Statically Resolvable Ambiguity

VIKTOR PALMKVIST, KTH Royal Institute of Technology, Sweden
ELIAS CASTEGREN, Uppsala University, Sweden
PHILIPP HALLER, KTH Royal Institute of Technology, Sweden
DAVID BROMAN, KTH Royal Institute of Technology, Sweden

Traditionally, a grammar defining the syntax of a programming language is typically both context free and unambiguous. However, recent work suggests that an attractive alternative is to use ambiguous grammars, thus postponing the task of resolving the ambiguity to the end user. If all programs accepted by an ambiguous grammar can be rewritten unambiguously, then the parser for the grammar is said to be resolvably ambiguous. Guaranteeing resolvable ambiguity statically—for all programs—is hard, where previous work only solves it partially using techniques based on property-based testing. In this paper, we present the first efficient, practical, and proven correct solution to the statically resolvable ambiguity problem. Our approach introduces several key ideas, including splittable productions, operator sequences, and the concept of a grouper that works in tandem with a standard parser. We prove static resolvability using a Coq mechanization and demonstrate its efficiency and practical applicability by implementing and integrating resolvable ambiguity into an essential part of the standard OCaml parser.

CCS Concepts: • **Software and its engineering** → **Syntax**; **Parsers**.

Additional Key Words and Phrases: Parser, Resolvable Ambiguity, Coq, OCaml

ACM Reference Format:
Viktor Palmkvist, Elias Castegren, Philipp Haller, and David Broman. 2023. Statically Resolvable Ambiguity. *Proc. ACM Program. Lang.* 7, POPL, Article 58 (January 2023), 27 pages. https://doi.org/10.1145/3571251

1 INTRODUCTION

The traditional approach to developing grammars and parsers for programming languages is to construct *unambiguous* context-free grammars: parsing a program always results in at most one parse tree. However, ever since the early 1960s, it is well-known that the problem of deciding whether a context-free grammar is ambiguous or not is generally undecidable [Cantor 1962]. As a consequence, many variants of restricted grammars have been developed which make it decidable to check if it is ambiguous or not [Aho et al. 2006; Cooper and Torczon 2011; Ginsburg and Ullian 1966; Sudkamp 1997; Webber 2003]. Designing restricted unambiguous grammars has traditionally been seen as *the* way of developing syntax specifications and parsers for programming languages.

Recently, we introduced the idea of *resolvable ambiguity* [Palmkvist et al. 2021]. Instead of requiring the grammar to be unambiguous, the ambiguity check is postponed to parse time. That is, if a *program* cannot be parsed unambiguously, the compiler rejects the program and asks the user to resolve the program by rewriting the program in such a way that it can be parsed unambiguously. Several authors of previous work acknowledge the usefulness of enabling ambiguity in grammars [Danielsson and Norell 2011; de Souza Amorim and Visser 2020; Palmkvist and Broman 2019; The dafny-lang community 2022]. However, an ambiguous grammar is only useful if all

Authors' addresses: Viktor Palmkvist, Digital Futures and EECS, KTH Royal Institute of Technology, Stockholm, Sweden; Elias Castegren, Uppsala University, Uppsala, Sweden; Philipp Haller, Digital Futures and EECS, KTH Royal Institute of Technology, Stockholm, Sweden; David Broman, Digital Futures and EECS, KTH Royal Institute of Technology, Stockholm, Sweden.

© 2023 Copyright held by the owner/author(s).
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of the ACM on Programming Languages*, https://doi.org/10.1145/3571251.

programs allowed by the grammar can be resolved. Specifically, it would be directly dissatisfactory if a user can write a program that is parsed and reported by the compiler to be ambiguous, but is impossible to rewrite in an unambiguous manner. State-of-the-art solutions [Palmkvist et al. 2021] can perform comprehensive property-based testing, which gives fairly good confidence that the grammar is resolvably ambiguous, but there are no static guarantees.

It is currently unknown whether the general decision problem of resolvable ambiguity for context-free grammars is decidable, but it is likely not the case. As a consequence, it is an open problem to design a solution that (i) is expressive enough to accept standard programming language syntax, (ii) gives formal guarantees for the absence of unresolvable ambiguities statically, and (iii) makes it possible to implement checkers and parsers that are efficient and useful in practice. We say that a solution to this problem handles *statically resolvable ambiguity* in a correct and efficient way.

In this paper, we solve this open problem by designing a modular approach where the new concept of a *grouper* is used together with a parser. The solution is extensible and expressive (any standard parser can be used with the grouper), efficient (we show how essential parts of the standard OCaml parser can be efficiently replaced with our approach), and correct (we formalize the semantics in Coq and prove that all programs are resolvable given certain mild assumptions). Key ideas of our solution include the concepts of *splittable productions* and grouping of *operator sequences*, where the standard parser produces a sequence of operators that are later grouped into an abstract syntax tree.

Concretely, we make the following contributions:

- We develop and formalize the novel approach of implicit and explicit grouping that allows an end-user to resolve ambiguities. The grouper can be embedded in a traditional parser in a straightforward manner (Section 3).
- We provide a mechanized proof of correctness in Coq, stating that only resolvable ambiguities are possible, given certain mild assumptions about splittable productions (Section 4).
- We implement our approach as a library and evaluate it empirically using two case studies. Firstly, we show how the standard OCaml compiler's parser can be replaced with our approach, yielding a complete parser with resolvable ambiguities that is both efficient and usable in practice. Secondly, we develop a new parser generator for encoding ambiguous grammars, where non-trivial domain-specific languages (DSLs) can be defined using ambiguous grammars with the static resolvable ambiguity property (Sections 5 and 6).

2 MOTIVATING RESOLVABLE AMBIGUITY AND OVERVIEW

Splitting ambiguities into two categories, resolvable and unresolvable, is not common praxis, thus we begin this section by motivating resolvable ambiguity through examples. Next, we explain why a static guarantee of resolvability is a strongly desirable property. Finally, we give an overview of our approach and how it integrates into a conventional parsing approach.

2.1 Motivating Resolvable Ambiguity

Following our previous work [Palmkvist et al. 2021] we consider parsing to be a function *parse* from programs to sets of ASTs. In this view, ambiguity is defined in terms of the size of the returned set:

Definition 2.1. A program p is *ambiguous* if $\exists t_1, t_2 \in \mathit{parse}(p).\ t_1 \neq t_2$, i.e., it can parse as at least two distinct ASTs.

A program is *resolvable* if all its ASTs can be written unambiguously:

Definition 2.2. A program p is *resolvable* if $\forall t \in parse(p).\ \exists p'.\ parse(p') = \{t\}$.

Correspondingly, a program is *unambiguous* if it is not ambiguous, and *unresolvable* if it is not resolvable.

We approach the motivation from three angles: pre-existing languages, composing languages or combining libraries, and creating new domain-specific languages.

2.1.1 Pre-existing Languages. Consider the operators **==** (equality) and **&** (bitwise and). These operators are are commonly available with the same syntax and semantics in many programming languages. This means that, e.g., the expression **1 & 2 == 3** is valid in a large number of programming languages. However, despite knowing the semantics of every component of the expression, a reader can be forgiven for being unsure of what it should evaluate to, since it still differs between languages! For example, these are the informal evaluation steps for the same expression in C and Python:

C **1 & 3 == 1** → **1 & (3 == 1)** → **1 & 0** → **0**
Python **1 & 3 == 1** → **(1 & 3) == 1** → **1 == 1** → **True**

Note that the difference is not due to different semantics of the operators involved between the languages, rather it is due to differing precedence. Both languages make a choice that can surprise programmers; both Clang and GCC have a flag **-Wparentheses** to warn about the surprising behavior, and Python does not follow C's precedence conventions, even though most other programming languages do.

Resolvable ambiguity presents a third choice: leave the precedence undefined and produce an ambiguity error instead. The programmer can resolve the error by adding parentheses (with assistance from the compiler, see the error message below), which as an added benefit means that a later reader of the code also cannot be surprised by the behavior. Note that this approach is more principled than the approach behind the **-Wparentheses** flag mentioned above.

```
1 Error: The program is ambiguous:
2
3 1 & ( 3 == 1 )
4 ( 1 & 3 ) == 1
```

However, this does not mean that the programmer has to explicitly group every part of every expression. For example, a & b & c and $a + b == c$ can both still be unambigous (through left-associativity and precedence, respectively); ambiguity only arises when the language designer has chosen to leave the relative precedence of two operators undefined.

This last point somewhat subtle; precedence tends to be a total order, which leaves no room for ambiguity, resolvable or not. Defining the meaning of precedence in the absence of a total order is non-obvious but necessary to combine the convenience of precedence where it is commonly known (e.g., arithmetic, comparators, etc.) with compiler-assisted explicitness in less clear cases.

As a non-operator related example, certain language constructs do not have an explicit end marker, which means that nested uses are ambiguous in a naive grammar. "Dangling else" is a commonly known example of this; nesting two **if**s with only one **else** makes it non-obvious to which **if** the **else** belongs. However, in our experience nested **match**es in OCaml cause problems more often than nested **if**s, especially for beginners, thus we focus on this case instead.

Consider the following example (slightly modified from [Palmkvist and Broman 2019]), where the intent is that the last case should belong to the outer **match**, but OCaml's chosen interpretation groups it with the inner **match** instead:

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

In this case, the types of the values being matched differ (**int** vs **string**), thus the OCaml compiler provides a type error:

```
File "nested.ml", line 4, characters 2-3:
4 | | 2 -> "two"
      ^
Error: This pattern matches values of type int
       but a pattern was expected which matches
       values of type string
```

This is surprising, because the solution is to put parentheses around the inner **match**, yet parenthesizing an expression does not change its type. Furthermore, if the types happened to agree, then there would be no error and the code would be silently wrong. If instead the nesting was left ambiguous the compiler could catch the error and give explicit suggestions on how to fix it.

2.1.2 Custom Operators in Libraries/Language Composition. Not all operators and operator precedences are chosen by language designers; some programming languages support used-defined operators, e.g., Haskell, Swift, F#, and Scala. In these languages, a library author can design an interface that uses symbolic operators, a domain-specific language of sorts. These operators are typically designed to be used together and given precedences that make them compose sensibly. However, when combining operators from multiple libraries in a single expression, the precedence is no longer obvious.

Some systems (e.g., Haskell) give each operator a precedence level in the form of an integer. In these systems, operators from different libraries will have some defined precedence, however, this precedence is essentially incidental, especially if the libraries were designed by different authors.

In a system allowing ambiguity, such incidental precedence need not appear, instead we can leave it undefined unless explicitly specified.

