An Experimental Ambiguity Detection Tool

Sylvain Schmitz

An Experimental Ambiguity Detection Tool*

Sylvain Schmitz

LSV, ENS Cachan & CNRS, France
Sylvain.Schmitz@lsv.ens-cachan.fr

Abstract

Although programs convey an unambiguous meaning, the grammars used in practice to describe their syntax are often ambiguous, and completed with disambiguation rules. Whether these rules achieve to the removal of all the ambiguities while preserving the original intended language can be difficult to ensure. We present an experimental ambiguity detection tool for GNU Bison, and illustrate how it can assist a grammatical development for a subset of Standard ML.

Key words: grammar verification, disambiguation, GLR

1 Introduction

With the broad availability of parser generators that implement Generalized LR (GLR) (Tomita, 1986) or the Earley (1970) algorithm, it might seem that struggles with the dreaded report

```
grammar.y: conflicts: 223 shift/reduce, 35 reduce/reduce
```

are now over. General parsers of these families simulate the various nondeterministic choices in parallel with good performance, and return all the legitimate parses for the input (see Scott and Johnstone (2006) for a survey).

What our naive account overlooks is that all the legitimate parses according to the grammar might not always be correct in the intended language. With programming languages in particular, a program is expected to have a unique interpretation, and thus a single parse should be returned. Nevertheless, the grammar developed to describe the language is often ambiguous: ambiguous grammars are more concise and readable (Aho et al., 1975). The language definition should therefore include some *disambiguation* rules to decide which parse to choose.

In this paper, we present a tool for GNU Bison (Donnely and Stallman, 2006)[1] that pinpoints possible ambiguities in context-free grammars (CFGs). Grammar and parser developers can then use the ambiguities reported by the

*Expanded version of an article presented at the 7th Workshop on Language Descriptions, Tools and Applications (LDTA'07). Published in *Science of Computer Programming* 75(1–2):71–84, 2010. doi:10.1016/j.scico.2009.07.002.

[1]The modified Bison source is available from the author's webpage, at the address http://www.lsv.ens-cachan.fr/~schmitz/.

tool to write disambiguation rules where they are needed. Since the problem of finding all the ambiguities in a CFG is undecidable (Cantor, 1962; Chomsky and Schützenberger, 1963; Floyd, 1962), our tool implements a conservative algorithm (Schmitz, 2007): it guarantees that no ambiguity will be overlooked, but it might return false positives as well. We attempt to motivate the use of such a tool for grammatical engineering (Klint et al., 2005).

- We first describe a well-known difficult subset of the syntax of Standard ML (Milner et al., 1997) (Section 2.1) that combines a genuine ambiguity with a LR conflict requiring unbounded lookahead (Section 2.2). A generalized parser parses correctly the corresponding Standard ML programs, but might return more than one parse (Section 2.3).
- We detail how our tool identifies the ambiguity as such and discards the conflict (Section 3) before succinctly presenting the algorithm we employ.
- We put our technique to the test and compare it experimentally with other conservative ambiguity methods (Section 4).
- Finally, we examine the shortcomings of the tool and provide some leads for its improvement (Section 5).

2 A Difficult Syntactic Issue

In this section, we consider a subset of the grammar of Standard ML, and use it to illustrate some of the difficulties encountered with classical LALR(1) parser generators in the tradition of YACC (Johnson, 1975). Unlike the grammars sometimes provided in other programming language references, the grammar defined by Milner et al. (1997, Appendix B) is not put in LALR(1) form. In fact, it clearly values simplicity over ease of implementation, and includes highly ambiguous rules like $\langle dec\rangle \rightarrow \langle dec\rangle\ \langle dec\rangle$.

2.1 Case Expressions in Standard ML

Kahrs (1993) describes a situation in the Standard ML syntax where an unbounded lookahead is needed by a deterministic parser in order to correctly parse certain strings. The issue arises with alternatives in function value binding and **case** expressions. A small set of grammar rules from the language specification that illustrates the issue is given in Figure 1.

The rules describe Standard ML declarations $\langle dec\rangle$ for functions, where each function name *vid* is bound, for a sequence $\langle atpats\rangle$ of atomic patterns, to an expression $\langle expr\rangle$ using the rule $\langle sfvalbind\rangle \rightarrow vid\ \langle atpats\rangle = \langle exp\rangle$. Different function value bindings can be separated by alternation symbols "|". Standard ML **case** expressions associate an expression $\langle exp\rangle$ with a $\langle match\rangle$, which is a sequence of matching rules $\langle mrule\rangle$ of form $\langle pat\rangle$ => $\langle exp\rangle$, separated by alternation symbols "|".

Example 1 Using mostly these rules, the `filter` function of the SML/NJ Library could be written (Lee, 1997) as:

$$
\begin{array}{rcl}
\langle \textit{dec} \rangle & \to & \textbf{fun}\ \langle \textit{fvalbind} \rangle \\
\langle \textit{fvalbind} \rangle & \to & \langle \textit{fvalbind} \rangle\ '|'\ \langle \textit{sfvalbind} \rangle \\
 & | & \langle \textit{sfvalbind} \rangle \\
\langle \textit{sfvalbind} \rangle & \to & \textit{vid}\ \langle \textit{atpats} \rangle = \langle \textit{exp} \rangle \\
\langle \textit{atpats} \rangle & \to & \langle \textit{atpats} \rangle\ \langle \textit{atpat} \rangle \\
 & | & \langle \textit{atpat} \rangle \\
\langle \textit{exp} \rangle & \to & \textbf{case}\ \langle \textit{exp} \rangle\ \textbf{of}\ \langle \textit{match} \rangle \\
 & | & \textit{vid} \\
\langle \textit{match} \rangle & \to & \langle \textit{match} \rangle\ '|'\ \langle \textit{mrule} \rangle \\
 & | & \langle \textit{mrule} \rangle \\
\langle \textit{mrule} \rangle & \to & \langle \textit{pat} \rangle \Rightarrow \langle \textit{exp} \rangle \\
\langle \textit{pat} \rangle & \to & \textit{vid}\ \langle \textit{atpat} \rangle \\
\langle \textit{atpat} \rangle & \to & \textit{vid}
\end{array}
$$

Figure 1: Syntax of function value binding and **case** expressions in Standard ML. We translated the rules from their original extended form into BNF. We write ⟨*nonterminals*⟩ between angle brackets and *terminals* as such, except for the terminal alternation symbol ′|′, quoted in order to avoid confusion with the choice meta character |.

```
datatype 'a option = NONE | SOME of 'a
fun filter pred l =
  let
    fun filterP (x::r, l) =
        case (pred x)
          of SOME y => filterP(r, y::l)
           | NONE => filterP(r, l)
     | filterP ([], l) = rev l
  in
    filterP (l, [])
  end
```

The Standard ML compilers consistently reject this correct input, often pinpointing the error at the equal sign in "| filterP ([], l) = rev l". Let us investigate why they behave this way.

