Safe Specification of Operator Precedence Rules

Ali Afroozeh[1], Mark van den Brand[3], Adrian Johnstone[4], Elizabeth Scott[4], and Jurgen Vinju[1,2]

[1] Centrum Wiskunde & Informatica, 1098 XG Amsterdam, The Netherlands
[2] INRIA Lille Nord Europe, France
ali.afroozeh@cwi.nl, jurgen.vinju@cwi.nl
[3] Eindhoven University of Technology, NL-5612 AZ Eindhoven, The Netherlands
m.g.j.v.d.brand@tue.nl
[4] Royal Holloway, University of London, Egham, Surrey, TW20 0EX, UK
a.johnstone@rhul.ac.uk, e.scott@rhul.ac.uk

Abstract. In this paper we present an approach to specifying operator precedence based on declarative disambiguation constructs and an implementation mechanism based on grammar rewriting. We observed a problem with existing generalized context-free parsing and disambiguation technology: generating a correct parser for a language such as OCaml using declaraive precedence specification is not possible without resorting to some manual grammar transformation. Our approach provides a fully declarative solution to operator precedence specification for context-free grammars, is independent of any parsing technology, and is safe in that it guarantees that the language of the resulting grammar will be the same as the language of the specification grammar. We evaluate our new approach by specifying the precedence rules from the OCaml reference manual against the highly ambiguous reference grammar and validate the output of our generated parser.

1 Introduction

There is an increasing demand for front-ends for programming and domain-specific languages. We are interested in parser generation technology that can cover a wide range of programming languages, their dialects and embeddings. These front-ends are used for example to implement reverse engineering tools, to build quality assessment tools, to execute research in mining software repositories, or to build (embedded) domain specific languages. In these contexts the creation of the parser is a necessary and important step, but it is also an overhead cost that would preferably be mitigated. In such language engineering applications, as opposed to compiler construction, we may expect frequent updates and maintenance to deal with changes in the grammar.

Expression grammars are an important part of virtually every programming language. The natural specification of expressions is usually ambiguous. In programming languages books and reference manuals, the semantic definition of expressions usually includes a table of binary and unary operators accompanied with their priority and associativity relationships. This approach feels very

natural, probably because this is the way we learn basic arithmetic expressions at school. Virtually all disambiguation techniques for expression grammars are driven by such precedence rules. However, the implementation of such rules varies considerably.

The implementation of operator precedence in grammars may considerably deviate from such an initial design the language engineer has in mind. In manual rewriting approaches, grammars are *factored* to remove ambiguities. These approaches are not attractive for us because the resulting grammars are usually large, and hard to read and understand. For example, languages such as OCaml, C# and Java have dozens of operators with dozens of priority levels and mutual associativity relations. Manually transforming such an expression grammar to encode precedence rules is a significant undertaking. To make matters worse, we expect changes and evolution of grammars. Every time a new operator is introduced we would have to re-think or even re-do the whole complex and error-prone process. Therefore, we consider declarative approaches in which the parser is generated from the set of precedence rules.

Generalized context-free parsing algorithms provide the opportunity to write any context-free grammar, and allow for language compositions, which helps in modeling embeddings and dialects. This makes generalized context-free parsing a good starting point for our purpose: satisfying the demand for powerful and maintainable front-ends. This is especially desired in the fields of domain-specific languages, reverse engineering, and formal semantics where grammars should be easy to understand, evolvable, and maintainable. Therefore, the focus of this paper is mainly targeted at providing a declarative framework for specification of precedence rules in generalized context-free parsing algorithms, such as Earley [1], GLR [2,3,4,5] and GLL [6].

1.1 From Yacc to SDF

In this section, we discuss two disambiguation techniques that influenced our work the most, and are related to generating parsers from ambiguous grammars using a set of precedence rules. Aho, Johnson, and Ullman [7] (AJU) present an approach in which the LR(1) parsing tables are modified to eliminate shift/reduce conflicts based on the declaration of precedence of operator tokens. The AJU method is not only a disambiguation method, it is also a *nondeterminism reducer*, meaning that it has to resolve all shift/reduce and reduce/reduce conflicts, even when there is no ambiguity, to make the parser deterministic. This implies that the approach cannot predictably deal with expression grammars that are not inherently LR(1), unless the language engineer understands how additional shift/reduce and reduce/reduce actions, used for making the parser deterministic, affect the language. More importantly, the AJU precedence semantics is defined in terms of the deterministic LR parsers: to understand the semantics of the precedence rules, one must understand what an LR(1) conflict is and why it happens. Finally, this method is not directly applicable to non-LR parsers.

The AJU approach is implemented in Yacc [1] and is very popular. For example, the OCaml parser uses `ocamlyacc`, which is a variant of Yacc [2]. However, the OCaml grammar used in `ocamlyacc` is heavily factored and is very different from the nice, concise reference manual grammar of OCaml.

Although the AJU method is fast and effective when used in the context of arithmetic expressions, because it is bound to LR(1) parsing, it does not fit into our definition of declarative operator precedence techniques. We require that a declarative specification of operator precedence (1) be independent of the underlying parsing technology, so that we can reason about the precedence semantics or use the mechanism in other parsing technologies, and (2) be *safe*, meaning that the disambiguation mechanism derived from precedence rules should not change the underlying sentences of the grammar.

There has been a number of efforts to formalize a parser-independent semantics of precedence, and provide declarative precedence rules. The most notable one is SDF[3] in which the semantics of operator precedence is defined as a filter on derivation trees. SDF precedence filters are implemented by removing transitions corresponding to filtered productions from adapted SLR(1) tables [8]. Although we believe that SDF was in the right direction in defining a declarative precedence rules, its filters lack the safety requirements. For example, precedence rules in SDF fail to disambiguate a left-associative binary operator having higher priority than a unary prefix operator. The limitations of SDF are discussed in detail in Section 2.1.

1.2 Contributions and Roadmap

In this paper we present a new semantics for the declarative specification of precedence rules for context-free grammars. The key enablers of our technique are the safety and support for resolving deeply nested precedence conflicts. We also support indirect precedence conflicts when expression grammars are not expressed using a single recursive nonterminal but rather more. The new algorithms proposed in this paper are part of the implementation of the parser generator for Rascal. Using this implementation, we show that our approach is powerful enough to allow declarative specification of operator precedence in OCaml. More importantly, the semantics of our technique is implemented as a grammar transformation, making it independent of the underlying parsing technology. We also guarantee that the parsers we generate produce exact same parse trees (as if the original grammar was used). The completeness proof of our semantics —whether our semantics resolves all the precedence style ambiguities— and the soundness proof of the transformation —whether the transformation exactly implement the semantics— are future work.

