Ordered Context-Free Grammars

Ziemowit Laski

Technical Report 99–18

University of California, Irvine
Irvine, CA 92697-3425

laski@ics.uci.edu
http://www.ics.uci.edu/~laski

April 26, 1999
Updated: January 25, 2000

Abstract

Context-free grammars (CFGs) provide an intuitive and powerful formalism for describing the syntactic structure of parsable input streams. Unfortunately, existing online parsing algorithms for such streams admit only a subset of possible CFG descriptions. Theoretically, it is possible to parse any deterministic context-free language (CFL) in a single pass, as long as the grammar describing the CFL belongs to the LR(k), $k \geq 1$ subset of CFGs. However, obtaining a suitable LR(k) description for a language is not an easy task — especially when $k = 1$ — and usually entails an increase in complexity of the rewritten CFG. More importantly, such rewriting inevitably obfuscates the syntactic structure of the language and complicates the placement of semantic bindings. Instead of searching for yet another subclass of CFGs amenable to parsing, we propose to augment the definition of the CFG itself by allowing associativity and precedence to be specified for each production in the grammar. We call the resulting formalism an *ordered context-free grammar* (OCFG). Compared to ordinary context-free grammars, OCFGs can often reduce the number of distinct derivation trees for a given sentence in a CFL; those parse trees that remain can be arranged into a strict partial order. These characteristics make it very easy to craft unambiguous descriptions for context-free languages using OCFGs. At the same time, parsers constructed from OCFGs rely on deterministic pushdown automata and are structurally identical to their CFG counterparts. For the well-known LR(k) and LALR(k) subsets of CFGs, we define analogous subsets of OCFGs, called LRP(k) and LALRP(k), and illustrate how these may be used for defining programming language constructs much more succinctly. Finally, we briefly describe

the $\mathfrak{Bertha}^{\text{TM}}$ parser generator designed to construct bottom-up parsers from LALRP(1) grammar specifications.

1 Introduction and Motivation

Parsing plays an important role in software development and deployment. Traditionally employed as translator front-ends, parsers have also found use in processing streaming content such as markup information [BPSM98] and mobile code [FK97] as it is received over a network. In addition, parsing algorithms can be modified to facilitate the *output* of syntactically structured data, in a process sometimes referred to as "anti-parsing." [Bea95]. In all of these situations, context-free grammars (CFGs) represent the formalism of choice for describing the language to be processed. CFGs foster a separation of concerns (and hierarchical decomposition) by requiring that each construct and syntactic subset be defined apart from the others. Just as importantly, they lend themselves to straightforward and efficient parsing algorithms.

Although numerous deterministic algorithms for CFGs have been developed [GJ98], only *online algorithms* [IK97], performing a single left-to-right scan of the input stream are suitable for the kinds of applications mentioned above. Such algorithms must be able to commence the processing of input before all of it is known. Moreover, their execution time should be linear in the length of the input stream, and their space requirements should be constant or asymptotic, since no upper bound on stream length can be guaranteed to exist in the general case.

Parsing techniques possessing the foregoing characteristics do exist, but impose restrictions on the subsets of CFGs which may be used; the LR(k) class [Knu65, Ear70] is the most general of such parsable subsets.[1] Context-free languages (CFLs) are in LR(k) only if they can be unambiguously recognized using a shift-reduce parser employing at most k tokens of lookahead; CFLs that do not have this property for any finite k are called *nondeterministic* [AU72] since they are incapable of being parsed by any deterministic pushdown automaton (DPDA). Deterministic context-free languages (DCFLs) hence form the broadest subset of CFLs amenable to directional parsing, although it is generally undecidable whether a given CFL is also a DCFL [HU79]. Every DCFL is guaranteed to have an LR(1) grammar; this, however, is largely of theoretical significance. As expressive as the LR(1) grammars are, the construction of parsers for them can be quite costly in terms of both space and time.[2] The LALR(1) subset [DP82] of LR(1) significantly reduces the state space requirements associated with canonical LR(1) parsing, and entails only a minimal loss of expressivity.

[1] LR-Regular (LRR) grammars proposed in [CC73] are even more expressive, but require a *reverse scan* of the input prior to parsing. LAR(m) parsers [BS90] require auxiliary finite state automata to process lookahead conflicts and may have execution time that is quadratic in the length of the input.

[2] An algorithm proposed in [Pag77] improves on this by combining LR(1) parser states whenever possible. An even more efficient algorithm presented in [Spe88] accomplishes the same task by splitting inadequate LR(0) states.

The construction of LALR(1) parsers can also be significantly faster than for unrestricted LR(1) [BL89]. Not surprisingly, LALR(1) parser generators such as `yacc` and `bison` [LMB92] are by far the most commonly used tools, as they offer a sensible compromise between space efficiency, run-time performance and expressive power.

Although an LR(1) grammar is guaranteed to exist for every DCFL, finding it is not always an easy task. Constructing an LALR(1) grammar for a DCFL will generally be even more difficult, if not impossible. More often than not, a CFG submitted to an LR(1) or LALR(1) parser generator will be rejected as ambiguous.[3] The ambiguities in the grammar will take the form of shift-reduce and/or reduce-reduce conflicts in the DPDA which is to perform the parse. This does not necessarily mean that the corresponding CFL is nondeterministic (or inherently ambiguous); it may still be possible for the parser to recognize the language if the ambiguities in the grammar can be removed.

One technique aimed at removing ambiguities from LR(k) CFGs involves rewriting the grammar into a form that is not ambiguous but still describes the same CFL. Several canonical forms of CFGs amenable to DPDA parsing, such as Chomsky Normal Form and Greibach Normal Form,[4] have been proposed and shown to exist for every CFL [Sal73, Woo87]. In some cases, an automated transformation of CFGs to these forms is possible. However, rewriting a CFG in this way can adversely impact its readability and maintainability. For one, the unambiguous CFG will almost always be more complex than the original, ambiguous one [Gru71], containing semantically useless *single productions* [ASU86] that simply enforce associativity and precedence. More importantly, the new CFG will no longer reflect the logical or conceptual structure of the language being defined. Together, these effects complicate the placement of semantic actions with respect to language constructs. Furthermore, should the original grammar already contain semantic bindings, it may be extremely difficult or impossible to transform those bindings as well [PQ96].[5]

Instead of rewriting the grammar, one may also attempt to resolve existing shift-reduce and reduce-reduce conflicts by increasing the length k of the lookahead string from its typical value of one. Doing so is, in fact, strongly advocated by [PQ96]. Still, increasing lookahead in order to eliminate a few conflicts may not be justifiable in terms of the added space and time requirements. This is especially true if the ambiguities in question are not resolvable for any reasonable value of k.

The alternative approach, and the one explored in this paper, relies on

[3] For purposes of this paper, ambiguity shall be defined with respect to the specific class of grammars — LR(1), LALR(1), and so on — for which a parser is sought. Ambiguity in the general sense (i.e., the existence of more than one parse tree for some sentence) for a CFG is undecidable; see Section 2.

[4] Greibach Normal Form can even be used in conjunction with a top-down LL(k) parser, as long as the value of k is sufficiently large.

[5] Semantics can also be associated with specialized *action symbols*, or nullable nonterminals, in the grammar. Action symbols better lend themselves to automated transformations than traditional semantic bindings [Chr99]. Unfortunately, this approach is not very expressive and requires that a predictive parser, such as LL(1), be used.

supplementary precedence and associativity information, rather than grammar rewriting, to resolve parsing ambiguities. Per se, the use of precedence in grammar specifications is not new, and forms the basis of an entire family of *operator grammars* [Flo63, AU72, MP72]. Unfortunately, operator grammars have expressive power far weaker than that of LR(k) [AU72], and usually require that the precedence relation be defined via a separate matrix whose construction is less than intuitive [Col70]. On the other hand, the `yacc` and `bison` parser generators [LMB92] as well as the newer `CUP` parser generator for Java [Hud99] allow ad hoc precedence and associativity information to be provided *in addition* to the formal context-free grammar.[6] This information is used to resolve any conflicts arising during the construction of the LALR(1) DPDA [AJU75], effectively enlarging the set of grammar descriptions handled by these tools. In all of the foregoing approaches, however, both associativity and precedence are bound only to individual *tokens*. This is not satisfactory, as any token may have precedence, associativity and semantics that are dependent on its context of use. For example, the minus sign functions as both a unary and binary operator in most programming languages. The approach presented in this paper circumvents this problem by allowing associativity and precedence attributes to be assigned to *entire productions* rather than individual symbols. This additional information is easily embedded *within* a grammar specification, and need not be provided in a separate data structure.[7] It may be used to control derivation sequences and (perhaps more importantly) resolve parser conflicts. The resulting formalism shall be henceforth referred to as an *ordered context-free grammar* (OCFG).

For purposes of this paper, we assume that the reader is familiar with the basic properties of context-free grammars, and with the inner workings of LALR(1) parser construction tools such as `yacc` or `bison`. An accessible introductory treatment of the foregoing is provided in [App98], while [ASU86] and [AU72] serve as excellent reference sources. The notation used generally follows established formal language nomenclature [AU72, BL89]; unless otherwise indicated, upper-case Latin symbols are used to represent nonterminals, lower-case Greek symbols represent sentential forms (sequences of terminals and nonterminals), and lower-case Latin symbols represent sentences (sequences of terminals only).

Section 2 provides a definition of both a CFG and an OCFG as well the properties of valid derivations for the two. In terms of expressive power, OCFGs equal ordinary CFGs in that they can enumerate exactly the set of all context-free languages (CFLs). The usefulness of OCFGs lies in the fact that they can actually reduce the number of possible derivation trees for a sentence in a CFL. This property has the effect of reducing, or even completely eliminating, the degree of ambiguity in a grammar for a context-free language. Even if the ambiguity is not removed, it is usually possible to induce a strict partial order among the derivations that remain, and then choose the one that comes "first." The particulars of generating sentences from OCFGs naturally lead into a dis-

[6] A similar approach is proposed in [Ear75].

