	s 3	a	2
2	s 3, r $L\to LL$	r $L\to LL$	2
3	r $L\to\mathtt{l}$, s 4	r $L\to\mathtt{l}$	÷
4	s 5	÷	÷
5	r $L\to\mathtt{lll}$	r $L\to\mathtt{lll}$	÷

Suppose we resolve the conflicts using the heuristic from Yacc [13] of choosing shifts over reductions. When presented with a sentence like `ll`, 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 `l`, we choose to reduce $L \rightarrow LL$. 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 2 is needed. Given the non-deterministic parser constructed by the 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:

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

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 works in linear time. In [1] is found a simple combinatorial test which works for classes of LL but not LR parsers, and [8] 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 structural ambiguity. When Knuth in 1965 presented the canonical LR technique [16], 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. If we want to use an ambiguous grammar directly in the semantic definition of a formal language, as, for example, a programming language or a specification language, we need to ensure that the ambiguity is semantically irrelevant. So far, proving that the ambiguity of an ambiguous grammar is semantically irrelevant has been done by hand [3]. 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.

Our identification of structural ambiguity might be more generally applicable, say to the theory of natural languages, for already in 1956 when Chomsky [6] introduced context-free grammars, he noticed that the study of structural ambiguity is one of the most instructive ways of determining the empirical adequacy of a grammar (direct citation from his later paper [7] with Shützenberger).

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 v as a sub-parse tree. 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 $L \rightarrow \mathtt{111}$.

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. This ability to influence the set of canonical parse trees is crucial in connection with, say, grammar reification, where "better" parse trees are introduced to the grammar, for example by adding special case productions. Such reification makes no sense if we cannot ascertain that these better parse trees get picked by the generated parser.

When comparing our ambiguity resolution based on priorities with the traditional use of disambiguating rules [2], one difference is, of course, the already mentioned guarantees of completeness and termination given by our techniques. 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. The technique independence of the ambiguity resolution given by our rewriting rules make them resemble more ambiguity resolutions like those given in the syntactic metalanguage SDF [14]. 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. A crucial difference is that SDF provides no efficient means for finding the minimal, or canonical, parse tree for a sentence.

Our techniques cannot be successful for all grammars, but at least they can tell when they are not successful. An obvious limitation of our work is that the ambiguity has to be characterizable via a finite set of rewriting rules. Prominent examples of grammars for which this is not the case are grammars containing a dangling else. For such grammars, the best our techniques can do is to start listing an infinite set of rewriting rules. Already the first of these rewriting rules should be sufficient for the user to diagnose the semantic problem in the grammar. This situation is expected to be typical: that the lack of a finite characterization of the ambiguity of a grammar is a signal of a semantic problem. Notice that though the requirement of semantic unambiguity of grammars is natural, it is controversial in connection with traditional parser construction, e.g. [1, 25] are only concerned with the semantics of the parse trees found by the parser. Thus, in order to find the semantics of the generated language, one has to know the concrete parser in use! The advantages of restricting ourselves to 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 [27], 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.

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 to check by hand the semantic unambiguity 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 causing semantic problems and problems in the parser construction. 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 [10] where compiler tyros 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 (67 pages) the paper will be focussed around the canonical LR(1) technique [16] 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.

The paper is divided into two parts: Part I presenting the results, and Part II presenting the relevant proofs and algorithms. A key to our handling of ambiguity is the generalization of the focus from sentence to derived productions (if X is a grammar symbol deriving a string α of grammar symbols, then $X \rightarrow^* \alpha$ is a derived production). Part I is divided as follows: Section 1 describes the basic notion of grammars in terms of derived productions, and Section 2 describes their semantics. In Section 3 we formalize the idea of finitely characterized ambiguity and its relation to semantics. Section 4 reviews parsing, modified to deal with derived productions, and Section 5 formalizes the idea of rewriting rules rewriting all parse trees into the canonical parse trees found by a parser. Section 6 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 7 makes some further comparisons with previous work. In Section 8 we discuss the new opportunities for grammar reification, and in Section 9 we will see how our new approach to ambiguity facilitates modularity in the design and implementation of grammars. Section 10 contains the conclusive "proof" that our techniques are necessary: two language designers working independently on different parts of grammar, both doing perfectly intelligent things, but putting their contributions together, the result is a disaster that neither of the previous techniques can warn against (the disaster is not indicated by any new conflicts in the action tables). Finally, in Section 11, we will discuss the different types of grammars for which our techniques are not successful. This includes a further discussion of the dangling else. Part II starts with an introductory section outlining that part.