The rest of this paper is organized as follows. After this introduction, we give formal definitions which we need in the rest of this paper. Then, we explain the problems with SDF in detail in Section 2.1. After that, the formal semantics

[1] `http://dinosaur.compilertools.net/yacc/`
[2] `http://caml.inria.fr/pub/docs/manual-ocaml/manual026.html`
[3] `http://www.syntax-definition.org`

of precedence rules and its implementation as a grammar transformation are presented in sections 3 and 4. We present the results of parsing the OCaml test suite in Section 5. Finally, a discussion of related work and a conclusion of this work are given in sections 6 and 7, respectively.

2 Motivation

A grammar is a 4-tuple (N, T, P, S) where N is a set of nonterminals, T a set of terminals, P a set of production rules of the form $A ::= \alpha$ where A, the *head* of the production rule, is a nonterminal and α is a string in $(T \cup N)^*$. We shall assume that there are no repeated rules, so we can identify a grammar rule by writing its left and right hand sides. $S \in N$ is the start symbol of the grammar. By convention, in this paper, nonterminals and terminals start with uppercase and lowercase letters, respectively. In addition, symbols, such as $+$ or $*$, are terminals. We use lowercase letters u, v, w to denote non-empty sequences of terminal symbols. A group of production rules that have the same head can be grouped as $A ::= \alpha_1|\alpha_2|...|\alpha_n$ where each $A ::= \alpha_i$ is a production. In this representation, each α_i is called an *alternate* of A.

A derivation step is of the form $\alpha A\beta \Rightarrow \alpha\gamma\beta$ where $\alpha, \beta \in (T \cup N)^*$ and $A ::= \gamma$ is a grammar rule. In a derivation step a nonterminal A is replaced with the body of its production rule. A derivation of σ from τ is a possibly empty sequence of derivation steps of the form $\tau \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow ... \Rightarrow \sigma$, which is also written as $\tau \overset{*}{\Rightarrow} \sigma$. A derivation is left-most if at each step its left most nonterminal is rewritten. A derivation from the start symbol is called a *sentential form* which is a sequence of terminals or nonterminals. A sentential form consisting only of terminal symbols is called a *sentence*.

A sentence is *ambiguous* if it has more than one left-most derivation. *Disambiguation* is a process which eliminates derivations. A disambiguation is said to be *safe* if it does not remove all derivations. Therefore, a safe disambiguation mechanism does not change the underlying language generated by a grammar.

2.1 Limitations of SDF

SDF features three meta notations $>$, *left*, and *right*, which specify the precedence, left and right associativity of operators, respectively [9]. Having $A ::= \gamma > B ::= \alpha$ [4] disallows the derivations of $B ::= \alpha$ from all B's in γ. $A ::= A\alpha$ {*left*} means that th A in $A\alpha$ should not derive $A ::= A\alpha$ itself. Right associativity is the same as the left, but applied on the right-most A. There are three problems with the semantics of SDF[5] disambiguation filters:

- It is *unsafe*: A filter is applied even if there is no ambiguity. For example, having ($E ::= E \ ^\wedge \ E > E ::= -E$) rejects the string $1 \ ^\wedge -1$, even though

[4] SDF adheres to algebraic notations and writes $A ::= \gamma$ as $\gamma \rightarrow A$. In this paper we use the more common ::= notation.

[5] We describe here SDF version 2 [9] but we simply call it SDF.

this string is not ambiguous. The reason is that based on the semantics of SDF, $-E$ cannot appear under any of the E's. SDF also allows the user to specify under which nonterminal the filtering should be carried out. For example, the user can specify that the filtering should be carried out under the first E only as ($E ::= E \ ^\wedge E <0>> E ::= -E$). More importantly, the explicit selection of the nonterminal to be filtered is transitively applied to all levels below, even where it should not be applied, producing wrong results.

- It is *incomplete:* The precedence relationship in SDF is defined as a one-level relationship. As a result, it cannot resolve ambiguities in some cases that require deeper than one level searching in the derivation trees. For example, a left-associative binary operator having higher priority than a prefix unary operator remaines ambiguous. The problem with one-level filtering is explained in Section 2.2.
- It is *limited* to directly recursive rules. Although SDF has some extensions to filter priority modulo chain rules, general indirect recursion is not supported. Rules such as $E ::= E\,A$, where the right-most nonterminal, A, can eventually produce an E at the right-most position cannot be filtered using SDF priorities.

These limitations are encountered in practice. For example, the `if-then-else` operator in functional programming languages such as OCaml and Haskell acts as a unary operator with lower priority than left-associative binary operators. Indirect recursion also happens for example in the reference grammar of OCaml.

2.2 Problem with one-level filtering

To illustrate the problem with one-level filtering, we consider the `if-then-else` construct in OCaml, which has lower priority than `+`. For example, the expression `1 + if x then 2 else 3 + 4` is interpreted as `1 + (if x then 2 else (3 + 4))` rather than `(1 + (if x then 2 else 3)) + 4`. For notational simplicity, the `if...then..else` part is replaced with *if* .

$$\begin{aligned} E ::= & E + E \\ & \mid \mathit{if}\ E \\ & \mid \mathit{Num} \end{aligned}$$

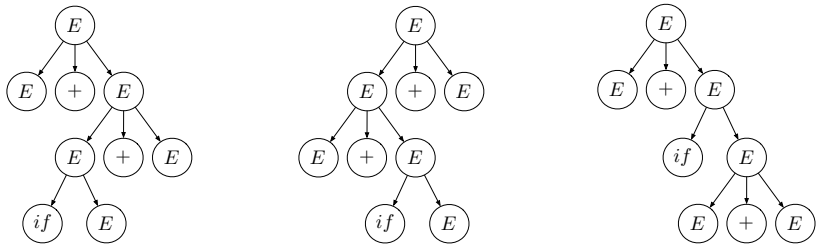
Fig. 1 shows the parse trees resulting from parsing the input `1 + if 2 + 3`. For a more compact presentation the terminals (1, 2 , 3) are removed.