[7] The Syntax Definition Formalism (SDF) [HHKR89] also allows productions to be ordered, albeit separately from the actual context-free grammar.

cussion of automata suitable for parsing such sentences. To this end, Section 3 conceptually describes the construction algorithm for an abstract *nondeterministicc pushdown automaton* (NPDA), with $k \geq 0$ token lookahead, from a CFG or an OCFG. Each state of this automaton can be described as a set of so-called *CFG items* or, alternatively, as a partial order of *OCFG items*. As it turns out, our NPDA formulation provides a convenient mechanism for defining various subsets of *deterministic* context-free languages. In Section 4, we rely on CFG items to formally characterize the well-known LR(k) and LALR(k) subsets of CFGs; analogously, OCFG items become the basis for the definitions of LRP(k) and LALRP(k) subclasses of OCFGs. Parser construction algorithms for all four of these deterministic subsets are described in Section 5. As may be expected, parser construction for LRP(k) and LALRP(k) is a mere generalization of corresponding LR(k) and LALR(k) algorithms, incorporating the additional the precedence and associativity information contained in OCFGs. The $\mathfrak{Bertha}^{\text{TM}}$ parser generator capable of constructing Java$^{\text{TM}}$-based LALRP(1) parsers from type-safe OCFG specifications is presented in Section 6. Section 7 concludes the paper.

2 Grammar Definitions and Derivation Properties

We begin our discussion by formalizing the notion of a context-free grammar and of a derivation based on such a grammar. Without loss of generality, we shall henceforth assume that the start symbol S never occurs on the right-hand side of any production; this will merely obviate the necessity of defining a separate augmented grammar when bottom-up parsing is discussed later on.

Definition 1. A *context-free grammar* (CFG) is a quadruple $G = (V_N, V_T, S, P)$, where

(1) V_N is the set of *nonterminals* in the grammar,
(2) V_T is the set of *terminals*, such that
   (a) $V_N \cap V_T = \emptyset$, and
   (b) $V =_{def} V_N \cup V_T$ constitutes the *alphabet* of the grammar;
(3) $S \in V_N$ is the *start symbol* of the grammar; and
(4) $P \subseteq V_N \times V^*$ is the set of *productions* in the grammar, so that
   (a) If $p = \langle A, \omega \rangle \in P$, we may instead write $p = [A \longrightarrow \omega] \in P$, and
   (b) Nonterminal $A \in V_N$ forms the *left part*, and sentential form $\omega \in V^*$ the *right part*, of production p.

Definition 2. Let $G = (V_N, V_T, S, P)$ be a CFG per Definition 1. Then

(1) $\Longrightarrow$ is a relation on sentential forms over V and represents a single CFG derivation step; $\alpha_1 \Longrightarrow \alpha_2$ if and only if
   (a) $\alpha_1 = \gamma_1 A \gamma_2$,

(b) $\alpha_2 = \gamma_1 \omega \gamma_2$, and
(c) $[A \longrightarrow \omega] \in P$;

(2) $\overset{+}{\Longrightarrow}$ is the transitive closure of $\Longrightarrow$, and
(3) $\overset{*}{\Longrightarrow}$ is the transitive-reflexive closure of $\Longrightarrow$.

We further define $L(G) =_{def} \{w \in V_T^* \mid S \overset{+}{\Longrightarrow} w\}$ as the *language*, or set of all *sentences*, generated by G.

Definitions 1 and 2 are not ground-breaking in any respect; they are provided solely as a notational foundation for the formal definition of an ordered context-free grammar (OCFG) that follows. When generating the sentences in $L(G)$, we may choose to rely solely on *leftmost* derivation steps ($\underset{\text{lm}}{\Longrightarrow}$), where only the leftmost nonterminal in each sentential form is expanded. Analogously, we may use *rightmost* derivation steps ($\underset{\text{rm}}{\Longrightarrow}$); as it turns out, a bottom-up parse of a sentence reconstructs, in reverse order, a rightmost derivation $\overset{+}{\underset{\text{rm}}{\Longrightarrow}}$ for that sentence. These strategies are interchangeable since for every sentential form $\gamma \in V^*$ such that $S \overset{*}{\Longrightarrow} \gamma$, it must also be true that $S \overset{*}{\underset{\text{lm}}{\Longrightarrow}} \gamma$ and $S \overset{*}{\underset{\text{rm}}{\Longrightarrow}} \gamma$. This interchangeability of derivations is a key property of context-free grammars; as we shall see later on, it will hold for OCFGs as well.

Definition 3. An *ordered context-free grammar* (OCFG) is a quad-ruple $G = (V_N, V_T, S, P)$, where

(1) V_N, V_T, V and S are as per Definition 1; and
(2) $P : V_N \longmapsto R$, where $R \subseteq \mathbb{P}(\leq_{\text{P}}, V^* \longmapsto \{0,1\}^*)$, enumerates the productions in G; specifically,
   (a) Function P maps every left part $A \in V_N$ of a production to a partial order[8] R of possible right parts $\omega \in V^*$ for A,
   (b) The *recursion count* $rc(A, \omega)$ denotes the number of occurrences of nonterminal A in the sentential form ω,
   (c) Each right part ω is, in turn, mapped to a Boolean *associativity vector* $a : \{0,1\}^{rc(A,\omega)}$, [9] and
   (d) If $p = \langle A, R \rangle \in P$ and $\langle \omega, a \rangle \in R$, then we may instead write $p = [A \longrightarrow \omega : a] \in P$.

Associativity vectors generalize the notion of associativity used in operator grammars and in the handling of ambiguous CFGs by tools such as `yacc`. If there exists a $p = [A \longrightarrow \omega : a]$ such that $rc(A, \omega) \geq 1$, then the contents of the vector a indicate which of the occurrences of A in ω may actually be expanded during a directly recursive application of p. By setting a to appropriate

[8] The terminology for set orderings sometimes varies; our definition of a partial order is adopted from [GH93] and requires that the $\leq_{\text{P}}$ relation on R be transitive, reflexive and antisymmetric. A *strict* partial order ($<_{\text{P}}$) and an equivalence relation ($=_{\text{P}}$) are also induced on R. Whereas it would also be possible to structure R as a *linear* (i.e., total) order where any two productions in R would be comparable, doing so may unduly restrict the generative power of OCFGs; see Definition 5 below.

[9] If $rc(A, \omega) = 0$, then the associativity vector is empty: $a = \Lambda$, $|\Lambda| = 0$.

values it is possible to control, in a very general way, the allowed and forbidden associativities of p. For example, if $a = 1^{rc(A,\omega)}$, then p is said to be *fully associative*, whereas if $a = 0^{rc(A,\omega)}$ then it is *non-associative*. If $rc(A,\omega) \geq 2$, left- and right-associativity may be established by setting a to $1^{rc(A,\omega)-1}0$ and $01^{rc(A,\omega)-1}$, respectively.[10]

Since only a partial ordering is required of OCFG productions sharing the same left part $A \in V_N$ (and only those), it is perfectly valid for these productions to remain unordered (i.e., pairwise incomparable). Similarly, the associativity restrictions allowed by the OCFG formalism may be avoided simply by making each and every $p \in P$ fully associative. Hence, OCFGs may be viewed as mere generalizations of context-free grammars, and all valid CFGs as special cases of ordered context-free grammars. Conversely, it is occasionally convenient to "reduce" an OCFG to a CFG, even if doing so entails a loss of semantic information. We shall call the resulting grammar a *core CFG*; it can be constructed simply by discarding the precedence and associativity information contained in the corresponding OCFG.

Definition 4. Let $G = (V_N, V_T, S, P)$ be an OCFG per Definition 3. Then there also exists a CFG $G' = \text{CORE}(G) =_{def} (V'_N, V'_T, S', P')$, called the *core grammar* of G, where

(1) $V'_N = V_N$, $V'_T = V_T$ and $S' = S$; and
(2) $[A \longrightarrow \omega] \in P'$ if and only if $[A \longrightarrow \omega : a] \in P$ for some associativity vector $a : \{0,1\}^{rc(A,\omega)}$.

In keeping with established set ordering notation, we have formalized the production ordering R in terms of the $\leq_{\text{P}}$ relation; in practice, it is usually more convenient to specify such orderings using $>_{\text{P}}$ and $=_{\text{P}}$. Definition 3 partitions the production set $P = \{P_A \mid A \in V_N\}$ into subsets, each containing productions for a given nonterminal symbol A. Note that the $\leq_{\text{P}}$ relation between two productions p_1 and p_2 may hold only if both of p_1 and p_2 belong to the same subset P_A. Furthermore, $p_1 = p_2$ always implies that $p_1 =_{\text{P}} p_2$, although the converse need not be true. On the other hand, if p_1 and p_2 reside in different subsets P_A and P_B, where $B \neq A \wedge P_A \cap P_B = \emptyset$, then they must always be incomparable: $p_1 \parallel_{\text{P}} p_2$.

Whereas one could allow $\leq_{\text{P}}$ to hold as well when $A \neq B$, doing so would introduce undesirable side-effects (in addition to being semantically dubious). One of the nice properties of CFGs — and of CFG derivations — is that the nonterminals found in a given sentential form can be expanded in any order. Because of this, for every derivation $S \overset{+}{\Longrightarrow} w$ of a sentence w, there is also guaranteed to exist a leftmost derivation $S \overset{+}{\underset{\text{lm}}{\Longrightarrow}} w$ and a rightmost derivation $S \overset{+}{\underset{\text{rm}}{\Longrightarrow}} w$ such that all three derivations generate the same parse tree for w. By confining the $\leq_{\text{P}}$ relation to productions with identical left parts, we assure that this derivation interchangeability holds for ordered context-free grammars as well.

[10] Trivially, for $rc(A,\omega) = 1$, left- and right-associativity is equivalent to full associativity; when $rc(A,\omega) = 0$, production associativity is undefined.