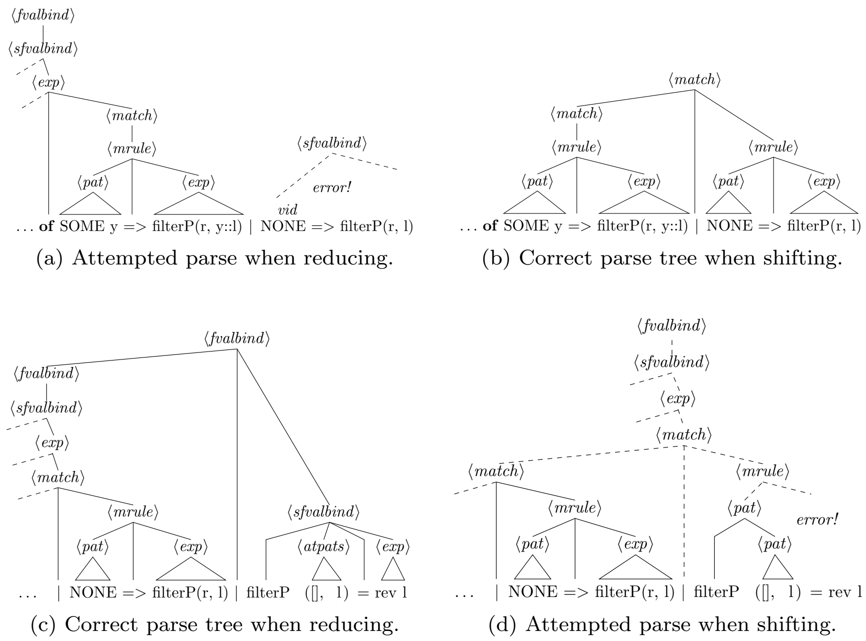
2.2 The Conflict

We implemented our set of grammar rules in GNU Bison (Donnely and Stallman, 2006), and the result of a run in LALR(1) mode is a single shift/reduce conflict, a nondeterministic choice between two parsing actions:

```
state 20
    6 exp: "case" exp "of" match .
    8 match: match . '|' mrule

    '|'  shift, and go to state 24
    '|'       [reduce using rule 6 (exp)]
```

The conflict has to be solved in two different places with the program of



(a) Attempted parse when reducing. (b) Correct parse tree when shifting.

(c) Correct parse tree when reducing. (d) Attempted parse when shifting.

Figure 2: Partial parse trees corresponding to the two occurrences of the conflict in Example 1.

Example 1, corresponding to the two different occurrences of the alternation symbol "|".

If we choose arbitrarily one of the actions—shift or reduce—over the other, we obtain the parses drawn in Figure 2. The shift action is chosen by default by Bison, and ends on a parse error when seeing the equal sign where a double arrow was expected, exactly where the Standard ML compilers report an error (Figure 2d).

We could make the correct decision if we had more information at our disposal. The "=" sign in the lookahead string "| filterP ([], l) = rev l" indicates that the alternative is at the topmost function value binding ⟨*fvalbind*⟩ level, and not at the "**case**" level, or it would be a "=>" sign. But the sign can be arbitrarily far away in the lookahead string: an atomic pattern ⟨*atpat*⟩ can derive a sequence of tokens of unbounded length. The conflict requires an unbounded lookahead.

Example 2 The issue is made further complicated by the presence of a dangling ambiguity:

case a **of** b => **case** b **of** c => c | d => d

In this expression, should the dangling "d => d" matching rule be attached to "**case** b" or to "**case** a"? The Standard ML definition indicates that the matching rule should be attached to "**case** b". In this case, the shift should be chosen rather than the reduction, which explains the choice made by developers of the various Standard ML parsers.

This issue in the syntax of Standard ML is one of its few major defects according to a survey by Rossberg (2006):

> [Parsing] this would either require horrendous grammar transformations, backtracking, or some nasty and expensive lexer hack.

Fortunately, the detailed analysis of the conflict we conducted, as well as the ugly or expensive solutions mentioned by Rossberg, are not necessary with a general parser.[2]

2.3 General Parsing

A general parser returns all the possible parses for the provided input, and as such discards the incorrect parses of Figures 2a and 2d and only returns the correct ones of Figures 2b and 2c. In particular, a generalized LALR(1) parser explores the two possibilities of the conflict, until it reaches the "=>" or "=" sign, at which point the incorrect partial parses of Figures 2a and 2d fail.

Our tool tackles an issue that appeared with the recent popularity of general algorithms for programming languages parsers. The user does not know *a priori* whether the conflict reported by Bison in the LALR(1) automaton is caused by an ambiguity or by an insufficient lookahead length. A casual investigation of its source might only reveal the unbounded lookahead aspect of the conflict as with Example 1, and overlook the ambiguity triggered by embedded case expressions like the one of Example 2. The result might be a collection of parse trees—a *parse forest*—where a single parse tree was expected, hampering the reliability of the computations that follow the parsing phase.

Two notions pertain to the current use of parse forests in parsing tools.

- The *sharing* of common subtrees bounds the forest space complexity by a polynomial function of the input length (Billot and Lang, 1989). Figure 3 shows a shared forest for our ambiguity, with a topmost $\langle match \rangle$ node that merges the two alternative interpretations of the input of Example 2.

- Klint and Visser (1994) developed the general notion of *disambiguation filters* that reject some of the trees of the parse forest, with the hope of ending the selection process with a single tree. Such a mechanism is implemented in one form or in another in many GLR tools, including SDF (van den Brand et al., 2002), Elkhound (McPeak and Necula, 2004), and Bison (Donnely and Stallman, 2006).

2.3.1 Merge Functions

Unexpected ambiguities are acute with GLR parsers that compute semantic *attributes* as they reduce partial trees. The GLR implementations of GNU Bison (Donnely and Stallman, 2006) and of Elkhound (McPeak and Necula, 2004) are in this situation. Attribute values are synthesized for each parse tree node, and in a situation like the one depicted in Figure 3, the values obtained

[2] Some deterministic parsing algorithms—LR-Regular (Čulik and Cohen, 1973; Baker, 1981; Boullier, 1984), noncanonical (Szymanski and Williams, 1976; Tai, 1979; Schmitz, 2006), or LL-Regular (Poplawski, 1979; Parr, 2007)—albeit perhaps less known, are able to exploit unbounded lookahead lengths. Our ambiguity detection algorithm employs similar principles.

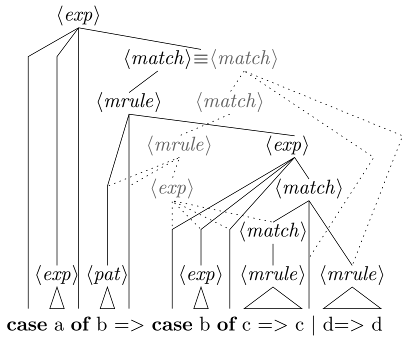


Figure 3: The shared parse forest for the input of Example 2.

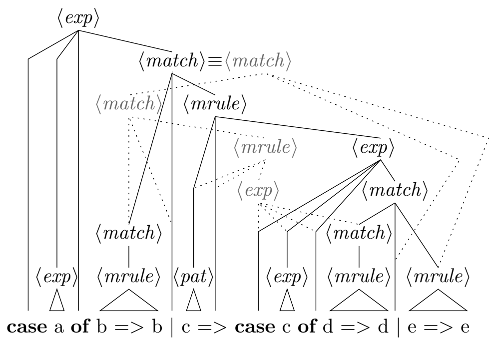


Figure 4: The shared parse forest for the input of Example 3.

for the two alternatives of a shared node have to be merged into a single value for the shared node as a whole. The user of these tools should thus provide a *merge* function that returns the value of the shared node from the attributes of its competing alternatives.

Failure to provide a merge function where it is needed forces the parser to choose arbitrarily between the possibilities, which is highly unsafe. Another line of action is to abort parsing with a message exhibiting the ambiguity; this can be set with an option in Elkhound, and it is the behavior of Bison.

2.3.2 A Detailed Knowledge of Ambiguities

Example 3 Let us suppose that the user has encountered the ambiguity of Example 2, and is using a disambiguation filter (in the form of a merge function in Bison or Elkhound) that discards the dotted alternative of Figure 3, leaving only the correct parse according to the Standard ML definition. A simple way to achieve this is to check whether we are reducing using rule $\langle match\rangle \rightarrow \langle match\rangle' | '\langle mrule\rangle$ or with rule $\langle match\rangle \rightarrow \langle mrule\rangle$. Filters of this variety are quite common, and are given a specific `dprec` directive in Bison, also corresponding to the `prefer` and `avoid` filters in SDF2 (van den Brand et al., 2002).

