Operator Precedence for Data-dependent Grammars

Ali Afroozeh Anastasia Izmaylova

Centrum Wiskunde & Informatica, Amsterdam, The Netherlands

{ali.afroozeh, anastasia.izmaylova}@cwi.nl

Abstract

Constructing parsers based on declarative specification of operator precedence is a very old research topic, and there are various existing approaches. However, these approaches are either tied to a particular parsing technique, or cannot deal with all corner cases found in programming languages.

In this paper we present an implementation of declarative specification of operator precedence for general parsing that (1) is independent of the underlying parsing algorithm, (2) does not require any grammar transformation that increases the size of the grammar, (3) preserves the shape of parse trees of the original, natural grammar, and (4) can deal with intricate cases of operator precedence found in functional programming languages such as OCaml.

Our new approach to operator precedence is formulated using data-dependent grammars, which extend context-free grammars with arbitrary computation, variable binding and constraints. We implemented our approach using Iguana, a data-dependent parsing framework, and evaluated it by parsing Java and OCaml source files. The results show that our approach is practical for parsing programming languages with complicated operator precedence rules.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Syntax; D.3.4 [*Programming Languages*]: Processors—Parsing

Keywords Parsing, data-dependent grammars, disambiguation, operator precedence, declarative syntax definition

1. Introduction

Expressions are basic blocks of programming languages, and perhaps, one of the most difficult parts when it comes to parsing. In reference manuals of programming languages it is common to specify the semantics of expressions using the precedence and associativity of operators. For example, consider an excerpt of OCaml expression grammar [22] in Figure 1 (left) and its accompanying precedence table (right). Expression grammars in their concise and natural form, e.g., in Figure 1, are ambiguous, which makes constructing parsers from such grammars challenging.

A common approach to unambiguously parse expression grammars is to encode operator precedence rules by rewriting the grammar. This rewriting, which introduces a new nonterminal for each precedence level, is not trivial for grammars of real programming languages, and leads to large grammars. This rewriting is particularly problematic in parsing techniques that do not support left recursion, as the resulting parse trees are considerably different from the ones of the original grammar.

```
expr ::= expr '.' field
       | expr expr
       | '-' expr
       | expr '*' expr
       | expr '+' expr
       | expr '-' expr
       | 'if' expr 'then' expr
       | expr ';' expr
       | '(' expr ')'
```

Operator	Associativity
.	–
function appl	left
- (unary)	–
*	left
+ -	left
if	–
;	right

Figure 1. A simplified excerpt of OCaml expression grammar (left), and its corresponding table of operator precedence (right).

Instead of rewriting the grammar, it is more convenient to use an ambiguous grammar and a set of *declarative* constructs to specify operator precedence. Constructing parsers based on declarative specification of operator precedence is a very old research topic, dating back to the work of Floyd [11] on operator precedence grammars in 1963. Floyd's operator precedence grammars are a limited subset of deterministic grammars, and are not used in practice to fully specify a grammar. Rather, they are mainly used in handwritten recursive-descent parsers for efficient parsing of expressions.

One of the most-widely used operator precedence techniques is presented by Aho *et al.* [5]. This approach, which is based on LR parsing and implemented in Yacc [16], maps operator precedence information to shift/reduce conflicts. The Yacc-style operator precedence is efficient and powerful. For example, the parser for OCaml, which has one of the most complicated expression grammars, is written using `ocamlyacc`, an OCaml port of Yacc.

When machine resources were scarce, only deterministic parsing techniques were considered. The success of Yacc [16], and its underlying LR parsing theory [8, 21] that has been developed in the 70s, enabled generation of linear parsers from a BNF grammar specification. Deterministic parsing techniques are efficient, and guarantee that no ambiguity will be left in the grammar. The main problem with deterministic parsing techniques is that they are not expressive enough to support the syntax of programming languages out of the box. This means that the grammar writer needs to massage a grammar into a deterministic form.

As machines became more powerful and the need for front-ends in areas other than traditional compiler construction increased, more expressive parsing techniques were considered. For example, in areas such as source code analysis and development of domain-specific languages (DSLs), it is desirable to quickly construct (prototype) a parser. General parsing algorithms [9, 26, 29] can deal with any context-free grammar, and therefore, free the user from the

PEPM'16, January 18–19, 2016, St. Petersburg, FL, USA
ACM. 978-1-4503-4097-7/16/01...
http://dx.doi.org/10.1145/2847538.2847540

```
expr ::= expr '.' field
       > expr expr                          left
       > '-' expr
       > expr '*' expr                      left
       > (expr '+' expr | expr '-' expr)    left
       > 'if' expr 'then' expr
       > expr ';' expr                      right
       | '(' expr ')'
```

Figure 2. A simplified excerpt of OCaml expression grammar augmented with declarative operator precedence constructs.

restrictions of a particular deterministic parsing technique. Moreover, general parsers can run nearly linearly on grammars of real programming languages, while keeping the cubic bound on worst-case, highly ambiguous grammars [26, 27]. Of course, the machinery of a general parser imposes performance overhead, but the performance of general parsing [17] is not a reason to avoid them.

The main fear of using general parsing algorithms is ambiguity. Sentences can be ambiguous, and it is not always easy to pinpoint the cause of ambiguity and resolve it. Disambiguation is the process of selecting a parse tree from a set of resulting parse trees. Theoretically, disambiguation can be considered as post-parse filtering of a parse forest. However, in practice, it is desirable to apply disambiguation while parsing, to reduce nondeterminism, and to terminate parsing paths that lead to ambiguity as early as possible.

In a declarative syntax definition formalism, such as SDF [19], disambiguation constructs are declared by the user, rather than being imposed by the underlying parsing technique, e.g., order of alternatives in PEGs [12]. For example, in a declarative approach, operator precedence information in Figure 1(right) can be expressed using `>`, **`left`** and **`right`**, as shown in Figure 2. As can be seen, the precedence information is specified using the `>` construct, and it is decreasing: the first alternative has the highest precedence. Associativity is described using **`left`** and **`right`**. In case of `+` and `-`, which have the same precedence, but are left-associative with respect to each other, a left associativity group is used.

We distinguish between the notation, semantics, and implementation of an operator precedence approach. For example, in Yacc, the precedence of operators is globally specified based on tokens, as opposed to Figure 2 where precedence is locally defined for alternatives of an expression nonterminal. A token-based notation for specifying operator precedence has two shortcomings. First, an extra mechanism is needed to distinguish between tokens that have different meanings in different rules, e.g., unary and binary minus. Second, there is no native way to specify the operator precedence of an *invisible* operator. For example, the function application operator, `expr ::= expr expr`, in Figure 1. The semantics of Yacc-style operator precedence is described in terms of shift/reduce conflicts in LR parsing, which is exactly how it is implemented. While Yacc is powerful and widely used, its operator precedence semantics is bound to the internal workings of LR parsing, and cannot be ported to non-LR parsing algorithms.

There has been much work [1, 6, 20, 28, 33] that provide a parser-independent semantics of operator precedence. We discuss these work in detail in Section 5. Among them, SDF2 [33] is of particular importance as it provides an intuitive tree-based semantics of operator precedence, and is implemented in context of general parsing, Scannerless GLR (SGLR) [31]. The operator precedence semantics of SDF2 works for most cases, but in some cases it is too strong, removing sentences from the language when there is no ambiguity, and in some cases, it cannot deal with corner cases of operator precedence in programming languages such as OCaml.

In our previous work [4], we proposed an extension of SDF2 semantics that does not remove sentences from the language when there is no operator precedence ambiguity, while being able to cover corner cases of operator precedence ambiguity in programming languages such as OCaml. This semantics has the same effect as Yacc semantics, but does not depend on a specific aspects of a parsing algorithm, e.g., shift/reduce conflicts in LALR parsing, and can be implemented in the context of different parsing techniques. We proposed a grammar rewriting [4] to implement this semantics. This rewriting preserves the shape of derivation trees. However, it leads to large grammars and introduces unnecessary nondeterminism to the grammar.

Data-dependent grammars [15] extend traditional context-free grammars with arbitrary computation, parametrized nonterminals, variable binding and constraints. Jim *et al.* [15] present the semantics of data-dependent grammars that does not depend on a particular general parsing algorithm, and is rather straightforward. In our recent work [3], we showed that data-dependent grammars can be used as an intermediate layer for parser-independent implementation of various disambiguation strategies. We provided an implementation of the operator precedence semantics of [4], by desugaring the high-level operator precedence notation (`>`, **`left`** ,and **`right`**) to data-dependent grammars [3]. Compared to the grammar rewriting in [4], the desugaring preserves the size of the original grammar, while having comparable performance in practice.

In this paper we extend and improve our previous translation [3] of operator precedence to data-dependent grammars. We extend it to support indirect operator precedence and thus being able to parse OCaml. We improve its performance by using a new strategy for left-recursive nonterminals. Our translation can deal with deep and indirect precedence cases in programming languages such as OCaml. We evaluate our approach by parsing real OCaml and Java source files using the Iguana parsing framework [2, 3].

The rest of this paper is organized as follows. Section 2 describes the problem of operator precedence in parsing. Section 3 introduces our solution to operator precedence using data-dependent grammars. Section 4 presents the evaluation of our technique using grammars of Java and OCaml. Section 5 discusses related work, and Section 6 concludes.

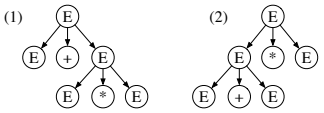
2. The Problem of Operator Precedence

In this section we discuss expression grammars and operator precedence in detail. When explaining the examples, we use two different semantics of operator precedence, namely, Aho *et al.*'s approach [5] based on shift/reduce conflicts, which we refer to as Yacc-style semantics, and the SDF2 semantics [33] based on tree patterns. Understanding the difference between Yacc-style and SDF2-style semantics helps to understand the problem, and motivates our approach to operator precedence in Section 3.

