

LABORATOIRE



INFORMATIQUE, SIGNAUX ET SYSTÈMES
DE SOPHIA ANTIPOLIS
UMR 6070

AN EXPERIMENTAL AMBIGUITY DETECTION TOOL

Sylvain Schmitz

Projet LANGAGES

Rapport de recherche
ISRN I3S/RR-2006-37-FR

Décembre 2006

RÉSUMÉ :

Bien que les programmes n'admettent qu'une seule interprétation, les grammaires employées en pratique pour décrire leur syntaxe sont souvent ambiguës, complétées à l'aide de règles de désambiguation. Il peut être délicat de s'assurer que ces règles parviennent à retirer toute ambiguïté tout en préservant le langage originellement décrit. Nous présentons un outil expérimental pour la détection d'ambiguïtés implémenté dans GNU/bison, et nous illustrons comment il peut aider au cours du développement d'une grammaire pour un sous-ensemble de Standard ML.

MOTS CLÉS :

vérification de grammaires, désambiguation, GLR

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.

KEY WORDS :

grammar verification, disambiguation, GLR

An Experimental Ambiguity Detection Tool

Sylvain Schmitz

Laboratoire I3S, Université de Nice - Sophia Antipolis & CNRS, France
schmitz@i3s.unice.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 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.

Key words: grammar verification, disambiguation, GLR

1 Introduction

With the broad availability of parser generators that implement the Generalized LR (GLR) [34] 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 [32]).

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

In this paper, we present a tool for GNU Bison [10]¹ 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 [20].

¹The modified Bison source is available from the author's webpage, at the address <http://www.i3s.unice.fr/~schmitz/>.

$$\begin{aligned}
\langle dec \rangle &\rightarrow \mathbf{fun} \langle fvalbind \rangle \\
\langle fvalbind \rangle &\rightarrow \langle sivalbind \rangle \mid \langle fvalbind \rangle \text{'|'} \langle sivalbind \rangle \\
\langle sivalbind \rangle &\rightarrow \mathit{vid} \langle atpats \rangle = \langle exp \rangle \\
\langle atpats \rangle &\rightarrow \langle atpat \rangle \mid \langle atpats \rangle \langle atpat \rangle \\
\langle exp \rangle &\rightarrow \mathbf{case} \langle exp \rangle \mathbf{of} \langle match \rangle \mid \mathit{vid} \\
\langle match \rangle &\rightarrow \langle mrule \rangle \mid \langle match \rangle \text{'|'} \langle mrule \rangle \\
\langle mrule \rangle &\rightarrow \langle pat \rangle \Rightarrow \langle exp \rangle \\
\langle pat \rangle &\rightarrow \mathit{vid} \langle atpat \rangle \\
\langle atpat \rangle &\rightarrow \mathit{vid}
\end{aligned}$$

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

- We first describe a well-known difficult subset of the syntax of Standard ML [24] (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. [24, 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 [17] 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 1.²

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

²We 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 \mid .

expression $\langle expr \rangle$ using the rule $\langle s\text{valbind} \rangle \rightarrow \text{vid } \langle at\text{pats} \rangle = \langle exp \rangle$. 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 $\langle pat \rangle \Rightarrow \langle exp \rangle$, separated by alternation symbols “|”.