Permitting $\leq_{\mathrm{p}}$ to hold among *any* two productions in G would bring the definition of an OCFG closer to that of an *ordered grammar*, previously studied by researchers [Fri68, Fri69, Sal73]. Ordered grammars allow production ordering regardless without regard to their left parts, which has the effect of allowing the generation of some *context-sensitive* languages, while (often inadvertently) limiting the extent to which context-free constructs can be enumerated.[11] The same holds true with regard to so-called *programmed grammars* [Ros69]. In a programmed grammar, each production p is associated with two sets $P_{success}$ and $P_{failure}$ of other productions. Only productions in $P_{success}$ may follow p in any derivation. If p cannot be applied, then only productions in $P_{failure}$ may be used instead. *Indexed grammars* [Aho68, HU79], yet another generalization of CFGs, allow special indices $i \in I$, where $I \cap V = \emptyset$, to be appended to nonterminal symbols. The indices are created through special productions of the form $A \longrightarrow Bi$. When B is subsequently expanded with an ordinary production $B \longrightarrow \omega$, the indices propagate to all nonterminals in ω so as to preserve information about their origin. Conversely, an index immediately following a nonterminal may be *consumed* by applying productions of the form $Bi \longrightarrow \omega$. Such productions are clearly not context-free, and are context-sensitive only if $\omega = \phi i$ for some $\phi \in (V \cup I)^+$. As it turns out, indexed languages as a whole form a proper subset of context-sensitive languages.

Having provided and motivated the definition for an OCFG, we now turn to the problem of constructing a derivation sequence for every sentence (or sentential form) enumerable by such a grammar. Recall that in the case of "ordinary" CFG derivations (see Definition 2), a nonterminal $A \in V_N$ within some sentential form ω could be expanded using *any* of the existing productions for A. Constructing an OCFG derivation, however, is a bit more complicated. Given some sentence $w \in V_T^*$, it no longer suffices to present any parse tree whose frontier (i.e., the left-to-right sequence of leaf nodes) spells out w; such a parse tree must also observe the precedence relations — and associativities — of the productions corresponding to its inner nodes.

As was mentioned previously, ordered context-free grammars have the potential of reducing the number of admissible derivation trees for sentences in a language L. It may be tempting to follow this premise to its conclusion and establish derivation semantics guaranteeing that only a single parse tree exists for each distinct sentence in L. Using such an approach would guarantee that every OCFG G is capable of enumerating exactly the same set of context-free languages that a corresponding CFG $G' = \mathrm{CORE}(G)$ enumerates (i.e., $L = L(G) = L(G')$), since a derivation of a given sentence (or sentential form) would be rejected only when another, valid one exists. Unfortunately, establishing the existence of several distinct derivations for a sentential form is equivalent to determining the (non-)ambiguity of the entire grammar, which is well-known to be undecidable in the general case [Woo87]. In principle, decidability could be achieved via a bottom-up parser construction algorithm, were

[11] Using a linear order instead of partial order in an ordered grammar would further restrict its expressive power; see [Sal73].

we able to restrict our field of inquiry to rightmost derivations of DCFLs. As was already mentioned, however, it also cannot generally be established whether a given CFL is deterministic.

A more fundamental reason for not using associativity and precedence as mere disambiguation aids is that, per se, they are not intended to eliminate ambiguity. The principal motivation behind creating the OCFG formalism has been to facilitate the generation of "computationally correct" derivation trees, and the reconstruction of such trees from sentences through parsing.[12] The requirements for achieving this computational correctness are outlined in Definition 5 below. Those derivations that induce an incorrect order of computation (say, addition before multiplication) will be rejected as invalid. To be sure, this process will tend to reduce, and sometimes even eliminate, parse tree ambiguities. In certain situations, *all* existing CFG derivation trees for a sentence may be invalidated, rendering the sentence syntactically incorrect (see below). Still, all of these should be viewed as secondary effects rather than a primary characteristic of ordered context-free grammars.

Hence, we shall not concern ourselves with ambiguity when constructing OCFG derivations. Instead, we focus on constructing a parse tree which, during a depth-first traversal, leads to computation of semantic actions in an order that is consistent with the associativity and precedence provided. A formal characterization of this process is outlined in Definition 5 below. As in a CFG, each non-leaf node of the parse tree corresponds to a production in the grammar. If any two such internal nodes are adjacent (i.e., connected by an edge), then the top node should have precedence that is *no higher* than that of the bottom node. This will hold true when the top node has lower precedence than the bottom node, the same precedence level as the bottom node, or if the two nodes hold different nonterminal symbols and are hence incomparable. If the precedences of the two nodes are equal, then the associativity characteristics of the top node determine if and how the bottom node may be in fact be attached to it. All of the foregoing conditions are formalized below.

Definition 5. Let $G = (V_N, V_T, S, P)$ be an OCFG per Definition 3. Then

(1) $\underset{\mathrm{P}}{\Longrightarrow}$ is a relation on sentential forms over V and represents a single OCFG derivation step; $\alpha_1 \underset{\mathrm{P}}{\Longrightarrow} \alpha_2$ if and only if
   (a) $\alpha_1 = \gamma_1\phi_1 A\phi_2\gamma_2$,
   (b) $\alpha_2 = \gamma_1\phi_1\omega\phi_2\gamma_2$,
   (c) $p_1 = [A \longrightarrow \omega : a_1] \in P$, and
   (d) if $\alpha_0 \underset{\mathrm{P}}{\Longrightarrow} \alpha_1$, where
      i. $\alpha_0 = \gamma_1 A\gamma_2$,
      ii. $p_0 = [A \longrightarrow \phi_1 A\phi_2 : a_0] \in P$,
   then one of the following holds:
      i. $p_1 \parallel_{\mathrm{P}} p_0$,

[12] [Aas95] has introduced the somewhat analogous notion of "precedence correct" syntax trees, applicable to operator grammars.

ii. $p_1 >_{\text{P}} p_0$, or
iii. $p_1 =_{\text{P}} p_0 \wedge a_0[rc(A, \phi_1) + 1] = 1$.[13]

(2) $\overset{+}{\underset{\text{P}}{\Longrightarrow}}$ is the transitive closure of $\underset{\text{P}}{\Longrightarrow}$, and
(3) $\underset{\text{P}}{\overset{*}{\Longrightarrow}}$ is the transitive-reflexive closure of $\underset{\text{P}}{\Longrightarrow}$.

Analogously to CFGs, we define $L(G) =_{def} \{w \in V_T^* \mid S \overset{+}{\underset{\text{P}}{\Longrightarrow}} w\}$ as the language generated by OCFG G.

An important insight contained in [Aas95] is that languages described by grammars should properly be viewed as collections of syntax trees and not merely of sentences; this is certainly true for languages enumerated by ordered context-free grammars defined above. We establish the well-formedness of each OCFG syntax tree by verifying the validity of every path from the root node to a frontier node of the tree. For each path, we establish its correctness inductively using the individual OCFG expansions, beginning with the initial expansion of the start symbol S. If the last step of a derivation involves the expansion of some $A \in V_N$ in a sentential form. then an appropriate production $p_1 = [A \longrightarrow \omega : a_1] \in P$ for A must exist as in ordinary CFG derivations. For OCFG derivations, we must additionally determine whether the nonterminal A was itself created from *yet another* instance of A through the use of some recursive production $p_0 = [A \longrightarrow \phi_1 A \phi_2 : a_0] \in P$. If so, then p_0 must have precedence that is no higher than that of p_1; the boundary condition $p_0 =_{\text{P}} p_1$ indicates that we are dealing with two productions of equal precedence — or possibly a single production $p = p_0 = p_1$ — being applied twice in succession. In this situation, applying p_1 after p_0 will be valid as long as it is used to expand some i-th occurrence of A in $\phi_1 A \phi_2$ for which $a_0[i] = 1$; any other expansions will violate the associativity properties of p_0.

Note that if two distinct productions $p_0 \neq p_1$ share a precedence level ($p_0 =_{\text{P}} p_1$), this does *not* require them to share associativity characteristics; if p_0 and p_1 correspond to adjacent nodes in the derivation tree, only the associativity vector of the parent node is used to determine the construction's validity. This separation of associativity and precedence is quite different from the semantics embedded in `yacc` and `bison`, which require that operators with equal precedence also have identical associativities. When we discuss OCFG parser construction in Section 5, we will see that any shift-reduce conflicts that arise can still be meaningfully resolved.

Having provided formal definitions of ordered context-free grammars and valid OCFG derivations, we now present a concrete (and hopefully illuminating) example. Figure 1 shows an ordered context-free grammar for a subset of programming language expressions. The unary operators "-" and "+" are jointly assigned the highest precedence and made non-associative to forbid nested constructions such as "- - E". These are followed by the right-associative exponentiation operator "↑" and the multiplicative and additive binary operators. (Note the non-associativity of "-" and "/".) The ternary "?:" operator, similar to

[13] $a[i]$, $i \geq 1$, denotes the i-th bit of the associativity vector a.

$$
\begin{array}{l}
[E \longrightarrow num : \Lambda] \\
[E \longrightarrow \text{“-”} E : 0] =_{\mathrm{P}} [E \longrightarrow \text{“+”} E : 0] >_{\mathrm{P}} [E \longrightarrow E \text{“}\uparrow\text{”} E : 01] \\
\qquad\qquad >_{\mathrm{P}} [E \longrightarrow E \text{“*”} E : 10] =_{\mathrm{P}} [E \longrightarrow E \text{“/”} E : 00] \\
\qquad\qquad >_{\mathrm{P}} [E \longrightarrow E \text{“+”} E : 10] =_{\mathrm{P}} [E \longrightarrow E \text{“-”} E : 00] \\
\qquad\qquad >_{\mathrm{P}} [E \longrightarrow E \text{“?”} E \text{“:”} E : 011] \\
[E \longrightarrow \text{“(”} E \text{“)”} : 1]
\end{array}
$$

Figure 1: An example OCFG for a subset of programming language expressions; productions are comparable only where indicated

the one found in C, C++ and Java, is made right-associative and assigned the lowest precedence.

Our grammar example includes a production representing the “()” grouping operator. An appropriate choice of precedence for this production is not immediately obvious. Following deeply-rooted habits, we may initially attempt to assign the highest possible precedence to the parentheses, since expressions contained within them are normally evaluated first. Because the “()” operator is defined via a recursive production, however, any expression immediately contained inside the parentheses will need to have precedence that is at least as high (see Definition 5), thereby precluding us from actually using any of the other operators. If we choose the lowest precedence for “()”, the other operators will be able to appear inside the parentheses, but the parentheses themselves will only be valid as the outermost operator (i.e., the root node of the entire derivation tree). The sole correct solution in this case is to render the production for “()” *incomparable* to all the others by grouping it in a separate linear order. Had we provided another grouping operator, say “[]”, in our grammar, it too would have had to be incomparable to all the others (including “()”). In the same vein, the retrieval of an expression value from a terminal symbol *num* is unrelated to the precedence assigned the various operators in our grammar.