2.1 Binary Operators

Consider a simple expression grammar with two binary operators `'*'` and `'+'`. Parsing `a+a*a` with this grammar results in two derivation trees, corresponding to the groupings `a+(a*a)` and `(a+a)*a`:

```
E ::= E '*' E
    | E '+' E
    | 'a'
```



Based on operator precedence in arithmetics, the first grouping is correct. Both Yacc-style and SDF2-style semantics of operator precedence can deal with this case. We first consider Yacc. This grammar leads to shift/reduce conflicts in the following LR states:

```
(1) E ::= E .'+' E          (2) E ::= E .'*' E
    E ::= E  '+' E.             E ::= E  '*' E.
    E ::= E .'*' E              E ::= E .'+' E
```

In the first state, the shift/reduce conflict between `E ::= E '+' E.` and `E ::= E .'*' E` corresponds to the precedence of `'+'` and `'*'`. If we give `'*'` higher precedence than `'+'`, Yacc resolves this conflict in favor of shifting `'*'`. The shift/reduce conflict between `E ::= E '+' E.` and `E ::= E .'+' E` corresponds to the associativity of `'+'`. If we define `'+'` left-associative, this conflict will be resolved in favor of reduce. The same holds for the second LR state.

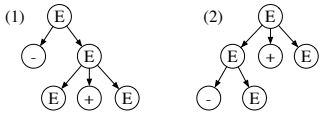
SDF2 [33] uses an operator precedence semantics based on patterns in derivation trees. This parser-independent semantics allows the language engineer to think in terms of tree patterns rather than shift/reduce conflicts. In SDF2, `>` defines a precedence relationship between two alternatives of a nonterminal. For example, `E ::= E '*' E > E ::= E '+' E`[1] means that all `E's` in the body of the `'*'`-rule cannot derive `E ::= E '+' E`. This effectively disallows derivation trees that correspond to the grouping of `'+'` under `'*'`. Associativity in SDF2 is specified using **`left`** and **`right`**. For example, `E ::= E '*' E` **`left`** means that the second `E` in the body of the `'*'`-rule cannot derive itself. SDF2 semantics can be applied during parsing, by modifying parse tables to remove violating derivation trees, or as a post-parse filtering step. In this paper, we are concerned with the semantics of SDF2, and not a particular implementation.

2.2 Unary and Binary Operators

Combining both unary and binary operators makes the implementation of operator precedence more complicated. In this section we consider two common examples: one from the basic arithmetics and one from functional programming languages.

Arithmetics We consider a combination of unary `'-'` and binary `'+'` operators. Parsing `-a+a` with such a grammar results in two derivation trees, corresponding to the groupings `-(a+a)` or `(-a)+a`:

```
E ::= '-' E
    | E '+' E
    | 'a'
```



Based on operator precedence in arithmetics, where unary `'-'` has higher precedence than binary `'+'`, the second derivation tree is correct. Both Yacc-style and SDF2-style semantics can disambiguate this case. The explanation is similar to the one we gave for the binary-only example of the previous section.

Functional languages So far, we focused on expression grammars that have conventional precedence rules as in basic arithmetics. However, in functional programming languages, there are some combinations of precedence rules which are not found in basic arithmetics. In this paper, we focus on OCaml [22], a popular dialect of ML, which allows imperative, object-oriented and functional styles of programming. The syntax of OCaml can be seen as a large expression grammar, as almost each construct is an expression. For example, consider the following conditional expression:

```
if b then x else x + 1
```

In OCaml, `'if-then-else'` acts as a unary prefix operator that has lower precedence than binary infix operators. Therefore, this expression should be grouped as `if b then x else (x + 1)` and not as `(if b then x else x) + 1`.

To observe Yacc's behavior on this grammar, we consider the following conflicting LR states:

```
(1) E ::= 'if' E 'then' E 'else' E.
    E ::= E .'+' E

(2) E ::= E .'+' E
    E ::= E  '+' E.
```

[1] SDF2 adheres to algebraic notation and writes $A ::= \alpha$ as $\alpha \rightarrow A$. In this paper, we use an EBNF-like notation for writing grammar rules.

State (2) corresponds to the associativity of the `'+'` operator, which we discussed in the previous section. State (1) corresponds to the precedence of `'if-then-else'` and `'+'`. As can be seen, in this state we can either reduce the `'if-then-else'` rule or shift `'+'`. Given that `'+'` in OCaml has higher precedence that `'if-then-else'`, the shift action is performed, producing the correct derivation tree.

For this input, the SDF2 semantics can also produce the correct derivation tree. Recall that SDF2 uses `>` to define precedence, and, in this case, `E ::= E '+' E > E ::= 'if' E 'then' E 'else' E` means that the `E`'s in the body of the `'+'`-rule cannot derive `'if-then-else'`, preventing the wrong derivation. We now consider a slightly different input:

```
1 + if b then x else x
```

where the `'if-then-else'` expression is the right operand of `'+'`. This expression is in fact unambiguous and can only have one grouping: `1 + (if b then x else x)`. Yacc can successfully parse this expression, producing the expected derivation tree. However, SDF2 gives a parse error for this example. As `'+'` has higher precedence than `'if-then-else'`, no `E` in the body of the `'+'` can derive `'if-then-else'`, and therefore, this input is rejected. To observe this behavior of the SDF2 semantics, we need a binary operator with higher precedence than a unary operator. In arithmetics, this case only happens for the power operator (`'^'`), for example, for the input `1 ^ - 1`.

The last example shows that in some cases SDF2 semantics is too strong and can remove sentences from the language even if there is no ambiguity. Therefore, the SDF2 semantics for operator precedence is not safe. We call a disambiguation mechanism safe *iff* it does not remove sentences from a language [4]. In other words, when there is no ambiguity, a safe disambiguation mechanism must not apply. Although many disambiguation mechanisms are not safe, for example, longest match, for the operator precedence ambiguity we can ensure safety.

In SDF2, one can fine-tune the behavior of `>` by specifying the exact nonterminal under which filtering should happen. For example, to enforce that filtering should only happen under the left `E`, we can write `E ::= E '+' E <0> > 'if' E 'then' E 'else' E`. Here, `<0>` specifies that filtering should only happen under the nonterminal at position zero. This way of specifying precedence is tedious as the `<i>` operator is not transitive across alternatives, but even if we consider a transitive version, there is another problem with this tree-based semantics of operator precedence: in some cases, this semantic is too weak and cannot disambiguate an operator precedence ambiguity. Consider the following example:

```
1 + if b then x else x + 1
```

According to the precedence and associativity rules in OCaml, this example should be parsed as:

```
(a) 1 + (if b then x else (x + 1))
```

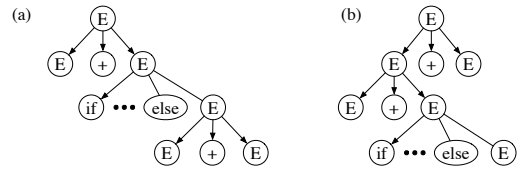
and not as two other alternative derivation trees:

```
(b) (1 + (if b then x else x)) + 1
(c) 1 + ((if b then x else x) + 1)
```

The last two interpretations are precedence-incorrect as they correspond to the cases where `'if-then-else'` binds stronger than `'+'`, and `'+'` is right-associative.

Yacc can deal with this input and produces the expected derivation tree (a). In the conflicting state (1), shown before, Yacc prefers to shift `'+'`, which effectively prevents `'if-then-else'` to bind stronger when it is followed by `'+'`, as in the derivations (b) and (c). In contrast, the SDF2 semantics cannot disambiguate this case, producing the derivation trees (a) and (b). The derivation tree (c) is rejected based on left associativity of `'+'`. To show the parent/child

relationships in derivation trees, the two remaining derivation trees are shown below:



The SDF2 semantics for operator precedence is defined as a one-level relationship between a parent and a child rule. Based on this relationship, a derivation step from a nonterminal in the body of the parent rule is prohibited. If we examine the derivation trees above, the nodes in both trees are precedence correct with respect to their immediate children: `E ::= E + E` does not appear under the rightmost `E` in the `'+'`-rule, and `E ::= 'if' E 'then' E 'else' E` does not appear under the leftmost `E` in the `'+'`-rule.

The last example illustrates that in order to disambiguate this case, a semantics of operator precedence should only restrict derivation of `'if-then-else'` under the rightmost `E` of the `'+'` rule when this `E` is derived from the leftmost `E` of the `'+'`-rule. We call such cases of operator precedence *deep*. Deep cases commonly, but not only, happen when a left-associative binary operator has higher precedence than a unary prefix operator. This also holds for a right-associative binary operator with higher precedence than a unary postfix operator. Such cases do not happen in arithmetics, but are essential to allow natural writing (without parentheses) of expressions in languages such as OCaml.

In a previous work [4], we provided a semantics of operator precedence that can deal with such deep cases (see Section 3.1). We also presented a grammar rewriting technique [4] that implements this semantics. There are mainly two problems with this rewriting technique. First, the size of the generated grammar can be rather large. We have not done a formal analysis, but it appears that the size of the generated grammar is quadratic with respect to the original grammar. To preserve the shape of the original derivation trees, many intermediate nonterminals are introduced. These intermediate nonterminals may introduce nondeterminism into the grammar, and lead to inefficiency in parsing (see Section 3.6).

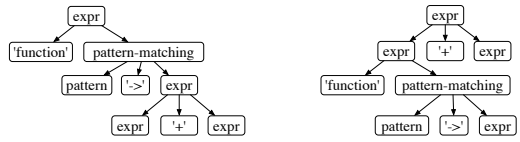
2.3 Indirect Recursive Nonterminals

Dealing with deep cases alone is not enough to resolve all precedence ambiguities in OCaml. For example, consider the following (simplified) rules from the OCaml language specification:

```
expr ::= expr '+' expr
       | 'match' expr 'with'  pattern-matching
       | 'function' pattern-matching
       | 'try' expr 'with' pattern-matching

pattern-matching ::= pattern '->' expr
```

As can be seen, the common pattern matching syntax is factored out into a separate `pattern-matching` nonterminal. As `pattern-matching` ends with `expr`, it can cause precedence ambiguity. All these unary prefix operators, `'match'`, `'function'`, and `try`, have lower precedence than binary operators in OCaml, thus, an input string such as `function x -> x + 1` should be parsed as `function x -> (x + 1)`, and not as `(function x -> x) + 1`:



In a declarative syntax formalism with support for operator precedence, we would like the following definition to be able to return the correct derivation tree.

```
expr ::= expr '+' expr
       > 'function' pattern-matching
```

In [4], we only conjectured on the implementation of such indirect cases, by copying the full grammar part reachable from an indirect nonterminal and rewriting left or rightmost recursive ends. In this paper, we show a dynamic, more systematic way of dealing with indirect cases of operator precedence. It should be noted that the Yacc-style semantics can deal with indirect cases, for example, in this case, when the token `'->'` is given higher precedence than `'+'` (Yacc considers the precedence of the last terminal of a rule as the precedence of the rule). The reason why Yacc can deal with indirect cases directly corresponds to how an LR automaton is constructed, more specifically closure on LR items. In our example, the closure on the item `pattern-matching ::= pattern '->' . expr` imports the item `expr ::= . expr '+' expr`, leading to the following LR state after a transition on `expr`:

```
pattern-matching ::= pattern '->' expr .
expr ::= expr .'+' expr
```

This state leads to a shift/reduce conflict that can be resolved based on the precedence relationship between `'+'` and `'->'`.

