SugarScala: Syntactic Extensibility for Scala

SugarScala: Syntaktische Erweiterbarkeit für Scala
Master-Thesis von Florian Jakob
Februar 2014

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Fachgebiet Softwaretechnik

SugarScala: Syntactic Extensibility for Scala
SugarScala: Syntaktische Erweiterbarkeit für Scala

Vorgelegte Master-Thesis von Florian Jakob

1. Gutachten: Prof. Dr.-Ing. Mira Mezini
2. Gutachten: Dr. rer. nat. Sebastian Erdweg

Tag der Einreichung:

Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 14. Februar 2014

(Florian Jakob)

Abstract

Scala is a general-purpose programming language combining functional and object-oriented paradigms with type inference in a rich syntax and good support for DSL (domain-specific language) embedding. Scala's support for XML (Extensible Markup Language) however is outstanding, as it is build into the compiler and allows to directly use XML syntax in regular Scala source code. The embeddings of other DSLs in contrast are bound to existing Scala syntax, unless they are implemented as a modification of the Scala compiler. We argue that Scala should support user-defined syntactic extensions in form of desugarings to allow programmers the embedding of other DSLs with their respective direct syntaxes. For this purpose we present SugarScala, an instantiation of the Sugar* framework, which enables syntactic extensibility for Scala. The basis for SugarScala is an extensible context-free grammar for Scala in SDF2 (Syntax Definition Formalism). This grammar is also part of our work and we evaluate it independently from SugarScala with the testing capabilities of the Spoofax Language Workbench and a self-developed batch testing utility. We show the suitability of SugarScala with case studies for XML and EScala. The last part of this work investigates the possibility to directly integrate desugarings into the Scala compiler and yields a promising approach.

Inhaltsangabe

Scala ist eine Allzweck-Programmiersprache, die funktionale- und objektorientierte Paradigmen in einer reichhaltigen Syntax mit Typinferenz vereint und sich gut zur Einbettung von DSLs (domänenspezifischen Sprachen) eignet. Die Unterstützung für XML (Extensible Markup Language) stich dabei jedoch besonders hervor, da diese in den Scala-Compiler integriert ist und die direkte Verwendung von XML-Syntax in regulärem Scala-Quelltext ermöglicht. Im Gegensatz dazu sind die Einbettungen anderer DSLs an die existierende Scala-Syntax gebunden, außer diese sind durch eine Modifikation des Scala-Compilers implementiert. Wir behaupten, dass Scala benutzerdefinierte syntaktische Erweiterungen in Form von Desugarings unterstützen sollte, um damit die Einbettung anderer DSLs mit deren direkten Syntaxen zu ermöglichen. Dazu präsentieren wir SugarScala, eine Instantiierung des Sugar*-Frameworks, welche syntaktische Erweiterbarkeit für Scala ermöglicht. Die Grundlage für SugarScala ist eine erweiterbare kontextfreie Grammatik für Scala, formuliert in SDF2 (Syntax Definition Formalism). Diese Grammatik ist auch ein Teil unserer Arbeit und wird von uns unabhängig von SugarScala mit den Testmöglichkeiten der Spoofax Language Workbench und einem selbstentwickelten Batch-Testwerkzeug evaluiert. Wir zeigen die Eignung von SugarScala anhand von Fallbeispielen für XML und EScala. Der letzte Teil dieser Arbeit untersucht die Möglichkeit Desugarings direkt in den Scala-Compiler zu integrieren und liefert dabei einen aussichtsreichen Ansatz.

Contents

1 Introduction

Scala [25] is a rather young programming language, influenced by object-oriented, as well as functional programming paradigms. It runs on the Java Virtual Machine (JVM) and is bytecode-compatible to Java, which allows seamless interoperability between Scala and Java. Scala is an acronym for “Scalable Language” — according to the designer of Scala, Martin Odersky, ”this means that Scala grows with you“[1].

One example for such growth may be Scala’s direct support for eXtensible Markup Language (XML) [5] literals in the code. It is possible to enter XML-trees directly into the code, intertwined with Scala expressions, as one would expect from a template engine of choice. This support is implemented directly into the Scala compiler, which maintains an own XML-mode and rules for the scanning of XML expressions [24]. The Scala compiler translates the parsed XML expressions into instantiations of regular Scala objects in the Scala Standard XML library. This way XML literals can be used in pattern matching and procedurally manipulated like any other Scala object.

After parsing the XML expressions are desugared into objects of a provided XML library, which allows one to use them as regular Scala objects, including pattern matching support and procedural manipulation of the resulting XML tree.

This effort to support XML is probably motivated by the affinity of the Java ecosystem for XML. Much in the Java world is configured or persisted in XML. One prominent example being Ant [22], a Java build tool, which uses platform independent XML files to describe the build process of software artifacts. Another being Maven[2], which further allows to describe build life cycles and dependency management.

As nice as XML might be in respect to extensibility and standard-conforming tool support, it is also criticized to be verbose and not-well suited to be read and especially written by humans. So there exist alternatives for configuration files and persistence, ranging from simple key-value-paired formats like INI-files[3] or *.properties files, to more elaborate tree-based forms as JSON [9] or YAML [1] which are favoured in rapid web development frameworks as *Ruby on Rails*[4] or the Python equivalent *Django*[5]. It would be nice if Scala had the same syntactical support for these formats as for XML, but it is unlikely that each of this formats gets its own lexical mode in the Scala compiler.

At this point the question arises: why should XML be supported on compiler level if these other DSLs probably never will? In the end the support for XML is only syntactic sugar for the underlying library. Dedicating a whole mode for the parser, just to have support for syntactic XML, seems excessive. Instead of a special XML mode, there should be means to generally define syntactic extensions for the language. XML support could then be implemented based on these

[1] http://scala-lang.org/what-is-scala.html
[2] `http://maven.apache.org/`
[3] `http://en.wikipedia.org/wiki/INI_file`
[4] `http://rubyonrails.org`
[5] `https://www.djangoproject.com/`

means, as well as support for any of the other mentioned configuration and serialization formats. Support for syntactic extensions would allow to better separate the definition of core Scala syntax and the parsing of possible DSLs. It would additionally follow Scala's goal to grow with its users.

The term *syntactic extensibility* in this work refers to the ability to define syntactic extensions for an already existing syntax. This syntax is also referred to as *base syntax*. A *syntactic extension* specifies three things: (1) arbitrary new context-free syntax, (2) where this new syntax may be used in the base syntax and (3) how the new syntax can be expressed with help of the base syntax. New syntax which can be expressed with the base syntax but is easier to read and comprehend is called *syntactic sugar*. The process of translating the new syntax to the base syntax is called *desugaring*.

It is the purpose of this work to investigate the possibilities of syntactic extensibility for Scala. For this purpose we provide a base syntax for Scala in form of an extensible context-free grammar in SDF2[6]. We test this grammar for conformity with Scala v2.10.3, excluding support for XML and Unicode, with two different methods. The first method tests the parse results of 370 selected syntax fragments against certain expectations. The second method tests the successful and unambiguous parsability of the 1531 Scala source files in the `src` directory of the Scala v2.10.3 repository with a self-developed batch testing utility[7].

We further developed SugarScala[8] as instantiation of the Sugar* framework using our grammar. Two case studies for XML[9] and EScala[10] show how syntactic extensibility for Scala can be achieved with the help of SugarScala. The last part of this work investigates the feasibility to integrate syntactic extensibility into the Scala compiler[11] and shows a promising approach.

The structure of the remaining document is as follows: Chapter 2 discusses the definition of an extensible base syntax for Scala in form of a context-free grammar definition in SDF. Chapter 3 presents SugarScala as instantiation of the Sugar* framework and shows how it can be used to practically define syntactic extensions for Scala to support XML and EScala. Chapter 4 investigates how syntactic extensibility could be included in the Scala compiler and delivers a promising approach. Chapter 5 compares this work to related work and identifies it as a useful complement rather than a competitor. Chapter 6 concludes the success of this work and provides a look-out to future work.

6 `https://github.com/fjak/scala-grammar`
7 `https://github.com/fjak/spoofax-batchtest`
8 `https://github.com/fjak/lang-scala`
9 `https://github.com/fjak/xml-casestudy`
10 `https://github.com/fjak/escala-casestudy`
11 `https://github.com/fjak/sugsc`

2 Scala Grammar in SDF

To be able to implement syntactic extensions, one first needs means to describe these syntactic extensions. But furthermore one needs a base syntax which can actually be extended. Preferably the language to describe the base syntax and the language to describe the syntactic extensions should be the same.

A common choice to describe syntax is in the form of a context-free grammar. The Backus-Naur Form is a well-known notation to express context-free grammars and is commonly used in an extended form with better expressiveness, which is called Extended Backus-Naur Form (EBNF). A context-free grammar is however not sufficant to describe one definit parser for a syntax, yet alone because it does not define the output of the parser.

The tools used to derive a parser for a grammar are called parser generators. They usually use a formalism akin to EBNF, but take extra input to further specify the behaviour of the resulting parser. Depending on the kind of parser to generate the context-free grammar might also need to fulfill additional constraints. Parser generators for LL(k) or LL(*) parsers for example can usually not handle left recursive grammars.

With regard to the future definitions of the syntactic extensions we decided to use SDF as our tool of choice to specify the Scala base syntax. We provide an introduction to SDF and discuss its advantages in the next Section (2.1). This introduction also helps to better understand the encountered problems and the proposed solutions in Section 2.2. The resulting grammar is evaluated in Section 2.3 and fully provided in Appendix A.

2.1 Introduction to SDF

The Syntax Definition Formalism (SDF) [17] is a parser generator for context-free grammars. It has its focus on maintainability and extensibility of the grammars, which motivates some of its concepts and is a distinction to well-known parser generators like *ANTLR*, *yacc* or *bison*. SDF generates a scannerless generalized left-to-right bottom-up parser (SGLR) [33]. This kind of parser can handle all context-free grammars, even those including left recursive productions. This property allows authors to focus on providing a concise grammar, rather than tuning it towards compatibility for the generated parser. As the parser is scannerless, SDF grammars must also specify the lexical syntax. This requirement together with the generalization may lead to ambiguous parse results – parse forests – which need disambiguation to be reduced to unambiguous parse trees [32]. The disambiguation concepts used by SDF are production priorities, productions rejects, preference attributes and follow restrictions (look-aheads). Additionally the grammars can be split into modules, which, together with renamings, makes them composable.