Example 1 Using mostly these rules, the `filter` function of the SML/NJ Library could be written [22] 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)]
```

The conflict takes place just before “| filterP ([], l) = rev l” with the program of Example 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 The issue is made further complicated by 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.

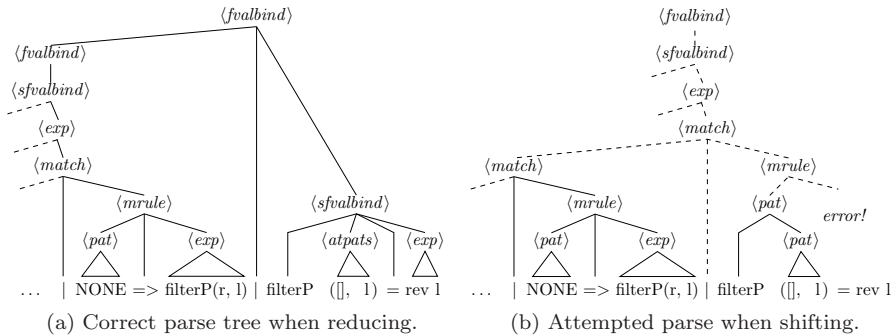


Figure 2: Partial parse trees corresponding to the two actions in conflict on Example 1.

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 ([, 1) = rev l” indicates that the alternative is at the topmost function value binding $\langle fvalbind \rangle$ 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 $\langle atpat \rangle$ 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 [29]:

[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.³

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 (see Appendix A for a more detailed description of the process).

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

³Some deterministic parsing algorithms—LR-Regular [9, 3], noncanonical [33, 12], or LL-Regular [26, 25]—albeit perhaps less known, are able to exploit unbounded lookahead lengths. Our ambiguity detection algorithm employs similar principles.

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 [35].

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 $\langle match \rangle \rightarrow \langle match \rangle' | \langle mrule \rangle$. 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 and 3. Furthermore, the reported ambiguities do not mention anything visibly related to the difficult conflict of Example 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:

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

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 $\langle exp \rangle \rightarrow \mathbf{case} \langle exp \rangle \mathbf{of} \langle match \rangle$, 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:

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

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

$$(\langle match \rangle \rightarrow \langle match \rangle \cdot ' | \langle mrule \rangle, \langle sfulbind \rangle \rightarrow \mathbf{vid} \langle atpats \rangle = \langle exp \rangle \cdot) \quad (4)$$

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

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

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

At this point, we find $\langle match \rangle \rightarrow \langle match \rangle \cdot ' | \langle mrule \rangle$, 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):

$$(\langle match \rangle \rightarrow \langle match \rangle \cdot ' | \langle mrule \rangle, \langle fvalbind \rangle \rightarrow \langle sfulbind \rangle \cdot) \quad (7)$$

$$(\langle match \rangle \rightarrow \langle match \rangle \cdot ' | \langle mrule \rangle, \langle fvalbind \rangle \rightarrow \langle fvalbind \rangle \cdot ' | \langle sfulbind \rangle) \quad (8)$$

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:

$$(\langle match \rangle \rightarrow \langle match \rangle ' | \cdot \langle mrule \rangle, \langle fvalbind \rangle \rightarrow \langle fvalbind \rangle ' | \cdot \langle sfulbind \rangle) \quad (9)$$

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:

$$(\langle mrule \rangle \rightarrow \cdot \langle pat \rangle \Rightarrow \langle exp \rangle, \langle fvalbind \rangle \rightarrow \langle fvalbind \rangle ' | \cdot \langle sivalbind \rangle) \quad (10)$$

$$(\langle mrule \rangle \rightarrow \cdot \langle pat \rangle \Rightarrow \langle exp \rangle, \langle sivalbind \rangle \rightarrow \cdot vid \langle atpats \rangle = \langle exp \rangle) \quad (11)$$

$$(\langle pat \rangle \rightarrow \cdot vid \langle atpat \rangle, \langle sivalbind \rangle \rightarrow \cdot vid \langle atpats \rangle = \langle exp \rangle) \quad (12)$$

$$(\langle pat \rangle \rightarrow vid \cdot \langle atpat \rangle, \langle sivalbind \rangle \rightarrow vid \cdot \langle atpats \rangle = \langle exp \rangle) \quad (13)$$

$$(\langle pat \rangle \rightarrow vid \cdot \langle atpat \rangle, \langle atpats \rangle \rightarrow \cdot \langle atpat \rangle) \quad (14)$$

$$(\langle pat \rangle \rightarrow vid \langle atpat \rangle \cdot, \langle atpats \rangle \rightarrow \langle atpat \rangle \cdot) \quad (15)$$

