Resolvable Ambiguity: Principled Resolution of Syntactically Ambiguous Programs

Viktor Palmkvist
vipa@kth.se
Digital Futures and EECS
KTH Royal Institute of Technology
Stockholm, Sweden

Elias Castegren
eliasca@kth.se
Digital Futures and EECS
KTH Royal Institute of Technology
Stockholm, Sweden

Philipp Haller
phaller@kth.se
Digital Futures and EECS
KTH Royal Institute of Technology
Stockholm, Sweden

David Broman
dbro@kth.se
Digital Futures and EECS
KTH Royal Institute of Technology
Stockholm, Sweden

Abstract

When building a new programming language, it can be useful to compose parts of existing languages to avoid repeating implementation work. However, this is problematic already at the syntax level, as composing the grammars of language fragments can easily lead to an ambiguous grammar. State-of-the-art parser tools cannot handle ambiguity truly well: either the grammar cannot be handled at all, or the tools give little help to an end-user who writes an ambiguous program. This composability problem is twofold: (i) how can we detect if the composed grammar is ambiguous, and (ii) if it is ambiguous, how can we help a user resolve an ambiguous program? In this paper, we depart from the traditional view of unambiguous grammar design and enable a language designer to work with an ambiguous grammar, while giving users the tools needed to handle these ambiguities. We introduce the concept of *resolvable ambiguity* wherein a user can resolve an ambiguous program by editing it, as well as an approach to computing the resolutions of an ambiguous program. Furthermore, we present a method based on property-based testing to identify if a composed grammar is unambiguous, resolvably ambiguous, or unresolvably ambiguous. The method is implemented in Haskell and evaluated on a large set of language fragments selected from different languages. The evaluation shows that (i) the approach can handle significantly more cases of language compositions compared to approaches which ban ambiguity altogether, and (ii) that the approach is fast enough to be used in practice.

CC '21, March 2–3, 2021, Virtual, Republic of Korea

CCS Concepts: • **Theory of computation → Grammars and context-free languages**; *Regular languages*; • **Software and its engineering → Parsers**; • **General and reference** → *Performance*; *Evaluation.*

Keywords: Syntax, Ambiguity

ACM Reference Format:
Viktor Palmkvist, Elias Castegren, Philipp Haller, and David Broman. 2021. Resolvable Ambiguity: Principled Resolution of Syntactically Ambiguous Programs. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction (CC '21), March 2–3, 2021, Virtual, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3446804.3446846

1 Introduction

The potential advantages of a domain-specific language (DSL) over a general-purpose programming language are quite appealing: a tailor-made tool for a particular kind of problems, enabling higher-level reasoning, which in turn gives opportunities for more analysis and optimization. However, implementing a programming language is no easy task, even if the final product is much smaller in scope than a general-purpose programming language.

A key concept in software engineering is that of compositionality: making a greater whole by composing smaller pieces. When implementing a programming language this can take the shape of language composition [10, 18, 30, 31, 40]: composing smaller language fragments to create a larger language. However, this approach brings its own set of challenges. In this paper we consider one of these challenges: the syntax of a composed language (i.e., its grammar) can become ambiguous even if its composed language fragments are individually unambiguous. Worse still, we generally cannot know if the composed grammar is ambiguous; detecting if a context-free grammar is ambiguous is undecidable [11]. Numerous approaches exist for dealing with this problem, ranging from heuristics for ambiguity detection [5, 7, 8], to restrictions on the composed grammars [24].

Note that all these efforts are for a singular cause: to reject all ambiguous *grammars*. This is a natural approach given that most literature describes this as *the* singular way to handle ambiguity [3, 13, 21, 38, 43]. Following our previous work [31] we take a slightly less drastic approach: we merely reject ambiguous *programs*, similarly to [14, 16]. This means that grammar composition is less likely to be problematic because grammar ambiguity is not automatically a problem. However, it presents us with a new question: what do we do when programmers write an ambiguous program?

An ambiguous program is normally defined as a program with more than one parse tree (or equivalently, multiple left-most derivations). We consider a slightly different definition: an ambiguous program is a program with more than one *abstract syntax tree* (AST). We thus consider parsing to be a function *parse* from programs to sets of ASTs, where each AST is a different interpretation of a parsed program.

Previous approaches that handle ambiguity do so by showing each valid (localized) AST [14, 16, 31], with the expectation that the user figures out how to resolve the ambiguity, typically using grouping parentheses. This is quite useful for a language designer, but less appropriate for an end-user; the AST is an implementation detail, and it may not be possible for an end-user to disambiguate the program.

Our approach instead presents an unambiguous program for each possible interpretation, i.e., we present concrete rewrites that resolve the ambiguity. Slightly more formally, given an ambiguous program p, and for all $t \in \mathit{parse}(p)$ we wish to present another program p' such that $\mathit{parse}(p') = \{t\}$. As mentioned, this is not always possible: not all trees t have a corresponding unambiguous program p'. In such a case we say that p is *unresolvably ambiguous*.

Extending this concept to grammars, we have three possible classifications of a grammar:

1. It is unambiguous.
2. It is *resolvably* ambiguous, i.e., every ambiguous program can be resolved by a programmer.
3. It is *unresolvably* ambiguous, i.e., there is at least one ambiguous program that a programmer cannot resolve.

The state of the art in parsing does not distinguish the latter two. We make a distinction between resolvable and unresolvable ambiguities, and show how to automatically suggest resolutions for the former.

However, detecting whether a grammar is ambiguous (resolvably or unresolvably so) is still difficult. Our approach builds upon property-based testing (PBT) [12]. We formulate "unambiguous" and "unambiguous or resolvably ambiguous" as properties of programs, then generate an arbitrarily large amount of syntactically valid programs of increasing size for a given grammar and test if the properties hold. Any counterexamples are shrunk and then reported to the user. The formulation is simple, and surprisingly effective in practice: when a property does not hold for a grammar we typically find a counterexample within a few seconds.

As a further refinement, we also allow a language designer to mark a discovered resolvable ambiguity as accepted, whereby our PBT-tool no longer reports that ambiguity (or variations of it), allowing other ambiguities to be reported, should they be present.

In summary, we make the following contributions:

- The concept of *resolvable ambiguity* and an associated language design workflow (Section 2).
- To enable computing resolutions we introduce the concept of an AST-language, the set of programs that can be parsed as (at least) a given AST. A key insight is to formulate these as visibly-pushdown languages [4], which enables later analysis (Section 4).
- We use these AST-languages to compute the resolutions of resolvably ambiguous programs. Our approach localizes the ambiguous part of the program (similar to previous approaches), and additionally, in the common case also produces concrete rewritings into unambiguous programs (Section 5).

We also make the following claims: (i) our approach can handle significantly more cases of language composition without additionally modifying the composed language when compared with approaches requiring unambiguous compositions, and (ii) our implementation is fast enough for practical use, both for resolving ambiguities in written programs and finding ambiguities in grammars using PBT. We support these claims with an implementation that we describe and evaluate in Section 6. Note that we make no contributions or claims on the process of parsing itself, though we do place some restrictions on its output, see Section 5.

