More Intuitive Bottom-Up Parsing

Ziemowit Laski

Technical Report 99–48

University of California, Irvine
Irvine, CA 92697-3425

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

October 28, 1999
Updated: November 10, 1999

Abstract

Although bottom-up parsing techniques are sufficiently powerful for most compiler construction tasks, the crafting of suitable grammar specifications can be a difficult and elusive process. The lack of proper abstraction mechanisms in specification metalanguages themselves often exacerbates these difficulties. We present Bertha, a prototype parser generator designed to address these and other concerns. Bertha is built upon the notion of *ordered context-free grammars*, a formal and systematic augmentation of context-free grammars with associativity and precedence information, that affords it greater expressivity. Furthermore, Bertha's input metalanguage allows for hierarchical grammar and symbol definitions, facilitating the use of closures and inherited attributes. Finally, all grammar specifications are type-safe and properly encapsulated, and represent a minimal departure from the underlying Java™ language.

1 Introduction

Context-free grammars (CFGs) are an indispensable part of modern software engineering practice, since they facilitate formal yet easily understood specifications of languages. Typically, such specifications are used as inputs to parsing algorithms. Numerous approaches to context-free language parsing have been proposed over the years, each constituting a different trade-off between expressivity and computational complexity. For purposes of compiler construction, one is typically restricted to *directional* parsing algorithms [GnJ98] employing

some fixed (usually single-token) lookahead; these are further subdivided into *top-down* and *bottom-up* algorithms.

The principal advantage of top-down parsers, such as those for LL(1) grammars, lies in their simplicity and appeal to user intuition. Not only is it easy to understand how a top-down parsing algorithm works, but also quite straightforward to implement a predictive parser using recursive descent. Top-down parsing techniques can be adequate for handling small languages such as Oberon-2 [MW91], but do not scale well to more complex languages. Restrictions on LL(1) grammars, such as the lack of left-recursion or distinct left factors [ASU86], make it very difficult and impractical to craft suitable grammar specifications in many cases.

Bottom-up parsers are far more versatile in this regard. The general LR(1) class of grammars [Knu65] is sufficient to describe *any* deterministic context-free language, although parsers created from them tend to be quite large. Its LALR(1) subclass [Der69] remains very expressive, yet achieves a substantial reduction in parser size. For this reason, LALR(1) parsers (and parser generators, such as `yacc` and its derivatives) have found widespread use in the software engineering community.

Unfortunately, devising specificationss suitable for bottom-up parser generation can be a daunting task. Programmers have far less intuition to guide them than in the case of top-down specifications, since it is not at all clear what LR(1) or LALR(1) grammars should "look like." Consequently, should a tool such as `yacc` reject a grammar specification, the programmer may find it nearly impossible to find an elegant solution to the problem, opting instead to duplicate portions of the grammar to achieve disambiguation. This results in code bloat which adversely impacts system maintainability.

Alternatively, one may mechanically rewrite the problematic grammar into some *normal form* (e.g., Chomsky Normal Form, Greibach Normal Form) that is guaranteed to exist and whose amenability to parsing is known [Woo87, Chr99]. This approach is satisfactory in situations where one is interested solely in constructing a language *recognizer*, i.e., an automaton capable of deciding whether a given input sentence is part of a language. However, there is much more to parsing than mere recognition of sentences [PQ96]. For a parser to perform useful work, *semantic actions* need to be inserted at appropriate places in the grammar specification. This placement of semantic actions can be seriously hampered after the grammar is rewritten since the logical structure of the underlying language is invariably obfuscated in the process.

Rather than rewriting a grammar specification to remove ambiguities, it is often possible to *augment* it with additional semantic information [AJU75, Ear75] so as to guide the parser generation algorithm. For example, the `yacc` [LMB92] formalism allows the user to specify associativities and precedences of various terminal symbols; this information is then used to implicitly (or explicitly, via the `%prec` directive) resolve shift/reduce conflicts between productions and tokens in the grammar. This approach works fine if the number of conflicts is very small, but becomes burdensome soon thereafter. It is also an ad hoc solution in that it lacks a good formal grounding. Few properties can reasonably be

inferred about grammars annotated in this manner, save perhaps for the fact that they can be handled by a deterministic parser.