The paper is self-contained in order to make it accessible to a broad audience, including both the parsing and the semantics communities. However, the preliminary definitions are rather dense, so the reader is referred to [2] or [20, 21] for a standard text on grammars and parsing, to [24] for a standard text on denotational semantics, and to [3] for a discussion of the denotational semantics of possibly ambiguous grammars.

Part I
Theory and results

1 Grammars in terms of derived productions

As was first noticed algebraically in [4, 5] 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.

In the following we always understand an underlying grammar when we talk about grammar symbols, productions, etc. Moreover, we adopt the following notational convention from [2]: 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) Productions
$S \to A$
$A \to B$
$A \to BC$
$B \to \texttt{b}$
$C \to \texttt{c}$
$C \to \varepsilon$

Nonterminals
S, A, B, C

As in all later examples, the start symbol is understood to be the first mentioned nonterminal, and the terminals are understood to be all the symbols in `typewriter` font. Thus in (2) we have S as the start symbol, and `b` and `c` as terminals.

We are now ready to introduce *derived productions*. Like productions, derived production are pairs (X, α) where X is a grammar symbol and α is a string of grammar symbols. It is, however, not required that X be a nonterminal. A derived production (X, α) is written $X \to^* \alpha$. The set of derived productions is defined recursively as follows:

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

Thus, in our example (2) we have that $A \to^* \texttt{b}C$ is a derived production. Often we will ignore the syntactic difference between derived productions and productions when writing statements like "all productions are derived productions" where formally we should have written something like "for all productions $X \to \alpha$ we have that $X \to^* \alpha$ is a derived production". The leftside and rightside of a derived production is defined as for productions.

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

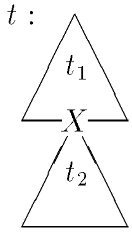
As with sentences, we are interested in the way a derived production is generated from its recursive definition. This is done in terms of *parse trees*. A parse tree is an ordered rooted tree where each node is labeled either with a grammar symbol or with ε. Since our parse tree is ordered, the sons 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 node 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, in our example (2) we have the following parse trees with root symbol A and frontier B:

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

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

If we have two parse trees, like $t_{(2)}$ and $u_{(2)}$, generating the same derived production, we say that they are *equivalent*, and then the underlying grammar is said to be *ambiguous*. Thus our grammar (2) is ambiguous. Traditionally, one only says that a grammar is ambiguous if it 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 t 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.



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

2 Syntax-directed definitions

In the following, grammars will appear in syntax-directed definitions showing their intended semantics, for we want to qualify the difference between, on the one hand, purely syntactical ambiguity and, on the other hand, semantic ambiguity. This section formally defines how syntax-directed definitions give semantics to derived productions. In order to simplify later mathematical reasoning, in the syntax-directed definition, rather than the standard operational style, we shall use a denotational style as in [3].

A *syntax-directed definition* is a grammar where each nonterminal A is associated with a set $\mathcal{C}[A]$ called the *carrier*, and each production p is associated with a function $\mathcal{D}[p]$ called the *denotational function*. Let p be a production with the leftside A and the sequence $A_1, \ldots, A_m$ of nonterminals on the rightside. Then $\mathcal{D}[p]$ is always of type $\mathcal{C}[A_1] \times \cdots \times \mathcal{C}[A_m] \to \mathcal{C}[A]$. The definition is illustrated by a syntax-directed definition for conjunctive expressions:

(3)	Production	Denotational Function
	$B \to B{*}B$	$\lambda xy.\, x \wedge y$
	$B \to \mathtt{0} \mid \mathtt{1}$	false | true
	Nonterminal	Carrier
	B	$\mathbf{B} = \{\text{false}, \text{true}\}$

Above we have used λ-notation for the denotational functions: "$\lambda x_1 \cdots x_p.\, exp$" means "the function f defined by $f(x_1, \ldots, x_n) = exp$". Here, as in the future, for nullary functions we have let the prefix 'λ.' be implicit, e.g. above $\mathcal{D}[B \to \mathsf{0}] = \lambda.\text{false}$.

As for productions, the *semantics of parse trees* are denotational functions. We will only consider the semantics of parse trees whose root symbols are nonterminals. For such a parse tree t the denotational function associated with t is denoted by $\mathcal{D}[t]$. If t has root symbol A and the sequence $A_1, \ldots, A_n$ of nonterminals in the frontier, then $\mathcal{C}[A_1] \times \cdots \times \mathcal{C}[A_n] \to \mathcal{C}[A]$ is the type of $\mathcal{D}[t]$. Below $\mathcal{D}[t]$ is defined recursively.