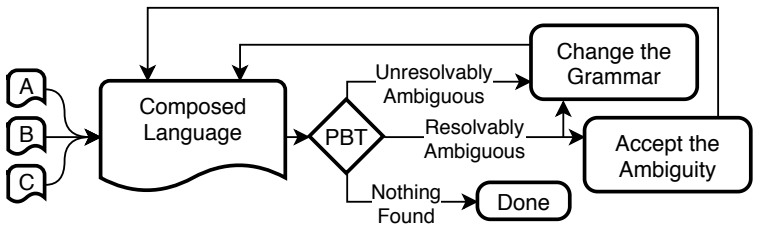
2 Resolvable Ambiguity and a Language Design Workflow

Given a function *parse* from programs to sets of ASTs, we say that a program p is resolvable iff:

$$\forall t \in \mathit{parse}(p).\ \exists p'.\ \mathit{parse}(p') = \{t\}$$

Note that an unambiguous program is trivially resolvable. With this new tool in hand we propose the workflow presented in Figure 1a. The language designer starts by picking the language fragments they wish to include, which yields a composed language. Next, we run our PBT analysis which gives one of three results:

- Resolvably ambiguous. In this case we have a choice: either we change the grammar to remove the ambiguity, or we accept the ambiguity and leave it in the language. If we leave it the tool will not report this particular ambiguity again.




```
Ambiguity error with 2 alternatives.
  if ( a ) { if ( ... ) ... } else ...
  if ( a ) { if ( ... ) ... else ... }
example#2:1:
  2: if (a)
  3:   if (foo(1 + 2)) return 1
  4:   else return 2
```

Figure 1. (a) The workflow for a language designer. (b) Example ambiguity error.

- Unresolvably ambiguous. In this case we must change the grammar to remove the ambiguity, since an end-user encountering the ambiguity would be stuck, they would not be able to resolve the ambiguity.
- No unaccepted ambiguities. In this case we are done, the composed language most likely[1] only contains ambiguities the end-user can solve.

For an end-user, the workflow is now quite simple. An ambiguous program is treated as any other error the compiler can detect; compilation ends and the user gets an error message with guidance on how to solve the issue. As an example, Figure 1b shows the output of our tool when encountering the "dangling else" ambiguity.

3 Describing Concrete Syntax and ASTs

This section describes the grammars we use to describe the concrete syntax and corresponding abstract syntax trees (ASTs) of programs. Most literature does not explicitly deal with ASTs, choosing instead to work with parse trees or left-most derivations, leaving AST construction as an exercise for the reader. Our analysis needs to deal with ASTs directly, thus we must be explicit in their description. However, none of the ideas in this section are novel, we thus exemplify previous uses of similar ideas with references below. We begin with context-free grammars, then make the following extensions:

1. Our grammars are in Extended Backus-Naur Form, i.e., production right-hand sides are regular expressions (cf. [23]).
2. Productions are labelled (cf. [42]).
3. We add special support for grouping parentheses (cf. [42]).
4. Terminals are designated as semantically unimportant or important (cf. [22]).
5. Precedence and associativity are expressed through special annotations we call *exclusions* (cf. [2, 26]).

We use the following small language with addition, multiplication, lists, let-expressions, and numeric literals as a running example:

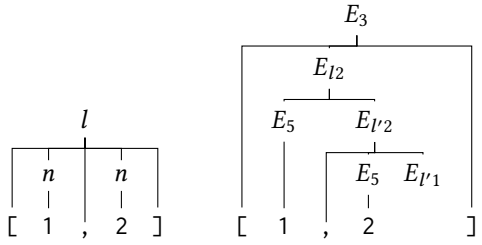
$$
\begin{array}{lcl}
E & \to & E\ \texttt{'+'}\ E \\
E & \to & E\ \texttt{'*'}\ E \\
E & \to & \texttt{'['}\ E_l\ \texttt{']'} \\
E & \to & \texttt{'let'}\ \texttt{Ident} \\
 & & \texttt{'='}\ E\ \texttt{'in'}\ E \\
E & \to & \texttt{Int}
\end{array}
\qquad
\begin{array}{lcl}
E_l & \to & \epsilon \\
E_l & \to & E\ E_{l'} \\
E_{l'} & \to & \epsilon \\
E_{l'} & \to & \texttt{','}\ E\ E_{l'}
\end{array}
$$

Monospace font represents terminals, e.g. `'let'` and `Ident`, while ϵ represents the empty sequence.

First we change the right-hand side of each production to be a regular expression instead of a sequence (i.e., our grammars are in Extended Backus-Naur Form, or EBNF). This serves to lessen the amount of intermediate nodes in the AST. We also add a label to each production, to give them unique identifiers. We use parentheses for grouping, + for union, and * for Kleene star.

$$
\begin{array}{lcll}
E & \to & a: & E\ \texttt{'+'}\ E \\
E & \to & m: & E\ \texttt{'*'}\ E \\
E & \to & l: & \texttt{'['}\ (E\ (\texttt{','}\ E)^* + \epsilon)\ \texttt{']'} \\
E & \to & x: & \texttt{'let'}\ \texttt{Ident}\ \texttt{'='}\ E\ \texttt{'in'}\ E \\
E & \to & n: & \texttt{Int}
\end{array}
$$

As an example, consider the string `'[1,2]'` with EBNF and labels (left) and without (right, using subscripts to specify which production produced the node, e.g., E_{l2} is the second production of the E_l non-terminal):



The extra internal nodes in the right tree would present problems for disambiguation later.

Next we add support for grouping parentheses in the form of a pair of terminals per non-terminal. Syntactically speaking these are equivalent with adding a production $E \to$ `'('`E`')'`, except they are discarded when constructing the AST. Note the difference between (x) and `'('`x`')'`; the former is the same as x, the latter is x surrounded by two terminals that happen to be parentheses. We also designate each

[1] This is an inherent limitation of PBT, see Section 6.2.

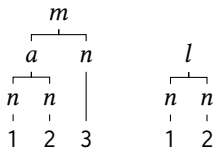
Productions				Grouping
E	$\rightarrow$	$a:$	$E_{\{x\}}$ `'+'` $E_{\{x,a\}}$	
E	$\rightarrow$	$m:$	$E_{\{x,a\}}$ `'*'` $E_{\{x,a,m\}}$	
E	$\rightarrow$	$l:$	`'['` $(E\,(\texttt{','}\ E)^* + \epsilon)$ `']'`	`'('`E`')'`
E	$\rightarrow$	$x:$	`'let'` Ident `'='` E `'in'` E	
E	$\rightarrow$	$n:$	Int	

Figure 2. Completed grammar of the running example.

terminal as semantically important or unimportant. Since most terminals fall into the second category we designate the important ones by underlining them:

Productions				Grouping
E	$\rightarrow$	$a:$	E `'+'` E	
E	$\rightarrow$	$m:$	E `'*'` E	
E	$\rightarrow$	$l:$	`'['` $(E\,(\texttt{','}\ E)^* + \epsilon)$ `']'`	`'('`E`')'`
E	$\rightarrow$	$x:$	`'let'` Ident `'='` E `'in'` E	
E	$\rightarrow$	$n:$	Int	

These changes mean that we can use grouping parentheses and unimportant terminals for disambiguation. The former is often used for operator expressions, while the latter can be used, e.g., to disambiguate with optional semicolons. For example, the strings `'(1+2)*3'` and `'[1,2]'` produce these ASTs:



Note that all unimportant terminals are discarded from both ASTs, as well as the grouping parentheses in the left AST.