2.4 Discussion

So far, we discussed the problem of operator precedence in parsing using Yacc and SDF2 as two leading semantics. The Yacc-style semantics is safe, and can deal with deep and indirect cases. However, this semantics is bound to the inner workings of LR parsing, and cannot be ported to other non-LR parsing techniques. In fact, Yacc was designed to work with LALR grammars, not arbitrary context-free grammars. As a result, for example, the Yacc-style operator precedence cannot be used in a scannerless GLR parser that inserts layout (whitespace and comment) between symbols.

Our goal is to provide a declarative semantics of operator precedence, so that the grammar writer can think in terms of the grammar, rather than inner workings of a parsing algorithm. In this paper we use a semantics of operator precedence that is defined in terms of derivation trees. Our semantics (Section 3.1) can be seen as an extension of SDF2 semantics, that makes it safe, and allows for deep and indirect operator precedence cases. The main contribution of this paper is how to implement this semantics using data-dependent grammars. Our implementation preserves the size of the original grammar, does not depend on a particular parsing algorithm and is efficient.

3. Operator Precedence for Data-dependent Grammars

3.1 Notation and Semantics for Operator Precedence

We use `>` as a high-level construct for declarative specification of operator precedence. In our semantics, `>` defines a partial order on alternatives of a nonterminal. For any two grammar rules r_1 and r_2 with the same head E, $r_1 > r_2$ applies if one of the rules is left-recursive ($E ::= E\beta$) and the other is right-recursive ($E ::= \alpha E$). This means that rules that are neither left- nor right-recursive, e.g., `E ::= '(' E ')'` are not affected by `>`.

We define operator precedence as a relationship between alternatives of a nonterminal, and not tokens. We consider three possible types of rules. Unary prefix rules ($E ::= \alpha E$) where α is nonempty and does not start with E, unary postfix rules ($E ::= E\beta$) where β is nonempty and does not end with E, and binary rules of the form

```
E ::= E '.' Id                E(p) ::= [3>=p] l=E(p) [l==0||l>=3] '.' Id        {0}                    // 3
    | E '.[' E ']'                     | [3>=p] l=E(p) [l==0||l>=3] '.[' E(0) ']' {0}                    // 3
    > E '+' E                          | [2>=p] l=E(p) [l==0||l>=2] '+' r=E(2)    {r==0 ? 2 : min(r,2)} // 2
    > 'if' E 'then' E                  |        'if' E(0) 'then' E(1)            {1}                    // 1
    | '(' E ')'                        |        '(' E(0) ')'                     {0}                    // -
    | 'a'                              |        'a'                              {0}                    // -
```

Figure 3. Translation of precedence rules into data-dependent grammars.

$E ::= E\gamma E$, where γ is a possibly empty sequence of symbols. In this setting, α, β and γ act as operators.

The reason why we only consider left- and right-recursive rules is that only left- and right-recursive ends can participate in an operator precedence ambiguity. In an operator precedence ambiguity involving two operators, the derivations differ in steps corresponding to the order of application of the respective operator rules. For example, for the binary operators in Section 2.1 we have the following leftmost derivations for the input `a+a*a`:

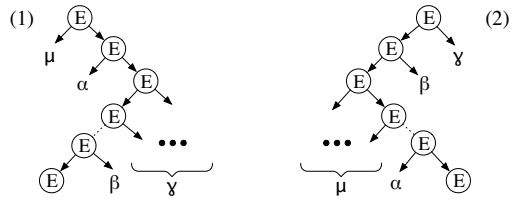
$$E \Rightarrow E+E \Rightarrow a+E \Rightarrow a+E*E \quad E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow a+E*E$$

The presence of two derivations for an operator ambiguity, and the fact that `>` only removes one of them, is the basic reasoning behind the safety of our operator precedence technique. More generally, for any two left- and right-recursive rules $E ::= E\beta$ and $E ::= \alpha E$ we have the following two leftmost derivations:

$$(1)\ \mu E \Rightarrow \mu\alpha E \overset{*}{\underset{lm}{\Rightarrow}} \mu\nu E\gamma \Rightarrow \mu\nu E\beta\gamma$$

$$(2)\ E\gamma \Rightarrow E\beta\gamma \overset{*}{\underset{lm}{\Rightarrow}} \mu E\beta\gamma \Rightarrow \mu\alpha E\beta\gamma \overset{*}{\underset{lm}{\Rightarrow}} \mu\nu E\beta\gamma$$

where both derivations have identical sub-derivations $\alpha \overset{*}{\Rightarrow} \nu$. The parse trees corresponding to these derivations have the shapes:



In the first parse tree, β binds stronger than α, and vice versa for the second parse tree. As can be seen, there can be an arbitrary distance ($\overset{*}{\Rightarrow}$) between the application of $E ::= \alpha E$ and $E ::= E\beta$. This captures the deep cases of operator precedence. If $E ::= \alpha E > E\beta$, the first derivation tree should be removed and vice versa. We discuss indirect cases in Section 3.5.

The semantics of associativity constructs, **`left`** and **`right`**, is similar to the one of the precedence. However, in contrast to precedence, we define associativity to affect only binary operators and apply only at one level and not arbitrary deep. The latter decision is based on the fact that we could not find a practical example, where deep application of associativity rules was useful, and where other precedence mechanism could not be used instead. We also support the **`nonassoc`** notation for defining non-associativity, e.g., for `E ::= E '>' E`, where no nesting of the same rule is allowed. In contrast to **`left`** and **`right`**, **`nonassoc`** is not safe as it removes sentences from the language.

3.2 Data-dependent Grammars

Data-dependent grammars [15] are an extension of context-free grammars that support arbitrary computation, parameters, variable binding, and constraints. In a context-free grammar, a rule is defined as $A ::= \alpha$, where A (head) is a nonterminal, and α (body) is a possibly empty sequence of terminal and nonterminals. Data-dependent grammars allow definition of parametrized nonterminals, e.g., $A(p)$, similar to the way a function is defined. In addition to terminal and nonterminals, the body of rules in data-dependent grammars can have the following new types of symbols:

- Constraints of the form $[c]$. If c evaluates to false, the current parsing path terminates.
- Bindings of the form $x = A(a)$, where a is an argument to A, and x is a variable holding the value returned by the call $A(a)$.
- Arbitrary expressions of the form $\{e\}$.

Our data-dependent framework [3] also supports return values. An expression $\{e\}$ as the last symbol of a rule defines the return value.

Jim *et al.* [15] introduce the data-dependent automata to represent data-dependent grammars and use these automata to provide a stack-evaluation-based, nondeterministic operational semantics for data-dependent grammars. This semantics is very intuitive. For example, consider the following definition of a fixed-length iteration of a nonterminal `A`:

```
Iter(n) ::= [n > 0] Iter(n - 1) A | [n == 0] ε
```

Here, `Iter` gets an integer parameter `n` which is used to determine the choice of the alternative. If `n > 0`, the first alternative is selected, otherwise, the second. Using this definition, `Iter(5)` is much like a function call in a backtracking, recursive-descent parser. Direct implementation of this semantics of data-dependent grammars can result in exponential runtime and nontermination in presence of left-recursive rules. Therefore, to implement data-dependent grammars, Jim et al. use a modified Earley parsing algorithm, and we use our modified GLL parsing algorithm [2, 3].

3.3 Precedence

In this section, we show how grammars that specify precedence rules using `>` are translated into data-dependent counterparts. In our discussion we use the grammar of Figure 3 (left) as a running example. The translation scheme consists of the following steps:

Assign a precedence level to a rule A number pr_i, the *precedence level* of a rule, is assigned to each left- and/or right-recursive alternative of a nonterminal, where i indicates the i-th alternative. Numbering follows the reverse order of the alternatives and uses the current value of a counter. The counter starts from 1 and increments each time `>` is encountered.

In Figure 3 (left), nonterminal `E` has four left- and/or right-recursive alternatives. The first two (from the top) act as postfix operators, the third as a binary operator, and the fourth as a prefix operator. The number assigned to each alternative is shown in the comment next to the alternative (Figure 3, right). There are two observations. First, as precedence rules do not apply between two postfix or two prefix operators, `|` is used instead of `>` between the first two alternatives, resulting in the same number assigned to both of them. Second, the alternatives that are not left- or right-recursive do not get a number.

Pass a precedence level The nonterminal gets a parameter p, so that the precedence level of an alternative can be passed. The ar-

guments to the left- and/or right-recursive ends of the i-th alternative are defined as follows. Each right-recursive alternative passes its precedence level to its right end: $E(p) ::= \alpha E(\mathrm{pr}_i)$. Each left-recursive alternative passes the precedence level of a parent alternative to its left end: $E(p) ::= E(p)\alpha$. The argument 0 is passed to the nonterminal when it occurs at a position other than the left or right recursive end, i.e., where the precedence rules do not apply. In Figure 3 (right), `p` is passed to all the left ends, `2` to the right end of binary `'+'`, and `1` to the right end of prefix `'if'`.

