Finding Counterexamples from Parsing Conflicts

Chinawat Isradisaikul   Andrew C. Myers

Department of Computer Science
Cornell University
Ithaca, New York, United States
chinawat@cs.cornell.edu   andru@cs.cornell.edu

Abstract

Writing a parser remains remarkably painful. Automatic parser generators offer a powerful and systematic way to parse complex grammars, but debugging conflicts in grammars can be time-consuming even for experienced language designers. Better tools for diagnosing parsing conflicts will alleviate this difficulty. This paper proposes a practical algorithm that generates compact, helpful counterexamples for LALR grammars. For each parsing conflict in a grammar, a counterexample demonstrating the conflict is constructed. When the grammar in question is ambiguous, the algorithm usually generates a compact counterexample illustrating the ambiguity. This algorithm has been implemented as an extension to the CUP parser generator. The results from applying this implementation to a diverse collection of faulty grammars show that the algorithm is practical, effective, and suitable for inclusion in other LALR parser generators.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Debugging aids, Diagnostics; D.3.4 [*Programming Languages*]: Processors—Parsing

General Terms Languages

Keywords Context-free grammar; shift-reduce parser; ambiguous grammar; error diagnosis; lookahead-sensitive path; product parser

1. Introduction

An early triumph of programming language research was the development of parser generators, tools that in principle provide a concise, declarative way to solve the ubiquitous problem of parsing. Although LALR parser generators are powerful and have been available since the 1970s [15], they remain difficult to use, largely because of the challenges that arise when debugging grammars to eliminate shift/reduce and reduce/reduce conflicts.

Currently, debugging LALR grammars requires a solid understanding of the internal mechanism of LR parsers, a topic that is often but not always taught in undergraduate-level compiler courses. Even with this understanding, language designers can spend hours trying to understand how a grammar specification leads to the observed conflicts. The predictable result is that software developers tend to hand-code parsers even for tasks to which parser generators are ideally suited. Hand-coded parsers lead to code that is more verbose, less maintainable, and more likely to create security vulnerabilities when applied to untrusted input [9, 10]. Developers may also compromise the language syntax in order to simplify parsing, or avoid domain-specific languages and data formats altogether.

Despite the intrinsic limitations of LL grammars, top-down parser generators such as ANTLR [20, 21] are popular perhaps because their error messages are less inscrutable. It is surprising that there does not seem to have been much effort to improve debugging of conflicts in the more powerful LR grammars. Generalized LR parsers [28] enable programmers to resolve ambiguities programmatically, but even with GLR parsers, ambiguities could be better understood and avoided. Moving towards this goal, Elkhound [17] reports parse trees but only when the user provides a counterexample illustrating the ambiguity. Some LALR parser generators attempt to report counterexamples [18, 30] but can produce misleading counterexamples because their algorithms fail to take lookahead symbols into account. Existing tools that do construct correct counterexamples [8, 27] use a brute-force search over all possible grammar derivations. This approach is impractically slow and does not help diagnose unambiguous grammars that are not LALR.

We improve the standard error messages produced by LR parser generators by giving short, illustrative counterexamples that identify ambiguities in a grammar and show how conflicts arise. For ambiguous grammars, we seek a *unifying counterexample*, a string of symbols having two distinct parses. Determining whether a context-free grammar is ambiguous is undecidable, however [13], so the search for a unifying counterexample cannot be guaranteed to terminate. When a unifying counterexample cannot be found in a reasonable time, we seek a *nonunifying counterexample*, a pair of derivable strings of symbols sharing a common prefix up to the point of conflict. Nonunifying counterexamples are also reported when the grammar is determined to be unambiguous but not LALR.

Our main contribution is a search algorithm that exploits the LR state machine to construct both kinds of counterexamples. Our evaluation shows that the algorithm is efficient in practice. A key insight behind this efficiency is to expand the search frontier from the *conflict state* instead of the *start state*.

The remainder of the paper is organized as follows. Section 2 reviews how LR parser generators work and how parsing conflicts arise. Section 3 outlines properties of good counterexamples. Sections 4 and 5 explore algorithms for finding nonunifying and unifying counterexamples. An implementation of the algorithm that works well in practice is discussed in Section 6. Using various grammars, we evaluate the effectiveness, efficiency, and scalability of the algorithm in Section 7. Section 8 discusses related work, and Section 9 concludes.

PLDI'15, June 13–17, 2015, Portland, OR, USA

http://dx.doi.org/10.1145/2737924.2737961

2. Background

We assume the reader has some familiarity with LR grammars and parser generators. This section briefly reviews the construction of an LR parser and shows how LR parsing conflicts arise.

2.1 Parser State Machine

Starting from a context-free grammar like the one in Figure 1, the first step in generating an LR(1) parser is the construction of a parser state machine for the grammar. Each state contains a collection of *transitions* on symbols and a collection of *production items*. Each transition is either a *shift action* on a terminal symbol or a *goto* on a nonterminal symbol. A production item (abbreviated *item*) tracks the progress on completing the right-hand side of a production. Each item contains a *dot* (•) indicating transitions that have already been made on symbols within the production, and a *lookahead set* of possible terminals that can follow the production.

The items within a state include those that result from taking transitions from a predecessor state, and also those generated by the *closure* of all the productions of any nonterminal that follows a dot. For the start state, the items include those of productions of the start symbol and their closure[1]. Figure 2 shows a partial parser state diagram for the example grammar.

A parser maintains a stack of symbols during parsing. A shift action on the next input symbol t is performed when a transition on t is available in the current state; t is pushed onto the stack. A reduction is performed when the current state contains an item of the form $A \rightarrow X_1\ X_2\ \cdots\ X_m\ \bullet$, whose lookahead set contains the next input symbol; m symbols are popped from the stack, and the nonterminal A is then pushed onto the stack. If neither a shift action nor a reduction is possible, a syntax error occurs.

2.2 Shift/Reduce Conflicts

For LR(1) grammars, actions on parser state machines are deterministic: given a state and the next input symbol, either a shift action or a reduction is executed. Otherwise, a state may contain a pair of items that create a *shift/reduce conflict* on a terminal symbol t:

- a *shift item* of the form $A \rightarrow X_1\ X_2\ \cdots\ X_k\ \bullet\ X_{k+1}\ \cdots\ X_m$, where $X_{k+1} = t$ for some $k \geq 0$ and $m \geq 1$, and
- a *reduce item* of the form $B \rightarrow Y_1\ Y_2\ \cdots\ Y_n\ \bullet$, whose lookahead set contains t.

The example grammar has a shift/reduce conflict, because the two items in State 10 match the criteria above on lookahead `else`. This is the classic *dangling else* problem. The grammar is ambiguous because there are two ways to parse this statement:

`if` *expr* `then` `if` *expr* `then` *stmt* • `else` *stmt*

Even though the grammar is ambiguous, not every conflict must contribute to the ambiguity. Conflicts may also occur even if the grammar is not ambiguous. For instance, the grammar in Figure 3 has a shift/reduce conflict between shift action $Y \rightarrow$ `a` • `a b` and reduction $X \rightarrow$ `a` • under symbol `a`. Nevertheless, this grammar is LR(2) and hence unambiguous.

2.3 Reduce/Reduce Conflicts

A state may also contain a pair of distinct reduce items that create a *reduce/reduce conflict* because their lookahead sets intersect:

- $A \rightarrow X_1\ X_2\ \cdots\ X_m\ \bullet$, with lookahead set $\mathcal{L}_A$, and
- $B \rightarrow Y_1\ Y_2\ \cdots\ Y_n\ \bullet$, with lookahead set $\mathcal{L}_B$ such that $\mathcal{L}_A \cap \mathcal{L}_B \neq \emptyset$.