2.1.3 Domain-specific Languages. Suppose we are defining a new textual modeling language, where components are composed in series using an infix operator `--`, and in parallel using an infix operator `||`. For instance, an expression `C1 -- C2` puts components `C1` and `C2` in series, whereas `C1 || C2` composes them in parallel. In such a case, what is then the meaning of the following expression?

```
C1 -- C2 || C3 || C4
```

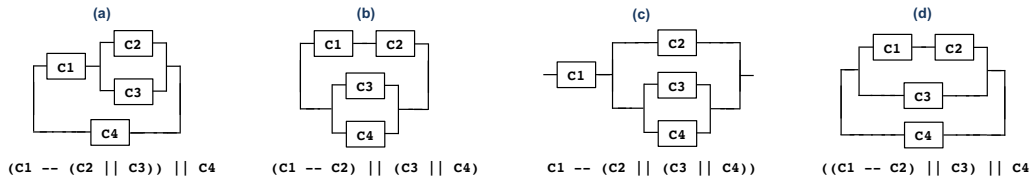


Fig. 1. The figure shows four different alternatives for disambiguating the expression `C1 - C2 || C3 || C4`. Note that there is a fifth alternative `C1 - ((C2 || C3) || C4)`. However, the meaning of this expression is the same as (c) assuming the parallel operator is associative. In such a case, this expression and the expression in (c) both mean that components `C2`, `C3`, and `C4` are composed in parallel.

Are there natural associativity and precedence rules for these new operators? If there are no predefined rules of how to disambiguate this expression within the language definition, it is an ambiguous expression, and a parser generates a set of parse trees. Consider Figure 1 which depicts four different alternatives, each with a different meaning, depending on how the ambiguity has been resolved. Clearly, the expression has totally different meanings depending on how the end-user places the parentheses. However, if a language designer is forced to make the grammar of the syntax definition unambiguous, a specific choice has to be made for precedence and associativity. For instance, assume that the designer makes the arbitrary choice that serial composition has higher precedence than parallel composition, and that both operators are left-associative. In such a case, the expression without parentheses is parsed as Figure 1(d). The question is why such an arbitrary choice—which is forced by the traditional design of unambiguous grammars—is the correct way to interpret a domain-specific expression. The alternative, as argued for in this paper, is to postpone the decision, and instead give an error message to the end-user (programmer or modeler), and expose different alternatives that disambiguate the expression.

2.2 The Importance of Static Guarantees for Resolvability

The definition of a resolvable program (Definition 2.2 on page 3) is not a tautology; we can construct a language in which not all syntactically valid programs are resolvable. As a somewhat trivial example, consider an expression language with arithmetic operators, integers, and parentheses, but no precedence. In such a language, most non-trivial expressions are ambiguous, e.g., $2 \cdot 3 + 4$ could evaluate to either 10 or 14. We can resolve the ambiguity by writing $(2 \cdot 3) + 4$ or $2 \cdot (3 + 4)$, respectively, depending on our intent.

However, if the language did not have parentheses then the above example would be unresolvably ambiguous; neither AST can be written in any other way. This is a problem, because it means that a programmer can be faced with an error that they *cannot* solve; a program that cannot be written without a parse error. In fact, only a language designer can solve the error, and only by changing the syntax of the language.

However, a static guarantee that all syntactically valid programs are resolvable makes such a situation impossible. A programmer might still be able to write an ambiguous program, but they would be able to rewrite it (with compiler assistance) into an unambiguous form, thereby solving the error.

2.3 Overview of Our Approach

Our approach centers around the idea of what we call *implicit and explicit grouping*. As a simple example, consider this equation:

$$a + b \cdot c - d \cdot e = ((a + (b \cdot c)) - (d \cdot e))$$

The right-hand side uses explicit grouping (parentheses), while the left-hand side uses implicit grouping (via precedence and left-associativity) to describe the same expression. We view an expression as a sequence of operators (e.g., + and −, but also variables and other terminals, as *nullary* operators) and provide a *grouper* that takes such a sequence (i.e., the left-hand side above) and assembles it into a fully grouped expression (i.e., the right-hand side). The process of parsing in our approach is thus a collaborative effort between two components:

- A traditional *parser* that identifies the components of each expression, in the form of a sequence of operators. The parser should be unambiguous, i.e., only one sequence is produced.

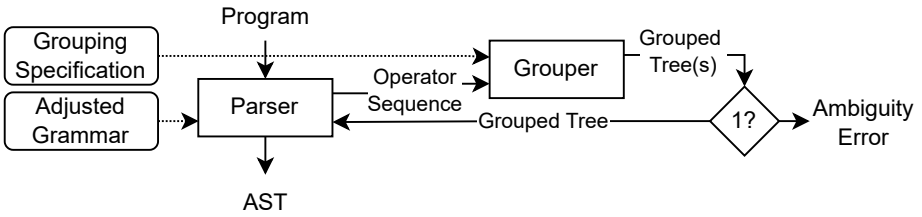


Fig. 2. An overview of a system embedding our approach in a conventional parser. The grammar used by the parser is adjusted to parse expressions as sequences of operators, which the grouper then arranges according to a grouping specification. If there are multiple valid trees, we produce an ambiguity error, otherwise the singular tree is returned to the parser, which continues parsing the rest of the program.

- A *grouper* that takes such a sequence and returns a fully grouped tree to the parser. The grouper handles ambiguity, i.e., it can produce multiple grouped trees, in which case the user receives an ambiguity error.

A grouper is thus used as a subroutine by the parser, as can be seen in the overview in Figure 2.

Note that there is a direct analogue between a parser and a grouper: a parser reads a sequence of tokens and produces an AST; a grouper reads a sequence of operators and produces a fully grouped expression. Similarly, the behavior of a parser is typically specified by a *grammar* of some kind, while our grouper follows a *grouping specification*:

	Parser	Grouper
Specification	Grammar	Grouping Specification
Input	Token Sequence	Operator Sequence
Output	AST	Grouped Tree(s)

We now walk through the components and flows in Figure 2 in a bit more detail. We center the discussion around a small example language and a program written therein, both of which can be found in Figure 3.

Splittable Productions. One production in Figure 3b bears special scrutiny: the **`if-then-else`** production. The optional **`else`** means that nested **`if`**s can be ambiguous if there are more **`if`**s than **`else`**s, i.e., the dangling else ambiguity. At this point, we have a choice: either we leave the (resolvable) ambiguity, or we resolve it implicitly through longest match.

Regardless, all implicit grouping in our approach is handled by the grouper, thus we first express the dangling else ambiguity in terms of operators. We do this by splitting **`if-then-else`** in two:

```
1  ① if isRed || isBlue then
2    ② if verbose then
3      ③ print ④ ("R/B")
4    ⑤ else
5      ⑥ print ⑦ ("?")
```

(a) The example program being parsed. The circled numbers are labels; they are not part of the program, they are used by later text and figures.

$$E ::= \textbf{Ident} \mid \textbf{String} \mid \textbf{(}E\textbf{)} \mid \textbf{if}\ E\ \textbf{then}\ E\ (\textbf{else}\ E)^{?} \mid E\ \textbf{||}\ E \mid E\textbf{(}E\textbf{)}$$

(b) A naive (and ambiguous) grammar for the example language, in EBNF ("?" denotes "zero or one").

Fig. 3. The program and language used as a running example.

$$
\begin{aligned}
E ::= \ & \texttt{Ident} \mid \texttt{String} \mid \texttt{(}E\texttt{)} \\
& \mid \texttt{if}\ E\ \texttt{then}\ E \\
& \mid E\ \texttt{else}\ E \\
& \mid E\ \texttt{||}\ E \mid E\texttt{(}E\texttt{)}
\end{aligned}
$$

(a) An intermediate version of Figure 3b where **if-then-else** has been split into two operators.

$$
\begin{aligned}
E &::= E^*_{prefix}\ E_{atom}\ E^*_{postfix} (E_{infix}\ E^*_{prefix}\ E_{atom}\ E^*_{postfix})^* \\
E_{atom} &::= \texttt{Ident} \mid \texttt{String} \mid \texttt{(}E\texttt{)} \\
E_{prefix} &::= \texttt{if}\ E\ \texttt{then} \\
E_{infix} &::= \texttt{||} \mid \texttt{else} \\
E_{postfix} &::= \texttt{(}E\texttt{)}
\end{aligned}
$$

(b) The final altered grammar that recognizes operator sequences. Note that no productions have direct edge recursion; all direct recursion is internal.

Fig. 4. Two modified grammars based on Figure 3b. Note that the left grammar is merely an intermediate result, the right grammar is the one used by our approach.

prefix **if-then** and infix **else**, resulting in the grammar found in Figure 4a. Note that this new grammar is more permissive than the one in Figure 3b; this is addressed later, in the grouper.

In this section we choose to implicitly resolve the ambiguity through longest match; we return to the ambiguous choice in Section 3.

Operator Sequences. Next, to produce the adjusted grammar mentioned to the left in Figure 2 we modify the grammar in Figure 4a to recognize operator sequences. The transformation consists of a simple set of steps: 1) We partition the productions of E based on the presence or absence of direct edge recursion, 2) we create a new non-terminal for each partition (i.e., E_{atom}, E_{prefix}, E_{infix}, and $E_{postfix}$), 3) we remove the edge recursion, and 4) add one new production to E. This new production is a regular expression that recognizes a well-formed sequence of operators, which can be seen in the beginning of Figure 4b (in EBNF, i.e., "$*$" is Kleene star). Note that each production with a subscripted E as its left-hand side represents an operator; it contributes one element to the operator sequence.

At this point we can make it fully clear when the grouper is invoked: every time the parser completes a production with E as its left-hand side, i.e., when we complete an operator sequence. In the example this happens five times: **isRed || isBlue** (three operators); **verbose**, **"R/B"** and **"?"** (one nullary operator each); and the entire program (seven operators). The fifth operator sequence can be seen in Figure 5. Each operator sequence is grouped separately, and the fully grouped result may end up inside an operator in another sequence. For example, in Figure 5, the operators labelled 1, 2, 4, and 7 each carry the results of grouping one of the other four sequences.

Grouping. The extra information carried by some operators is not used by the grouper, it is merely passed on. This is analogous to the information carried by a token; a parser examining a string token does not consider the exact string value, it only cares about it being a string; similarly, the grouper does not consider the AST for the condition in **if-then**, only that it is an **if-then** operator.

The grouper then transforms the operator sequence into one or more grouped trees according to the grouping specification, mentioned to the left in Figure 2. Intuitively speaking, the grouping specification has two components: the relative binding strength of two (ordered) operators,[1] and which operators are allowed as direct children for each operator. The former lets us express precedence, associativity, and longest match; the latter ensures that **if-then** and **else** pair properly, which addresses the overly permissive grammar in Figure 4b.

[1]The knowledgeable reader may spot a similarity to the $\lessdot$ and $\gtrdot$ relations of operator precedence grammars, see Section 7 for a more thorough comparison.

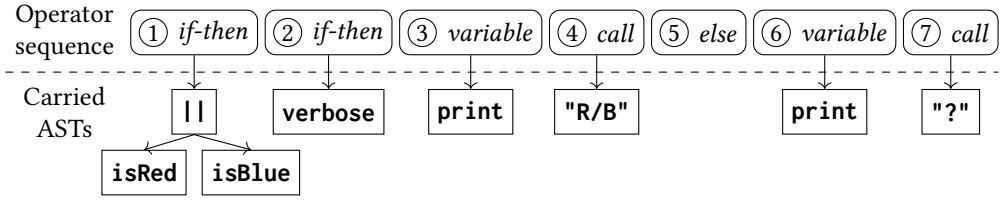


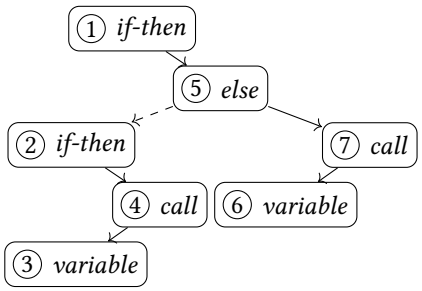
Fig. 5. A sequence of operators (rounded rectangles) to be passed to the grouper. Each operator may carry one or more previously constructed ASTs (AST-nodes are shown as rectangles with sharp corners) that the grouper will not touch; they pass through unchanged and do not affect the grouping. Note that **`if-then-else`** is in split form, **`else`** appears alone.