- If $t = \mathrm{p}\underline{\mathrm{t}}(A)$ then $\mathcal{D}[t]$ is the identity on $\mathcal{C}[A]$.
- Suppose $t = \mathrm{p}\underline{\mathrm{t}}(A; t_1, \ldots, t_p)$ and let $X_1, \ldots, X_p$ be the root symbols of $t_1, \ldots, t_p$. Let $u_1, \ldots, u_m$ be the sequence of parse trees amongst $t_1, \ldots, t_p$ whose root symbols are nonterminals. For $i = 1, \ldots, m$, let B_i be the root symbol of u_i, and let $B_{i,1}, \ldots, B_{i,n_i}$ be the sequence of nonterminals in the frontier of u_i. Notice that $B_{1,1}, \ldots, B_{1,n_i}, \ldots, B_{m,1}, \ldots, B_{m,n_m} = A_1, \ldots, A_n$. Now, the denotational function $\mathcal{D}(t)$ is defined by the equation:

$$\begin{aligned}&\mathcal{D}[t](x_{1,1}, \ldots, x_{1,n_i}, \ldots, x_{m,1}, \ldots, x_{m,n_m})\\ &= \mathcal{D}[X \to X_1 \cdots X_m](\mathcal{D}[u_1](x_{1,1}, \ldots, x_{1,n_i}), \ldots, \mathcal{D}[u_m](x_{m,1}, \ldots, x_{m,n_m})).\end{aligned}$$

A syntax-directed definition is defined to be *semantically unambiguous* if any two equivalent parse trees are associated with the same semantics. In a semantically unambiguous syntax-directed definition, the *semantics of a derived production* p is defined to be the semantics associated with the parse trees generating p.

The syntax-directed definition in (3) can be shown to be semantically unambiguous. For example, we have the following parse trees for the derived production $B \to^* B{*}B{*}B{*}B$:

```
         B                B                                   B                B
         |\\              |\\              B                  | \\             | \\
         B*B              B*B              | \\               B     *B         B     *B
s(3) :    |\\    , t(3) :   | \\   , u(3) : B   *B    , v(3) : |\\         , w(3) : | \\
          B*B              B   *B          |\\  |\\           B*B              B   *B
            |\\            |\\             B*B B*B              |\\            |\\
            B*B            B*B                                  B*B            B*B
```

Using the recursive definition, we get

$$\begin{aligned}&\mathcal{D}[s_{(3)}] = \lambda x_1x_2x_3x_4.\, x_1 \wedge (x_2 \wedge (x_3 \wedge x_4)),\ \mathcal{D}[t_{(3)}] = \lambda x_1x_2x_3x_4.\, x_1 \wedge ((x_2 \wedge x_3) \wedge x_4),\\ &\mathcal{D}[u_{(3)}] = \lambda x_1x_2x_3x_4.\, (x_1 \wedge x_2) \wedge (x_3 \wedge x_4),\ \mathcal{D}[v_{(3)}] = \lambda x_1x_2x_3x_4.\, (x_1 \wedge (x_2 \wedge x_3)) \wedge x_4,\\ &\text{and } \mathcal{D}[w_{(3)}] = \lambda x_1x_2x_3x_4.\, ((x_1 \wedge x_2) \wedge x_3) \wedge x_4.\end{aligned}$$

all of which are equal due to the associativity of $\wedge$. Thus $\lambda x_1x_2x_3x_4.\, x_1 \wedge x_2 \wedge x_3 \wedge x_4$ is the semantics of $B \to^* B{*}B{*}B{*}B$. The associativity of $\wedge$ is crucial for the semantic unambiguity of (3). Suppose, for example, in (3) we changed the denotational function of $B \to B{*}B$ to $\lambda xy.\, x \Rightarrow y$. The operator $\Rightarrow$ is not associative since

$$\text{false} \Rightarrow (\text{true} \Rightarrow \text{false}) = \text{true} \ \neq\ (\text{false} \Rightarrow \text{true}) \Rightarrow \text{false} = \text{false},$$

and by insertion it follows that all of $\mathcal{D}[s_{(3)}]$, $\mathcal{D}[t_{(3)}]$, $\mathcal{D}[u_{(3)}]$, $\mathcal{D}[v_{(3)}]$, and $\mathcal{D}[w_{(3)}]$ becomes different with this change.