[1] The actual parser construction adds a special start symbol and production, which are omitted in this section.

stmt → `if` *expr* `then` *stmt* `else` *stmt*
  | `if` *expr* `then` *stmt*
  | *expr* `?` *stmt* *stmt*
  | `arr [` *expr* `] :=` *expr*
expr → *num* | *expr* `+` *expr*
num → ⟨*digit*⟩ | *num* ⟨*digit*⟩

Figure 1. An ambiguous CFG

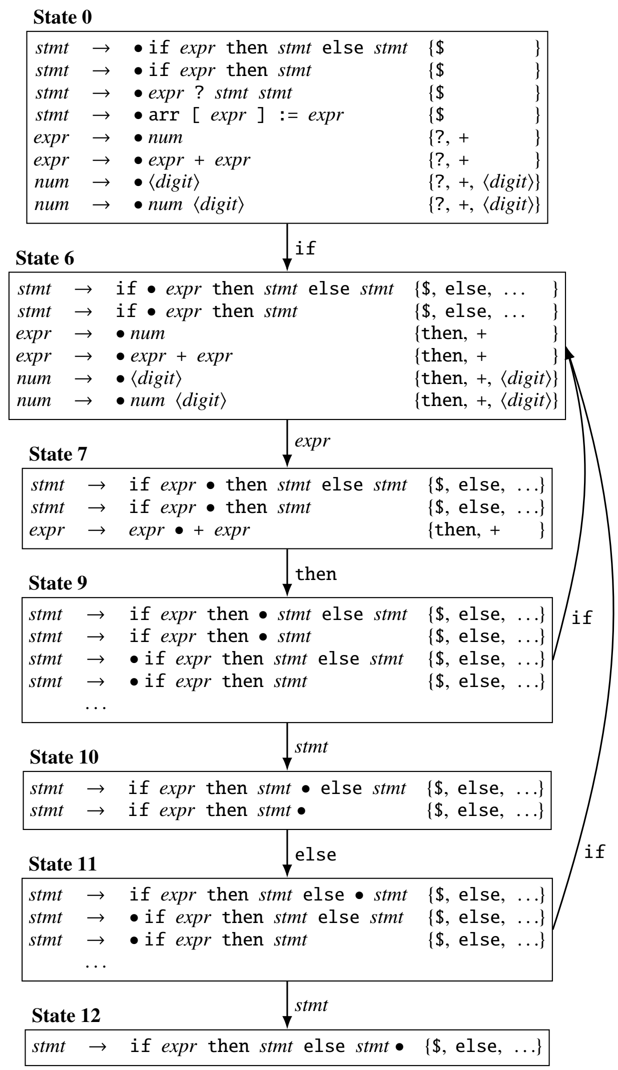

Figure 2. Selected parser states for the ambiguous CFG. Symbol $ indicates the end of input.

$S \rightarrow T \mid S\ T$
$T \rightarrow X \mid Y$
$X \rightarrow$ `a`
$Y \rightarrow$ `a a b`

Figure 3. An unambiguous CFG with a shift/reduce conflict

2.4 Precedence

To simplify grammar writing, precedence and associativity declarations can be used to resolve shift/reduce conflicts. For example, the grammar in Figure 1 has a shift/reduce conflict between shift item *expr* → *expr* • + *expr* and reduce item *expr* → *expr* + *expr* • under symbol +, exhibited by the counterexample *expr* + *expr* • + *expr*. Declaring operator + left-associative causes the reduction to win.

3. Counterexamples

The familiar shift/reduce conflicts in the previous section are easily diagnosed by experienced programming language designers. In general, the source of conflicts can be more difficult to find.

3.1 A Challenging Conflict

The example grammar in Figure 1 has another shift/reduce conflict in State 1 (not shown in Figure 2) between

- shift item *num* → *num* • ⟨*digit*⟩, and
- reduce item *expr* → *num* •

under terminal symbol ⟨*digit*⟩. It is probably not immediately clear why this conflict is possible, let alone what counterexample explains the conflict. In fact, an experienced language designer in our research group spent some time to discover this counterexample by hand[2]:

expr ? `arr [` *expr* `] :=` *num* • ⟨*digit*⟩ ⟨*digit*⟩ ? *stmt stmt*.

This statement can be derived in two ways from the production *stmt* → *expr* ? $stmt_1$ $stmt_2$. First, we can use the reduce item:

- $stmt_1 \to^*$ `arr [` *expr* `] :=` *num*
- $stmt_2 \to^*$ ⟨*digit*⟩ ⟨*digit*⟩ ? *stmt stmt*

Second, we use the shift item:

- $stmt_1 \to^*$ `arr [` *expr* `] :=` *num* ⟨*digit*⟩
- $stmt_2 \to^*$ ⟨*digit*⟩ ? *stmt stmt*

This counterexample, along with its two possible derivations, immediately clarifies why there is an ambiguity and helps guide the designer towards a better syntax, e.g., demarcating $stmt_1$ and $stmt_2$. Our goal is to generate such useful counterexamples automatically.

3.2 Properties of Good Counterexamples

Useful counterexamples should be concise and simple enough to help the user understand parsing conflicts effortlessly. This principle leads us to prefer counterexamples that are no more concrete than necessary. Although a sequence of terminal symbols that takes the parser from the start state to the conflict state through a series of shift actions and reductions might be considered a good counterexample, some of these terminals may distract the user from diagnosing the real conflict. For example, the following input takes the parser from State 0 to State 10 in Figure 2:

```
if 2 + 5 then arr[4] := 7
```

But the expression `2 + 5` could be replaced with any other expression, and the statement `arr[4] := 7` with any other statement. Good counterexamples should use nonterminal symbols whenever the corresponding terminals are not germane to the conflict.

As discussed earlier, LALR parsing conflicts may or may not be associated with an ambiguity in a grammar. Counterexamples should be tailored to each kind of conflict.

Unifying counterexamples When possible, we prefer a *unifying counterexample*: a string of symbols (terminals or nonterminals) having two distinct parses. A unifying counterexample is a clear demonstration that a grammar is ambiguous. The counterexample given for the challenging conflict above is unifying, for example.

Good unifying counterexamples should be derivations of the innermost nonterminal that causes the ambiguity, rather than full sentential forms, to avoid distracting the user with extraneous symbols. For instance, a good unifying counterexample for the conflict in Section 2.4 is *expr* + *expr* • + *expr*, a derivation of the nonterminal *expr*, rather than *expr* + *expr* • + *expr* ? *stmt stmt*, a derivation of the start symbol.

Nonunifying counterexamples When a unifying counterexample cannot be found, there is still value in a *nonunifying counterexample*: a pair of derivable strings of symbols sharing a common prefix up to the point of conflict but diverging thereafter. The common prefix shows that the conflict state is reachable by deriving some nonterminal in the grammar. For example, the following is a possible nonunifying counterexample for the challenging conflict, where each bracket groups symbols derived from the nonterminal *stmt*:

stmt *stmt*
expr ? `arr [` *expr* `] :=` *num* • ⟨*digit*⟩ ? *stmt stmt*
expr ? `arr [` *expr* `] :=` *num* • ⟨*digit*⟩ *stmt*
stmt

Like unifying counterexamples, good nonunifying counterexamples should be derivations of the innermost nonterminal that can reach the conflict state.

Nonunifying counterexamples are produced for unambiguous grammars that are not LALR. Additionally, since ambiguity detection is undecidable, no algorithm can always provide a unifying counterexample for every ambiguous grammar. In this case, providing a nonunifying counterexample is a suitable fallback strategy.

