Multi-purpose Syntax Definition with SDF3

de Souza Amorim, Luis; Visser, Eelco

DOI
10.1007/978-3-030-58768-0_1

Publication date
2020

Document Version
Final published version

Published in
Software Engineering and Formal Methods

Citation (APA)
de Souza Amorim, L., & Visser, E. (2020). Multi-purpose Syntax Definition with SDF3. In F. de Boer, & A. Cerone (Eds.), *Software Engineering and Formal Methods: 18th International Conference, SEFM 2020, Proceedings* (Vol. 12310, pp. 1-23). (Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics); Vol. 12310). Springer. https://doi.org/10.1007/978-3-030-58768-0_1

Multi-purpose Syntax Definition with SDF3

Luís Eduardo de Souza Amorim[1] and Eelco Visser[2(✉)]

[1] Australian National University, Canberra, Australia
[2] Delft University of Technology, Delft, The Netherlands
e.visser@tudelft.nl

Abstract. SDF3 is a syntax definition formalism that extends plain context-free grammars with features such as constructor declarations, declarative disambiguation rules, character-level grammars, permissive syntax, layout constraints, formatting templates, placeholder syntax, and modular composition. These features support the multi-purpose interpretation of syntax definitions, including derivation of type schemas for abstract syntax tree representations, scannerless generalized parsing of the full class of context-free grammars, error recovery, layout-sensitive parsing, parenthesization and formatting, and syntactic completion. This paper gives a high level overview of SDF3 by means of examples and provides a guide to the literature for further details.

Keywords: Syntax definition · Programming language · Parsing

1 Introduction

A syntax definition formalism is a formal language to describe the syntax of formal languages. At the core of a syntax definition formalism is a *grammar formalism* in the tradition of Chomsky's context-free grammars [14] and the Backus-Naur Form [4]. But syntax definition is concerned with more than just phrase structure, and encompasses all aspects of the syntax of languages.

In this paper, we give an overview of the syntax definition formalism SDF3 and its tool ecosystem that supports the multi-purpose interpretation of syntax definitions. The paper does not present any new technical contributions, but it is the first paper to give a (high-level) overview of all aspects of SDF3 and serves as a guide to the literature. SDF3 is the third generation in the SDF family of syntax definition formalisms, which were developed in the context of the ASF+SDF [5], Stratego/XT [10], and Spoofax [38] language workbenches.

The first SDF [23] supported modular composition of syntax definition, a direct correspondence between concrete and abstract syntax, and parsing with the full class of context-free grammars enabled by the Generalized-LR (GLR) parsing algorithm [44,56]. Its programming environment, as part of the ASF+SDF MetaEnvironment [40], focused on live development of syntax definitions through incremental and modular scanner and parser generation [24–26] in order to provide fast turnaround times during language development.

F. de Boer and A. Cerone (Eds.): SEFM 2020, LNCS 12310, pp. 1–23, 2020.
https://doi.org/10.1007/978-3-030-58768-0_1

The second generation, SDF2 encompassed a redesign of the internals of SDF without changing the surface syntax. The front-end of the implementation consisted of a transformation pipeline from the rich surface syntax to a minimal core (kernel) language [58] that served as input for parser generation. The key change of SDF2 was its integration of lexical and context-free syntax, supported by Scannerless GLR (SGLR) parsing [60,61], enabling composition of languages with different lexical syntax [12].

SDF3 is the latest member of the family and inherits many features of its predecessors. The most recognizable change is to the syntax of productions that should make it more familiar to users of other grammar formalisms. Further, it introduces new features in order to support multi-purpose interpretations of syntax definitions. The *goals of the design of SDF3* are (1) to support the definition of the concrete and abstract syntax of formal languages (with an emphasis on programming languages), (2) to support *declarative* syntax definition so that there is no need to understand parsing algorithms in order to understand definitions [39], (3) to make syntax definitions *readable and understandable* so that they can be used as reference documentation, and (4) to support *execution* of syntax definitions as parsers, but also for other syntactic operations, i.e to support *multi-purpose* interpretation based on a single source. The focus on multi-purpose interpretation is driven by the role of SDF3 in the Spoofax language workbench [38].

In this paper, we give a high-level overview of the features of SDF3 and how they support multi-purpose syntax definition. We give explanations by means of examples, assuming some familiarity of the reader with grammars. We refer to the literature for formal definitions of the concepts that we introduce. Figure 1 presents the complete syntax definition of a tiny functional language (inspired by OCaml [42]), which we will use as running example without (necessarily) referring to it explicitly.

2 Phrase Structure

A programming language is more than a set of flat sentences. It is the structure of those sentences that matters. Users understand programs in terms of structural elements such as expressions, functions, patterns, and modules. Language designers, and the tools they build to implement a language, operate on programs through their underlying (tree) structure. The productions in a context-free grammar create the connection between the tokens that form the textual representation of programs and their phrase structure [14]. Such productions can be interpreted as parsing rules to convert a text into a tree. But SDF3 emphasizes the interpretation of productions as definitions of structure [39].

A sort (also known as non-terminal) represents a syntactic category such as expression (`Exp`), pattern match case (`Case`), or pattern (`Pat`). A production defines the structure of a language construct. For example, the production

```
Exp.Add = Exp "+" Exp
```

```
module fun
imports lex
context-free start-symbols Exp
sorts Exp Case Bnd Pat
context-free syntax
  Exp      = <(<Exp>)> {bracket}
  Exp.Int = INT
  Exp.Var = ID
  Exp.Min = [-[Exp]]
  Exp.Sub = <<Exp> - <Exp>> {left}
  Exp.Add = <<Exp> + <Exp>> {left}
  Exp.Eq  = <<Exp> == <Exp>> {left}
  Exp.Fun = [fun [ID*] -> [Exp]]
  Exp.App = <<Exp> <Exp>> {left}
  Exp.Let = <
    let <{Bnd "\n\n"}*>
     in <Exp>
  >
  Exp.IfE = <
    if <Exp> then
      <Exp>
    else
      <Exp>
  >
  Exp.IfT = <
    if <Exp> then
      <Exp>
  >
  Exp.Match = <
    match <Exp>
     with <{Case "\n"}+>
  > {longest-match}
  Bnd.Bnd   = <<ID> = <Exp>>
  Case.Case = [| [Pat] -> [Exp]]
  Pat.PVar  = ID
  Pat.PApp  = <<Pat> <Pat>> {left}
  Pat       = <(<Pat>)> {bracket}
context-free priorities
  Exp.Min > Exp.App
  > {left: Exp.Sub Exp.Add}
  > Exp.Eq > Exp.IfE > Exp.IfT
  > Exp.Match > Exp.Fun > Exp.Let,
  Exp.App <1> .> Exp.Min
template options
  ID = keyword {reject}
  keyword -/- [a-zA-Z0-9]
```

```
module lex

lexical sorts ID
lexical syntax
  ID = [a-zA-Z] [a-zA-Z0-9]*
lexical restrictions
  ID -/- [a-zA-Z0-9]

lexical sorts INT
lexical syntax
  INT = [\-]? [0-9]+
lexical restrictions
  INT -/- [0-9]
context-free restrictions
  "-" -/- [0-9]

lexical sorts AST EOF
lexical syntax
  LAYOUT = [\ \t\n\r]

  LAYOUT = Com
  Com    = "/*"
             (~[\*] | Ast | Com)*
           "*/"
  Ast    = [\*]

  LAYOUT = "//" ~[\n\r]*
                ([\n\r] | EOF)
  EOF    =
lexical restrictions
  AST -/- [\/]
  EOF -/- ~[]

context-free restrictions
  LAYOUT? -/- [\ \t\n\r]
  LAYOUT? -/- [\/].[\/]
  LAYOUT? -/- [\/].[\*]
```

```
let // length of a list
   len = fun xs ->
    match xs
     with | nil -> 0
          | cons x xs -> 1 + len xs
in len (cons 1 (cons 2 nil))
```

Fig. 1. Syntax of a small functional language in SDF3 and an example program.

defines that an addition expression is one alternative for the `Exp` sort and that it is the composition of two expressions. A production makes the connection with sentences by means of literals in productions. In the production above, the two expressions making an addition are separated by a `+` operator. Finally, a production defines a constructor name for the abstract syntax tree structure of a program (`Add` in the production above). The pairs consisting of sort and constructor names should be unique within a grammar and can be used to identify productions. (Such explicit constructor names are new in SDF3 compared to SDF2.) A set of such productions is a grammar.