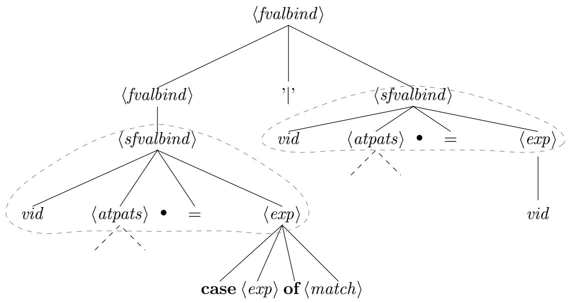


Figure 5: Two equivalent positions under the LR(0) item approximation, corresponding to the single item $[\langle sfvalbind \rangle \rightarrow vid \ \langle atpats \rangle \bullet = \langle exp \rangle]$.

The above solution is unfortunately unable to deal with yet another form of ambiguity with $\langle match \rangle$, namely the ambiguity encountered with the input:

case a **of** b => b | c => **case** c **of** d => d | e => e

Indeed, with this input, the two shared $\langle match \rangle$ nodes are obtained through reductions using the same rule $\langle match \rangle \rightarrow \langle match \rangle' | ' \langle mrule \rangle$, as can be seen in Figure 4. Had we trusted our filter to handle all the ambiguities, we would be running our parser under a sword of Damocles.

This last example shows that a precise knowledge of the ambiguous cases is needed for the development of a reliable GLR parser. While the problem of detecting ambiguities is undecidable, conservative answers could point developers in the right direction.

3 Detecting Ambiguities

Our tool is implemented in C as a new option in GNU Bison that triggers an ambiguity detection computation instead of the parser generation. The output of this verification on our subset of the Standard ML grammar reports two pairs of positions in the grammar that exhibit potential ambiguities:

```
2 potential ambiguities with LR(0) precision detected:
    (match -> mrule . , match -> match . '|' mrule )
    (match -> match . '|' mrule , match -> match '|' mrule . )
```

From this ambiguity report, two things can be noted: that user-friendliness is not a strong point of the tool in its current form, and that the two detected ambiguities correspond to the two ambiguities of Examples 2 and 3. Furthermore, the reported ambiguities do not mention anything visibly related to the difficult conflict of Example 1.

3.1 Overview

Our ambiguity checking algorithm attempts to find ambiguities as two different parse trees describing the same sentence. Of course, there is in general an infinite number of parse trees with an infinite number of derived sentences, and we make therefore some approximations when visiting the trees.

We consider here approximations based on LR(0) items: a dot in a grammar production $A \to \alpha \bullet \beta$ can also be seen as a position in an elementary tree—a tree of height one—with root A and leaves labeled by $\alpha\beta$. When moving from item to item, we are also moving inside all the syntax trees that contain the corresponding elementary trees. The LR(0) item approximation is such that positions represented by the same item are considered as identical regardless of their actual context; Figure 5 presents two such equivalent positions in a derivation tree. We call this equivalence relation item_0.

In order to find potential ambiguities modulo our approximation, we further need to walk through the derivation trees. With LR(0) items, this means that we can move inside a dotted production without any loss of precision, but that upwards moves are performed regardless of any context. These eligible single moves from item to item are in fact the transitions in a *nondeterministic LR(0) automaton* (thereafter called LR(0) NFA). All the moves from item to item that we describe in the following can be checked on the trees of Figures 2 and 3.

Since we want to find two different trees, we work with pairs of concurrent items, starting from a pair $(S \to \bullet \langle dec \rangle\ \$,\ \ S \to \bullet \langle dec \rangle\ \$)$ at the beginning of all trees, and ending on a pair $(S \to \langle dec \rangle\ \$ \bullet,\ \ S \to \langle dec \rangle\ \$ \bullet)$. Between these, we pair items that could be reached upon reading a common prefix of a sentential form, hence following trees that derive the same sentence modulo our approximations.

The notion of equivalence of positions in derivation trees is the basis for a framework for context-free grammar approximations, which generalizes more complex constructions, like the item_Π equivalence of LR-Regular parsers (Čulik and Cohen, 1973; Heilbrunner, 1983). The LR(0) NFA is a special case of a more general *position automaton* that abstracts left-to-right walks inside the grammar trees. Our algorithm in its full generality guarantees that all ambiguities are caught for any such position automaton (Schmitz, 2007).

3.2 Example Run

We present here our algorithm with LR(0) items on the relevant portion of our grammar. Let us start with the pair of items reported as being in conflict by Bison; just like Bison, our algorithm has found out that the two positions might be reached by reading a common prefix from the beginning of the input:

$$(\langle match \rangle \to \langle match \rangle \bullet\ '|'\ \langle mrule \rangle,\ \ \langle exp \rangle \to \mathbf{case}\ \langle exp \rangle\ \mathbf{of}\ \langle match \rangle \bullet) \tag{A}$$

Unlike Bison, when confronted with a conflict, the algorithm attempts to see whether we can keep reading the same sentence until we reach the end of the input. Since we are at the extreme right of the elementary tree for rule $\langle exp \rangle \to$ **case** $\langle exp \rangle$ **of** $\langle match \rangle$, we are also to the immediate right of the nonterminal $\langle exp \rangle$ in some rule right part. Our algorithm explores all the possibilities, thus yielding the three pairs:

$$(\langle match \rangle \to \langle match \rangle \bullet\ '|'\ \langle mrule \rangle,\ \ \langle mrule \rangle \to \langle pat \rangle \texttt{=>} \langle exp \rangle \bullet) \tag{A.1}$$

$$(\langle match \rangle \to \langle match \rangle \bullet\ '|'\ \langle mrule \rangle,\ \ \langle exp \rangle \to \mathbf{case}\ \langle exp \rangle \bullet \mathbf{of}\ \langle match \rangle) \tag{A.2}$$

$$(\langle match \rangle \to \langle match \rangle \bullet\ '|'\ \langle mrule \rangle,\ \ \langle sfvalbind \rangle \to vid\ \langle atpats \rangle = \langle exp \rangle \bullet) \tag{A.3}$$

Applying the same idea to the conflicting pair (A.1), we should explore all the items with the dot to the right of $\langle mrule \rangle$.

$$(\langle match\rangle \to \langle match\rangle\,\bullet\ '|'\ \langle mrule\rangle,\quad \langle match\rangle \to \langle mrule\rangle\,\bullet) \tag{A.1.1}$$

$$(\langle match\rangle \to \langle match\rangle\,\bullet\ '|'\ \langle mrule\rangle,\quad \langle match\rangle \to \langle match\rangle\ '|'\ \langle mrule\rangle\,\bullet) \tag{A.1.2}$$

At this point, we find $[\langle match\rangle \to \langle match\rangle\,\bullet\ '|'\ \langle mrule\rangle]$, our competing item, among the items with the dot to the right of $\langle match\rangle$: from our approximations, the strings we can expect to the right of the items in the pairs (A.1.1) and (A.1.2) are the same, and we report the pairs as potential ambiguities.

Our ambiguity detection is not over yet: from (A.3), we can also reach (showing only the relevant possibilities):

$$(\langle match\rangle \to \langle match\rangle\,\bullet\ '|'\ \langle mrule\rangle,\quad \langle fvalbind\rangle \to \langle sfvalbind\rangle\,\bullet) \tag{A.3.1}$$

$$(\langle match\rangle \to \langle match\rangle\,\bullet\ '|'\ \langle mrule\rangle,\quad \langle fvalbind\rangle \to \langle fvalbind\rangle\,\bullet\ '|'\ \langle sfvalbind\rangle) \tag{A.3.1.1}$$

In this last pair, the dot is to the left of the same symbol, meaning that the following item pair might also be reached by reading the same string from the beginning of the input:

$$(\langle match\rangle \to \langle match\rangle\ '|'\ \bullet\langle mrule\rangle,\quad \langle fvalbind\rangle \to \langle fvalbind\rangle\ '|'\ \bullet\langle sfvalbind\rangle) \tag{B}$$