The initial proposal and design for SDF dates back to the year 1986. It was redesigned in 1997 to SDF2 [34] to unify lexical and context-free syntax, amongst other things. At the time of this writing SDF is in transition to SDF3. The version used in this work is tailored towards SDF2 and deployed as part of the Spoofax Language Workbench [20]. Spoofax is based on Eclipse

```
module Contacts

imports
  ContactsLexical [CONTACT-ID => ID]

exports

context-free start-symbols
  Contact

context-free syntax
  "contact" ID "{" {Data ","}* "}"     -> Contact                 {cons("Contact")}
  "name" ID+                           -> Name                       {cons("Name")}
  Modifier? "alias" ID+                -> Alias                     {cons("Alias")}
  "private"                            -> Modifier                {cons("Private")}
  Name                                 -> Data
  Alias                                -> Data
```

Listing 2.1: Contacts.sdf – Definition of the *Contacts* module

and bundles SDF with a SGLR parser implemented in Java (JSGLR). The result of the language definition in SDF is not directly an executable parser, but a parse table, which can be used by JSGLR to create an abstract syntax tree (AST) in the ATerm format.

2.1.1 Contacts Language

We introduce SDF with a simple artificial language *Contacts*. The language is defined in two modules *Contacts* and *ContactsLexical*. The modules are defined in their own files, `Contacts.sdf` and `ContactsLexical.sdf`, respectively.

Modules in SDF are defined with the keyword **module** followed by the name of the module. All following statements until another **module** or the end of the file are then part of the module. A module can import sorts from other modules and can export start-symbols, lexical and context-free syntax. The **imports** keyword followed by module names denotes that all sorts of the named modules are imported into the current module. The imported sorts from a module can be renamed by providing a mapping in brackets behind the module name. All statements after the **exports** keyword are exported from the module and may be imported in other modules. Sorts listed after **context-free start symbols** are possible start symbols for the defined grammar.

Lexical and context-free syntax is defined with the help of productions after the keywords **lexical syntax** and **context-free syntax**, respectively. The general form of a production is:

```
metasyntax -> sort {attributes}
```

The *metasyntax* is a composition of SDF *symbols*. The *sort* is the name of the production and semantically comparable to a non-terminal in terms of grammar definitions. The *attributes* are a set of values or keywords to influence the derivation of the abstract syntax tree.

The *Contacts* module is defined in Listing 2.1. It has the name *Contacts* and imports all sorts from the *ContactsLexical* module. The import maps the sort *CONTACT-ID* from the *ContactsLexical* module to the local name *ID*. The module exports the context-free start symbol `Contact` and six

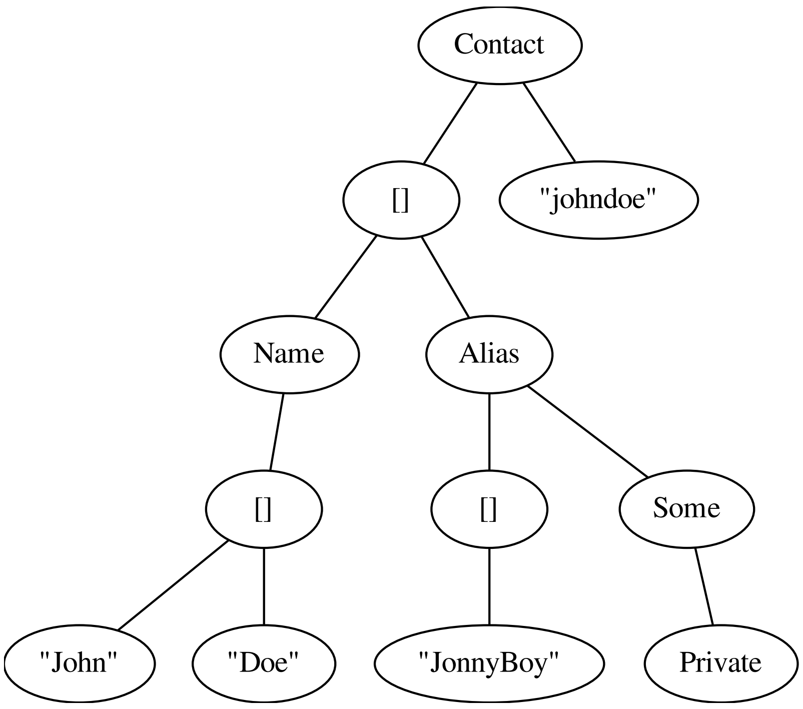


Figure 2.1.: Parse tree for the *johndoe* contact input

context-free syntax productions for the five different sorts `Contact`, `Alias`, `Name`, `Modifier` and `Data`.

The metasyntax in the production for the `Contact` sort is composed of five symbols: `"contact"`, `ID`, `"{"`, `{Data ","}*` and `"}"`. Symbols in quotation marks like `"contact"` are called literals. They require each character inside the quotations marks to appear in exactly that order in the input. Literals only consisting of letters are usually keywords and are automatically marked to be highlighted by Spoofax. Simple identifiers like `ID` reference sorts and their respective productions. `{Data ","}*` is a special regular expression symbol. It is equivalent to the repetition of the `Data` sort zero or more times separated by the `","` literal. Using a + instead of the * would require the `Data` sort to appear at least once. Writing the symbols one after another is an implicit sequence and requires all symbols to appear in this order for the production to match with the input.

The characters *, + and ? have the same meaning as in regular expressions and are quantifiers for the previous symbol. The `Modifier?` symbol denotes an optional appearance of the `Modifier` sort. Similarly does `ID+` denote the appearance of the `ID` sort at least once. Multiple productions for the same sort are alternatives for this sort, as it is the case with the `Data` sort.

The influence of the **cons** attribute for the derivation of a abstract syntax tree is best described with the help of a sample input and the resulting abstract syntax tree. The following input will result in the AST shown in Figure 2.1:

```
contact␣johndoe␣{
␣␣name␣John␣Doe,
␣␣private␣alias␣JonnyBoy
}
```

The ATerm notation for that abstract syntax tree is the following:

```
Contact("johndoe", [Name(["John", "Doe"]), Alias(Some(Private()), ["JonnyBoy"])])
```

```
module ContactsLexical

exports

lexical syntax
  "alias"         -> CONTACT-KEYWORD
  "contact"       -> CONTACT-KEYWORD
  "name"          -> CONTACT-KEYWORD
  "private"       -> CONTACT-KEYWORD
  [a-zA-Z]+       -> CONTACT-ID
  CONTACT-KEYWORD -> CONTACT-ID {reject}
  [\ \t\n\r]      -> LAYOUT

lexical restrictions
  CONTACT-ID -/- [a-zA-Z]

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

Listing 2.2: ContactsLexical.sdf – Definition of the module *ContactsLexical*

The **cons** attribute specifies a *constructor* for a production. If the parser uses this production then the constructor is used to create a new node in the abstract syntax tree. This node is also called *constructor application* and often just abbreviated as *application*. The arity of a constructor is the number of non-literal symbols in the production. It is always equal to the number of children in the application. Used optional symbols are wrapped in the *Some/1* constructor; unused optional symbols are represented with the help of the *None/0* constructor. The notation *$cons/$n* denotes the constructor with the name *$cons* and the arity of *$n*. Applications have the ATerm format `$cons($c1, $c2, ...)`. They are always written with the parentheses even if they have an arity of zero, as can be seen with the *Private/0* application. Both productions for the `Data` sort have no constructor attribute and thus do not introduce any new nodes in the abstract syntax tree even if used by the parser.

Literals from the productions do not appear in the abstract syntax tree. They can be derived from the application as long as the constructor and its arity are unique for one production. String literals in the abstract syntax tree represent parse results of lexical productions. The ATerm notation further represents lists as comma-separated subnodes in between brackets. The use of * and + symbols in a production will always result in a list node in the AST, even if the list is empty.

The *ContactsLexical* module is defined in Listing 2.2. As the name suggests it defines the lexical syntax for the *Contacts* language. The first production for the `CONTACT-ID` sort uses a character group symbol, denoted by brackets. Character group symbols resemble the same functionality as bracket expressions in regular expressions: They define a set of characters of which one may appear at that position. Together with the + quantifier the character group symbol in the mentioned production defines a `CONTACT-ID` to be one or more letters.

The second production for `CONTACT-ID` uses the **reject** attribute. Productions with the **reject** attribute are adequately called *reject productions*. They have the contrary semantics to regular productions. Whereas a regular production increases the possible derivations for a sort, the reject production decreases the possible derivations for a sort. The input `alias` is a valid derivation for CONTACT-ID according to the first production. The reject production however rejects all derivations of CONTACT-KEYWORD in CONTACT-ID position. This way "alias", "name" and "private" may not be used as CONTACT-ID. Rejecting keywords as identifiers is a common pattern in formal language design to avoid confusion for the user of the language and make regular expression based syntax highlighting easier.

The lexical **LAYOUT** sort has a special role in SDF. It specifies derivations which are only used to layout the code and help a human reader to better understand the code. Commonly these derivations are comments or characters used for indentation and newlines. Derivations from **LAYOUT** will not appear in the abstract syntax tree. The **LAYOUT** sort is transparently inserted with a ? quantifier between other symbols in context-free syntax productions, but not in lexical syntax productions. This is actually the main distinction between context-free and lexical syntax. Without **LAYOUT** the johndoe example would need to written as the following, as no whitespace were allowed:

```
contactjohndoe{nameJohnDoe,privatealiasJonnyBoy}
```

Another important concept of SDF are follow restrictions. They can be defined for lexical or context-free *symbols* after the **lexical restrictions** or **context-free restrictions** keywords, respectively. The general form of a restriction is `$lhs -/- $rhs`, where *$lhs* is a list of symbols and *$rhs* is one or more character groups separated by dots. A restriction does not allow a derivation of any of the symbols on the left hand side to be followed by the characters from the right hand side. As of this property restrictions can be used to enforce a longest-match policy on lexical productions.

The lexical restriction on CONTACT-ID does not allow a CONTACT-ID to be followed by letters. This is necessary to ensure an unambiguous parse result for a list of CONTACT-ID as used in the productions for Name and Alias. Without the restriction the input `jon` parsed by the symbol CONTACT-ID+ would have the ambiguous results `["jon"]`, `["jo", "n"]`, `["j", "on"]` and `["j", "o", "n"]` as it would not be clear where one identifier ends and where the next one starts. With the restriction however only the result `["jon"]` is valid, as all others are followed by letters.

2.1.2 Ambiguity and Disambiguation

Another important aspect of SDF is the disambiguation of the provided grammars and the resulting abstract syntax trees. Consider the following simple grammar for arithmetic infix expressions:

```
int  = ("1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"), {? any digit ?}
op   = "+" | "-" | "*" | "/"
expr = int | expr, op, expr
```

A naive translation to SDF, which lacks any form of disambiguation, is given in Listing 2.3. The approach is straight-forward: The lexical syntax only consists of integers, here named INT, which must start with a digit from one to nine and then may be followed by an arbitrary number of digits. The only other lexical production is LAYOUT to allow whitespace in the context-free syntax. The start symbol of the grammar is Expr. An Expr is inductively definied to be either an INT, or of the form `Expr OP Expr`, where OP is replaced by the respective operator and the constructor for the node is named accordingly.