In SDF, the precedence and associativity rules for disambiguating this case will be:

$$E ::= E + E\ \{\mathit{left}\} \qquad \text{(Definition 1)}$$

$$E ::= E + E > E ::= \mathit{if}\ E \qquad \text{(Definition 2)}$$

The disambiguation is not safe in this case, as $E ::= \mathit{if}\ E$ is applied under both E's, rejecting a sentence such as 1+ *if* 2+3. We can make it safe by changing



(a) $(E + ((if E) + E))$ (b) $(E(E + (if E)) + E)$ (c) $(E + (if(E + E)))$

Fig. 1: Parse trees from parsing `1 + if 2 + 3`

Definition 2 into ($E ::= E + E$ <0>> $E ::= if\ E$). Now if we examine the effect of the definitions on the shown parse trees in Fig. 1 we can observe that the left-associativity removes the derivation in Fig. 2a. However, none of the definitions affect the remaining two parse trees, and thus the disambiguation fails. The reason that SDF definitions fail to disambiguate this grammar is that patterns of depth greater than two are required. The first E in $E ::= E + E$ can first derive $E ::= E + E$ and then the second E in the newly derived rule derives $E ::= if\ E$. In other words, the following derivation sequence

$$E \Rightarrow E + E \Rightarrow E + E + E \Rightarrow E + if\ \ E + E$$

which is not rejected by any of the defined patterns, but it is semantically incorrect. The derivation in Fig 1c is correct and is the only one that should remain after disambiguation.

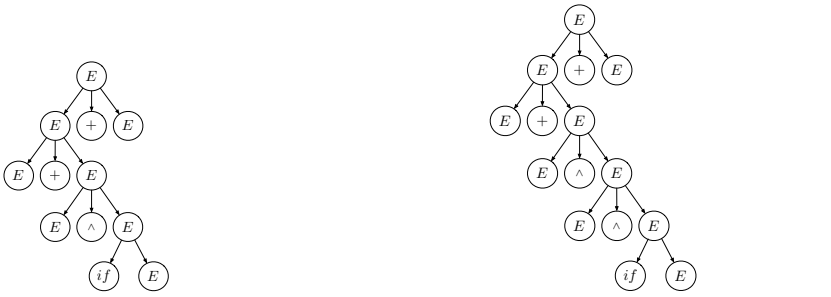
For this grammar, a two level filtering can solve the problem, but in general, we may need filters of arbitrary depth. For example, consider the following grammar which has an additional expression rule $E ::= E\ \hat{}\ E$, where $\hat{}$ is right associative and has the highest priority.

$$\begin{aligned} E ::= & E\ \hat{}\ E \\ | & E\ +\ E \\ | & if\ E \\ | & Num \end{aligned}$$

To illustrate why filters of arbitrary depth may be needed, consider the following derivation:

$$E \Rightarrow E + E \Rightarrow E + E + E \Rightarrow E + E\ \hat{}\ E + E \overset{*}{\Rightarrow} E + E\ \hat{}\ E\ \hat{}\ ...\ \hat{}\ E + E$$

As can be seen, after deriving $E+E$, the second E may unboundedly produce $E\ \hat{}\ E$, leading to wrong derivation trees. Fig. 2 shows two of such derivations. For disambiguation of such grammars, either an infinite number of filters or a mechanism to define filters with variable length is needed. It is not trivial to



(a) $(E(E + E(E\ \hat{}\ (\ if\ E))) + E)$ (b) $(E(E + E(E\ \hat{}\ (E\ \hat{}\ E(\ if\ E)))) + E)$

Fig. 2: For some expression grammars filters of arbitrary depth may be required.

implement a variable length filter during parsing and it is most likely that the performance of such an implementation will suffer.

We have now established the gap in resolving ambiguity in expression grammars. In the following we propose a general solution that will solve the aforementioned limitation, and at the same time improve on several other quality aspects.

3 Syntax and Semantics for operator-style disambiguation

Expression-style grammar rules display a specific kind of ambiguity, which we call *operator-style* ambiguity. We characterize and define two complementary and safe ambiguity removal schemes for exactly this kind of ambiguity: priority and associativity. Note that this does not imply that our mechanisms completely disambiguate any expression grammar. There may be other ambiguity hidden in the same rules with different causes. This other ambiguity should be left untouched for safety.

3.1 Definitions

Definition 1 (Operator-style ambiguity) *An operator-style-ambiguity exists if for some grammar nonterminal E there exist two leftmost derivations*

$$xE\mu \Rightarrow x\beta E\mu \underset{lm}{\overset{*}{\Rightarrow}} xvE\mu \Rightarrow xvE\alpha\mu \tag{1}$$

$$xE\mu \Rightarrow xE\alpha\mu \Rightarrow x\beta E\alpha\mu \underset{lm}{\overset{*}{\Rightarrow}} xvE\alpha\mu \tag{2}$$

which contain identical sub-derivations $\beta \underset{lm}{\overset{*}{\Rightarrow}} v$.

The first derivation in the above definition effectively corresponds to the binding $x(\beta E)\alpha\mu$ and the second derivation corresponds to binding $x\beta(E\alpha)\mu$. Both derivations correspond to the same sentential form, but between them the order

of applying $E\alpha$ and βE as been inverted. Note that it may happen that $\alpha = \beta$, but only for binary recursive rules $E ::= E\gamma E$.

The benefit of the above characterization of operator-style ambiguity is that we use pairs of derivation sequences that specifically allow an arbitrary distance ($\stackrel{*}{\Rightarrow}$) between application of βE and $E\alpha$. This creates the potential for supporting deeper ambiguities, and indirectly recursive expression grammars. In addition, we now have defined clearly what it means for operator-style ambiguity removal to be safe: never both derivations (1) and (2) may be removed at the same time.

Given a grammar which contains operator-style ambiguity, the engineer has to specify, somehow, which derivation should be removed. There are many situations in which the engineer wishes always (i.e. for all strings) to choose one sequence over the other. We first describe *priority-based ambiguity removal*.

Definition 2 (Priority-based ambiguity removal via $>$) *The user specifies a strict partial order $>$ (irreflexive, antisymmetric and transitive) between the alternates of E. For all $\beta E > E\alpha$, derivations which contain sequences of the form* (2) *are always removed. Vice versa, for all $E\alpha > \beta E$, we choose to remove* (1)*. Note that we do not intend to apply the partial order on other cases of ambiguity, only in the case of the* (1) *and* (2) *pair it serves to choose one over the other.*

This definition correlates with the common use of operator priority to specify disambiguation, for example choosing the first derivation gives the β "operator" priority over the α "operator". Since all derivations $\beta \stackrel{*}{\Rightarrow} v$ are available for both choices, priority disambiguation does not put constraints on other disambiguation choices.

The fact that $>$ is asserted to define a strict partial order is an important detail for satisfying the safety requirement. If there would be both $\alpha > \beta$ and $\beta > \alpha$ for example, then the above definitions would together remove all derivations for both some or all sentences that α and β generate. Similarly $\alpha > \alpha$ is not allowed. The fact that $>$ is allowed to be partial implies that under-specified orderings may leave some operator ambiguity intact. This means it is up to the language engineer to fully declare what the relative precedence of operator is, and also that the priority relation can safely be developed incrementally.