Return a precedence level In addition to passing the precedence level to its right end, each right-recursive alternative also returns a value that depends on its precedence level. We distinguish the following two cases for a right-recursive alternative:

1. If there is a prefix operator of lower precedence than the alternative, the return expression is defined as follows:

$E(p) ::= \alpha\ r{=}E(\mathrm{pr}_i)\ \{r{=}0\ ?\ \mathrm{pr}_i : \min(r,\mathrm{pr}_i)\}$,

where variable r holds the value returned by the call to the right end. The construct _?_ : _ defines a conditional expression such that if the value of r is equal to 0, the alternative returns its precedence level, otherwise the value is defined as $\min(r,\mathrm{pr}_i)$. Intuitively, min will propagate the lowest precedence level upwards, in the chain of the recursive calls corresponding to the rightmost recursive ends.

2. If there is no such a prefix operator, the alternative simply returns its precedence level: $E(p) ::= \alpha E(\mathrm{pr}_i)\ \{\mathrm{pr}_i\}$.

Finally, we also need to provide a default return value for alternatives that are not right recursive, i.e., postfix operators or alternatives without recursive ends. We use 0 as the default value.

In Figure 3 (right), only the binary and prefix operators return a non-zero value: prefix `'if'` returns its precedence level, and binary `'+'` returns `r==0 ? 2 : min(r,2)` as there is prefix `'if'` which is of lower precedence than binary `'+'`.

Add constraints to a rule based on its precedence level Finally, each left-recursive alternative gets two constraints:

$E(p) ::= [\mathrm{pr}_i \geq p]\ l{=}E(p)\ [l{=}0 \,\|\, l \geq \mathrm{pr}_i]\ \alpha$,

where variable l holds the value returned by the call to the left end. The first constraint, $[\mathrm{pr}_i \geq p]$, is a precondition to the alternative. This constraint effectively excludes the current left-recursive alternative from the right end of a parent right-recursive alternative if the current alternative is of lower precedence than the parent one. Passing 0 to E makes any precondition true, therefore we refer to $E(0)$ as the unrestricted use of E. The second constraint, $[l{=}0 \,\|\, l \geq \mathrm{pr}_i]$, is a postcondition to the first symbol of the alternative. This constraint terminates the current left-recursive alternative if the value produced by the left end corresponds to a child alternative of lower precedence than the current alternative.

3.3.1 Discussion on Semantics and Implementation

We now discuss how our data-dependent encoding prevents undesired, precedence-violating derivation trees by restricting the left and right end of an alternative. Similar to Section 3.1, we consider a grammar with a left-recursive alternative $E ::= E\beta$ and a right-recursive alternative $E ::= \alpha E$. In addition, we assume that the grammar has other left- and right-recursive alternatives: $E ::= E\sigma_i$, $1 \leq i \leq m$, and $E ::= \gamma_j E$, $1 \leq j \leq k$. We use the following leftmost derivations:

(1) $E \Rightarrow E\beta \overset{*}{\underset{lm}{\Rightarrow}} \mu E\beta \Rightarrow \mu\alpha E\beta$, where $\overset{*}{\Rightarrow}$ indicates zero or more intermediate steps deriving the right-recursive alternatives $E ::= \gamma_j E$, such that $\gamma_1 \overset{*}{\underset{lm}{\Rightarrow}} \nu_1, \ldots, \gamma_k \overset{*}{\underset{lm}{\Rightarrow}} \nu_k$, and $\mu = \nu_1 \ldots \nu_k$ is a possibly empty sequence of terminals.

(2) $E \Rightarrow \alpha E \overset{*}{\underset{lm}{\Rightarrow}} \nu E \overset{*}{\underset{lm}{\Rightarrow}} \nu E\sigma \Rightarrow \nu E\beta\sigma$, where $\alpha \overset{*}{\underset{lm}{\Rightarrow}} \nu$, and the second $\overset{*}{\Rightarrow}$ indicates zero or more intermediate steps deriving the left-recursive alternatives $E ::= E\sigma_i$, such that $\sigma = \sigma_m \ldots \sigma_1$ is a possibly empty sequence of terminals and nonterminals.

First, we consider derivation (1) and the case of direct nesting, i.e., zero intermediate steps. According to the precedence semantics of Section 3.1, the derivation step $E\beta \Rightarrow \alpha E\beta$ should only be valid if $E ::= \alpha E$ has the same or higher precedence than $E ::= E\beta$. We now look at how our encoding to data-dependent grammars achieves this. In our translation scheme, each left-recursive alternative $E ::= E\beta$ is translated into:

$E(p) ::= [\mathrm{pr}_\beta \geq p]\ l{=}E(p)\ [l{=}0 \,\|\, l \geq \mathrm{pr}_\beta]\ \beta$.

We consider an unrestricted call $E(0)$ (restricted calls $E(\mathrm{pr})$, $\mathrm{pr} > 0$, are discussed later in the context of derivation (2)). The precondition evaluates to true, and the alternative can only succeed if the postcondition to the recursive call is also true. As postfix operators and alternatives without recursive ends return 0, the postcondition permits these forms of alternatives at the left end. The postcondition, however, permits a right-recursive alternative $E(p) ::= \alpha E(\mathrm{pr}_\alpha)\ \{\mathrm{pr}_\alpha\}$ only if it is of the same or higher precedence than the current alternative, thus enforcing $\mathrm{pr}_\alpha \geq \mathrm{pr}_\beta$.

To enforce precedence rules at arbitrary depth (the case of multiple intermediate steps in the derivation), our translation scheme introduces min to the return expression of a right-recursive alternative. This propagates the lowest precedence level upwards, in the chain of the recursive calls corresponding to the rightmost recursive ends, to the left end of the current alternative. For example, consider the following chain of such recursive calls, where each call ($\rightarrow$) is shown in the context of the respective right-recursive alternative and is made from its right end:

$\gamma_1\ r_1{=}E(\mathrm{pr}_{\gamma_1}) \rightarrow \cdots \rightarrow \gamma_k\ r_k{=}E(\mathrm{pr}_{\gamma_k}) \rightarrow \alpha\ r{=}E(\mathrm{pr}_\alpha)$.

Here, r_j, $1 \leq j \leq k$, and r hold the value returned by the respective call. As our translation adds return expression $r{=}0\ ?\ \mathrm{pr}_{\gamma_j} : \min(r,\mathrm{pr}_{\gamma_j})$ to right-recursive rule $E ::= \gamma_j E$, these calls, when return, produce the following bindings: $r_k = \min(r,\mathrm{pr}_\alpha)$, $r_{j-1} = \min(r_j,\mathrm{pr}_{\gamma_j})$, $2 \leq j \leq k$, and finally, $l = \min(r_1,\mathrm{pr}_{\gamma_1})$. Thus, the postcondition above also requires that all $E ::= \gamma_j E$ and $E ::= \alpha E$ are of the same or higher precedence than $E ::= E\beta$.

The observant reader will note, however, that the return expression of a right-recursive alternative depends on whether there is a prefix operator of lower precedence. Below we discuss the role of the precondition to a left-recursive alternative. After that, it can be seen that it is sufficient to simply return the precedence level for a right-recursive alternative (no min is needed) if there is no prefix operator of lower precedence than the alternative.

Now, we consider derivation (2) and the case of zero intermediate steps in the second $\overset{*}{\Rightarrow}$. According to the precedence semantics of Section 3.1, the last derivation step should only be valid if $E ::= E\beta$ has the same or higher precedence than $E ::= \alpha E$. In our translation, each right-recursive alternative passes its precedence level to the right end: $E(p) ::= \alpha E(\mathrm{pr}_\alpha)$ (for brevity, we omitted the return expression). Given that only left-recursive alternatives are guarded with a precondition, call $E(\mathrm{pr}_\alpha)$ will try all prefix operators and alternatives without recursive ends. However, the call will only try the left-recursive alternative $E ::= E\beta$ if its precondition is true (see the translation above), thus enforcing $\mathrm{pr}_\beta \geq \mathrm{pr}_\alpha$.

To enforce precedence rules at arbitrary depth (the case of multiple intermediate steps in the second $\overset{*}{\Rightarrow}$), if the precondition to the left-recursive alternative is true, the precedence level of the parent alternative is passed to the left end of the current alternative. This way, the precedence level of the parent alternative, pr_α, also restricts the chain of recursive calls corresponding to the leftmost recursive ends: $E(\mathrm{pr}_\alpha)\sigma_1 \rightarrow \cdots \rightarrow E(\mathrm{pr}_\alpha)\sigma_m \rightarrow E(\mathrm{pr}_\alpha)\beta$, where each consecutive call to E is made from the left end of the respective left-recursive alternative. In our translation, each of these calls

is guarded by the precondition: $\mathrm{pr}_{\sigma_i} \geq p$, $1 \leq i \leq m$, and $\mathrm{pr}_\beta \geq p$, thus enforcing all $E ::= E\sigma_i$ and $E ::= E\beta$ to be of the same or higher precedence than $E ::= \alpha E$.

3.4 Associativity

Using data dependency, associativity rules can be encoded in a similar way to precedence. We consider left- and right-associative rules, declared using **`left`** and **`right`**, and non-associative rules, declared using **`nonassoc`**. Our general scheme to handle both precedence and associativity consists of the following steps:

Assign a unique number to a rule specifying associativity We use the same counter as before. However, now, the counter also increments when an alternative specifying associativity is encountered, but only if this alternative shares the same precedence with the next or previous alternative. The current value of the counter is then assigned to the alternative, thus giving it a unique number within the same precedence group. All the alternatives within the same precedence group that do not specify associativity are assigned the same number. Consider $E ::= \alpha_4 > \alpha_3 \mid \alpha_2$ left $\mid \alpha_1 > \alpha_0$, where $\mid$ binds stronger than $>$, each $E ::= \alpha_i$ is left- and/or right-recursive, and $E ::= \alpha_2$ is binary. If the value of the counter when encountering the first $>$ (in the reverse order of the alternatives) is 1, the counter increments, and the number assigned to $E ::= \alpha_1$ is 2. The next alternative, $E ::= \alpha_2$, specifies associativity and has the same precedence as $E ::= \alpha_1$ and $E ::= \alpha_3$. Thus, the counter increments again, and the number assigned to $E ::= \alpha_2$ is 3. $E ::= \alpha_3$ is assigned 2, which is the same as for $E ::= \alpha_1$.