We leave the formal description of the grouping specification for the running example until Section 3; for now it is sufficient to know that function calls have higher precedence than **`if`**, and dangling else is resolved with longest match.

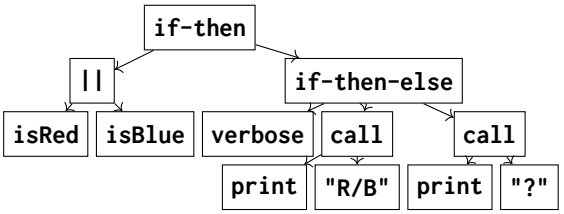
With these assumptions, the grouper takes the operator sequence in Figure 5 and produces the (single, unambiguous) grouped tree in Figure 6a. Note that **`if-then-else`** is still split; **`else`** appears as its own node. Note also that we consider **`else`** to be the parent of the **`if-then`** to which it belongs (denoted by the dashed arrow). This initially unintuitive formulation is needed later, for our guarantee of static resolvability.

To build intuition for this formulation, consider a split production as a sequence of its constituent operators; in this case, a sequence of length two: **`if-then`** and **`else`**. This sequence must be encoded in the grouped tree somehow. We choose to store it as a linked list, where the left child is the *previous* element of the list, hence **`else`** is the parent of **`if-then`**.

Reassembling split productions. However, most ASTs do not have the form we present in our grouped trees, e.g., **`if-then-else`** is typically *one* node with three children, and the fact that



(a) The tree produced by the grouper, with the carried ASTs omitted to reduce clutter. Note that **`if-then-else`** is still in split form; when constructing the final AST we merge **`else`** with the inner **`if`**, denoted by a dashed arrow. The choice to see **`else`** as the parent of **`if-then`** is initially surprising, but is necessary for our approach to static resolvability.



(b) The final AST after merging **`if-then-else`** and turning all operators into ASTs. Typically nullary operators (e.g., *variable*) are replaced by their carried AST, while other operators create a new node (e.g., *call* becomes **`call`**).

Fig. 6. The output of the grouper, first as a grouped tree, then as the final AST.

a function call appears syntactically like a postfix operator is unimportant. Because of this we post-process the grouped tree, merging split productions and generally transforming the tree into the shape of an AST. For example, in our OCaml case study we use this step to treat semicolons as list element separators or sequential composition, depending on the context in which they appear. For our running example the final AST can be seen in Figure 6b.

Static resolvability. Finally, before describing the formal semantics of our grouper in the next section, we briefly touch on the restrictions we later use to ensure static resolvability (in Section 4) in informal terms:

- All split productions must be encoded as linked lists where the left child is the previous element. Note that normal operators are trivial cases; lists of length one.
- We can partition the set of operators based on where they can be in a sequence representing a split production, either they *must* be first, or they *must not* be first.

The running example adheres to this: **`if-then`** is the left child of its paired **`else`**, no other split productions exist; and we have a clean partition, **`else`** must not be first, all other operators can only be first. Intuitively speaking these restrictions are quite mild:

- The choice of how to encode a sequence is arbitrary and easily adjusted.
- The partitioning of operators implies that there is an operator that signals the beginning of each production, which is typically already the case in a programming language, e.g., **`if`**, **`for`**, or **`let`**.

3 A FORMAL SEMANTICS FOR EXPLICIT AND IMPLICIT GROUPING

We now return to the matter of the grouping specification and the workings of our grouper. The entirety of our definitions can be found in Figure 7, the pieces of which we elaborate on in the upcoming paragraphs.

In this formal context, an operator is represented by a label o from a language-specified set Op. Our rules center around the concept of a partially grouped tree, denoted by e in the top-left of Figure 7. Note that g and p are more restrictive versions of e; the former represents a fully grouped tree, while the latter could also be an explicitly grouped sub-sequence. We use h to introduce *holes*, denoted by "_", which represent children yet to be determined. Note that holes can only appear as the direct child of a root node; g allows no holes. To cut down on redundancy in our rules we also use left and right focused frames, e.g., $\langle \_\ o]$ could represent $\langle \_\ o\rangle$, $\langle \_\ o\ \_\rangle$, or $\langle \_\ o\ g\rangle$, for some g.

Each e has an associated label, normally an $o \in Op$, except explicit grouping, which uses a distinct label $\pi \notin Op$. We refer to this label using op, in the top-right of Figure 7.

A grouping specification has four components: $bind$, $A_{\swarrow}$, $A_{\searrow}$, and A_{root}, seen middle-left in Figure 7. For example, the grouping specification for our running example, used to produce the grouped tree in Figure 6a in Section 2.3, is as follows:

$bind(o_1, o_2)$	*or*	*func-call*	*else*
or	$\leftarrow$	$\rightarrow$	$\rightarrow$
if-then	$\rightarrow$	$\rightarrow$	$\rightarrow$
else	$\rightarrow$	$\rightarrow$	$\rightarrow$

$$Op = \{\mathit{if\text{-}then}, \mathit{else}, \mathit{var}, \mathit{func\text{-}call}, \mathit{or}\}$$
$$A_{root}(o) = A_{\searrow}(o) = Op \cup \{\pi\}$$
$$A_{\swarrow}(o) = \begin{cases} Op \cup \{\pi\} & \text{if } o \neq \mathit{else} \\ \{\mathit{if\text{-}then}\} & \text{if } o = \mathit{else} \end{cases}$$

$bind$ defines the relative binding strength of two operators appearing adjacent. It can be used to express precedence (e.g., *func-call* has higher precedence than *or*, thus $bind(or, \mathit{func\text{-}call}) = \rightarrow$), associativity (e.g., *or* is left-associative, thus $bind(or, or) = \leftarrow$), and longest match (e.g., $\forall o \in Op.\ bind(o, \mathit{else}) = \rightarrow$).

A_{root}, $A_{\swarrow}$, and $A_{\searrow}$ define *shallow restrictions*, as sets of allowed labels, which are used to ensure that splittable productions reassemble correctly. For example, $A_{\swarrow}(\mathit{else}) = \{\mathit{if\text{-}then}\}$ means that the

Operators	$o, o_i \in Op$
Label sets	$s \in S \subseteq Op \cup \{\pi\}$
Incomplete expressions	$e ::= o \mid \langle \circ\, h\rangle \mid \langle h\, \circ\rangle \mid \langle h\ o\ h\rangle \mid (e\ \bar{e})$
Grouped expressions	$g ::= o \mid \langle \circ\, g\rangle \mid \langle g\, \circ\rangle \mid \langle g\ o\ g\rangle$
…or explicit grouping	$p ::= g \mid (e\ \bar{e})$
…or a hole	$h ::= g \mid \_$
Left-focused frame	$\langle \square\, \circ]$ $::= \langle \square\, \circ\rangle \mid \langle \square\ o\ h\rangle$
Right-focused frame	$[\circ\, \square\rangle$ $::= \langle \circ\, \square\rangle \mid \langle h\ o\ \square\rangle$

The label of an expression

$$op(o) = o$$
$$op(\langle \circ\, h\rangle) = o$$
$$op(\langle h\, \circ\rangle) = o$$
$$op(\langle h_1\ o\ h_2\rangle) = o$$
$$op((e\ \bar{e})) = \pi$$

$bind(o_1, o_2) \in \{-, \leftarrow, \rightarrow, \leftrightarrow\}$
$A_{\swarrow}(o) \subseteq Op \cup \{\pi\}$
$A_{\searrow}(o) \subseteq Op \cup \{\pi\}$
$A_{root} \subseteq Op \cup \{\pi\}$

$$groupings(o_1, s, o_2) = \begin{cases} - & \text{if } A_{\searrow}(o_1) \not\ni s \notin A_{\swarrow}(o_2) \\ \leftarrow & \text{if } A_{\searrow}(o_1) \ni s \notin A_{\swarrow}(o_2) \\ \rightarrow & \text{if } A_{\searrow}(o_1) \not\ni s \in A_{\swarrow}(o_2) \\ bind(o_1, o_2) & \text{if } A_{\searrow}(o_1) \ni s \in A_{\swarrow}(o_2) \end{cases}$$

$$\text{L-Edge}\ \frac{A_{\searrow}(o) \vdash p \Rightarrow g}{\bar{e}\ [\circ\ \_\rangle\ p \rightarrow \bar{e}\ [\circ\, g\rangle} \qquad \text{L-Mid}\ \frac{A_{\searrow}(o_1) \vdash p \Rightarrow g \quad groupings(o_1, op(p), o_2) \in \{\leftarrow, \leftrightarrow\}}{\bar{e}_1\ [o_1\ \_\rangle\ p\ \langle\ \_\ o_2]\ \bar{e}_2 \rightarrow \bar{e}_1\ [o_1\, g\rangle\ \langle\ \_\ o_2]\ \bar{e}_2}$$

$$\text{R-Edge}\ \frac{A_{\swarrow}(o) \vdash p \Rightarrow g}{p\ \langle\ \_\ o]\ \bar{e} \rightarrow \langle g\, o]\ \bar{e}} \qquad \text{R-Mid}\ \frac{A_{\swarrow}(o_2) \vdash p \Rightarrow g \quad groupings(o_1, op(p), o_2) \in \{\rightarrow, \leftrightarrow\}}{\bar{e}_1\ [o_1\ \_\rangle\ p\ \langle\ \_\ o_2]\ \bar{e}_2 \rightarrow \bar{e}_1\ [o_1\ \_\rangle\ \langle g\, o_2]\ \bar{e}_2}$$

$$\text{Unwrap}\ \frac{\pi \in S \quad A_{root} \vdash e\ \bar{e} \Rightarrow g}{S \vdash (e\ \bar{e}) \Rightarrow g} \qquad \text{LR-Steps}\ \frac{op(g) \in S \quad \bar{e} \rightarrow^* g}{S \vdash \bar{e} \Rightarrow g}$$

Fig. 7. Implicit and explicit grouping as a formal semantics. Note that there are two mutually recursive relations: small-step $\rightarrow$ for grouping in the top sequence and big-step $\Rightarrow$ for grouping sub-sequences (inside explicit grouping) and checking shallow restrictions.

left child of *else* must be precisely an *if-then*, which ensures that no *else* appears alone. Note also that these sets may include or exclude π, e.g., $\pi \notin A_{\swarrow}(else)$. This prevents explicit grouping in the middle of a split production, e.g., `(if a then b) else c` is not syntactically valid.

