An Experimental Ambiguity Detection Tool

Sylvain Schmitz[1]

Laboratoire I3S
Université de Nice - Sophia Antipolis & CNRS
France

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 remove 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.

Keywords: grammar verification, disambiguation, GLR

1 Introduction

With the broad availability of parser generators that implement the Generalized LR (GLR) [33] or the Earley [11] algorithm, it might seem that the 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 for a survey [31]).

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 [1]. The language definition should then include some *disambiguation* rules to decide which parse to choose.

---

[1] Email: schmitz@i3s.unice.fr

In this paper, we present a tool for GNU Bison [10][2] that pinpoints possible ambiguities in context-free grammars (CFGs). Grammar and parser developers can then use the ambiguities reported by the tool to write disambiguation rules where they are needed. Since the problem of finding all the ambiguities in a CFG is undecidable [6,8], our tool implements a conservative algorithm [30]: 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 [18].

- We first describe a well-known difficult subset of the syntax of Standard ML [23] (Section 2.1) that combines a genuine ambiguity with a LR conflict requiring unbounded lookahead (Section 2.2). A generalized parser accomplishes to parse 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). We complete this overview of our tool with more experimental results in Section 4.

- At last, 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 [15]. Unlike the grammars sometimes provided in other programming language references, the grammar defined by Miller et al. [23, 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 [16] 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 follows in Figure 1.[3]

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

---

[2] The modified Bison source is available from the author's webpage, at the address http://www.i3s.unice.fr/~schmitz/.

[3] We translated the original rules from their extended representation into classical BNF. We note $\langle nonterminals \rangle$ between angle brackets and *terminals* as such, except for the terminal alternation symbol $'|'$, quoted in order to avoid confusion with the choice meta character |.

$$\langle dec \rangle \rightarrow \textbf{fun } \langle fvalbind \rangle$$
$$\langle fvalbind \rangle \rightarrow \langle sfvalbind \rangle \mid \langle fvalbind \rangle \; '|' \; \langle sfvalbind \rangle$$
$$\langle sfvalbind \rangle \rightarrow vid \; \langle atpats \rangle = \langle exp \rangle$$
$$\langle atpats \rangle \rightarrow \langle atpat \rangle \mid \langle atpats \rangle \; \langle atpat \rangle$$
$$\langle exp \rangle \rightarrow \textbf{case } \langle exp \rangle \textbf{ of } \langle match \rangle \mid vid$$
$$\langle match \rangle \rightarrow \langle mrule \rangle \mid \langle match \rangle \; '|' \; \langle mrule \rangle$$
$$\langle mrule \rangle \rightarrow \langle pat \rangle => \langle exp \rangle$$
$$\langle pat \rangle \rightarrow vid \; \langle atpat \rangle$$
$$\langle atpat \rangle \rightarrow vid$$

Fig. 1. Syntax of function value binding and **case** expressions in Standard ML.

matching rules $\langle mrule \rangle$ of form $\langle pat \rangle => \langle exp \rangle$, separated by alternation symbols "|".

Example 2.1 Using mostly these rules, the filter function of the SML/NJ Library could be written [21] as:

```
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 [10], 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)]
```
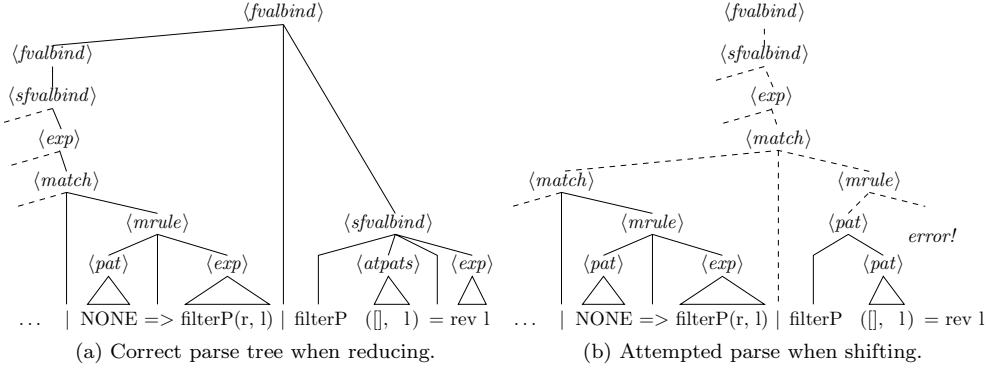
(a) Correct parse tree when reducing.   (b) Attempted parse when shifting.