This way, alternatives of the same precedence are now described by a range $[\mathrm{pr}_i, \mathrm{pr}_j]$, $\mathrm{pr}_i \leq \mathrm{pr}_j$, $i \leq j$, where the i-th alternative is the first alternative after the last occurrence of >, and the j-th alternative is the alternative with the largest number before the next occurrence of >. We use $\mathrm{pr}_k \in [\mathrm{pr}_i, \mathrm{pr}_j]$ to refer to the number assigned to the alternatives that do not specify associativity within the group.

Pass the rule's unique number along with the precedence level The nonterminal gets two parameters, the first one to pass a precedence level, and the second one to pass its unique number. If a binary alternative is defined as left- or non-associative, its unique number is passed to its right end along with its precedence level: $E(p, p') ::= \alpha\, E(\mathrm{pr}_k, \mathrm{pr}_i)$, where pr_k is used as the alternative's precedence level. Otherwise, 0 is passed as the second argument to the right end: $E(p, p') ::= \alpha\, E(\mathrm{pr}_k, 0)$.
All left-recursive rules of the nonterminal, i.e., binary and postfix operators, pass 0, along with the precedence level of a parent alternative, to its left end: $E(p, p') ::= E(p, 0)\, \alpha$. Passing 0 to the left end of an alternative prevents deep application of associativity rules, in contrast to precedence.

Return the rule's unique number along with the precedence level If a binary alternative is defined as right- or non-associative, its unique number is returned along with its precedence level. If there is no prefix operator of lower precedence than the alternative:

$E(p, p') ::= \alpha\ E(\mathrm{pr}_k, 0)\ \{(\mathrm{pr}_k, \mathrm{pr}_i)\}$ (right-associative)
$E(p, p') ::= \alpha\ E(\mathrm{pr}_k, \mathrm{pr}_i)\ \{(\mathrm{pr}_k, \mathrm{pr}_i)\}$ (non-associative)

where $(\mathrm{pr}_k, \mathrm{pr}_i)$ is a tuple expression. If there is such a prefix operator, the return expressions above are replaced with $(r.1 = 0\ ?\ \mathrm{pr}_k : \min(r.1, \mathrm{pr}_k), \mathrm{pr}_i)$, where variable r (see Section 3.3) holds the value returned by the call to the right end, and $r.1$ accesses the first element of the tuple. For all the other alternatives of the nonterminal, 0 is used as the second element of the tuple.

Add constraints to the rule based on its unique number If a binary alternative is defined as left- or non-associative, precondition $p' \neq \mathrm{pr}_i$ is also added to the alternative, resulting in $[\mathrm{pr}_k \geq p,\ p' \neq \mathrm{pr}_i]$, where the comma inside the brackets defines logical AND. If a binary alternative is defined as right- or non-associative, postcondition $l.2 \neq \mathrm{pr}_i$ is added to the alternative, resulting in $[l.1 = 0 \,\|\, l.1 \geq \mathrm{pr}_k,\ l.2 \neq \mathrm{pr}_i]$, where variable l (see Section 3.3) holds the value returned by the call to the left end, and $l.1$ and $l.2$ access the first and second elements of the tuple, respectively.

```
expr(p) ::=
   [7>=p] l=expr(p) [l==0||l>=7] '.' field {0}
 | [6>=p] l=expr(p)[l==0||l>=6] r=expr(7)
                                  {(r==0)? 6 : min(r,6)}
 |        '-' r=expr(5)           {(r==0)? 5 : min(r,5)}
 | [4>=p] l=expr(p) [l==0||l>=4] '*' r=expr(5)
                                  {(r==0)? 4 : min(r,4)}
 | [3>=p] l=expr(p) [l==0||l>=3] '-' r=expr(4)
                                  {(r==0)? 3 : min(r,3)}
 | [3>=p] l=expr(p) [l==0||l>=3] '+' r=expr(4)
                                  {(r==0)? 3 : min(r,3)}
 |        'if' expr(0) 'then' expr(2)          {2}
 | [1>=p] l=expr(p) [l==0||l>=2] ';' expr(1)   {1}
 |        '(' expr(0) ')'                      {0}
```

Figure 4. The translation of the OCaml excerpt from Figure 2 into a data-dependent grammar.

Associativity groups The general scheme above is also applicable for binary alternatives forming an associativity group. For example, two binary operators, such as + and - (Figure 2), can be specified to be left-associative with respect to each other. In such cases, the left or/and right ends of a binary alternative in an associativity group must exclude the other binary alternatives of the group including the alternative itself. To encode this, all the binary alternatives of the associativity group are assigned the same unique number, say pr_m. This way, associativity related constraints, $p' \neq \mathrm{pr}_m$ and $l.2 \neq \mathrm{pr}_m$, effectively exclude all the alternatives of an associativity group from the left and/or right ends of the alternatives.

3.4.1 Optimization

Is it always necessary to operate with two arguments and tuples when both precedence and associativity rules are used? The answer is no. The translation can use one argument and return a single number when alternatives specifying associativity do not share the same precedence with other alternatives, and, in case of an associativity group, when alternatives inside the group do not share the same precedence with alternatives outside the group. This corresponds to cases where $\mathrm{pr}_i = \mathrm{pr}_j$, $i \leq j$, and when our general scheme produces, for example, the following translation for a left-associative alternative (here, we assume the case of no prefix operator of lower precedence):

$E(p, p') ::= [\mathrm{pr}_i \geq p, p' \neq \mathrm{pr}_i]\ \alpha\ E(\mathrm{pr}_i, \mathrm{pr}_i)\ \{(\mathrm{pr}_i, 0)\}$

Instead, in such cases, we simplify the translation to:

$E(p) ::= [\mathrm{pr}_i \geq p]\ \alpha\ E(\mathrm{pr}_i{+}1)\ \{\mathrm{pr}_i\}$

This translation uses only one argument and passes the precedence level plus one to the right end, thus also excluding the alternative itself and disallowing right-associative derivation trees. The call $E(\mathrm{pr}_i{+}1)$ will propagate its argument to the left end of left-recursive alternatives. However, in this case, it cannot lead to deep application of the associativity rule, as only left-recursive alternatives of (strictly) higher precedence than $E ::= \alpha E$ can be tried, thus disallowing deep nesting of $E ::= \alpha E$. Similarly, for the left end of a right-associative alternative, the simplified translation is:

$E(p) ::= [\mathrm{pr}_i \geq p]\ l{=}E(p)\ [l = 0 \,\|\, l \geq \mathrm{pr}_i{+}1]\ \alpha,$

where $\mathrm{pr}_i{+}1$, the lower bound on l, excludes from the left end alternatives of lower precedence and the alternative itself, thus

disallowing left-associative derivation trees. The translation of the OCaml excerpt from Figure 2 into a data-dependent grammar is shown in Figure 4. This translation requires only one parameter.

3.5 Support for Indirect Cases

To extend the derivations of Section 3.1 to indirect cases, we consider more general forms of left- and right-recursive rules: $E ::= Y\beta$ and $E ::= \alpha X$, where $Y \underset{lm}{\overset{*}{\Rightarrow}} E\sigma$ and $X \underset{lm}{\overset{*}{\Rightarrow}} \tau E$. Then, we have:

(1) $\mu E \Rightarrow \mu\alpha X \underset{lm}{\overset{*}{\Rightarrow}} \mu\nu\tau E \underset{lm}{\overset{*}{\Rightarrow}} \mu\nu\tau E\gamma \Rightarrow \mu\nu\tau Y\beta\gamma \underset{lm}{\overset{*}{\Rightarrow}} \mu\nu\tau E\sigma\beta\gamma$

(2) $E\gamma \Rightarrow Y\beta\gamma \underset{lm}{\overset{*}{\Rightarrow}} E\sigma\beta\gamma \underset{lm}{\overset{*}{\Rightarrow}} \mu E\sigma\beta\gamma \Rightarrow \mu\alpha X\sigma\beta\gamma \underset{lm}{\overset{*}{\Rightarrow}} \mu\nu\tau E\sigma\beta\gamma$

In other words, nonterminals Y and X indirectly derive the left and right E-end, respectively. In addition, sub-derivations corresponding to the second and the last $\overset{*}{\Rightarrow}$ in (1) and (2) permit multiple intermediate nonterminals. Thus, in general, the rules containing the left and right E-end, such as $Z ::= E\theta_1$ and $W ::= \theta_2 E$, may have a different head (Z and W) than Y and X.

In our translation, we rely on reachability analysis that computes indirectly derivable left and right ends. The basic idea of our translation is to propagate the precedence level to/from the nonterminal's indirect left ($Z ::= \underline{E}\theta_1$) and right ($W ::= \theta_2\underline{E}$) ends via intermediate nonterminals by passing and returning values. Our translation adds parameter p_E to Y, X, Z and W. The argument passed to X or Y, and propagated via W or Z, depends on how X or Y is used in E. We distinguish two cases: (a) X and Y represent the same nonterminal that occurs in E as both the left and right end; and (b) X and Y represent distinct nonterminals, Y occurs in E only as the left end, and X only as the right end. In case (a), p_E is a tuple that encodes the left and right uses of X as follows:

$E(p) ::= X((p,\$))\,\beta, \quad E(p) ::= \alpha X((\$, \mathrm{pr}_\alpha))$,

where the first element of the tuple is undefined (we use \$ for undefined values) when X is the right end, and the second element of the tuple is undefined when X is the left end. The other elements of the tuple are defined as if X was E. These arguments are propagated to Z and W, via X, and are used as follows:

$Z(p_E) ::= E(p_E.1 = \$\,?\,0 : p_E.1)\,\theta_1$

$W(p_E) ::= \theta_2\, E(p_E.2 = \$\,?\,0 : p_E.2)$

where $p_E.1$ and $p_E.2$ access the first and the second element of the tuple, respectively. In other words, the left end of $Z ::= E\theta_1$ is only restricted if X is called as the left end, and the right end of $W ::= \theta_2 E$ is only restricted if X is called as the right end.

In case (b), it is possible to directly use p and pr_α without the need to introduce a tuple. In the following, we only focus on case (a) as a more general case. Also, we only consider the case of one parameter to E as our discussion can be straightforwardly generalized to the case of two parameters.