Finally, we turn our attention to the grouping rules at the end of Figure 7. From a formal viewpoint, grouping takes a sequence of ungrouped expressions (i.e., a sequence $\bar{e}$ where each h is a hole _) to a single g. Note that each occurrence of explicit grouping $(e\ \bar{e})$ in the input sequence contains a nested sequence. We define grouping through two mutually recursive relations, strongly inspired by operational semantics (both small-step and big-step):

- $\bar{e} \rightarrow \bar{e}$, which groups a single e towards an adjacent operator, making the former a child of the latter. Repeatedly applying this relation (through the reflexive closure $\rightarrow^*$) groups a sequence into a single tree.
- $S \vdash \bar{e} \Rightarrow g$ ensures that all nested sequences are fully grouped, either by recursing inside explicit grouping (rule Unwrap) or by applying $\rightarrow^*$ (rule LR-Steps).

Both $\rightarrow$ and $\Rightarrow$ take sequences of expressions to sequences of expressions (g denotes a subset of $\bar{e}$). The relation $\Rightarrow$ additionally has a set of allowed labels S. This is used to check the shallow restrictions in a context: $S \vdash \bar{e} \Rightarrow g$ states that $\bar{e}$ groups to g and is shallowly allowed by S. If $\bar{e}$ is an explicitly grouped sub-sequence (i.e., $(\ldots)$) this amounts to $\pi \in S$, otherwise $op(g) \in S$.

The rules for $\longrightarrow$ share a lot of structure, thus we explain one rule in detail and generalize from there. L-Mid groups a child to the *left* and applies when said child is somewhere in the *middle* of the sequence (as opposed to at the edge, i.e., first or last). We take a p (a fully grouped tree or an explicitly grouped sub-sequence), surrounded by two operators o_1 and o_2 and group it to the left, i.e., it becomes the right child of o_1. Everything else is left unchanged (the surrounding sequences $\bar{e_1}$ and $\bar{e_2}$ as well as the other operator o_2). The first premise checks the shallow restrictions implied by o_1 and ensures that the child is fully grouped (note that we use the fully grouped g on the right-hand side of the conclusion, not p). The second premise uses the helper function *groupings* to check that the grouping specification allows grouping left.

The definition of *groupings* considers three labels: the operator o_1 on the left, the potential child with label s, and the operator o_2 on the right. If only one direction is allowed by the shallow restrictions we return that, otherwise we defer to *bind*.

R-Mid then flips the direction (and exchanges $A_{\searrow}(o_1)$ with $A_{\swarrow}(o_2)$), while L-Edge and R-Edge drop the second premise since there is only one adjacent operator.

Returning to our running example, consider the operator sequence in Figure 5. The right-most derivation can be seen below. Note three things: 1) we abbreviate *if-then* as *if* and *func-call* as *fc*, 2) we highlight the child that was grouped with a gray background, and 3) we show the relevant parts of the grouping specification on the right, omitting the repeated $A_{\searrow}$ and $A_{\swarrow}$ to reduce clutter.

Derivation	Grouping Specification	
$\langle \mathit{if}\ \_\rangle\ \langle \mathit{if}\ \_\rangle\ \mathit{var}\ \langle \_\ \mathit{fc}\rangle\ \langle \_\ \mathit{else}\ \_\rangle\ \mathit{var}\ \langle \_\ \mathit{fc}\rangle$	$A_{\searrow}(o_1) \ni \quad s \quad \in A_{\swarrow}(o_2)$	$\mathit{bind}(o_1, o_2)$
$\xrightarrow{\text{R-Mid}} \langle \mathit{if}\ \_\rangle\ \langle \mathit{if}\ \_\rangle\ \mathit{var}\ \langle \_\ \mathit{fc}\rangle\ \langle \_\ \mathit{else}\ \_\rangle\ \langle\ \mathit{var}\ \ \mathit{fc}\rangle$	$\mathit{else} \ni \mathit{var} \in \mathit{fc}$	$\longrightarrow$
$\xrightarrow{\text{L-Edge}} \langle \mathit{if}\ \_\rangle\ \langle \mathit{if}\ \_\rangle\ \mathit{var}\ \langle \_\ \mathit{fc}\rangle\ \langle \_\ \mathit{else}\ \langle \mathit{var}\ \mathit{fc}\rangle\ \rangle$	$\mathit{else} \ni \quad \mathit{fc}$	
$\xrightarrow{\text{R-Mid}} \langle \mathit{if}\ \_\rangle\ \langle \mathit{if}\ \_\rangle\ \langle\ \mathit{var}\ \ \mathit{fc}\rangle\ \langle \_\ \mathit{else}\ \langle \mathit{var}\ \mathit{fc}\rangle\rangle$	$\mathit{if} \ni \mathit{var} \in \mathit{fc}$	$\longrightarrow$
$\xrightarrow{\text{L-Mid}} \langle \mathit{if}\ \_\rangle\ \langle \mathit{if}\ \langle \mathit{var}\ \mathit{fc}\rangle\ \rangle\ \langle \_\ \mathit{else}\ \langle \mathit{var}\ \mathit{fc}\rangle\rangle$	$\mathit{if} \ni \quad \mathit{fc} \quad \notin \mathit{else}$	
$\xrightarrow{\text{R-Mid}} \langle \mathit{if}\ \_\rangle\ \langle\ \langle \mathit{if}\ \langle \mathit{var}\ \mathit{fc}\rangle\rangle\ \ \mathit{else}\ \langle \mathit{var}\ \mathit{fc}\rangle\rangle$	$\mathit{if} \ni \quad \mathit{if} \quad \in \mathit{else}$	$\longrightarrow$
$\xrightarrow{\text{L-Edge}} \langle \mathit{if}\ \langle\langle \mathit{if}\ \langle \mathit{var}\ \mathit{fc}\rangle\rangle\ \mathit{else}\ \langle \mathit{var}\ \mathit{fc}\rangle\rangle\ \rangle$	$\mathit{if} \ni \mathit{else}$	
$\xRightarrow{\text{LR-Steps}} \langle \mathit{if}\ \langle\langle \mathit{if}\ \langle \mathit{var}\ \mathit{fc}\rangle\rangle\ \mathit{else}\ \langle \mathit{var}\ \mathit{fc}\rangle\rangle\rangle$	$\mathit{if} \in A_{\mathit{root}}$	

Note that the above derivation is not unique, e.g., we could group the first *func-call* earlier without changing the final result. However, all valid derivations for this input sequence give the same result; this input is unambiguous.

As an ambiguous example, consider the case where we redefine *bind* such that $\mathit{bind}(o, \mathit{else}) = \leftrightarrow$ for all o (instead of $\rightarrow$). This has the effect of removing longest match disambiguation for dangling else. As such, the above derivation is still valid with one minor change: the final application of R-Mid has $\leftrightarrow$ under $\mathit{bind}(o_1, o_2)$ instead of $\rightarrow$. More interestingly, we can now change that step from R-Mid to L-Mid:

Derivation	Grouping Specification	
$\langle if\ \_\rangle\ \langle if\ \_\rangle\ var\ \langle \_\ fc\rangle\ \langle \_\ else\ \_\rangle\ var\ \langle \_\ fc\rangle$	$\underline{A_{\searrow}(o_1) \ni s \in A_{\swarrow}(o_2)}$	$\underline{bind(o_1, o_2)}$
…The first four $\longrightarrow$ steps as before…		
$\xrightarrow{\text{L-Mid}} \langle if\ \langle if\ \langle var\ fc\rangle\rangle\ \rangle\ \langle \_\ else\ \langle var\ fc\rangle\rangle$	$if \ni if \in else$	$\leftrightarrow$
$\xrightarrow{\text{R-Edge}} \langle\ \langle if\ \langle if\ \langle var\ fc\rangle\rangle\rangle\ else\ \langle var\ fc\rangle\rangle$	$if \in else$	
$\xrightarrow{\text{LR-Steps}} \langle\langle if\ \langle if\ \langle var\ fc\rangle\rangle\rangle\ else\ \langle var\ fc\rangle\rangle$	$if \in A_{root}$	

Both of these derivations are now valid, and they give different final results, i.e., the input is ambiguous.

We are now ready to formally state the assumptions we use to ensure static resolvability. Our assumptions center around consistent handling of splittable productions; how we find the operators that belong to the same split production, and whether each operator must be first or not in such a production. We consider a child to be in the same split production as its parent if surrounding the child with explicit grouping would make a derivation of the given grouped tree impossible.

For example, if we surround the *if-then* in $\langle\langle$*if-then var*$\rangle$ *else var*$\rangle$ with explicit grouping we can no longer construct a valid derivation; there is no derivation $A_{root} \vdash (\langle$*if-then* $\_\rangle$ *var*$)\ \langle \_$ *else* $\_\rangle$ *var* $\Longrightarrow$ $\langle\langle$*if-then var*$\rangle$ *else var*$\rangle$. Equivalently: we can add explicit grouping without losing the derivation if *if-then* $\in A_{root}$ and $\pi \in A_{\swarrow}(else)$. In this case, the latter is not true.

The assumption we require is that only the left child can ever be in the same split production, and we can split *Op* in two disjoint sets based on whether an operator *must* appear first in a split production or *must not* appear first. Formally:

Definition 3.1. A grouping specification is *consistently split* if:

(1) $\forall o \in Op.\ \pi \in A_{\searrow}(o)$, i.e., all operators allow a parenthesized expression as their right child.
(2) $\forall o \in Op.\ A_{\searrow}(o) \subseteq A_{root}$, i.e., all operators that can be right children can also appear as the root of an expression.
(3) For each operator $o \in Op$ one of the following must apply:
   - $\pi \notin A_{\swarrow}(o) \vee A_{\swarrow}(o) \cap A_{root} = \emptyset$, i.e., parenthesizing the left child invalidates the current tree.
   - $\pi \in A_{\swarrow}(o) \wedge A_{\swarrow}(o) \subseteq A_{root}$, i.e., parenthesizing the left child does *not* invalidate the current tree.

Points 1 and 2 together imply that splittable productions must connect through their left children, while point 3 partitions *Op*: if the left child *must* be in the same split production, then the parent *must not* be first in the split production, and vice versa. Note that the grouping specification for our running example fulfills these criteria; it is *consistently split*.

We now turn to the task of proving that this assumption is sufficient for static resolvability.