4. Constructing Nonunifying Counterexamples

We first describe an algorithm for constructing nonunifying counterexamples that are derivations of the start symbol. The algorithm for constructing unifying counterexamples, described in Section 5, identifies the innermost nonterminal that can reach the conflict state.

Recall that certain terminals in a counterexample can be replaced with a nonterminal without invalidating the counterexample. Such terminals must have been part of a reduction. Therefore, a counterexample can be constructed from a walk along transition edges in the parser state diagram from the start state to the conflict state. Not all such walks constitute valid counterexamples, however. In particular, the shortest path is often invalid. For example, the input `if` *expr* `then` *stmt* forms the shortest path to State 10 in Figure 2, but a conflict does not arise at this point. If the next input symbol is `else`, the shift action is performed; if the end of input is reached, the reduction occurs. For a counterexample to be valid, the lookahead sets of parser items must be considered as well.

Instead of finding the shortest path in the state diagram, our algorithm finds the shortest *lookahead-sensitive path* to the conflict state. Intuitively, a lookahead-sensitive path is a sequence of transitions and production steps[3] between parser states that also keeps track of terminals that actually can follow the current production.

To define lookahead-sensitive paths formally, we first define a *lookahead-sensitive graph*, an extension of an LR(1) parser state diagram in which production steps are represented explicitly. Each vertex is a triple (*s*, *itm*, *L*), where *s* is a state number, *itm* is an item within *s*, and *L* is a *precise lookahead set*. The edges in this graph are defined as follows:

[2] This conflict was originally part of a larger grammar.

[3] A production step picks a specific production of a nonterminal to work on. These steps are implicit in an LR closure.

- *transition* (Figure 4(a)): For every transition in the parser, there is an edge between appropriate parser states and items, preserving the precise lookahead set between the vertices.
- *production step* (Figure 4(b)): For every item whose symbol after • is a nonterminal, there is an edge from this item to each item associated with a production of the nonterminal within the same state. The precise lookahead set changes to the set of terminals that actually can follow the production. Denoted $follow_L(itm)$, the *precise follow set* for *itm* in *L*'s context is defined as follows:
  - $follow_L(A \to X_1 \cdots X_{n-1} \bullet X_n) \triangleq L$.
  - $follow_L(A \to X_1 \cdots X_k \bullet X_{k+1} X_{k+2} \cdots X_n) \triangleq \{X_{k+2}\}$ if X_{k+2} is a terminal.
  - $follow_L(A \to X_1 \cdots X_k \bullet X_{k+1} X_{k+2} \cdots X_n) \triangleq \mathrm{FIRST}(X_{k+2})$ if X_{k+2} is a nonnullable nonterminal, i.e., a nonterminal that cannot derive ε. FIRST(N) is the set of terminals that can begin a derivation of N.
  - $follow_L(A \to X_1 \cdots X_k \bullet X_{k+1} X_{k+2} \cdots X_n) \triangleq \mathrm{FIRST}(X_{k+2}) \cup follow_L(A \to X_1 \cdots X_{k+1} \bullet X_{k+2} \cdots X_n)$ if X_{k+2} is a nullable nonterminal.

A shortest lookahead-sensitive path is a shortest path in the lookahead-sensitive graph. To construct a counterexample, the algorithm starts by finding a shortest lookahead-sensitive path from $(s_0, itm_0, \{\$\})$ to (s', itm', L'), where s_0 is the start state, itm_0 is the start item, s' is the conflict state, itm' is the conflict reduce item[4], and L' contains the conflict symbol. The symbols associated with the transition edges form the first part of a counterexample. For instance, Figure 5(a) shows the shortest lookahead-sensitive path to the conflict reduce item in State 10 of Figure 2. This path gives the prefix of the expected counterexample:

`if` *expr* `then` `if` *expr* `then` *stmt* •

To avoid excessive memory consumption, our algorithm does not construct the lookahead-sensitive graph in its entirety. Rather, vertices and edges are created as they are discovered.

The partial counterexample constructed so far takes the parser to the conflict state. Counterexamples can be constructed in full by completing all the productions made on the shortest lookahead-sensitive path. Since the conflict terminal is a vital part of counterexamples, this terminal must immediately follow •. In the example above, a production step was made in State 9 (step 5 in Figure 5(a)), where the next symbol to be parsed is the conflict terminal `else`. In this case, the production can be completed immediately, yielding the counterexample

`if` *expr* `then` `if` *expr* `then` *stmt* • `else` *stmt*

On the other hand, if the symbol immediately after • is a nonterminal, a derivation of that nonterminal beginning with the conflict symbol is required. Consider once again the conflict between *expr* → *num* • and *num* → *num* • ⟨*digit*⟩ under lookahead ⟨*digit*⟩. The shortest lookahead-sensitive path to the reduce item gives the prefix

expr `?` `arr` `[` *expr* `]` `:=` *num* •

but the next symbol to be parsed is *stmt*. In this case, we must find a statement that starts with a digit, e.g., ⟨*digit*⟩ `?` *stmt stmt*, yielding the counterexample

expr `?` `arr` `[` *expr* `]` `:=` *num* • ⟨*digit*⟩ `?` *stmt* *stmt*

The shortest lookahead-sensitive path only reveals a counterexample that uses the conflict reduce item. A counterexample that uses the conflict shift item can be discovered by exploring the states on the shortest lookahead-sensitive path as follows. Since transitions on input symbols must be between the same states for both counterexamples, the only difference is that the derivation using the shift item must take different production steps within such states. To determine these production steps, our algorithm starts at the conflict shift item and explores backward all the productions that may be used in the states along the shortest lookahead-sentitive path, until an item used in the derivation using the reduce item is found. For example, Figure 5(b) shows the reverse sequence leading to the shift item of the dangling-else conflict. Observe that this sequence follows the same states as in the shortest lookahead-sensitive path when making transitions, namely, [0, 6, 7, 9, 6, 7, 9, 10]. Even though this sequence yields the same counterexample as above, the derivation is different.

[4] The conflict shift item cannot be used because we have no information about the lookahead symbol after the completion of the shift item.

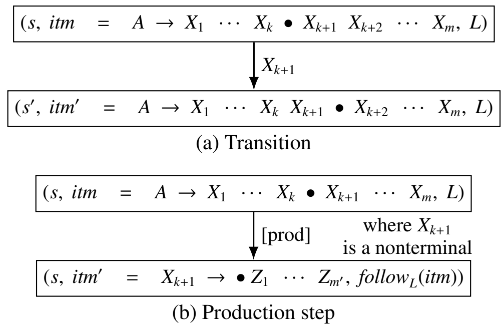

Figure 4. Edges of a lookahead-sensitive graph

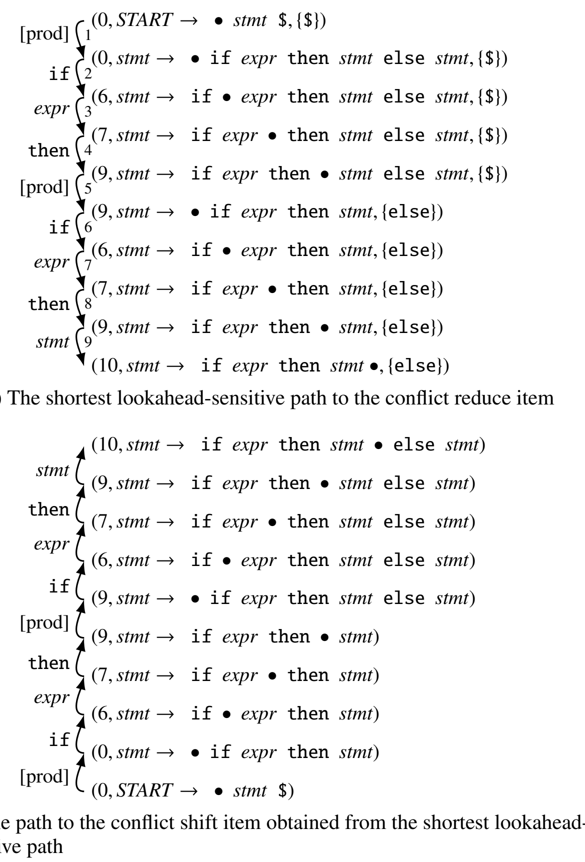

Figure 5. Paths to the dangling-else shift/reduce conflict

5. Constructing Unifying Counterexamples

The algorithm for constructing nonunifying counterexamples does not guarantee that the resulting counterexamples will be ambiguous if the grammar is. To aid the diagnosis of an ambiguity, the symbols beyond the conflict terminal must agree so that the entire string can be parsed in two different ways using the two conflict items. Since these conflict items force parser actions to diverge after the conflict state, the algorithm must keep track of both parses simultaneously.