Because the dot is to the immediate left of a nonterminal symbol, it is also at the beginning of all the right parts of the productions of this symbol, yielding successively:

$$(\langle mrule\rangle \to \bullet\langle pat\rangle\texttt{=>}\langle exp\rangle,\quad \langle fvalbind\rangle \to \langle fvalbind\rangle\ '|'\ \bullet\langle sfvalbind\rangle) \tag{B.1}$$

$$(\langle mrule\rangle \to \bullet\langle pat\rangle\texttt{=>}\langle exp\rangle,\quad \langle sfvalbind\rangle \to \bullet\, vid\ \langle atpats\rangle = \langle exp\rangle) \tag{B.1.1}$$

$$(\langle pat\rangle \to \bullet\, vid\ \langle atpat\rangle,\quad \langle sfvalbind\rangle \to \bullet\, vid\ \langle atpats\rangle = \langle exp\rangle) \tag{B.1.1.1}$$

$$(\langle pat\rangle \to vid\,\bullet\langle atpat\rangle,\quad \langle sfvalbind\rangle \to vid\,\bullet\langle atpats\rangle = \langle exp\rangle) \tag{C}$$

$$(\langle pat\rangle \to vid\,\bullet\langle atpat\rangle,\quad \langle atpats\rangle \to \bullet\langle atpat\rangle) \tag{C.1}$$

$$(\langle pat\rangle \to vid\langle atpat\rangle\,\bullet,\quad \langle atpats\rangle \to \langle atpat\rangle\,\bullet) \tag{D}$$

$$(\langle mrule\rangle \to \langle pat\rangle\,\bullet\texttt{=>}\langle exp\rangle,\quad \langle atpats\rangle \to \langle atpat\rangle\,\bullet) \tag{D.1}$$

$$(\langle mrule\rangle \to \langle pat\rangle\,\bullet\texttt{=>}\langle exp\rangle,\quad \langle sfvalbind\rangle \to vid\ \langle atpats\rangle\,\bullet\ = \langle exp\rangle) \tag{D.1.1}$$

Our exploration stops with this last item pair: its concurrent items expect different terminal symbols, and thus cannot reach the end of the input upon reading the same string. The algorithm has successfully found how to discriminate between the two possibilities in conflict in Example 1.

3.3 Presentation of the Algorithm

The example run presented above relates pairs of items. We call this relation the mutual accessibility relation ma, and define it as the union of several primitive relations:

mas for terminal and nonterminal shifts, holding for instance between pairs (A.3.1.1) and (B), but also between (C.1) and (D),

mae for downwards closures, holding for instance between pairs (B) and (B.1),

mac for upwards closures in case of a conflict, i.e. when one of the items in the pair has its dot to the extreme right of the rule right part and the concurrent item is different from it, holding for instance between pairs (A.1) and (A.1.1). Formally, our notion of a conflict coincides with that of Aho and Ullman (1972, Theorem 5.9).

The algorithm thus constructs the image of the initial pair ($S' \to \bullet S\$$, $S' \to \bullet S\$$) by the reflexive transitive closure ma^* of the ma relation. If at some point we reach a pair holding two copies of the same item from a pair with different items, we report an ambiguity.[3] The algorithm is reminiscent of *noncanonical* parsing techniques (Szymanski and Williams, 1976), and we call it the *noncanonical unambiguity* (NU) test.

The size of the ma relation is bounded by the square of the size of the position automaton, here the LR(0) NFA. Let $|\mathcal{G}|$ denote the size of the context-free grammar $\mathcal{G}$, i.e. the sum of the length of all the rules right parts, and $|P|$ denote the number of rules; then, in the LR(0) case, the algorithm time and space complexity are bounded by $\mathcal{O}((|\mathcal{G}|\ |P|)^2)$.

3.4 Implementation Details

The experimental tool currently implements the algorithm with LR(0) items, SLR(1) items—meaning that simple lookahead sets are considered for the conflict relation mac—and LR(1) items. Although the space required by LR(1) item pairs is really large, we need this level of precision in order to guarantee an improvement over the LALR(1) construction. The implementation is changed in a few details.

3.4.1 NFA Size Optimization

We construct a nondeterministic automaton (Hunt III et al., 1974; Grune and Jacobs, 1990) whose states are either dotted rule items of form $A \to \alpha \bullet \beta$, or some nonterminal items of form $\bullet A$ or $A \bullet$. For instance, a nonterminal item would be used when computing the mutual accessibility of (A.1) and before reaching (A.1.1):

$$(\langle match \rangle \to \langle match \rangle \bullet \ '|' \ \langle mrule \rangle, \quad \langle mrule \rangle \bullet).$$

The size of the NFA then becomes bounded by $\mathcal{O}(|\mathcal{G}|)$ in the LR(0) and SLR(1) case, and $\mathcal{O}(|\mathcal{G}||T|^2)$—where $|T|$ is the number of terminal symbols—in the LR(1) case, and the complexity of the algorithm is thus bounded by the square of these numbers.

[3] Since this occurs as soon as we find a mac relation that reaches the same item twice, the mar relation and the boolean flag described in the general algorithm (Schmitz, 2007) are not needed.

3.4.2 Static Disambiguation

We consider the associativity and static precedence directives (Aho et al., 1975) of Bison in the conflict relation mac, and thus we do not report statically resolved ambiguities. Dynamic merge functions are a different matter, discussed in Section 5.3.

3.4.3 Ordering Conflicts

We order our items pairs to avoid redundancy in reduce/reduce conflicts. In such a conflict, we can choose to follow one reduction or the other, and we might find a point of ambiguity sooner or later depending on this choice. Let us consider for instance the grammar with rules

$$S \to aA,\ A \to aB \mid aa,\ B \to a\ .$$

The conflicting positions

$$(A \to aa\bullet,\quad B \to a\bullet) \tag{E}$$

can reach through mac^*

$$(A \to aa\bullet,\quad A \to aB\bullet) \tag{E.1}$$

$$(S \to aA\bullet,\quad B \to a\bullet) \tag{E.2}$$

$$(S \to aA\bullet,\quad A \to aB\bullet) \tag{E.2.1}$$

$$(S \to aA\bullet,\quad S \to aA\bullet) \tag{E.2.1.1}$$

where the pairs (E.1) and (E.2.1.1) denote the same potential ambiguity.

The same kind of issue was met by McPeak and Necula with Elkhound (McPeak and Necula, 2004), where a strict bottom-up order was enforced using an ordering on the nonterminals and the portion of the input string covered by each reduction.

We solve our issue in a similar fashion, the difference being that we do not have a finite input string at our disposal, and thus we adopt a more conservative ordering. We say that A and B are in a *right corner* relation, noted $A \mathrel{\underline{\setminus}} B$, if there is a rule $A \to \alpha B$. Our order is then the transitive reflexive closure $\mathrel{\underline{\setminus}}^*$ of the right corner relation. In a reduce/reduce conflict between reductions to A and B, we follow the reduction of A if $A \not\mathrel{\underline{\setminus}}^* B$ or if both $A \mathrel{\underline{\setminus}}^* B$ and $B \mathrel{\underline{\setminus}}^* A$. In our small example, this disallows the exploration of the pair (E.2) and thus of the remaining pairs (E.2.1) and (E.2.1.1).

4 Experimental Comparisons

The choice of a conservative ambiguity detection algorithm is currently rather limited. Nevertheless, several parsing techniques define proper subsets of the unambiguous grammars, and as such can be employed as unambiguity tests. The most common of all is the LALR(1) construction, but, as argued earlier, the presence of a conflict is not very informative as far as ambiguity is concerned. We present in this section several comparisons between our algorithm and its competitors.

4.1 Other Conservative Algorithms

4.1.1 LR(k) Construction

The class of LR(k) grammars uses a fixed amount k of lookahead to dispel conflicts. Although it is widely considered that even a setting of $k = 1$ leads to impractical parser sizes, there exist compression techniques, and a few implementations are available (e.g. MSTA (Makarov, 1999)).

The grammar family $\mathcal{G}_3^n$ demonstrates the complexity gains with our algorithm as compared to LR(k) parsing:

$$S \to A \mid B_n,\ A \to Aaa \mid a,\ B_1 \to aa,\ B_2 \to B_1B_1,\ \ldots,\ B_n \to B_{n-1}B_{n-1} \qquad (\mathcal{G}_3^n)$$

While a LR(2^n) test is needed in order to tell that $\mathcal{G}_3^n$ is unambiguous, the grammar is found unambiguous with our algorithm using LR(0) items.