The productions of a grammar generate a set of well-formed syntax trees. For example, Fig. 2 shows a well-formed tree over the example grammar. The language defined by a grammar are the sentences obtained by taking the yields of those trees, where the yield of a syntax tree is the concatenation of its leaves. Thus, the sentence corresponding to the tree in Fig. 2 is `(fun x -> x + 3) y`.

The grammars of programming languages frequently feature lists, including lists of statements in a block, lists of field declarations in a class, and lists of parameters of a function. SDF3 supports direct expression of such list sorts by means of Kleene star and plus operators on sorts. In Fig. 1 the formal parameters of a `Fun` is defined as `ID*`, a list of zero or more identifiers. Other kinds of list include `A+` (one or more `A`s), `{A sep}*` (zero or more `A`s separated by `sep`s), and `{A sep}+` (one or more `A`s separated by `sep`s). Lists with separators are convenient to model, for example, the arguments of a function as `{Exp ","}*`, i.e. a list of zero or more expressions separated by commas.

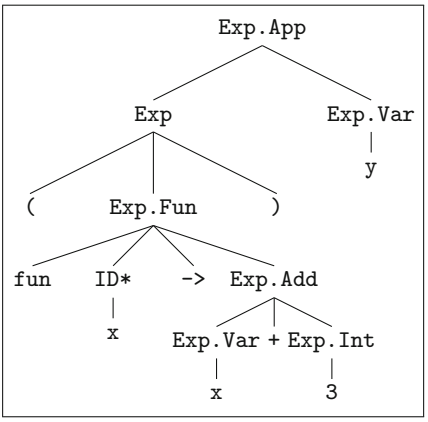

Fig. 2. Concrete syntax tree

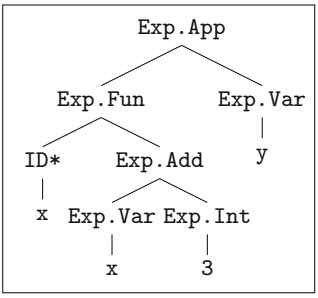

Fig. 3. Abstract syntax tree

Abstract Syntax. Concrete syntax trees contain irrelevant details such as keywords, operator symbols, and parentheses (as identified by the bracket attribute on productions). These details are irrelevant since the constructor of a production of a node uniquely identifies the language construct concerned. Thus, from

a concrete syntax tree we obtain an *abstract syntax tree* by omitting such irrelevant details. Figure 3 shows the abstract syntax tree obtained from the concrete syntax tree in Fig. 2. Abstract syntax trees can be represented by means of first-order terms in which a constructor is applied to a (possibly empty) sequence of sub-terms. For example, the abstract syntax tree of Fig. 3 is represented by the term.

```
App(Fun(["x"], Add(Var("x"), Int("3"))), Var("y"))
```

Note that lists are represented by sequences of terms between square brackets.

Signatures. A grammar is a schema for describing well-formed concrete and abstract syntax trees. That is, we can check that a tree is well-formed by checking that the subtrees of a constructor node have the right sort according to the corresponding production, and a parser based on a grammar is guaranteed to produce such well-formed trees. To further process trees after parsing, we can work on a generic tree representation such as XML or ATerms [6], or we can work with a typed representation. The schemas for such typed representations can be derived automatically from a grammar. For example, the Statix language for static semantics specification [3] uses algebraic signatures to describe well-formed terms. The following signature in Statix defines the algebraic signature of a selection of the constructors of the example language:

```
signature
sorts Exp
constructors
Fun : list(ID) * Exp -> Exp
Add : Exp * Exp -> Exp
App : Exp * Exp -> Exp
Var : ID -> Exp
Int : INT -> Exp
```

The SDF3 compiler automatically generates signatures for Statix [3], Stratego [10], and DynSem [57].

3 Declarative Disambiguation

Multiple trees over a grammar can have the same yield. Or, vice versa, a sentence in the language of a grammar can have multiple trees. If this is the case, the sentence, and hence the grammar is *ambiguous*.

One strategy to disambiguate a grammar is to transform it to an unambiguous grammar that describes the same language, but has exactly one tree per sentence in the language. However, this may not be easy to do, may distort the structure of the trees associated with the grammar, and changes the typing scheme associated with the grammar. SDF3 supports the disambiguation of an ambiguous grammar by means of declarative disambiguation rules. In this section we describe disambiguation by means of associativity and priority rules. In the next section we describe lexical disambiguation rules.

Disambiguation by Associativity and Priority Rules. Many language reference manuals define the disambiguation of expression grammars by means of priority and associativity tables. SDF3 formalizes such tables as explicit associativity and priority rules over the productions of an ambiguous context-free grammar. While grammar formalisms such as YACC also define associativity and priority rules, these are defined in terms of low-level implementation details (e.g. choosing sides in a shift/reduce conflict.) The semantics of SDF3 associativity and priority rules has a direct formal semantics that is defined independently of a particular implementation [53]. The semantics is defined by means of *subtree exclusion*, that is, disambiguation rules are interpreted by *rejecting* trees that match one of the subtree exclusion patterns generated by a set of disambiguation rules. If a set of rules is sound and complete (there is a rule for each pair of productions), then disambiguation is sound and complete, i.e. assigns a single tree to a sentence. (Read the fine print in [53].)

A priority rule `A.C1 > A.C2` defines that (the production identified by the constructor) `A.C1` has higher priority than (the production identified by the constructor) `A.C2`. This means that (a tree with root constructor) `A.C2` cannot occur as a left, respectively right recursive child of (a tree node with constructor) `A.C1` if `A.C2` is right, respectively left recursive. A left associativity rule `A.C1 left A.C2` defines that `A.C1` and `A.C2` are mutually left associative. This means that `A.C2` cannot occur as a right recursive child of `A.C1`. (Right associativity is defined symmetrically.)

Figure 1 defines the disambiguation rules for the example language. According to these rules the expression `a - b + c == d` should be parsed as `((a - b) + c) == d` (since `Sub` and `Add` are left associative and have higher priority than `Eq`) and the expression `match a with | b -> c + d` should be parsed as `match a with | b -> (c + d)` (since `Add` has higher priority than `Match`).

The semantics of priority shown above is particularly relevant for prefix and postfix operators. A prefix operator (such as `Match`) may occur as right child of an infix operator (such as `Sub`), even if it has lower priority, since such a combination of productions is not ambiguous. For example, the expression `a - match b with | c -> d` has only one abstract syntax tree.