```
module Arithmetics

exports

lexical syntax
  [\ \t\n\r] -> LAYOUT
  [1-9] [0-9]* -> INT

context-free start-symbols
  Expr

context-free syntax
  INT -> Expr
  Expr "+" Expr -> Expr {cons("Plus")}
  Expr "-" Expr -> Expr {cons("Minus")}
  Expr "*" Expr -> Expr {cons("Mult")}
  Expr "/" Expr -> Expr {cons("Div")}
```

Listing 2.3: Ambiguous arithmetic expressions

left, right: The provided grammar would be able to parse the expressions `42`, `23+42`, `13 * 23` and `1␣/␣2` unambiguously. But the expression `1 + 2 + 3` introduces an ambiguity. Both parse results `Plus(1, Plus(2, 3))` and `Plus(Plus(1, 2), 3)` are valid according to the grammar. This problem is well-known as the *associativity* of operators. The first parse result is called *right-associative* because the production is recursively used on the right-hand sort. The other is analogously called *left-associative*. Figure 2.2 visualizes the two parse results as trees to better illustrate the issue.

SDF has direct support to express the associativity of operator productions to avoid these kind of ambiguities. For this purpose the attributes **left** or **right** can be added to the affected productions which declares the production as left- or right-associative, respectively. So the production

```
Expr "+" Expr -> Expr {cons("Plus"), left}
```

is the corrected and unambiguous left-associative version for the plus operator.

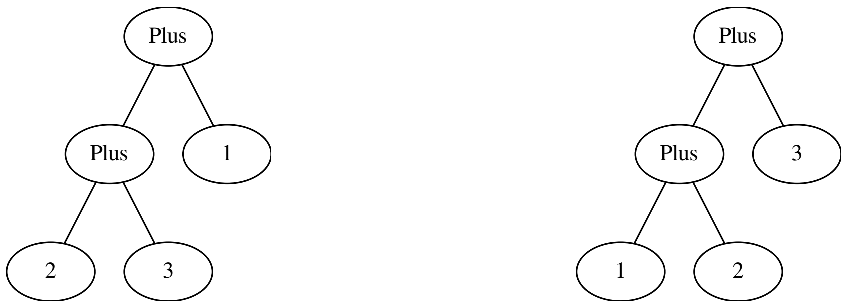
prefer, avoid: Other disambiguation attributes SDF supports are **avoid** and **prefer**. Consider the following productions to introduce a typical if-then-else construct:

```
"if" Expr "then" Expr             -> Expr {cons("If")}
"if" Expr "then" Expr "else" Expr -> Expr {cons("IfElse")}
```

These productions introduce the infamous *dangling else* problem. The problem gets immanent when trying to parse the following input:

```
if c then
  if ... then ...
else e
```

The indentation reflects the way the programmer expects the given code to parse: The outer *if-else* guards the inner *if*. If *c* is not true then *e* is executed. This indentation is however not expressed in the grammar and the input is ambiguous. A different indentation for the input illustrates the alternative parse result:



(a) Right associative **(b)** Left associative

Figure 2.2.: Ambiguous parse results for expression `1 + 2 + 3`

```
if c then
  if ... then ...
  else e
```

In this case the outer simple *if* guards the inner *if-else*. If *c* is not true then nothing is executed. In summary it is not clear if the *dangling else* should be associated with the inner or the outer *if*. The attributes **avoid** and **prefer** help to disambiguate this problem. As the names suggest a production flagged with **avoid** or **prefer** is avoided or preferred over another possible production, respectively. Adding the **prefer** attribute to the *IfElse* production reflects the intent of the first indentation as the *if-else* is preferred over the simple *if*. Adding **prefer** to the attributes of *If* reflects the intent of the second indentation. Alternatively **avoid** could be added to the opposite production to achieve the same result. In this context the productions

```
"if" Expr "then" Expr                 -> Expr {cons("If")}
"if" Expr "then" Expr "else" Expr -> Expr {cons("IfElse"), avoid}
```

also reflect the intent of the second form of indentation.

Priority Groups: Another disambiguation tool in SDF are priority groups. The naive expression grammar from Listing 2.3 can again be used to motivate the use of these priority groups. The following input is ambiguous with this grammar:

```
1 + 2 * 3
```

The two possible parse results are `Mult(Plus(1, 2), 3)` and `Plus(1, Mult(2, 3))`. This problem looks similar to the problem of operator associativity which could be resolved with the **left** and **right** attributes. It is however different, as the ambiguity is between two different `Expr` productions, `Mult` and `Plus`, whereas the associativity problem is an ambiguity introduced by only one production. The targeted parse result is `Plus(1, Mult(2, 3))` as it is common sense from arithmetics that the multiplication operator (`*`) should bind stronger than the plus operator (`+`). In other words: the multiplication operator has a higher priority than the plus operator. But the multiplication operator has the same priority as the division operator (`/`) and the plus operator has the same priority as the minus operator (`-`).

The arithmetics grammar using priority groups is given in Listing 2.4. The **context-free priorities** keyword indicates the upcoming use of priorities and replaces the previous **context-free syntax** keyword. The operator rules are moved into blocks according to their priority. A greater than sign combines the blocks indicating that the first block has a higher priority than the second

```
context-free priorities
  INT -> Expr
  > {
    Expr "*" Expr -> Expr {cons("Mult"), left}
    Expr "/" Expr -> Expr {cons("Div"), left}
  }
  > {
    Expr "+" Expr -> Expr {cons("Plus"), left}
    Expr "-" Expr -> Expr {cons("Minus"), left}
  }
```

Listing 2.4: Arithmetics with priority groups

block. This version of the grammar is unambiguous as it defines priorities of the operators as well as their associativities.

Selective Priorities: Priorities defined in this manner are in effect for all recursive sorts in the production. Generally this property is desired, but there are exceptions from the rule. Consider an addition to the language to abstract over the arithmetic expressions and allow to bind subexpressions to variables:

```
let x = 20 + 3 {
  x * 42
}
```

But it should not be possible to repeat the let statement in binding position or use it as operand. So the following expressions should not be valid syntax:

```
let x = let y = 666 {y}{
  x
}

let x = 42 {x} + 3
```

The SDF arithmetics grammar can express this new let statement with the following addition to the context-free priorities:

```
> "let" ID "=" Expr "{" Expr "}" -> Expr {cons("Let")}
> "let" ID "=" Expr "{" Expr "}" -> Expr {cons("Let")}
```

Technically the SDF disambiguator prunes all parse trees which are created from productions having lower priority as the production they are transitive children of. As the let production is added at the end of the priorities it has lower priority as all other productions. Because of this the disambiguator will disallow a let production as an operand. Additionally the let production has lower priority as itself which further disallows nested let statements. Unfortunately this disallows all nested let expressions, including the following, which is a desired one:

```
let x = 20 + 3 {
  let y = 42 {
    x * y
  }
}
```

To allow the last expression but disallow a nested let in binding position a SDF grammar can use selective priorities. These are expressed by writing the offset to the symbol in question in angle brackets before the greater than sign of the priority. The desired result is expressed by changing the repeated let priority to the following:

```
<3> > "let" ID "=" Expr "{" Expr "}" -> Expr {cons("Let")}
```

The offset starts from zero, so <3> refers to the first *Expr* sort in the production. With the selective priority a nested **let** is not allowed in binding position of another **let**, but can still be used as its body.

2.2 Challenges and their Respective Solutions

Providing an SDF grammar for Scala includes some challenges. The Scala Language Specification Version 2.8 [24] does indeed provide a context-free grammar to define the syntax. This grammar however avoids left-recursive productions and is thus not well-suited for an idiomatic SDF definition. It additionally does not reflect important aspects concerning the parsing, for example associativity of operators or different interpretation of newlines depending on code region. The specification only provides this information in form of text. It also lacks information for macro definitions and string interpolation, as these were first introduced in Scala v2.10.

Nevertheless we took the grammar from the specification as a basis, but had to modify it in large parts to make it suitable for SDF. Scalas rich syntax makes it easy to introduce ambiguities, which was a general challenge while developing the grammar. In cases of uncertainty of the correct parse result we relied on the behaviour of the Scala compiler v2.10.3. The greatest challenges and their solutions are discussed in the following.

2.2.1 Versatile Identifiers

Scala allows Unicode characters for identifiers. This includes identifiers for operators, where for example ⇒ can be used for => or ← for <-. SDF2 has unfortunately no support for Unicode, which reduces the allowed input tokens to ASCII characters. This requires all Unicode identifiers to be converted to ASCII substitutes in input sources, but should not have a larger inpact on the grammar. It should be easy to add Unicode characters to the lexical definitions as soon as Unicode support is added to SDF.

Identifiers in Scala come in multiple forms. They are composed of four types of character classes: *lower*, *upper*, *digit* and *opchar*. These classes are originally defined over Unicode in the specification but reduced to ASCII they can be listed as follows:

```
upper  = ? A-Z ? | "$" | "_"
lower  = ? a-z ?
digit  = ? 0-9 ?
opchar = "!" | "#" | "%" | "&" | "*" | "+" | "-" | "/" | ":"
       | "<" | "=" | ">" | "?" | "@" | "\" | "^" | "|" | "~"
```

From these character classes, identifiers can be composed by the following productions:

```
op       = opchar, {opchar}
varid    = lower, idrest
plainid = upper, idrest
         | varid
         | op
id       = plainid
         | "‘" stringLit "‘"
idrest   = {letter | digit}, ["_", op]
```

Pure operators are arbitrary characters from the *opchar* character class. Variable identifiers start with a lower case letter, followed by the rest. The counterpart to variable identifiers are constant identifiers, which start with an upper case letter, followed by the rest. Plain identifiers are either constant identifiers, pure operators or variable identifiers. Apart from plain identifiers also exist free-form identifiers, which are delimited by backticks. They allow all characters in between the backticks which may be used in string literals, even including whitespace. The *rest* is composed of letters and digits and may have an optional operator suffix which is set apart with an underscore. The distinction between variable and constant identifiers is necessary for pattern matching expressions, where variable identifiers are allowed at different positions compared to plain identifiers.

In Scala it is possible to define arbitrary operators for classes and objects. To do this one defines the operators as regular methods where the name of the method is the operator of choice. The Scala compiler has in-built desugarings which translate prefix, infix or postfix expressions to method applications. For example an infix expression of the form `foo + bar` is desugared to `foo.+(bar)`. Only the operators `-`, `+`, `~` and `!` are allowed to be used as prefix operators. These are desugared to applications of methods, where *unary_* is prepended to the operator. For example is `!foo` desugared to `foo.unary_!`. Postfix expressions are respectively desugared from `foo!` to `foo.!`. It is noteworthy that this desugaring also happens for non-operator methods. So `foo and bar` also desugars to `foo.and(bar)`.

A disambiguation challenge arises if there are multiple underscores in an identifier and there is no whitespace between identifiers. The expression _a_+b_- is a good example. Possible parse results without disambiguation could be

```
Infix(_a_, +, b_-),
Postfix(_a_+, b_-) or
Postfix(Infix(_a_, +, b_), -).
```

Per definition the expressions are disambiguated by using a longest-match rule for identifiers, which makes the second possibility the winner. Expressing this longest-match rule in SDF is achieved with follow restrictions on the lexical syntax.

The SDF definitions which allow parsing variable identifiers with the longest-match property are given in Listing 2.5. The character classes `UPPER`, `LOWER`, `LETTER` and `OPCHAR` are defined according to the preceding description. But notably the underscore has been removed from `UPPER` as it plays a special role as delimiter for the operator suffix. The only production that can be reused from the specification is that for pure operators (`OP`) and free-form identifiers (`FREE-FORM-ID`). All others productions need to be changed to correctly express the longest-match rule in combination with operator suffixes.

```
lexical syntax
  [A-Z] | [\$]    -> UPPER
  [a-z]           -> LOWER
  UPPER | LOWER   -> LETTER
  [0-9]           -> DIGIT
  LETTER | DIGIT  -> ID-REST

  [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~] -> OP-CHAR
  OP-CHAR+ -> OP

  LOWER                            -> IVAR-ID
  (IVAR-ID | IVAR-ID-USS) ID-REST  -> IVAR-ID
  (IVAR-ID | IVAR-ID-USS) [\_]     -> IVAR-ID-USS
  IVAR-ID-USS OP                   -> IVAR-ID-OP

  IVAR-ID      -> VAR-ID
  IVAR-ID-USS  -> VAR-ID-USS
  IVAR-ID-OP   -> VAR-ID-OP
  (VAR-ID | VAR-ID-USS | VAR-ID-OP) -> IVAR-PLAIN-ID
  IVAR-PLAIN-ID -> VAR-PLAIN-ID

  %% ... -> *CONST-ID*

  IVAR-PLAIN-ID    -> IPLAIN-ID
  ICONST-PLAIN-ID  -> IPLAIN-ID
  OP               -> IPLAIN-ID

  IPLAIN-ID  -> PLAIN-ID

  KEYWORD  -> VAR-PLAIN-ID {reject}
  KEYWORD  -> PLAIN-ID     {reject}

  "‘" ~[\‘]+ "‘"  -> FREE-FORM-ID
  FREE-FORM-ID    -> PLAIN-ID

lexical restrictions
  OP  -/- [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]
  VAR-ID      -/- [A-Za-z0-9\$\_]
  VAR-ID-USS  -/- [A-Za-z0-9\$\_] \/ [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]

context-free syntax
  PLAIN-ID  -> Id {"Id"}
```

Listing 2.5: SDF productions for longest-match identifiers

The definition of variable identifiers is split into three cases: (1) a pure variable identifier (`VAR-ID`), which does not contain any operator characters and does not end with an underscore, (2) an underscore suffix (`VAR-ID-USS`) variable identifier, which ends with an underscore, and (3) a variable operator identifier (`VAR-ID-OP`), which has an operator suffix. All of these are defined with the help of intermediate (`I*`) counterparts which do not have follow restrictions. These are needed because otherwise the identifiers would be refused too early even if the final result would be valid. `IVAR-PLAIN-ID` is the combination of any of the variable identifiers.

```
context-free priorities
  PREFIX-OP Expr        -> Expr {"PrefixExpr"}
  > PREFIX-OP Expr      -> Expr {"PrefixExpr"}
  > Expr INFIX-OP Expr  -> Expr {"InfixExpr"}
  > Expr POSTFIX-OP     -> Expr {"PostfixExpr"}
  > Expr POSTFIX-OP     -> Expr {"PostfixExpr"}
```

Listing 2.6: Expression priorities encoded in SDF

Constant identifiers are definied anologuous to variable identifiers. Plain identifiers (`IPLAIN-ID`) are any of the variable, constant or pure-operator identifiers. Even if there is a distinction between plain identifiers and free-form identifiers in the reference manual, free-form identifiers have been definied as plain identifiers for simplicity in our SDF grammar. The difference between `IVAR-PLAIN-ID` and `VAR-PLAIN-ID` is, that `VAR-PLAIN-ID` does not allow keywords. The same is true for `IPLAIN-ID` and `PLAIN-ID`.

Operators must not be followed by any operator char. If they are followed by an operator char, this char is also part of the operator. This definition conforms to the longest-match property for all operator identifiers, including the `VAR-ID-OP`, which is defined with the help of `OP`. Non-underscore suffix variable identifiers must not be followed by any letter nor underscores, which conforms to the longest-match rule property for `VAR-ID`. An underscore suffix variable identifier must neither be followed by a letter nor an operator char. If followed by a letter, it no longer has an underscore suffix and if followed by an operator, it is instead a `VAR-ID-OP`.

In summary `PLAIN-ID` is the lexical sort that represents all valid Scala identifiers with the necessary longest-match rules. The context-free counterpart is the `Id` sort, which wraps all identifiers in the *Id* constructor.

2.2.2 Rich Layout-Sensitive Expressions

Scala features a rich expression syntax, combining concepts known from functional and imperative programming languages. As a result, well-known imperative concepts, for example conditionals, loops or try-catch constructs, which are untyped statements in Java, are realized as typable expressions in Scala. As they are expressions, they are combinable in many different ways which can easily lead to ambiguous constructs. The first part of this section explains the difficulties with expressions on the subset of prefix, infix and postfix expressions. The second part explains the challenge of different interpretations of newlines depending on code region.

Prefix, Infix and Postfix Expressions

Scala defines prefix, infix and postfix expressions. Prefix expressions have the form `prefix-op expr`, infix expressions have the form `expr infix-op expr` and postfix expressions have the form `expr postfix-op`. Infix and postfix operators can be arbitrary identifiers, but prefix operators can only be `-`, `+`, `!`, and `~`. Prefix, infix and postfix expressions can be recursively combined to compose more complex expressions. This poses some challenges on the parsing of these expressions.

```
context-free priorities
  Expr SPECIAL-OP Expr      -> Expr {"InfixExpr"}
  > Expr MULT-OP Expr       -> Expr {"InfixExpr"}
  > Expr SUM-OP Expr        -> Expr {"InfixExpr"}
  > Expr COLON-OP Expr      -> Expr {"InfixExpr"}
  > Expr CMPR-OP Expr       -> Expr {"InfixExpr"}
  > Expr BRACKET-OP Expr    -> Expr {"InfixExpr"}
  > Expr AMPERSAND-OP Expr  -> Expr {"InfixExpr"}
  > Expr CIRCUMFLEX-OP Expr -> Expr {"InfixExpr"}
  > Expr BAR-OP Expr        -> Expr {"InfixExpr"}
  > Expr LETTER-OP Expr     -> Expr {"InfixExpr"}
  > Expr ASSIGN-OP Expr     -> Expr {"InfixExpr"}
```

Listing 2.7: Infix operator precedences encoded in SDF

Priorities: The first challenge is the order of priority for these kinds of expressions. Without further restriction or disambiguation it is not clear how to parse the input `foo and bar`. It could either result to `Infix(foo, and, bar)` or to `Postfix(Postfix(foo, and), bar)`. A prefix expression for this input is not possible, as no prefix operator is used. The desired result for the given input is the infix expression. Another example input term is `-b + c`, which could either be `Prefix(-, Infix(b, +, c))` or `Infix(Prefix(-, b), +, c)`. In this case the latter is the desired result. Another example is `! !true`, which will be refused by the Scala compiler as syntax error. In summary prefix expressions have the highest priority (bind the closest), followed by infix expressions, followed by postfix expressions, which have the least priority.

The code to express these priorities in SDF is given in Listing 2.6. It refuses all prefix, infix or postfix expressions nested in prefix expressions, all postfix expressions nested in infix expressions and all postfix expressions nested in postfix expressions. The repeated lines are no error and really needed as otherwise the first given example of `foo and bar` could not be correctly disambiguated and `! !true` would not be rejected.

Operator Precedence: The second challenge for parsing infix expressions is the operator precedence. As Scala allows arbitrary identifiers for operators, the precedence of the operators is defined on the basis of the first character used in the operator. The order is (from lowest to highest): letter, `|`, ^, &, < or >, = or `!`, `:`, + or `-`, * or / or % and finally any other character. An exception from the rule are assignment operators, which always have the lowest precedence. An assignment operator ends with =, must not be <=, >= or `!=` and must also not start with =.

Operator precedences in SDF can be expressed the same way as the priorities of prefix, infix and postfix expressions. The code is given in Listing 2.7. The solution is to repeat the infix expression production in order of precedence and using the corresponding operator lexical production. The lexical operator productions are given in Listing 2.8. The definitions of the `*-OP` sorts are straight forward. An `ASSIGN-OP` consists of an arbitrary amount of `OP-CHAR`, followed by =. A `LETTER-OP` is a variable- or constant identifier, which starts with a non-operator char and can have an optional operator suffix. A bar operator is a `|` followed by an arbitrary amount of `OP-CHAR`, and so on. The reject productions starting at line 14 are needed to avoid confusion of comparison operators with assignment operators and of any operator with keywords. The productions starting at line 31 are needed to disambiguate assignment operators from other infix operators. An operator of the form += could be a sum operator, because of the leading +, but it could also be an assignment operator, because of the trailing =. These productions make sure