Following the results on LR(k) testing (Hunt III et al., 1974), we implemented a canonical LR test in our tool using the same item pairing technique as for the NU test. More precisely, we compute the image of the initial pair of items through $(\mathsf{mas} \cup \mathsf{mae})^*$ and report a LR conflict as soon as we find an item pair that could follow a conflict relation mac.

4.1.2 LR-Regular Construction

Beyond LR(k) parsing, LR-Regular parsing (Čulik and Cohen, 1973) employs a regular approximation of the right context of conflicts in an attempt to find the correct parsing action. In practice it explores a regular cover of the right context of LR conflicts with a finite automaton (Baker, 1981; Boullier, 1984).

Grammar $\mathcal{G}_5$ is a non-LRR grammar with rules

$$S \to AC \mid BCb,\ A \to a,\ B \to a,\ C \to cCb \mid cb. \qquad (\mathcal{G}_5)$$

It is found to be unambiguous by our algorithm using LR(0) items.

Still with our item pairing approach, we implemented a LRR test (Heilbrunner, 1983), where item pairs after a conflict—i.e. after a mac relation—have to follow the same terminal symbols (and not any symbol in V as with mas), and can move downwards and upwards freely. A potential ambiguity is reported whenever a pair containing twice the same item is reached at some point during the exploration of the right context of conflicts, as with the NU test.

4.1.3 Horizontal and Vertical Ambiguity

A different approach, unrelated to any parsing method, was proposed by Brabrand et al. (2010) with their horizontal and vertical unambiguity test (HVRU). Horizontal ambiguity appears with overlapping concatenated languages, and vertical ambiguity with non-disjoint unions; their method thus follows exactly how the context-free grammar was formed. Their intended application is to test grammars that describe RNA secondary structures (Reeder et al., 2005).

Grammars $\mathcal{G}_6$ and $\mathcal{G}_7$ show that our method is not comparable with the horizontal and vertical ambiguity detection method of Brabrand et al.. Grammar $\mathcal{G}_6$ is a palindrome grammar with rules

$$S \to aSa \mid bSb \mid a \mid b \mid \varepsilon \qquad (\mathcal{G}_6)$$

Table 1: Reported potential ambiguities in the comparison grammars.

Grammar	actual class	LALR(1)	HVRU	NU(item_0)
$\mathcal{G}_3^n$	LR(2^n)	1	0	0
$\mathcal{G}_5$	non-LRR	1	1	0
$\mathcal{G}_6$	non-LRR	6	0	9
$\mathcal{G}_7$	LR(0)	0	1	0

Table 2: Reported potential ambiguities in the RNA grammars discussed by Reeder et al. (2005).

Grammar	actual class	LALR(1)	HVRU	NU(item_1)
RNA_1	ambiguous	30	6	14
RNA_2	ambiguous	33	7	13
RNA_3	non-LRR	4	0	2
RNA_4	SLR(1)	0	0	0
RNA_5	SLR(1)	0	0	0
RNA_6	LALR(1)	0	0	0
RNA_7	non-LRR	5	0	3
RNA_8	LALR(1)	0	0	0

that our method finds erroneously ambiguous. Conversely, grammar $\mathcal{G}_7$ with rules

$$S \to AA,\ A \to aAa \mid b \qquad (\mathcal{G}_7)$$

is a LR(0) grammar, and the test of Brabrand et al. finds it horizontally ambiguous and not vertically ambiguous.

Table 1 compiles the results obtained on these grammars. The "LALR(1)" column provides the total number of conflicts (shift/reduce as well as reduce/reduce) reported by Bison, the "HVRU" column the number of potential ambiguities (horizontal or vertical) reported by the HVRU algorithm with unfolding, and the "NU(item_0)" column the number of potential ambiguities reported by our algorithm with LR(0) items. For completeness, we also present the results of our tool on the RNA grammars of Reeder et al. (2005) in Table 2.

4.1.4 Precision Settings

Several conservative ambiguity detection methods are thus possible: LR(k) and LR-Regular testing, horizontal and vertical unambiguity testing, and NU testing. Each of these methods can employ different scales of precision:

- our implementation of the LR, LR-Regular and NU methods can employ LR(0), SLR(1) or LR(1) items and notions of conflicts;
- GNU Bison and MSTA further provide respectively a LALR(1) precision and a LR(k) precision with an arbitrary fixed k for the LR method;

- the results published by Brabrand et al. with horizontal and vertical unambiguity also take advantage of the possibility to *unfold* the grammar in order to improve the precision of their tests. The approximation they build without unfolding follows the technique of Mohri and Nederhof (2001), and is slightly better than the one provided by LR(0) items, because they identify the strongly regular portions of the grammar and avoid some unnecessary approximations. The results we present in the following for HVRU ambiguity detection do not take unfolding into account, but we dedicate a few more words on the matter in the addendum at the end this section.

4.2 Experiments on Grammars for Programming Languages

We ran our implementations of the LR, LRR and NU methods on seven different ambiguous grammars for programming languages:

Pascal an ISO-7185 Pascal grammar retrieved from the `comp.compilers` FTP at `ftp://ftp.iecc.com/pub/file/`, it is LALR(1) except for a dangling else ambiguity,

Mini C a simplified C grammar written by Jacques Farré for a compilers course, it is LALR(1) except for a dangling else ambiguity,

ANSI C (Kernighan and Ritchie, 1988, Appendix A.13), also retrieved from the `comp.compilers` FTP. The grammar is LALR(1), except for a dangling else ambiguity. The **ANSI C'** grammar is the same grammar modified by setting typedef names to be a nonterminal, with a single production $\langle \textit{typedef-name} \rangle \rightarrow \textit{identifier}$. The modification reflects the fact that GLR parsers should not rely on side-effects like the *lexer feedback hack* for disambiguation (see McPeak and Necula, 2004, Section 5.1).

Standard ML, extracted from the language definition (Milner et al., 1997, Appendix B). As mentioned in Section 2, this is a highly ambiguous grammar, and no effort whatsoever was made to ease its implementation with a parser generator.

Elsa C++, developed with the Elkhound GLR parser generator (McPeak and Necula, 2004), and a smaller version without class declarations nor function bodies. Although this is a grammar written for a GLR parser generator, it allows deterministic parsing whenever possible in an attempt to improve performance.

In order to provide a better ground for comparisons between LR, LRR and NU testing, we implemented an option that computes the number of initial LR(0) item pairs in conflict—for instance pair (A)—that can reach a point of ambiguity—for instance pair (A.1.1)—through the ma relation. Table 3 presents the number of such initial conflicting pairs with our tests when employing LR(0) items, SLR(1) items, and LR(1) items. We completed our implementation by counting conflicting LR(0) item pairs for the LALR(1) conflicts in the parsing tables generated by Bison, which are shown in the LALR(1) column of Table 3.

This measure of the initial LR(0) conflicts is far from perfect. In particular, our Standard ML subset has a single LR(0) conflict that mingles an actual

Table 3: Number of initial LR(0) conflicting pairs remaining with the LR, LRR and NU tests employing successively LR(0), SLR(1), LALR(1), and LR(1) precision.

Precision	LR(0)			SLR(1)			LALR(1)	LR(1)		
Method	LR	LRR	NU	LR	LRR	NU	LR	LR	LRR	NU
Pascal	119	55	55	5	5	5	1	1	1	1
Mini C	153	11	10	5	5	4	1	1	1	1
ANSI C	261	13	2	13	13	2	1	1	1	1
ANSI C'	265	117	106	22	22	11	9	9	-	-
Standard ML	306	163	158	130	129	124	109	109	107	107
Small Elsa C++	509	285	239	25	22	22	24	24	-	-
Elsa C++	973	560	560	61	58	58	53	-	-	-

ambiguity with a conflict requiring an unbounded lookahead exploration: the measure would thus show no improvement when using our test. The measure is not comparable with the numbers of potential ambiguities reported by NU; for instance, NU(item_1) would report 89 potential ambiguities for Standard ML, and 52 for ANSI C'. Another means to compare ambiguity detection tools is thus investigated in the next subsection.