This paper presents **Bertha**, a parser generator capable of processing a subset of *ordered context-free grammars* (OCFGs) [Las99]. Ordered context-free grammars are an extension of traditional context-free grammars that systematically includes associativity and precedence information as part of the core formalism. They are described in Sect. 2 below. In Sect. 3, we introduce the **Bertha** tool itself. In its current form, **Bertha** handles ordered context-free grammars belonging to the LALRP(1) subset; LALRP(1) is analogous to, and a strict superset of, LALR(1) grammars. Section 4 demonstrates how the **Bertha** metalanguage may be used to describe various language constructs in a far more straightforward and intuitive manner than is possible using `yacc`, even when the latter is augmented with aforementioned annotations. Finally, Sect. 5 discusses some possible future improvements to the **Bertha** tool and concludes the paper.

It is assumed that the reader is familiar with bottom-up parsing techniques and terminology such as LR(0), LALR(1) and LR(1). A good introduction to the subject is contained in [App98], while [ASU86] can serve as a comprehensive reference.

2 An Overview of Ordered Context-Free Grammars

Our primary motivation for creating ordered context-free grammars (OCFGs) has been to thoroughly and systematically integrate associativity and precedence rules into the context-free grammar formalism. In this paper, we describe OCFGs only to the extent necessary to gain an understanding of how the **Bertha** parser generator works; the reader is referred to [Las99] for a more rigorous treatment of the topic.

Ordered context-free grammars differ from ordinary CFGs in two respects. First, all productions in the grammar sharing the same left part (i.e., deriving the same nonterminal symbol) are arranged into a *partial order*[1] of precedence. For example, in a language denoting arithmetic expressions, the production $E \rightarrow E * E$ will most likely be assigned a higher precedence than $E \rightarrow E + E$.

Secondly, each production has a *recursion count* associated with it. For an arbitrary production $A \rightarrow \omega$, its recursion count $rc(A, \omega)$ equals the number of occurrences of nonterminal A in the sentential form ω. For example, $rc(E, E * E) = rc(E, E + E) = 2$. Every production must then be assigned a Boolean *associativity vector* whose length equals the recursion count. If some $A \rightarrow \omega$ is assigned the vector a, where $|a| = rc(A, \omega)$, then we may simply write $A \rightarrow \omega : a$. The use of the associativity vector will be explained shortly.

[1] A partial order is understood here as a reflexive, transitive and antisymmetric relation $\leq_P$ on the set of productions with the same left part. The relation is partial since some pairs of productions may not be comparable. Furthermore, productions with differing left parts must always be incomparable; see [Las99] for an explanation of this.

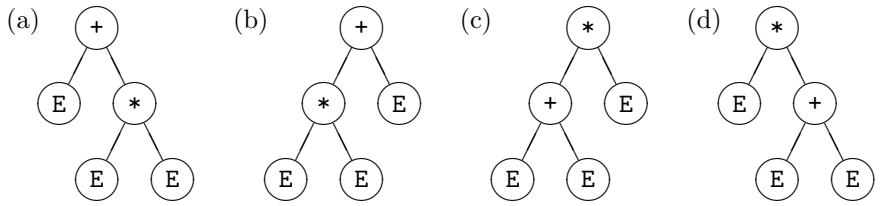

Figure 1: Computationally valid (a, b) and invalid (c, d) syntax trees for arithmetic expressions $E + E * E$ and $E * E + E$

Recall that the precedence and associativity annotations allowed by `yacc` are ad hoc in nature, and are of use only when shift-reduce or reduce-reduce conflicts arise. By contrast, the partial orders and associativity vectors present in OCFGs serve to control both parsing and *generation* of sentences. Both of these objectives are accomplished through a single mechanism — namely, by requiring that the syntax trees constructed for sentences in the language observe the invariants of *precedence correctness* [Aas95].