```
lexical syntax
  OP-CHAR* [\=]                 -> ASSIGN-OP
  VAR-PLAIN-ID | CONST-PLAIN-ID -> LETTER-OP
  [\|] OP-CHAR*                 -> BAR-OP
  [\^] OP-CHAR*                 -> CIRCUMFLEX-OP
  [\&] OP-CHAR*                 -> AMPERSAND-OP
  ([\<] | [\>]) OP-CHAR*        -> BRACKET-OP
  ([\=] | [\!]) OP-CHAR*        -> CMPR-OP
  [\:] OP-CHAR*                 -> COLON-OP
  ([\+] | [\-]) OP-CHAR*        -> SUM-OP
  ([\*] | [\/] | [\%]) OP-CHAR* -> MULT-OP
  [\#\?\@\\\~] OP-CHAR*         -> SPECIAL-OP

  [\=] OP-CHAR* [\=] -> ASSIGN-OP  {reject}
  "="                -> ASSIGN-OP  {reject}
  "<="               -> ASSIGN-OP  {reject}
  ">="               -> ASSIGN-OP  {reject}
  "!="               -> ASSIGN-OP  {reject}
  "="                -> CMPR-OP    {reject}
  "=>"               -> CMPR-OP    {reject}
  ":"                -> COLON-OP   {reject}
  "<-"               -> BRACKET-OP {reject}
  "<:"               -> BRACKET-OP {reject}
  "<%"               -> BRACKET-OP {reject}
  ">:"               -> BRACKET-OP {reject}
  "#"                -> SPECIAL-OP {reject}
  "@"                -> SPECIAL-OP {reject}

  %% ASSIGN-OP is exception and thus more
  %% important than the other ops
  ASSIGN-OP -> LETTER-OP     {reject}
  ASSIGN-OP -> BAR-OP        {reject}
  ASSIGN-OP -> CIRCUMFLEX-OP {reject}
  ASSIGN-OP -> AMPERSAND-OP  {reject}
  ASSIGN-OP -> BRACKET-OP    {reject}
  ASSIGN-OP -> CMPR-OP       {reject}
  ASSIGN-OP -> COLON-OP      {reject}
  ASSIGN-OP -> SUM-OP        {reject}
  ASSIGN-OP -> MULT-OP       {reject}
  ASSIGN-OP -> SPECIAL-OP    {reject}

lexical restrictions
  ASSIGN-OP       -/- [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]
  LETTER-OP       -/- [a-zA-Z0-9]
  BAR-OP          -/- [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]
  CIRCUMFLEX-OP   -/- [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]
  AMPERSAND-OP    -/- [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]
  BRACKET-OP      -/- [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]
  CMPR-OP         -/- [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]
  COLON-OP        -/- [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]
  SUM-OP          -/- [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]
  MULT-OP         -/- [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]
  SPECIAL-OP      -/- [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]
```

Listing 2.8: Lexical productions for the different infix operators

```
context-free priorities
%% ...
  > {
    Expr SPECIAL-OP Expr          -> Expr {"InfixExpr", left}
    Expr RASSOC-SPECIAL-OP Expr   -> Expr {"InfixExpr", right}
  }
  %% ...
  > {
    Expr LETTER-OP Expr          -> Expr {"InfixExpr", left}
    Expr RASSOC-LETTER-OP Expr   -> Expr {"InfixExpr", right}
  }
  > Expr ASSIGN-OP Expr     -> Expr {"InfixExpr", left}
%% ...
```

Listing 2.9: Code to support associativities for infix expressions

```
lexical syntax
  (VAR-PLAIN-ID | CONST-PLAIN-ID) [\:] -> RASSOC-LETTER-OP
  [\|] OP-CHAR* [\:]                   -> RASSOC-BAR-OP
  %% ...
  [\#\?\@\\\~] OP-CHAR* [\:]           -> RASSOC-SPECIAL-OP

  "<:" -> RASSOC-BRACKET-OP {reject}
  ">:" -> RASSOC-BRACKET-OP {reject}

  %% Right associative identifiers have higher priority than
  %% their left associative counterparts
  RASSOC-LETTER-OP     -> LETTER-OP     {reject}
  %% ...
  RASSOC-SPECIAL-OP    -> SPECIAL-OP    {reject}

lexical restrictions
  RASSOC-LETTER-OP      -/- [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]
  %% ...
  RASSOC-SPECIAL-OP     -/- [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]
```

Listing 2.10: Definition of right associative infix operators

that assignment operators can not be parsed as other operators. The lexical restrictions starting from line 43 are required to enforce the longest-match property on the operators.

Associativity: The third challenge is the associativity of infix expressions. In a similar fashion to operator precedence, associativity is defined on the last character of the operator. All operators ending in `:` are right-associative, the others are left-associative. A common example for a right-associative operator is the list constructor `::`, as used in `1 :: 2 :: Nil`. This expression is expected to parse to `Infix(1, ::, Infix(2, :: Nil))`, opposed to the left-associative variant of `Infix(Infix(1, ::, 2), ::, Nil)`.

Fortunately SDF has support for parsing different associativities, which yields the code in Listing 2.9. `RASSOC-*-OP` definitions are added besides the `*-OP` definitions. These have the same precedences as their left-associative counterparts and are thus added into a block with the same priority. Solely the associativity is adapted accordingly by adding the **right** keyword to the attributes. The `RASSOC-*-OP` lexical sorts are given in Listing 2.10. Similar to the assignment

```
context-free priorities
%% ...
  > PREFIX Expr
    -> Expr {"PrefixExpr", prefer, layout("1.last.line == 2.first.line")}
  > {
    Expr INFIX-OP Expr
    -> Expr {"InfixExpr", left, layout(
      "1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
    Expr RASSOC-INFIX-OP Expr
    -> Expr {"InfixExpr", right, layout(
      "1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
  }
  > Expr Id
    -> Expr {"PostfixExpr", avoid, layout("1.last.line == 2.first.line")}
%% ...
```

Listing 2.11: Layout constraints for prefix-, infix- and postfix expressions

operator, the right associative operators are defined by requiring a trailing `:`. Here again the priority is enforced with rejection productions and the longest-match property is described with the help of restrictions.

Layout Constraints: The fourth and last challenge concerning prefix, infix and postfix expressions are optional newlines and same-line constraints. Prefix and postfix expressions need their operator on the same line, or will otherwise be splitted into two seperate expressions. Infix expressions may have one optional newline after the operator.

Both optional newlines and same-line constraints can be expressed in SDF with layout constraints [13]. The general idea behind the layout-sensitive parsing is to attach line and column information to the beginning and end of parsed symbols. With help of a small DSL and new attributes, which can be attached to productions, one can then define constraints on relative positions between symbols in one production. Originally developed to make parsing of layout-sensitive languages like Python or Haskell possible with SDF, the layout constraint extension also has some useful features to handle newlines.

The resulting code for expressions using layout constraints is given in Listing 2.11. Prefix expressions get the constraint that the line of the last token of the operator must be the same as the line of the first token of the `Expr`. The constraint for the postfix expression is analogous. The infix operation has the constraint that the line of the last token of the left-hand-side expression is equal to the line of the first token of the operator and that the last token of the operator and the first token of the right-hand-side expression are not more than one line apart.

Newline Regions

The Scala Language Specification defines a special `nl` token. Whether this token can be consumed as such or just acts as layout depends on the code before and after this token. It is deactivated in between parentheses (`"("`, `")"`), but can be reactivated between curly braces (`"{"`, `"}"`). A direct translation of this condition to SDF is not possible but can be encoded with layout constraints and two sorts for expressions: One with layout constraints enabled and the other which does not enforce layout constraints.

```
%% With Layout Constraints
context-free priorities
  %% ...
  "(" {NoLExpr ","}* ")" -> Expr {"TupleExpr"}
  %% ...
  > BlockExpr -> Expr
  %% ...
  > Expr Op Expr -> Expr {"InfixExpr", left,
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
  %% ...

%% No Layout Constraints
context-free priorities
  %% ...
  "(" {NoLExpr ","}* ")" -> NoLExpr {"TupleExpr"}
  %% ...
  > BlockExpr -> NoLExpr
  %% ...
  > NoLExpr Op NoLExpr -> NoLExpr {"InfixExpr"}
  %% ...

%% Common
context-free syntax
  %% ...
  "{" Block "}" -> BlockExpr {"BlockExpr"}
  %% ...
```

Listing 2.12: Newline regions in expression productions

An excerpt of the resulting productions is given in Listing 2.12. It shows the two different sorts for expressions and three productions for each. The sort with layout constraints enabled is `Expr`. The sort without layout constraints is `NoLExpr`. The three sample productions chosen are tuples (expressions in parentheses), block expressions (expressions in curly braces) and infix expressions. As stated earlier the `nl` is deactivated inside of parentheses. So all expressions sorts inside parentheses are `NoLExprs`. Inside curly braces the `nl` is activated, so all expression symbols used in `Block` are `Expr`.

This way the `nl` can be activated or deactivated depending on the code region. It unfortunately requires to copy all 51 `Expr` productions to `NoLExpr` productions with the layout information removed. This is a major drawback in terms of extensibility and maintainability, as a change in the `Expr` productions must be manually propagated to the `NoLExpr` productions. It however yields a satisfiable parser and no other working approaches could be found.

2.2.3 Statement Termination

Newline as Statement Terminator

Scala allows the termination of statements with either semicolons or newlines. This property seems to be trivial to implement in a context-free grammar definition. Listing 2.13 shows a prototypical example of how one would like to express package declarations statements which can be terminated with newlines. Unfortunately this example does not work.