Notice that from the semantics of derived productions it is easy to define the traditional semantics for sentences. Consider a sentence s with the parse tree t. Then t generates the derived production $S \rightarrow^* s$ where S is the start symbol. As no nonterminals occur in s the denotational function associated with $S \rightarrow^* s$ is a nullary function with a constant value $c \in \mathcal{C}[S]$. Then c is the semantics associated with the parse tree t. By requiring semantic unambiguity of derived productions, rather than just of sentences, we avoid hidden semantic ambiguity, and thereby we take future extensions into account. Future extensions were considered in [4, 5] where they were discussed in the algebraic framework presented in [3]. In fact, derived productions were originally invented to make this work more intuitive.

Recall that denotational semantics are as powerful as operational semantics. Clearly they are can deal with synthesized attributes, and it is standard to simulate inherited attributes using functions in the carriers [24].

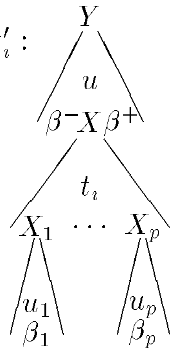
3 Generating sets of parse tree pairs

Given a possibly ambiguous grammar, our first interest is to identify the ambiguity so that we can check that it is semantically irrelevant; but first we have to define what we mean by identifying the ambiguity of a grammar.

By a *parse tree pair* we refer to an ordered pair of equivalent parse trees. Our identification of the ambiguity of a grammar will be of the form of a "generating" set of parse tree pairs with the property that if for each pair the two parse trees are equivalent with respect to some semantics, then the grammar is semantically unambiguous with respect to these semantics. Trivially, the set of all parse tree pairs satisfies this condition. The interesting point is the case where there are infinitely many parse tree pairs, but where we have a finite generating set of parse tree pairs. Below, generating sets will be defined syntactically. Afterwards the formal relation to semantics will be established.

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.

$t'_\imath$:



Let $\mathcal{E}$ be a set of parse tree pairs. Then by $\mathcal{E}^{\equiv}$ we denote the closure of $\mathcal{E}$ under reflexivity, symmetry, transitivity, and substitution. The set $\mathcal{E}$ is said to be *generating* if $\mathcal{E}^{\equiv}$ contains all parse tree pairs of the underlying grammar. The claimed relationship between generating sets and semantics is as follows:

Observation 3.1 *Given a grammar $\mathcal{G}$ and a set $\mathcal{E}$ of parse tree pairs, then the following statements are equivalent:*

- *$\mathcal{E}$ is generating.*
- *$\mathcal{G}$ is semantically unambiguous with respect to any semantics for which $\mathcal{D}[t] = \mathcal{D}[u]$ for all $(t, u) \in \mathcal{E}$.*

Proof: The result is a grammar version of the completeness result for many-sorted equational logic [12] ■

Notice that generating sets could only be defined because we started considering parse trees for derived production instead of just for sentences or sentential forms.

For large classes of ambiguous as well as unambiguous grammars, the techniques presented in Part II automatically provide us with a finite generating set of parse tree pairs. Given the grammar in (3), they would return the following generating set of parse tree pairs:

$$\mathcal{E}_{(3)} : \left\{ a_{(3)} : \begin{pmatrix} \begin{array}{l} B \\ \;|\backslash\backslash \\ B{*}B \\ \qquad |\backslash\backslash \\ \qquad B{*}B \end{array} & , & \begin{array}{l} B \\ \;|\;\backslash\backslash \\ B \quad {*}B \\ |\backslash\backslash \\ B{*}B \end{array} \end{pmatrix} \right\},$$

possibly with the parse trees in the pair swapped. We will illustrate the above definitions by showing how the parse tree pair

$$b_{(3)} : \begin{pmatrix} \begin{array}{l} B \\ \;|\backslash\backslash \\ B{*}B \\ \qquad |\;\backslash\backslash \\ \qquad B \quad {*}B \\ \qquad |\backslash\backslash \\ \qquad B{*}B \end{array} & , & \begin{array}{l} B \\ \;|\;\backslash\backslash \\ B \quad {*}B \\ |\backslash\backslash \quad |\backslash\backslash \\ B{*}B \; B{*}B \end{array} \end{pmatrix}$$

can be derived from $\mathcal{E}_{(3)} = \{a_{(3)}\}$. First, by substitution from $a_{(3)}$, we get the parse tree pairs (**Boldface** indicates the position of $a_{(3)}$)