5.1 Product Parser

The idea of keeping track of two parses is similar to the intuition behind generalized LR parsing [28], but instead of running the parser on actual inputs, our approach simulates possible parser actions and constructs counterexamples at parser generation time. Two copies of the parser are simulated in parallel. One copy is required to take the reduction and the other to take the shift action of the conflict. If both copies accept an input at the same time, then this input is a unifying counterexample. A distinct sequence of parser actions taken by each copy describes one possible derivation of the counterexample.

More formally, the parallel simulation can be represented by actions on a *product parser*, whose states are the Cartesian product of the original parser items. Two stacks are used, one for each original parser. This construction resembles that of a direct product of nondeterministic pushdown automata [1], but here the states are more tightly coupled to make parser actions easier to understand. Like a lookahead-sensitive graph, a product parser represents production steps explicitly. Actions on a product parser are defined as follows:

- *transition*: If both items in a state of the product parser have a transition on symbol Z in the original parser, there is a corresponding transition on Z in the product parser (Figure 6(a)). When this transition is taken, Z is pushed onto both stacks.
- *production step*: If an item in a state of the product parser has a nonterminal after •, there is a production step on this nonterminal in the product parser (Figure 6(b)). Both stacks remain unchanged when a production step is taken.
- *reduction*: If an item in a state of the product parser is a reduce item, a reduction can be performed on the original parser associated with this item, respecting its loookahead set, while leaving the other item and its associated stack unchanged.

For a conflict between items itm_1 and itm_2, a string accepted by the product parser that also takes the parser through state (itm_1, itm_2) is a unifying counterexample for the conflict. The remainder of this section describes an algorithm that efficiently simulates the product parser without exploring irrelevant states.

5.2 Outward Search from the Conflict State

The strategy of using shortest lookahead-sensitive paths to avoid exploring too many states does not work in general, because symbols required after • might be incompatible with the productions already made on these paths. For example, the grammar in Figure 7 has two shift/reduce conflicts in the same state, between reduce item $A \to$ `a` • and two shift items $B \to$ `a • b c` and $B \to$ `a • b d` under symbol `b`. The shortest lookahead-sensitive path gives prefix `n a •`, which is compatible with a unifying counterexample for the first shift item, namely, `n a • b c`. Still, no unifying counterexamples that use the second shift item can begin with `n a •`. An extra `n` is required before •, as in `n n a • b d c`. This example suggests that deciding on the productions to use before reaching the conflict state is inimical to discovering unifying counterexamples.

To avoid making such decisions, our search algorithm starts from the conflict state and completes derivations outward. Each search state, denoted *configuration* henceforth, contains two pairs of (1) a sequence of items representing valid transitions and production steps in the original parser, and (2) partial derivations associated with transitions between items, as shown in Figure 8(a). The initial configuration contains (1) singleton sequences of the conflict items and (2) empty derivations, as shown in Figure 8(b). As partial derivations are expanded, configurations progress through four stages, which are illustrated in Figure 9 for the challenging conflict from Section 3.1. The four stages are as follows:

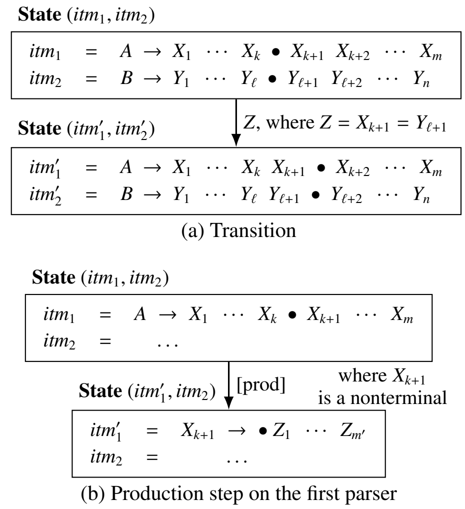

Figure 6. Components of the state machine for a product parser

$S \to N$ | N `c`
$N \to$ `n` N `d` | `n` N `c` | `n` A `b` | `n` B
$A \to$ `a`
$B \to$ `a b c` | `a b d`

Figure 7. An ambiguous grammar where the shortest lookahead-sensitive path does not yield a unifying counterexample

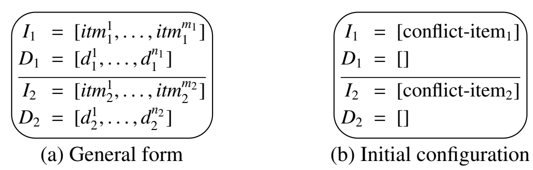

Figure 8. Configurations. Each *itm* is an item in the original parser, and each d is a derivation associated with a transition between items.

1. Completion of the conflict reduce item: the counterexample must contain derivations of all symbols in the reduce item. All reduce/reduce conflict items are completed in this stage.
2. Completion of the conflict shift item: the counterexample must also contain derivations of all symbols in the shift item. This stage is not needed for reduce/reduce conflicts.
3. Discovery of the unifying nonterminal: the counterexample must be a derivation of a single nonterminal. This stage also identifies the innermost nonterminal for nonunifying counterexamples.
4. Completion of the entire unifying counterexample: the final counterexample must complete all the unfinished productions. This stage attempts to find the remaining symbols so that the derivation of the nonterminal found in Stage 3 can be completed at the same time on both copies of the parser.

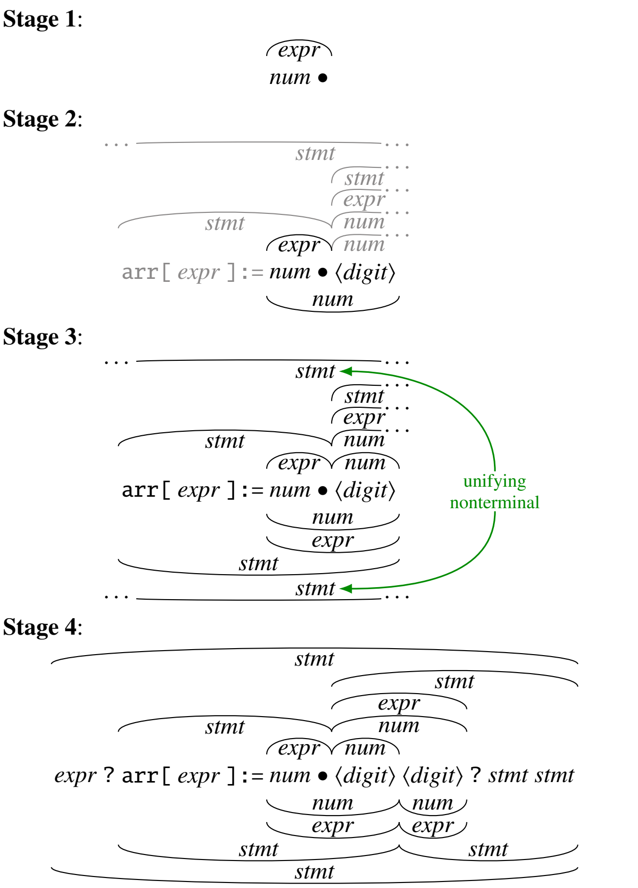

Figure 9. Counterexamples and derivations associated with configurations after finishing each stage for the challenging conflict. The derivation above each counterexample uses the reduce item; the one below uses the shift item. The gray portion of the configuration is not required for completing the stage.