Fig. 2. Partial parse trees corresponding to the two actions in conflict on Example 2.1.

The conflict takes place just before "| filterP ([], l) = rev l" with the program of Example 2.1.

   If we choose 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.

Example 2.2 The issue is made further complicated by a dangling ambiguity:

case a **of** b $\Rightarrow$ **case** b **of** c $\Rightarrow$ c | d $\Rightarrow$ 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.

   Our two examples show that we cannot blindly choose one particular action over the other. Nonetheless, 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.

   This issue in the syntax of Standard ML is one of its few major defects according to a survey by Rossberg [28]:

   [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. [4]

---

[4] Some deterministic parsing algorithms—LR-Regular [9,3], noncanonical [32,12], or LL-Regular [25,24]—albeit perhaps less known, are able to exploit unbounded lookahead lengths. Our ambiguity detection

Fig. 3. The shared parse forest for the input of Example 2.2.

2.3   General Parsing

A general parser returns all the possible parses for the provided input, and as such discards the incorrect parse of Figure 2b and only returns the correct one of Figure 2a. In particular, a generalized LALR(1) parser explores the two possibilities of the conflict, until it reaches the = sign, at which point the incorrect partial parse of Figure 2b fails.

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 2.1, and overlook the ambiguity triggered by embedded case expressions like the one of Example 2.2. 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 [4]. Figure 3 shows a shared forest for our ambiguity, with a topmost ⟨match⟩ node that merges the two alternative interpretations of the input of Example 2.2.

- Klint and Visser [19] 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 [34], Elkhound [22], and Bison [10].

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 [10] and of Elkhound [22] 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

---

algorithm employs similar principles.

obtained 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 2.3 Let us suppose that the user has found out the ambiguity of Example 2.2, 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 [34].

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 $\Rightarrow$ b | c $\Rightarrow$ **case** c **of** d $\Rightarrow$ d | e $\Rightarrow$ e