$$c_{(3)}:\left(\begin{array}{l}B\\ |\backslash\backslash\\ B{*}\boldsymbol{B}\\ \quad|\backslash\backslash\\ \quad\boldsymbol{B{*}B}\\ \qquad|\backslash\backslash\\ \qquad\boldsymbol{B{*}B}\end{array},\begin{array}{l}B\\ |\backslash\backslash\\ B{*}\boldsymbol{B}\\ \quad\ |\ \backslash\backslash\\ \quad\boldsymbol{B}\quad{*}\boldsymbol{B}\\ \quad|\backslash\backslash\\ \quad\boldsymbol{B{*}B}\end{array}\right) \text{ and } d_{(3)}:\left(\begin{array}{l}\boldsymbol{B}\\ |\backslash\backslash\\ \boldsymbol{B{*}B}\\ \quad|\backslash\backslash\\ \quad\boldsymbol{B{*}B}\\ \qquad|\backslash\backslash\\ \qquad B{*}B\end{array},\begin{array}{l}\boldsymbol{B}\\ |\ \backslash\backslash\\ \boldsymbol{B}\quad{*}\boldsymbol{B}\\ |\backslash\backslash\quad|\backslash\backslash\\ \boldsymbol{B{*}B}\ B{*}B\end{array}\right).$$

By symmetry from $c_{(3)}$ we get the parse tree pair

$$e_{(3)}:\left(\begin{array}{l}B\\ |\backslash\backslash\\ B{*}B\\ \quad\ |\ \backslash\backslash\\ \quad B\quad{*}B\\ \quad|\backslash\backslash\\ \quad B{*}B\end{array},\begin{array}{l}B\\ |\backslash\backslash\\ B{*}B\\ \quad|\backslash\backslash\\ \quad B{*}B\\ \qquad|\backslash\backslash\\ \qquad B{*}B\end{array}\right).$$

Now, the desired parse tree pair $b_{(3)}$ follows by transitivity from $e_{(3)}$ and $d_{(3)}$.

As already mentioned, our techniques ascertain that the set $\mathcal{E}_{(3)}$ is generating. Moreover, the two parse trees in the parse tree pair in $\mathcal{E}_{(3)}$ are semantically equivalent since

$$\forall x_1 x_2 x_3 :\ x_1 \wedge (x_2 \wedge x_3) = (x_1 \wedge x_2) \wedge x_3.$$

Thus, it follows from Observation 3.1 that, indeed, the syntax-directed definition (3) is semantically unambiguous. Also observe that if, as in the last section, we substitute $\wedge$ with $\Rightarrow$ in the semantics, then the two parse trees in the parse tree pair in $\mathcal{E}_{(3)}$ stops being semantically equivalent, in agreement with the observation from last section that the syntax-directed definition (3) thereby stops being semantically unambiguous.

In Section 11, we shall see that there are grammars, like those containing a dangling-else, which have no finite generating set. Fortunately, it seems rather unlikely that grammars with no finite generating sets of parse tree pairs should be unambiguous with respect to some non-trivial natural semantics. In any case, 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 generating sets of parse tree pairs. The result of this paper is that these techniques can be generalized to deal with correspondingly limited but important classes of ambiguous grammars with finite generating 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.

As experienced in [3], the difficulty in proving that a syntax-directed definition is semantically unambiguous is not so much to find the, often intended, syntactic ambiguity, as it is to prove that there is no hidden syntactic ambiguity. Thus, the significance of the techniques presented in Part II is not so much that they can "guess" a generating set of parse tree pairs, as it is that they can ascertain that a given set of parse tree pairs is generating.

4 Parsers

Knowing that a grammar is semantically unambiguous with respect to the intended semantics, our next problem is to find an efficient way of *parsing*, i.e. of recognizing derived productions and selecting parse trees that generate them. Notice that this notion of parsing is controversial in the sense that we are concerned with derived productions instead of just with sentences. Generalizing to sentential forms has already been used in connection with error recovery and incremental parsing [19], but, as already mentioned, derived productions are even more general.

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, but modified for dealing with 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 the entry in the action table given by the state and the input symbol.