Finally we consider precedence and associativity. A common approach is to create a *precedence ladder* by splitting each non-terminal into several non-terminals, one per precedence level. Explicitly doing this is undesireable in our setting, for two reasons:

1. It requires total precedence, but we wish to allow, e.g., leaving the precedence between operators from different languages undefined.
2. Grouping parentheses as described above recurse to the *same* non-terminal, which would then not contain productions representing lower precedence operators.

The former suggests that we cannot use precedence ladders unchanged, and the latter suggests that we need some explicit support in our grammars; we cannot simply rewrite the grammar without breaking grouping parentheses.

We thus adopt an approach that coincides with precedence ladders if precedence is total and transitive, but allows the absence of both these properties. To simplify later analysis we treat precedence and associativity as user-facing conveniences to be translated to a simpler form. We describe the simpler form first, then the translation, then illustrate both with examples.

E	$\rightarrow$	`'let'` Ident `'='` E `'in'` E
E	$\rightarrow$	E_1
E_1	$\rightarrow$	E_1 `'+'` E_2
E_1	$\rightarrow$	E_2
E_2	$\rightarrow$	E_2 `'*'` E_3
E_2	$\rightarrow$	E_3
E_3	$\rightarrow$	`'['` $(E\,(\texttt{','}\ E)^* + \epsilon)$ `']'`
E_3	$\rightarrow$	Int
E_3	$\rightarrow$	`'('` E `')'`

Figure 3. The same grammar as Figure 2 but in EBNF with a precedence ladder.

We introduce *exclusions*, annotations on the non-terminals appearing on the right-hand side of a production. Each exclusion is a set of production labels denoting the productions that may *not* appear in that position. For example, $E_{\{a,x\}}$ may parse as neither the production labelled a, nor the production labelled x. It may, however, parse as a pair of grouping parentheses that contain either of these productions. We write $E_\emptyset$ as E to reduce clutter.

Translation is then a process of examining the relative precedence of every pair of productions and the associativity of each production and adding exclusions as appropriate. For example, if $E \rightarrow m : E$ `'*'` E has higher precedence than $E \rightarrow a : E$ `'+'` E we update m to $E \rightarrow m : E_{\{a\}}$ `'*'` $E_{\{a\}}$. If m is additionally left-associative we update it to $E \rightarrow m : E_{\{a\}}$ `'*'` $E_{\{a,m\}}$.

Applying the usual precedences to our running example (let < addition < multiplication) as well as associativities (left-associative multiplication and addition) yields the grammar in Figure 2[2]. Note that production a has exclusions containing x due to precedence and a due to associativity, and correspondingly m has exclusions containing both x and a due to precedence, and m due to associativity.

To illustrate the concrete syntax in more familiar terms we also show a translation to normal EBNF grammars using a precedence ladder in Figure 3.

When describing grammar formalisms a natural question is that of expressiveness: how expressive are the grammars we present here? Technically speaking they retain the full expressiveness of context-free languages; we can trivially translate a CFG by simply generating unique labels for each production and designating all terminals important. However, such a translation does not use any of our added features

[2]The observant reader may notice that this grammar no longer recognizes, e.g., `1 + let x = 2 in 3`. This is a limitation we retain from precedence ladders, see Section 6.5.2 for more discussion

and would thus not benefit from any of our work; no disambiguation would be possible. We explore the implications of this in the evaluation, in Section 6.5.

We are now ready to describe the language of an AST.

4 The Language of an Abstract Syntax Tree

Central to our approach is the (seemingly obvious) idea that there need not be a one-to-one correspondence between programs and ASTs. One side of this is obvious from the presence of ambiguity: an ambiguous program corresponds to multiple ASTs. The other side is more important in our context: a single AST corresponds to multiple written programs, i.e., it can be written in multiple ways. This fact is necessary to enable disambiguation; given an ambiguous program p and an intended AST $t \in parse(p)$ we need to find another program p' such that $parse(p') = \{t\}$, but this would not be possible if each AST had only one corresponding program.

Note that this is achieved by ASTs not being perfect representations of written programs; some information is discarded, in our case tokens that are marked semantically unimportant and grouping parentheses. Finding a disambiguating program p' is thus a matter of "re-adding" whatever *parse* discarded, except we now have to consider all possible combinations of discards.

For example, in a language with arithmetics the AST obtained by parsing `'1 + 2 * 3'` could also have been obtained from, e.g., `'(1) + 2 * 3'`, `'1 + (2 * 3)'`, or `1 + ((2 * (3)))`. The set of such programs is typically infinite, since we can add any number of redundant grouping parentheses.

A key insight here is that this set of programs, the set of programs that can be parsed as (at least) the given AST, is a language in the language-theoretic sense: it is a set of words. We refer to this language as an *AST-language*, denoted by $words(t)$ for an AST t. We can thus reframe the problem of disambiguation as a language-theoretic one: searching for an unambiguous program is the same as searching for a program that belongs to $words(t)$ for *exactly one* t.

This section thus describes the construction of an automaton that recognizes $words(t)$ for a given t. To facilitate later analysis we ensure that the automaton is visibly pushdown, since visibly pushdown automata (VPDA) are closed under boolean operations [4]. Section 4.1 successively builds the automaton for a relatively simple case in order to give an intuition for the process, then Section 4.2 mentions additional considerations required for operation in general.

4.1 A Simple Case

To make figures tractably small we switch our running example language to a smaller one containing two unary operators (prefix `'!'` and postfix `'?'`, to show precedence) and a list with optional trailing comma (to show a slightly more complex discarded sequence of terminals):

Productions

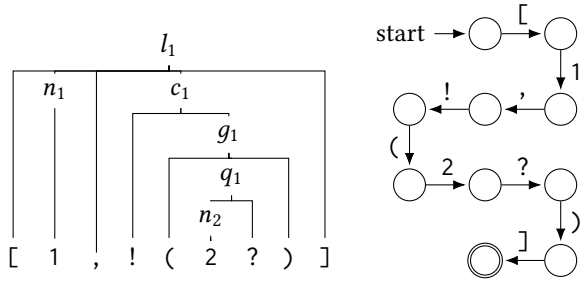
$$
\begin{array}{lcll}
E & \rightarrow & q: & E\ \texttt{'?'} \\
E & \rightarrow & c: & \texttt{'!'}\ E_{\{q\}} \\
E & \rightarrow & l: & \texttt{'['}\ (E\ (\texttt{','}\ E)^* + \epsilon)\ (\texttt{','} + \epsilon)\ \texttt{']'} \\
E & \rightarrow & n: & \underline{\texttt{Int}}
\end{array}
$$

Grouping

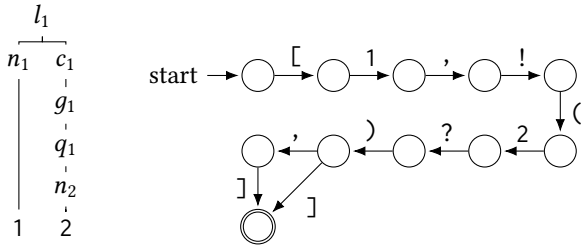
$$\texttt{'('}E\texttt{')'}$$

Our example constructs an automaton for the AST obtained from parsing `'[1, !(2?)]'` using the above grammar. Additionally, our process requires a way to uniquely identify each node in an AST, thus this section adds indices to the labels used for nodes in displayed ASTs. We consider our extensions in the same order as the previous section, i.e., at first we only consider a labelled EBNF grammar and ignore all other features, then we consider discarded nodes, then exclusions.