When we compare Definition 5 with Definition 2, we see that the set of allowable OCFG derivations is always a subset (though not necessarily a *proper* subset) of corresponding core CFG derivations. The difference of these two sets contains derivations which, while valid for ordinary context-free grammars, do not observe the appropriate order of computation imposed by the additional associativity and precedence of OCFG productions. As was mentioned previously, such a reduction in the number of OCFG derivations cannot and does not guarantee derivation unambiguity for any particular sentence. On the other hand, it may be possible for an OCFG to exclude *all* derivations (and hence derivation trees) of a sentence successfully generated by the corresponding core CFG. The expressive power of any given OCFG is therefore never greater than that of the its core CFG, and may sometimes be smaller: $L(G) \subseteq L(CORE(G))$.[14]

[14]However, since nothing ever prevents us from constructing an OCFG G such that $L(G) =$

This state of affairs should not be viewed as problematic; should some sentence w generated by a CFG G cease to be derivable under an OCFG G' where $G = \text{CORE}(G')$, it simply means that w cannot be constructed without violating the associativity and precedence constraints imposed by G'. Therefore, whenever w is encountered on the input stream, it should properly be rejected as syntactically incorrect.

Thus, removing computationally incorrect parse trees has the effect of reducing — but not necessarily eliminating — derivation ambiguities in an OCFG. This fact potentially complicates our key objective, namely, the construction of deterministic parsers for ordered context-free grammars. Shift-reduce and reduce-reduce conflicts that hindered the construction of CFG parsers may, in the worst case, persist for OCFGs. Fortunately, since all productions with the same left-hand side in an OCFG must be arranged into a partial order, we may use this fact to construct a canonical ordering of the OCFG derivations themselves.[15] Before we do so, we need to introduce some additional notation. Whereas $S \overset{*}{\underset{\text{P}}{\Longrightarrow}} \omega$ denotes the existence of at least one distinct derivation for a sentential form ω, we shall use $D = [\![A \overset{*}{\underset{\text{rm P}}{\Longrightarrow}} \omega]\!]$ to refer to a particular leftmost derivation for ω, in this case labeled D, from some nonterminal A.[16] We may write $[\![A \overset{*}{\underset{\text{rm P}}{\Longrightarrow}} \omega_1]\!] = [\![A \overset{*}{\underset{\text{rm P}}{\Longrightarrow}} \omega_2]\!]$ (which necessarily implies that $[\![A \overset{*}{\underset{\text{rm P}}{\Longrightarrow}} \omega_1]\!] =_{\text{P}} [\![A \overset{*}{\underset{\text{rm P}}{\Longrightarrow}} \omega_2]\!]$) if and only if $\omega_1 = \omega_2$ and the two derivations construct the same parse tree.

Definition 6. Let $G = (V_N, V_T, S, P)$ be an OCFG per Definition 3. If there exist two valid rightmost derivations $D = [\![A \underset{\text{rm P}}{\Longrightarrow} \alpha_1 \overset{*}{\underset{\text{rm P}}{\Longrightarrow}} \omega_1]\!]$ and $E = [\![A \underset{\text{rm P}}{\Longrightarrow} \alpha_2 \overset{*}{\underset{\text{rm P}}{\Longrightarrow}} \omega_2]\!]$ for $A \in V_N$ and yielding, respectively, $\omega_1 \in V^*$ and $\omega_2 \in V^*$, then one can establish a canonical partial order $\leq_{\text{rm P}}$ between D and E according to the following rules:

(1) If $[A \longrightarrow \alpha_1 : a_1] <_{\text{P}} [A \longrightarrow \alpha_2 : a_2]$, then $D <_{\text{rm P}} E$; otherwise,
(2) If $[A \longrightarrow \alpha_1 : a_1] =_{\text{P}} [A \longrightarrow \alpha_2 : a_2]$, then
  (a) If $\omega_1 = \alpha_1 \vee \omega_2 = \alpha_2$, then $D =_{\text{rm P}} E$;[17] otherwise,
  (b) If $B \in V_N$ is the rightmost nonterminal in both α_1 and α_2, then
    i. If $[\![B \overset{*}{\underset{\text{rm P}}{\Longrightarrow}} \psi_1]\!] =_{\text{P}} [\![B \overset{*}{\underset{\text{rm P}}{\Longrightarrow}} \psi_2]\!]$, then $D =_{\text{rm P}} E$; otherwise,
    ii. If $[\![B \overset{*}{\underset{\text{rm P}}{\Longrightarrow}} \psi_1]\!] <_{\text{rm P}} [\![B \overset{*}{\underset{\text{rm P}}{\Longrightarrow}} \psi_2]\!]$, then $D <_{\text{rm P}} E$.
(3) In all other situations, the derivations are incomparable: $D \parallel_{\text{rm P}} E$.

If D and E are leftmost derivations, a partial order $\leq_{\text{lm P}}$ may be induced among them in a similar way.

The partial order among OCFG derivations D and E is induced directly from precedences of constituent productions; naturally, both D and E must

$L(CORE(G))$, OCFGs are capable of enumerating all context-free languages similarly to CFGs.

[15] [Tho94] proposes that a *prioritization* of semantically equivalent syntax subtrees be established, and that the minimal element of each such equivalence class be chosen when resolving ambiguities.

[16] In the most general case, of course, we have $A = S$.

[17] Additionally, if $\omega_1 = \alpha_1 \wedge \omega_2 = \alpha_2$, then the two derivations are identical: $D = E$.

represent *valid* rightmost derivations per Definition 5 to begin with. Note that it is necessary to choose some consistent method which nonterminals are expanded in each sentential form; in the absence of such a method, it can easily be seen that inducing an ordering of OCFG derivations would be impossible. We chose rightmost derivation for our method since it intuitively corresponds to a left-to-right, bottom-up parse of the input, and because it easily and conveniently maps into item closure computations within each DPDA state, as well as to state transitions. Note that if $\omega_1 = \omega_2$, then Definition 6 establishes a unique ordering among two derivations of the same sentential form, hence allowing us to consistently pick the one that is "first". The OCFG derivation ordering will enable us to resolve many reduce-reduce conflicts arising during parser construction. Even if $\omega_1 \neq \omega_2$, the ordering between D and E is still useful because it aids us in eliminating some of the shift-reduce conflicts. All parser conflict resolution techniques shall be discussed in Section 5.

3 Non-Deterministic Pushdown Automata as Hypothetical Parsers

Theoretically, every context-free language (CFL) can be parsed with a *non-deterministic pushdown automaton* (NPDA) [Woo87]. In practice, only deterministic pushdown automata (DPDA) for directional, online algorithms are constructed; [18] their power of recognition is restricted to deterministic context-free languages (DCFLs). The LR(k) and LALR(k) classes of languages, as well as the LRP(k) and LALRP(k) classes we introduce in Section 4, are all subsets of DCFLs. It turns out, however, that all of these deterministic classes can be defined much more conveniently as subsets of *all* CFGs or OCFGs, simply by imposing various restrictions on a special NPDA which we shall presently define.

One of the main tenets of bottom-up parsing is that every *viable prefix* $\gamma \in V^*$, appearing on the stack at some point during the parse, corresponds to a *state* in the pushdown automaton [ASU86]. As the number of distinct viable prefixes is potentially infinite, every state of the DPDA necessarily corresponds to an entire equivalence class of viable prefixes. On the other hand, each state of our NPDA corresponds to exactly one viable prefix. In defining the NPDA, we rely on the existence of rightmost derivations $\underset{\text{rm}}{\Longrightarrow}$ in the underlying grammar G; the definition of the automaton itself shall proceed by construction. Initially, we shall also assume (without loss of generality) that G is an ordinary CFG. For OCFGs, the ordering of derivations corresponding to an NDPA state matters as well, as we will see shortly.

The state space of an NPDA (or of any other automaton) may be viewed as a digraph where the nodes correspond to states and the edges to state transitions. An NPDA state associated with the viable prefix $\gamma \in V^*$ found on its parse

[18] On-the-fly generation of NPDA states by a deterministic online algorithm, although possible, is very inefficient.

stack shall henceforth simply be called "state γ." Conceptually, viable prefix γ encapsulates all of the shift and reduce operations which have taken place *before* state γ was reached; shift and/or reduce actions taking place *at* state γ will yield transitions to other NPDA states. The *start state* of the NPDA is assigned the empty viable prefix ϵ since no operations have taken place yet. The remaining states are obtained by computing the *closure* of possible state transitions as dictated by grammar G.

First, we take any node α already in the digraph (initially, this will be node ϵ, where $|\epsilon| = 0$). The existence of node α implies that $S \underset{\text{rm}}{\overset{*}{\Longrightarrow}} \alpha z$ for some remaining input $z \in Z \subseteq V_T^*$. Next, we compute the set $\mathcal{S}_\alpha = \text{FIRST}_1(Z) \subseteq V_T$ of tokens which may be retrieved from the input in state α. For each such token $s \in \mathcal{S}_\alpha$, create a node in the digraph labeled αs and insert a directed edge from α to αs. Then, we determine the set $\mathcal{R}_\alpha = \{[A \longrightarrow \omega] \in P \,|\, S \underset{\text{rm}}{\overset{*}{\Longrightarrow}} \beta A z \Longrightarrow \beta\omega z = \alpha z\}$ of productions which may be used to *reduce* the contents α of the parse stack. For each such production, we must determine the appropriate value of β in the foregoing expression, and add a directed edge between the current node α and node labeled βA, which should already exist. Finally, we repeat the previous steps for all newly created nodes; for cycle-free grammars, this process eventually terminates. Should the grammar contain any direct or indirect recursion, however, the number of viable prefixes — and hence distinct NPDA states — will become infinite. This is not problematic, since this infinite NPDA state space is implicitly folded into a finite DPDA state space during the parser construction process.

In the foregoing, we have used the shift and reduce operations allowable in state α of our NDPA solely to compute transitions to other states. For the purposes of parsing, it is also necessary to encode these permissible operations in each state itself. This is most commonly done using a set of so-called *items*; each item contains some production $p = [A \longrightarrow \omega] \in P$ and a *position*, $\eta \bullet \mu \,|\, \eta\mu = \omega$ indicating which leftmost portion η of ω has already been reconstructed from the input. Additionally, each item stores a lookahead w which may follow $\eta\mu$ on the input stream. Using items allows us to characterize each α of our NPDA independently of other states, as the following definition shows.