Let $X \stackrel{?}{\rightarrow} \alpha$ 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 $\mathrm{p\underline{t}}(X)$ 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. 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`	$\div$	`s`	`s`	$\div$	2	$\div$	5	6
	2	$\div$	`s`	$\div$	$\div$	`a`	$\div$	3	$\div$	$\div$
	3	`s`	$\div$	`s`	`s`	$\div$	4	$\div$	5	6
	4	$\div$	`s`	$\div$	$\div$	`r` $B \rightarrow B{*}B$	$\div$	3	$\div$	$\div$
	5	$\div$	`r` $B \rightarrow \mathtt{0}$	$\div$	$\div$	`r` $B \rightarrow \mathtt{0}$	$\div$	$\div$	$\div$	$\div$
	6	$\div$	`r` $B \rightarrow \mathtt{1}$	$\div$	$\div$	`r` $B \rightarrow \mathtt{1}$	$\div$	$\div$	$\div$	$\div$
$*$	7	$\div$	`s`	$\div$	$\div$	$\div$	$\div$	8	$\div$	$\div$
	8	$\div$	$\div$	$\div$	$\div$	`a`	$\div$	$\div$	$\div$	$\div$
`0`	9	$\div$	$\div$	`s`	$\div$	$\div$	$\div$	$\div$	10	$\div$
	10	$\div$	$\div$	$\div$	$\div$	`a`	$\div$	$\div$	$\div$	$\div$
`1`	11	$\div$	$\div$	$\div$	`s`	$\div$	$\div$	$\div$	$\div$	12
	12	$\div$	$\div$	$\div$	$\div$	`a`	$\div$	$\div$	$\div$	$\div$

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

STACK	INPUT-BUFFER	ACTION
$\boxed{1}$	$\mathtt{1}{*}B{*}B\$$	shift
$\boxed{1}\ \mathtt{1}_{\mathrm{p\underline{t}}(\mathtt{1})}\ \boxed{6}$	${*}B{*}B\$$	reduce $B \rightarrow \mathtt{1}$
$\boxed{1}\ B_{\mathrm{p\underline{t}}(B;\mathrm{p\underline{t}}(\mathtt{1}))}\ \boxed{2}$	${*}B{*}B\$$	shift
$\boxed{1}\ B_{\mathrm{p\underline{t}}(B;\mathrm{p\underline{t}}(\mathtt{1}))}\ \boxed{2}\ {*}_{\mathrm{p\underline{t}}(*)}\ \boxed{3}$	$B{*}B\$$	shift
$\boxed{1}\ B_{\mathrm{p\underline{t}}(B;\mathrm{p\underline{t}}(\mathtt{1}))}\ \boxed{2}\ {*}_{\mathrm{p\underline{t}}(*)}\ \boxed{3}\ B_{\mathrm{p\underline{t}}(B)}\ \boxed{4}$	${*}B\$$	shift
$\boxed{1}\ B_{\mathrm{p\underline{t}}(B;\mathrm{p\underline{t}}(\mathtt{1}))}\ \boxed{2}\ {*}_{\mathrm{p\underline{t}}(*)}\ \boxed{3}\ B_{\mathrm{p\underline{t}}(B)}\ \boxed{4}\ {*}_{\mathrm{p\underline{t}}(*)}\ \boxed{3}$	$B\$$	shift
$\boxed{1}\ B_{\mathrm{p\underline{t}}(B;\mathrm{p\underline{t}}(\mathtt{1}))}\ \boxed{2}\ {*}_{\mathrm{p\underline{t}}(*)}\ \boxed{3}\ B_{\mathrm{p\underline{t}}(B)}\ \boxed{4}\ {*}_{\mathrm{p\underline{t}}(*)}\ \boxed{3}\ B_{\mathrm{p\underline{t}}(B)}\ \boxed{4}$	$\$$	reduce $B \rightarrow B{*}B$
$\boxed{1}\ B_{\mathrm{p\underline{t}}(B;\mathrm{p\underline{t}}(\mathtt{1}))}\ \boxed{2}\ {*}_{\mathrm{p\underline{t}}(*)}\ \boxed{3}\ B_{\mathrm{p\underline{t}}(B;\mathrm{p\underline{t}}(B),\mathrm{p\underline{t}}(*),\mathrm{p\underline{t}}(B))}\ \boxed{4}$	$\$$	reduce $B \rightarrow B{*}B$
$\boxed{1}\ B_{\mathrm{p\underline{t}}(B;\mathrm{p\underline{t}}(B;\mathrm{p\underline{t}}(\mathtt{1})),\mathrm{p\underline{t}}(*),\mathrm{p\underline{t}}(B;\mathrm{p\underline{t}}(B),\mathrm{p\underline{t}}(*),\mathrm{p\underline{t}}(B)))}\ \boxed{2}$	$\$$	accept

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

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

The traditional techniques for construction of LL and LR parsers only work for classes of unambiguous grammars. However, in Part II 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 [16] Knuth showed that if a parser 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 parser 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.