When parsing using a labelled EBNF grammar, without discarding any information, a given AST can only be parsed in exactly one way, namely the flattening of the tree. In this case, the AST (which is then more like a parse tree) looks as on the left (using g for grouping parentheses), and the AST-language is recognized by the pushdown automaton on the right (though we have no need of the stack yet):



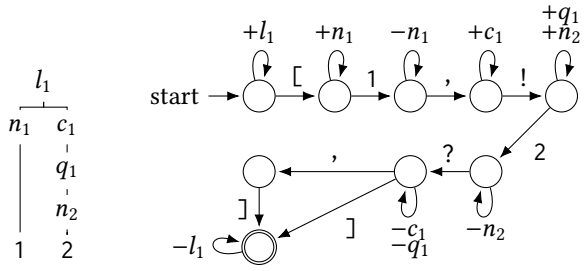
When ASTs additionally discard unimportant terminals the AST-language is no longer merely a flattening of the tree; it instead has to represent all possible discarded sequences of terminals. In some cases this yields no difference, e.g., production q discards exactly one terminal `'?'`. In other cases there is a small difference, e.g., the end of production l may discard either `',]'` or merely `']'`. Our running example thus yields:



This automaton recognizes `'[1, !(2?), ]'` as a different program that can parse as the same AST, which is correct since the trailing comma is discarded after parsing.

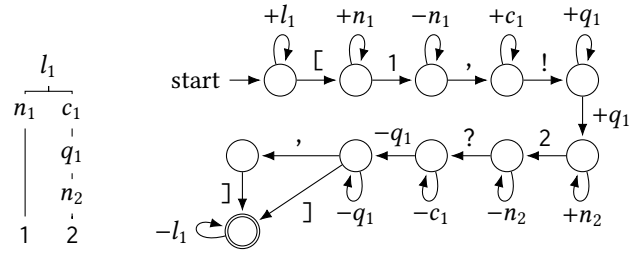
In general, the discarded sequence of terminals between two retained points in a production must be described by a regular language since all non-terminals are retained. For example, the AST obtained by parsing `'[]'` consists of a single node l with no children. The l production is described by the regular expression $\texttt{'['}\ (E\ \texttt{','}\ E)^* + \epsilon)\ (\texttt{','} + \epsilon)\ \texttt{']'}$. However, the sub-expression $E\ \texttt{','}\ E)^*$ cannot be part of the AST-language for this tree since it contains a non-terminal, which cannot be discarded. Replacing this sub-expression with the empty language and then simplifying we find that the AST-language for this tree is $\texttt{'['('','} + \epsilon)\texttt{']'}$.

Returning to the AST-language for `'[1, !(2?)]'`, we now take discarded grouping parentheses into account. The AST no longer contains the g_1 node, and the automaton must allow inserting an arbitrary (balanced) amount of parentheses around each node. We ensure this by pushing to the stack when recognizing a `'('` and popping on `')'`. To ensure that parentheses pair correctly around each node we use the (unique) label of the node as the stack symbol. To reduce clutter we write $\circ \xrightarrow{+a} \circ$ as shorthand for an edge that recognizes `'('` and pushes a on the stack, while $\circ \xrightarrow{-a} \circ$ correspondingly recognizes `')'` and pops a from the stack[3].



Note that this automaton recognizes `'[1, !2?]'` which is incorrect; this program cannot parse as the same AST because of the exclusion in production c, which we have yet to take into account.

An exclusion has the effect of forbidding a particular node from being a direct child without at least one intermediate pair of grouping parentheses. The AST-language must thus require at least one pair in such a location. In this case we have only one relevant exclusion: q may not be a direct child of c. The final AST-language is thus:



[3]This works for our example since we have only a single form of grouping parentheses, otherwise we would have to be explicit.

Note that there are now two required transitions $\circ \xrightarrow{+q_1} \circ$ and $\circ \xrightarrow{-q_1} \circ$, i.e., there is a required pair of parentheses around the q_1 node, which removes the incorrectly recognized program from the last step.

4.2 Trickier Cases

The previous section is sufficient for operation in the common case and gives a good intuition, but is not enough for the general case. Additionally, the following points need to be addressed:

- We need to maintain the visibly pushdown property even in the presence of brackets in productions.
- We need to track exactly which symbol produced each node in the AST.
- Non-total precedence cannot be parsed using precedence ladders and instead requires a different approach.

This does not change the intuition from the previous section, but the interested reader can read more on these points (especially our solutions to them) at doi:10.5281/zenodo.4458159.

5 Finding the Resolutions of an Ambiguity

This section describes our approach to finding the resolutions of an ambiguity using the automata described in the previous section. As a reminder, given an ambiguous program p, and for every $t \in parse(p)$ we wish to find another program p' such that $parse(p') = \{t\}$.

The core of our approach is simple: the VPDAs from Section 4 let us compute $A_t := words(t) \backslash \bigcup_{t' \in parse(p) \backslash \{t\}} words(t')$. Note that this is possible because VPDAs are closed under boolean operations [4]; it would not be possible for normal pushdown automata. If A_t recognizes the empty language, the ambiguity is unresolvable. Otherwise we can find a shortest program p' recognized by A_t. This program can parse as t, but no other $t' \in parse(p)$, by the construction of A_t. In practice p' is almost always unambiguous (c.f. Section 6.3), i.e., a valid resolution of the ambiguity, but this is not guaranteed. Thus, we additionally reparse p' and produce an error if it turns out to be ambiguous.

We also localize the ambiguity. In the common case the vast majority of a program is irrelevant for an ambiguity, e.g., in a language without precedence `'1 + (2 + (3 * 4) * 5)'` has the same ambiguity as `'2 + x * 5'`, which is a simpler, smaller program. Note that this can remove sibling trees, ancestors, and subtrees, all of which can be arbitrarily large. This greatly shrinks the problem size, and as an additional bonus presents smaller ambiguities to the user.

To do this, we exploit the sharing present in shared packed parse forests (SPPFs) [39], similar to [16]. For example, consider parsing the program `'1 + (2 + (3 * 4) * 5)'` with the following grammar:

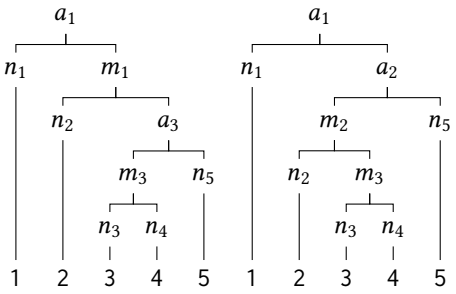


Figure 4. Two ASTs describing an ambiguous program.