4 STATIC RESOLVABILITY AND ITS MECHANIZED PROOF

In this section we present a proof that for the formalization in Section 3, given a grouping specification that is consistently split as defined in Definition 3.1, all ambiguous input is resolvable. In other words, for any input $\bar{e}$ that groups to some result g, there is some input $\bar{e}'$ which groups unambiguously to g. We begin by outlining the proof in Section 4.1 and then present the mechanization in Coq in 4.2.

$$
\begin{array}{rcl}
\texttt{flatten}(o) & = & o \\
\texttt{flatten}(\langle g_l\ \mathrm{o}\rangle) & = & \texttt{wrap?}(A_{\swarrow}(o), \texttt{flatten}(g_l))\ \langle \_\ \mathrm{o}\rangle \\
\texttt{flatten}(\langle \mathrm{o}\ g_r\rangle) & = & \langle \mathrm{o}\ \_\rangle\ \texttt{wrap?}(A_{\searrow}(o), \texttt{flatten}(g_r)) \\
\texttt{flatten}(\langle g_l\ o\ g_r\rangle) & = & \texttt{wrap?}(A_{\swarrow}(o), \texttt{flatten}(g_l))\ \langle \_\ o\ \_\rangle\ \texttt{wrap?}(A_{\searrow}(o), \texttt{flatten}(g_r))
\end{array}
$$

$$
\texttt{wrap?}(S, \bar{e}) = \begin{cases} (\bar{e}) & \text{if } \pi \in S \\ \bar{e} & \text{if } \pi \notin S \end{cases}
\qquad
\begin{array}{l}
\text{LOpen}\ \dfrac{}{lopen(\ \langle \_\ \mathrm{o}]\ )} \\[2ex]
\text{ROpen}\ \dfrac{}{ropen(\ [\mathrm{o}\ \_\rangle\ )}
\end{array}
\qquad
\text{PF-Single}\ \frac{\begin{array}{c}\neg lopen(e) \quad \neg ropen(e) \\ op(e) \in S \quad op(e) \neq \pi\end{array}}{S \vdash parform(e)}
$$

$$
\text{PF-()-L}\ \frac{\begin{array}{c}lopen(e') \quad \neg ropen(e') \\ op(e') \in S \quad \pi \in A_{\swarrow}(op(e')) \\ A_{root} \vdash parform(\bar{e})\end{array}}{S \vdash parform((\bar{e})\ e')}
\qquad
\text{PF-()-R}\ \frac{\begin{array}{c}\neg lopen(e') \quad ropen(e') \\ op(e') \in S \quad A_{root} \vdash parform(\bar{e})\end{array}}{S \vdash parform(e'\ (\bar{e}))}
$$

$$
\text{PF-()-LR}\ \frac{\begin{array}{c}lopen(e') \quad ropen(e') \\ op(e') \in S \quad \pi \in A_{\swarrow}(op(e')) \\ A_{root} \vdash parform(\bar{e}_l) \quad A_{root} \vdash parform(\bar{e}_r)\end{array}}{S \vdash parform((\bar{e}_l)\ e'\ (\bar{e}_r))}
\qquad
\text{PF-L}\ \frac{\begin{array}{c}lopen(e') \quad \neg ropen(e') \\ op(e') \in S \quad \pi \notin A_{\swarrow}(op(e')) \\ A_{\swarrow}(op(e')) \vdash parform(\bar{e})\end{array}}{S \vdash parform(\bar{e}\ e')}
$$

$$
\text{PF-LR}\ \frac{\begin{array}{c}lopen(e') \quad ropen(e') \\ op(e') \in S \quad \pi \notin A_{\swarrow}(op(e')) \\ A_{\swarrow}(op(e')) \vdash parform(\bar{e}_l) \quad A_{root} \vdash parform(\bar{e}_r)\end{array}}{S \vdash parform(\bar{e}_l\ e'\ (\bar{e}_r))}
$$

$$
\text{LM-LR-Steps}\ \frac{op(g) \in S \qquad \bar{e} \rightarrow^*_{LM} g}{S \vdash \bar{e} \Rightarrow_{LM} g}
\qquad
\text{LM-L-Edge}\ \frac{\begin{array}{c}\forall e \in \bar{e}\ .\ ropen(e) \\ A_{\searrow}(o) \vdash p \Rightarrow_{LM} g\end{array}}{\bar{e}\ [\mathrm{o}\ \_\rangle\ p \rightarrow_{LM} \bar{e}\ [\mathrm{o}\ g\rangle}
\qquad \ldots
$$

Fig. 8. Operations and properties used in the proof of static resolvability. The remaining rules of the relations $\Rightarrow_{LM}$ and $\rightarrow_{LM}$ are similar to the ones shown here.

4.1 Proof Outline

The intuition for our method of deciding resolvability is as follows. We start from some sequence $\bar{e}$ which is fully ungrouped (all occurrences of h is a hole _) and group it to some fully grouped expression g. We can then flatten g to a different fully ungrouped sequence $\bar{e}'$ with the maximal number of grouping parentheses inserted. If this sequence can be grouped in exactly one way (i.e., back to g), then the original input $\bar{e}$ is resolvably ambiguous. Conversely, if the fully parenthesized sequence $\bar{e}'$ *is* ambiguous, then there is *no* input that unambiguously groups to g (in other words, adding more parentheses does not disambiguate the sequence further).

The property we are after is that of static resolvability, meaning that *any* input $\bar{e}$ which groups to some g has a corresponding $\bar{e}'$ which unambiguously groups to g. Intuitively we show this by proving that for consistently split grouping specifications one can always add enough parentheses to allow unambiguous parsing.

Figure 8 contains the operations and properties used in the proof. We define an operation `flatten`(g) which takes a fully grouped expression to a fully ungrouped, fully parenthesized (where allowed) sequence. We also define the property *parform*($\bar{e}$), which is the shape of a sequence that has been `flatten`'d, or that is the result of (partially) grouping such a sequence under a consistently split grouping specification. The label set S in the relation is used to track that sub-expressions only appear where they are allowed. It will generally be omitted below, as it is uninteresting for the top-level expression (any label set containing the label of the root expression is valid). Note that *parform* only requires parentheses around ungrouped expressions – in particular, *parform*(g) holds even though a fully grouped expression g contains *no* parentheses. Also note that there is no rule with a non-parenthesized sequence to the right of an expression. This is because a consistently split grouping specification always allows parentheses around the right child of a node.

We show that the `flatten` operation results in a sequence that is in *parform* and that it gives a sequence that can be grouped to its origin:

Lemma 4.1. *For consistently split specifications, for all grouped expressions g, parform*(`flatten`(g))

Lemma 4.2. *For consistently split specifications, for all grouped expressions g, if op*$(g) \in S$ *then* $S \vdash$ `flatten`$(g) \Rightarrow g$

In order to simplify determining if a sequence can be grouped in exactly one way we introduce a variant of the grouping semantics that always groups the left-most expression in a sequence. We use $S \vdash \bar{e} \Rightarrow_{LM} g$ and $\bar{e} \rightarrow_{LM} \bar{e}'$ to denote such left-most grouping. Two of the rules are shown in Figure 8. The rules are same as in Figure 7, except that they use $\Rightarrow_{LM}$ and $\rightarrow_{LM}$ in their premises. The rules for $\rightarrow_{LM}$ additionally adds the requirement that every expression to the left of the expression being grouped ($\bar{e}$ in L-Edge and $\bar{e}_1$ in L-Mid and R-Mid) is a right-focused frame with a hole: $[\mathsf{o}\ \_\rangle$ (captured by the rule ROpen). This requirement ensures that a grouping step always performs the left-most grouping possible.

The left-most grouping semantics has two important properties. Firstly, it preserves *parform*:

Lemma 4.3. *Given a sequence $\bar{e}$ for which parform*$(\bar{e})$ *holds, if $\bar{e} \rightarrow_{LM} \bar{e}'$, then parform*$(\bar{e}')$.

Secondly, starting from a sequence in *parform*, the left-most grouping semantics is deterministic:

Lemma 4.4. *For consistently split specifications, if parform*$(\bar{e})$, $\bar{e} \rightarrow_{LM} \bar{e}_1$ *and* $\bar{e} \rightarrow_{LM} \bar{e}_2$, *then* $\bar{e}_1 = \bar{e}_2$.

Together, Lemma 4.3 and 4.4 give that any sequence in *parform* is unambiguously grouped to some g using left-most grouping:

Lemma 4.5. *For consistently split specifications, if parform*$(\bar{e})$, $S \vdash \bar{e} \Rightarrow_{LM} g_1$ *and* $S \vdash \bar{e} \Rightarrow_{LM} g_2$, *then* $g_1 = g_2$.

Finally, we show equivalence between the two versions of the semantics:

Lemma 4.6. $S \vdash \bar{e} \Rightarrow g$ *iff* $S \vdash \bar{e} \Rightarrow_{LM} g$.

The proof going from right to left is straightforward, as a left-most derivation is also a valid derivation in the original semantics. Going in the other direction is slightly more involved as grouping can happen in any order.

Using these lemmas we can show that the assumption of consistently split grouping specifications is sufficient for static resolvability:

Theorem 4.7 (Static Resolvability). *For consistently split specifications, given any input sequence $\bar{e}$ such that $S \vdash \bar{e} \Rightarrow g$, there exists some sequence $\bar{e}'$ such that for all g' where $S \vdash \bar{e}' \Rightarrow g'$, we have $g' = g$.*

Proof. For the input sequence $\bar{e}$ where $S \vdash \bar{e} \Rightarrow g$, we have that $S \vdash \texttt{flatten}(g) \Rightarrow g$ by Lemma 4.2. We also have *parform*($\texttt{flatten}(g)$) by Lemma 4.1. By Lemma 4.6 we have that $S \vdash \texttt{flatten}(g) \Rightarrow_{LM} g$ and by Lemma 4.5 that for any g', $S \vdash \texttt{flatten}(g) \Rightarrow_{LM} g'$ implies $g' = g$. Through Lemma 4.6 we can lift that final result back to the original semantics, giving us that for any g', $S \vdash \texttt{flatten}(g) \Rightarrow g'$ implies $g' = g$. □

4.2 Mechanization

We have mechanized the proof of static resolvability in Coq (~7000 lines), using the non-constructive TLC library [Charguéraud 2022]. The formulation closely follows the semantics from Figure 7 and the proof outlined above. The existing mechanization of Lemmas 4.1 through 4.5 is for an older version of the semantics, but we have proved equivalence between the new and old versions so that we can transfer the results to the semantics presented in Figure 7.[2]

The syntax for expressions presented in Figure 7 is represented as a single type `node`, corresponding to e extended with holes `_`. We use predicates to distinguish between the different kinds of expressions, e.g., `is_node` for nodes in e and `is_grouped` for nodes in g. We represent sets as functions to bool. The inductive relations representing semantics for grouping, `GSteps` for $\Rightarrow$ and `LRSteps` for $\rightarrow$, are parameterized over a predicate `Pre: list node -> Prop` that must hold for the prefix of the sequence being operated on. This allows us to express both $\Rightarrow$ and $\Rightarrow_{LM}$ using the same inductive definition. For example, the left-to-right case of Lemma 4.6 is formulated as below:

```
Lemma leftmost_to_gsteps :
  forall allowed input g,
    GSteps (Forall is_ropen) allowed input \[g] →
    GSteps (fun _ ⇒ true) allowed input \[g].
```

The conclusion of the theorem is equivalent to $S \vdash \bar{e} \Rightarrow g$, where `allowed` is S and `input` is $\bar{e}$ (`\[g]` is the syntax we use for singleton lists, to avoid a syntax clash with TLC). The predicate `is_ropen` in the premise holds for right-focused frames. Additionally, we define a version of the grouping semantics that ensures that we start from ungrouped (`fully_open`) input:

```
Definition InputStepsTo (ns : list node) (n :  node) :=
  Forall is_fully_open ns ∧ GSteps (fun _ ⇒ true) allowed_top ns \[n].
```

Definition 3.1 is translated in a straightforward manner:

```
Definition GroupingAssumption := forall o,
(* 1 *) allowed_right o par_label
(* 2 *) ∧ (forall o',  allowed_right o o' → allowed_top o')
(* 3 *) ∧ ((  allowed_left o par_label ∨
          (forall o',    allowed_left o o' ∨  allowed_top o'))
          ∨ (allowed_left o par_label ∧
              forall o',  allowed_left o o' → allowed_top o')).
```

The definitions of (un)ambiguity and resolvability are parameterised over types `input` and `output` and a relation `ParsesAs: input -> output`:

```
Definition Unambiguous (text : input) :=
  forall p p':  output,
    ParsesAs text p ∧ ParsesAs text p' →
    p = p'.
```

```
Definition Resolvable (text : input) :=
  forall p: output,
    ParsesAs text p →
    exists text',  ParsesAs text' p ∧
                   Unambiguous text'.
```

Finally, our main theorem is stated as below, which after expansion is equivalent to Theorem 4.7:

```
Theorem static_resolvability : forall input,
  GroupingAssumption → Resolvable InputStepsTo input.
```

[2]Formulating the proof without the old semantics is just a matter of proof engineering.