Definition 7. Let G be a CFG per Definition 1. Then $\mathfrak{I}_\alpha$ corresponds to state α in the NPDA constructed for G, and is a set of *CFG items* in that state. For each such CFG item $\langle p, d, w\rangle \in \mathfrak{I}_\alpha$,

(1) $p = [A \longrightarrow \mu\eta] \in P$;
(2) $d = |\mu|$;
(3) $S \underset{\text{rm}}{\overset{*}{\Longrightarrow}} \delta A y \underset{\text{rm}}{\Longrightarrow} \delta\mu\eta w$; and
(4) $\alpha = \delta\mu$.

Each CFG item may alternately be written as $[A \longrightarrow \mu \bullet \eta \,|\, w] \in \mathfrak{I}_\alpha$.

One of the consequences of using an NPDA, visible in Definition 7, is that each and every CFG item in an NPDA state α is associated with *exactly one* rightmost CFG derivation $S \underset{\text{rm}}{\overset{*}{\Longrightarrow}} \delta A y \underset{\text{rm}}{\Longrightarrow} \alpha\eta w$, with ηw remaining to be read

from the input stream.[19] We make use of this property when constructing NPDA states for an OCFG. Since OCFG derivations form a partial order (see Definition 6), so can OCFG *items* residing in a given NPDA state.

Definition 8. Let G be an OCFG per Definition 3. Then ${}_P\mathfrak{I}_\alpha$ corresponds to state α in the NPDA constructed for G, and is a partial order of *OCFG items* in that state. For each such OCFG item $\langle p, d, w\rangle \in {}_P\mathfrak{I}_\alpha$,

(1) $p = [A \longrightarrow \mu\eta : a] \in P$;
(2) $d = |\mu|$;
(3) $S \overset{*}{\underset{\text{rm P}}{\Longrightarrow}} \delta A y \underset{\text{rm P}}{\Longrightarrow} \delta\mu\eta w$; and
(4) $\alpha = \delta\mu$.

Each OCFG item may alternately be written as $[A \longrightarrow \mu \bullet \eta : a \,|\, w] \in {}_P\mathfrak{I}_\alpha$. The partial order $\leq_{\text{rm P}}$ of the OCFG items in NPDA state α is exactly the same as that of their corresponding righmost OCFG derivations, identified in (3).

The rationale for establishing an ordering among the OCFG items is that it allows us to resolve some of the action conflicts arising during the construction of a deterministic parser.[20] Should a conflict involving two distinct items in a state arise which cannot be resolved using lookahead alone, we may resolve it "in favor" of the OCFG item with the higher precedence, as determined by its accompanying OCFG derivation. If the conflict involves one and the same OCFG item, the associativity of the underlying production can be used to choose either a shift or a reduce action. Since only a partial order exists among OCFG items, the foregoing conflict resolution mechanism may not always succeed. Nevertheless, this still represents a marked improvement over ordinary CFGs, where such resolutions are *never* possible.

It is important to note that the foregoing definitions allow for unlimited lookahead on the input stream. In practical parsing scenarios, lookahead is typically restricted to some fixed length $k \geq 0$. Hence, it is sensible to also impose this restriction on CFG and OCFG items that make up states in our hypothetical NPDA.

Definition 9. Let $\mathfrak{I}_\alpha$ be a set of CFG items in NPDA state α per Definition 7. Then $\mathfrak{I}^k_\alpha$, where $k \geq 0$, is a set of *CFG items with k-token lookahead* in state α, so that $\langle p, d, \text{FIRST}_k(w)\rangle \in \mathfrak{I}^k_\alpha$ for every $\langle p, d, w\rangle \in \mathfrak{I}_\alpha$.

Definition 10. Let ${}_P\mathfrak{I}_\alpha$ be a partial order of OCFG items in NPDA state α per Definition 8. Then ${}_P\mathfrak{I}^k_\alpha$, where $k \geq 0$, is a strict partial order of *OCFG items with k-token lookahead* in state α, so that $\langle p, d, \text{FIRST}_k(w)\rangle \in {}_P\mathfrak{I}^k_\alpha$ for every $\langle p, d, w\rangle \in {}_P\mathfrak{I}_\alpha$, with the ordering preserved: $\langle p_1, d_1, \text{FIRST}_k(w_1)\rangle \leq_{\text{rm P}} \langle p_2, d_2, \text{FIRST}_k(w_2)\rangle$ if and only if $\langle p_1, d_1, w_1\rangle \leq_{\text{rm P}} \langle p_2, d_2, w_2\rangle$.

[19] The unique derivation property will no longer hold, of course, once the NPDA states are folded into DPDA states.

[20] In our NPDA, conflicting actions are simply pursued simultaneously, in a manner typical for nondeterministic automata.

When only a finite number k of initial tokens is stored for each lookahead string, it is possible for two previously distinct sets of CFG or OCFG items to become equal: $\mathfrak{I}_\alpha^k = \mathfrak{I}_\beta^k$ or ${}_P\mathfrak{I}_\alpha^k = {}_P\mathfrak{I}_\beta^k$, for $\alpha \neq \beta$. As we shall see in Section 4 below, the different parsable subsets of CFGs and OCFGs differ precisely as to when these equalities hold. Note that for ${}_P\mathfrak{I}_\alpha^k = {}_P\mathfrak{I}_\beta^k$ to hold, the ordering of the constituent items, as well as the items themselves (possibly modulo the lookahead sets), must be identical.

It is possible to construct the theoretical automaton just described for *any* CFG or OCFG. Shift and reduce operations allowed at a state α, as indicated by the sets $\mathcal{S}_\alpha$ and $\mathcal{R}_\alpha$, may all be attempted simultaneously if need be; there is no notion of a conflict in a nondeterministic automaton. More than one of these attempts (or none at all) may succeed, possibly leading to multiple derivation trees being constructed for the same input sentence. In fact, *all* possible syntax trees would be constructed for each sentence, yielding a so-called *universal* parser [Tho94]. Deterministic bottom-up parsers encountered in practice, however, may produce at most a single parse tree for any given input. Furthermore, operations can no longer be undertaken simultaneously; the automaton must perform a single shift or reduce, possibly after consulting lookahead obtained from the input stream. This necessitates that a suitable restriction be placed on allowable input grammars, as discussed in the next section.

4 Deterministic Subsets of CFGs and OCFGs: LR(k), LALR(k), LRP(k) and LALRP(k)

In order to use bottom-up parsing algorithms, we will need to restrict our choice of input grammars to subsets capable of being handled by a DPDA with fixed-length lookahead. Ideally, such a restriction would encompass exactly the grammars describing deterministic context-free languages (DCFLs). However, since the determinism of a context-free language is not decidable, neither can be the properties of any grammar describing such a language. Fortunately, one can define subsets of such grammars, corresponding to subsets of DCFLs, whose boundaries are known. For any grammar G contained in these subsets, one can then construct a DPDA capable of *unambiguously* recognizing all sentences in $L(G)$. The LR(k) subset of CFGs is actually capable of describing all of the DCFLs; in fact, LR(1) is sufficient for the task [Knu65, AU72], although an appropriate LR(1) grammar for a DCFL may be difficult to obtain (see Section 1). To formally define the LR(k) subset, we rely on the NPDA constructed in the previous section.

Definition 11. Let $G = (V_N, V_T, S, P)$ be a CFG per Definition 1. We say that $G \in \text{LR}(k)$, for $k \geq 0$, if and only if for any two states α, β in the NPDA for G, the conditions

(1) $[A \longrightarrow \mu_1 \bullet \eta_1 \,|\, w_1] \in \mathfrak{I}_\alpha^k$;

(2) $[B \longrightarrow \mu_2 \bullet \eta_2 \,|\, w_2] \in \mathfrak{I}_\beta^k$;
(3) $\mathfrak{I}_\alpha^k = \mathfrak{I}_\beta^k$; and
(4) $[A \longrightarrow \mu_1 \bullet \eta_1 \,|\, w_1] \neq [B \longrightarrow \mu_2 \bullet \eta_2 \,|\, w_2]$

jointly imply that $\text{FIRST}_k(\eta_1 w_1) \cap \text{FIRST}_k(\eta_2 w_2) = \emptyset$.

The LALR(k) subset of LR(k) is obtained by merging the states of the LR(k) DPDA until it becomes an LR(0) DPDA, but nevertheless employs k-tuples of lookahead to resolve action conflicts. Compared with LR(k) grammars, LALR(k) grammars have significantly reduced storage requirements, while incurring only a slight loss of expressivity. Note that our use of an NPDA prevents us from actually having to specify the process of merging of lookaheads that is necessary during DPDA construction. (Traditionally, researchers have found it difficult to elegantly define LALR(k) grammars [DP82].) The brevity obtained in Definitions 12 and 14 that follow is possible due to our use of a nondeterministic automaton as the hypothetical parser engine.

Definition 12. Let $G = (V_N, V_T, S, P)$ be a CFG per Definition 1. We say that $G \in \text{LALR}(k)$, for $k \geq 0$, if and only if for any two states α, β in the NPDA for G, the conditions

(1) $[A \longrightarrow \mu_1 \bullet \eta_1 \,|\, w_1] \in \mathfrak{I}_\alpha^k$;
(2) $[B \longrightarrow \mu_2 \bullet \eta_2 \,|\, w_2] \in \mathfrak{I}_\beta^k$;
(3) $\mathfrak{I}_\alpha^0 = \mathfrak{I}_\beta^0$; and
(4) $[A \longrightarrow \mu_1 \bullet \eta_1 \,|\, w_1] \neq [B \longrightarrow \mu_2 \bullet \eta_2 \,|\, w_2]$

jointly imply that $\text{FIRST}_k(\eta_1 w_1) \cap \text{FIRST}_k(\eta_2 w_2) = \emptyset$.