In addition to getting parameters and arguments, X, Z and W return values. This way, the precedence level can be propagated upwards from the indirect left and right ends, via Z or W, to the uses of X. In case (a), the return values are also tuples:

$Z(p_E) ::= l{=}E(p_E.1 = \$\,?\,0 : p_E.1)\ \theta_1\ \{(l,\$)\}$

$W(p_E) ::= \theta_2\ r{=}E(p_E.2 = \$\,?\,0 : p_E.2)\ \{(\$, r)\}$

and are used in E as follows:

$E(p) ::= x{=}X((p,\$))\ [x.1{=}\$\,\|\,(\mathrm{pr}_\beta \geq p,\ x.1{=}0\,\|\,x.1 \geq \mathrm{pr}_\beta)]\ \beta$

$E(p) ::= \alpha\, x{=}X((\$, \mathrm{pr}_\alpha))\ \{x.2{=}\$\,?\,0 : (x.2{=}0\,?\,\mathrm{pr}_\alpha : \min(x.2, \mathrm{pr}_\alpha))\}$

where $x.1{=}\$$ and $x.2{=}\$$ check the presence of the indirect left and right end, respectively. In the second case, the check affects the return value: 0 if the value of $x.2$ corresponds to an alternative without the indirect right end for E, otherwise the value computed as if X was E (here, we only show the return expression for the case where there is a prefix operator of lower precedence than $E ::= \alpha X$). In the first case, the check affects pre- and postconditions. In the general case, the condition $\mathrm{pr}_\beta \geq p$ has to become a postcondition, except for the case when $x.1$ is never equal to \$, and none of pre- and postconditions is triggered if the value of $x.1$ corresponds to an alternative without the indirect left end for E.

3.6 Comparison with our Previous Translation Scheme

Finally, we discuss the design decision in our translation scheme that relates to the use of both parameters and return values. The use of return values is the main difference between the translation scheme we propose in this paper and our previous work [3]. Specifically, to restrict the left and right ends of a nonterminal, we pass an argument to a right end, and propagate the argument passed by a parent alternative and use return values to restrict a left end. General parsing algorithms are efficient in dealing with left-recursive rules. For example, in GLL, left recursion is terminated after the first recursive call at the same input position, allowing non-left-recursive rules to produce results. Then, the left-recursive rules are re-tried, in a form of a loop, as long as new results can be produced.

Our experience with the grammar rewriting technique of [4] shows that the introduction of new, indexed nonterminals for the left ends, which also involves copying the rules to the new nonterminals, directly affects the efficiency in dealing with left recursion. In particular, it increases the stack of leftmost calls, corresponding to the new nonterminals, and does not allow sharing of parsing results corresponding to the copied rules. Our previous translation scheme introduces the same inefficiency problem as the rewriting technique. In that scheme, we do not use return values to restrict left ends, and the use of parameters and arguments for left ends essentially simulates introduction of indices to the left ends.

In contrast to the rewriting and our previous scheme, our current translation does not increase the stack of leftmost calls, as the argument of a parent alternative is passed to the left ends, thus allowing termination of left recursion as soon as possible. When the left-recursive alternatives are further re-tried in a loop, there is just an extra, precedence-related condition that needs to be checked. For right-recursive rules, however, the context of the current alternative can be used to restrict its right end, and therefore, our translation passes the precedence level to the right end of the alternative. In practice, we observed that the median and maximum speedup of parsers for Java using our new translation compared to the rewriting and previous translation are (1.5, 2.5) and (1.7, 3), respectively.

4. Evaluation

In this section we evaluate the performance of our approach to operator precedence using the Iguana parsing framework [3][2], an implementation of data-dependent grammars on top of the GLL parsing algorithm. For the evaluation we use the grammars of Java and OCaml[3]. The experiments were carried out on a machine with a quad-core Intel Core i7 2.6 GHz CPU and 16 GB of memory, running Mac OS X 10.10.5 and a 64-Bit Oracle HotSpot™ JVM version 1.8.0_51. Each file was parsed 10 times and the mean running time (CPU user time) was reported. The three first runs of each file were skipped to allow for JIT optimization.

4.1 Java

We have chosen the grammar of Java 7 from the main part of the Java language specification [13]. This grammar has an unambiguous, left-recursive expression part that encodes operator precedence by introducing new nonterminals for each precedence level. We have replaced the expression part of the Java specification grammar with a natural expression grammar, and specified operator prece-

[2] https://github.com/iguana-parser

[3] https://github.com/iguana-parser/grammars

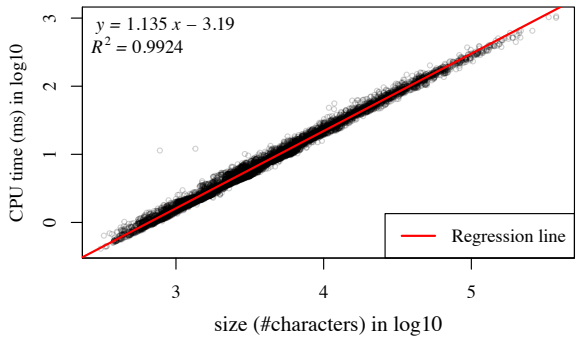


Figure 5. Running time of Iguana on the natural grammar of Java.

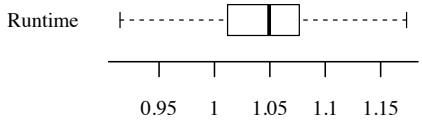


Figure 6. Runtime performance of Iguana using the natural grammar of Java vs. the specification grammar of Java.

dence and associativity using `>`, **`left`** and **`right`**. We parsed 7449 files from the source distribution of JDK 1.7.0_60-b19. All files parsed successfully and without ambiguity.

Figure 5 shows the results of parsing Java files in a log-log (base 10) plot. The goodness of the fit is indicated by the R^2 value of 0.9924, and the equation of the regression line (log-log scale) is written in the plot. As the regression is calculated after a log transform of the original data, and the coefficient (1.135) is close to 1, we can conclude that the parser runs nearly linearly ($y \approx x^{1.135}$) on the natural grammar of Java.

To compare the speed of parsing with the natural grammar of Java that handles operator precedence at runtime, and the specification grammar of Java, we ran Iguana on the same set of 7449 Java source files. The relative runtime performance of the parser for the natural grammar of Java vs. the parser for the specification grammar of Java is shown in Figure 6. As can be seen, the median difference is 1.05, meaning that our approach to operator precedence is only on average 5% slower than for the specification grammar.

4.2 OCaml

Compared to Java, the OCaml language specification takes a very different approach to specifying its grammar. The expression grammar is ambiguous, and operator precedence and associativity rules are specified in a table, similar to Figure 1. We used the ambiguous expression grammar of OCaml and specified operator precedence and associativity of its operators using `>`, **`left`**, and **`right`**. We have parsed 945 files from the source distribution of the OCaml compiler version 4.02. From 945 files, 894 (94%) parse successfully and without operator precedence ambiguity. Figure 7 shows the running time of parsing these files. The goodness of the fit is indicated by the R^2 value of 0.9205, and the equation of the linear regression (log-log scale) is written in the plot. As can be seen, the running time shows a near-linear behavior ($y \approx x^{1.21}$), as the coefficient value (1.21) is close to 1.

OCaml, compared to Java, is a much more difficult language to parse. First, the syntax is ambiguously specified, and in many parts of the specification, the discussion of the desired parse tree is not precise enough. More importantly, there are syntactic extensions to

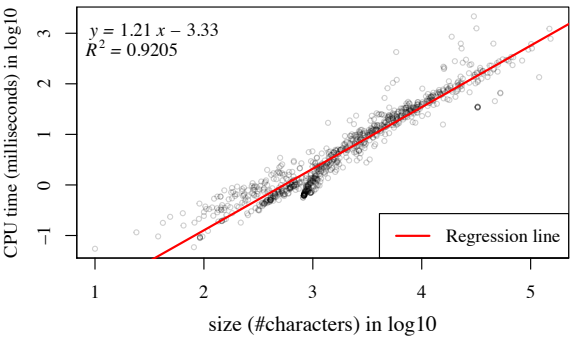


Figure 7. Running time of Iguana on the grammar of OCaml.

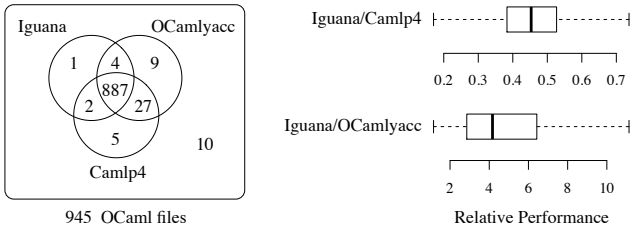


Figure 8. Distribution of the OCaml files parsed by each parser (left), and the relative performance of Iguana compared to `ocamlyacc` and `camlp4` (right).

OCaml which clash with the original syntax, and it is not clear if these extensions should only be enabled via a special flag to the compiler. To support a wider range of OCaml programs, we have incorporated some of the syntactic extensions into the grammar, and in many places we had to consult the LALR grammar of OCaml to determine how some parts should be disambiguated. For this evaluation, we also used the `ocamlyacc` grammar of OCaml, used by the OCaml compiler, and `camlp4`, an extensible syntax system for OCaml. However, even with these two parsers, we could not parse all the 945 `.ml` files. It is most likely that we are not aware of a configuration or flag while parsing those files.

Figure 8 (left) shows how many files from the source distribution of OCaml could be parsed by each parser. 10 files could not be parsed by any parser, but the majority of files, 887 (93%), could be parsed by all the parsers. Both `ocamlyacc` and `camlp4` use a separate lexing phase before parsing. Therefore, a performance comparison with a character-level grammar would not be fair. For performance comparison with `camlp4` and `ocamlyacc`, we used the context-aware [30] version of our OCaml grammar (see [3] for a discussion of context-aware scanning in Iguana).