$$(\langle mrule \rangle \rightarrow \langle pat \rangle \cdot \Rightarrow \langle exp \rangle, \langle atpats \rangle \rightarrow \langle atpat \rangle \cdot) \quad (16)$$

$$(\langle mrule \rangle \rightarrow \langle pat \rangle \cdot \Rightarrow \langle exp \rangle, \langle sivalbind \rangle \rightarrow vid \langle atpats \rangle \cdot = \langle exp \rangle) \quad (17)$$

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 1.

3.2 Overview of the Algorithm

The example run detailed above relates couples of items. We call this relation the mutual accessibility relation \mathbf{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 [1, Theorem 5.9].

The algorithm thus constructs the image of the initial pair $(S' \rightarrow \cdot S \$, S' \rightarrow \cdot S \$)$ by the \mathbf{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 \mathbf{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 $A \rightarrow \alpha \cdot \beta$, or some nonterminal items of form $\cdot A$ or $A \cdot$. For instance, a nonterminal item would be used when computing the mutual accessibility of (2) and before reaching (5):

$$(\langle match \rangle \rightarrow \langle match \rangle \cdot ' | ' \langle mrule \rangle, \langle mrule \rangle \cdot). \quad (18)$$

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 [2] 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 [23], 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 $A \succ B$, if there is a rule $A \rightarrow \alpha B$; our order is then \succ^* . In a reduce/reduce conflict between reductions to A and B , we follow the reduction of A if $A \succ^* B$ or if both $A \succ^* B$ and $B \succ^* A$.

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 [33], 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 [28].

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

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

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(item₀)” 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:

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

$$S \rightarrow A \mid B_n a, A \rightarrow Aaa \mid a, B_1 \rightarrow aa, B_2 \rightarrow B_1B_1, \dots, B_n \rightarrow B_{n-1}B_{n-1}. \quad (\mathcal{G}_4^n)$$

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.

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

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

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

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

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

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

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. [28] in Table 2.

Table 2: Reported potential ambiguities in the RNA grammars from [28].

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

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 [18, 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 $\langle \text{typedef-name} \rangle \rightarrow \text{identifier}$. The modification reflects the fact that GLR parsers should not rely on the *lexer hack* for disambiguation.

Standard ML, extracted from the language definition [24, 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 [23], 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

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 Test	LR(0)			SLR(1)			LALR(1)	LR(1)		
	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, $\text{NU}(\text{item}_1)$ 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).

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 [25].

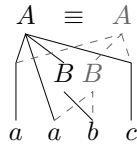


Figure 4: The shared parse forest for input $aabc$ with grammar \mathcal{G}_8 .

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) parser sizes by Purdom [27] 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 [21]—is not lost.

The theory behind noncanonical LALR parsing [31] 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

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

Our tool reports an ambiguity on the item pair $(B \rightarrow ab \bullet, B \rightarrow b \bullet)$, 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 Stan-

dard 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 author is highly grateful to Jacques Farré for his help in the preparation of this paper, to Sébastien Verel for granting him the access to a fast computer, and to the anonymous referees of LDTA'07 for their numerous helpful suggestions.

References