To illustrate this, let us return to the productions $E \rightarrow E * E$ and $E \rightarrow E + E$ introduced previously. We have assigned a higher precedence to the first production since multiplication traditionally enjoys precedence over addition. Given the arithmetic expressions $E + E * E$ and $E * E + E$, multiplication must clearly take place first, which corresponds to syntax trees (a) and (b) in Fig. 1. Note that trees (c) and (d) represent invalid computations given our stated precedences. These observations lead us to formulate the following invariant for precedence correct syntax trees:

Rule 1. If two adjacent nodes in an OCFG syntax tree correspond to productions with the same left part, then the bottom production must have precedence that is *neither lower nor equal* than that of the top production.

In other words, the bottom production must either have higher precedence than the top production, share the same precedence level, or the two productions must be incomparable. This rule is motivated at length and stated more formally in [Las99].

Now let us consider the expressions $E + E + E$ and $E * E * E$. The possible syntax syntax trees corresponding to these expressions are shown in Fig. 2. According to Rule 1, none of them are computationally valid since the bottom production has the same precedence as the top production in each case. Clearly, this is not what we intended, since repeated application of multiplication and addition in arithmetic expressions is obviously allowed. On the other hand, if we assume that both addition and multiplication must be evaluated from left to right,[2] syntax trees (c) and (d) truly cannot be valid since they impose an

[2]Such an assumption may be a convenient one to make for a programming language, especially if constituent subexpressions are allowed to produce side-effects.

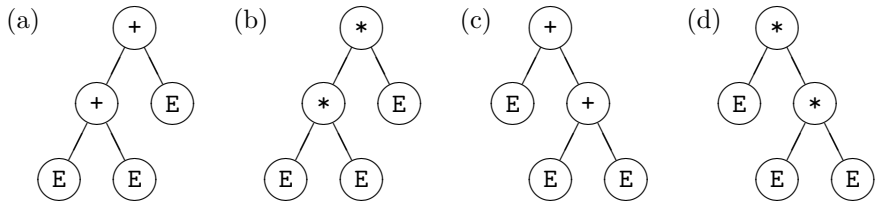

Figure 2: Computationally valid (a, b) and invalid (c, d) syntax trees for expressions $E + E + E$ and $E * E * E$, assuming the left-associativity of addition and multiplication

opposite order of evaluation. What we would like, therefore, is to be able to include syntax trees (a) and (b) only. As it turns out, this may be accomplished quite easily by assigning an appropriate associativity vector to each production and by introducing an additional rule when constructing precedence correct syntax trees from OCFGs:

Rule 2. Two adjacent nodes in an OCFG syntax tree may correspond to productions with equal precedence (and hence with the same left part) *only if* the appropriate bit in the top production's associativity vector is set.

To make our example work, we set the associativity vectors for both productions to 10, obtaining $E \to E + E : 10$ and $E \to E * E : 10$. The leftmost bit in both vectors is set, meaning — per Rule 2 — that the leftmost occurrence of E on the right side may be expanded using a production with the same precedence as the production to which it belongs. This is exactly what trees (a) and (b) in Fig. 2 depict. On the other hand, since the rightmost bit is cleared, the rightmost instance of E may *not* be expanded with a production of equal precedence, and we must resort to productions allowed by Rule 1 instead.

In the foregoing example, the associativity vector 10 roughly corresponded to a `%left` declaration in `yacc`. We could also have made either production right-associative (`%right`) or non-associative (`%nonassoc`) by setting its associativity vector to 01 or 00, respectively. Finally, if the order of evaluation did not matter — a distinct possibility given that addition and multiplication commute — we could have declared the productions *fully associative* by using the vector 11. Marking a production fully associative, an option not explicitly available in metalanguages such as `yacc`,[3] allows more sentential forms to be generated from the grammar. Conversely, it affords the parser more leeway in resolving conflicts; this is related to the idea of *semantically irrelevant ambiguity* as discussed in [Tho94].