5 ADAPTING OCAML EXPRESSIONS TO USE A GROUPER

The bulk of our empirical evaluation consists of a modification to the canonical OCaml compiler to use our approach for parsing expressions. This section describes our overall approach to making such a modification, and the challenges encountered along the way.

The OCaml compiler uses Menhir [Pottier and Régis-Gianas 2005], an LR(1) parser generator, for its parser. We thus edit the Menhir grammar such that it parses operator sequences and hands them to our grouper as a part of the semantic action for each of the relevant productions.

The precedence and associativity of expressions in the original grammar is expressed through a mix of Menhir's precedence annotations and splitting expressions into multiple non-terminals. Our modification follows the process described in Section 2.3; we collect the various expression productions, then partition them based on edge recursion, and so on. However, the regular expression describing operator sequences is not quite as simple as the one in Figure 4b; OCaml has a number of operators that may not appear syntactically adjacent. For example, function application and unary negation may not appear adjacent; `1 - 2` should be interpreted as $1\ \langle \_ - \_ \rangle\ 2$, not $1\ \langle \_\ \_ \rangle\ \langle \text{-}\ \_ \rangle\ 2$.

Handling this is tedious but straightforward; we describe an operator sequence using a set of mutually recursive non-terminals named according to the last operator in the sequence. In practice we need four such non-terminals, each describing an operator sequence ending in:

- …a right-closed operator.
- …a right-closed operator *except* postfix semicolon (to handle an LR(1) conflict between postfix semicolon and infix semicolon).
- …a right-open operator.
- …a right-open operator *except* function application.

Each production consists of up to two symbols: possibly a non-terminal for the preceeding operator sequence, and a non-terminal choosing between one or more operators, paired to restrict adjacent operators correctly. The sole exception is infix semicolon; the corresponding productions parse the next operator as well, to avoid a reduce/reduce LR(1) conflict, mentioned above. For example, the grammar below shows the productions relating to semicolon and minus:

$$
\begin{array}{lll}
\mathit{ExprSeq}_{\mathtt{rclosed}} ::= \mathit{ExprSeq}_{\mathtt{rclosed}}\ \mathit{Expr}_{\mathtt{semi}} & & \text{(Postfix \texttt{;})} \\
\quad \mid \mathit{ExprSeq}_{\mathtt{rclosed}}\ \mathit{Expr}_{\mathtt{semi}}\ \mathit{Expr}_{\mathtt{Atom}} & & \text{(Infix \texttt{;}, then a nullary operator)} \\
\mathit{ExprSeq}_{\mathtt{ropen}} ::= \mathit{ExprSeq}_{\mathtt{rclosed}}\ \mathit{Expr}_{\mathtt{semi}}\ \mathit{Expr}_{\mathtt{Prefix}} & & \text{(Infix \texttt{;}, then a prefix operator)} \\
\quad \mid \mathit{ExprSeq}_{\mathtt{ropen,\ no\ app}}\ \mathit{Expr}_{\mathtt{minus}} & & \text{(Prefix \texttt{-})} \\
\quad \mid \mathit{Expr}_{\mathtt{minus}} & & \text{(Prefix \texttt{-}, beginning of sequence)}
\end{array}
$$

Note that prefix `-` either starts the operator sequence (the last production), or the previous operator must not be a function application, as denoted by $\mathit{ExprSeq}_{\mathtt{ropen,\ no\ app}}$.

5.1 Challenges

As expected of any more complicated language, OCaml is not without its challenges to describe. This section lists the most significant challenges we encountered throughout the course of development.

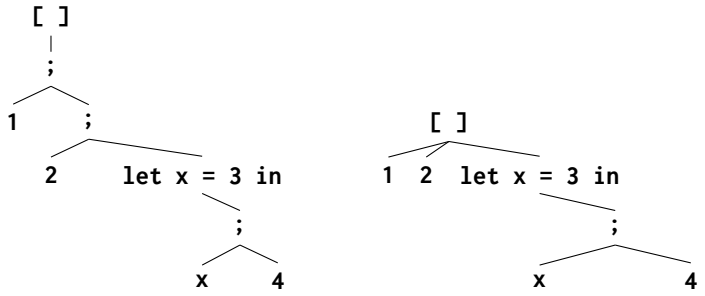
Semicolons as Sequential Composition and a Separator. A semicolon in OCaml code can have one of two meanings:

- It could be sequential composition, i.e., an infix operator such that `a; b` evaluates `a` for its side-effects, then evaluates `b` and returns its result.
- It could be a separator in a list or a record, e.g., `[1; 2; 3]` is a list of three elements.

Furthermore, sequential composition is not the lowest precedence operator. For example, `let` has lower precedence, thus `[1; let x = 2 in x; 3]` is a list of two elements, not three. The canonical

```
[1; 2; let x = 3 in x; 4]
```

(a) Example input



(b) Concrete syntax tree (c) Abstract syntax tree

Fig. 9. Turning sequential composition into list element separators, before (concrete syntax) and after (abstract syntax).

OCaml parser has specialized non-terminals for expressions in these two types of context. We solve it slightly differently: we group expressions exactly the same, always treating semicolon as sequential composition when grouping, and then treating them differently when converting the grouped tree to an AST. When we are in a list context we translate sequential composition along the top of the expression to list elements, otherwise to sequential composition as normal. See Figure 9 for an example.

An important point here is why this is at all possible and correct: the OCaml grammar is written such that the precedence level of semicolon is consistent, regardless of whether it is sequential composition or an element separator in a list. Our implementation makes this explicit. For records the story is a bit more complicated, see Section 5.1.

Finally, OCaml also allows terminating an expression with semicolon, to mirror its use in more imperative languages. In our system this takes the shape of a postfix operator. However, due to a technicality of the Menhir grammar this operator cannot appear as the left child of an operator. The examples below[3] use the infix operator **|>**, where **x |> f** is alternate syntax for **f x**, i.e., **f** applied to **x**. The comment on the right is the result of attempting to evaluate the expression using the canonical OCaml compiler:

```
1 |> fun x -> x          (* 1 *)
1; |> fun x -> x         (* Error: Syntax error at |> *)
let f x = x in 1; |> f   (* Error: Unbound value f *)

(* The last expression is seen as follows: *)
(let f x = x in 1;) |> f
```

This behavior is surprising because both **;** and **|>** have higher precedence than **let**, yet **let** ends up binding stronger than **|>** since otherwise **;** would be the left child of **|>**, which is not allowed according to the OCaml grammar.

Though unconventional, this behavior is nonetheless simple to express in our system; we merely add the shallow restriction that postfix semicolon must never be a left child of any node.

[3]The third expression is a simplified version of one appearing in the OCaml source code.

Representing **`if-then-else`** *as a Split Production.* We split **`if-then-else`** as presented earlier (e.g., Section 2.3): a prefix operator **`if-then`** and an infix operator **`else`**, where **`else`** may only appear as the right child of **`if-then`**. To implement longest match we define $bind(\mathit{if\text{-}then}, \mathit{else}) = \rightarrow$. If we instead want to keep the dangling else ambiguity, we define $bind(\mathit{if\text{-}then}, \mathit{else}) = \leftrightarrow$, with no other changes necessary.

Additionally, as an interesting sidenote, this approach of applying precedence to **`if-then-else`** does not quite preserve the language of a naive grammar without precedence[4]. For example, **`if a then b; c else d`** is not syntactically valid in OCaml, since **`;`** has lower precedence than **`if`**, even though a naive grammar would recognize this expression[5].

Representing **`match`** *as a Split Production.* OCaml has three constructs for pattern matching: **`match`**, **`function`**, and **`try`**. Each of these has their own syntax for how to start a match, but share syntax for the list of cases. We split these into three prefix operators (**`match-with`**, **`try-with`**, and **`function`**) and one infix operator (**`match-arm`**). Additionally, we state that the left child of **`match-arm`** must be precisely one of these four operators. Note that this is quite similar to the representation of **`if-then-else`**: each time one of these split productions is parsed it is represented by one prefix operator and some number of infix operators. There are two major differences:

- Match has three options for which prefix operator is used: **`match-with`**, **`try-with`**, or **`function`**, whereas if only has **`if-then`**.
- **`if-then-else`** can have zero or *one* infix operator while match can have zero or *more*, due to the shallow restrictions chosen.

Splitting Records. OCaml records, similarly to lists, use semicolon as a separator between fields. Each field has an identifier, a type, and a value, but only the identifier is required:

```
let pun = true in
{ a: int = 4; b = 2; pun }

(* is the same as *)

let pun = true in
{ a: int = 4; b: int = 2; pun: bool = pun }
```

Once again we need precedence information to distinguish separating semicolons from sequential composition, thus we wish to parse the entire record content as a single expression, then turn the grouped tree into the appropropriate record. Unfortunately, the syntax of an ordinary OCaml expression has two distinctions from the syntax of a record:

- The infix comparison operator, **`=`**, has higher precedence than some other operators that could appear in the value of a field, and it also associates to the left. We can work around this by post-processing the grouped tree to rearrange it such that the left-most **`=`** is higher in the tree.
- Type constraints do not have the form **`expr: type`**, rather they are written **`(expr: type)`**, where the surrounding parentheses are required.

The latter issue is not easily worked around. We believe that it should be possible to exploit the large overlap between the syntax of types and the syntax of expressions to use our approach to parse both at the same time, instead of separately, but leave that for future work.

Fortunately, explicit type annotations on record fields are rare: only two files in the OCaml compiler use them, and then only in the testsuite to make sure they are supported properly. After

[4]This is also true for the approach used by the canonical OCaml compiler
[5]In fact, the expression is unambiguous in the naive grammar

removing these occurrences from the files in question our version of the compiler parses them the same as the canonical version.

6 IMPLEMENTATION AND EVALUATION

To evaluate our approach in a practical setting, we have implemented a grouper following the semantics in Section 3, as a library. We have then used this library in two larger projects:

- First, we have adjusted the parser in the canonical OCaml compiler such that expressions use our approach (c.f. Section 5). Note that OCaml is an expression-focused language, i.e., syntax that would be statements or definitions in other languages are typically included in expressions instead, thus this change is more significant than it may initially appear.
- Second, we have implemented a parser generator from scratch for the purpose of writing syntax for domain-specific languages, where the grammars may be resolvably ambiguous.