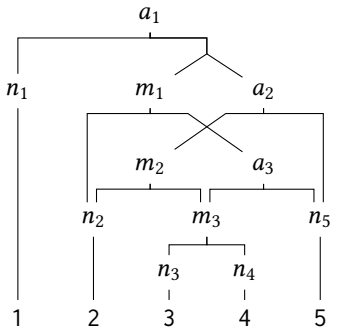


Figure 5. An SPPF describing the same program as Figure 4.

Productions				Grouping
E	$\rightarrow$	a :	E `'+'` E	
E	$\rightarrow$	m :	E `'*'` E	`'('`E`')'`
E	$\rightarrow$	n :	`Int`	

Figure 4 shows the two valid ASTs (the node indices are chosen to reflect possible sharing). Displaying these two trees as a single SPPF (Figure 5) makes the possible sharing explicit. Note that there is an ambiguity beginning in the right child of a_1 (which could be either m_1 or a_2) and that the ambiguity "stops" at n_2, m_3, and n_5; these descendants are shared amongst all alternatives. We can extend our approach for finding ambiguity resolutions to use this information in a fairly straightforward manner: instead of looking at the full ASTs we consider the subtrees rooted in m_1 and a_2, and instead of computing the full AST-language for n_2, m_3, and n_5 we treat them as new unique terminals.

This can give us drastically smaller automata, but it has an additional benefit: it can split a single ambiguous program into multiple independent ambiguities. For example, consider the program `'(1 + 2 + 3) + (4 + 5 + 6 + 7)'`. Without localization this is an ambiguous program with 10 ASTs, but with localization it is an ambiguous program containing two independent ambiguities with 2 and 5 alternatives, respectively. Additionally, each of the 7 subtrees in the localized version is smaller than each of the 10 trees in the non-localized version. Without localization the amount of work grows multiplicatively, with localization it grows additively.

Because of these advantages, our implementation parses directly to SPPFs, as opposed to sets of parse trees, as we have presented thus far.

6 Implementation and Evaluation

This section describes and evaluates our implementation to substantiate our claims in Section 1.

6.1 Implementation and Optimizations

Our implementation consists of ~4.4k lines of Haskell including a custom Earley [17] parser using optimizations presented in [6, 28], written to produce SPPFs [36]. The parser is written as a separate (but unpublished) package and handles grammars in LR(*) in linear time (since it uses [28]). Our VPDA operations follow [4] rather directly, thus there is likely space for optimization. Regardless, these operations have high time complexities, thus we also implement a shortcut: we simulate all of the VPDAs in lockstep along all possible paths until we find a word that is accepted by only one of them. This skips determinizing the VPDAs and computing differences, meaning it is typically faster than computing A_t for each t, but it is not guaranteed to terminate. In particular, simulating in lockstep might not terminate on an unresolvable ambiguity. When analyzing an ambiguity we start both of these processes and pick the result from the one that produces an answer first.

Figure 6 illustrates the path from an ambiguous program to an ambiguity error. We first parse the program into an SPPF, then use localization to produce a set of ASTs per ambiguity. For each ambiguity we continue by constructing the AST-language for each AST and attempt to find resolutions. As mentioned, we do this in two ways in parallel; simulating in lockstep (which directly produces a program for each AST), and computing A_t followed by finding a shortest recognized program for each. We pick the results from whichever finishes first and finally present each found program as a resolution. The ASTs for which no unambiguous program was found are unresolvable and are presented as such.

Our property-based testing approach for finding ambiguities in grammars, which should find unresolvably ambiguous grammars before they reach a user, uses an off-the-shelf PBT library[4] to generate concrete syntax trees (CSTs). In brief, the generator works by randomly picking productions with equal probabilities for each occurrence of a non-terminal, then at a certain depth switching to a modified grammar from which recursion has been removed. The depth differs per run; we start with smaller trees and then increase the size over time. The produced CST is then flattened, parsed, and taken through the path in Figure 6.

[4]Hedgehog: https://hackage.haskell.org/package/hedgehog

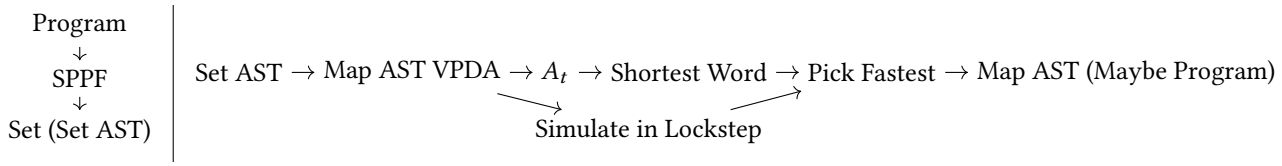


Figure 6. The path from an ambiguous program to an ambiguity error message, in two parts. Left: ambiguous program to a set of ambiguities, right: single ambiguity to ambiguity error.

The entire implementation can be found on GitHub[5].

6.2 Experimental Setup

We have created a total of 122 language fragments, consisting of one to seven productions each, where the vast majority have only one production. These fragments are created to follow the syntax of constructs available in three general-purpose programming languages and two DSLs: Haskell, OCaml, JavaScript, LINQ, and ScalaTest.[6] Examples include list comprehensions, match-expressions, if, lambdas, do-notation, operator sections, assertions, and LINQ-expressions. To enable reasonable compositions, we have ensured that all fragments use the same non-terminals for top-level declarations, statements, expressions, patterns, and types, and also that each production has a globally unique label. This enables the generation of composed grammars which, e.g., allow expressions mixing productions from Haskell, OCaml and JavaScript.

Composition is then a matter of picking a random subset of the 122 fragments of a desirable size, with a few restrictions. We have designated some fragments to be mutually exclusive, since they introduce trivially unresolvable ambiguities. We consider an unresolvable ambiguity trivial in two cases:

- Two productions have exactly the same right-hand side, e.g., JavaScript boolean negation and OCaml dereferencing both use `'!'` as a prefix operator.
- Two productions have almost exactly the same syntax, and they express the same concept in two different languages. For example, lists in Haskell and arrays in JavaScript are not exactly the same, JavaScript allows spread syntax (`[a, ...bs]`) and a trailing comma, but they represent the same concept and the Haskell syntax is a subset of the JavaScript syntax.

We construct 2000 composed languages consisting of 1 to 100 language fragments (20 languages per size), then run our PBT tool on them to collect data. Each run of the PBT tool generates 100 programs which we then parse and analyze. Each language gets up to two runs of the tool: once to look for unresolvable ambiguities, and if that finds nothing, once to look for resolvable ambiguities. The results let us classify the languages as:

Unresolvably ambiguous if the first finds something.
Resolvably ambiguous if the second finds something.
Unambiguos if neither run finds anything.

Note that due to the nature of randomized testing, the last two classifications are merely *probable:* tests can only prove the presence of issues (here, ambiguities), not their absence.

The data in this section is produced on a machine with an Intel Xeon Gold 6136 (12 cores) and 62 GiB of RAM. Additionally, we have data from a laptop with an Intel Core i7-8550U (4 cores) and 16 GiB of RAM.