Precedence as used in OCFGs differs from the annotations present in `yacc` in that it is applied to entire productions rather than to individual tokens. Thus, productions with different precedence can easily share a terminal symbol (e.g.,

[3] Unless otherwise marked, productions are treated by `yacc` as fully associative.

$E \rightarrow -E$, $E \rightarrow E - E$).[4] The associativity semantics allowed by OCFGs are more fine-grained, since they allow one to control each occurrence of the left part symbol in the right part, rather than specifying associativity of productions as a whole.

3 The Bertha Parser Generator

The current version of the Bertha tool is designed to handle the LALRP(1) subset [Las99] of ordered context-free grammars. These are analogous to the LALR(1) subset of CFGs in that states containing items differing only in their lookahead sets are merged [ASU86]. (Similarly, the LRP(1) subset is analogous to LR(1) in that such items are *not* merged.) But parsers constructed from OCFGs differ from their CFG counterparts in the very notion of parser states they employ. In the latter, states are merely sets of items; in the former, they are *partial orders* of items, induced by the precedences of the productions themselves. All of these issues are discussed in detail in [Las99], but are not pivotal to the understanding of Bertha from an end-user perspective.

Treating parser states as partial orders of items implies that a parser constructed from an ordered context-free grammar may contain states differing only in the ordering of the constituent items rather than in the items themselves. We shall demonstrate such a grammar in Sect. 4. At the same time, any existing context-free grammar may be viewed as a special case of an ordered context-free grammar in which all productions are completely unordered and fully associative. For these reasons, LALRP(1) represents a strictly more powerful formalism than LALR(1).

In its input syntax and semantics, Bertha draws heavily on the Java™ programming language [GJS96]. The consequences of this design choice are twofold. First, Bertha input specifications can be easily transformed into bottom-up parsers in the form of Java programs. Secondly, the specifications themselves can take advantage of many features of the Java language such as type-safety, encapsulation and inner classes. The last feature is especially useful as it allows us to easily implement nested grammars and symbol closures.

Figure 3 shows an example Bertha grammar for a simple calculator. After processing, this grammar will become a self-contained application (note the presence of the `main` entry point) that may be compiled and executed on any Java platform. All `symbol`s in the specification will be transformed into `class`es, and all `reduce(...)` productions into methods. The `start symbol` of the grammar will further be made a subclass of a special run-time class from which it will inherit the table-driven bottom-up parser engine.

The `prec(...)` and `assoc(...)` constructs are used to establish the required partial order among productions and to assign appropriate associativity vectors to each. Associativity vectors may be either specified explicitly as Boolean strings (e.g., `"10"`) or by using the keywords `left`, `right`, `full` or