This section first describes the implementation in broad strokes, followed by the two use cases.

6.1 Grouping as a Library

The grouper is implemented as a library consisting of ~800 lines of MCore, the OCaml-like core language of the Miking project [Broman 2019], which we compile to OCaml. The interface is given in a streaming style, i.e., each operator is supplied one at a time, instead of as the complete sequence. To achieve efficiency even for highly ambiguous inputs we do not construct all valid trees directly, instead we compute a shared packed parse forest (SPPF) representing the set of valid trees.

The interface uses generalized algebraic datatypes to signal whether an operator is left and/or right open. Technically, this is not necessary for the implementation, but it provides additional safeguards when using the library. For example, if the type of an operator is **`Operator l r`**, then an infix operator should have type **`Operator LOpen ROpen`**, while a prefix operator should have type **`Operator LClosed ROpen`**.

The grouping specification is given in the form of five functions:

- **`topAllowed`** : $\forall l, r.$ **`Operator`** $l\ r \rightarrow$ **`Bool`**, corresponding to A_{root}.
- **`leftAllowed`** : $\forall l, l', r.$ **`Operator LOpen`** $r \times$ **`Operator`** $l'\ r \rightarrow$ **`Bool`**, corresponding to $A_{\swarrow}$.
- **`rightAllowed`** : $\forall l, r, r'.$ **`Operator`** l **`ROpen`** $\times$ **`Operator`** $l\ r' \rightarrow$ **`Bool`**, corresponding to $A_{\searrow}$.
- **`groupingsAllowed`** : $\forall l, r.$ **`Operator`** l **`ROpen`** $\times$ **`Operator LOpen`** $r \rightarrow$ **`AllowedDirection`**, corresponding to *bind*. **`AllowedDirection`** is here an enumeration with one option for each element in $\{-, \leftarrow, \rightarrow, \leftrightarrow\}$.
- Finally, for implementation reasons, checking if $\pi \in A_{\swarrow}(o)$ and $\pi \in A_{\searrow}(o)$ is given its own function: **`parenAllowed`** : $\forall l, r.$ **`Operator`** $l\ r \rightarrow$ **`AllowedDirection`**.

The library also provides a function for computing the resolutions of an ambiguous program. However, the number of alternatives in a particularly ambiguous expression can be $O(n!)$ in the input length, thus the library computes *partial* resolutions that only resolve the top node in the AST, thus producing $O(n)$ resolutions. This means that a user might encounter an ambiguity error, follow one of the suggested resolutions, then be given a new, smaller, ambiguity error. In practice this appears to be a minor nuisance at worst and helpful in reducing clutter at best. However, there should be no practical obstacle to implementing complete resolutions and falling back to partial resolutions if the input is too ambiguous, but this is presently not implemented.

6.2 Re-implementing Expressions in OCaml

We have used our grouper library to describe OCaml's expression language, including integrating it into the canonical OCaml compiler.[6] We do this in two steps: first we follow the canonical parser closely, then we drop some disambiguation choices made in the canonical compiler. The first step is quite accurate: when parsing source of the OCaml compiler itself we produce identical parse trees (including location information) for 2281/2283 files. The second step has two variations: one with dangling-else, and one with both dangling-else and no longest match disambiguation for **`match`**. All in all, we have four versions of the compiler:

`original`: The unmodified canonical OCaml compiler. We base all our experiments on version 4.14.0, git commit cd105d9.

`unamb`: The compiler using our approach for expressions that follows the canonical compiler as closely as possible.

`ambif`: The compiler with our approach and the dangling else ambiguity.

`ambmatch`: The compiler with our approach, the dangling else ambiguity, and no longest match disambiguation for nested **`match`**.

Our formulation of the OCaml expression language adheres to the assumption presented in Definition 3.1 on page 12, thus we know that the ambiguities introduced in the **`ambif`** and **`ambmatch`** versions of the compiler are resolvable. Note also that the changes between **`unamb`**, **`ambif`**, and **`ambmatch`** are trivial; two constants are changed in total (from **`GRight`** to **`GEither`**).

To evaluate the overhead of our approach we parse all source files (**`.ml`** and **`.mli`**) in all packages we could download through Opam, the OCaml package manager[7]; approximately 1,000,000 files over 1533 packages. However, to achieve a fair comparison we only consider the compiler versions that we have fully bootstrapped and compiled to machine code. This includes **`original`**, **`unamb`**, and **`ambif`**, but not **`ambmatch`**.

This is because sometimes the original author of a file relied on implicit disambiguation, some of which we remove in **`ambif`** and **`ambmatch`**. These files must thus be updated to be explicit for the bootstrap to work. For **`ambif`** the task is simple, only 10 files require updates, but for **`ambmatch`** it is more work; at least 139 files, typically with multiple nested matches requiring updates.

Because of this we have fully bootstrapped versions of **`original`**, **`unamb`**, and **`ambif`**, but only a bytecode version of **`ambmatch`**. This is enough to test correctness, but to ensure a fair performance comparison we only use the versions that are compiled to machine code.

All in all our implementation consists of three parts:

- The syntax of operator sequences as Menhir productions, consisting of ~80 lines describing operator sequences, and ~250 lines describing the operators themselves.
- Supporting code supplying shallow restrictions and how to re-merge split productions into the normal AST produced by the parser, consisting of ~450 lines.
- ~1300 lines of generated code describing the *bind* function based on a rather smaller precedence table.

6.2.1 Testing the Accuracy of **`unamb`**. The OCaml compiler already provides a means to examine the effects of changes to the parser: the compiler can output debug representations of each parsed AST, including source code location information. We run this on each file in the OCaml compiler repository, comparing the results produced by both **`original`** and **`unamb`**.

[6] https://github.com/ocaml/ocaml
[7] https://opam.ocaml.org/

In summary, we compare the ASTs for a total of 2283 files and produce the exact same output for all but 2 of them. This includes files that fail to parse, typically part of the testsuite. The files we parse incorrectly test more exotic syntactic features, as described in Section 5.1.

We have not compared AST representations across the packages in Opam, however, out of the 1,065,363 files for which parsing succeeds with **`original`**, all but 433 succeed with **`unamb`** (i.e., ∼0.04 % failure rate), suggesting that these features are indeed quite rarely used.

6.2.2 Benchmarks. To evaluate the overhead, we compare parse times across source code available via Opam for the canonical compiler (**`original`**) and our two compiled versions (**`unamb`** and **`ambif`**).

Our approach is designed for user-authored code. For parsing of computer generated data (e.g., JSON and the like) there is no benefit to resolvable ambiguity, thus a more traditional approach using unambiguous grammars is a better fit. For this reason all our benchmarking is focused on user-authored code.

We use Opam to download all available packages, use the debug log to extract the location of the downloaded source code, then parse all **`.ml`** and **`.mli`** files present.

To provide an accurate baseline for our comparisons we must measure only the time taken to parse a file, not the startup cost of launching the OCaml compiler or other unrelated costs. Doing this correctly in a large complicated project is non-trivial from an engineering point of view. Fortunately, the OCaml compiler already has a flag for doing this: **`-dprofile`**. Unfortunately, this has two notable threats to the validity of our analysis:

- Timing information is only presented when parsing succeeds. We argue that this is the most important case, since it is typically acceptable to wait a little longer for an error message, as long as it truly is only a *little* longer. For this reason we additionally measure the complete execution time taken in case of parse failure. Of course, this new measurement contains unrelated overheads and is thus not useful for comparisons, instead, we use it to show that our approach is at least not prohibitively slow in case of parse failure. The slowest time recorded in this way was 0.077 s, i.e., a user does not have to wait long for a parse error.
- The presented resolution is quite low in relation to common parse-times, 0.001 s, while a significant number of files parse in less time. However, if the overhead of our approach is noticeable to a user, then it would also be noticeable on measurements with this resolution, thus this is sufficient to find a bound on the overhead for our purposes.

The latter point also means that a significant number of files are reported as parsing in 0 s, which affects speedup/slowdown calculations. We will return to this point.

Each file is parsed five times with each compiler version; we use the median in our plots and analysis. We choose median instead of mean due to the low parse-times in general in combination with the low precision; small variations in parse-time, cache-effects, and spikes in the underlying system can easily cause outliers.

The benchmarks were run over a weekend on a computer running Ubuntu 20.04 with an Intel Core i7-8550U (4 cores) and 16 GiB of RAM. The total running time of the experiment was ∼58 h.

Note that the OCaml parser is quite fast, e.g., 99 % of files parse in at most 0.012 s (c.f. 0.018 s for **`unamb`** and **`ambif`**). Table 1 shows these metrics.

Note that the metrics for absolute slowdown consider all files, while relative slowdown ignores files where **`original`** finished faster than 0.001 s. As mentioned earlier, these are reported as 0 s, and we cannot divide by zero. Roughly 45 % the parsed files are reported as 0 s by **`original`**. Of those files, 87 % are also reported as 0 s by **`unamb`** and **`ambif`**, while the remaining files are parsed in at most 0.006 s.

Note also that the highest relative slowdown is 6×, for both **`unamb`** and **`ambif`**. While this may initially seem significant, closer examination of the data shows this to be a slowdown from 0.001 s

Table 1. Summary statistics when comparing parse times across the three compiler versions. We parse each file five times, record the median, then compute the above metrics.

	Parse time (s)			Relative slowdown		Absolute slowdown (s)	
	`original`	**`unamb`**	**`ambif`**	**`unamb`**	**`ambif`**	**`unamb`**	**`ambif`**
Worst case	0.486	0.720	0.720	6×	6×	0.234	0.234
99-percentile	0.012	0.018	0.018	2×	2×	0.006	0.006
Mean	0.002	0.002	0.002	1.39×	1.39×	0.001	0.001

to 0.006 s, for two files. The individual parsetimes for these files vary between 0.002 s and 0.007 s, i.e., relatively high variance (78 % of files have no recorded variance at all).

Figure 10 compares parsetimes for individual files between our approach and **`original`**, both as fractions (left) and differences (right).

Note that the left subfigure is highly clustered around 1 and 2. This is due to the aforementioned low resolution of the measurements reported by **`-dprofile`**.

In the context of parsing user-authored code for compilation this seems an acceptable amount of overhead; after all, parsing is typically a rather insignificant portion of the total compilation time.