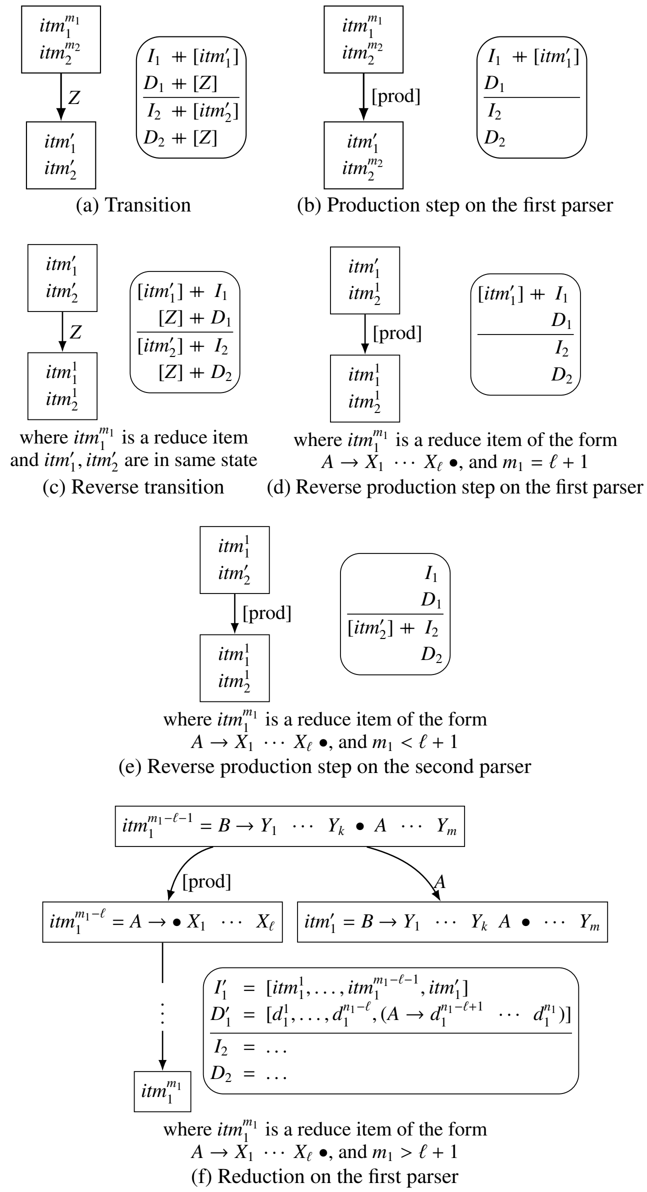

Figure 10. Successor configurations. Each kind of edge in the product parser corresponds to a particular successor configuration. Operator ++ denotes list concatenation.

5.3 Successor Configurations

We now present a strategy for computing successor configurations. Figure 10 pictures some of the possible successor configurations that can be reached from the configuration shown in Figure 8(a) via various actions in the product parser:

- *transition* (Figure 10(a)): If the product parser has a transition on symbol Z from the last item in the current configuration, append the current configuration with appropriate items and symbols.
- *production step on the first parser* (Figure 10(b)): If the product parser has a production step on the first parser from the last item in the current configuration, append the item resulting from taking the production step (itm'_1) to the sequence of items for the first parser (I_1). A production step on the second parser is symmetric.
- *preparation of the first parser for a reduction*: If the last item for the first parser is a reduce item, but there are not enough items to simulate a reduction moving forward, then more items must be prepended to the configuration. That is, we must work backward to ready the reduction. Preparing the second parser for a reduction is symmetric. Successor configurations depend on the first item in the current configuration:
  - *reverse transition* (Figure 10(c)): If the product parser has a transition on symbol Z to the first item in the current configuration, prepend the current configuration with appropriate items and symbols. The prepended items must belong to the same state in the original parser. Additionally, the lookahead set of the item prepended to the first parser (itm'_1) must contain the conflict symbol if the current configuration is yet to complete Stage 1.
  - *reverse production step on the first parser* (Figure 10(d)): If the product parser has a production step on the first parser to the first item in the current configuration, prepend the item prior to taking the production step (itm'_1) to I_1.
  - *reverse production step on the second parser* (Figure 10(e)): Occasionally, the second parser will require a reverse production step so that further reverse transitions can be made. In this case, prepend the item prior to taking the production step (itm'_2) to the sequence of items for the second parser (I_2).

◆ *reduction on the first parser* (Figure 10(f)): If the last item for the first parser is a reduce item of the form $A \to X_1 \cdots X_\ell \bullet$, and the configuration has enough items, then the first parser is ready for a reduction. A successor configuration is obtained by (1) removing the last $\ell + 1$ items that are part of the reduction from I_1, which simulates popping the parser stack, (2) appending the result of taking the goto on A (itm'_1) to I_1, and (3) rearranging the partial derivations (D_1) to complete the derivation for A. The second parser remains unchanged throughout the reduction. A reduction on the second parser is symmetric.

5.4 Completing the Search

The search algorithm computes successor configurations until it encounters a configuration C_f that has completed Stages 1 and 2, where both sequences of items in C_f are of the form

$$[? \to \cdots \bullet A \cdots, ? \to \cdots A \bullet \cdots]$$

for some nonterminal A. The partial derivations associated with these sequences, which must be of the form $[A \to \cdots]$, show that nonterminal A is ambiguous. The unifying counterexample is the sequence of the leaf symbols within these derivations.

Several observations can be made about the algorithm. First, the algorithm maintains an invariant that the head of both sequences of items in any configuration belong to the same parser state, as the sequence of states prior to the conflict must be identical for different derivations of the unifying counterexample. Second, a configuration generates multiple successor configurations only when a production step (forward or backward) or a reverse transition is taken. Therefore, the branching factor of the search is proportional to the ratio of the number of these actions to the number of items in the parser.

The third observation is that a production step may be taken repeatedly within the same state, such as one for items of the form $A \to \bullet A \cdots$. To avoid infinite expansions on one configuration without making progress on others, the search algorithm must postpone such an expansion until other configurations have been considered. The algorithm imposes different costs on different kinds of actions and considers configurations in order of increasing cost.

Finally, the algorithm is guaranteed to find a unifying counterexample for every ambiguous grammar, but the search will not terminate when infinite expansions are possible on unambiguous grammars. In other words, this semi-decision procedure for determining ambiguity is sound and complete. Since a naive implementation of this algorithm is too slow for practical use, the next section discusses techniques that speed up the search but still maintain the quality of counterexamples.

6. Implementation

Our counterexample finder has been implemented in Java as a module extending the CUP LALR parser generator [14] version 0.11b 20150326[5]. The module contains 1478 non-comment, nonempty lines of code. Figure 11 shows an error message reported by our implementation for the shift/reduce conflict in Section 2.4. One interesting design choice was the tradeoff between finding unifying counterexamples when they exist, and avoiding long, possibly fruitless searches when a nonunifying counterexample might suffice.