[4] This can be achieved in `yacc` by employing the `%prec` directive in conjunction with an additional placeholder token; Sect. 4 contains a (rather verbose) example of this.

```
// 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[]) throws ParseError {
    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 3: A sample **Bertha** grammar specification

`none`. The `assoc(...)` modifier may be omitted from a production, which is semantically equivalent to specifying `assoc(full)`. As can be seen in Fig. 3, the `prec(...)` scopes may be nested in order to achieve the desired partial order. In our example, addition and subtraction share a precedence level and have a jointly lower precedence than do multiplication and division. (Note that it is possible for two productions with equal precedence to have different associativities.) Alternatively, a `symbol` may use no `prec(...)` declarations at all, which is equivalent to a single `prec(incomparable)` declaration encompassing all the productions for that symbol. If the entire grammar does not contain any `assoc(...)` or `prec(...)` constructs, it is an ordinary context-free grammar.

Note that Bertha relies on the Java inner class construct to support nested symbols. In Fig. 3, the `symbol num` is nested inside of the `start symbol E`. All symbols in a grammar must, in fact, be nested within their corresponding start symbols. This is not mere lexical nesting but rather a *closure* that binds each instance of the nested symbol to that of an enclosing symbol. Thus, the semantic action defined for `symbol num` can access field `val` of `start symbol E`. Symbols can be nested arbitrarily deeply, as long as they conform to Java scoping rules.

In addition to nested symbols, Bertha also supports *nested grammars*. The idea behind nested grammars is to allow the user to specify stream content that may occur *between* the tokens of an enclosing grammar. Figure 4 depicts a fragment of a hypothetical grammar for the Java language that delegates the handling of comments to an inner grammar. In turn, the `comments` grammar uses yet another grammar, called `whitespace`, to consume any non-printing characters from the input. When Bertha processes file `java_grammar.bertha`, it will produce a file `java_grammar.java` containing *three* separate sets of parse tables, one for each grammar in question.

The operational semantics for a parse employing grammars nested in this fashion is as follows. Whenever the parser for an enclosing grammar (for example, `java_grammar`) is *about* to read the next token from the input stream,[5] it first invokes a parse of its immediate inner grammar (in our case, `comments`) to extract and parse any sentences recognizable by the inner grammar. Similarly, before reading in a token, the `comments` grammar instructs the `whitespace` grammar to consume any sentences composed of non-printing characters. For this scheme to work properly, the programmer must ensure that inner grammars do not consume tokens actually "belonging to" outer grammars.

Inner grammars not only facilitate a separation of concerns between the handling of a core language and its comments, but also allow for a much more thorough handling of the latter. As Fig. 4 illustrates, delegating Java comment processing to a separate grammar enables us to easily extract and process embedded `javadoc` [GJS96] documentation. We could also handle nested comments found in languages such as Pascal and Oberon-2 [Wir96], since we have a pushdown automaton at our disposal. Lexical analyzers like `lex`, on the other hand, construct only finite-state automata (FSA). In order to recognize recur-

[5] This should not be confused with *shifting* tokens onto the parse stack, which occurs later.

```
// java_grammar.bertha
  :
start symbol java_grammar {
    :
  // nested grammar for comments
  start symbol comments {
      // yet another grammar, just for whitespace, which will extract
      // non-printing characters:
      //   -- within Java comments; and
      //   -- in between Java tokens (identifier, operators, etc.).
      start symbol whitespace {
        reduce((' '|'\t'|'\n')*) { ... }
      }
      reduce("//", #* / "\n", "\n") { ... }
      reduce("/*", #* / "*/", "*/") { ... }
      reduce("/**", javadoc, "*/") { ... }
      symbol javadoc {
        reduce(comment_lines, comment_entries) { ... }
        reduce(comment_entries) { ... }
        reduce() { ... }
      }
      symbol comment_lines {
        String comment;
        reduce(#* / ("*/" | "@") str) { comment = str; }
      }
      symbol comment_entries {
        reduce(comment_entries, comment_entry) { ... }
        reduce(comment_entry) { ... }
      }
      symbol comment_entry {
        reduce("@author", comment_lines) { ... }
        reduce("@since", comment_lines) { ... }
        reduce("@see", comment_lines) { ... }
          :
      }
      :
    }
  }
  // begin Java grammar
  reduce(package_information, class_definitions) { ... }
  :
}
```

Figure 4: Specifying the phrase structure of Java comments using **Bertha** nested grammars

sive structures of any kind, the programmer is required to explicitly maintain several FSA start (`%s`) states [LMB92].

All the declared `symbol`s in the Bertha metalanguage are nonterminals; to specify the lexical structure of tokens, the programmer must rely on anonymous regular expression literals. Figure 4 contains several examples of such literals. For the most part, the regular expression notation adopted by Bertha is similar to those used with `lex` or on UNIX platforms, and should be self-explanatory. The `#` symbol is similar to the `lex` period (`.`) but will match all characters, including `'\n'`. The `/` (forward slash) symbol is a lookahead operator with the same semantics as in `lex`: The patterns on both sides of `/` must match the input, but only the left pattern will actually be consumed.

4 Comparing Bertha and yacc

Figures 3 and 4 demonstrate some of the advantages of the Bertha metalanguage over those of more conventional parser generators. In this section, we shall present additional examples which will hopefully illustrate this further.

First, consider the subset of the Java language which deals with expressions. As do its C and C++ forebears, Java relies on a well-defined hierarchy of operator precedence along with associativity characteristics for each precedence level. The designers of the Java language provided a pure (i.e., unadorned with `yacc`-style annotations) LALR(1) grammar for it [GJS96], presumably to serve as a formal reference for the concrete syntax. While this specification is sufficient for reference purposes, using it for purposes of parsing can be quite awkward.

In Fig. 5, we have attempted to rewrite portions of the Java grammar dealing with expressions — with a few simplifications — using the precedence and associativity annotations provided by `yacc`. This definition is considerably shorter (and contains fewer nonterminals) than the original, making semantic instrumentation much easier. Unfortunately, the code in Fig. 5 is far from intuitive. Even though precedence clearly applies to productions, it must be specified with respect to tokens. In fact, several tokens such as `TYPECAST` must be "invented" just to assure a proper ordering; a complete Java parser written this way is sure to contain many more of them. In addition, a separate lexical analyzer must be provided to actually read the tokens from an input stream.

Compare this with Fig. 6, which depicts the same Java language fragment described using Bertha metalanguage syntax. Structurally, this definition is quite similar to the one from Fig. 5, and one might expect both of them to yield the same bottom-up parser. Recall, however, that precedence and associativity are used by `yacc` *only* to resolve shift/reduce conflicts, whereas in Bertha they form an integral part of the grammar formalism and are consulted at all times to prevent the construction of computationally incorrect parse trees. Consequently, the `yacc`-generated parser will accept derivations, such as those shown in Fig. 7, that are clearly not desirable from the viewpoint of computation order.

Another advantage to using Bertha lies in the notational convenience its metalanguage provides. One no longer needs an elaborate arrangement of tokens