This semantics is safe, i.e. it does not reject any sentences that are in the language of the underlying context-free grammar. However, with the rules defined so far the semantics is not complete. As an example consider two of the trees for the sentence `a - match b with | c -> d + e` in Fig. 4. Both these trees are conflict free according to the rules above; a `Match` may occur as right hand child of a `Sub` and `Sub` and `Add` are left associative. The problem is that the conflict between `Match` as a left child of `Add` is hidden by the `Sub` tree. To capture such *deep conflicts*, the priority rule involving `Add`, `Sub` and `Match` is amended to require that a right-most occurrence of a production `A.C2` in the left recursive argument of a production `A.C1` is rejected if `A.C1 > A.C2`. (And symmetrically for left-most occurrences in right recursive arguments.) Thus, the priority rules of Fig. 1 select the left tree of Fig. 4.

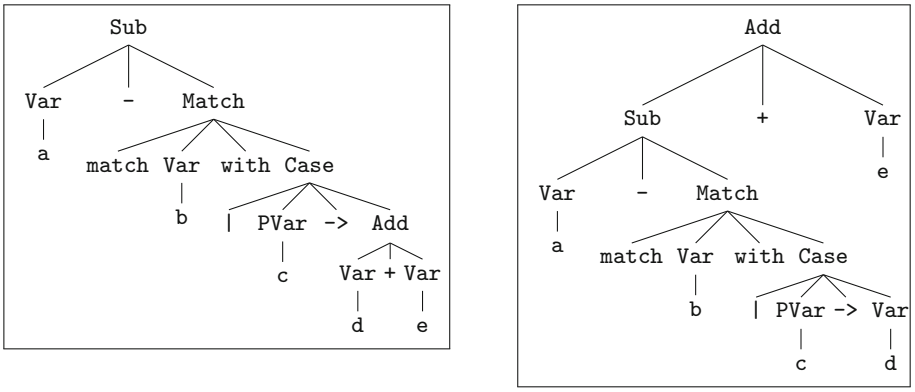

Fig. 4. Concrete syntax trees for the expression `a - match b with | c -> d + e`.

The longest-match attribute of the `Match` production is a short hand for deep priority conflicts for lists. The `Match` construct gives rise to nested pattern match clauses such as the following

```
match a with | d -> match e with | f -> g + h | i -> j + k
```

The longest match attributes disambiguates such nested lists by associating trailing cases with the nearest match statement.

Afroozeh et al. [1] showed that semantics of disambiguation in SDF2 [7,61] was not safe. They define a safe interpretation of disambiguation rules by means of a grammar transformation. Amorim and Visser [53] define a direct semantics of associativity and priority rules by means of subtree exclusion including prefix and postfix operators, mixfix productions, and indirect recursion. They show that the semantics is safe and complete for safe and complete sets of disambiguation rules for expression grammars without overlap. They also discuss the influence of overlap on disambiguation of expression grammars. For example, in Fig. 1, the productions `Min`, `Sub`, and `App` have overlap. The expression `x - y` can be parsed as `App(Var("x"), Min(Var("y")))` or as `Sub(Var("x"), Var("y"))`. This is not an ambiguity that can be solved by means of safe associativity and priority rules. The *indexed* priority rule `Exp.App <1> .> Exp.Min` solves this ambiguity by forbidding the occurrence of `Min` as second argument of `App`. (The index is 0 based.)

Amorim et al. show that deep conflicts are not only an artifact of grammars, but do actually occur in the wild, i.e. that they do occur in real programs [52]. One possible implementation of disambiguation with deep conflicts is by means of data dependent parsers. Amorim et al. show that such parsers can have near zero overhead when compared to disambiguation by grammar rewriting [55].

Parenthesization. In the previous section we saw that parentheses, i.e. productions annotated with the bracket attribute, are omitted when transforming a

concrete syntax tree to an abstract syntax tree (Fig. 3). Furthermore, by using declarative disambiguation, the typing scheme for abstract syntax trees allows arbitrary combinations of constructors in well-formed abstract syntax trees. This is convenient, since it allows transformations on trees to create new trees without regard for disambiguation rules. Before formatting such trees (Sect. 5), parentheses need to be inserted in order to prevent creating a sentence that has a different (abstract) syntax tree when parsed. That is, we want the equation $\mathsf{parse}(\mathsf{format}(t)) = t$ to hold for any well-formed abstract syntax tree.

The advantage of declarative disambiguation rules is that they can be interpreted not only to define disambiguation during parsing, but can also be interpreted to detect trees that need disambiguation. For example, without parenthesization the tree `Add(Eq(Var("a"), Var("b")), Var("c"))` would be formatted as `a == b + c`, which would be parsed as `Add(Var("a"), Eq(Var("b"), Var("c")))`. Parenthesization recognizes that the first tree has a priority conflict between `Add` and `Eq` and inserts parentheses around the equality expression, such that the tree is formatted as `(a == b) + c`, which has the original tree as its abstract syntax tree. The implementation of SDF3 in Spoofax supports parenthesization following the disambiguation semantics of Amorim and Visser [53].

4 Lexical Syntax

The lexical syntax of a language concerns the lexemes, words, or tokens of the language and typically includes identifiers, numbers, strings, keywords, operators, and delimiters. In traditional parsers and parser generators, parsing is divided into a lexical analysis (or scanning) phase in which the characters of a program are merged into tokens, and a context-free analysis phase in which a stream of tokens is parsed into phrase structure. Inspired by Salomon and Cormack [45], SDF2 adopted *character-level grammars* using the single formalism of context-free productions to define lexical and context-free syntax, supported by scannerless parsing [60]. SDF3 has inherited this feature.

Character-Level Grammars. In character-level grammars, the terminals of the grammar are individual characters. In SDF3, characters are indicated by means of character classes. For example, the definition of identifiers uses the character class `[a-zA-Z0-9]` comprising of lower and upper case letters and digits. Tokens are defined using the same productions that we use for context-free phrase structure, except that it is not required to associate a constructor with a lexical production. For example, the syntax of identifiers is defined using the production `ID = [a-zA-Z] [a-zA-Z0-9]*`, i.e. an identifier starts with a letter, which is followed by zero or more letters or digits. In a production such as `Exp.Let = "let" Bind* "in" Exp` it appears that `"let"` and `"in"` are terminals. However, SDF3 defines such literals by means of a lexical production in which the literal acts as a sort, which is defined in terms of character classes. Thus, the use of the literal `"let"` implies a production `"let" = [l] [e] [t]`. SDF3 also supports case-insensitive literals; in this case, the literal `'let'` implies a production `'let' = [lL] [eE] [tT]`.

Lexical Disambiguation. Just as phrase structure, lexical syntax may be ambiguous, requiring lexical disambiguation. The root cause of lexical ambiguity is overlap between lexical categories. For example, an identifier `ab` overlaps with the prefix of a longer identifier `abc` and `let` may be an identifier or a keyword. The two common lexical disambiguation policies are (1) prefer longest match, and (2) prefer one category over another. In scanner specification languages such as LEX [43] these policies are realized by (1) preferring the longest match and by (2) ordering the definitions of lexical rules and selecting the first rule that applies. This works well when recognizing tokens independent of the context in which they appear.

In a character-level grammar that approach does not work, since tokenization may depend on the phrase structure context (see also the discussion on language composition below), and due to modularity of a syntax definition, there is no canonical order of lexical rules. Thus, lexical disambiguation is defined analogously to subtree exclusion for phrase structure in the previous section, by defining what is not allowed using *follow restrictions* and *reject productions*. We discuss an example of each. The expression `ab` can be a single identifier or the application of `a` to `b`, i.e. `App(Var("a"),Var("b"))`. This ambiguity is solved by means of the follow restriction `ID -/- [a-zA-Z0-9]` which states that an identifier cannot be followed directly by a letter or digit. The expression `if x then y` can be an if-then expression, i.e., `IfT(Var("x"), Var("y"))`, or it can be the application of the variable `if` to some other variables, i.e.,

```
App(App(App(Var("if"), Var("x")), Var("then")), Var("y"))
```

This ambiguity is solved by means of reject productions `ID = "if" {reject}` and `ID = "else" {reject}` to forbid the use of the keywords `if` and `else` as identifiers.