The results of performance comparison are shown in Figure 8 (right). Each box plot shows the relative runtime of Iguana compared to the runtime of `camlp4` or `ocamlyacc`, for all 887 files that all parsers can successfully parse. As can be seen, the median running time of Iguana compared to `camlp4` is 0.45, meaning that Iguana is on average 2.2 times faster. The median running time of Iguana compared to `ocamlyacc` is 4.16, meaning that it is on average 4.16 times slower. These are promising results for a general parsing technique that uses a declarative approach to operator precedence.

4.3 Other Ambiguities in OCaml

For parsing the expression part of OCaml unambiguously, only specifying operator precedence is not enough. There are some other kinds of ambiguity in OCaml which we discuss here.

Overlapping rules Consider the simplified excerpt of OCaml, augmented with operator precedence constructs, in Figure 2. For this grammar, the input string `a-a` is ambiguous with two derivation trees that correspond to the following groupings: `a(-a)` (the function application of `a` on `-a`) and `a-a` (binary minus). Although this ambiguity looks similar to operator precedence ambiguity, it cannot be disambiguated by using **`left`**, **`right`**, and `>`. To resolve this ambiguity, we use an except construct, which disallows the derivation of a certain rule at a certain grammar position. In this case, we can write `expr ::= expr expr !umins`, where `uminus` refers to the unary minus rule. This definition effectively disallows unary minus to be derived at the right-most `expr` of a function application.

Longest match ambiguities Another ambiguity that happens in the expression part of OCaml is related to nested patterns. For example consider the following OCaml pseudo-code:

```
let f = function
        | 0 -> match ... with
               | a -> ...
               | b -> ...
```

Even with specifying all operator precedence rules, and applying them in a deep and indirect setting, this sentence remains ambiguous. The reason is that the `b`-case can belong either to function `f` or to the `match` of the `0`-case. The OCaml language specification states that a pattern matching construct extends as long as possible (longest match). We resolved this issue by adding a custom follow restriction that bypasses layout: `... pattern-matching !>>> '|'`. These constructs are explained in our previous work [3] in detail.

Dangling else ambiguity Finally, we discuss the infamous dangling else ambiguity, which occurs between rules of the form:

```
Stmt ::= 'if' Expr 'then' Stmt
       | 'if' Expr 'then' Stmt 'else' Stmt
```

As can be seen, both rules have right-recursive ends, and the first rule is a prefix of the second rule. The dangling-else ambiguity, although looks very similar to operator precedence ambiguity, does not fit our semantics of operator precedence because both rules involved in the ambiguity have only right-recursive ends. Recall that in our operator precedence semantics, `>` triggers when one of the two rules is left- and the other one is right-recursive. In fact, the dangling-else ambiguity is an instance of longest match ambiguity, for which we use a follow restriction: `Stmt ::= 'if' Expr 'then' Stmt !>>> 'else'`.

5. Related Work

Throughout this paper, we discussed the Yacc- and SDF-style operator precedence semantics, which we do not repeat here. In this section we discuss a number of related work that are directly related to our solution and inspired us the most.

5.1 Parsing OCaml

In Section 2 we motivated our new approach to operator precedence using examples of OCaml. The first question that comes to mind is how OCaml is actually parsed in practice.

OCamlyacc The parser for the OCaml compiler is written using `ocamlyacc` [22], a port of Yacc to OCaml. As we showed in Section 2, the Yacc-style resolution of operator precedence ambiguity can deal with all difficult cases in OCaml, provided that the grammar is LALR. This means that the grammar used for the OCaml compiler is not the natural, highly ambiguous specification grammar, rather it is an LALR version. Consider the following grammar rules, which are taken from the expression part of the OCaml specification grammar. The alternatives of `expr` are ordered based on precedence, with the highest precedence on top.

```
expr ::= expr '#' method-name
       | expr argument+
       | 'let' 'rec'? let-binding ('and' let-binding)*
               'in' expr
argument ::= expr | '~' label-name ':'  expr
```

The LALR counterpart of the rules above is as follows:

```
expr : simple_expr %prec below_SHARP
     | simple_expr simple_labeled_expr_list
     | let_bindings IN seq_expr;
let_bindings: let_binding | let_bindings and_let_binding;
let_binding: LET rec_flag let_binding_body;
rec_flag: REC | // empty;
simple_labeled_expr_list: labeled_simple_expr
          | simple_labeled_expr_list labeled_simple_expr;
labeled_simple_expr: simple_expr %prec below_SHARP
                   | label_expr;
label_expr: LABEL simple_expr %prec below_SHARP;
```

As can be seen, the grammar is larger and contains many other nonterminals. Part of this verbosity is due to lack of support for EBNF in Yacc, e.g. `simple_labeled_expr_list` and `rec_flag`. Some of operator precedence information is encoded declaratively, e.g., `%prec below_SHARP` specifies that the precedence of the rule `expr: simple_expr` is lower than `#`, but some others are encoded by using new nonterminals, e.g., `simple_expr`.

Moreover, the lexer used for the LALR grammar of OCaml is handwritten. This allows to hide some peculiarities in the syntax of OCaml from the LALR parser generator, e.g., how labeled arguments are parsed. OCaml also supports nested comments, which are dealt with in the lexer. It appears that the frontend for the OCaml compiler has been developed with considerable effort. The benefit of such a parser is that it is very fast. In contrast, our approach to parsing OCaml is fully declarative. The user encodes the specification grammar of OCaml using a single formalism for both lexical and context free parts, and resolves ambiguities using declarative disambiguation constructs.

Camlp4 Camlp4[4] (and Camlp5[5]), which stands for Caml Preprocessor and Pretty-Printer, is a system for writing extensible syntax for programming languages. Camlp4 is mostly used to allow syntactic extensions to OCaml programs.

Camlp4 uses a top-down recursive-descent parsing technique that interprets an in-memory representation of a grammar, and a handwritten lexer that conforms to the lexical syntax of OCaml. Camlp4 also allows the user to write a natural expression grammar, and to declaratively specify operator precedence and associativity. For example, an expression grammar for floating point arithmetic expressions, containing `'+'`, `'-'` and `'**'`, where `'**'` is right-associative and has higher precedence than left-associative `'+'` and `'-'`, can be encoded in Camlp4 as follows:

```
expr: [ "minus" LEFTA
        [ x = SELF; "+"; y = SELF -> x +. y
        | x = SELF; "-"; y = SELF -> x -. y]
      | "power" RIGHTA
        [ x = SELF; "**"; y = SELF -> x ** y]
      | "simple"
        [ x = INT -> float_of_int x ]];
```

`SELF` in this example refers to the `expr` nonterminal itself. The operator precedence scheme in Camlp4 works as follows. The parser tries alternatives in order, and each alternative that can parse at least one token is matched. The grammar is internally left-factorized to facilitate one token lookahead.

[4] `https://github.com/ocaml/camlp4`

[5] `http://camlp5.gforge.inria.fr/`

To deal with left-recursive calls, Camlp4 divides rules into two groups: `start` and `continue`. If a rule is left-recursive, i.e., starting with the nonterminal head or `SELF`, the rule is parsed using the `continue` parser, otherwise, using the `start` parser. In the example above, the `"simple"` rule belongs to the start group, while the other rules belong to the `continue` group. Parsing starts by calling the start function, and then a continue parser, based on the precedence and associativity level, is called with the value of the start function as argument. This approach to operator precedence can parse OCaml, but it is tied to the way left-recursion is implemented in a deterministic LL(1)-like parsing strategy.

5.2 General Parsers

In this section we discuss a number of operator precedence techniques that are implemented in context of general parsing.

Dypgen Dypgen[6] is a parser generator written in OCaml that allows definition of extensible grammars. Dypgen is based on GLR parsing and can handle all context-free grammars. A distinguishing feature of Dypgen is that it natively allows passing values though parsing states. Dypgen allows definition of operator precedence via precedence *relations*. For example, consider an expression grammar consisting of `'*'` and `'+'` operators, where `'*'` has higher precedence, and both operators are left associative. This grammar, along with precedence relations, can be encoded in Dypgen as:

```
expr: INT                          p1
    | expr(<=p2) + expr(<p2)       p2
    | expr(<=p3) * expr(<p3)       p3
```

The precedence relation for this grammar is defined as `p1<p2<p3`, where `pi` is the precedence of the `i`th rule. The semantics of passing values in Dypgen is as follows. When a rule is reduced, its precedence is returned. Consequently, a shift action can only happen when the precedence returned from the previous reduce action matches the condition in the body of rules, e.g., `(<=p2)`.

Dypgen does not provide high-level notation for specifying operator precedence, and the user has to manually encode operator precedence using relations and conditions in the body of rules. In addition, the semantics of passing values in Dypgen is bound to the underlying LR automaton. In contrast to Dypgen, we provide high-level notation for specifying operator precedence, and desugar them into data-dependent grammars. Jim and Mandelbaum [14] report that they could implement data-dependent grammars on top of GLR parsing, by mapping to Dypgen's native features. Therefore, Dypgen can be used as a backend to realize our approach to operator precedence in GLR parsing.

Elkhound Elkhound [23] is a fast GLR parser generator that switches to the machinery of an LR parser on deterministic parts of the grammar. For dealing with operator precedence, Elkhound essentially uses the same approach as Yacc: shift/reduce conflicts are resolved by precedence and associativity of operators. However, because Elkhound is based on GLR parsing, it does not need to resolve all shift/reduce conflicts while parsing. Conflicts that do not correspond to precedence ambiguity are left intact and are effectively explored in parallel by GLR. Moreover, Elkhound uses a separate lexing phase, which discards layout. This allows correct resolution of precedence ambiguity in conflicting states.

Is it possible to use Elkhound's way of dealing with operator precedence in a scannerless setting where layout is part of the grammar? The answer is yes, but we need to put layout nonterminals after each terminal, as in [25] or [18]. The SDF-style layout insertion, i.e., between each two symbols in body of rules, does not work with shift/reduce way of resolving precedence ambiguity. In a conflicting state, the parser needs to decide to shift based on the next operator, but this operator is hidden behind a layout nonterminal.