```
// JavaExpr.y
%right ASSIGN_OP
%right ':' '?'  %left OR_OR  %left AND_AND   %left '|'
%left '^'       %left '&'    %left EQUAL_OP  %left UNEQUAL_OP
%left SHIFT_OP  %left INFIX_ADD              %left MUL_OP
%nonassoc PARENTH            %nonassoc TYPECAST
%right '~'      %right '!'   %right ADD_OP   %nonassoc INCR_OP
%left '.' '[' '('            %nonassoc NUM
%%
JavaExpr:
    NUM
  | JavaExpr '(' ')' | JavaExpr '[' ']' | JavaExpr '.' JavaExpr
  | JavaExpr INCR_OP
  | INCR_OP JavaExpr
  | ADD_OP JavaExpr
  | '!' JavaExpr
  | '~' JavaExpr
  | '(' JavaExpr ')' JavaExpr   %prec TYPECAST
  | '(' JavaExpr ')'   %prec PARENTH
  | JavaExpr MUL_OP JavaExpr
  | JavaExpr ADD_OP JavaExpr   %prec INFIX_ADD
  | JavaExpr SHIFT_OP JavaExpr
  | JavaExpr UNEQUAL_OP JavaExpr
  | JavaExpr EQUAL_OP JavaExpr
  | JavaExpr '&' JavaExpr
  | JavaExpr '^' JavaExpr
  | JavaExpr '|' JavaExpr
  | JavaExpr AND_AND JavaExpr
  | JavaExpr OR_OR JavaExpr
  | JavaExpr '?' JavaExpr ':' JavaExpr
  | JavaExpr ASSIGN_OP JavaExpr
  ;
```

Figure 5: A deterministic (but non-LALR(1)) `yacc` specification for Java expressions