5 Prioritized parsers

Our key idea for finding generating sets of parse tree pairs and constructing complete linear time parsers is to do both in parallel by "prioritizing" the parsing as described in this section.

Given a set $\mathcal{E}$ of parse tree pairs, by $\mathcal{E}^{\prec}$ we denote the closure of $\mathcal{E}$ under substitution and transitivity. Notice that in contrast to $\mathcal{E}^{\equiv}$ the relation $\mathcal{E}^{\prec}$ might not be symmetric. In fact, we are especially interested in the case where $\mathcal{E}^{\prec}$ forms a strict partial ordering, i.e. the case where $\mathcal{E}^{\prec}$ is irreflexive. In the following, whenever a set $\mathcal{O}$ of parse tree pairs is viewed as an ordering, if the parse tree pair (t, u) is in the set, then t is seen as *smaller than u with respect to $\mathcal{O}$*.

A set $\mathcal{E}$ of parse tree pairs is said to be *prioritizing* if $\mathcal{E}^{\prec}$ is a well-founded (no infinite descending sequence) strict partial ordering, and no two $\mathcal{E}^{\prec}$-minimal parse trees are equivalent. Now, if $\mathcal{P}$ is a parser and the $\mathcal{P}$-canonical parse trees coincide with the $\mathcal{E}^{\prec}$-minimal parse trees, then $\mathcal{E}^{\prec}$ *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 5.1 *If a set $\mathcal{E}$ of parse tree pairs prioritize a parser $\mathcal{P}$, then $\mathcal{E}$ is generating and $\mathcal{P}$ complete.*

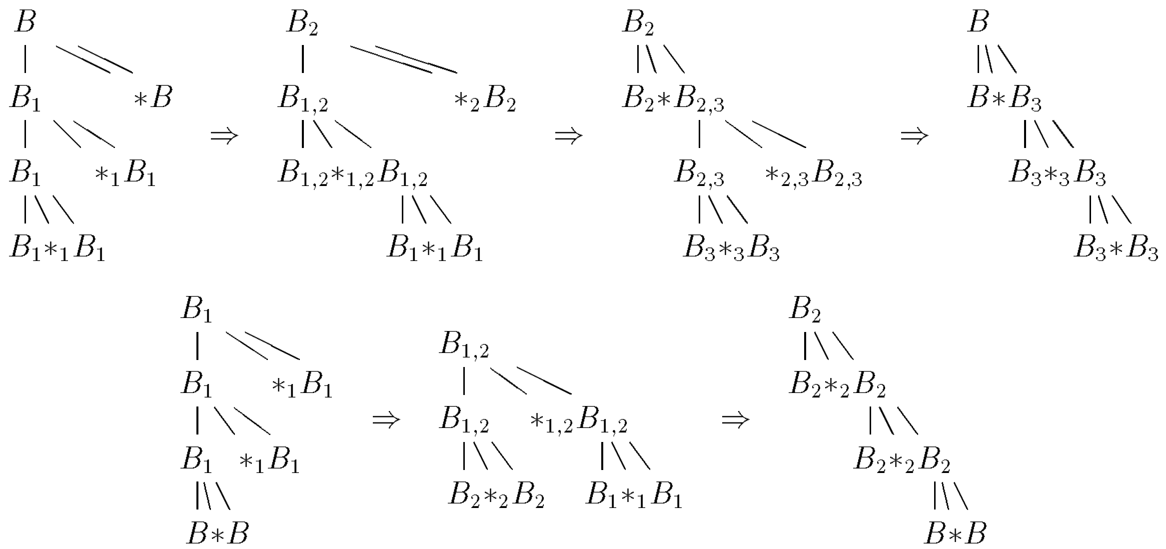
Proof: Let $X \rightarrow^* \alpha$ be a derived production. By definition, $X \rightarrow^* \alpha$ is generated by at least one parse tree u. Since $\mathcal{E}^{\prec}$ is well-founded, it follows that there is an $\mathcal{E}^{\prec}$-minimal, and hence

$\mathcal{P}$-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 $X \rightarrow \alpha$, whence $\mathcal{P}$ is complete.