- [1] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling. Volume I: Parsing*. Series in Automatic Computation. Prentice Hall, 1972. ISBN 0-13-914556-7.
- [2] Alfred V. Aho, Stephen C. Johnson, and Jeffrey D. Ullman. Deterministic parsing of ambiguous grammars. *Communications of the ACM*, 18(8):441–452, 1975. ISSN 0001-0782. doi: 10.1145/360933.360969.
- [3] Manuel E. Bermudez and Karl M. Schimpf. Practical arbitrary lookahead LR parsing. *Journal of Computer and System Sciences*, 41(2):230–250, 1990. ISSN 0022-0000. doi: 10.1016/0022-0000(90)90037-L.
- [4] Sylvie Billot and Bernard Lang. The structure of shared forests in ambiguous parsing. In *ACL'89*, pages 143–151. ACL Press, 1989. URL <http://www.aclweb.org/anthology/P89-1018>.
- [5] Claus Brabrand, Robert Giegerich, and Anders Møller. Analyzing ambiguity of context-free grammars. In Miroslav Balík and Jan Holub, editors, *CIAA'07*, 2007. URL <http://www.brics.dk/~brabrand/grambiguity/>. To appear in *Lecture Notes in Computer Science*.
- [6] David G. Cantor. On the ambiguity problem of Backus systems. *Journal of the ACM*, 9(4):477–479, 1962. ISSN 0004-5411. doi: 10.1145/321138.321145.
- [7] Didier Caucal. Graphes canoniques de graphes algébriques. *RAIRO - Theoretical Informatics and Applications*, 24(4):339–352, 1990. URL <http://www.inria.fr/rrrt/rr-0872.html>.
- [8] Noam Chomsky and Marcel Paul Schützenberger. The algebraic theory of context-free languages. In P. Braffort and D. Hirshberg, editors, *Computer Programming and Formal Systems*, Studies in Logic, pages 118–161. North-Holland Publishing, 1963.
- [9] Karel Čulik and Rina Cohen. LR-Regular grammars—an extension of LR(k) grammars. *Journal of Computer and System Sciences*, 7:66–96, 1973. ISSN 0022-0000.

- [10] Charles Donnelly and Richard Stallman. *Bison version 2.1*, September 2005. URL <http://www.gnu.org/software/bison/manual/>.
- [11] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970. ISSN 0001-0782. doi: 10.1145/362007.362035.
- [12] José Fortes Gálvez, Sylvain Schmitz, and Jacques Farré. Shift-resolve parsing: Simple, linear time, unbounded lookahead. In Oscar H. Ibarra and Hsu-Chun Yen, editors, *CIAA'06*, volume 4094 of *Lecture Notes in Computer Science*, pages 253–264. Springer, 2006. ISBN 3-540-37213-X. doi: 10.1007/11812128_24.
- [13] Dick Grune and Criel J. H. Jacobs. *Parsing Techniques: A Practical Guide*. Ellis Horwood Limited, 1990. ISBN 0-13-651431-6. URL <http://www.cs.vu.nl/~dick/PTAPG.html>.
- [14] Harry B. Hunt III, Thomas G. Szymanski, and Jeffrey D. Ullman. 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, 1974.
- [15] Stephen C. Johnson. YACC — yet another compiler compiler. Computing science technical report 32, AT&T Bell Laboratories, Murray Hill, New Jersey, July 1975.
- [16] Adrian Johnstone, Elizabeth Scott, and Giorgios Economopoulos. Evaluating GLR parsing algorithms. *Science of Computer Programming*, 61(3): 228–244, 2006. ISSN 0167-6423. doi: 10.1016/j.scico.2006.04.004.
- [17] Stefan Kahrs. 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/>.
- [18] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, 1988. ISBN 0-13-110362-8.
- [19] Paul Klint and Eelco 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, pages 89–100. Università di Milano, 1994. URL <http://citeseer.ist.psu.edu/klint94using.html>.
- [20] Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering and Methodology*, 14(3):331–380, 2005. ISSN 1049-331X. doi: 10.1145/1072997.1073000.
- [21] Ralf Lämmel and Chris Verhoef. Semi-automatic grammar recovery. *Software: Practice & Experience*, 31:1395–1438, 2001. doi: 10.1002/spe.423.
- [22] Peter Lee. *Using the SML/NJ System*. Carnegie Mellon University, 1997. URL <http://www.cs.cmu.edu/~petel/smlguide/smlnj.htm>.

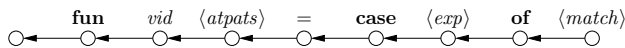
- [23] Scott McPeak and George C. Necula. Elkhound: A fast, practical GLR parser generator. In Evelyn Duesterwald, editor, *CC'04*, volume 2985 of *Lecture Notes in Computer Science*, pages 73–88. Springer, 2004. ISBN 3-540-21297-3. doi: 10.1007/b95956.
- [24] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The definition of Standard ML*. MIT Press, revised edition, 1997. ISBN 0-262-63181-4.
- [25] Terence J. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Programmers, 2007. ISBN 0-9787392-5-6.
- [26] David A. Poplawski. On LL-Regular grammars. *Journal of Computer and System Sciences*, 18(3):218–227, 1979. ISSN 0022-0000. doi: 10.1016/0022-0000(79)90031-X.
- [27] Paul Purdom. The size of LALR(1) parsers. *BIT Numerical Mathematics*, 14(3):326–337, 1974. ISSN 0006-3835. doi: 10.1007/BF01933232.