Indeed, with this input, the two shared $\langle match \rangle$ nodes are obtained through reductions using the same rule $\langle match \rangle \rightarrow \langle match \rangle' | '\langle mrule \rangle$. 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 is:

```
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 2.2 and 2.3. Furthermore, the re-

ported ambiguities do not mention anything visibly related to the difficult conflict of Example 2.1.

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. The algorithm in its full generality is described in [30], along with the proof that all ambiguities are caught, and more insights on the false positives returned along the way.

We detail here the algorithm on the relevant portion of our grammar, and consider to this end approximations based on LR(0) items: a dot in a grammar production $A \rightarrow \alpha \cdot \beta$ 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. 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 $(S \rightarrow \cdot \langle dec \rangle \ \$, \ S \rightarrow \cdot \langle dec \rangle \ \$)$ at the beginning of all trees, and ending on a pair $(S \rightarrow \langle dec \rangle \ \$ \cdot, \ S \rightarrow \langle dec \rangle \ \$ \cdot)$. Between these, we pair items that could be reached upon reading a common sentence prefix, hence following trees that derive the same sentence.

3.1 Example Run

Let us start with the couple 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:

$$(\langle match \rangle \rightarrow \langle match \rangle \cdot \ '|' \ \langle mrule \rangle, \ \langle exp \rangle \rightarrow \textbf{case} \ \langle exp \rangle \ \textbf{of} \ \langle match \rangle \cdot) \qquad (1)$$

Unlike Bison, 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 $\langle exp \rangle \rightarrow \textbf{case} \ \langle exp \rangle \ \textbf{of} \ \langle match \rangle$, 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 couples:

$$(\langle match \rangle \rightarrow \langle match \rangle \cdot \ '|' \ \langle mrule \rangle, \ \langle mrule \rangle \rightarrow \langle pat \rangle => \langle exp \rangle \cdot) \qquad (2)$$

$$(\langle match \rangle \rightarrow \langle match \rangle \cdot \ '|' \ \langle mrule \rangle, \ \langle exp \rangle \rightarrow \textbf{case} \ \langle exp \rangle \cdot \textbf{of} \ \langle match \rangle) \qquad (3)$$

$$(\langle match \rangle \rightarrow \langle match \rangle \cdot \ '|' \ \langle mrule \rangle, \ \langle sfvalbind \rangle \rightarrow vid \ \langle atpats \rangle = \langle exp \rangle \cdot) \qquad (4)$$

Applying the same idea to the pair (2), we should explore all the items with the dot to the right of $\langle mrule \rangle$.

$$(\langle match \rangle \rightarrow \langle match \rangle \cdot \ '|' \ \langle mrule \rangle, \ \langle match \rangle \rightarrow \langle mrule \rangle \cdot) \qquad (5)$$

$$(\langle match \rangle \rightarrow \langle match \rangle \cdot \ '|' \ \langle mrule \rangle, \ \langle match \rangle \rightarrow \langle match \rangle \ '|' \ \langle mrule \rangle \cdot) \qquad (6)$$

At this point, we find $\langle match\rangle\rightarrow\langle match\rangle\bullet\,'|'\,\langle mrule\rangle$, 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 (5) and (6) are the same, and we report the pairs as potential ambiguities.

Our ambiguity detection is not over yet: from (4), we could reach successively (showing only the relevant possibilities):

$$(\langle match\rangle\rightarrow\langle match\rangle\bullet\,'|'\,\langle mrule\rangle,\quad\langle fvalbind\rangle\rightarrow\langle sfvalbind\rangle\bullet)\tag{7}$$

$$(\langle match\rangle\rightarrow\langle match\rangle\bullet\,'|'\,\langle mrule\rangle,\quad\langle fvalbind\rangle\rightarrow\langle fvalbind\rangle\bullet\,'|'\,\langle sfvalbind\rangle)\tag{8}$$

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:

$$(\langle match\rangle\rightarrow\langle match\rangle\,'|'\,\bullet\langle mrule\rangle,\quad\langle fvalbind\rangle\rightarrow\langle fvalbind\rangle\,'|'\,\bullet\langle sfvalbind\rangle)\tag{9}$$

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

$$(\langle mrule\rangle\rightarrow\bullet\langle pat\rangle{=}{>}\langle exp\rangle,\quad\langle fvalbind\rangle\rightarrow\langle fvalbind\rangle\,'|'\,\bullet\langle sfvalbind\rangle)\tag{10}$$

$$(\langle mrule\rangle\rightarrow\bullet\langle pat\rangle{=}{>}\langle exp\rangle,\quad\langle sfvalbind\rangle\rightarrow\bullet\,vid\,\langle atpats\rangle=\langle exp\rangle)\tag{11}$$

$$(\langle pat\rangle\rightarrow\bullet\,vid\,\langle atpat\rangle,\quad\langle sfvalbind\rangle\rightarrow\bullet\,vid\,\langle atpats\rangle=\langle exp\rangle)\tag{12}$$