```
// JavaExpr.bertha
start symbol JavaExpr {
  prec(decreasing) {
    reduce("#" i) { }
    prec(equal) {
      assoc(left) reduce(JavaExpr, "(", ")") { }
      assoc(left) reduce(JavaExpr, "[", "]") { }
      assoc(left) reduce(JavaExpr l, ".", JavaExpr r) { }
    }
    prec(equal) {
      assoc(none) reduce(JavaExpr, "++" | "--" op) { }
      assoc(none) reduce("++" | "--" op, JavaExpr) { }
    }
    reduce("+" | "-" op, JavaExpr) { }
    assoc("1") reduce("!", JavaExpr) { }
    assoc("1") reduce("~", JavaExpr) { }
    reduce("(", JavaExpr, ")", JavaExpr) { }
    reduce("(", JavaExpr, ")") { }  // this resolves the s/r conflict!
    assoc(left) reduce(JavaExpr, "*" | "/" | "%" op, JavaExpr) { }
    assoc(left) reduce(JavaExpr, "+" | "-" op, JavaExpr) { }
    assoc(left) reduce(JavaExpr, "<<" | ">>" | ">>>" op,
        JavaExpr) { }
    assoc(left) reduce(JavaExpr,
        "<" | "<=" | ">" | ">=" | "instanceof" op, JavaExpr) { }
    assoc(left) reduce(JavaExpr, "==" | "!=" op, JavaExpr) { }
    assoc(left) reduce(JavaExpr, "&", JavaExpr) { }
    assoc(left) reduce(JavaExpr, "^", JavaExpr) { }
    assoc(left) reduce(JavaExpr, "|", JavaExpr) { }
    assoc(left) reduce(JavaExpr, "&&", JavaExpr) { }
    assoc(left) reduce(JavaExpr, "||", JavaExpr) { }
    assoc("011") reduce(JavaExpr, "?", JavaExpr, ":", JavaExpr) { }
    assoc(right) reduce(JavaExpr, "=" | "*=" | "/=" | "%=" | "+=" |
        "-=" | "<<=" | ">>="| ">>>=" | "&=" | "^=" | "|=" op, JavaExpr) { }
  } // prec(decreasing)
}
```

Figure 6: Specifying Java expressions in Bertha

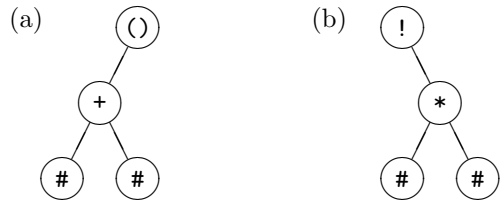

Figure 7: Examples of parse trees constructed with the `yacc` grammar in Fig. 5 that would be disallowed by the Bertha grammar in Fig. 6