All data from the experiment, including language fragments, logs, and a compiled executable can be found in a Docker image at doi:10.5281/zenodo.4458159.

The running time of this experiment was ~19h 17min.

6.3 Results

Figure 7 shows the number of composed languages for each category of ambiguity as a stacked area graph, plotted by the number of productions. Note that the number of resolvable languages is significantly larger than of those that are merely unambiguous, and that ambiguity appears almost immediately.

Figure 7 also shows where our approach produces an incorrect result; the red area represents languages where at least one suggested resolution did not fully resolve the ambiguity. As outlined in Section 5, this is an expected result, and it appears that roughly 28% of the random compositions exhibit this behavior. However, this only appears to happen on unresolvable languages, i.e., our approach still handles resolvably ambiguous languages correctly.

Figure 8 shows the distribution of running times of analyses on single ambiguities. This represents the time needed to produce an ambiguity error to an end-user. Note the timeout at 10s, 7.5% did not finish before this point. However, most runs finish quickly with a long tail, e.g., 87% have finished by 1s and 89% by 2s. Additionally, even in cases of timeout we can still pinpoint the location of the ambiguity and show it to the user, since localization is a simple traversal of the SPPF (benchmarks suggest ~3% of runtime, compared to ~90% for analysis and ~7% for parsing).

Figure 9 displays the distribution of running times for the PBT tool, split by whether the tool encountered the kind of ambiguity it was looking for or not. This represents the time a language designer needs to wait to know what kind of ambiguities are present in their language. Note that the

[5] https://github.com/miking-lang/syncon
[6] ScalaTest: https://www.scalatest.org/

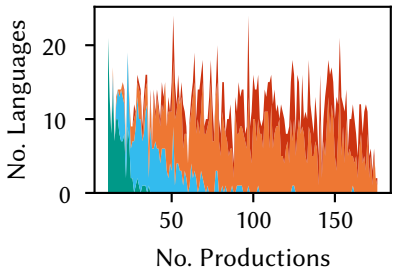


Figure 7. The distribution of unambiguous (green), resolvably ambiguous (blue), and unresolvably ambiguous grammars (orange and red), by number of productions.

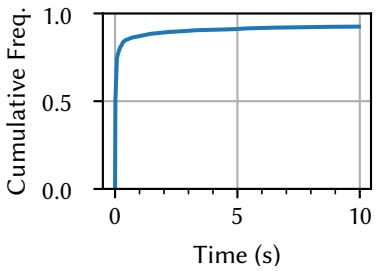


Figure 8. The normalized cumulative frequency of ambiguity analysis running times.

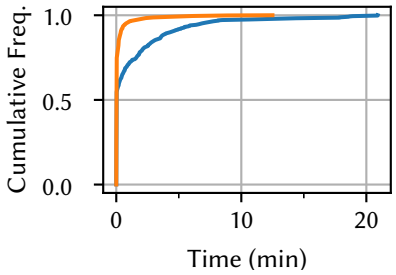


Figure 9. The normalized cumulative frequency of PBT running times where an ambiguity (of either kind) was found (orange) or not (blue). Each line ends at the slowest run.

frequencies are normalized for comparison. Most runs finish quickly, especially when ambiguity is found. This means that you typically get a quick answer when there is something you need to address in the grammar, otherwise you may need to wait a few minutes. For example, 95% of runs that find an ambiguity finish before 1s, while 95% of runs that do not find an ambiguity finish before 5min.

However, Figure 8 suggests that the timeout has sharply diminishing returns. Rerunning the experiment with 1s timeout instead of 10s corroborates this: 1938/2000 languages received the same classification, while the total running time of the experiment went from ~19h 17min to ~3h 34min. Of the languages that changed classifications, 43 received a less ambiguous class and 19 received a more ambiguous class. Recall that a more ambiguous class is the result of a concrete example, i.e., a more ambiguous class is more correct. We expect some languages to be reclassified since PBT is based on randomness, but the low number of changes bodes well for the accuracy of the approach.

Testing the same languages on a laptop with 1s timeout also produces similar results. Total running time was ~3h 54min, 51 languages received a less ambiguous label, and 11 languages received a more ambiguous label. The results are thus comparable, even on a weaker machine.

6.4 Case Study: Informal OCaml Grammar

We have additionally created a mostly complete grammar of OCaml (339 productions), to explore the applicability of our approach in a non-compositional context. Our grammar follows the informal grammar presented in the OCaml manual [29], which is meant to be understandable to users, but it is ambiguous. Many (though not all) of the ambiguities that arise are resolvable, whereby our implementation produces suggested resolutions. One interesting example (originally from [31]) is that of nested **`match`**-expressions. OCaml ignores whitespace and uses longest match to disambiguate nested matches, thus the following code has a type error:

```
match 1 with
| 1 -> match "one" with
       | str -> str
| 2 -> "two"
```

The issue is that the match-arm on the last line was intended to belong to the outer match, but OCaml puts it with the inner one. Our implementation instead sees an ambiguity and suggests putting parentheses to clarify. This is thus a case where resolvable ambiguity can present an error with a concrete solution to the problem, while OCaml might not present an error at all; if the types agreed the code would be silently wrong.

As a contrasting example, where it is not useful to leave disambiguation to the user, consider the expression `'Foo.bar'`. The informal grammar gives two possible interpretations: it is either a qualified value `'bar'` in the module `'Foo'`, or an access of the field `'bar'` of the nullary constructor `'Foo'`. OCaml uses the former, since the latter can never typecheck. Our implementation finds the ambiguity and reports it as unresolvable; the field access can be written as `'(Foo).bar'` but the qualified value has no other written form (expressions can be surrounded by parentheses, modules in a qualified name cannot).

6.5 Limitations

This section describes the limitations we have encountered while designing grammars using our tool.

6.5.1 Longest Match. Some commonly occurring language constructs depend on longest match to ensure that their syntax is unambiguous, perhaps most commonly if-then-else, commonly referred to as the "dangling else" problem. Our tool presently does not support this, which means that a language designer cannot disambiguate a grammar based on longest match, but it may still be possible for a programmer to disambiguate their program if the ambiguity is resolvable. Dangling else in particular is resolvable, see Figure 1b.

6.5.2 Low Precedence Unary Operators. Precedence ladders can change the recognized language in the presence of low-precedence unary operators [1]. For example, using normal CFGs, a language containing addition, let-expressions, and numeric literals could be written naively (left) or with a precedence ladder (right):

$$
\begin{array}{lcl}
E & \to & \texttt{'let' Ident} \\
 & & \texttt{'='}\ E\ \texttt{'in'}\ E \\
E & \to & E\ \texttt{'+'}\ E \\
E & \to & \texttt{Int}
\end{array}
\qquad
\begin{array}{lcl}
E_1 & \to & \texttt{'let' Ident} \\
 & & \texttt{'='}\ E_1\ \texttt{'in'}\ E_1 \\
E_1 & \to & E_2 \\
E_2 & \to & E_2\ \texttt{'+'}\ E_3 \\
E_2 & \to & E_3 \\
E_3 & \to & \texttt{Int}
\end{array}
$$

The left language recognizes `1 + let x = 2 in 3` while the right does not. Since our approach is based on precedence ladders we retain this limitation.