Although we ran our tests on a machine equipped with a 3.2GHz Xeon and 3GiB of physical memory, several tests employing LR(1) items exhausted the memory, resulting in the "-" entries in Table 3. The explosive number of LR(1) items is also responsible for a huge slowdown: for the small Elsa grammar, the NU test with SLR(1) items ran in 0.22 seconds, against more than 2 minutes for the corresponding canonical LR(1) test (and managed to return a better conflict report).

4.3 Micro-Benchmarks

Basten (2008) compared several means to detect ambiguities in context-free grammars, including our own implementation in GNU Bison, the AMBER generative test (Schröer, 2001), and the MSTA LR(k) parser generator (Makarov, 1999). Also confronted with the difficulty of measuring ambiguity in a meaningful way, he opted for a micro-benchmark approach, performing the tests on 36 small unambiguous grammars and 48 ambiguous ones from various sources.

4.3.1 Basten's Results

The conservative accuracy ratios Basten (2008) obtained with our tool, computed as the number of grammars correctly classified as unambiguous, divided by the number of tested unambiguous grammars, were of 61%, 69%, and 86% in the LR(0), SLR(1), and LR(1) modes respectively. This compares rather well to the LR(k) tests, where the ratio drops to 75%, with attempted k values as high as 50. Interestingly, when run against the same collection, our LRR test with LR(1) precision chokes on the same grammars as the LR(k) tests, and obtains the same 75% ratio. Furthermore, the grammars on which the NU(item_1) test failed were all of the same mold (1-, 2-, and 4-letters palindromes, and the RNA grammars RNA_3 and RNA_7 of Reeder et al. (2005)).

Table 4: Number of conflicts obtained with Bison, Brabrand et al.'s tool, and our tool in LRR and NU modes with various precision settings.

Method	actual	LR	HVRU	LRR		NU	
Precision	class	LALR(1)	$\geq$LR(0)	LR(1)	LR(0)	SLR(1)	LR(1)
90-10-042	LR(2)	2	0	14	7	7	6
98-05-030	non LR	1	10	26	0	0	0
98-08-215	LR(2)	1	0	0	0	0	0
03-02-124	LR(2)	1	0	0	0	0	0
03-09-027	LR(2)	2	0	0	0	0	0
03-09-081	LR(3)	2	0	0	0	0	0
05-03-114	LR(2)	1	0	0	0	0	0
Ada "is"	LR(2)	1	0	0	0	0	0
Ada calls	non-LR	1	0	0	1	0	0
C++ qualified IDs	non-LRR	1	5	21	0	0	0
Java modifiers	non-LR	31	0	0	3	0	0
Java names	non-LR	1	0	0	0	0	0
Java arrays	LR(2)	1	0	0	0	0	0
Java casts	LR(2)	1	0	0	0	0	0
Pascal typed	LR(2)	1	0	0	0	0	0
Set expressions	non-LR	8	19	119	2	2	2

4.3.2 A Larger Collection

We gathered a few more unambiguous grammars from programming languages constructs in order to improve the representativity of Basten's grammar collection in this domain.

The comp.compilers Collection A first set of seven unambiguous grammars was found in the comp.compilers archive when querying the word "conflict" and after ruling out ambiguous grammars and LL-related conflicts:[4]

90-10-042 an excerpt of the YACC syntax, which has an optional semicolon as end of rule marker that makes it LR(2);

98-05-030 a non LR excerpt of the Tiger syntax;

98-08-215 a LR(2) grammar;

03-02-124 a LR(2) excerpt of the C# grammar;

03-09-027 a LR(2) grammar;

03-09-081 a LR(3) grammar;

05-03-114 a LR(2) grammar.

[4] The names `xx-xx-xxx` are the message identifiers on the archive, respectively available at `http://compilers.iecc.com/comparch/article/xx-xx-xxx`.

Table 5: Accuracy ratios of each method (a) on our set of 16 small grammars, (b) on the complete set of 52 unambiguous small grammars, and (c) on the set of 26 non-LALR(1) small grammars.

	Method	LR		HVRU	LRR	NU		
	Precision	LALR(1)	LR(k)	$\geq$LR(0)	LR(1)	LR(0)	SLR(1)	LR(1)
(a)	Accuracy/improvement	0%	62%	81%	75%	75%	87%	87%
(b)	Overall accuracy	50%	69%	69%	75%	65%	75%	87%
(c)	Overall improvement	0%	42%	69%	50%	58%	65%	73%

The Literature Collection A second set of nine unambiguous grammars was compiled using grammars from the literature, notably from the literature on LR-Regular and noncanonical parsing techniques:

Ada "is" a LR(2) snippet of the Ada syntax (ANSI, 1983), pointed out by Baker (1981) and Boullier (1984);

Ada calls a non LR fragment of the Ada syntax, pointed out by Boullier (1984);

C++ qualified IDs a non LR-Regular portion of the C++ syntax (ISO, 1998);

Java modifiers a non LR excerpt of the Java syntax, which was detailed by Gosling et al. (1996) in their Sections 19.1.2 and 19.1.3;

Java names a non LR excerpt given in their Section 19.1.1;

Java arrays a LR(2) excerpt given in their Section 19.1.4;

Java casts a LR(2) excerpt given in their Section 19.1.5;

Pascal typed a LR(2) grammar for Pascal variable declarations that enforces type correctness, given by Tai (1979);

Set expressions a non LR grammar that distinguishes between arithmetic and set expressions, given by Čulik and Cohen (1973).

Results We ran several conservative ambiguity detection tests on Basten's grammar collection and on our small collection. Table 4 shows the results of our micro-benchmarks, and Table 5 compiles the accuracy ratios we obtained. Our small collection (a) contains only non-LALR(1) grammars, and as such the accuracy of the various tools can also be seen as an improvement ratio over LALR(1). The overall accuracy (b) score takes into account the complete collection of 52 unambiguous grammars using both our grammars and Basten's; 26 grammars are not LALR(1) in this full collection, giving rise to the overall improvement score (c).

The ability to freely specify lookahead lengths in a LR(k) parser improves over LALR(1) parsing, but is significantly less powerful than the methods that take an unbounded lookahead into account. An interesting point is that the results of our tool in LR(1) precision with Brabrand et al. horizontal and vertical ambiguity check are not highly correlated, and a simple conjunction of

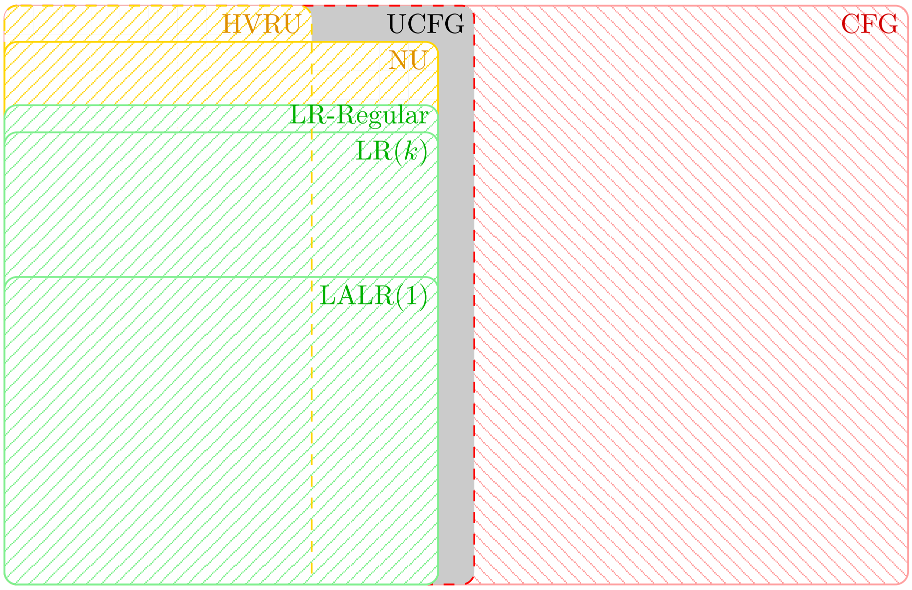


Figure 6: Grammar classes inclusions of various context-free grammar classes. The surface of each rectangle is roughly commensurate with its importance in the full collection of small grammars.

the two tools would obtain an overall 88% improvement rate, or 94% on our small collection only.