6.3 A Parser Generator for DSLs using Resolvable Ambiguity

We have implemented a parser generator for DSL development, also using the library presented in Section 6.1, along with a table-driven LL(1) parser. The parser generator is self-hosted, in the sense that its DSL for describing syntax is described in itself. Each production described also defines

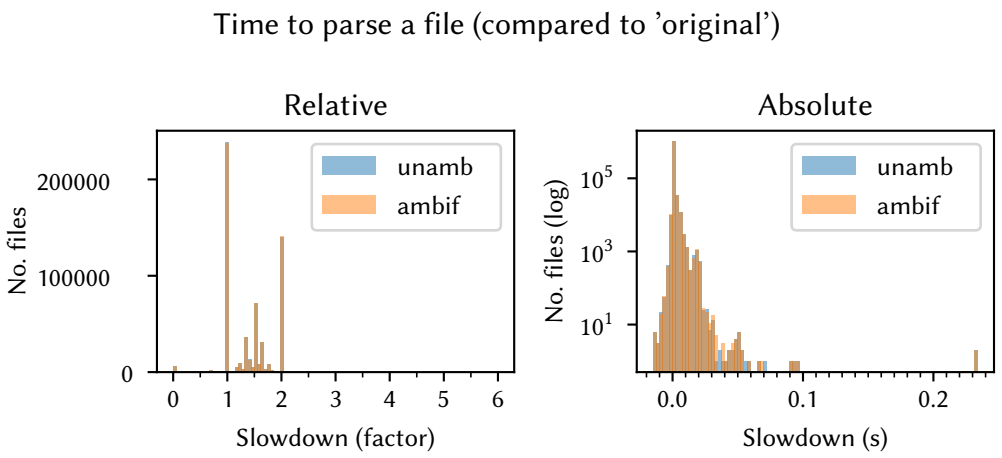


Fig. 10. Histograms over time taken to parse a file with our approach compared with **`original`**, both in relative terms (left) and absolute terms (right). The former is reported as a slowdown factor, e.g., "2" on the horizontal axis means the parsetime was twice that of **`original`**. The latter is an absolute measure, e.g., "0.1" on the horizontal axis means that parsing took 0.1 s longer than **`original`**. Note that the vertical axis is log-scale in the right figure. The left graph extends to 6× because of two outliers, i.e., the bar is too short to see.

an AST node; the types for the AST are auto-generated, along with various convenient helper functions for AST manipulation.

The tool supports partial precedence and associativity, automatically translated to an appropriate *bind* definition, using partial precedence tables. For example, the following example describes expressions with explicit grouping, basic arithmetic, and comparators, with standard precedence and associativity except for the comparators, which are left ambiguous relative to each other:

```
type Exp {
  grouping = "(" ")",
}

prod Int: Exp = val:Integer
infix left Plus: Exp = "+"
infix left Times: Exp = "*"
infix Equal: Exp = "="
infix NotEqual: Exp = "!="

precedence {
  Times;
  Plus;
  Equal NotEqual;
} except {
  Equal ? NotEqual;
}
```

Operators appearing higher in the precedence table have higher precedence. Note also the **`except`** block, which states that this precedence table does not affect grouping between **`Equal`** and **`NotEqual`**. Using this language definition **`1 + 2 * 3 = 7`** parses the same as **`(1 + (2 * 3)) = 7`**, while **`2 = 1 + 1 != 3`** gives an ambiguity error:

```
FILE "example" 1:0-1:14 ERROR: Ambiguity error
  2 = ( 1 + 1 != 3 )
  ( 2 = 1 + 1 ) != 3
```

The translation from precedence and associativity to *bind* is straightforward, for each pair of operators o_1 and o_2:

- If o_1 has higher precedence than o_2, define $bind(o_1, o_2) = \leftarrow$ and $bind(o_2, o_1) = \rightarrow$.
- If o_1 and o_2 have the same precedence and associativity (e.g., both left-associative) define $bind(o_1, o_2) = bind(o_2, o_1) = \leftarrow$. Right-associativity merely flips the arrow.
- In all other cases, define $bind(o_1, o_2) = bind(o_2, o_1) = \leftrightarrow$.

The tool presently provides no high-level access to splittable productions, even though the underlying library supports it, since there are some open questions on how to do this in a user-friendly way. Ideally, a language designer need not consider splittable productions at all, the tool should handle this automatically, but this we leave for future work.

7 RELATED WORK

Prior work. We previously introduced and defined the term *resolvable ambiguity*, along with a language formalism supporting resolvable ambiguities [Palmkvist et al. 2021]. This language formalism is somewhat more expressive than what we present in this paper; explicit grouping is not the only mechanism of ambiguity resolution. For example, reintroducing optional tokens, such as semi-colons, can potentially resolve an ambiguity. The difference in expressive power compared to this paper is notable, however, in practice most interesting ambiguities tend to be resolvable

through explicit grouping, thus we find the tradeoff acceptable, since our previous approach did not provide static guarantees, only property-based testing.

Formalisms of Precedence. Our *bind* function (c.f. Section 3) bears a strong resemblence to the precedence relation of operator-precedence grammars (OPGs) [Floyd 1963]: both relate tokens (operators in our case) with a non-symmetric relation that is used for parsing. We can create an OPG precedence matrix from a *bind* definition by filling each cell as follows (where o_1 and o_2 denote the relevant pair of operators):

- If o_1 is right-open and o_2 is left-open, examine $bind(o_1, o_2)$ and use the following equivalences: $- = \emptyset$, $\rightarrow = \{\lessdot\}$, $\leftarrow = \{\gtrdot\}$, and $\leftrightarrow = \{\lessdot, \gtrdot\}$.
- If o_1 is right-closed and o_2 is left-closed, use $\emptyset$.
- If o_1 is right-open and o_2 is left-closed, use $\{\lessdot\}$.
- If o_1 is right-closed and o_2 is left-open, use $\{\gtrdot\}$.

Note that $\leftrightarrow$ does not produce a singleton set (since it represents an ambiguity), and that $\doteq$ is not used (since we handle mixfix operators either before the grouper, or through splittable productions). From this perspective our grouper implementation could be seen as a parser that handles a subset of OPGs (no $\doteq$) but extended to allow ambiguity (which is not technically allowed in an OPG). However, shallow restrictions are not easily translated; OPG parsing only considers tokens in its decision-making, not any interspersed and already reduced productions, i.e., the identity of a potential direct child is not considered. OPGs and our groupers are thus strongly related, though neither describes a superset of the other.

Aasa [1995] presents a formal definition of precedence, independent of any parsing algorithm. However, the definition requires a total precedence relation, which precludes the possibility of ambiguous precedence, which we advocate in this paper.

The operator ambiguity removal patterns of Afroozeh et al. [2013] present a different means of expressing precedence through grammar rewrites. These rewrites frequently increase the size of the grammars used significantly, however, their later work [Afroozeh and Izmaylova 2015] avoids this issue through the use of data-dependent grammars. The rewrites and data-dependent rules are relatively subtle to properly model precedence in nested expressions compared to the local rewrite rules used in our approach, and their precedence relation requires at least transitivity, which is more restrictive than ours.

Danielsson and Norell [2011] provide an approach for writing grammars containing mixfix operators, and also allow ambiguous grammars, choosing only to reject ambiguous programs. However, while their precedence relation is quite unrestricted, e.g., not requiring transitivity or totality, their notion of a *precedence correct* expression on the other hand is. For example, in a language where `+` and `*` have no defined relative precedence the expression `1 + 2 * 3` is syntactically invalid, as opposed to valid but ambiguous.

Other approaches to ambiguity. Parsing expression grammars [Ford 2004] handle ambiguity in grammars differently: the formalism does not allow it in the first place. However, this naturally does not allow the beneficial effects of ambiguity we advocate in this paper.

Parser generators in common use tend to be based on unambiguous CFG subclasses, e.g., LL(k), LR(k), or LR(*). Menhir in particular, which is the parser generator we integrate with in Section 6.2 uses LR(1) [Pottier and Régis-Gianas 2005]. Others do not fit neatly in the Chomsky hierarchy, but still produce a single parse tree per parse, e.g., LL(*) [Parr and Fisher 2011] and ALL(*) [Parr et al. 2014]. There are also general parsers in use that can handle ambiguous grammars, producing multiple parse trees or other forms of parse forests, e.g., GLR [Lang 1974], GLL [Scott and Johnstone 2010], and Earley [1970].

SugarJ [Erdweg et al. 2011] enables composition of syntactic language extensions, which can naturally lead to ambiguities. These ambiguities are resolvable through essentially qualifying which extension is used for a given ambiguous expression. This is a good approach when the ambiguity concerns *what* constructs are involved (e.g., an XML or HTML literal) rather than *how* the constructs connect (implicit grouping), which is the focus of this paper.

Integrated development environments (IDEs) often need to parse partially correct programs, which easily leads to ambiguity. For example, SMIE [Monnier 2020] also uses an OPG-like parser in combination with a more traditional parser, including the trick of splitting **`if-then-else`**.[8]

The detection of ambiguity in contexct-free grammars, though undecidable in general [Cantor 1962], has nonetheless been explored in great detail. Examples include linguistic characterizations and regular language approximations [Brabrand et al. 2007], using SAT-solvers [Axelsson et al. 2008], and other conservative approaches [Schmitz 2007]. An overview and additional approaches can be found in the PhD thesis of Basten [2011].

In summary, our approach differs from the state of the art in that it combines the following properties in one approach:

- A solution to the ambiguity problem that does not outright forbid ambiguity.
- All ambiguities are statically known to be resolvable.
- The approach is flexible and expressive enough to describe a large, complicated, pre-existing language (OCaml's expression language).
- The overhead of using our approach is minor.

8 CONCLUSION

In this paper we give—for the first time—a solution to the problem of statically resolvable ambiguity. More specifically, our solution is based on the idea of having a *grouper* that works in tandem with a standard parser, where the grouper consumes operator sequences and produces grouped trees. We show that our solution can both be implemented efficiently and proven correct within Coq, given some mild assumptions. Moreover, the approach is expressive enough to handle an essential part of the standard OCaml parser, demonstrating practical feasibility. As for future work, we envision the approach being designed and implemented in a complete compiler toolchain, where a language designer develops a language in a high-level syntax specification, thus hiding internal details of splittable productions from the user.

ACKNOWLEDGMENTS

We thank the reviewers for their excellent comments. This project is financially supported by the Swedish Foundation for Strategic Research (FFL15-0032).

REFERENCES

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

Ali Afroozeh and Anastasia Izmaylova. 2015. One Parser to Rule Them All. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!) (Onward! 2015)*. Association for Computing Machinery, New York, NY, USA, 151–170. https://doi.org/10.1145/2814228.2814242

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

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

[8] https://www.gnu.org/software/emacs/manual/html_node/elisp/SMIE-Tricks.html

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

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

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

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

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

Arthur Charguéraud. 2022. The TLC Coq Library.

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

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

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

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

Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. 2011. SugarJ: Library-Based Syntactic Language Extensibility. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '11)*. Association for Computing Machinery, New York, NY, USA, 391–406. https://doi.org/10.1145/2048066.2048099

Robert W. Floyd. 1963. Syntactic Analysis and Operator Precedence. *J. ACM* 10, 3 (July 1963), 316–333. https://doi.org/10.1145/321172.321179

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

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

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

Stefan Monnier. 2020. SMIE: Weakness Is Power! *The Art, Science, and Engineering of Programming* 5, 1 (June 2020), 1:1–1:26. https://doi.org/10.22152/programming-journal.org/2021/5/1

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

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 2021)*. Association for Computing Machinery, New York, NY, USA, 153–164. https://doi.org/10.1145/3446804.3446846

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

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

François Pottier and Yann Régis-Gianas. 2005. The Menhir Parser Generator.

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

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

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

The dafny-lang community. 2022. Dafny Documentation. https://dafny-lang.github.io/dafny/DafnyRef/DafnyRef.html.

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

Received 2022-07-07; accepted 2022-11-07