There are, however, common situations in which we do not want to use or cannot enforce a strict partial order as required by $>$. In particular, if an expression-style rule has an alternate with both immediate left and right recursion, $E ::= E\gamma E$, then it is not possible to specify priority with itself, since $>$ must be irreflexive and antisymmetric. More generally, there may be two alternates $E ::= E\gamma E \mid E\delta E$ where γ and δ are required to have a symmetric relation (such as $+$ and $-$ in arithmetic expressions), which also contradicts a strict partial order.

Definition 3 (Symmetric Operator-style Ambiguity) *Instantiating α and β from derivations (1) and (2) above as $\beta = E\delta$ and $\alpha = \gamma E$ both rules are now*

binary recursive. We can instantiate derivations (1) and (2) above like:

$$xE\mu \Rightarrow xE\delta E\mu \underset{lm}{\overset{*}{\Rightarrow}} xvE\mu \Rightarrow xvE\gamma E\mu \quad (1')$$

$$xE\mu \Rightarrow xE\gamma E\mu \Rightarrow xE\delta E\gamma E\mu \underset{lm}{\overset{*}{\Rightarrow}} xvE\gamma E\mu \quad (2')$$

Also, taking $\beta = E\gamma$ and $\alpha = \delta E$ we can write derivations (1) and (2) above as

$$xE\mu \Rightarrow xE\gamma E\mu \underset{lm}{\overset{*}{\Rightarrow}} xvE\mu \Rightarrow xvE\delta E\mu \quad (1")$$

$$xE\mu \Rightarrow xE\delta E\mu \Rightarrow xE\gamma E\delta E\mu \underset{lm}{\overset{*}{\Rightarrow}} xvE\delta E\mu \quad (2")$$

Symmetric operator-style ambiguity is a special case of operator-style ambiguity in which both rules are binary. Often we have $\delta = \gamma$, although this is not necessary. To see why we call the ambiguity symmetric, consider the example where $\gamma = +$ and $\delta = -$, $(1')$ and $(2')$ both derive $y + y - y$ and, $(1'')$ and $(2'')$ both derive $y - y + y$. Then, $(1')$ and $(1'')$ represent $(y + y) - y$ and $(y - y) + y$, respectively.

Definition 4 (Associativity-based ambiguity removal via *left* and *right*) *We define two binary relations "left" and "right" between binary alternates, for which holds that*

$$(left \ \cap \ right = \emptyset) \ \wedge \ (left \ \cap \ '>' = \emptyset) \ \wedge \ (right \ \cap \ '>' = \emptyset)$$

In other words, $>$, left and right are mutually exclusive relations.

When $(\alpha, \beta) \in left$, associativity based ambiguity removal removes the sequences of the form $(2')$, corresponding to grouping γ and δ to the left, i.e. to choosing $x(w\delta E)\gamma E\mu$ over $w\delta(E\gamma E)\mu$. This correlates with left associativity. Similarly, when $(\alpha, \beta) \in right$, removing derivations with sequences of the form $(1')$ corresponds to right associativity.

The restriction of $>$, *left* and *right* being mutually exclusive is a sufficient restriction for guaranteeing safety since now only one relation is allowed to be active at the same time and each of the relations is safe in itself.

Since $>$, *left* and *right* need to define an order between all alternates of expression languages with dozens of rules, we cannot expect the language engineer to specify each combination manually. This problem is dealt with in our formalism, which is described later, by providing automatic transitive closure for $>$ and a computation akin to Cartesian product for *left* and *right* groups of rules.

In summary, the three relations $>$, *left* and *right* allow a language engineer to remove all operator-style ambiguity of the form in Definition 1, either using an anti-symmetric, irreflexive, transitive relation $>$, or using one of the possibly reflexive, possibly symmetric and possibly non-transitive *left* and *right* relations as long as the three relations exclude each other. Note that in theory all operator-style ambiguity can be removed by simply asserting a full ordering among all recursive alternates using $>$ or by putting all rules in a single *left* or *right* group,

but this has no practical value. Instead, complete disambiguation of the operator-style ambiguity in a language definition needs to be considered language-by-language (see Section 5).

3.2 Pattern notation for illegal derivation sequences

As an intermediate step we now introduce a short notation for the sequences $(1), (2), (1')$ and $(2')$, called "patterns". Each pattern is specific for a given grammar and combination of two alternate rules. In the next section, we demonstrate how to compute a unified set of patterns from a context-free grammar $+(>, \mathit{left}, \mathit{right})$, and how to use this set of patterns to compute a grammar transformation that implements the above semantics.

Definition 5 (Operator ambiguity removal pattern) *An operator ambiguity removal pattern (pattern for short) is a 4-tuple of the form* $(\mathit{head}, \mathit{parent}, i, \mathit{child})$*, where head is the nonterminal head of the expression grammar for which the precedence rules are defined, parent is an alternate of head, i is the index of a nonterminal in the body of parent, and child is the alternate that should be filtered from the nonterminal at position i of parent. The nonterminal at position i is called the* filtered nonterminal.

In this paper we write a pattern as $(E, \alpha \bullet \beta\,,\, \gamma)$ *where E is the head, and* $\alpha \bullet \beta$ *and* γ *are the parent and the child alternates, respectively, and the filtered nonterminal is identified by a dot before it.*

The semantics of patterns are the same as derivations discussed above. For example, the derivations (1) and (2) can be expressed as the patterns $(E, \alpha \bullet E\,,\, E\beta)$ and $(E, \bullet\, E\alpha\,,\, E\beta)$, respectively. Note that patterns are not implementation mechanisms. In Section 4 we show a grammar rewriting algorithm to implement patterns.

We now explain informally how to arrive at a set of patterns starting from a context-free grammar augmented with $(>, \mathit{left}, \mathit{right})$. Table 1 documents the semantics of priority in terms of patterns that are generated for each combination of left, right and binary recursive expression rules. Note that for binary rules sometimes two pattern are generated for the same combination of rules. The semantics of *left* in terms of the patterns is expressed similarly in Table 2. We leave the table for right associativity to the reader.

As can be seen, not all combinations of expression rules generate patterns. Exactly when the combination of rules would *not* be ambiguous and filtering would be unsafe no pattern is generated. This corresponds to the sequence definitions $(1), (2), (1'), (2')$ using specific combinations of left and right recursive rules. In Section 4 we implement these tables, trivially generalizing them to allow indirect left and right recursion as well.

Table 1: The semantics of the $>$ operator in terms of patterns.