Data structures The search algorithm requires many queries on possible parser actions, but parser generators usually do not provide an infrastructure for fast lookups. In particular, reverse transitions and production steps are not represented directly. Before working on the first conflict within a grammar, our implementation generates several lookup tables for these actions.

[5] Available at `https://github.com/polyglot-compiler/polyglot/tree/master/tools/java_cup`.

```
Warning : *** Shift/Reduce conflict found in state #13
  between reduction on expr ::= expr PLUS expr •
  and shift on         expr ::= expr • PLUS expr
  under symbol PLUS
  Ambiguity detected for nonterminal expr
  Example: expr PLUS expr • PLUS expr
  Derivation using reduction:
    expr ::= [expr ::= [expr PLUS expr •] PLUS expr]
  Derivation using shift:
    expr ::= [expr PLUS expr ::= [expr • PLUS expr]]
```

Figure 11. A sample error message reported by the implementation. The first four lines are original to CUP.

Finding shortest lookahead-sensitive path Blindly searching for the shortest path from the start state might explore all parser states. As an optimization, only states that can reach the reduce conflict item need be considered. These states can be found quickly using the lookup tables for reverse transitions and reverse production steps.

Constructing unifying counterexamples The main search algorithm is also unguided. As a tradeoff, the algorithm only considers states on the shortest lookahead-sensitive path when making reverse transitions. This restriction makes the algorithm incomplete, causing it to miss unifying counterexamples that use parser states outside the shortest path. Nevertheless, a counterexample that follows the shortest lookahead-sensitive path will take the parser to the conflict state as quickly as possible. These compact counterexamples seem as helpful as unifying ones, so our tool does report them. The option `-extendedsearch` can be used to force a full search.

Constructing nonunifying counterexamples The search for unifying counterexamples may fail in two cases: first, when eligible configurations run out; second, when a production step in an unambiguous grammar is taken repeatedly, resulting in nontermination. Therefore, our implementation imposes a 5-second time limit on the main search algorithm. When the search fails, a nonunifying counterexample is constructed and reported instead.

The implementation also imposes a 2-minute time limit on the cumulative running time of the unifying counterexample finder. After two minutes, at least 20 conflicts must have been accompanied with counterexamples, so the user is likely to prefer resolving them first. Our tool seeks only nonunifying counterexamples thereafter.

Exploiting precedence Precedence and associativity are not part of the parser state diagram, and hence are not part of the generated lookup tables. Therefore, our implementation inspects precedence declared with relevant terminals and productions during the search.

7. Evaluation

Our evaluation aims to answer three questions:

◆ Is our implementation effective on different kinds of grammars?

◆ Is our implementation efficient compared to existing ambiguity detection tools?

◆ Does our implementation scale to reasonably large grammars?

7.1 Grammar Examples

We have evaluated our implementation on a variety of grammars. For each grammar, Table 1 lists the complexity (the numbers of nonterminals and productions, and the number of states in the parser state machine) and the number of conflicts. The grammars are partitioned into the following categories:

Our grammars All grammars shown in this paper are evaluated. Other grammars that motivated the development of our tool, and a

Table 1. Evaluation results. T/L indicates 5-second time limit exceeded on all conflicts. Times in parentheses indicate running time for the state-of-the-art ambiguity detector [2, 5].

Grammar	# nonterms	# prods	# states	# conflicts	Amb?	# unif	# nonunif	# time out	Time (seconds) Total	Time (seconds) Average
figure1	3	9	24	3	✔	3	0	0	0.072	0.024
figure3	4	7	10	1	✗	0	1	0	0.010	0.010
figure7	4	10	16	2	✔	2	0	0	0.016	0.008
ambfailed01	6	10	17	1	✔	0	1	0	0.010	0.010
abcd	5	11	22	3	✔	3	0	0	0.024	0.008
simp2	10	41	70	1	✔	1	0	0	0.548	0.548
xi	16	41	82	6	✔	6	0	0	0.155	0.026
eqn	14	67	133	1	✔	1	0	0	0.169	0.169
java-ext1	185	445	767	2	✗	0	0	2	T/L	T/L
java-ext2	234	599	1255	1	✗	0	0	1	T/L	T/L
stackexc01	2	7	13	3	✔	3	0	0	0.023	0.008
stackexc02	6	11	15	1	✗	0	1	0	0.008	0.008
stackovf01	2	5	9	1	✗	0	1	0	0.009	0.009
stackovf02	2	5	9	4	✔	4	0	0	0.043	0.011
stackovf03	2	6	10	1	✔	1	0	0	0.017	0.017
stackovf04	5	9	13	1	✗	0	1	0	0.009	0.009
stackovf05	5	10	14	1	✔	1	0	0	0.010	0.010
stackovf06	6	10	15	2	✗	0	2	0	0.012	0.006
stackovf07	7	12	17	3	✔	3	0	0	0.028	0.009
stackovf08	3	13	21	8	✗	0	8	0	0.025	0.003
stackovf09	6	12	27	1	✗	0	1	0	0.017	0.017
stackovf10	9	20	53	19	✔	19	0	0	0.140	0.007
SQL.1	8	23	46	1	✔	1	0	0	0.024	0.024 (1.8s)
SQL.2	29	81	151	1	✔	1	0	0	0.060	0.060 (0.1s)
SQL.3	29	81	149	1	✔	1	0	0	0.024	0.024 (0.1s)
SQL.4	29	81	151	1	✔	1	0	0	0.031	0.031 (0.0s)
SQL.5	29	81	151	1	✔	1	0	0	0.030	0.030 (0.4s)
Pascal.1	79	177	323	3	✔	2	0	1	0.196	0.098 (0.3s)
Pascal.2	79	177	324	5	✔	5	0	0	0.296	0.059 (0.1s)
Pascal.3	79	177	321	1	✔	1	0	0	0.070	0.070 (1.2s)
Pascal.4	79	177	322	1	✔	1	0	0	0.081	0.081 (0.3s)
Pascal.5	79	177	322	1	✔	1	0	0	0.113	0.113 (0.3s)
C.1	64	214	369	1	✔	1	0	0	0.327	0.327 (1.3s)
C.2	64	214	368	1	✔	1	0	0	0.219	0.219 (1.11h)
C.3	64	214	368	4	✔	4	0	0	1.015	0.254 (0.5s)
C.4	64	214	369	1	✔	0	0	1	T/L	T/L (1.3s)
C.5	64	214	370	1	✔	1	0	0	0.212	0.212 (4.9s)
Java.1	152	351	607	1	✔	1	0	0	0.569	0.569 (32.4s)
Java.2	152	351	606	1133	✔	141	0	9 (983)	35.384	0.251 (0.4s)
Java.3	152	351	608	2	✔	2	0	0	0.435	0.218 (35.1s)
Java.4	152	351	608	14	✔	6	2	6	2.042	0.255 (6.5s)
Java.5	152	351	607	3	✔	3	0	0	0.526	0.175 (3.3s)

Amb?
Whether the grammar is ambiguous.

 # unif
Number of conflicts for which unifying counterexamples are found within the time limit.

 # nonunif
Number of conflicts for which nonunifying counterexamples are found within the time limit.

 # time out
Number of conflicts for which the tool times out. Nonunifying counterexamples are reported for these conflicts.

Total time
Time used when counterexamples are found within the time limit. (Average of 15 runs, with a standard deviation of at most 15%, so the margin of error is at most 9% at 95% confidence.)