Having provided alternative definitions for the well-known LR(k) and LALR(k) subclasses of CFGs, we are now ready to define analogous subclasses of LRP(k) ("LR(k) with Precedence") and LALRP(k) ("LALR(k) with Precedence") for ordered context-free grammars. The definitions for LRP(k) and LALRP(k) that follow are almost identical to those for LR(k) and LALR(k), except that each NPDA state ${}_P\mathfrak{I}_\alpha$ is now viewed as a partial order of items rather than an unordered set. As one may expect, this additional ordering of items may make it "harder" for any two states ${}_P\mathfrak{I}_\alpha^k$ and ${}_P\mathfrak{I}_\beta^k$, $k \geq 0$ to be deemed equivalent, and hence foldable into the same DPDA state. A deterministic parser constructed for an OCFG may therefore contain more states than one for a corresponding core CFG. In practice, it is somewhat difficult to find an OCFG G for which the number of DPDA states is greater than for CORE(G); Section 7 elaborates on this matter further.

Definition 13. Let $G = (V_N, V_T, S, P)$ be an OCFG per Definition 3. We say that $G \in \text{LRP}(k)$, for $k \geq 0$, if and only if for any two states α, β in the NPDA for G, the conditions

(1) $[A \longrightarrow \mu_1 \bullet \eta_1 : a \,|\, w_1] \in {}_P\mathfrak{I}_\alpha^k$;
(2) $[B \longrightarrow \mu_2 \bullet \eta_2 : b \,|\, w_2] \in {}_P\mathfrak{I}_\beta^k$;
(3) ${}_P\mathfrak{I}_\alpha^k = {}_P\mathfrak{I}_\beta^k$; and

(4) $[A \longrightarrow \mu_1 \bullet \eta_1 : a \,|\, w_1] \neq [B \longrightarrow \mu_2 \bullet \eta_2 : b \,|\, w_2]$

jointly imply that $\text{FIRST}_k(\eta_1 w_1) \cap \text{FIRST}_k(\eta_2 w_2) = \emptyset$.

Definition 14. Let $G = (V_N, V_T, S, P)$ be an OCFG per Definition 3. We say that $G \in \text{LALRP}(k)$, for $k \geq 0$, if and only if for any two states α, β in the NPDA for G, the conditions

(1) $[A \longrightarrow \mu_1 \bullet \eta_1 : a \,|\, w_1] \in {}_P\mathfrak{I}^k_\alpha$;
(2) $[B \longrightarrow \mu_2 \bullet \eta_2 : b \,|\, w_2] \in {}_P\mathfrak{I}^k_\beta$;
(3) ${}_P\mathfrak{I}^0_\alpha = {}_P\mathfrak{I}^0_\beta$; and
(4) $[A \longrightarrow \mu_1 \bullet \eta_1 : a \,|\, w_1] \neq [B \longrightarrow \mu_2 \bullet \eta_2 : b \,|\, w_2]$

jointly imply that $\text{FIRST}_k(\eta_1 w_1) \cap \text{FIRST}_k(\eta_2 w_2) = \emptyset$.

Interestingly, even though a grammar $G \in \text{LRP}(k)$, this does *not* necessarily mean that $\text{CORE}(G) \in \text{LR}(k)$! Recall that the allowable derivations in an OCFG must be a subset of those in a CFG (see Definition 5); many CFG derivations do not respect the appropriate invariants for associativity and precedence, and are pruned from the set. A lesser number of possible derivations for a sentence translates into fewer conflicts during deterministic parsing of that sentence. Conversely, a core grammar $G' = \text{CORE}(G)$ may contain shift-reduce and/or reduce-reduce conflicts even if the original OCFG G does not so that, in fact, $G' \notin \text{LR}(k)$. The same argument can easily be made for for LALRP(k) and LALR(k) grammars; one of the strengths of the Bertha parser generator, discussed in Section 6, lies in its ability to construct LALRP(1) parsers for grammars that are not in LALR(1).

5 Parser Construction for LRP(k) and LALRP(k) Grammars

In the previous section, we have presented axiomatic definitions for LR(k), LRP(k), LALR(k) and LALRP(k) grammars, phrased in terms of an abstract NDPA. As concise as these definitions are, they fall somewhat short in guiding the design and construction of actual parsers for these grammars. Deterministic parsing algorithms for LR(k) and LALR(k), especially when $k = 1$, have been a subject of extensive research in the past and we will not analyze them here. Instead, we shall outline the parser construction techniques for LRP(k) and LALRP(k) grammars on the assumption that the reader is already familiar with the corresponding LR(1) and LALR(1) techniques. In what follows, we informally outline the phases that the parser construction algorithm should consist of. These phases were chosen to allow for a separation of concerns within the algorithm, and to make it as understandable as possible. Clearly, the objectives of our exposition may conflict with the objective of parsing efficiency, a topic which has successfully been addressed elsewhere [Pag77, DP82, Spe88, BL89]. An actual implementation of the LALRP(k) parser construction algorithm (for $k = 1$) may be found in the Bertha™ tool, discussed in Section 6 below.

Obviously, the parser construction process for any of the grammars discussed in Section 4 does *not* involve the actual creation of an entire NPDA and its subsequent reduction to a DPDA, since the number of distinct NPDA states can potentially be infinite. Each NPDA state $\mathfrak{I}_\alpha^k$ or ${}_P\mathfrak{I}_\alpha^k$, as it is computed, is instead immediately folded into an appropriate DPDA state $\mathfrak{I}_\beta^k$ or ${}_P\mathfrak{I}_\beta^k$, respectively, per equivalence criterion (3) in Definitions 11, 12, 13 or 14. During the folding operation, the set of items comprising the NPDA state is merged into the set of items in the DPDA state: $\mathfrak{I}_\beta^k \leftarrow \mathfrak{I}_\beta^k \cup \mathfrak{I}_\alpha^k$ or ${}_P\mathfrak{I}_\beta^k \leftarrow {}_P\mathfrak{I}_\beta^k \cup {}_P\mathfrak{I}_\alpha^k$. For LALR($k$) or LALRP($k$) grammars, this may actually enlarge $|\mathfrak{I}_\beta^k|$ or $|{}_P\mathfrak{I}_\beta^k|$ due to differing lookaheads.

If no appropriate $\mathfrak{I}_\beta^k$ or ${}_P\mathfrak{I}_\beta^k$ DPDA state exists, one is simply created from $\mathfrak{I}_\alpha^k$ or ${}_P\mathfrak{I}_\alpha^k$. After this is done, all applicable shift transitions out of this new state are computed as well. All shift actions correspond to either a valid k-tuple on the input stream which needs to be read in, or to a k-tuple of symbols found on the top of the parse stack. In any event, we need not consider the reduction actions at this phase of parser construction, since they cannot lead to new DPDA states. At the same time, shift actions *per se* cannot lead to conflicts during our transition computation.

Once the DPDA state closure is complete, we may compute the reduce transitions that are applicable in each state. It is only during this computation that shift-reduce and reduce-reduce conflicts may arise. In the case of LR(k), LALR(k) or some other subset of CFGs, such conflicts cannot be resolved, as no ordering of any kind exists among CFG items in each state. Instead, the conflicts must be eliminated altogether through a rewriting of the grammar (or the use of ad hoc directives allowable by tools such as `yacc` or `bison`). But for OCFG subsets such as LRP(k) or LALRP(k), the OCFG items in each state *do* possess an ordering (see Definition 8), reflecting the ordering of their underlying OCFG derivations. When confronted with a conflict, we simply choose the action which corresponds to applying the OCFG item with the higher precedence. Furthermore, some of the OCFG items may not need to be consulted at all, since their inclusion would lead to an invalid derivation tree costruction.

Should a shift-reduce conflict involve two OCFG items with equal precedence, we check if the associativity vectors of the corresponding productions allow us to consistently exclude one of the actions when constructing a valid derivation. As an example, consider a subset of the grammar depicted in Figure 1 containing only productions $p_1 = [E \longrightarrow E\text{“*”}E : 10]$ and $p_2 = [E \longrightarrow E\text{“/”}E : 00]$, where $p_1 =_{\mathrm{P}} p_2$. Of the different states in the resulting LALRP(1) DPDA for this grammar, two are of interest here. The first, which we shall label $\mathcal{S}_1$, contains the partial order

$$\begin{aligned}[E \longrightarrow E\text{“*”}E\bullet : 10 \,|\, \{\text{“}*\text{”}, \text{“}/\text{”}\}] &=_{\mathrm{rm\,P}} [E \bullet \longrightarrow E\text{“*”}E : 10 \,|\, \{\text{“}*\text{”}, \text{“}/\text{”}\}] \\ &=_{\mathrm{rm\,P}} [E \bullet \longrightarrow E\text{“/”}E : 10 \,|\, \{\text{“}*\text{”}, \text{“}/\text{”}\}]\end{aligned}$$

of OCFG items. The second, labeled $\mathcal{S}_2$, contains items

$$[E \longrightarrow E \text{“/”} E\bullet : 10 \,|\, \{\text{“}*\text{”}, \text{“/”}\}] =_{\text{rm P}} [E \bullet \longrightarrow E \text{“*”} E : 10 \,|\, \{\text{“}*\text{”}, \text{“/”}\}]$$
$$=_{\text{rm P}} [E \bullet \longrightarrow E \text{“/”} E : 10 \,|\, \{\text{“}*\text{”}, \text{“/”}\}],$$

again arranged into a partial order. Because all the items in $\mathcal{S}_1$ and $\mathcal{S}_2$ have identical precedence, the shift-reduce conflicts present in both states cannot be resolved using precedence alone. We therefore consult the associativity vectors for p_1 and p_2 in the hope of finding a possible resolution.