$$(\langle pat\rangle\rightarrow vid\bullet\langle atpat\rangle,\quad\langle sfvalbind\rangle\rightarrow vid\bullet\langle atpats\rangle=\langle exp\rangle)\tag{13}$$

$$(\langle pat\rangle\rightarrow vid\bullet\langle atpat\rangle,\quad\langle atpats\rangle\rightarrow\bullet\langle atpat\rangle)\tag{14}$$

$$(\langle pat\rangle\rightarrow vid\langle atpat\rangle\bullet,\quad\langle atpats\rangle\rightarrow\langle atpat\rangle\bullet)\tag{15}$$

$$(\langle mrule\rangle\rightarrow\langle pat\rangle\bullet{=}{>}\langle exp\rangle,\quad\langle atpats\rangle\rightarrow\langle atpat\rangle\bullet)\tag{16}$$

$$(\langle mrule\rangle\rightarrow\langle pat\rangle\bullet{=}{>}\langle exp\rangle,\quad\langle sfvalbind\rangle\rightarrow vid\,\langle atpats\rangle\bullet=\langle exp\rangle)\tag{17}$$

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 the two possibilities in conflict in Example 2.1.

3.2 Overview of the Algorithm

The example run detailed above relates couples 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 (8) and (9), but also between (14) and (15),

mae for downwards closures, holding for instance between pairs (9) and (10),

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 (2) and (5). Formally, our notion of a conflict coincides with that of Aho and Ullman [2, Theorem 5.9].

The algorithm thus constructs the image of the initial pair $(S' \rightarrow \bullet S\$, \ S' \rightarrow \bullet S\$)$ by the ma^* relation. If at some point we reach a pair holding twice the same item from a pair with different items, we report an ambiguity.

The eligible single moves from item to item are in fact the transitions in a *nondeterministic LR(0) automaton* (thereafter called LR(0) NFA). The size of the ma relation is bounded by the square of the size of this 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 is bounded by $\mathcal{O}((|\mathcal{G}|\,|P|)^2)$.

3.3 Implementation Details

The experimental tool currently implements the algorithm with LR(0), SLR(1), 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 changes a few details:

- We construct a nondeterministic automaton [14,13] whose states are either the items of form $A \rightarrow \alpha \bullet \beta$, 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 (2) and before reaching (5):

$$(\langle match \rangle \rightarrow \langle match \rangle \bullet \ '|' \ \langle mrule \rangle, \ \ \langle mrule \rangle \bullet). \tag{18}$$

  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.

- We consider the associativity and static precedence directives [1] of Bison and thus we do not report resolved ambiguities.

- 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. The same issue was met by McPeak and Necula with Elkhound [22], 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 dominates B, noted $A \searrow B$, if there is a rule $A \rightarrow \alpha B$; our order is then $\searrow^*$. In a reduce/reduce conflict between reductions to A and B, we follow the reduction of A if $A \not\searrow^* B$ or if both $A \searrow^* B$ and $B \searrow^* A$.

Table 1
Reported ambiguities in the grammars from [30].

Grammar	actual class	Bison	HVRU [5]	NU(item_0)
$\mathcal{G}_3^n$	LR(2^n)	1	-	0
$\mathcal{G}_4^n$	ambiguous	1	-	1
$\mathcal{G}_5$	non-LRR	1	-	0
$\mathcal{G}_6$	non-LRR	6	0	9
$\mathcal{G}_7$	LR(0)	0	1	0

4 Experimental Comparisons

The choice of a conservative ambiguity detection algorithm is currently rather limited. Several parsing techniques define subsets of the unambiguous grammars, and beyond LR(k) parsing, two major parsing strategies exist: LR-Regular parsing [9], which in practice explores a regular cover of the right context of LR conflicts with a finite automaton [3], and noncanonical parsing [32], where the exploration is performed by the parser itself. Since we follow the latter principle with our algorithm, we call it a *noncanonical unambiguity* (NU) test.

A different approach, unrelated to any parsing method, was proposed by Brabrand et al. [5] with their horizontal and vertical unambiguity check (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 [27].

We implemented a LR and a LRR test using the same item pairing technique as our NU algorithm. We present experimental comparisons with these, as well as with the HVRU algorithm when the data is available.

4.1 General Comparisons

The formal comparisons of our algorithm with various other methods given in [30] are sustained by several small grammars. Table 1 compiles the results obtained on these grammars. The "Bison" 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, and the "NU(item_0)" column the number of potential ambiguities reported by our algorithm with LR(0) items.

4.1.1 LR and LR-Regular