$>$	$E ::= E\alpha_2 E$	$E ::= E\alpha_2$	$E ::= \alpha_2 E$
$E ::= E\alpha_1 E$	$(E, \bullet E\alpha_1 E\ ,\ E\alpha_2 E)$ $(E, E\alpha_1 \bullet E\ ,\ E\alpha_2 E)$	$(E, E\alpha_1 \bullet E\ ,\ E\alpha_2)$	$(E, \bullet E\alpha_1 E\ ,\ \alpha_2 E)$
$E ::= E\alpha_1$	$(E, \bullet E\alpha_1\ ,\ E\alpha_2 E)$	—	$(E, \bullet E\alpha_1\ ,\ E\alpha_2)$
$E ::= \alpha_1 E$	$(E, \alpha_1 \bullet E\ ,\ E\alpha_2 E)$	$(E, \alpha_1 \bullet E\ ,\ E\alpha_2)$	—

Table 2: The semantics of *left* associativity

left	$E ::= E\alpha_1 E$	$E ::= E\alpha_2 E$
$E ::= E\alpha_1 E$	$(E, E\alpha_1 \bullet E\ ,\ E\alpha_1 E)$	$(E, E\alpha_1 \bullet E\ ,\ E\alpha_2 E)$
$E ::= E\alpha_2 E$	$(E, E\alpha_2 \bullet E\ ,\ E\alpha_1 E)$	$(E, E\alpha_2 \bullet E\ ,\ E\alpha_2 E)$

3.3 Defining $>$, *left* and *right* in practice

The following three features, which are taken from the design of SDF [10], are described here for completeness sake. They are essential for having concise expression grammars, as mentioned above.

Firstly, our formalism automatically transitively (but not reflexively) closes the $>$ relation precedence operator. As a result, when the language engineer defines $p_1 > p_2$ and $p_2 > p_3$ we automatically derive $p_1 > p_3$. Furthermore, when they accidentally define $p_1 > p_1$, or both $p_1 > p_2$ and $p_2 > p_1$, either directly, or indirectly via the closure, an error message must be produced. Now we can allow the short-hand $p_1 > p_2 > p_3$ to obtain elegant definitions. Note that the transitive closure step is carried out before generating the actual patterns. The actual patterns are generated from the calculated priority pairs only when is there is an operator-style ambiguity, as defined in Section 3.2 and documented in Table 1.

Secondly, many programming languages have groups of binary operators that have the same precedence level. For example, in $E ::= E + E \mid E - E$ both operators have the same precedence level but should be left associative with respect to each other. We define a left associative group containing a set of rules $(p_1 | \ldots | p_n)(\mathit{left})$ to generate a set of associativity declarations:

$$\bigcup_{1 \leq i,j \leq n} p_i\ \mathit{left}\ p_j, \textbf{when}\ \ (p_i, p_j) \notin \mathit{right} \wedge (p_i, p_j)\ \notin\ '>'\ \ \wedge (p_j, p_i)\ \notin\ '>'$$

Similarly for right associative groups. The groups simply compute the Cartesian product, but do not add tuples that would contradict a relation defined elsewhere. Finally, associativity groups may occur in the middle of a priority chain, as in $(p_1 | \ldots | p_n)(A) > (q_1 | \ldots | q_n)(B)$. In this case $>$ will be extended by combining each element of the two groups pairwise (and before closure). An

$$
\begin{array}{rl}
E ::= & E\ Arg+ \quad \text{//function application} \\
| & -\ E \quad\quad \text{//unary minus} \\
| & E\ **\ E \\
| & E\ +\ E \\
| & E\ -\ E \\
| & \mathit{if}\ E\ \mathit{then}\ E\ \mathit{else}\ E \\
| & Id \\
Arg\ ::= & \ E \\
| & \sim label : E
\end{array}
$$

Operator	Associativity
function application	–
unary minus	–
**	right
+, -	left
if-then-else	–

Fig. 3: Excerpt from OCaml's grammar with "challenging" operator precedence.

$$
\begin{array}{rll}
E ::= & E\ Arg+ & (non{-}assoc) \\
> & -\ E & \\
> & E\ **\ E & (right) \\
> & (\ E\ +\ E \mid E\ -\ E\ ) & (left) \\
> & \mathit{if}\ E\ \mathit{then}\ E\ \mathit{else}\ E & \\
| & Id & \\
Arg\ ::= & \ E & \\
| & \sim label : E &
\end{array}
$$

Fig. 4: Example definition of challenging operator precedence rules.

additional safety feature (which is novel) is to simply statically check for $>$, *left* and *right* to be non-overlapping as required.

Finally, some expression languages disallow certain direct nesting while indirect nesting is allowed. For example `1 == 2 == 3` should be not allowed while `true == (2 == 2)` is allowed. Normally we would have to introduce a new expression nonterminal just to disallow this direct nesting. So, in order to be able to write concise grammars we add *non–assoc* declarations with the following semantics. If p_1 *non–assoc* p_2, then $(p_1 \mathit{left}\ p_2) \wedge (p_1\ \mathit{right}\ p_2)$. Notice that *non–assoc* declarations are not safe: they remove sentences from the language as generated by the grammar *intentionally* and *explicitly*. We extend the associativity group semantics with *non–assoc* as well. Necessarily, any static safety checks on *left* and *right* need to be done before the tuples from *non–assoc* have been added.

To illustrate the syntax of our approach we use the following example grammar the priority and associativity properties are taken from the OCaml reference manual[6]. The grammar and the precedence rules can now be written as in Figure 4. We use ::=, $>$, *left*, *right* and *non–assoc* meta notation to encode both the syntax and the precedence table in one go.

[6] `http://caml.inria.fr/pub/docs/manual-ocaml-4.00/expr.html`

4 Grammar rewriting to exclude illegal derivations

In this section we present an algorithm for transforming a grammar accompanied with a set of priority and associativity rules to a grammar that prevents the generation of illegal derivations (see Figures 5 and 6).

1. We translate the definitions to a set of patterns (GENERATEPATTERNS).
2. We apply these patterns to transform the grammar (REWRITEGRAMMAR)

The generation of patterns in Algorithm 5 follows exactly the semantics as defined earlier in Tables 1 and 2. EXTRACTDEFINITIONS produces a set of binary tuples which represent the associativity and priority declarations in a grammar. This set is an over-approximation of the patterns that will be generated later, since they are not specific for positions in the parents yet and may be ignored entirely if no ambiguity may arise. RIGHTRECURSIVE and LEFTRECURSIVE compute for a specific nonterminal which other nonterminals contribute to an eventual left/right recursion of that nonterminal. The GENERATEPATTERN function then filters the extracted definitions making sure to introduce a pattern only where left recursion tangles with right recursion and vice versa, i.e. modeling exactly the priority and associativity semantics of Section 3.