First, observe that p_1 is left-associative; thus, while in $\mathcal{S}_1$, if faced with a choice of reducing using p_1 or shifting using p_1 (as indicated by a lookahead of “*”), we should clearly reduce. However, if the lookahead is “/”, then the shift operation will involve p_2 rather than p_1. We can no longer reduce with p_1 at this point; had we done so, the resulting sentential form would need to be subsequently reduced with p_2, which is not possible since p_2 is non-associative (see Definition 5). We also cannot shift out of $\mathcal{S}_1$ using p_2; doing this would eventually lead to a reduction into the rightmost E in p_1, violating the left-associativity of p_1. Hence, the proper action on lookahead “/” in state $\mathcal{S}_1$ is to signal an associativity error.[21] Similar reasoning is applied when determining plausible transitions out of $\mathcal{S}_2$. Given a lookahead of “*”, we reduce the stack using p_2, yielding the leftmost occurrence of E in p_1. If the lookahead is “/”, we again signal an associativity error condition. Although we succeeded in eliminating all of the shift-reduce conflicts in this example, this need not be true in the general case. For example, a directional DPDA parser cannot resolve shift-reduce conflicts when neither production is left- or right-associative (i.e., when both a_1 and a_2 are of the form $0^m 1^{rc(A,\omega)-m-n} 0^n$, for some $m \geq 1$ and $n \geq 1$), unless unbounded lookahead is permitted.[22]

In all cases — LR(k), LALR(k), LRP(k) or LALRP(k) — the foregoing procedure will yield a bottom-up parser capable of deciding if an input sentence is enumerated by the corresponding grammar. An OCFG parser may have more states than a corresponding CFG parser; since OCFG states are partial orders (see Section 4 above), several distinct orderings of the same set of OCFG items may well map to a single CFG state. At the same time, LR(k) and LRP(k) parsers will usually have more states than their LALR(k) and LALRP(k) counterparts — the latter always rely on an LR(0) automaton — although the number of additional states can be kept to a minimum using algorithms presented in [Pag77, Spe88]. Generally, the resulting parsers — whether table-driven or code-driven — will be slightly larger for OCFGs than for CFGs. Not only may the total number of states in OCFG parsers be greater, but improved conflict resolution may lead to a greater number of encoded actions within each state as well. The increase in size should properly be viewed as a consequence of

[21] Associativity errors differ from shift-reduce conflicts in that *neither* action (as opposed to *both*) is valid in this state. The `bison` and `yacc` tools also make this distinction, whereas Berkeley yacc (`byacc`) `does not`.

[22] Presumably, non-directional parsing methods such as CYK [GJ98] could be used instead, though this has not been formally investigated.

increased recognition power, and not as a drawback of LRP(k) and LALRP(k) parser construction techniques.

6 The Bertha™ Parser Generator

Bertha is a parser generator for LALRP(1) grammars. It relies extensively on the Java™ [GJS96] programming language — not only for its implementation, but also for the syntax, structure and semantics of acceptable input grammars. The Bertha input language is merely a simple extension of Java that introduces additional keywords such as `prec`, `symbol` and `reduce`, among a few others. The advantage of extending an existing source language in this way, aside from an obviously simplified learning curve, is that one can easily leverage many of the language's desirable properties. In the case of Bertha, all grammar specifications are properly encapsulated and completely type-safe. The fact that the input language is based on Java also greatly simplifies the generation of the output — which, not surprisingly, is a Java program itself.

Figure 2 contains the source code for a grammar specification — complete with semantic actions — processable by the Bertha parser generator. This specification corresponds to the grammar shown in Figure 1. By convention, each Bertha input file must possess the extension `.bertha`, and its base name must be identical to that of the outermost start symbol (`E` in our case) contained in it. The `.java` output file created by Bertha will also share this base name. It should be fairly obvious that the `start symbol E` shown in Figure 2 is very similar to an ordinary Java `class`; in fact, each Bertha `symbol` is converted to a `class` when the output file is written.

The remaining `symbol`s are also implemented as classes, and are nested inside the `start symbol`s of their respective grammars. They may also be nested inside of each other. In this way, every symbol may access attributes and methods defined in all enclosing symbols — including the `start symbol` — and hence facilitate the use of inherited attributes. This can be seen in Figure 2, where the `reduce()` routine of the `num` symbol is directly accessing the `val` field of the `E` symbol. (Bertha and Java will both ensure that an outer symbol exists before any inner symbol is instantiated in its scope.) It is important to note that traditional bottom-up parsers cannot support inherited attributes as it is not known which production is in the process of being completed; the availability of inherited attributes in Bertha-generated parsers must therefore come at the expense of the overall expressivity of the grammar. In such cases, inner nonterminals (such as `symbol num` shown in Figure 2) may be referenced only by productions defined in the scope of the enclosing symbol. Indeed, it is possible to devise grammars that are essentially *top-down* in nature, and which expose the attributes of *all* enclosing symbols during the parse process.

All Bertha `symbol`s are nonterminals, and must have one or more productions, in the form of `reduce(...)` methods, defined. Terminal symbols are not declared explicitly; rather, they occur as anonymous literals and regular expressions and are passed to their respective `reduce(...)` methods as Java `String`s.

```
// E.bertha
import zll.bertha2.run.ParseError;
start symbol E {
  // the program processes what is on the command line
  public static void main(String args[]) {
    E calc = new E(new java.io.StringReader(ArgString(args)));
    calc.Parse(); System.out.println("The result is " + calc.val);
  }
  int val = 0; // attribute for nonterminal E
  symbol num { // nested terminal symbol
    reduce(('0'-'9')+ str) {
      val = Integer.decode(str);
    }
  }
  // back to the scope of E
  prec(incomparable) {
    reduce(num) { /* no need to do more! */ }
    prec(decreasing) {
      prec(equal) {
        assoc("0") reduce("-", E s) { val = - s.val; }
        assoc(none) reduce("+", E s) { val = + s.val; }
      }
      assoc("01") reduce(E base, "^", E exponent) {
        val = (int)Math.pow(base.val, exponent.val);
      }
      prec(equal) {
        assoc(left) reduce(E op1, "*", E op2) { val = op1.val * op2.val; }
        assoc(none) reduce(E op1, "/", E op2) { val = op1.val / op2.val; }
      }
      prec(equal) {
        assoc("10") reduce(E op1, "+", E op2) { val = op1.val + op2.val; }
        assoc("00") reduce(E op1, "-", E op2) { val = op1.val - op2.val; }
      }
      assoc(right) reduce(E op1, ":", E op2, "?", E op3) {
        val = (op1.val != 0? op2val: op3val);
      }
    }
    assoc(full) reduce("(", E op, ")") { val = op.val; }
  }
}
```

Figure 2: A sample Bertha grammar specification (`E.bertha`) for the OCFG depicted in Figure 1

For example, `symbol num` in Figure 2 above relies on a string parameter `str`, constructed by matching input stream characters against the regular expression `(0́-9́)+`. This approach greatly simplifies the integration of lexical analysis with parsing, and obviates the necessity of using separate lexical analyzers such as `lex` or `flex`.

Nonterminal `symbol`s usually define `reduce(...)` methods via an enclosing `prec(...)` construct specifying the precedence ordering of the productions contained within it. For example, to specify a linear order of increasing or decreasing precedences, the `reduce(...)` methods must be placed inside a `prec(increasing)` or `prec(decreasing)` scope, respectively. Similarly, to specify a precedence equivalence class for a group of productions, one should surround the corresponding methods with a `prec(equal)` construct. Finally, a lack of an ordering relation among several productions may be indicated by placing them inside of a `prec(incomparable)` scope. In some cases, it is possible for `prec(...)` constructs to be nested,[23] as Figure 2 illustrates. It is possible for a `symbol` not to use `prec(...)` at all, which is semantically equivalent to using `prec(incomparable)`.

Each `reduce(...)` method definition may be preceded by an `assoc(...)` clause. If omitted in the source, this clause is assumed to be `assoc(full)`, thus making the corresponding production fully associative. In addition to `full`, production associativites may be specified as `left`, `right`, `none` or as a Boolean vector (entered as a string literal) described in Definition 3. For non-recursive productions (i.e., where $rc(A, \omega) = 0$), the `assoc(...)` clause has no valid semantics and may not be specified. Note that if neither `prec(...)` nor `assoc(...)` are used, the resulting Bertha grammar is an ordinary CFG without any notion of ordering.

Although Figure 2 does not show it, it is also possible for a `start symbol` to have another `start symbol` — corresponding to a separate grammar — nested within it. In such a situation, both grammars share the underlying parser engine as well as the input stream. A sub-parse using the nested grammar is performed whenever the outer grammar is *about* to retrieve a terminal symbol, whether by shifting or lookahead, from the input. A typical application of nested grammars is in the construction of parsers for programming languages, where the outer grammar describes the actual language constructs while the inner grammar lays out the structure of comments and other white space. No longer limited by a finite-state acceptor such as that found in `lex` or `flex`, one can easily parse nested comments and extract comment-embedded content such as `javadoc` [GJS96] documentation.

Also not shown is the fact that all `symbol`s are free to inherit fields and methods from other Java classes. This inheritance behavior is rather easy to implement as a consequence of using an object-oriented language as the basis for Bertha, and can be used to factor out similarities among various constructs. For example, the nonterminals `int_expr`, `real_expr` and `bool_expr` in some

[23] Note that it is not possible to nest `prec(increasing)`, `prec(decreasing)` or `prec(incomparable)` scopes inside a `prec(equal)` construct — doing so would violate the partial order invariants.

$[S \longrightarrow A : \Lambda]$ $[A \longrightarrow C : \Lambda]$ $[C \longrightarrow \text{“A”} : \Lambda]$

$[S \longrightarrow \text{“!”}\ B : \Lambda]$ $[A \longrightarrow D : \Lambda]$ $[C \longrightarrow \text{“B”} : \Lambda]$

$[B \longrightarrow D : \Lambda]$ $[D \longrightarrow \text{“B”} : \Lambda]$

$[B \longrightarrow C : \Lambda]$ $[D \longrightarrow \text{“A”} : \Lambda]$

Figure 3: A contrived OCFG requiring additional DPDA states

hypothetical language can all inherit common characteristics from an `expr` base class. More generally, the consistent use of classes in conjunction with the type-safety of Java implies that the parsers constructed by **Bertha** are themselves type-safe. Java inner classes provide a convenient closure mechanism through which further encapsulation and separation of concerns may be achieved. Both type-safety and encapsulation are essential for the construction of reliable software systems, yet are missing from popular parser generators such as `bison` and `yacc`.

7 Conclusions and Future Work

In this paper, we have presented a new, ordered variant of context-free grammars capable along with the means to generate the context-free languages they describe and to construct parsers for them. As an initial work, the paper consisted mostly of definitions; theoretical results pertaining to OCFGs, LRP(k) and LALRP(k) are yet to be obtained. In particular, it would be interesting to obtain a tight upper bound on the number of distinct DPDA states in a parser constructed for an LRP(k) or LALRP(k) grammar. On the one hand, it seems reasonable to assume that the number of states may be greater than those obtained for a corresponding core LR(k) or LALR(k) grammar, since there may exist multiple states containing the same items whose ordering differs. However, in the course of constructing **Bertha** grammars for various programming languages, we have almost never encountered such a situation.