Average time

$$\frac{\text{Total time}}{\text{\# unif} + \text{\# nonunif}}$$

few grammars in previous software projects that pose challenging parsing conflicts are also part of the evaluation.

Grammars from StackOverflow and StackExchange We evaluate our tool against grammars posted on StackOverflow and StackExchange by developers who had difficulty understanding the conflicts. This section of Table 1 links to the corresponding web pages.

Grammars from existing tool To compare our implementation with the state of the art, we run our tool against the grammars used to evaluate the grammar filtering technique [5]. These grammars, which we call the *BV10 grammars* hereafter, were constructed by injecting conflicts into correct grammars for mainstream programming languages. In some grammars (e.g., Java.2), the addition of a nullable production generates a large number of conflicts.

7.2 Effectiveness

Our tool always gives a counterexample for each conflict in every grammar. Table 1 reports the numbers of conflicts for which our tool successfully finds a unifying counterexample (if the grammar is ambiguous), for which our tool determines that no unifying counterexample exists, and for which our tool times out and hence reports a nonunifying counterexample. For grammars requiring more than two minutes of the main search algorithm, the number of remaining conflicts is shown in parentheses. Our implementation finishes within the time limit on 92% of the conflicts.

The main search algorithm may fail to find a unifying counterexample even if the grammar is ambiguous. One reason is the tradeoff used to reduce the number of configurations, as explained in Section 6. Grammar ambfailed01 illustrates this problem. Another reason is that the configuration describing the unifying counterexample has a cost too high for the algorithm to reach within the time limit. For instance, the ambiguous counterexample for grammar C.4 requires a long sequence of production steps. For these failures, nonunifying counterexamples are reported instead.

We also compare effectiveness against prior versions of the Polyglot Parser Generator (PPG) [18], which attempt to report only nonunifying counterexamples. PPG produces misleading results on ten benchmark grammars: figure1, figure7, abcd, simp2, SQL.5, Pascal.3, C.2, Java.1, Java.3, and Java.4. Incorrect counterexamples are generated because PPG's algorithm ignores conflict lookahead symbols. For instance, PPG reports this invalid counterexample for the dangling-else conflict:

```
if expr then stmt • else
if expr then stmt • else stmt
```

The unifying counterexamples given by our algorithm provide a more accurate explanation of how parsing conflicts arise. Our algorithm has been integrated into a new version of PPG.

7.3 Efficiency

We have measured the running time of the algorithm on the conflicts that our tool runs within the time limit. These measurements were performed on an Intel Core2 Duo E8500 3.16GHz, 4GB RAM, Windows 7 64-bit machine. The results are shown in the last two columns of Table 1. For the BV10 grammars, we also include in parentheses the time used on a similar machine by a grammar-filtering variant of CFGAnalyzer [2, 5], which is the fastest, on average, among the ambiguity detection tools we have found. This state-of-the-art ambiguity detector terminates as soon as it finds one ambiguous counterexample, whereas our tool finds a counterexample for every conflict. Hence, the running time of the state-of-the-art tool is compared against the average time taken per conflict in our implementation.

On average, when the time limit is not exceeded, the algorithm spends 0.18 seconds per conflict to construct a counterexample. For grammars taken from StackOverflow and StackExchange, the average is 8 milliseconds.

For the BV10 grammars, our algorithm outperforms the filtering technique. Based on a geometric average, our tool is 10.7 times faster than the variant of CFGAnalyzer, which takes more than 30 seconds to find a counterexample for certain grammars. (One grammar takes 0.0s for both tools and therefore dropped from the average.) For most of these grammars, the time our implementation takes to find counterexamples for all conflicts is less than that of the state-of-the-art tool trying to find just one counterexample. For grammar C.4, the CFGAnalyzer variant finds a unifying counterexample, but our tool fails to do so within the time limit. This result suggests that grammar filtering would be a useful addition to our approach.

7.4 Scalability

The evaluation results show that the running time of our algorithm only increases marginally on larger grammars, such as those for mainstream programming languages. The performance shown here demonstrate that, unlike prior tools, our counterexample finder is practical and suitable for inclusion in LALR parser generators.

8. Related Work

Generating counterexamples is just one way to help address parsing conflicts. In general, several lines of work address ways to deal with such problems. We discuss each of them in turn.

Ambiguity detection Several semi-decision procedures have been devised to detect ambiguity. Pandey provides a survey [19] on these methods, some of which we discuss below.

One way to avoid undecidability is to approximate input CFGs. The Noncanonical Unambiguity (NU) test [25] uses equivalence relations to reduce the number of distinguishable derivations of a grammar, reducing the size of the search space but overapproximating the language. Its mutual accessibility relations are analogous to actions in our product parser. Basten extends the NU test to identify a nonterminal that is the root cause of ambiguity [3]. One challenge of the NU test is choosing appropriate equivalence relations.

A brute-force way to test ambiguity is to enumerate all strings derivable from a given grammar and check for duplicates. This approach, used by AMBER [27], is accurate but prohibitively slow. Grammar filtering [5] combines this exhaustive approach with the approximative approach from the NU test to speed up discovery of ambiguities. AmbiDexter [4] uses parallel simulation similar to our approach, but on the state machine of an LR(0), grammar-filtered approximation that accepts a superset of the actual language. This allows false positives.

CFGAnalyzer [2] converts CFGs into constraints in propositional logic that are satisfiable if any nonterminal can derive an ambiguous phrase whose length is within a given bound. This bound is incremented until a SAT solver finds the constraints satisfiable. CFGAnalyzer does report counterexamples, but never terminates on unambiguous input grammars even if there is a parsing conflict.

Schmitz's experimental ambiguity detection tool [26] for Bison constructs a nondeterministic automaton (NFA) of pairs of parser items similar to our product parser states. Its reports of detected and potential ambiguities remain similar to parsing conflict reports and hence difficult to interpret. Counterexample generation remains future work for Schmitz's tool. To obtain precise ambiguity reports for LALR(1) construction, this tool must resort to constructing NFAs for LR(1) item pairs.

SinBAD [29] randomly picks a production of a nonterminal to expand when generating sentences, increasing the chance of discovering ambiguity without exhaustively exploring the grammar. SinBAD's search still begins at the start symbol, so reported counterexamples might not identify the ambiguous nonterminal.

Counterexample generation Some additional attempts have been made to generate counterexamples that illustrate ambiguities or parsing conflicts in a grammar.

Methods for finding counterexamples for LALR grammars can be traced back to the work of DeRemer and Pennello [11], who show how to generate nonunifying counterexamples using relations used to compute LALR(1) lookahead sets. Unfortunately, modern implementations of parser generators do not compute these relations. Our method provides an alternative for finding nonunifying counterexamples without requiring such relations, and offers a bonus of finding unifying counterexamples when possible.

DMS [8] is a program analysis and transformation system whose embedded parser generator allows users to write grammars directly within the system. When a conflict is encountered, DMS uses an iterative-deepening [16] brute-force search on all grammar rules to find an ambiguous sentence [7]. This strategy can only discover counterexamples of limited length in an acceptably short time.

CUP2 [30] reports the shortest path to the conflict state, while prior versions of PPG [18] attempt to report nonunifying counterexamples. These parser generators often produce invalid counterexamples because they fail to consider lookahead symbols.

While less powerful than LR grammars, LL grammars can also produce conflicts. The ANTLR 3 parser generator [20] constructs counterexamples, but they can be difficult to interpret. For instance, ANTLR 3 provides the counterexample ⟨*digit*⟩⟨*digit*⟩ for the the challenging conflict in Section 3.1[6]. Our technique describes the ambiguity more clearly.