Given the set of patterns generated by GENERATEPATTERNS, we can now transform the grammar using the REWRITEGRAMMAR function as shown in Algorithm 6. It is important to note that we use indexed nonterminals names, such that when building parse trees, no new names for nonterminals is generated (indices can be removed easily). As each rewrite action can only remove some alternates, no new shapes of rules are created by the algorithm (no additional chain rules). This preserve the shape of the parse forest as the language engineer specified in the original grammar.

The algorithm first deterministically generates a set of nonterminals to implement single-level filtering. Lines 14–20 reserve fresh nonterminal names. Lines 21–23 change existing rules to use the new nonterminals at the right positions. Lines 24–28 generate definitions for the new nonterminals by cloning the original while leaving our the filtered alternative. Then, in a fixed point computation (lines 29–46) we treat each level of newly generated nonterminals to a procedure for eliminating deeply nested cases. For left recursive positions (lines 40–46), we make sure that a nonterminal is generated which cannot derive a given postfix operator at arbitrary depth at the right-most position which has lower priority, and vice versa for right recursive positions (lines 33–39). The APPLYPATTERN helper function does the same as lines 21–46 for the first level, but it includes an explicit check for the existence of generated nonterminals to reuse. This check is necessary for termination as well as efficiency. The fixed point computation will terminate because a new nonterminal is only created in APPLYPATTERN if a nonterminal which defines the same subset of alternates does not already exist. Since every step removes an alternate, eventually —in a worst case scenario— all singleton sets will have been generated and the algorithm terminates.

We can illustrate the algorithm using the following example: Grammar G:

$$E ::= E + E(\mathit{left}) \ > \ iE \mid a;$$

function EXTRACTDEFINITIONS(G)
  $'>' \leftarrow '>' \cup \{(p_i, q_j) | (p_1 \dots p_i)(A) > (q_1 \dots q_j)(B) \in G\}$ ▷ expand the groups
  $\mathrm{P} \leftarrow \{(p_1, p_2) \mid p_1 > p_2 \in G\}+$ ▷ note the transitive closure
  $\mathrm{L} \leftarrow \{(p, p) \mid p \ \mathit{left}\ p \ \in G\}, L' \leftarrow L$
  $\mathrm{R} \leftarrow \{(p, p) \mid p \ \mathit{right}\ p \ \in G\}$
  $\mathrm{L} \leftarrow L \cup \bigcup_{0 \le i,j \le n} \{(p_i, p_j) \mid (p_1 | \dots | p_n)(\mathit{left}) \in G, (p_i, p_j) \notin R\}$
  $\mathrm{R} \leftarrow R \cup \bigcup_{0 \le i,j \le n} \{(p_i, p_j) \mid (p_1 | \dots | p_n)(\mathit{right}) \in G, (p_i, p_j) \notin L'\}$
  return $P \cup L \cup R$
function RIGHTRECURSIVE(G, N) ▷ LEFTRECURSIVE is elided for brevity
  $R \leftarrow \{\mathrm{N}\}$
  while R changes **do** $R \leftarrow R \cup \{X | X ::= \alpha Y \in G, Y \in R\}$
  return R
function PLAIN(x) = x in which all N_i are replaced by N.
function RULES(G, N) = $\{\beta | N ::= \beta \in G\}$
function FRESH(N) = N_i where the integer index i has not been used before.
function GENERATEPATTERNS(G)
  $D \leftarrow$ EXTRACTDEFINITIONS(G)
  $R \leftarrow \{\}$
  for all $(A ::= X\alpha, A ::= \beta Y) \in D$ **do**
    if $X \in$ LEFTRECURSIVE$(G, A) \wedge Y \in$ RIGHTRECURSIVE(G, A) **then**
      $R \leftarrow R \cup \{(A, \bullet X\alpha, \beta Y)\}$
  for all $(A ::= \alpha X, A ::= Y\beta) \in D$ **do**
    if $X \in$ RIGHTRECURSIVE$(G, A) \wedge Y \in$ LEFTRECURSIVE(G, A) **then**
      $R \leftarrow R \cup \{(A, \alpha \bullet X, Y\beta)\}$
  return R

Fig. 5: Translating priority and associativity definitions to safe patterns

generates patterns P (see Figure 5): $\{(E,\ \cdot E + E,\ iE), \quad (E,\ E + \cdot E,\ E + E)\}$ Now the algorithm in Figure 6 can start. Lines 14–23 create the following grammar rule in G_1, having found two patterns to apply and allocating two fresh nonterminals: $E ::= E_1 + E_2 \mid iE \mid a$

Then, at lines 24–28 we define the two new nonterminals and extend G_1 with their definition:

$$
\begin{aligned}
E &::= E_1 + E_2 \mid iE \mid a \\
E_1 &::= E_1 + E_2 \mid a \\
E_2 &::= iE \mid a
\end{aligned}
$$

Finally we search for nested issues in lines 30–46. The outer loop executes twice. The first time, considering E_1 results in a new nonterminal E_3 and considering E_2 does nothing. The second time round nothing changes and we terminate with

```
1: function ApplyPattern(G, W, δ, V ::= μ′W′τ′))
2:     Y_alts = ∅
3:     for all ρ ∈ rules(G, W) do
4:         if plain(ρ) ≠ plain(δ) then add ρ to Y_alts
5:     if ∃Z ∈ G : (plain(Z) = plain(W)) ∨ (rules(G, Z) = Y_alts) then
6:         Y′ ← Z
7:     else
8:         Y′ ← fresh(W)
9:         for all β ∈ Y_alts do add Y′ ::= β to G
10:    remove V ::= μ′W′τ′ from G
11:    add V ::= μ′Y′τ′ to G
12:    return (G, Y′)
13: function RewriteGrammar((G, P))
14:    New ← ∅
15:    Slots[ ] ← ∅ ▷ an empty map from indexed nonterminal names to dotted rules
16:    for all patterns (Y, β · Yγ, δ) in P do    ▷ Stage 1, reserve nonterminal names
17:        Y_i ← fresh(Y)
18:        Slots[Y_i] ← β · Yγ
19:        add Y_i to New
20:        G_1 ← G
21:    for all patterns (Y, β · Yγ, δ) in P do
22:        if Slots[Y_i] = β · Yγ then                    ▷ Stage 2, update use sites
23:            replace Y ::= βYγ in G_1 with Y ::= βY_iγ
24:    for all Y_i in New do            ▷ Stage 3, add definitions for new nonterminals
25:        if Slots[Y_i] = β · Yγ then
26:            for all Y ::= α in G_1 do
27:                if ∄ a pattern (Y, β · Yγ, δ) ∈ P with plain(α) = δ then
28:                    add Y_i ::= α to G_1
29:    (G″, G′) ← (G_1, G)                            ▷ Stage 4, look for nested ambiguity
30:    while G′ ≠ G″ do
31:        (G′, New′) ← (G″, New)
32:        for all Y_i ∈ New′ do
33:            if Slots[Y_i] = ·Yγ then
34:                for all grammar rules Y_i ::= μW ∈ G′ do
35:                    if plain(W) = Y ∧ ∃Z(plain(Z) = Y then
36:                        ∧ W ∈ RightRecursive(G_1, Z)) then
37:                        for all patterns (Y, ·Yγ, δ) do
38:                            (G″, U) ← ApplyPattern(G″, W, δ, Y_i ::= μW)
39:                            (Slots[U], New) ← (Slots[W], New ∪ {U})
40:            if Slots[Y_i] = β · Y then
41:                for all grammar rules Y_i ::= Wμ in G″ do
42:                    if plain(W) = Y ∧ ∃Z : (plain(Z) = Y
43:                        ∧W ∈ leftRecursive(G_1, Z)) then
44:                        for all patterns (Y, β · Y, δ) do
45:                            (G″, U) ← applyPattern(G″, W, δ, Y_i ::= Wμ)
46:                            (Slots[U], NT) ← (Slots[W], New ∪ {U})
47:    return G″
```