The grammar families $\mathcal{G}_3^n$ and $\mathcal{G}_4^n$ demonstrate the complexity gains with our algorithm as compared to LR(k) parsing:

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

$$S \rightarrow A \mid B_n a, \ A \rightarrow Aaa \mid a, \ B_1 \rightarrow aa, \ B_2 \rightarrow B_1 B_1, \ \ldots, \ B_n \rightarrow B_{n-1} B_{n-1}. \qquad (\mathcal{G}_4^n)$$

Table 2
Reported potential ambiguities in the RNA grammars from [27].

Grammar	actual class	Bison	HVRU [5]	NU(item_1)
$\mathcal{G}_1$	ambiguous	30	6	14
$\mathcal{G}_2$	ambiguous	33	7	13
$\mathcal{G}_3$	non-LR	4	0	2
$\mathcal{G}_4$	SLR(1)	0	0	0
$\mathcal{G}_5$	SLR(1)	0	0	0
$\mathcal{G}_6$	LALR(1)	0	0	0
$\mathcal{G}_7$	non-LR	5	0	3
$\mathcal{G}_8$	LALR(1)	0	0	0

While a $\text{LR}(2^n)$ 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.

Grammar $\mathcal{G}_5$ is a non-LRR [9] grammar with rules

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

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

4.1.2 Horizontal and Vertical Ambiguity

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

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

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

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

is a LR(0) grammar, and the test of Brabrand et al. finds it horizontally ambiguous and not vertically ambiguous. For completeness, we also give the results of our tool on the RNA grammars of Reeder et al. [27] in Table 2.

4.2 Experiments with Programming Languages Grammars

We ran the LR, LRR and NU tests 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/`, LALR(1) except for a dangling else ambiguity,

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

ANSI C [17, Appendix A.13], also retrieved from the `comp.compilers` FTP. The grammar is LALR(1), except for a dangling else ambiguity. The **ANSI C'** gram-

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)		
Test	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	-	-	-

mar is the same grammar modified by setting typedef names to be a nonterminal, with a single production ⟨*typedef-name*⟩→*identifier*. The modification reflects the fact that GLR parsers should not rely on the *lexer hack* for disambiguation.

Standard ML, extracted from the language definition [23, 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 [22], 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 (1)—that can reach a point of ambiguity—for instance pair (5)—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 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'.

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. 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 LR(1) test (and managed to return a better conflict report).
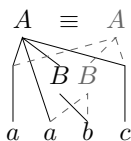
Fig. 4. The shared parse forest for input *aabc* with grammar $\mathcal{G}_8$.

5 Current Limitations

Our implementation is still a prototype. We describe several planned improvements (Sections 5.1 and 5.2), followed by a small 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, already adopted by Brabrand et al. [5], is to attempt to generate actually ambiguous inputs that match the detected ambiguities. The ambiguity report would then comprise of 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 finding the ambiguity or after having explored a given number of paths.

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 the upcoming version 3 of ANTLR [24].

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 [26] 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 [20]—is not lost.

The theory behind noncanonical LALR parsing [29] translates well into a special case of our algorithm for ambiguity detection, and future versions of the tool should implement it.

5.3 Dynamic Disambiguation Filters

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

$$A{\rightarrow}aBc \mid aaBc, \; B{\rightarrow}ab \mid b. \tag{$\mathcal{G}_8$}$$

Our tool reports an ambiguity on the item pair $(B{\rightarrow}ab\bullet, \quad B{\rightarrow}b\bullet)$, and is quite right: the input *aabc* is ambiguous. As shown in Figure 4, 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 [7].

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.

Acknowledgement

References

[1] Aho, A. V., S. C. Johnson and J. D. Ullman, *Deterministic parsing of ambiguous grammars*, Communications of the ACM **18** (1975), pp. 441–452.
DOI 10.1145/360933.360969

[2] Aho, A. V. and J. D. Ullman, "The Theory of Parsing, Translation, and Compiling. Volume I: Parsing," Series in Automatic Computation, Prentice Hall, 1972.

[3] Bermudez, M. E. and K. M. Schimpf, *Practical arbitrary lookahead LR parsing*, Journal of Computer and System Sciences **41** (1990), pp. 230–250.