ANTLR ANTLR [24] is a popular recursive-descent parser generator. Starting from version 4, ANTLR supports left-recursive rules and enables global backtracking using the ALL(*) strategy [24]. ANTLR 4 supports all context-free grammars except the ones with indirect or hidden left recursion. ANTLR 4 does not natively deal with left recursion, rather it uses a left-recursion transformation under-the-hood, and then transforms the trees back to the ones of the original, natural grammar. ANTLR 4 is not a general parser in the sense that it cannot deliver all the derivation trees in case of ambiguity, rather it uses an implicit ambiguity resolution scheme, in which the ambiguities are resolved based on the order of alternatives. Note that as ANTLR 4 uses a global backtracking scheme, it does not have the quirks of PEG-style [12] backtracking.

The support for left recursion and operator precedence in ANTLR are interwoven. When ANTLR rewrites left-recursive rules, it always adds precedence and associativity information to the rewritten rules: all rules are left-associative by default, and earlier alternatives have higher precedence. For example, an expression grammar containing `'+'` and `'%'` where both operators are left-associative and `'%'` has higher precedence than `'+'` is transformed to the following grammar [24]:

```
E[pr] ::= id ({3 >= pr}? '%' E[4] | {2 >= pr}? '+' E[3])*
```

This transformation scheme is known as *precedence climbing* which mimics Clarke's technique [6], and requires a non-left-recursive grammar. Left-associative derivation trees, however, require left recursion. In ANTLR 4, a flat list, resulting from the expansion of Kleene star, is interpreted as left-associative. Moreover this technique does not allow associativity groups for operators that have the same precedence, but are left- or right-associative with respect to each other. One way to get left-associative groups in ANTLR 4 is to group operators: `E ::= E ('+'|'-') E`.

Our approach to translation of operator precedence resembles precedence climbing, in the sense that precedence level is passed, and illegal derivations are excluded using predicates. However, our approach works in present of left recursion, thus being able to natively construct parse trees that conform to the original grammar. In addition, we also support associativity groups. Finally, our approach is fully declarative, and no default precedence or associativity is applied: if the user writes a partially disambiguated grammar, by not specifying the precedence or associativity of some operators, the parser returns all the ambiguities.

Dynamic operator precedence In programming languages such as Prolog it is possible to redefine the precedence of operators at runtime. Such systems are fundamentally different from other related work we discussed, in the sense that the user does not directly work with the syntax of expressions. Prolog and similar dynamic operator precedence approaches use operator precedence grammars [11] to dynamically store precedence relationships in a table, and then interpret it. As the user does not have access to grammar of expressions in such dynamic operator precedence systems, the expressivity limitations of operator precedence grammars is not a problem. Favero [10] presents a detailed, step-by-step analysis of how dynamic operator precedence systems can be implemented.

Danielsson and Norell [7] present an approach for parsing mixfix operators for a user-defined operator precedence setting. Mixfix operators are a generalization of prefix and postfix operators. For example, **`if-then-else`** can be considered as a mixfix operator: **`if`** `[]` **`then`** `[]` **`else`** `[]` which has three places for operands. This way of specifying operators is beneficial for systems in which the precedence and associativity of operators defined globally based on their token, not the rule in which they appear. The operator prece-

[6] `http://dypgen.free.fr/`

dence and associativity information in this approach is encoded in a precedence graph. To deal with user-defined operator precedence, expressions are treated as flat lists of tokens, and then parsed again when the precedence graph is composed at runtime. The semantic of this approach is the same as in SDF2, by applying one level relationship between parents and children. This means that, for example, a unary prefix operator with lower precedence will be rejected on right of a binary operator.

Other approaches So far, we discussed operator precedence techniques that are used in parsing tools. However, there are many other approaches which have not found their way in practice, or the tools that implemented them are not available any more. Most notable approaches in this category are by Aasa [1], Thorup [28], and Visser [32]. All these approaches use a grammar rewriting technique to implement operator precedence.

Aasa [1] introduces an approach for declarative specification of operator precedence by assigning weights to operators in a parse tree. These weights define parse trees that are precedence correct. Aasa's approach is safe, and correctly identifies deep cases of operator precedence. A shortcoming of this approach is that operator precedence is defined token-based and globally. Therefore, operators in this approach have to be unique.

Thorup [28] presents a general grammar transformation technique that gets a grammar and a set of illegal sub-parse trees as input, and produces a grammar that does not yield derivation trees that are illegal. This approach can be used to implement operator precedence, if ambiguities in operator precedence are specified as tree patterns. As some precedence ambiguity patterns are arbitrary deep, it is not clear how they can be specified in this approach.

Visser [32] introduces a transformation from context-free grammars to character-class grammars, by applying the SDF2 semantics. Visser's approach is similar to the rewriting approach in [4], with the difference that instead of using indexed nonterminals and operating on the grammar, it replaces nonterminals with a set of integers and removes violating patterns. Because this approach has the underlying SDF2 semantics, it may remove sentences from the language if there is no ambiguity, and cannot deal with deep cases.

6. Conclusions

In this paper we presented a technique for implementing a declarative specification of operator precedence, by desugaring to data-dependent grammars. Our approach is efficient and can deal with intricate cases of operator precedence found in functional programming languages such as OCaml. We evaluated our approach using the Iguana parsing framework, and the results show that our approach can be practical. Other general parsing algorithms such as GLR or Earley can also be used as a backend for data-dependent grammars, and adapt our operator precedence approach.

Acknowledgments

We are thankful to Eelco Visser and the anonymous reviewers for their constructive feedback on earlier versions of this paper.

References

[1] A. Aasa. Precedences in Specifications and Implementations of Programming Languages. *Theor. Comput. Sci.*, 142(1):3–26, May 1995.

[2] A. Afroozeh and A. Izmaylova. Faster, Practical GLL Parsing. In *CC '15*, pages 89–108. Springer, 2015.

[3] A. Afroozeh and A. Izmaylova. One Parser to Rule Them All. In *Onward! 15*, pages 151–170. ACM, 2015.

[4] A. Afroozeh, M. van den Brand, A. Johnstone, E. Scott, and J. J. Vinju. Safe Specification of Operator Precedence Rules. In *SLE'13*, pages 137–156. Springer, 2013.

[5] A. V. Aho, S. C. Johnson, and J. D. Ullman. Deterministic Parsing of Ambiguous Grammars. In *POPL '73*, pages 1–21, 1973.

[6] K. Clarke. The Top-down Parsing of Expressions. Technical report, Dept. of Computer Science and Statistics, Queen Mary College, 1986.

[7] N. Danielsson and U. Norell. Parsing Mixfix Operators. In *Implementation and Application of Functional Languages*, pages 80–99. 2011.

[8] F. L. DeRemer. *Practical Translators for LR(k) Languages*. PhD thesis, Massachusetts Institute of Technology, 1969.

[9] J. Earley. An Efficient Context-free Parsing Algorithm. *Commun. ACM*, 13(2):94–102, Feb. 1970. ISSN 0001-0782.

[10] E. L. Favero. The Simple and Powerful yfx Operator Precedence Parser. *Softw. Pract. Exper.*, 37(14):1451–1474, Nov. 2007.

[11] R. W. Floyd. Syntactic Analysis and Operator Precedence. *J. ACM*, 10(3):316–333, 1963.

[12] B. Ford. Parsing Expression Grammars: A Recognition-based Syntactic Foundation. In *POPL'04*, pages 111–122, 2004.

[13] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. The Java Language Specification Java SE 7 Edition, 2013.

[14] T. Jim and Y. Mandelbaum. A New Method for Dependent Parsing. In *ESOP'11*, pages 378–397. Springer, 2011.

[15] T. Jim, Y. Mandelbaum, and D. Walker. Semantics and Algorithms for Data-dependent Grammars. In *POPL'10*, pages 417–430, 2010.

[16] S. C. Johnson. Yacc: Yet Another Compiler-Compiler. Technical report, AT&T Bell Laboratories, 1979.

[17] A. Johnstone, E. Scott, and G. Economopoulos. Generalised Parsing: Some Costs. In *CC'04*, pages 89–103, 2004.

[18] A. Johnstone, E. Scott, and M. van den Brand. Modular Grammar Specification. *Sci. Comput. Prog.*, 87:23–43, 2014.

[19] L. C. Kats, E. Visser, and G. Wachsmuth. Pure and Declarative Syntax Definition: Paradise Lost and Regained. In *OOPSLA '10*, pages 918–932. ACM, 2010.

[20] P. Klint and E. Visser. Using Filters for the Disambiguation of Context-free Grammars. In *ASMICS Workshop on Parsing Theory*, pages 1–20. Tech. Rep. 126–1994, Università di Milano, 1994.

[21] D. E. Knuth. On the Translation of Languages from Left to Right. *Information and control*, 8(6):607–639, 1965.

[22] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. The OCaml system release 4.02. Technical report, Inria, 2014.

[23] S. McPeak and G. C. Necula. Elkhound: A Fast, Practical GLR Parser Generator. In *CC'04*, pages 73–88, 2004.

[24] T. Parr, S. Harwell, and K. Fisher. Adaptive LL(*) Parsing: The Power of Dynamic Analysis. OOPSLA '14, pages 579–598. ACM, 2014.

[25] D. J. Salomon and G. V. Cormack. Scannerless NSLR(1) Parsing of Programming Languages. In *PLDI '89*, pages 170–178, 1989.

[26] E. Scott and A. Johnstone. GLL Parse-tree Generation. *Science of Computer Programming*, 78(10):1828–1844, Oct. 2013.

[27] E. Scott, A. Johnstone, and R. Economopoulos. BRNGLR: A Cubic Tomita-style GLR Parsing Algorithm. *Acta informatica*, 44(6):427–461, 2007.

[28] M. Thorup. Disambiguating Grammars by Exclusion of Sub-Parse Trees. *Acta Inf.*, 33(6):511–522, 1996.

[29] M. Tomita. *Efficient Parsing for Natural Language*. Kluwer Academic Publishers, USA, 1985. ISBN 0898382025.

[30] E. R. Van Wyk and A. C. Schwerdfeger. Context-aware Scanning for Parsing Extensible Languages. In *GPCE'07*, pages 63–72. ACM, 2007.

[31] E. Visser. Scannerless Generalized-LR Parsing. Technical report, University of Amsterdam, 1997.

[32] E. Visser. From Context-Free Grammars With Priorities to Character Class Grammars. Technical report, University of Amsterdam, 1997.

[33] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.