```
// NotLALR1.bertha
start symbol NotLALR1 {
  prec(decreasing) {
    reduce("a", A, "d") { }
    reduce("a", B, "e") { }
  }
  prec(decreasing) {
    reduce("b", B, "d") { }
    reduce("b", A, "e") { }
  }
  symbol A {
    reduce("c") { }
  }
  symbol B {
    reduce("c") { }
  }
}
```

Figure 8: Disambiguating a non-LALR(1) grammar through use of precedence

to specify precedence, nor does one need to worry about the lexical analysis of tokens. The notation exploits features of the underlying Java language in order to establish an intuitive isomorphism between nonterminal symbols and their types. A Java `class` provides the type for each symbol, whereas the *name* of this class identifies the symbol itself within the OCFG framework. Parser generators such as `yacc` treat the typing of symbols as a separate (and entirely optional) concern.

Thus far, we have illustrated **Bertha**'s use of precedence in restricting the construction of computationally incorrect derivation trees from input strings. Conversely, it is equally noteworthy that there exist valid **Bertha** grammar specifications that *cannot* be analogously handled by `yacc`, regardless of the annotations provided. Recall that the states in a pushdown automaton constructed from an ordered context-free grammar are partial orders of items, which potentially allows for more parser states to exist.

Figure 8 depicts a grammar which is known not to be in LALR(1) [ASU86]. When compiled using `yacc`, it yields a parser with a reduce/reduce conflict. The reductions in question involve the productions $A \to c$ and $B \to c$, both of them sharing the same lookahead set: $\{d, e\}$. In order to eliminate this conflict, the grammar would need to be rewritten. When **Bertha** constructs the parser, however, the conflict will not occur. Rather that having a single state containing items with $A \to c$ and $B \to c$, the parser will contain two such states, differing only in the ordering of the items. With `yacc`, one could not specify an ordering among these two productions since they do not share a left part (and, at any rate, one would need *two* distinct orderings). With **Bertha**, this ordering is established transitively from the ordering of items in respective predecessor states; [Las99] explains this computation at length.

Note that the additional state obtained by establishing a partial order among the production could in most cases probably be obtained by enhancing an ordinary LALR(1) parser generator with a *state-splitting* algorithm, such as the one described in [Pag77, Spe88]. However, in the latter approach, the state

splitting is triggered only when an inadequate state is found and requires rather expensive backtracking through previously constructed parser states.

5 Conclusions and Future Work

The Bertha parser generator presented in this paper offers a twofold advantage over more traditional tools such as `yacc`. First, its input metalanguage is directly tied to the underlying Java language and is thus able to offer encapsulation, abstraction and type-safety to the user.[6] Second, adopting the semantics of ordered context-free grammars for the input metalanguage allows for a more intuitive and consistent treatment of production associativity and precedence that occur so frequently in language constructs. As was demonstrated above, LALRP(1) parsers also have strictly more disambiguation capability than LALR(1) parsers.

There are still areas in which the Bertha tool could be improved. For one, it may be beneficial to augment the current parser construction algorithm with a state-splitting operation [Pag77, Spe88] or the insertion of conflict-resolving *contextual predicates* [Tar82] into inadequate states. Of course, the present algorithm already introduces state splits in many places, as was illustrated in the last section. Nevertheless, there may exist remaining shift/reduce or reduce/reduce conflicts that cannot be disambiguated using production ordering alone, but that could be resolved using these other approaches.

Conversely, it is possible that the current LALRP(1) parser construction algorithm constructs additional states that are in fact redundant, i.e., recombining them would introduce no additional conflicts. Since it is advantageous to keep the number of states (and hence parse table size) to a minimum, adding a state-merging "clean-up" phase to our algorithm should be beneficial even if the number of candidate states is quite small.

Last, but certainly not least, is the issue of parser completeness. At present, Bertha shares the drawback with `yacc` of not being able to indicate to the programmer whether the constructed parser is complete, i.e., whether it always terminates and accepts all valid inputs. There exists a category of nondeterministic languages [ASU86] for which unambiguous LALR(1) grammars exist — for example, the language $L = \{a^n b^n \cup a^n b^{2n} \mid n \geq 1\}$. A bottom-up parser for such a language will be constructed without apparent problems, but will recognize only its subset [ST88]. Although it is undecidable whether an arbitrary context-free language is nondeterministic [HU79], an algorithm that deals with a subset of such languages has been proposed [Tho94] and its usefulness should be investigated.

[6] Of course, this reliance on Java could also be viewed as a limitation.

Acknowledgements

The author would like to express his gratitude to Peter Fröhlich, Joachim Feise, Christian Stork, Roy Fielding, Jules Winfield and Vincent Vega for their valuable comments and suggested improvements to this paper.

References

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

[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.

[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.

[Der69] Frank L. Deremer. *Practical Translators for* LR(k) *Languages*. PhD thesis, Dept. of Electrical Engineering, MIT, Cambridge, MA, 1969.

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

[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`.

[GnJ98] 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`.

[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.

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

[Las99] Ziemowit Laski. Ordered context-free grammars. Technical Report 99–18, Dept. of Information and Computer Science, University of California, Irvine, 26 April 1999. Available at `http://caesar.ics.uci.edu/laski/PrecedenceParsing.html`.

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

[MW91] Hanspeter Mössenböck and Niklaus Wirth. The programming language Oberon-2. *Structured Programming*, 12(4):179–195, 1991. Available at `http://www.ssw.uni-linz.ac.at/Research/Papers/Moe91a.html`.

[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.

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

[ST88] Eljas Soisalon-Soininen and Jorma Tarhio. Looping LR parsers. *Information Processing Letters*, 26(5):251–253, 11 January 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.

[Wir96] Niklaus Wirth. *Compiler Construction*. Addison-Wesley, 1996.

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