Layout. Another aspect of lexical syntax is the whitespace characters and comments that can appear between tokens, which are known as 'layout' in SDF. The definition of layout is a matter of lexical definition as that of any other lexical category. Module `lex` in Fig. 1 defines layout as whitespace, multi-line comments (delimited by `/*` and `*/`), and single-line comments (starting with `//`). The multi-line comments can be nested to enable commenting out code with comments. This is not supported by scanner generators based on regular expressions. Note the use of follow restrictions to ensure that an asterisk within a multi-line comment is not followed by a slash (which should be parsed as the end of the comment), and to characterize end-of-file as the empty string that is not followed by any character (which is in turn defined as the complement of the empty character class).

What is special about layout is that it can appear between any two ordinary tokens. In a scanner-based approach layout tokens are just skipped by the scanner, leaving only tokens that matter for the parser. A character-level grammar needs to be explicit about where layout can appear. This would result in boilerplate code as illustrated by the following explicit version of the `Fun` production:

```
syntax
  Exp-CF.Var = ID-CF
  Exp-CF.Add = Exp-CF LAYOUT?-CF "+" LAYOUT?-CF Exp-CF {left}
  ID-CF      = ID-LEX
  ID-LEX     = [\65-\90\97-\122] [\48-\57\65-\90\97-\122]*-LEX
  "+"        = [\43]
  LAYOUT?-CF =
  LAYOUT?-CF = LAYOUT-CF
  LAYOUT-CF  = LAYOUT-CF LAYOUT-CF {left}
  LAYOUT-CF  = LAYOUT-LEX
  LAYOUT-LEX = [\9-\10\13\32]
restrictions
  LAYOUT?-CF -/- [\9-\10\13\32]
  ID-LEX     -/- [\48-\57\65-\90\97-\122]
```

Fig. 5. Normalized syntax and restrictions for a selection of productions from Fig. 1.

```
Exp.Fun = "fun" LAYOUT? ID* LAYOUT? "->" LAYOUT? Exp
```

To avoid such boilerplate, the SDF3 compiler applies a transformation to productions in context-free syntax sections in order to inject optional layout [61]. Figure 5 shows the result of that normalization to a small selection of productions from Fig. 1. Note that in lexical productions (such as for `ID-LEX`) no layout is injected, since the characters of tokens should not be separated by layout. Note the use of `-LEX` and `-CF` suffixes on sorts to distinguish lexical sorts from context-free sorts (This transformation is currently applied to the entire grammar, which may hinder grammar composition between modules specifying different layout.)

Layout Sensitive Syntax. In Sect. 3 we showed how associativity and priority rules can be used to disambiguate an ambiguous grammar. For example, we saw how longest match for `Match` ensures that a match case is always associated with the nearest `match`. Similarly, Fig. 1 disambiguates the dangling-else ambiguity between `IfT` and `IfE` such that an `else` branch is always associated with the closest `if`.

An alternative approach to disambiguation is to take into account the layout of a program. For that purpose, SDF3 supports the use of *layout constraints*, which pose requirements on the two dimensional shape of programs [17,54]. We illustrate layout constraints with layout-sensitive disambiguations of the `Match` and `IfE` productions in Figs. 6 and 7.

The layout constraints in Fig. 6 require that the `if` and `else` keywords of the `IfE` production are aligned. The examples in the figure show how the `else` branch can be associated with either `if` by choosing the layout. In addition, the `indent` constraints require that the conditions and branches of the `IfT` and `IfE` constructs appear to the right of the `if` and `else` keywords. Figure 7 disambiguates the association of the match cases with a `match` by requiring that the

```
Exp.IfE = "if" exp1:Exp "then" exp2:Exp "else" exp3:Exp {
  layout(align "if" "else" && indent "if" "then"
         && indent "if" exp1 && indent "if" exp2 && indent "else" exp3)
}
Exp.IfT = "if" exp1:Exp "then" exp2:Exp {
  layout(indent "if" "then" && indent "if" exp1 && indent "if" exp2)
}
```

```
if a then
  if b then
    c
  else
    d
```

```
IfT(
  Var("a")
, IfE(Var("b"), Var("c"), Var("d"))
)
```

```
if a then
  if b then
    c
else
  d
```

```
IfE(
  Var("a")
, IfT(Var("b"), Var("c"))
, Var("d")
)
```

Fig. 6. Layout-sensitive disambiguation of dangling-else.

```
Exp.Match = "match" Exp "with" cases:Case+ {
  layout(indent "match" "with" && indent "match" exp
         && align-list cases)
}
```

```
match a
 with | d -> match e
              with | f -> g
                   | i -> j
```

```
match a
 with | d -> match e
               with | f -> g
      | i -> j
```

```
Match(
  Var("a")
, [ Case(
      PVar("d")
    , Match(
        Var("e")
      , [ Case(PVar("f"),Var("g"))
        , Case(PVar("i"),Var("j"))
        ]
      )
    )
  ]
)
```

```
Match(
  Var("a")
, [ Case(
      PVar("d")
    , Match(
        Var("e")
      , [Case(PVar("f"),Var("g"))]
      )
    )
  , Case(PVar("i"),Var("j"))
  ]
)
```

Fig. 7. Layout-sensitive disambiguation of longest match for nested match cases.

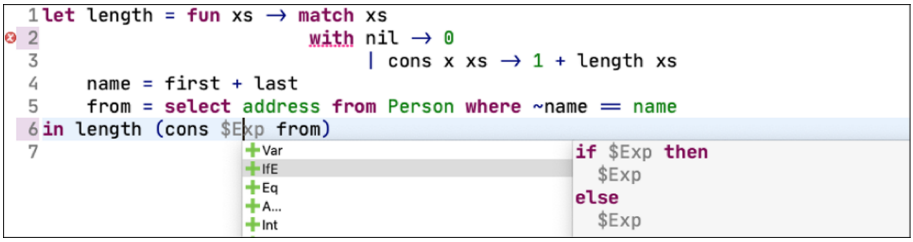

Fig. 8. Syntax-aware editor for the `fun-query` language with syntax highlighting, parse error recovery, error highlighting, and syntactic completion.

cases are aligned. Thus, one can obtain the non-longest match (second example) without using parentheses.

Syntax Highlighting. The Spoofax language workbench [38,64] generates a syntax-aware editor from a syntax definition. Based on the lexical syntax, it derives syntax highlighting for programs by assigning colors to the tokens in the syntax tree as illustrated in Fig. 8. The default coloring scheme assigns colors to lexical categories such as keywords, identifiers, numbers, and strings. The coloring scheme can be adjusted in a configuration file by associating colors with sorts and constructors.

Language Composition. SDF3 supports a simple module mechanism, allowing large syntax definitions to be divided into a collection of smaller modules, and allowing to define a library with reusable definitions. For example, the `lex` module provides a collection of common lexical syntax definitions. A module may extend the definition of syntactic categories of another module. This can be used, for example, to organize the syntax definition for a language as a collection of components (such as variables, functions, booleans, numbers) that each introduce constructs for a common set of syntactic categories (such as types and expressions).