- [28] Janina Reeder, Peter Steffen, and Robert Giegerich. Effective ambiguity checking in biosequence analysis. *BMC Bioinformatics*, 6:153, 2005. ISSN 1471-2105. doi: 10.1186/1471-2105-6-153.
- [29] Andreas Rossberg. Defects in the revised definition of Standard ML. Technical report, Saarland University, Saarbrücken, Germany, July 2006. URL http://ps.uni-sb.de/Papers/paper_info.php?label=sml-defects.
- [30] Sylvain Schmitz. Conservative ambiguity detection in context-free grammars. In *ICALP'07*, 2007. URL <http://www.i3s.unice.fr/~mh/RR/2006/RR-06.30-S.SCHMITZ.pdf>. To appear in *Lecture Notes in Computer Science*.
- [31] Sylvain Schmitz. Noncanonical LALR(1) parsing. In Zhe Dang and Oscar H. Ibarra, editors, *DLT'06*, volume 4036 of *Lecture Notes in Computer Science*, pages 95–107. Springer, 2006. ISBN 3-540-35428-X. doi: 10.1007/11779148_10.
- [32] Elizabeth Scott and Adrian Johnstone. Right nulled GLR parsers. *ACM Transactions on Programming Languages and Systems*, 28(4):577–618, 2006. ISSN 0164-0925. doi: 10.1145/1146809.1146810.
- [33] Thomas G. Szymanski and John H. Williams. Noncanonical extensions of bottom-up parsing techniques. *SIAM Journal on Computing*, 5(2):231–250, 1976. ISSN 0097-5397. doi: 10.1137/0205019.
- [34] Masaru Tomita. *Efficient Parsing for Natural Language*. Kluwer Academic Publishers, 1986. ISBN 0-89838-202-5.
- [35] Mark van den Brand, Jeroen Scheerder, Jurgen J. Vinju, and Eelco Visser. Disambiguation filters for scannerless generalized LR parsers. In R. Nigel Horspool, editor, *CC'02*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158. Springer, 2002. ISBN 3-540-43369-4. URL <http://www.springerlink.com/content/03359k0cerupftfh/>.

A GLR Stack Activity

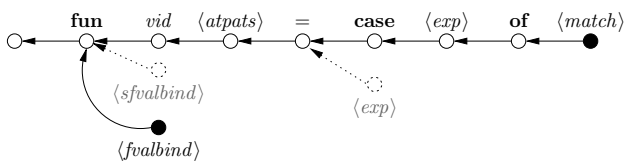
We present in this Appendix how a GLR parser manages to parse the input string of Example 1. We provide the step-by-step activity of the graph structured stack (GSS) of the parser.

The deterministic LALR(1) parser employs a regular stack where single states are pushed on shifts, or a number of states corresponding to the rule right-hand side length is popped on reductions, and a single state is pushed again on the subsequent goto on the corresponding rule left-hand side. The topmost stack state and the next token together give the next parsing action. In case of nondeterminism in the LALR(1) parser, a generalized parser “splits” the stack into different stacks, one for each possibility. The computational cost is contained by sharing the common parts of the different stacks. The interested reader can find a detailed description of this process, as well as a discussion on the complexity trade-offs involved, in an article by Johnstone *et al.* [16].

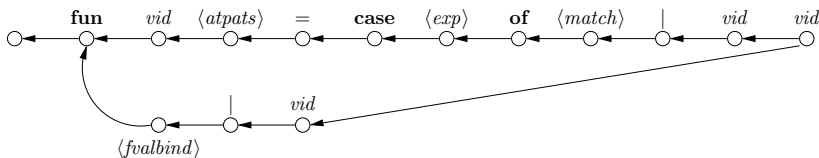
Let us start our explanations in the inconsistent state with a shift/reduce conflict: we made the goto to $\langle match \rangle$, and up to this point, we have a single stack.



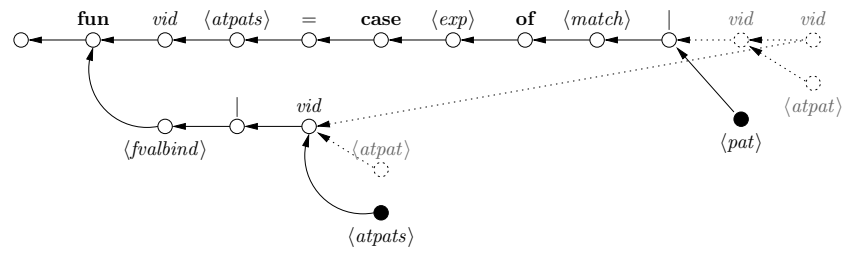
We need now to further perform all the possible reductions: the reduction in conflict creates a new GSS node for the goto on $\langle exp \rangle$. Owing to the possibility of a shift, we need to keep the nodes up to the $\langle match \rangle$ goto. The state in this node demands another reduction using rule $\langle s\text{fvalbind} \rangle \rightarrow vid \langle atpats \rangle = \langle exp \rangle$, and the previously created node is made inactive, while we create a new node for the goto of $\langle s\text{fvalbind} \rangle$. Again, the LALR(1) state in this new node demands yet another reduction, this time to $\langle f\text{valbind} \rangle$, and the node is made inactive, while we add a new node. We have thus two different stack frontiers, filled in black in the following figure.



Both these nodes accept to shift the next tokens “|”, vid , and vid . This last shift brings us to a stack with a single frontier node.



Performing all the possible reductions from this node separates again the two cases; in one, we end with a frontier node reached after a goto on $\langle pat \rangle$, whereas in the other, the frontier node is reached after a goto on $\langle atpats \rangle$.



Only the second of these two frontier nodes is compatible with the next shift of the equal sign “=”. The other stack is discarded. The parser has now a single stack, as if it had guessed the right parsing action in the conflict state.