A very contrived LALRP(1) grammar which does require two additional states beyond those of an LR(0) automaton is shown in Figure 3. After scanning in token `‘‘A’’`, the parser must choose whether to reduce it to a `C` or a `D`. This, however, depends on whether the `C` or `D` is to be derived from an `A` or a `B`. If an `A` is being derived, then `‘‘A’’` should clearly be reduced to a `C`, as $[A \longrightarrow C : \Lambda]$ has higher precedence than $[A \longrightarrow D : \Lambda]$. Conversely, if a `B` is being derived, then one must reduce the `‘‘A’’` to a `D` since $[B \longrightarrow D : \Lambda] >_{\text{P}} [B \longrightarrow C : \Lambda]$. Clearly, *two* distinct states, rather than one, must exist for reducing `‘‘A’’`; the same holds true for token `‘‘B’’`.

On the whole, then, it may be possible that on average, the number of states for a shift-reduce parser for an OCFG only barely exceeds that for the corresponding core CFG, even if the theoretical upper bound is less than promising.

This issue remains to be investigated. Of course, given the increased expressivity afforded by ordered context-free grammars, any additional states may construe an acceptable cost. There exists an analogy between creating such states in an LALRP(k) parser generator and the splitting of inadequate states [24] when constructing parsers for grammars that fail to be in LALR(k). In both cases, the underlying LR(0) automaton proves insufficient. It may therefore be advantageous to leverage both techniques in a single system. Bertha, when augmented with state-splitting or state-merging algorithms presented in [Spe88, Pag77], could become a full-fledged LR(1)/LRP(1) parser generator, but without the state explosion usually associated with LR(1) parsers. In addition, any remaining inadequate states could be equipped with *syntactic predicates* [Tar82] to that choose the appropriate action. Making such enhancements could become reasonable if the LALRP(1) formalism proves insufficiently expressive in practice.

The existence of nested `symbol`s in Bertha grammars may provide another opportunity for future improvements. Because all inner symbols have limited visibility and scope, additional context information may be inferred when these symbols are created. For example, since any enclosing symbols will eventually be instantiated, Bertha performs this instantiation "ahead of time" (i.e., with respect to traditional bottom-up parsing) and hence allows inherited as well as synthesized attributes to be used.[25] To the extent that we are able to deduce which productions (items) will need to be completed, we may be able to incorporate left-corner parsing techniques [RL70, Hor93] into the Bertha runtime system. Generally speaking, a left-corner parser proceeds in a bottom-up fashion until it can determine a unique production, or a unique set of nested productions, which must be completed next. If such production or productions are found, they are parsed in a top-down manner.

Undoubtedly, many other improvements to Bertha, and perhaps even the ordered context-free grammar formalism itself, are possible. Most likely, the exact nature of such changes will come to light only after extensive use of OCFGs in real-life software construction scenarios.

8 Acknowledgements

The author would like to thank his advisor, Prof. Dan Hirschberg, for reviewing this paper and offering numerous suggestions as to its content and (English) grammar. Thanks are also in order to fellow graduate students Peter Fröhlich and Christian Stork, and to Prof. Michael Franz, for volunteering their time to discuss OCFGs and related material, and for pointing out the occasional — and probably inevitable — shortcomings in the author's thinking and reasoning. Finally, the author is indebted to Chris F. Clark, as well as to several anonymous

[24] Or alternately, the merging of superfluous LR(1) states.

[25] While a grammar may always be rewritten to make use of synthesized attributes alone [Set96], inherited attributes are a very convenient abstraction for specifying information flow within a parse tree.

reviewers, for suggesting numerous improvements to the original version of this paper.

References

[Aas95] Annika Aasa. Precedences in specifications and implementations of programming languages. *Theoretical Computer Science*, 142(1):3–26, 1 May 1995.

[Aho68] Alfred V. Aho. Indexed grammars — an extension of context-free grammars. *Journal of the ACM*, 15(4):647–671, October 1968.

[AJU75] Alfred V. Aho, Stephen C. Johnson, and Jeffrey D. Ullman. Deterministic parsing of ambiguous grammars. *Communications of the ACM*, 15(8):441–452, August 1975.

[App98] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, Cambridge, United Kingdom, 1998.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, MA, March 1986.

[AU72] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation and Compiling: Volume I: Parsing*. Prentice-Hall Series in Automatic Computation. Prentice-Hall, Englewood Cliffs, NJ, 1972.

[Bea95] Steven S. Beaty. Parsesrap: Using one grammar to specify both input and output. *ACM SIGPLAN Notices*, 30(2):25–32, February 1995.

[BL89] Manuel E. Bermudez and George Logothetis. Simple computation of LALR(1) lookahead sets. *Information Processing Letters*, 31(5):233–238, 12 June 1989.

[BPSM98] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. Technical report, World Wide Web Consortium, 10 February 1998. Available at http://www.w3.org/TR/REC-xml.

[BS90] Manuel E. Bermudez and Karl M. Schimpf. Practical arbitrary lookahead LR parsing. *Journal of Computer and System Sciences*, 41(2), October 1990.

[CC73] Karel Culik II and Rina S. Cohen. LR–Regular grammars — an extension of LR(k) grammars. *Journal of Computer and System Sciences*, 7(1):66–96, February 1973.

[Chr99] Thomas W. Christopher. User manual for TCLLk: A strong LL(k) parser generator and parser. Technical Report 1999-3-#1-TC, Tools of Computing LLC, Evanston, IL, 12 March 1999.

[Col70] Alain Colmeauer. Total precedence relations. *Journal of the ACM*, 17(1):14–30, January 1970.

[DP82] Frank Deremer and Thomas Pennello. Efficient computation of LALR(1) look-ahead sets. *ACM Transactions on Programming Languages and Systems*, 4(4):615–649, October 1982.

[Ear70] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, February 1970.

[Ear75] Jay Earley. Ambiguity and precedence in syntax description. *Acta Informatica*, 4(2):183–192, 1975.

[FK97] Michael Franz and Thomas Kistler. Slim Binaries. *Communications of the ACM*, 40(12):87–94, December 1997.

[Flo63] Robert W. Floyd. Syntactic analysis and operator precedence. *Journal of the ACM*, 10(3):316–333, July 1963.

[Fri68] Ivan Frič. Grammars with partial ordering of the rules. *Information and Control*, 12(5/6):415–425, May–June 1968.

[Fri69] Ivan Frič. Errata: Grammars with partial ordering of the rules. *Information and Control*, 15(5):452–453, November 1969.

[GH93] John V. Guttag and James J. Horning. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993. With Stephen J. Garland, Kevin D. Jones, Andrés Modet and Jeannette M. Wing. Available at `http://www.sds.lcs.mit.edu/spd/larch/pub/`.

[GJ98] Dick Grune and Ceriel J. Jacobs. *Parsing Techniques: A Practical Guide*. Printout by the Authors, Vrije Universiteit, Amsterdam, the Netherlands, September 1998. Available at `http://www.cs.vu.nl/~dick/PTAPG.html`.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java^{TM} Language Specification*. Addison-Wesley, Reading, MA, 1996. Available at `http://www.javasoft.com/docs/books/jls/html/index.html`.

[Gru71] Jozef Gruska. Complexity and unambiguity of context-free grammars and languages. *Information and Control*, 18(5):502–519, June 1971.

[HHKR89] Jan Heering, Paul R. H. Hendriks, Paul Klint, and Jan Rekers. The syntax definition formalism SDF — reference manual. *ACM SIGPLAN Notices*, 24(11):43–75, November 1989.

[Hor93] R. Nigel Horspool. Recursive ascent-descent parsing. *Computer Languages*, 18(1):1–15, January 1993.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Series in Computer Science. Addison-Wesley Publishing Co., 1979.

[Hud99] Scott E. Hudson. *CUP – LALR Parser Generator for Java^{TM}*, July 1999. With Frank Flannery, C. Scott Ananian and Dan Wang. Available at http://www.cs.princeton.edu/~appel/modern/java/CUP/.

[IK97] Sandy Irani and Anna R. Karlin. On online computation. In Dorit S. Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, chapter 13. PWS Publishing Company, Boston, MA, 1997.

[Knu65] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, December 1965.

[LMB92] John Levine, Tony Mason, and Doug Brown. *Lex & Yacc*. O'Reilly & Associates, Inc., Sebastopol, CA, second edition, October 1992.

[MP72] J. McAfee and L. Presser. An algorithm for the design of simple precedence grammars. *Journal of the ACM*, 19(3):385–395, July 1972.

[Pag77] David Pager. A practical general method for constructing LR(k) parsers. *Acta Informatica*, 7:249–268, 1977.

[PQ96] Terence J. Parr and Russell W. Quong. LL and LR translators need $k > 1$ lookahead. *ACM SIGPLAN Notices*, 31(2):27–34, February 1996.

[RL70] Daniel J. Rosenkrantz and Philip M. Lewis II. Deterministic left corner parsing (extended abstract). In *Conference Record of 1970 Eleventh Annual Symposium on Switching and Automata Theory*, pages 139–152, Santa Monica, California, 28–30 October 1970. IEEE.

[Ros69] Daniel J. Rosenkrantz. Programmed grammars and classes of formal languages. *Journal of the ACM*, 16(1):107–131, January 1969.

[Sal73] Arto Salomaa. *Formal Languages*. ACM Monograph Series. Academic Press, New York, NY, 1973.

[Set96] Ravi Sethi. *Programming Languages: Concepts and Constructs*. Addison-Wesley, Reading, MA, second edition, 1996.

[Spe88] David Spector. Efficient full LR(1) parser generation. *ACM SIGPLAN Notices*, 23(12):143–150, December 1988.

[Tar82] Jorma Tarhio. LR parsing of some ambiguous grammars. *Information Processing Letters*, 14(3):101–103, 16 May 1982.

[Tho94] Mikkel Thorup. Controlled grammatic ambiguity. *ACM Transactions on Programming Languages and Systems*, 16(3):1024–1050, May 1994.

[Woo87] Derick S. Wood. *Theory of Computation*. Harper & Row, New York, NY, first edition, 1987.