Another application of the module mechanism is to compose the syntax definitions of different languages into a composite language. For example, Fig. 9 defines a tiny query language in module `query` and its composition with the `fun` language of Fig. 1. The composition introduces the use of a query as an expression, and a quoted expression as a query identifier. The languages have a different lexical syntax, i.e. the keywords of the `fun` language are not reserved in the `query` language, and vice versa. Thus, `from` can be used as a variable in a `fun` expression, while it is a keyword in a query (see Fig. 8). Language composition with SDF2/3 has been used for the embedding of domain-specific languages [12], for the embedding of query and scripting languages [9], and for the organization of composite languages such as AspectJ [11] and WebDSL [27,62].

A consequence of merging of productions for sorts with the same name and injecting layout between symbols of a production, is that the layout of com-

```
module query
sorts Query lexical sorts QID
lexical syntax
  QID = [a-zA-Z0-9]+
lexical restrictions
  QID -/- [a-zA-Z0-9]
context-free syntax
  Query.Select = <
    select <QID*> from <QID*> where <Cond>
  >
  Cond.Eq = <<QID> == <QID>>
template options
  QID = keyword {reject}
  keyword -/- [a-zA-Z0-9]
```

```
module fun-query
imports fun query
context-free syntax
  Exp.Query = Query
  QID.Exp = [~[Exp]]
  ID = [select] {reject}
```

Fig. 9. Composition of languages with different lexical syntax.

posed languages is unified. It is future work to preserve the layout of composed languages.

5 Formatting

Formatting is the process of mapping abstract syntax trees to text. This can be used to improve the layout of a manually written program, or it can be used to turn a generated or transformed abstract syntax tree into a program text. Formatting is preceded by parenthesization to correctly insert parentheses such that parsing the formatted text preserves the tree structure (see Sect. 3).

Template Productions. Formatting comes in two levels. The basic level of formatting, also known as ugly-printing, is concerned with inserting the 'irrelevant' notational details that were removed in the translation to abstract syntax. After ugly-printing, parsing the generated text should produce the original abstract syntax tree. This translation can be obtained from a grammar mechanically. For example, the Stratego/XT transformation tool suite featured a 'pretty-print' table generator [35] that formalized for each constructor a mapping to these notational details.

The second level of formatting, also known as pretty-printing, is concerned with producing white space to make the generated program text readable. The Box language [8,34] provides abstractions for horizontal and vertical composition and horizontal (e.g. indentation) and vertical (line breaks) spacing. This is a useful intermediate representation for formatting, which allows the pretty-printer writer to abstract from an actual pretty-print algorithm. (Libraries for pretty-printing are built on the same principle [29].) Still, a mapping from abstract syntax trees to Box expressions requires human judgement and cannot be derived mechanically from a grammar. The pretty-print table generator

mentioned above featured heuristics for associating Box expressions with language constructs. However, in many cases, it was necessary to edit the table to produce useful results, creating a bidirectional update problem to reflect changes to the grammar. SDF3 solves this problem by means of *template productions*, originally motivated to support syntactic completion (see below) [63]. (Template productions are a signature feature of SDF3, as they changed the syntax of productions from defined non-terminal on the right in SDF and SDF2, to defined non-terminal on the left, and the template quotes have a distinct influence on the typography of syntax definitions.)

A regular context-free grammar production (Sect. 2) such as

```
Exp.IfE = "if" Exp "then" Exp "else" Exp
```

combines sorts and literals. Sorts are identifiers referring to other productions and become the sub-terms of an abstract syntax tree node. Literals are quoted strings and are removed in the mapping to abstract syntax, needing to be restored during pretty-printing. Sorts and literals are implicitly separated by layout as discussed in Sect. 4.

In a template production the usual quotation is inverted. Consider the template version of the `IfE` production in Fig. 10. The outer quotes (`<if ...>`), quote a literal piece of text. The inner quotes (`<Exp>`) are escapes to sorts. A template not only captures literals and sorts, but also captures a two dimensional shape. For the purposes of parsing this shape is ignored. That is, whitespace between symbols is turned into optional layout analogous to the transformation discussed in Sect. 4. (For the purpose of layout-sensitive parsing it would be interesting to interpret the layout in a template as layout constraints, but it is not easy to distinguish which layout should be enforced, and which layout is incidental.)

```
Exp.IfE = <
  if <Exp> then
    <Exp>
  else
    <Exp>
>
```

Fig. 10. Template production

For the purpose of pretty-printing, the two dimensional shape is interpreted as horizontal and vertical composition and spacing. That is, newlines are interpreted as vertical space and spaces are interpreted as indentation (with respect to the first non-whitespace character of the template). The template in Fig. 11 shows how the spacing of list elements can be configured with whitespace in the separator.

```
Exp.Let = <
  let <{Bnd "\n\n"}*>
   in <Exp>
>
```

Fig. 11. Separator layout

Templates are translated to a transformation from abstract syntax terms to Box expressions. Thus, after every change to the grammar, the pretty-printer is automatically regenerated and up-to-date, without requiring a bidirectional update process. Plain productions with quoted literals can also be obtained automatically from template productions.

The formatters derived from SDF3 templates have some limitations, which are partly due to (the interpretation of) the Box intermediate representation.

First, formatting is fairly rigid. It does not take into account the composition and size of expressions, but formats a language construct always in the same manner. Furthermore, it is not customizable with user preferences, as is customary in integrated development environments such as Eclipse. When formatting manually written programs to improve their layout, or when formatting a program after applying some transformation (e.g. a refactoring), it can be important to preserve the layout (comments and/or whitespace) of the original program. De Jonge and Visser [32] developed a layout preserving formatting algorithm with heuristics for moving comment blocks. This algorithm is currently not integrated in the SDF3 tool suite.

Completion. Formatting is also an issue when proposing and inserting syntactic completions in an editor. The first version of Spoofax [38] featured syntactic completion templates instructing the editor what to do on particular triggers, which redundantly specified syntactic patterns. Vollebregt et al. [63] introduced template productions with the goal to automatically generate completion templates and support a program completion workflow in the style of structured editors. Amorim et al. [51] generate explicit placeholder syntax for all syntactic sorts in order to explicitly represent incomplete programs. Syntactic completion becomes a matter of generating completion proposals for placeholders based on the productions of the grammar. The resulting editor behaves like a combination of text editor and structure editor as illustrated in Fig. 8.

6 Parsing

Finally, we discuss the parsing strategy of SDF3. Character-level grammars do not fit in restricted grammar classes such as LL or LR grammars; deciding which alternative to take may require an unbounded number of characters of lookahead [61]. Furthermore, only the full class of context-free grammars is closed under composition [28], i.e. the composition of two LL or LR grammars is not necessarily an LL or LR grammar. Thus, SDF3 uses a generalized parsing algorithm that can deal with the full class of context-free grammars.

Lazy Parse Table Generation. The SDF3 compiler first transforms a modular syntax definition to a monolithic and normalized syntax definition, which makes layout and deep priority conflicts explicit in the grammar [53,61]. A static analysis checks whether all used sorts are defined and warns for missing associativity and priority rules. A parser generation algorithm is used to generate a shift/reduce parse table from the normalized grammar. The algorithm is based on SLR parse table generation [28] adapted to deal with shallow priority conflicts [59]. Follow restrictions are implemented by restricting the follow set of non-terminals in the parse table. Follow restrictions that are longer than one character are added as dynamic checks. The resulting table may contain shift/reduce conflicts.

```
a + b == c + d
```

```
amb([
  Add(Var("a"), amb([ Add(Eq(Var("b"), Var("c")), Var("d"))
                    , Eq(Var("b"), Add(Var("c"), Var("d")))]))
  , Add(amb([ Eq(Add(Var("a"), Var("b")), Var("c"))
            , Add(Var("a"), Eq(Var("b"), Var("c")))]), Var("d"))
  , Eq(Add(Var("a"), Var("b")), Add(Var("c"), Var("d")))
])
```

Fig. 12. Sentence and abstract syntax tree with (shared) ambiguities.