7 Previous and Related Work

In our previous work [31] we also create languages through composition and show ambiguity errors to users. However, the errors merely present a shallow view of the ASTs of the alternatives, no suggested resolutions, which are instead left for future work. This paper solves that problem.

Our related work falls in three categories: syntax definition formalisms (including subclasses of CFGs), language frameworks, and other approaches to ambiguity.

Afroozeh et al. [2]'s operator ambiguity removal patterns strongly resemble the exclusions presented in this paper. However, in special-casing (what in this paper would be) exclusions on left and right-recursions in productions they correctly handle low precedence unary operators (c.f. Section 6.5.2). Integrating these ideas, and later continuations [15] with our approach could be interesting future work.

Danielsson and Norell [14] give a method for specifying grammars for expressions containing mixfix operators. They allow non-transitive, non-total precedence, and similar to our approach, they do not reject ambiguous grammars, only ambiguous parses. They also introduce a concept of *precedence correct* expressions; expressions where direct children must have higher precedence than parents. This is more restrictive than our approach, e.g., in a language where `'+'` and `'*'` have no defined relative precedence they reject `'1 + 2 * 3'` as syntactically invalid, while we parse it as an ambiguous expression.

Parsing expression grammars [20] sidestep the issue of ambiguity by not introducing it at all. However, this also loses the potential gains of leaving certain ambiguities. Additionally, since the ordering of productions matter, composition of languages must be ordered, and the interactions between composed languages becomes non-obvious.

Most commonly used parser generators are based on unambiguous CFG subclasses, e.g., LL(k), LR(k), or LR(*). Others do not fit neatly in the Chomsky hierarchy, but still produce a single parse tree per parse, e.g., LL(*) [32] and ALL(*) [33]. Yet others produce multiple parse trees or other forms of parse forests, e.g., GLR [27], GLL [37], and Earley [17].

Silver [40], a system for defining extensible languages using attribute grammars, and its associated parser Copper [41] have a "Modular Well-Definedness Analysis" [24], the syntactic component of which can be found in [35]. This analysis guarantees that the composition of a base language and any number of extensions that have passed the analysis will compose to a grammar in LALR(1). This language class is more restrictive than resolvably ambiguous languages.

The detection of ambiguity in context-free grammars is undecidable in general [11], yet numerous heuristic approaches exist. Examples include linguistic characterizations and regular language approximations [8], using SAT-solvers [5], and other conservative approaches [34], For an overview, and additional approaches, see the PhD thesis of Basten [7].

SDF in its various versions [16, 22, 42] is the parser component of several language development tools (e.g. [25, 30]). It uses a general parser (in the case of SDF3, SGLR[16, 42]) and allows various forms of disambiguation, e.g., precedence and associativity, but also more advanced features such as layout [19] and typechecking [9]. In case of ambiguity, SDF produces an SPPF containing all valid ASTs.

8 Conclusion

In this paper, we introduce the concept of resolvable ambiguity. A language grammar is resolvably ambiguous if all ambiguities can be resolved by the end-user at parse time. We develop a method to compute resolutions based on visibly-pushdown automata and show how ambiguity localization can be performed. We evaluate our approach by implementing a toolchain that is based on property-based testing. The experiment shows (i) that our approach can handle significantly more cases of language compositions compared to state-of-the-art methods requiring unambiguous grammars, and (ii) that the toolchain is fast enough for practical use. An interesting direction of future work is to investigate conservative methods for determining resolvable ambiguity statically.

Acknowledgments

This project is financially supported by the Swedish Foundation for Strategic Research (FFL15-0032) and partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. The authors would like to thank the anonymous reviewers, Eelco Visser, and Sebastian Erdweg for valuable feedback.

References

[1] Annika Aasa. 1995. Precedences in Specifications and Implementations of Programming Languages. *Theoretical Computer Science* 142, 1 (May 1995), 3–26. https://doi.org/10.1016/0304-3975(95)90680-J

[2] Ali Afroozeh, Mark van den Brand, Adrian Johnstone, Elizabeth Scott, and Jurgen Vinju. 2013. Safe Specification of Operator Precedence Rules. In *Software Language Engineering (Lecture Notes in Computer Science)*, Martin Erwig, Richard F. Paige, and Eric Van Wyk (Eds.). Springer International Publishing, 137–156. https://doi.org/10.1007/978-3-319-02654-1_8

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools* (second ed.). Addison Wesley, Boston.

[4] Rajeev Alur and P. Madhusudan. 2004. Visibly Pushdown Languages. In *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing (STOC '04)*. ACM, New York, NY, USA, 202–211. https://doi.org/10.1145/1007352.1007390

[5] Roland Axelsson, Keijo Heljanko, and Martin Lange. 2008. Analyzing Context-Free Grammars Using an Incremental SAT Solver. In *Automata, Languages and Programming (Lecture Notes in Computer Science)*, Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz (Eds.). Springer Berlin Heidelberg, 410–422. https://doi.org/10.1007/978-3-540-70583-3_34

[6] John Aycock and R. Nigel Horspool. 2002. Practical Earley Parsing. *Comput. J.* 45, 6 (2002), 620–630. https://doi.org/10.1093/comjnl/45.6.620

[7] Bas Basten. 2011. *Ambiguity Detection for Programming Language Grammars*. Ph.D. Dissertation. Universiteit van Amsterdam.

[8] Claus Brabrand, Robert Giegerich, and Anders Møller. 2007. Analyzing Ambiguity of Context-Free Grammars. In *Implementation and Application of Automata (Lecture Notes in Computer Science)*, Jan Holub and Jan Žďárek (Eds.). Springer Berlin Heidelberg, 214–225. https://doi.org/10.1007/978-3-540-76336-9_21

[9] Martin Bravenboer, Rob Vermaas, Jurgen Vinju, and Eelco Visser. 2005. Generalized Type-Based Disambiguation of Meta Programs with Concrete Object Syntax. In *Generative Programming and Component Engineering (Lecture Notes in Computer Science)*, Robert Glück and Michael Lowry (Eds.). Springer, Berlin, Heidelberg, 157–172. https://doi.org/10.1007/11561347_12

[10] David Broman. 2019. A Vision of Miking: Interactive Programmatic Modeling, Sound Language Composition, and Self-Learning Compilation. In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2019)*. Association for Computing Machinery, New York, NY, USA, 55–60. https://doi.org/10.1145/3357766.3359531

[11] David G. Cantor. 1962. On The Ambiguity Problem of Backus Systems. *J. ACM* 9, 4 (Oct. 1962), 477–479. https://doi.org/10.1145/321138.321145

[12] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP*. 268–279. https://doi.org/10.1145/1988042.1988046

[13] Keith Cooper and Linda Torczon. 2011. *Engineering a Compiler* (second ed.). Elsevier.