Fig. 6: Core algorithm that rewrites a grammar, applying patterns to remove alternates from indexed nonterminals.

the final grammar:

$$
\begin{aligned}
E &::= E_1 + E_2 \mid iE \mid a \\
E_1 &::= E_1 + E_3 \mid a \\
E_2 &::= iE \mid a \\
E_3 &::= a
\end{aligned}
$$

5 Validation using the OCaml case

For the current paper, we have conducted an extensive validating experiment. The goal is to show that the our approach is indeed more powerful than SDF, and to provide evidence that the algorithm works for complicated, real-world examples.

5.1 Method

For this case study, we selected the OCaml (.ml) files in the test suite directory of the source release of OCaml 4.0.1. OCaml features the kind of ambiguity that SDF filtering semantics cannot solve and our method should be able to solve. The test suite contains numerous examples of different sizes and complexity, testing the language features. We believe the test suite is a good choice for testing our parser on safety and completeness, as the suite rigorously tests the language itself. The suite contains 387 files of which 162 (in the `tool-ocaml` folder) contain only source code comments that document expected output (assembler code) of the compiler. The other 225 files are examples of OCaml code that exercise all features of the language in different combinations to test the compiler.

We performed the experiments in Rascal [11], which is a meta-programming DSL supporting embedded syntax definitions as proposed in this paper. The parsing mechanism of Rascal is based on GLL [6].

Our goal is to provide solid evidence of the complete equivalence between the original OCaml parser and the parser generated from our approach. This means that no parse error should be produced by the Rascal parser if no parse error was produced by the original OCaml parser, and the generated parser should produce single parse trees (no ambiguities), and that the structure of the abstract syntax trees should be exactly the same.

To compare parse trees we adapted both the parser from the OCaml compiler and the output of our generated parser to produce exactly the same bracketed forms. The resulting files are then compared with `diff`, ignoring whitespace, to check for equivalence.

OCaml programs are basically composed of groups of expressions. The AST produced by the OCaml parser is complex and contains many features. However, because of the expression-like nature of the language, most of the unnecessary information can be removed, resulting in a bracketed form. We modified the default AST printer[7] to produce the bracketed form. For example, the original

[7] At `src/4.00/parsing/printast.ml` in the OCaml source release.

```
Ptop_def                                                    (
  [                                                          +
    structure_item ([1,0+0]..[1,0+5]) ghost                  (
      Pstr_eval                                                 1
      expression ([1,0+0]..[1,0+5])                             *
        Pexp_apply                                              (
        expression ([1,0+1]..[1,0+2])                              2
          Pexp_ident "+"                                           3
        [                                                       )
          <label> ""                                         )
            expression ([1,0+0]..[1,0+1])                  )
              Pexp_constant Const_int 1
          <label> ""
            expression ([1,0+2]..[1,0+5])
              Pexp_apply
              expression ([1,0+3]..[1,0+4])
                Pexp_ident "*"
              [
                <label> ""
                  expression ([1,0+2]..[1,0+3])
                    Pexp_constant Const_int 2
                <label> ""
                  expression ([1,0+4]..[1,0+5])
                    Pexp_constant Const_int 3
              ] ] ]
```

Fig. 7: The original AST print from the OCaml parser (left) and the stripped version containing only the structure and the labels.

AST and its bracketed form, resulting from parsing the string `1+2*3` is shown in Fig. 7. The bracketed forms of all the examples we examined are on GitHub.

For conducting the experiments we wrote a Rascal grammar definition using the notations defined in this paper. The grammar is obtained from the OCaml reference manual[8]. We try to be as faithful as possible to the grammar in the reference manual, avoiding changes as much as possible.

5.2 Results

The priority and associativity properties, retrieved from the precedence tables in the language manual, resulted in a grammar that uses $>$ and *left*, *right* and *non–assoc* declarations. These declarations result in about 800 derivation ambiguity removal patterns. The rewriting was performed as explained in Section 4.

The rewritten grammar provided us with a very close over-approximation of what the OCaml language designers had in mind. Only a handful of ambiguities, such as the dangling-else ambiguity and identifier conflicts with keywords, remained, which were resolved using other ambiguity resolution features of Rascal. The OCaml grammar written in Rascal is available at: `https://github.com/cwi-swat/ocaml-operator-ambiguity-experiment`.

We have performed the parsing and comparison process for the given 225 number of files in the case study. All files parse correctly and without ambiguity.

[8] `http://caml.inria.fr/pub/docs/manual-ocaml-400/language.html`

Of the 225 parsed files, 172 files (76%) generate ASTs that are identical in both versions. This means that our parser produces the same grouping as the original OCaml parser, providing evidence for the correctness of our algorithms. The remaining files which are left with minor differences caused by AST de-sugaring and normalization steps in the OCaml compiler.

5.3 Discussion and threats to validity

One of the difficulties in this study was how to compare ASTs. The AST from the OCaml parser, in some places, is significantly different from the grammar written in the reference manual. The reason is the trees have been normalized by the front-end for easier processing later in the compiler. For example, flat argument lists are converted to cons lists, presumably to simplify currying and partial function features in OCaml. These changes are not documented anywhere. We resolved them by observing the original AST output to deduce the normalization step. We then mimicked these normalization steps as rewrite rules in Rascal before outputting the final bracketed form.