Conflict resolution Generalized LR parsing [28] keeps track of all possible interpretations of the input seen so far by forking the parse stack. This technique avoids LR conflicts associated with having too few lookahead symbols but requires users to merge the outcomes of ambiguous parses at parse time. Our approach, which pinpoints ambiguities at parser construction time, is complementary and applicable to GLR parsing.

The GLR parsing algorithm is asymptotically efficient for typical grammars, but its constant factor is impractically high. Elkhound [17] is a more practical hybrid between GLR and LALR

[6] The example grammar was modified to eliminate left recursion.

parsing. It can display different derivations of ambiguous sentences, but the user must provide these sentences.

The eyapp tool [24], a yacc-like parser generator for Perl, postpones conflict resolution until actual parsing. Users can write code that inspects parser states and provides an appropriate resolution.

SAIDE [22, 23] is an LALR parser generator that automatically removes conflicts arising from insufficient number of lookaheads, and attempts to detect ambiguities by matching conflicts with predefined patterns of known cases. Although this approach guarantees termination, conflicts could be miscategorized.

Dr. Ambiguity [6] provides diagnostics explaining causes of ambiguities as an Eclipse [12] plugin, but a collection of parse trees demonstrating ambiguities must be provided as input.

ANTLR 4 [21] uses textual ordering of productions as precedence and abandons static detection of conflicts. Textual ordering makes grammars less declarative, but ambiguous inputs can still exist; any ambiguities are discovered only at parse time.

9. Conclusion

Better tools that help language designers quickly find potential flaws within language syntax can accelerate the design and implementation of programming languages and promote the use of parser generators for problems involving custom data formats. Our method finds useful counterexamples for faulty grammars, and evaluation of the implementation shows that the method is effective and practical. This paper suggests that the undecidability of ambiguity for context-free grammars should not be an excuse for parser generators to give poor feedback to their users.

Acknowledgments

Andrew Hirsch, Matthew Milano, Isaac Sheff, Owen Arden, Jed Liu, and Vivek Myers offered useful suggestions for the presentation. Steve Blackburn and Ben Hardekopf helped with the review process.

This work was supported by Office of Naval Research grant N00014-13-1-0089 and National Science Foundation grant 0964409.

References

[1] Tadashi Ae. Direct or cascade product of pushdown automata. *Journal of Computer and System Sciences*, 14(2):257–263, 1977.

[2] Roland Axelsson, Keijo Heljanko, and Martin Lange. Analyzing context-free grammars using an incremental SAT solver. In *Automata, Languages and Programming*, volume 5126 of *Lecture Notes in Computer Science*, pp. 410–422. Springer Berlin Heidelberg, 2008.

[3] H. J. S. Basten. Tracking down the origins of ambiguity in context-free grammars. In *Theoretical Aspects of Computing – ICTAC 2010*, volume 6255 of *Lecture Notes in Computer Science*, pp. 76–90. Springer Berlin Heidelberg, 2010.

[4] H. J. S. Basten and T. van der Storm. AmbiDexter: Practical ambiguity detection. In *Proc. 10th IEEE Int'l Workshop on Source Code Analysis and Manipulation (SCAM 2010)*, pp. 101–102, Sept 2010.

[5] H. J. S. Basten and J. J. Vinju. Faster ambiguity detection by grammar filtering. In *Proc. 10th Workshop on Language Descriptions, Tools and Applications*, pp. 5:1–5:9, 2010.

[6] Hendrikus J. S. Basten and Jurgen J. Vinju. Parse forest diagnostics with Dr. Ambiguity. In *Software Language Engineering*, volume 6940 of *Lecture Notes in Computer Science*, pp. 283–302. Springer Berlin Heidelberg, 2012.

[7] Ira Baxter. What is the easiest way of telling whether a BNF grammar is ambiguous or not? (answer). StackOverflow, July 2011. Retrieved November 11, 2014.

[8] Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. DMS®: Program transformations for practical scalable software evolution. In *Proc. 26th Int'l Conf. on Software Engineering (ICSE)*, pp. 625–634, May 2004.

[9] CVE. CVE-2013-0269. Common Vulnerabilities and Exposures, February 2013. Retrieved November 13, 2014.

[10] CVE. CVE-2013-4547. Common Vulnerabilities and Exposures, November 2013. Retrieved November 13, 2014.

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

[12] J. des Rivières and J. Wiegand. Eclipse: A platform for integrating development tools. *IBM Systems Journal*, 43(2):371–383, 2004.

[13] John Hopcroft and Jeffrey Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, MA, 1979. ISBN 978-0-201-02988-8.

[14] Scott E. Hudson, Frank Flannery, C. Scott Ananian, Dan Wang, and Andrew Appel. CUP LALR parser generator for Java, 1996. Software release. At `http://www.cs.princeton.edu/~appel/modern/java/CUP/`.

[15] Stephen C. Johnson. Yacc: Yet another compiler-compiler. Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ, July 1975.

[16] Richard E. Korf. Depth-first iterative-deepening: an optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, September 1985.

[17] Scott McPeak and George C. Necula. Elkhound: A fast, practical GLR parser generator. In *Proc. 13th Int'l Conf. on Compiler Construction (CC'04)*, pp. 73–88, 2004.

[18] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *Proc. 12th Int'l Conf. on Compiler Construction (CC'03)*, pp. 138–152, April 2003.

[19] Hari Mohan Pandey. Advances in ambiguity detection methods for formal grammars. *Procedia Engineering*, 24(0):700–707, 2011. International Conference on Advances in Engineering 2011.

[20] Terence Parr and Kathleen Fisher. LL(*): The foundation of the ANTLR parser generator. In *Proc. 32nd ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pp. 425–436, 2011.

[21] Terence Parr, Sam Harwell, and Kathleen Fisher. Adaptive LL(*) parsing: The power of dynamic analysis. In *Proc. 2014 ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 579–598, 2014.

[22] Leonardo Teixeira Passos, Mariza A. S. Bigonha, and Roberto S. Bigonha. A methodology for removing LALR(k) conflicts. *Journal of Universal Computer Science*, 13(6):737–752, June 2007.

[23] Leonardo Teixeira Passos, Mariza A. S. Bigonha, and Roberto S. Bigonha. An LALR parser generator supporting conflict resolution. *Journal of Universal Computer Science*, 14(21):3447–3464, December 2008.

[24] C. Rodriguez-Leon and L. Garcia-Forte. Solving difficult LR parsing conflicts by postponing them. *Comput. Sci. Inf. Syst.*, 8(2):517–531, 2011.

[25] Sylvain Schmitz. Conservative ambiguity detection in context-free grammars. In *Automata, Languages and Programming*, volume 4596 of *Lecture Notes in Computer Science*, pp. 692–703. Springer Berlin Heidelberg, 2007.

[26] Sylvain Schmitz. An experimental ambiguity detection tool. *Science of Computer Programming*, 75(1–2):71–84, 2010. Special Issue on ETAPS 2006 and 2007 Workshops on Language Descriptions, Tools, and Applications (LDTA '06 and '07).

[27] Friedrich Wilhelm Schröer. AMBER, an ambiguity checker for context-free grammars. Technical report, Fraunhofer Institute for Computer Architecture and Software Technology, 2001.

[28] Masaru Tomita, editor. *Generalized LR Parsing*. Springer US, 1991. ISBN 978-1-4613-6804-5.

[29] Naveneetha Vasudevan and Laurence Tratt. Detecting ambiguity in programming language grammars. In *Proc. 6th Int'l Conf. on Software Language Engineering*, pp. 157–176, October 2013.

[30] Andreas Wenger and Michael Petter. CUP2 LR parser generator for Java, 2014. Software beta release. At `http://www2.in.tum.de/cup2`.