[14] Nils Anders Danielsson and Ulf Norell. 2011. Parsing Mixfix Operators. In *Implementation and Application of Functional Languages (Lecture Notes in Computer Science)*, Sven-Bodo Scholz and Olaf Chitil (Eds.). Springer Berlin Heidelberg, 80–99. https://doi.org/10.1007/978-3-642-24452-0_5

[15] Luís Eduardo de Souza Amorim, Michael J. Steindorfer, and Eelco Visser. 2018. Towards Zero-Overhead Disambiguation of Deep Priority Conflicts. *Programming Journal* 2, 3 (2018), 13. https://doi.org/10.22152/programming-journal.org/2018/2/13

[16] Luís Eduardo de Souza Amorim and Eelco Visser. 2020. Multi-Purpose Syntax Definition with SDF3. In *Software Engineering and Formal Methods (Lecture Notes in Computer Science)*, Frank de Boer and Antonio Cerone (Eds.). Springer International Publishing, Cham, 1–23. https://doi.org/10.1007/978-3-030-58768-0_1

[17] Jay Earley. 1970. An Efficient Context-Free Parsing Algorithm. *Commun. ACM* 13, 2 (Feb. 1970), 94–102. https://doi.org/10.1145/362007.362035

[18] Torbjörn Ekman and Görel Hedin. 2007. The JastAdd System — Modular Extensible Compiler Construction. *Science of Computer Programming* 69, 1 (Dec. 2007), 14–26. https://doi.org/10.1016/j.scico.2007.02.003

[19] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. 2013. Layout-Sensitive Generalized Parsing. In *Software Language Engineering (Lecture Notes in Computer Science)*, Krzysztof Czarnecki and Görel Hedin (Eds.). Springer Berlin Heidelberg, 244–263. https://doi.org/10.1007/978-3-642-36089-3_14

[20] Bryan Ford. 2004. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '04)*. ACM, New York, NY, USA, 111–122. https://doi.org/10.1145/964001.964011

[21] Seymour Ginsburg and Joseph Ullian. 1966. Ambiguity in Context Free Languages. *J. ACM* 13, 1 (Jan. 1966), 62–89. https://doi.org/10.1145/321312.321318

[22] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. 1989. The Syntax Definition Formalism SDF—Reference Manual—. *SIGPLAN Not.* 24, 11 (Nov. 1989), 43–75. https://doi.org/10.1145/71605.71607

[23] Trevor Jim and Yitzhak Mandelbaum. 2010. Efficient Earley Parsing with Regular Right-Hand Sides. *Electronic Notes in Theoretical Computer Science* 253, 7 (Sept. 2010), 135–148. https://doi.org/10.1016/j.entcs.2010.08.037

[24] Ted Kaminski and Eric Van Wyk. 2013. Modular Well-Definedness Analysis for Attribute Grammars. In *Software Language Engineering (Lecture Notes in Computer Science)*, Krzysztof Czarnecki and Görel Hedin (Eds.). Springer Berlin Heidelberg, 352–371. https://doi.org/10.1007/978-3-642-36089-3_20

[25] Lennart C.L. Kats and Eelco Visser. 2010. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. ACM, New York, NY, USA, 444–463. https://doi.org/10.1145/1869459.1869497

[26] Paul Klint and Eelco Visser. 1994. Using Filters for the Disambiguation of Context-Free Grammars. In *Proceedings of the ASMICS Workshop on Parsing Theory*. Tech. Rep. 126–1994, Dipartimento di Scienze dell'Informazione, Università di Milano, Milano, Italy.

[27] Bernard Lang. 1974. Deterministic Techniques for Efficient Non-Deterministic Parsers. In *Automata, Languages and Programming (Lecture Notes in Computer Science)*, Jacques Loeckx (Ed.). Springer, Berlin, Heidelberg, 255–269. https://doi.org/10.1007/978-3-662-21545-6_18

[28] Joop M. I. M. Leo. 1991. A General Context-Free Parsing Algorithm Running in Linear Time on Every LR(k) Grammar without Using Lookahead. *Theoretical Computer Science* 82, 1 (May 1991), 165–176. https://doi.org/10.1016/0304-3975(91)90180-A

[29] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2018. *The OCaml System Release 4.07: Documentation and User's Manual*. Report.

[30] Florian Lorenzen and Sebastian Erdweg. 2016. Sound Type-Dependent Syntactic Language Extension. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 204–216. https://doi.org/10.1145/2837614.2837644

[31] Viktor Palmkvist and David Broman. 2019. Creating Domain-Specific Languages by Composing Syntactical Constructs. In *Practical Aspects of Declarative Languages (Lecture Notes in Computer Science)*, José Júlio Alferes and Moa Johansson (Eds.). Springer International Publishing, 187–203. https://doi.org/10.1007/978-3-030-05998-9_12

[32] Terence Parr and Kathleen Fisher. 2011. LL(*): The Foundation of the ANTLR Parser Generator. In *Proceedings of the 32Nd ACM*

SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11). ACM, New York, NY, USA, 425–436. https://doi.org/10.1145/1993498.1993548

[33] Terence Parr, Sam Harwell, and Kathleen Fisher. 2014. Adaptive LL(*) Parsing: The Power of Dynamic Analysis. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 579–598. https://doi.org/10.1145/2660193.2660202

[34] Sylvain Schmitz. 2007. Conservative Ambiguity Detection in Context-Free Grammars. In *Automata, Languages and Programming (Lecture Notes in Computer Science)*, Lars Arge, Christian Cachin, Tomasz Jurdziński, and Andrzej Tarlecki (Eds.). Springer Berlin Heidelberg, 692–703. https://doi.org/10.1007/978-3-540-73420-8_60

[35] August C. Schwerdfeger and Eric R. Van Wyk. 2009. Verifiable Composition of Deterministic Grammars. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 199–210. https://doi.org/10.1145/1542476.1542499

[36] Elizabeth Scott. 2008. SPPF-Style Parsing From Earley Recognisers. *Electronic Notes in Theoretical Computer Science* 203, 2 (April 2008), 53–67. https://doi.org/10.1016/j.entcs.2008.03.044

[37] Elizabeth Scott and Adrian Johnstone. 2010. GLL Parsing. *Electronic Notes in Theoretical Computer Science* 253, 7 (Sept. 2010), 177–189. https://doi.org/10.1016/j.entcs.2010.08.041

[38] Thomas A. Sudkamp. 1997. *Languages and Machines: An Introduction to the Theory of Computer Science*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[39] Masaru Tomita. 1985. An Efficient Context-Free Parsing Algorithm for Natural Languages. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 2 (IJCAI'85)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 756–764.

[40] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. 2010. Silver: An Extensible Attribute Grammar System. *Science of Computer Programming* 75, 1 (Jan. 2010), 39–54. https://doi.org/10.1016/j.scico.2009.07.004

[41] Eric R. Van Wyk and August C. Schwerdfeger. 2007. Context-Aware Scanning for Parsing Extensible Languages. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering (GPCE '07)*. ACM, New York, NY, USA, 63–72. https://doi.org/10.1145/1289971.1289983

[42] Eelco Visser. 1997. *Syntax Definition for Language Prototyping*. Ph.D. Dissertation. University of Amsterdam. Advisor(s) Paul Klint.

[43] Adam Brooks Webber. 2003. *Modern Programming Languages: A Practical Introduction*. Franklin, Beedle & Associates.