LR parse table generation is a non-local operation, requiring the entire grammar, implying that separate compilation is not possible. If one module of the syntax definition is changed, it needs to be recompiled entirely. This is a disadvantage for scenarios that depend on language extension [12,16]. Bravenboer and Visser developed a representation and algorithm for parse table composition that realized a form of separate compilation for syntax definitions [13]. However, the algorithm did not support cross-module priority declarations and was not adopted in practice. As a more pragmatic approach, Amorim et al. [52] adopted lazy parse table generation [26], which starts with an empty parse table, and only generates those states that are needed at parse time. This ensures fast turnaround times during development of syntax definitions.

Scannerless Generalized LR Parsing with Error Recovery. The shift/reduce parse tables generated from SDF3 definitions are not deterministic, i.e. may have shift/reduce conflicts due to proper ambiguities or unbounded lookahead. To handle both these cases, SDF3 uses a Scannerless Generalized-LR (SGLR) parsing algorithm [60].

The GLR algorithm handles conflicts in the parse table by forking off separate parsers for each alternative of a conflict [44]. If the parser has encountered a genuine ambiguity, the parallel parsers will eventually end up in the same parse state, and the branches give rise to alternative parse trees. The result of parsing is a *parse forest*, a compact representation of all possible parse trees. A language engineer using SDF3 can inspect the ambiguities of a grammar by inspecting the (abstract) syntax trees with ambiguities, instead of inspecting shift/reduce conflicts. Figure 12 shows an abstract syntax tree with ambiguities for an expression in the example language using a syntax definition without disambiguation rules.

Another reason for shift/reduce conflicts is the limited lookahead of the parser generator. For example, consider parsing the expression `a == b /* a comment */ + c`. After reading the identifier `b`, the parser can reduce to create `Eq(Var("a"),Var("b"))` or it can shift, expecting to eventually parse some subexpression of `Eq`, i.e. resulting in a term of the form `Eq(Var("a"),?(Var("b"),...))`. This decision can only be made when parsing the `+` operator. But before the parser sees that operator, it first needs to process the comment. Forking

```
module matching
language mpsd-sdf3
start symbol Exp
test match longest match [[
  match a with | b -> match c with | e -> f | g -> h
]] parse to [[
  match a with | b -> (match c with | e -> f | g -> h)
]]
```

Fig. 13. Testing longest match disambiguation of the match-with expression.

the parser allows delaying the decision. Eventually only one of the parsers will survive and produce a tree without ambiguities.

A GLR parser becomes a scannerless parser by reading characters as tokens and handling lexical disambiguation such that invalid forks are prevented or killed as early as possible [60]. Follow restrictions are handled by means of a dynamic lookahead check on reductions. Reject productions are implemented by rejecting states that are reached with a reject production. That requires postponing the reduction from *rejectable* states until it is certain no reject productions will appear.

The SGLR algorithm is extended to support parse error recovery and produce a parse tree even if a program text contains syntactic errors [30,33,36]. This is important in interactive settings such as editors in an integrated development environment in order to enable editor services such as syntax highlighting and type analysis for programs with errors, as arise during program development. Error recovery is realized by an extension of SDF3 with `recovery` productions, which are only used in case normal parsing fails. There are two main categories of recovery rules. Inspired by island grammars [31], so called *water productions* turn normal tokens into layout, which allows skipping some tokens when they cannot be parsed otherwise. Productions such as `")" = {recover}` allow the insertion of missing literals (or complete sorts). The SDF3 normalizer automatically generates a permissive grammar with recovery rules, but such rules can also be added manually. Error recovery is the basis for reporting syntax errors. Improving the localization and explanation of error messages is a topic for future work.

An extension of SGLR to support incremental parsing based on the work of Wagner et al. [65] is under development [49].

Testing. Testing SDF3 syntax definitions is supported by the Spoofax Testing (SPT) language, a domain-specific language for testing various aspects of language definitions, including parsing [37]. An SPT test quotes a language fragment and specifies a test expectation. For testing syntax, the expectations are `parse succeeds`, `parse fails`, and `parse to` a specific term structure. Figure 13 illustrates the testing of disambiguation in SPT by specifying the disambiguated expression as parse result.

7 Related Work

We have referred to previous and related work throughout this paper. The papers that we cited about particular aspects of SDF3 provide extensive discussions of related technical work, which is beyond the scope of this paper. Here we provide a couple of high-level pointers to related efforts.

The design and implementation of SDF3 is motivated by its use in the Spoofax language workbench [38,64]. Erdweg et al. [18,19] give an overview of general concerns of the design and implementation of language workbenches.

SDF3 is bootstrapped, i.e. the syntax of SDF3 has been defined in SDF3. Other significant applications of SDF3 are the NaBL2 [2] and Statix [3] languages for type system specification, the IceDust language for data modeling [20–22], and the FlowSpec language for data-flow analysis specification [50]. Many languages originally developed with SDF2 are being ported to SDF3, including the Stratego transformation language [10].

Several syntax definition languages share aims with SDF3, in particular regarding the support for language composition. The syntax definition sublanguage of the RASCAL meta-programming language [41] has a common root in SDF2. RASCAL has adopted generalized GLL parsing [48] instead of GLR parsing. The syntax definition language of the Silver [66] attribute grammar system takes a different approach to language composition. Instead of relying on scannerless generalized parsing, it relies on context-aware scanners and restrictions on grammars in order to guarantee absence of ambiguities in composed grammars [46]. Based on these restrictions it can support parse table composition for language composition [47]. The Eco editor [15] supports language composition using language boxes, where the editor keeps track of transitions between languages, avoiding the composition of grammars.

8 Conclusion

In this paper we have presented SDF3, a mature language for the definition of syntax. The design and implementation of SDF3 are based on many years of research and engineering, fed by the experience of numerous researchers, developers, and students. The multi-purpose interpretation of SDF3 specifications allows quick prototopying of language designs and enables testing these designs in a full-fledged environment with a syntax aware editor.

Acknowledgment. We would like to thank our numerous co-authors (see the References section) for their contributions to the SDF family of languages. We would like to thank Peter Mosses for comments on this paper.