Now, consider an arbitrary parse tree pair (u, v). By definition there can be at most one $\mathcal{P}$-canonical parse tree in each equivalence class, and by the completeness of $\mathcal{P}$, there is exactly one. Thus, since u and v are equivalent, they have a common canonical equivalent t with $(t, u), (t, v) \in \mathcal{E}^{\prec}$. But, by definition $\mathcal{E}^{\prec} \subseteq \mathcal{E}^{\equiv}$, and $\mathcal{E}^{\equiv}$ is both symmetric and transitive, so $(u, v) \in \mathcal{E}^{\equiv}$. Thus $\mathcal{E}$ is generating. ∎

Sometimes, as in the introduction of this paper, 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 t, u, and t' since u might occur at different positions in t'. 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 always arrive at a $\mathcal{P}$-canonical equivalent.

The techniques that will be presented in Part II can ascertain that the generating set $\mathcal{E}_{(3)}$ of parse tree pairs that we found at page 11 prioritizes the parser $\mathcal{P}_{(3)}$ that we found at page 15. Below is shown two different maximal rewriting sequences with $\mathcal{E}$, 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.

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-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! This 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 presented in Part II can help us prioritize the parsing of grammars.

6 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. In Part II, Section 14, the modified canonical LR(1) technique will be described formally.

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 construction 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 following grammar (which we will return to with some interesting semantics in Section 8):

(4) Productions
$E \rightarrow F$
$F \rightarrow E$
$F \rightarrow FF$

Nonterminals
E, F

Applying the canonical LR(1) technique [16], we get the following non-deterministic LR(1) parser (The actual construction and its exact properties will be described in Section 14).:

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

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

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 canonical LR(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 $\bot$. Take for example, the following parse tree pair from our example (4):

$$a_{(4)} : \left(\begin{array}{lcl} F & & F \\ \mid \;\backslash & & \mid\backslash \\ F \;\;\; F & & F\,F \\ \mid\backslash \;\; \mid & , & \mid \;\; \mid\backslash \\ F\,F\,E & & E\,F\,F \\ \quad\;\;\; \mid & & \mid \\ \quad\;\;\; F & & F \end{array} \right).$$

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

Stack	Buffer	Action	Stack	Buffer	Action
$\boxed{6}$	$FFF\$$	shift	$\boxed{6}$	$FFF\$$	shift
$\boxed{6}\,F\,\boxed{7}$	$FF\$$	$\underline{\text{shift}}$	$\boxed{6}\,F\,\boxed{7}$	$FF\$$	$\underline{\text{reduce } E \to F}$
$\boxed{6}\,F\,\boxed{7}\,F\,\boxed{5}$	$F\$$	reduce $F \to FF$	$\boxed{6}\,E\,\boxed{4}$	$FF\$$	reduce $F \to E$
$\boxed{6}\,F\,\boxed{7}$	$F\$$	shift	$\boxed{6}\,F\,\boxed{7}$	$FF\$$	shift
$\boxed{6}\,F\,\boxed{7}\,F\,\boxed{5}$	$\$$	reduce $E \to F$	$\boxed{6}\,F\,\boxed{7}\,F\,\boxed{5}$	$F\$$	shift
$\boxed{6}\,F\,\boxed{7}\,E\,\boxed{4}$	$\$$	reduce $F \to E$	$\boxed{6}\,F\,\boxed{7}\,F\,\boxed{5}\,F\,\boxed{5}$	$\$$	reduce $F \to FF$
$\boxed{6}\,F\,\boxed{7}\,F\,\boxed{5}$	$\$$	reduce $F \to FF$	$\boxed{6}\,F\,\boxed{7}\,F\,\boxed{5}$	$\$$	reduce $F \to FF$
$\boxed{6}\,F\,\boxed{7}$	$\$$	accept	$\boxed{6}\,F\,\boxed{7}$	$\$$	accept

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, it follows that if $\mathcal{E}$ is a set of parse tree pairs and there exists an ordering of $\mathcal{U}$ that contains $\mathcal{E}$, then there is a least ordering $\mathcal{O}$ of $\mathcal{U}$ containing $\mathcal{E}$. 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. Finally, we say that a set $\mathcal{E}$ of parse tree pairs *$\mathcal{U}$-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.

The above definition of $\mathcal{U}$-prioritizing is extremely important for the following. Loosely speaking, we have that, for a set $\mathcal{E}$ of parse tree pairs priorizing a parser $\mathcal{P}$ to $\mathcal{U}$-prioritizes $\mathcal{P}$, it is required that the finite universal parser can "see" that the rewritings never loop, and that no canonical parse tree can be rewritten.

Continuing our example (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 $\mathcal{E}_{(4)}^{\prec}$. The techniques from Part II 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 $\mathcal{P}_{(4)}$.