Figure 6 sums up the grammar class inclusions for the various methods we presented, and attempts to render their relative importance on the complete collection of 100 small grammars.

Let us finally point out that a much larger grammar collection would be needed in order to obtain more trustworthy micro-benchmark result. Such results might still not be very significant for large, complex grammars with a lot of interaction, where the precision of a method seems to be much more important than for small grammars: for instance, our NU method performs as well with SLR(1) precision as with LR(1) precision on our 16 small grammars (Table 4), but the results of Table 3 demonstrate a significant improvement when employing LR(1) items on real grammars.

Addendum Most recently, in their latest implementation of their tool, Brabrand et al. offer now the possibility to automatically unfold a grammar at a given depth from user-provided opening and closing parentheses. Thanks to one or two levels of unfolding, they can analyze correctly all the grammars of Table 4 as unambiguous.

There is still interest in NU testing. Besides having a better time complexity, the class of grammars it defines is not subsumed by HVRU with unfolding: for instance, grammars $\mathcal{G}_5$ and $\mathcal{G}_7$ presented in Section 4.1.3 are NU(item_0) but not HVRU for any level of unfolding.

The two examples $\mathcal{G}_6$ and $\mathcal{G}_7$ introduced in Section 4.1.3 that showed that

NU and HVRU were incomparable have practical counterparts in some existing grammars for programming languages:

- the already mentioned LR(2) excerpt of the YACC grammar is analyzed as potentially ambiguous by NU

$$S \to PS \mid P;S \mid \varepsilon,\ P \to i : R,\ R \to iS \mid \varepsilon \tag{90-10-042}$$

  unless we use item_2 approximations. Thus our algorithm does not bring any improvement over LR testing in this case.

- the following LR(2) grammar for expressions with casts, very similar to the grammar of Java casts discussed before, is reported as potentially ambiguous by Brabrand et al.'s tool for any amount of unfolding:

$$E \to E + F \mid F,\ F \to i \mid (E) \mid (i)E \tag{casts}$$

These two examples illustrate some typical constructs that either NU or HVRU cannot deal with, and further illustrate the interest of combining them.

5 Current Limitations

Our implementation is still a prototype. We describe several planned improvements (Sections 5.1 and 5.2), followed by a brief account on the difficulty of considering dynamic disambiguation filters and merge functions in the algorithm (Section 5.3).

5.1 Ambiguity Report

As mentioned in the beginning of Section 3, the ambiguity report returned by our tool is hard to interpret.

A first solution, also advocated by Brabrand et al. (2010), is to attempt to generate actual ambiguous inputs that exhibit the detected ambiguities. The ambiguity report would then comprise two parts, one for proven ambiguities with examples of input, and one for the potential ambiguities. The generation should only follow item pairs from which the potential ambiguities are reachable through ma relations, and stop whenever an ambiguity has been found or after having explored a given number of paths.

The good results Basten (2008) obtained with AMBER (Schröer, 2001) on his set of small ambiguous grammars emphasizes the interest for a mixed strategy, where the paths to potential ambiguities in ma^* could be employed to guide the generation of ambiguous sentential forms. The running time of AMBER on a full programming language grammar is currently rather prohibitive; running a generator on the portions of the grammar that might present an ambiguity according to our tool could improve it drastically. The initial experiments run by Basten in this direction are highly encouraging.

Displaying the (potentially) ambiguous paths in the grammar in a graphical form is a second possibility. This feature is implemented by ANTLRWorks, the development environment for ANTLR version 3 (Parr, 2007).

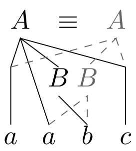


Figure 7: The shared parse forest for input *aabc* with grammar $\mathcal{G}_8$.

5.2 Running Time

The complexity of our algorithm is a square function of the grammar size. If, instead of item pairs, we considered deterministic states of items like LALR(1) does, the worst-case complexity would rise to an exponential function. Our algorithm is thus more robust.

Nonetheless, practical computations seem likely to be faster with LALR(1) item sets: a study of LALR(1) parsers sizes by Purdom (1974) showed that the size of the LALR(1) parser was usually a linear function of the size of the grammar. Therefore, all hope of analyzing large GLR grammars—like the Cobol grammar recovered by Lämmel and Verhoef (2001)—is not lost.

The theory behind noncanonical LALR parsing (Schmitz, 2006) might translate into a special case of our algorithm for ambiguity detection, yielding the missing tradeoff between SLR(1) and LR(1) precision.

5.3 Dynamic Disambiguation Filters

In contrast with its treatment of static precedence and associativity directives, our tool does not ignore potential ambiguities when the user has declared a merge function that might solve the issue. The rationale is simple: we do not know whether the merge function will actually solve the ambiguity. Consider for instance the rules

$$A \to aBc \mid aaBc,\ B \to ab \mid b. \qquad (\mathcal{G}_8)$$

Our tool reports an ambiguity on the item pair $(B \to ab\bullet,\quad B \to b\bullet)$, and is quite right: the input *aabc* is ambiguous. As shown in Figure 7, adding a merge function on the rules of B would not resolve the ambiguity: the merge function should be written for A.

If we consider arbitrary productions for B, a merge function might be useful only if the languages of the alternatives for B are not disjoint. We could thus improve our tool to detect some useless merge declarations. On the other hand, if the two languages are not equivalent, then there are cases where a merge function is needed on A—or even at a higher level. Ensuring equivalence is difficult, but could be attempted in some decidable cases, namely when we can detect that the languages of the alternatives of B are finite or regular, or using bisimulation equivalence (Caucal, 1990).

6 Conclusions

The paper reports on an ambiguity detection tool. In spite of its experimental state, the tool has been successfully used on a very difficult portion of the Standard ML grammar. The tool also improves on the dreaded LALR(1) conflicts report, albeit at a much higher computational price.

We hope that the need for such a tool, the results obtained with this first implementation, and the solutions described for the current limitations will encourage the investigation of better ambiguity detection techniques. The integration of our method with the one designed by Brabrand et al. is another promising solution.

Acknowledgements The work reported in this article was conducted at the Laboratoire I3S, Université de Nice - Sophia Antipolis & CNRS, France.

The author gratefully acknowledges the help received from Bas Basten with his grammar collection and from Claus Brabrand and Anders Møller with their ambiguity detection tool.

The author also thanks Jacques Farré for his help in the preparation of this paper and Sébastien Verel for granting him access to a fast computer.

References

Aho, A.V. and Ullman, J.D., 1972. *The Theory of Parsing, Translation, and Compiling. Volume I: Parsing*. Series in Automatic Computation. Prentice Hall. ISBN 0-13-914556-7. http://portal.acm.org/citation.cfm?id=SERIES11430.578789.

Aho, A.V., Johnson, S.C., and Ullman, J.D., 1975. Deterministic parsing of ambiguous grammars. *Communications of the ACM*, 18(8):441–452. doi:10.1145/360933.360969.

ANSI, 1983. *Reference Manual for the Ada Programming Language ANSI/MIL-STD-1815A-1983*. Springer. http://www.adahome.com/Resources/refs/83.html.

Baker, T.P., 1981. Extending lookahead for LR parsers. *Journal of Computer and System Sciences*, 22(2):243–259. doi:10.1016/0022-0000(81)90030-1.