References

1. Afroozeh, A., van den Brand, M., Johnstone, A., Scott, E., Vinju, J.: Safe specification of operator precedence rules. In: Erwig, M., Paige, R.F., Van Wyk, E. (eds.) SLE 2013. LNCS, vol. 8225, pp. 137–156. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-02654-1_8

2. van Antwerpen, H., Néron, P., Tolmach, A.P., Visser, E., Wachsmuth, G.: A constraint language for static semantic analysis based on scope graphs. In: Erwig, M., Rompf, T. (eds.) Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20–22, 2016, pp. 49–60. ACM (2016). https://doi.org/10.1145/2847538.2847543
3. van Antwerpen, H., Poulsen, C.B., Rouvoet, A., Visser, E.: Scopes as types. Proc. ACM Program. Lang. **2**(OOPSLA), 1–30 (2018). https://doi.org/10.1145/3276484
4. Backus, J.W.: The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference. In: IFIP Congress. pp. 125–131 (1959)
5. van den Brand, M.G.J., et al.: The ASF+SDF meta-environment: a component-based language development environment. In: Wilhelm, R. (ed.) Compiler Construction, 10th International Conference, CC 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2–6, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2027, pp. 365–370. Springer (2001). https://doi.org/10.1016/S1571-0661(04)80917-4
6. van den Brand, M.G.J., de Jong, H.A., Klint, P., Olivier, P.A.: Efficient annotated terms. Softw.: Pract. Exp. **30**(3), 259–291 (2000)
7. van den Brand, M.G.J., Scheerder, J., Vinju, J.J., Visser, E.: Disambiguation filters for scannerless generalized LR parsers. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 143–158. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45937-5_12
8. van den Brand, M.G.J., Visser, E.: Generation of formatters for context-free languages. ACM Trans. Softw. Eng. Methodol. **5**(1), 1–41 (1996). https://doi.org/10.1145/226155.226156
9. Bravenboer, M., Dolstra, E., Visser, E.: Preventing injection attacks with syntax embeddings. Sci. Comput. Program. **75**(7), 473–495 (2010). https://doi.org/10.1016/j.scico.2009.05.004
10. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. A language and toolset for program transformation. Sci. Comput. Program. **72**(12), 52–70 (2008). https://doi.org/10.1016/j.scico.2007.11.003
11. Bravenboer, M., Tanter, É., Visser, E.: Declarative, formal, and extensible syntax definition for AspectJ. In: Tarr, P.L., Cook, W.R. (eds.) Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, pp. 209–228. ACM (2006). https://doi.org/10.1145/1167473.1167491
12. Bravenboer, M., Visser, E.: Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In: Vlissides, J.M., Schmidt, D.C. (eds.) Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, pp. 365–383. ACM, Vancouver (2004). https://doi.org/10.1145/1028976.1029007
13. Bravenboer, M., Visser, E.: Parse table composition. In: Gašević, D., Lämmel, R., Van Wyk, E. (eds.) SLE 2008. LNCS, vol. 5452, pp. 74–94. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00434-6_6
14. Chomsky, N.: Three models for the description of language. IRE Trans. Inf. Theory **2**(3), 113–124 (1956). https://doi.org/10.1109/TIT.1956.1056813
15. Diekmann, L., Tratt, L.: Eco: a language composition editor. In: Combemale, B., Pearce, D.J., Barais, O., Vinju, J.J. (eds.) SLE 2014. LNCS, vol. 8706, pp. 82–101. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11245-9_5

16. Erdweg, S., Rendel, T., Kästner, C., Ostermann, K.: Sugarj: library-based syntactic language extensibility. In: Lopes, C.V., Fisher, K. (eds.) Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22–27, 2011, pp. 391–406. ACM (2011). https://doi.org/10.1145/2048066.2048099
17. Erdweg, S., Rendel, T., Kästner, C., Ostermann, K.: Layout-sensitive generalized parsing. In: Czarnecki, K., Hedin, G. (eds.) SLE 2012. LNCS, vol. 7745, pp. 244–263. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36089-3_14
18. Erdweg, S., et al.: The state of the art in language workbenches. In: Erwig, M., Paige, R.F., Van Wyk, E. (eds.) SLE 2013. LNCS, vol. 8225, pp. 197–217. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-02654-1_11
19. Erdweg, S., et al.: Evaluating and comparing language workbenches: existing results and benchmarks for the future. Comput. Lang. Syst. Struct. **44**, 24–47 (2015). https://doi.org/10.1016/j.cl.2015.08.007
20. Harkes, D., Groenewegen, D.M., Visser, E.: IceDust: incremental and eventual computation of derived values in persistent object graphs. In: Krishnamurthi, S., Lerner, B.S. (eds.) 30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18–22, 2016, Rome, Italy. LIPIcs, vol. 56. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2016). https://doi.org/10.4230/LIPIcs.ECOOP.2016.11
21. Harkes, D., Visser, E.: Unifying and generalizing relations in role-based data modeling and navigation. In: Combemale, B., Pearce, D.J., Barais, O., Vinju, J.J. (eds.) SLE 2014. LNCS, vol. 8706, pp. 241–260. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11245-9_14
22. Harkes, D., Visser, E.: IceDust 2: derived bidirectional relations and calculation strategy composition. In: Müller, P. (ed.) 31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19–23, 2017, Barcelona, Spain. LIPIcs, vol. 74. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017). https://doi.org/10.4230/LIPIcs.ECOOP.2017.14
23. Heering, J., Hendriks, P.R.H., Klint, P., Rekers, J.: The syntax definition formalism SDF - reference manual. SIGPLAN Not. **24**(11), 43–75 (1989). https://doi.org/10.1145/71605.71607
24. Heering, J., Klint, P., Rekers, J.: Incremental generation of parsers. IEEE Trans. Softw. Eng. **16**(12), 1344–1351 (1990)
25. Heering, J., Klint, P., Rekers, J.: Incremental generation of lexical scanners. ACM Trans. Program. Lang. Syst. **14**(4), 490–520 (1992). https://doi.org/10.1145/133233.133240
26. Heering, J., Klint, P., Rekers, J.: Lazy and incremental program generation. ACM Trans. Program. Lang. Syst. **16**(3), 1010–1023 (1994). https://doi.org/10.1145/177492.177750
27. Hemel, Z., Groenewegen, D.M., Kats, L.C.L., Visser, E.: Static consistency checking of web applications with WebDSL. J. Symb. Comput. **46**(2), 150–182 (2011). https://doi.org/10.1016/j.jsc.2010.08.006
28. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation, 3rd edn. Addison-Wesley, Boston (2006)
29. Hughes, J.: The design of a pretty-printing library. In: Jeuring, J., Meijer, E. (eds.) AFP 1995. LNCS, vol. 925, pp. 53–96. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-59451-5_3

30. de Jonge, M., Kats, L.C.L., Visser, E., Söderberg, E.: Natural and flexible error recovery for generated modular language environments. ACM Trans. Program. Lang. Syst. **34**(4), 15 (2012). https://doi.org/10.1145/2400676.2400678
31. de Jonge, M., Nilsson-Nyman, E., Kats, L.C.L., Visser, E.: Natural and flexible error recovery for generated parsers. In: van den Brand, M., Gašević, D., Gray, J. (eds.) SLE 2009. LNCS, vol. 5969, pp. 204–223. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12107-4_16
32. de Jonge, M., Visser, E.: An algorithm for layout preservation in refactoring transformations. In: Sloane, A., Aßmann, U. (eds.) SLE 2011. LNCS, vol. 6940, pp. 40–59. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28830-2_3
33. de Jonge, M., Visser, E.: Automated evaluation of syntax error recovery. In: Goedicke, M., Menzies, T., Saeki, M. (eds.) IEEE/ACM International Conference on Automated Software Engineering, ASE 2012, Essen, Germany, September 3–7, 2012, pp. 322–325. ACM (2012). https://doi.org/10.1145/2351676.2351736
34. de Jonge, M.: A pretty-printer for every occasion. In: The International Symposium on Constructing Software Engineering Tools (CoSET2000). University of Wollongong, Australia (2000)
35. de Jonge, M.: Pretty-printing for software reengineering. In: 18th International Conference on Software Maintenance (ICSM 2002), Maintaining Distributed Heterogeneous Systems, 3–6 October, 2002, Montreal, Quebec, Canada, pp. 550–559. IEEE Computer Society (2002)
36. Kats, L.C.L., de Jonge, M., Nilsson-Nyman, E., Visser, E.: Providing rapid feedback in generated modular language environments: adding error recovery to scannerless generalized-LR parsing. In: Arora, S., Leavens, G.T. (eds.) Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, pp. 445–464. ACM (2009). https://doi.org/10.1145/1640089.1640122
37. Kats, L.C.L., Vermaas, R., Visser, E.: Integrated language definition testing: enabling test-driven language development. In: Lopes, C.V., Fisher, K. (eds.) Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, Part of SPLASH 2011, Portland, OR, USA, October 22–27, 2011, pp. 139–154. ACM (2011). https://doi.org/10.1145/2048066.2048080
38. Kats, L.C.L., Visser, E.: The Spoofax language workbench: rules for declarative specification of languages and IDEs. In: Cook, W.R., Clarke, S., Rinard, M.C. (eds.) Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, pp. 444–463. ACM, Reno/Tahoe (2010). https://doi.org/10.1145/1869459.1869497
39. Kats, L.C.L., Visser, E., Wachsmuth, G.: Pure and declarative syntax definition: paradise lost and regained. In: Cook, W.R., Clarke, S., Rinard, M.C. (eds.) Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, pp. 918–932. ACM, Reno/Tahoe (2010). https://doi.org/10.1145/1869459.1869535
40. Klint, P.: A meta-environment for generating programming environments. ACM Trans. Softw. Eng. Methodol. **2**(2), 176–201 (1993). https://doi.org/10.1145/151257.151260
41. Klint, P., van der Storm, T., Vinju, J.J.: RASCAL: a domain specific language for source code analysis and manipulation. In: Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, Edmonton, Alberta, Canada, September 20–21, 2009, pp. 168–177. IEEE Computer Society (2009). https://doi.org/10.1109/SCAM.2009.28

42. Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., Vouillon, J.: The OCaml system release 4.10 (2020)
43. Lesk, M.E.: Lex–a lexical analyzer generator. Tech. rep. CS-39. AT&T Bell Laboratories, Murray Hill, N.J. (1975)
44. Rekers, J.: Parser generation for interactive environments. Ph.D. thesis, University of Amsterdam, Amsterdam, The Netherlands (January 1992)
45. Salomon, D.J., Cormack, G.V.: Scannerless NSLR(1) parsing of programming languages. In: PLDI, pp. 170–178 (1989)
46. Schwerdfeger, A., Wyk, E.V.: Verifiable composition of deterministic grammars. In: Hind, M., Diwan, A. (eds.) Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15–21, 2009, pp. 199–210. ACM (2009). https://doi.org/10.1145/1542476.1542499
47. Schwerdfeger, A., Van Wyk, E.: Verifiable parse table composition for deterministic parsing. In: van den Brand, M., Gašević, D., Gray, J. (eds.) SLE 2009. LNCS, vol. 5969, pp. 184–203. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12107-4_15
48. Scott, E., Johnstone, A.: GLL parsing. Electron. Notes Theor. Comput. Sci. **253**(7), 177–189 (2010). https://doi.org/10.1016/j.entcs.2010.08.041
49. Sijm, M.P.: Incremental scannerless generalized LR parsing. In: Smaragdakis, Y. (ed.) Proceedings Companion of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, SPLASH 2019, Athens, Greece, October 20–25, 2019, pp. 54–56. ACM (2019). https://doi.org/10.1145/3359061.3361085
50. Smits, J., Visser, E.: FlowSpec: declarative dataflow analysis specification. In: Combemale, B., Mernik, M., Rumpe, B. (eds.) Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, Vancouver, BC, Canada, October 23–24, 2017, pp. 221–231. ACM (2017). https://doi.org/10.1145/3136014.3136029
51. de Souza Amorim, L.E., Erdweg, S., Wachsmuth, G., Visser, E.: Principled syntactic code completion using placeholders. In: van der Storm, T., Balland, E., Varró, D. (eds.) Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31–November 1, 2016, pp. 163–175. ACM (2016). https://doi.org/10.1145/2997364.2997374
52. de Souza Amorim, L.E., Steindorfer, M.J., Visser, E.: Deep priority conflicts in the wild: a pilot study. In: Combemale, B., Mernik, M., Rumpe, B. (eds.) Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, Vancouver, BC, Canada, October 23–24, 2017, pp. 55–66. ACM (2017). https://doi.org/10.1145/3136014.3136020
53. de Souza Amorim, L.E., Visser, E.: A direct semantics of declarative disambiguation rules. In: ACM TOPLAS (2020). under revision
54. de Souza Amorim, L.E., Steindorfer, M.J., Erdweg, S., Visser, E.: Declarative specification of indentation rules: a tooling perspective on parsing and pretty-printing layout-sensitive languages. In: 0005, D.P., Mayerhofer, T., Steimann, F. (eds.) Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2018, Boston, MA, USA, November 05–06, 2018, pp. 3–15. ACM (2018). https://doi.org/10.1145/3276604.3276607
55. de Souza Amorim, L.E., Steindorfer, M.J., Visser, E.: Towards zero-overhead disambiguation of deep priority conflicts. Programming Journal **2**(3), 13 (2018). https://doi.org/10.22152/programming-journal.org/2018/2/13

56. Tomita, M.: An efficient context-free parsing algorithm for natural languages. In: IJCAI, pp. 756–764 (1985)
57. Vergu, V.A., Néron, P., Visser, E.: DynSem: a DSL for dynamic semantics specification. In: Fernández, M. (ed.) 26th International Conference on Rewriting Techniques and Applications, RTA 2015, June 29 to July 1, 2015, Warsaw, Poland. LIPIcs, vol. 36, pp. 365–378. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2015). https://doi.org/10.4230/LIPIcs.RTA.2015.365
58. Visser, E.: A family of syntax definition formalisms. In: van den Brand, M.G.J., Estrela, V.V. (eds.) ASF+SDF 1995. A Workshop on Generating Tools from Algebraic Specifications. Technical report P9504, Programming Research Group, University of Amsterdam (May 1995)
59. Visser, E.: A case study in optimizing parsing schemata by disambiguation filters. In: International Workshop on Parsing Technology (IWPT 1997), pp. 210–224. Massachusetts Institute of Technology, Boston (September 1997)
60. Visser, E.: Scannerless generalized-LR parsing. Tech. rep. P9707, Programming Research Group, University of Amsterdam (July 1997)
61. Visser, E.: Syntax definition for language prototyping. Ph.D. thesis, University of Amsterdam (September 1997)
62. Visser, E.: WebDSL: a case study in domain-specific language engineering. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2007. LNCS, vol. 5235, pp. 291–373. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88643-3_7
63. Vollebregt, T., Kats, L.C.L., Visser, E.: Declarative specification of template-based textual editors. In: Sloane, A., Andova, S. (eds.) International Workshop on Language Descriptions, Tools, and Applications, LDTA 2012, Tallinn, Estonia, March 31–April 1, 2012, pp. 1–7. ACM (2012). https://doi.org/10.1145/2427048.2427056
64. Wachsmuth, G., Konat, G., Visser, E.: Language design with the Spoofax language workbench. IEEE Softw. **31**(5), 35–43 (2014). https://doi.org/10.1109/MS.2014.100
65. Wagner, T.A., Graham, S.L.: Efficient and flexible incremental parsing. ACM Trans. Program. Lang. Syst. **20**(5), 980–1013 (1998). https://doi.org/10.1145/293677.293678
66. Wyk, E.V., Bodin, D., Gao, J., Krishnan, L.: Silver: an extensible attribute grammar system. Sci. Comput. Program. **75**(1–2), 39–54 (2010). https://doi.org/10.1016/j.scico.2009.07.004