DOI 10.1016/0022-0000(90)90037-L

[4] Billot, S. and B. Lang, *The structure of shared forests in ambiguous parsing*, in: *ACL'89* (1989), pp. 143–151.
URL http://www.aclweb.org/anthology/P89-1018

[5] Brabrand, C., R. Giegerich and A. Møller, *Analyzing ambiguity of context-free grammars*, in: M. Balík and J. Holub, editors, *CIAA'07*, 2007, to appear in *Lecture Notes in Computer Science*.
URL http://www.brics.dk/~brabrand/grambiguity/

[6] Cantor, D. G., *On the ambiguity problem of Backus systems*, Journal of the ACM **9** (1962), pp. 477–479.
DOI 10.1145/321138.321145

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

[8] Chomsky, N. and M. P. Schützenberger, *The algebraic theory of context-free languages*, in: P. Braffort and D. Hirshberg, editors, *Computer Programming and Formal Systems*, Studies in Logic, North-Holland Publishing, 1963 pp. 118–161.

[9] Čulik, K. and R. Cohen, *LR-Regular grammars—an extension of LR(k) grammars*, Journal of Computer and System Sciences **7** (1973), pp. 66–96.

[10] Donnely, C. and R. Stallman, "Bison version 2.1," (2005).
URL http://www.gnu.org/software/bison/manual/

[11] Earley, J., *An efficient context-free parsing algorithm*, Communications of the ACM **13** (1970), pp. 94–102.
DOI 10.1145/362007.362035

[12] Fortes Gálvez, J., S. Schmitz and J. Farré, *Shift-resolve parsing: Simple, linear time, unbounded lookahead*, in: O. H. Ibarra and H.-C. Yen, editors, *CIAA'06*, Lecture Notes in Computer Science **4094** (2006), pp. 253–264.
DOI 10.1007/11812128_24

[13] Grune, D. and C. J. H. Jacobs, "Parsing Techniques: A Practical Guide," Ellis Horwood Limited, 1990.
URL http://www.cs.vu.nl/~dick/PTAPG.html

[14] Hunt III, H. B., T. G. Szymanski and J. D. Ullman, *Operations on sparse relations and efficient algorithms for grammar problems*, in: *15th Annual Symposium on Switching and Automata Theory* (1974), pp. 127–132.

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

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

[17] Kernighan, B. W. and D. M. Ritchie, "The C Programming Language," Prentice-Hall, 1988.

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

[19] Klint, P. and E. Visser, *Using filters for the disambiguation of context-free grammars*, in: G. Pighizzini and P. San Pietro, editors, *ASMICS Workshop on Parsing Theory*, Technical Report 126-1994 (1994), pp. 89–100.
URL http://citeseer.ist.psu.edu/klint94using.html

[20] Lämmel, R. and C. Verhoef, *Semi-automatic grammar recovery*, Software: Practice & Experience **31** (2001), pp. 1395–1438.
DOI 10.1002/spe.423

[21] Lee, P., "Using the SML/NJ System," Carnegie Mellon University (1997).
URL http://www.cs.cmu.edu/~petel/smlguide/smlnj.htm

[22] McPeak, S. and G. C. Necula, *Elkhound: A fast, practical GLR parser generator*, in: E. Duesterwald, editor, *CC'04*, Lecture Notes in Computer Science **2985** (2004), pp. 73–88.
DOI 10.1007/b95956

[23] Milner, R., M. Tofte, R. Harper and D. MacQueen, "The definition of Standard ML," MIT Press, 1997, revised edition.

[24] Parr, T. J., "The Definitive ANTLR Reference: Building Domain-Specific Languages," The Pragmatic Programmers, 2007.

[25] Poplawski, D. A., *On LL-Regular grammars*, Journal of Computer and System Sciences **18** (1979), pp. 218–227.
DOI 10.1016/0022-0000(79)90031-X

[26] Purdom, P., *The size of LALR(1) parsers*, BIT Numerical Mathematics **14** (1974), pp. 326–337.
DOI 10.1007/BF01933232

[27] Reeder, J., P. Steffen and R. Giegerich, *Effective ambiguity checking in biosequence analysis*, BMC Bioinformatics **6** (2005), p. 153.
DOI 10.1186/1471-2105-6-153

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

[29] Schmitz, S., *Noncanonical LALR(1) parsing*, in: Z. Dang and O. H. Ibarra, editors, *DLT'06*, Lecture Notes in Computer Science **4036** (2006), pp. 95–107.
DOI 10.1007/11779148_10

[30] Schmitz, S., *Conservative ambiguity detection in context-free grammars*, in: *ICALP'07*, 2007, to appear in *Lecture Notes in Computer Science*.
URL http://www.i3s.unice.fr/~mh/RR/2006/RR-06.30-S.SCHMITZ.pdf

[31] Scott, E. and A. Johnstone, *Right nulled GLR parsers*, ACM Transactions on Programming Languages and Systems **28** (2006), pp. 577–618.
DOI 10.1145/1146809.1146810

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

[33] Tomita, M., "Efficient Parsing for Natural Language," Kluwer Academic Publishers, 1986.

[34] van den Brand, M., J. Scheerder, J. J. Vinju and E. Visser, *Disambiguation filters for scannerless generalized LR parsers*, in: R. N. Horspool, editor, *CC'02*, Lecture Notes in Computer Science **2304** (2002), pp. 143–158.
URL http://www.springerlink.com/content/03359k0cerupftfh/