Basten, H.J.S., 2008. The usability of ambiguity detection methods for context-free grammars. In Vinju, J. and Johnstone, A., editors, *LDTA'08*, volume 238(5) of *Electronic Notes in Theoretical Computer Science*, pages 35–46. Elsevier. doi:10.1016/j.entcs.2009.09.039.

Billot, S. and Lang, B., 1989. The structure of shared forests in ambiguous parsing. In *ACL'89*, pages 143–151. ACL Press. doi:10.3115/981623.981641.

Boullier, P., 1984. *Contribution à la construction automatique d'analyseurs lexicographiques et syntaxiques*. Thèse d'État, Université d'Orléans.

Brabrand, C., Giegerich, R., and Møller, A., 2010. Analyzing ambiguity of context-free grammars. *Science of Computer Programming*. doi:10.1016/j.scico.2009.11.002. In Press.

Cantor, D.G., 1962. On the ambiguity problem of Backus systems. *Journal of the ACM*, 9(4):477–479. doi:10.1145/321138.321145.

Caucal, D., 1990. Graphes canoniques de graphes algébriques. *RAIRO - Theoretical Informatics and Applications*, 24(4):339–352. http://www.inria.fr/rrrt/rr-0872.html.

Chomsky, N. and Schützenberger, M.P., 1963. The algebraic theory of context-free languages. In Braffort, P. and Hirshberg, D., editors, *Computer Programming and Formal Systems*, volume 35 of *Studies in Logic*, pages 118–161. North-Holland Publishing. doi:10.1016/S0049-237X(08)72023-8.

Čulik, K. and Cohen, R., 1973. LR-Regular grammars—an extension of LR(k) grammars. *Journal of Computer and System Sciences*, 7(1):66–96. doi:10.1016/S0022-0000(73)80050-9.

Donnely, C. and Stallman, R., 2006. *Bison version 2.3*. http://www.gnu.org/software/bison/manual/.

Earley, J., 1970. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102. doi:10.1145/362007.362035.

Floyd, R.W., 1962. On ambiguity in phrase structure languages. *Communications of the ACM*, 5(10):526. doi:10.1145/368959.368993.

Gosling, J., Joy, B., and Steele, G., 1996. *The Java™ Language Specification*. Addison-Wesley, first edition. ISBN 0-201-63451-1. http://java.sun.com/docs/books/jls/.

Grune, D. and Jacobs, C.J.H., 1990. *Parsing Techniques: A Practical Guide*. Ellis Horwood Limited. ISBN 0-13-651431-6. http://www.cs.vu.nl/~dick/PTAPG.html.

Heilbrunner, S., 1983. Tests for the LR-, LL-, and LC-Regular conditions. *Journal of Computer and System Sciences*, 27(1):1–13. doi:10.1016/0022-0000(83)90026-0.

Hunt III, H.B., Szymanski, T.G., and Ullman, J.D., 1974. Operations on sparse relations and efficient algorithms for grammar problems. In *15th Annual Symposium on Switching and Automata Theory*, pages 127–132. IEEE Computer Society. doi:10.1109/SWAT.1974.21.

ISO, 1998. *ISO/IEC 14882:1998: Programming Languages — C++*. International Organization for Standardization, Geneva, Switzerland.

Johnson, S.C., 1975. YACC — yet another compiler compiler. Computing science technical report 32, AT&T Bell Laboratories, Murray Hill, New Jersey.

Kahrs, S., 1993. Mistakes and ambiguities in the definition of Standard ML. Technical Report ECS-LFCS-93-257, University of Edinburgh, LFCS. http://www.lfcs.inf.ed.ac.uk/reports/93/ECS-LFCS-93-257/.

Kernighan, B.W. and Ritchie, D.M., 1988. *The C Programming Language*. Prentice-Hall. ISBN 0-13-110362-8.

Klint, P. and Visser, E., 1994. Using filters for the disambiguation of context-free grammars. In Pighizzini, G. and San Pietro, P., editors, *ASMICS Workshop on Parsing Theory*, Technical Report 126-1994, pages 89–100. Università di Milano. http://citeseer.ist.psu.edu/klint94using.html.

Klint, P., Lämmel, R., and Verhoef, C., 2005. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering and Methodology*, 14(3):331–380. doi:10.1145/1072997.1073000.

Lämmel, R. and Verhoef, C., 2001. Semi-automatic grammar recovery. *Software: Practice & Experience*, 31:1395–1438. doi:10.1002/spe.423.

Lee, P., 1997. *Using the SML/NJ System*. Carnegie Mellon University. http://www.cs.cmu.edu/~petel/smlguide/smlnj.htm.

Makarov, V., 1999. *MSTA (syntax description translator)*. http://cocom.sourceforge.net/msta.html.

McPeak, S. and Necula, G.C., 2004. Elkhound: A fast, practical GLR parser generator. In Duesterwald, E., editor, *CC'04*, volume 2985 of *Lecture Notes in Computer Science*, pages 73–88. Springer. doi:10.1007/b95956.

Milner, R., Tofte, M., Harper, R., and MacQueen, D., 1997. *The definition of Standard ML*. MIT Press, revised edition. ISBN 0-262-63181-4.

Mohri, M. and Nederhof, M.J., 2001. Regular approximations of context-free grammars through transformation. In Junqua, J.C. and van Noord, G., editors, *Robustness in Language and Speech Technology*, volume 17 of *Text, Speech and Language Technology*, chapter 9, pages 153–163. Kluwer Academic Publishers. http://citeseer.ist.psu.edu/mohri00regular.html.

Parr, T.J., 2007. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Programmers. ISBN 0-9787392-5-6.

Poplawski, D.A., 1979. On LL-Regular grammars. *Journal of Computer and System Sciences*, 18(3):218–227. doi:10.1016/0022-0000(79)90031-X.

Purdom, P., 1974. The size of LALR(1) parsers. *BIT Numerical Mathematics*, 14(3):326–337. doi:10.1007/BF01933232.

Reeder, J., Steffen, P., and Giegerich, R., 2005. Effective ambiguity checking in biosequence analysis. *BMC Bioinformatics*, 6:153. doi:10.1186/1471-2105-6-153.

Rossberg, A., 2006. Defects in the revised definition of Standard ML. Technical report, Saarland University, Saarbrücken, Germany. http://ps.uni-sb.de/Papers/paper_info.php?label=sml-defects.

Schmitz, S., 2006. Noncanonical LALR(1) parsing. In Dang, Z. and Ibarra, O.H., editors, *DLT'06*, volume 4036 of *Lecture Notes in Computer Science*, pages 95–107. Springer. doi:10.1007/11779148_10.

Schmitz, S., 2007. Conservative ambiguity detection in context-free grammars. In Arge, L., Cachin, C., Jurdziński, T., and Tarlecki, A., editors, *ICALP'07*, volume 4596 of *Lecture Notes in Computer Science*, pages 692–703. Springer. doi:10.1007/978-3-540-73420-8_60.

Schröer, F.W., 2001. AMBER, an ambiguity checker for context-free grammars. Technical report, compilertools.net. http://accent.compilertools.net/Amber.html.

Scott, E. and Johnstone, A., 2006. Right nulled GLR parsers. *ACM Transactions on Programming Languages and Systems*, 28(4):577–618. doi:10.1145/1146809.1146810.

Szymanski, T.G. and Williams, J.H., 1976. Noncanonical extensions of bottom-up parsing techniques. *SIAM Journal on Computing*, 5(2):231–250. doi:10.1137/0205019.

Tai, K.C., 1979. Noncanonical SLR(1) grammars. *ACM Transactions on Programming Languages and Systems*, 1(2):295–320. doi:10.1145/357073.357083.

Tomita, M., 1986. *Efficient Parsing for Natural Language*. Kluwer Academic Publishers. ISBN 0-89838-202-5.

van den Brand, M., Scheerder, J., Vinju, J.J., and Visser, E., 2002. Disambiguation filters for scannerless generalized LR parsers. In Horspool, R.N., editor, *CC'02*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158. Springer. doi:10.1007/3-540-45937-5_12.