Moreover, OCaml has some language extension and syntax varieties that are not documented in the main language reference document. The use of semicolon was particularly confusing. Semicolons is used in OCaml to separate expressions, defined by the rule $E ::= E\,;\,E$ which is right associative. However, in the inputs we parsed, we observed several occasions in which semicolon can end an expression regardless of being preceded by another expression. We resolved this issue by allowing optional semicolons at the end of expressions.

6 Related work

Besides the AJU and SDF methods which have been described so far, there are a number of work which present similar ideas. Aasa [12] proposes a framework for specification of precedences for implementing programming language. To the best of our knowledge, this is the only declarative model that supports deeper patterns. In [12], a tree is considered precedence correct based on the weights given to operators in its parse trees. This work correctly recognizes that, for example, a unary operator can be placed under the right most operand of a binary rule, regardless of their precedence. However, our approach in defining precedence semantics is very different. Instead of focusing on parse trees, we defined the semantics of precedence as derivations, which is closer to our implementation technique. The main shortcoming of [12] is that operators must be unique. They are considered separately from their context, e.g., there cannot be a unary minus and a binary minus at the same time. In addition, there is no discussion of indirect recursions in [12]. Similar to us, the disambiguation technique in [12] is implemented as a grammar rewriting.

Thorup [13] presents an algorithm for transforming an ambiguous grammar with a set of partial illegal parse trees to grammar excluding those derivations.

On the surface, the approach looks very similar to our technique shown in Section 4, but the inner working is very different. The rewriting technique in [13] expects a set of illegal parse trees, and in case the set is unbounded, as in Section 2.2, a set of parse forests with cycles. Then, the algorithm works bottom up, generating all production rules which do not produce any of those illegal parse trees. The result is an algorithm which is complicated and does not look as nice as the hand-written ones. The resulting grammar should go through another transformation be simplified. The problem of how to find sufficient illegal parse trees is addressed in another work by the same author [14]. The rewriting presented by Thorup is not directly aiming at providing a declarative disambiguation mechanism, rather it is more an implementation mechanism. It also covers a wider range or rewriting provided that enough illegal parse trees are provided, but the overall procedure is complicated. We are not aware of any practical parser generator that uses this technique.

Visser presents "From context-free grammars with priorities to character class grammars" [15], which describes a grammar transformation to give semantics to the SDF2 priority relation similar to our transformation. In a first step a grammar's nonterminals are replaced by explicit sets of identities (integers) of production alternates. Then, elements are removed from these sets based on the priority relation and parse table is generated normally. Since every rule is identified, the resulting parse trees do not show the signs of grammar transformation. Character class grammars do not guarantee to preserve the language and do not support indirect recursion, like our semantics do. Although character class grammars are formalized quite differently from our approach that directly manipulates grammars using indexed nonterminals, both methods use grammar transformation to implement the priority relation. Therefore, we can label Visser's work a predecessor of our contribution.

7 Conclusions

Constructing a parser that correctly implements precedence rules, for a language such as OCaml, using its ambiguous reference manual and the set of precedence rules is not possible without resorting to some manual grammar transformation. In this paper, we defined a parser-independent semantics for operator-style ambiguities that is safe and is able to deal with deeper level and indirect precedence ambiguities. We evaluated our approach using an extensive experiment by comparing the output of the standard OCaml compiler front-end with the output of our own parser, generated from Rascal BNF extended with priority and associativity declarations. The result is promising and shows that our approach is powerful enough to parse OCaml.

For other languages such as Haskell, F#, and Lua, which offer similar expression languages, our approach is expected to be equally beneficial. Although the focus of this paper is mainly on generalized parsing algorithms, we should also emphasize that our approach can be used by any parser generator that supports left recursion, such as SDF, ANTLR 4, Elkhound[4], or DMS [5]. As

future work, we plan to work on the soundness and completeness proofs of our grammar transformation of Section 4. Moreover, we will apply our approach on more programming languages that have complex precedence rules.

Acknowledgments. We would like to thank Peter Mosses who has originally identified the problem in OCaml. Also many thanks to Davy Landman and Mark Hills from CWI who assisted us in performing the validation experiments.

References

1. Earley, J.: An efficient context-free parsing algorithm. Commun. ACM **13**(2) (February 1970) 94–102
2. Tomita, M., ed.: Generalized LR parsing. Kluwer Academic Publishers (1991)
3. Rekers, J.: Parser Generation for Interactive Environments. PhD thesis, University of Amsterdam, The Netherlands (1992)
4. McPeak, S., Necula, G.C.: Elkhound: A fast, practical glr parser generator. In: CC. (2004) 73–88
5. Baxter, I.D., Pidgeon, C., Mehlich, M.: DMS®: Program transformations for practical scalable software evolution. In: Proceedings of the 26th International Conference on Software Engineering. ICSE '04, Washington, DC, USA, IEEE Computer Society (2004) 625–634
6. Scott, E., Johnstone, A.: GLL parse-tree generation. Science of Computer Programming (2012) to appear ISSN:0167-6423.
7. Aho, A.V., Johnson, S.C., Ullman, J.D.: Deterministic parsing of ambiguous grammars. In: Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages. POPL '73, ACM (1973) 1–21
8. Visser, E.: Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam (July 1997)
9. Klint, P., Visser, E.: Using filters for the disambiguation of context-free grammars. In Pighizzini, G., San Pietro, P., eds.: Proc. ASMICS Workshop on Parsing Theory, Milano, Italy, Tech. Rep. 126–1994, Dipartimento di Scienze dell'Informazione, Università di Milano (October 1994) 1–20
10. Visser, E.: Syntax Definition for Language Prototyping. PhD thesis, University of Amsterdam (1997)
11. Klint, P., van der Storm, T., Vinju, J.J.: EASY meta-programming with Rascal. leveraging the extract-analyze-synthesize paradigm for meta-programming. In: Proceedings of the 3rd International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'09). LNCS, Springer (2010)
12. Aasa, A.: Precedences in specifications and implementations of programming languages. Theor. Comput. Sci. **142**(1) (May 1995) 3–26
13. Thorup, M.: Disambiguating grammars by exclusion of sub-parse trees. Acta Informatica **33**(5) (1996) 511–522
14. Thorup, M.: Controlled grammatic ambiguity. ACM Trans. Program. Lang. Syst. **16**(3) (May 1994) 1024–1050
15. Visser, E.: From context-free grammars with priorities to character class grammars. In van Deursen, A., Brune, M., Heering, J., eds.: Dat Is Dus Heel Interessant, Liber Amicorum dedicated to Paul Klint. CWI (1997) 217–230