```
module PackageDeclaration
exports
  lexical syntax
    [a-zA-Z\_] -> ID

  context-free syntax
    "package" ID ";"     -> PackageDeclaration {cons("PackageDeclaration")}
    "package" ID "\n"    -> PackageDeclaration {cons("PackageDeclaration")}
    PackageDeclaration* -> CompilationUnit {cons("CompilationUnit")}

  context-free start symbols
    CompilationUnit
```

Listing 2.13: Naive approach to define newline-terminatable package declarations

```
%% ...
  lexical syntax
    "\n"* -> NL

  lexical-restrictions
    NL -/- "\n"

  context-free syntax
    NL "package" NL ID NL ";" NL -> PackageDeclaration {...}
    NL "package" NL ID "\n" NL    -> PackageDeclaration {...}
%% ...
```

Listing 2.14: Package declarations with explicit newlines

The reason for this is, that newlines can also be considered layout; characters insignificant for the derivation of the AST. As explained in Section 2.1 SDF uses a special LAYOUT sort to define these insignificant derivations like whitespace or commentaries. If LAYOUT however contains newline characters then these will be consumed by this sort and can not be used by other productions including the statement productions.

In summary using newlines in productions for statement termination (as in Listing 2.13) and adding newlines to LAYOUT is not possible together. Three workarounds for this problem have been examined in the making of this work:

1. Not adding newlines to LAYOUT and specify them explicitly in the grammar
2. Using simple SDF syntax and circumvent the injection of LAYOUT
3. Using experimental layout constraints

Explicit Use of Newlines: The most obvious approach to avoid adding newlines to LAYOUT and using simple context-free syntax is to not add newlines to LAYOUT. Instead one has to explicitly give optional newlines everywhere in the grammar where they could be used as insignificant characters. Reusing the PackageDeclaration sample, this would lead to a grammar similar to Listing 2.14. The leading NL is needed for mixed-indentation, or for newlines before the first package declaration. The restriction is necessary to disambiguate the case of one NL following another NL, in which the first will always consume all newlines and the second will be empty.

```
%% ...
  lexical syntax
    (" " | "\t")*  -> SPACE

  syntax
    "package" <LAYOUT?-CF> <ID-LEX> <SPACE-LEX> "\n"
      -> <PackageDeclaration-CF> {...}

  context-free syntax
    "package" ID ";" -> PackageDeclaration {cons("PackageDeclaration")}
%% ...
```

Listing 2.15: Package declarations with plain SDF syntax

One obvious drawback of this approach is the loss of comprehensibility. The grammar is harder to read with cluttering `NL` tokens in each production. The grammar is also more error prone: It is easy to forget the `NL` token before or after the semicolon. Especially when changing the grammar the adaption of the newline tokens is missed quickly.

Another unfortunate effect is the cluttering of the resulting AST. Each `NL` will be another argument for the constructor of the package declaration. So the constructor for the package declaration with the semicolon will be 5-ary (NL, NL, ID, NL, NL), whereas the constructor for the package declaration with the terminating newline will be 4-ary (NL, NL, ID, NL). The constructors could be rewritten in a post processing step but then one would have to deal with this explicitly as the auto-generation capabilities of *Spoofax* would not handle these cases.

Circumvent Injection of `LAYOUT`: An alternative approach, which allows adding of newlines to `LAYOUT`, is the use of plain syntax (non-lexical and non-context-free). Definition of this plain syntax is an undocumented feature of SDF and most probably not intended to be used by language developers. It however allows to circumvent the automagical insertion of the `LAYOUT` symbol as it would happen in the context-free syntax definition. This way the newline can be consumed by the `PackageDeclaration` sort. As the `LAYOUT` in plain syntax is no longer inserted automatically it however has to be inserted manually.

How to use plain syntax in context of the sample package declaration can be seen in Listing 2.15. The plain syntax definition is not as high-level as the lexical or context-free definition and distinguishes between the different sorts. Thus the `LAYOUT` sort must be specified as `<LAYOUT?-CF>` between the **package** keyword and the `ID` lexical token. As there may be tabulators or spaces after the `ID` and before the terminating newline the `SPACE` lexical token is introduced and used. This token also shows up in the resulting constructor, which will be *PackageDeclaration*(ID, SPACE).

Compared to the previous approach this one only requires a cumbersome notation in each production which needs to consume a newline character. In return the comprehensibility of these productions is even worse compared to those of the explicit-newline-approach. The biggest drawback however is the use of an undocumented feature which is most-likely not intended to be used by language developers.

Layout Constraints: The last approach investigated to express newline-terminated statements is the use of layout constraints for SDF. A grammar utilizing layout constraints to make parsing Scala-like package declarations possible is given in 2.16. The notable changes compared to

```
%% ...
lexical syntax
  ";" -> SEMI
      -> EOL

context-free syntax
  "package" ID SEMI -> PackageDeclaration {...}
  "package" ID EOL  -> PackageDeclaration {..., enforce-newline}
%% ...
```

Listing 2.16: Package declarations with layout constraints

the first version in Listing 2.13 are the introduction of lexical symbols SEMI and EOL and the use of the **enforce-newline** attribute for the newline-terminated package declaration. The EOL production is empty and can thus always be consumed. The **enforce-newline** attribute adds the constraint to the production, that there must be a newline between the second-to-last and last symbol in the production – all parse results where this is not the case will be pruned. The empty EOL is therefore needed to be the token on the next line. The SEMI symbol is not necessary for the desired behaviour, but introduced to keep the arity of the resulting constructor for package declarations equal. The EOL will always show up as the second parameter to the constructor and tree manipulations are easier if there are not two PackageDeclaration constructors with different arity.

Compared to the other two alternative approaches to handle newline-terminated statements the layout constraints keep the comprehensibility of the first trivial version from Listing 2.16. The grammar does not seem to be cluttered or overly-complicated with this approach. The experimental Spoofax release already has support for layout constraints out of the box, so layout constraints can be expected to be officially supported in future releases. For these two reasons we decided to express the termination of statements with the help of layout constraints.

Need for longest-match

Block statements are defined similar to package declaration statements and have the following form:

```
BlockStat SEMI  -> BlockStatSemi {...}
BlockStat EOL   -> BlockStatSemi {..., enforce-newline}
```

One of the productions for BlockStat is:

```
Expr -> BlockStat
```

BlockExpr and Block further have the productions:

```
"{" Block "}"  -> BlockExpr
BlockStatSemi* -> Block
```

With these definitions, the following block is ambiguous:

```
{
  f()
  c1 &&
  c2
}
```

The reason for the ambiguity are the two last lines of the block. It is not clear if the block should parse to a list of three block statements, namely the application of *f*, a postfix operation on *c1* with the operator && and the identifier *c2*, or to a list of only two block statements, namely the application of *f* and an infix expression over two lines. Infix and postfix expression are indeed disambiguated, but this disambiguation for the `Expr` sort does however not apply for the disambiguation of possible parse results for block statements. The right disambiguation for block statements is to always decide for the longest possible match. In the given example the infix expression over two lines is a longer match than the two short matches of the postfix expression and the identifier and should thus win. Fortunately the layout constraint extension has the keyword **longest-match** exactly for that purpose. Adding this keyword to the `BlockStatSemi` productions has the desired effect:

```
BlockStat SEMI  -> BlockStatSemi {..., longest-match}
BlockStat EOL   -> BlockStatSemi {..., enforce-newline, longest-match}
```

This same pattern is also applied for all other statement-like productions, e.g. statements on template level or enumerators.

2.3 Evaluation

We evaluate the grammar with two different methods:

The first method uses the testing capabilities provided by the Spoofax framework [19]. These testing capabilities allow to declare named tests on the grammar. Such a test is composed of the start symbol to use, an input fragment and an expected result. The input fragment is then parsed using the specified start symbol and compared to the expectation. The possible expectations can be an AST pattern, general unambiguous parsability or parse failure, among other things. We use this method to test the grammar for correctness.

The second method uses an individually developed command line batch processing utility. This utility takes a parse table and a list of files as input, parses the files one after another with the Spoofax JSGLR parser and reports on the parse results. Options for the tool are the start symbol to use, a timeout for each individual file and the output of the report as comma separated values (CSV) for further processing of the data. The report includes a label for the general parse result, time needed, lines of code, number of characters without line breaks and possible use of XML for each file. The label for the parse result is one of the following:

Success The file could be parsed successfully under the time limit, the start symbol did match and no ambiguity occurred.

Ambiguity The file could be parsed successfully under the time limit, the start symbol did match but at least one ambiguity occurred.

Startsymbol Mismatch The file could be parsed, but the start symbol did not match. Ambiguities may have occurred.

Timeout The file could not be parsed in the provided time limit.

Failure The file could not be parsed due to invalid syntax.

Error Some unexpected error occurred while parsing the file.

Because SDF2 can not handle UTF-8 encoding the batch testing utility is accompanied by a shell script to replace Unicode characters with ASCII counterparts. This script can be used to pre-process the input files before the actual test. We use this second evaluation method to test the grammar against possible ambiguities. It however also has the side effect of yielding data usable for performance measures.

One of the correctness tests which expects an AST looks like the following:

```
language Scala
start symbol Expr

test right assoc operations [[
        1 :: 2 :: Nil
]] parse to InfixExpr(Int("1"), "::", InfixExpr(Int("2"), "::", Id("Nil")))
```

It tells the testing framework to use the grammar of the language *Scala* and use `Expr` as start symbol. The test is named *right assoc operations*. It tests the input `1 :: 2 :: Nil` to be actually parsed to right associative infix expressions with the `::` operator.

An example for a negative test is the following:

```
test eta expansion invalid layout [[
        f
        _
]] parse fails
```

It does not respecify the language nor the start symbol, as these need to be specified only once before the first test occurs in a test file. This test checks for the layout constraint that an eta expansion can not span over multiple lines. As the start symbol is `Expr` this fragment should fail to parse. With a different start symbol this fragment may result in a list of two expressions.

If a test does not have an explicit expectation then the fragment is implicitly expected to parse unambiguously. An example for such a test is the following, which checks that a qualified instantiation without parameters is unambiguous:

```
test two element path class instantiation [[
        new mutable.Map
]]
```

These tests are a typical result of a detected ambiguity and serve as regression tests, so the ambiguity will not be reintroduced if the grammar may change in the future.

We provide 370 of correctness tests for the grammar. 194 expect an AST, 26 are negative tests and 150 check against unambiguous parsability. The results are shown in Table 2.1. Unfortunately only 366 of these tests pass, leaving four failing tests. Three of this failing tests are negative tests and check corner cases of layout used in class definitions which should be rejected, but are not. As a result our grammar is less restrictive than the Scala parser, but the cases are hopefully unlikely enough to not have a large impact. The remaining failing test checks the right-associativity of letter operations with a trailing colon, e.g. `colon_::`, which is not given with our grammar. This is a known issue with the lexical syntax for operators and can hopefully be fixed in the future.

We use the batch testing utility to evaluate our grammar on Scala source code files under the `src` directory in the public Scala v2.10.3 repository[1] as we consider the Scala sources an ambitious

[1] https://github.com/scala/scala/tree/v2.10.3

Expectation / Result	AST	Failure	Non-Ambiguity	Total
Positive	193	23	150	366
Negative	1	3	0	4
Total	194	26	150	370

Table 2.1.: Results for the correctness tests

Label / Size	Success	Ambiguity	Symbol Mismatch	Timeout	Failure	Error	Total
Files	1522	0	0	1	0	0	1523
LOC	265452	0	0	96	0	0	265548

Table 2.2.: Parse results after XML and Unicode correction

project and representative for typical Scala code. The parse result of these 1531 files with 268522 lines of code (LOC) has been measured on an Intel(R) Core(TM) i5-3470 CPU @ 3.20GHz with a Solid State Drive and is as follows: 1521 files (265227 LOC) parse successfully, nine files (3199 LOC) fail to parse and 1 file (96 LOC) can not be parsed due to a timeout after 60 seconds. None of the files yields an ambiguous parse result.

Eight of the nine parse failures are due to the use of XML elements in the source file. XML is however explicitly not part of the grammar and thus the parse failures do not indicate incorrectness on our work. The one remaining parse failure is due to a substitution of a Unicode character literal in between single quotes with two ASCII characters. The resulting illegal expression `'=>'` causes the parse failure. The file parses successfully if the expression is changed to the legal string literal `"=>"`. As these failures are thus not really failures, the results can be updated as follows: 1523 files (265548 LOC) total, 1522 files (265452 LOC) parse successfully and 1 file (96 LOC) can not be parsed due to a timeout. The corrected parse results are also provided in Table 2.2.

The important part of the file causing the timeout is given in Listing 2.17. We could not figure out the exact reason why this piece of code is so hard to parse, but it is a combination of the following circumstances:

- The long boolean expression from line five to 16 is wrapped over multiple lines. If written in one line the code can be parsed in about 7.3 seconds.
- The expression is the body of the case clause. Without the match expression the file can be parsed in about 1.8 seconds.
- The expression is the direct body of the case clause. If wrapped in a block the file can be parsed in about 7.5 seconds.
- The pattern variable begins with the character *c*. If *cs* is replaced by *s* the file can be parsed in about 4.7 seconds.

Several other — but not as drastic — outliers can be seen on the plot of parse time over LOC for all successfully parsed files in Figure 2.3. The parse time result characteristics are provided in Table 2.3. The initialization of the parser before the first file was parsed took 0.576s. The

```
class Settings {
  //...
  override def equals(that: Any): Boolean = that match {
    case cs: Settings =>
      this.gBf == cs.gBf &&
      this.uncheckedBf == cs.uncheckedBf &&
      this.classpathBf == cs.classpathBf &&
      this.sourcepathBf == cs.sourcepathBf &&
      this.sourcedirBf == cs.sourcedirBf &&
      this.bootclasspathBf == cs.bootclasspathBf &&
      this.extdirsBf == cs.extdirsBf &&
      this.dBf == cs.dBf &&
      this.encodingBf == cs.encodingBf &&
      this.targetBf == cs.targetBf &&
      this.optimiseBf == cs.optimiseBf &&
      this.extraParamsBf == cs.extraParamsBf
    case _ => false
  }
  //...
}
```

Listing 2.17: Piece of Scala code causing a timeout for the parser

Size	Parser Init.	Minimum	Q_1	Median	Q_3	IQR	Maximum
Time	0.576s	0ms	11ms	27ms	78ms	67ms	15.59s
LOC		1	34	67	163	129	7450

Table 2.3.: Parse times for all 1521 successfully parsed files

quartiles of the parse times in milliseconds are $(11, 27, 78)$, the interquartile range is 67. The minimum parse time was below one millisecond and was thus measured as 0, the maximum however is $15.59s$ for a file of only 1103 LOC. The quartiles of the lines of codes are $(34, 67, 163)$, the interquartile range is 129. The minimum for lines of codes is 1 and the maximum is 7450. The median of the parse times is visualized as a horizontal rule. The median of the lines of codes is visualized as a vertical rule. The remaining oblique line goes through the origin and the intersection of the medians to provide a visual help. We assume that most source code files will be below 1000 lines of code and thus provide another plot of the subset of the data with LOC $\leqslant$ 1000 and the same visual helper lines in Figure 2.4. This subset still covers 97.436% of all the files from the sample. Two additional plots with other magnifications are provided in Appendix B.

From a performance perspective about 95% of all the files can be parsed below 550ms, but the parser generated from the grammar has a strong weakness for outliers. Considering neither the grammar nor the parser are tailored towards performance and still open for improvement we however argue this is fair enough. More importantly allows the grammar to generate a parser which can successfully parse 99.9% of 1523 relevant files (265548 LOC) in the Scala source repository without ambiguities. It is additionally backed with 366 successful correctness tests which in summary indicate good suitability for the Scala language.

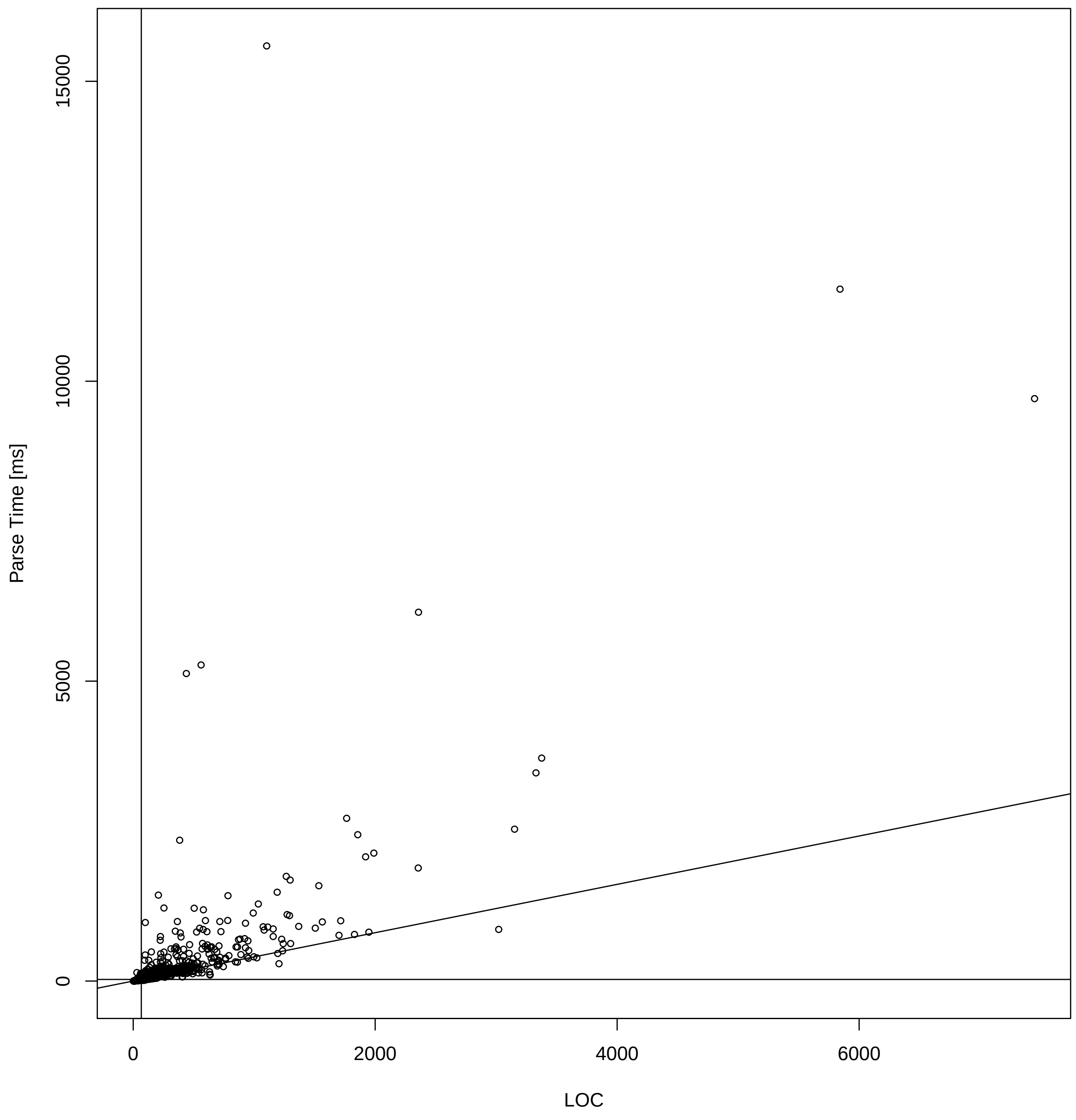


Figure 2.3.: Parse times for all 1521 successfully parsed files — lines mark medians and resulting gradient through origin

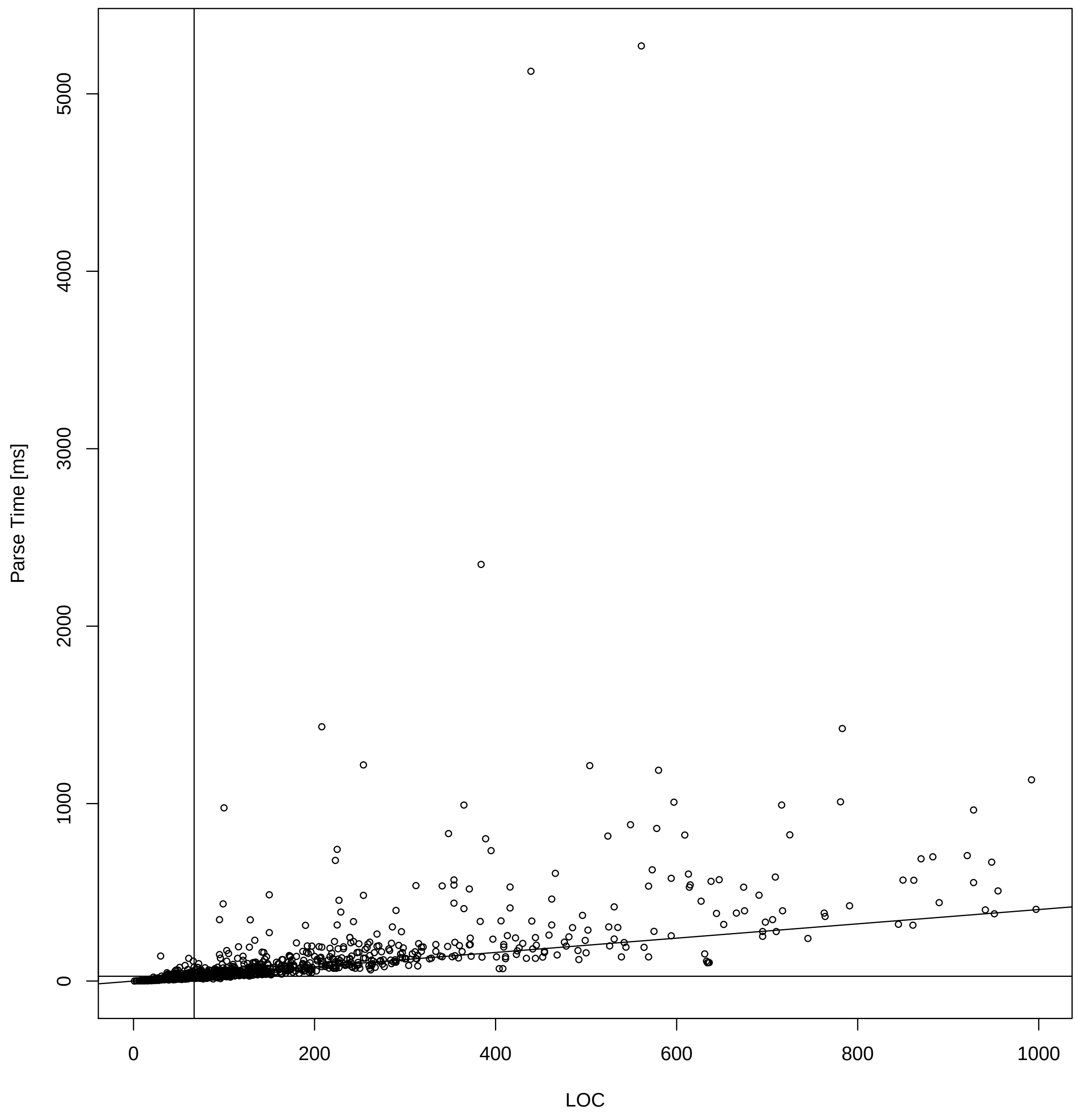


Figure 2.4.: Parse times for the 1482 successfully parsed files with LOC $\leqslant$ 1000 — lines mark medians and resulting gradient through origin

3 SugarScala as Sugar* Instantiation

SugarJ [12] allows library-based syntactic extensibility for the Java programming language. The SugarJ compiler is able to modify its parser if it encounters import statements resolving to sugar libraries. These sugar libraries provide new syntax in form of grammar extensions in SDF and accompanying Stratego rules to transform these syntatic extensions back into the base language. Additionally the sugar libraries can define static analysis for the new syntax, again in form of Stratego rules, or modify the appearance of the code in the Eclipse editor.

The Sugar* Framework [14] is an advancement from SugarJ, which adds support for arbitrary languages besides Java in the form of language plug-ins. Such a language plug-in is created by implementing two well-defined Java interfaces and providing the base language as a SDF grammar definition. Other Spoofax resources as pretty-printing tables, Stratego term rewriting rules or Eclipse editor definitions for syntax-highlighting are also supported.

The structure of this chapter is as follows: Section 3.1 explains unparsing of Scala ASTs created from the SDF grammar. Section 3.2 introduces Stratego, which is used for the desugarings. The implementation of SugarScala is described in Section 3.3. Sections 3.4 and 3.5 present the case studies for XML and EScala. Section 3.6 concludes this chapter with a discussion of SugarScala.

3.1 Unparsing of SDF Grammars

The process of turning the parse result back into source code is called *unparsing*, as it is the reverse operation to parsing. A specialized form of unparsing is *pretty-printing*, which has the additional ambition to create visually appealing results for humans or adhere to common style guides. This distinction is however vague and pretty-printing is often used synonymously to unparsing.

Unparsing is necessary in the context of SugarScala to turn ASTs into source code, which can be passed to the Scala compiler for the actual compilation. The Spoofax language workbench has built-in support for unparsing. It uses the *Box* [31] language to specify pretty-printing instructions for application constructors. These instructions are stored in pretty-printing tables and can be looked-up on unparsing of an AST. Spoofax automatically derives the pretty-printing table for a language with the help of the productions in the grammar, but allows language developers to manually override entries in the table. The automatic derivation of pretty-printing instructions unfortunately fails for productions defined in a **context-free priorities** scope or in combination with layout constraints.

The problem with the pretty-printing instructions for productions in **context-free priorities** can be solved by simply repeating the productions in **context-free syntax**. Layout constraints are however not understood by the pretty-printing table generator. This requires manual intervention on some of the pretty-printing rules to assure correct unparsing. The issue can be shown with the AST for the following source code:

```
{
  foo postOp

  bar
}
```

The AST of this source code is a block with two statements: a postfix expression and an identifier. Unparsing this AST with the automatically derived pretty-printing rules yields the following source code:

```
{ foo postOp  bar  }
```

Reparsing this source code would yield a block with only an infix expression, which is not correct. We thus manually modified some of the pretty-printing rules to produce the right amount of whitespace to avoid incorrect unparsings. The result is however not visually pleasing and only suitable for unparsing but not for pretty-printing.

3.2 Introduction to Stratego

Stratego is a domain specific language for term transformations. It is part of the Stratego/XT [3] tool suite and a large and mature project on its own. This work uses Stratego to transform parse results from SDF generated parsers. For this reason does the following introduction only focus on the use of Stratego in the context of parse tree rewriting in the Sugar* framework and leaves out parts of the big picture.

The terms encountered in this context can have three types: Constructor applications, lists and strings. Constructor applications have an identifier – the name of the constructor – and a variable amount of arguments. They resemble named nodes with zero or more children in a tree. The SGLR parser maps the constructor attributes from the SDF grammar to constructor application terms. Lists can also have a variable amount of arguments but do not have a name. Stratego could represent them as an application with the name *List*, but it knows special abstractions for lists to make working with them easier. Strings are primitives and can not have any arguments. Because of that they are always leaves in the term tree. The SGLR parser maps lexical SDF productions, for example identifiers, to strings in the term representation.

Stratego describes term transformations with the help of rewriting rules and strategies. The intention behind rules is the matching on one term and the conditional rewrite to another term, whereas the intention behind strategies is the description of term transformations on a higher level – for example the traversal of the term tree in top-down or bottom-up manner. The distinction between rules and strategies is conceptual rather than technical. More specifically a rule is a special form of a strategy. The following descriptions will only refer to strategies but do also implicitly apply to rules.

Every strategy has an implicit context which is the currently examined term. Additionally any strategy can take other strategies or terms as arguments. A strategy can use the built-in operator `?term` to match against a term and `!term` to set a term. The expression `?Foo(_, a)` matches the current context term against the application with constructor *Foo/2* and binds the second subterm to the variable *a*. The expression `!Bar("foo")` instead sets the current context term to the application of *Bar/1* with the subterm "*foo*".

A strategy can further use operators on strategies to define itself with the help of other strategies. The sequence operator `a;b` defines a strategy as a sequence of the strategies *a* and *b*. It first applies the strategy *a*. If *a* succeeds then `;` applies *b* on the result of strategy *a*. The result of the composed strategy is then the result of strategy *b*. If strategy *a* however fails then the whole strategy fails and strategy *b* is not even tried.

Another common operator is the deterministic left choice operator `a<+b`. This one will again first try *a*. If *a* succeeds the result of the composed strategy will be the result of *a* and <+ will not further apply *b*. But if *a* fails then <+ will try *b* with the original context; any possible changes made by *a* will be reset. A near relative of `a<+b` is `a+b`. The only difference between the two is the order in which they try the strategies. Where <+ strictly tries the strategies from left to right the order of + is arbitrary.

A key concept of Stratego is the passing of strategies to other strategies. The expression `s(a<+b)` passes the composed strategy `a<+b` to the strategy *s* without evaluating `a<+b` beforehand. The notation to apply a strategy instead of passing it on is the strategy surrounded by angle brackets. In comparison the expression `s(<a> v)` will apply *a* to *v* and pass the result to *s*.

The syntax to define strategies comes in two forms:

1. `strategy = id, "=", expr`
2. `rule = id, ":", pattern, "->", expr`

The first is the more general form and usually used to define conceptual strategies. The second syntax is sugar for the common strategy definition `$id = ?$pattern ; !$expr` which is a pattern match on the context term followed by the term it should be rewritten to on match. As mentioned before this form is also called a *rule*.

3.3 Implementation

An overview of the package structure of SugarScala is given in Figure 3.1. Every Sugar* instantiation must implement two interfaces: *IBaseLanguage* and *IBaseProcessor*. The former defines methods to query basic information about the base language, the latter defines methods to process source code of the language. To make the implementation of these two interfaces easier, the framework provides two abstract classes, *AbstractBaseLanguage* and *AbstractBaseProcessor*, respectively. These are intended to be extented by the base language implementor and provide common configuration and helper code. More precisely, the base language registry is written against *AbstractBaseLanguage* and *IBaseLanguage* references *AbstractBaseProcessor*, which makes the extension of the abstract classes a quasi-requirement.

3.3.1 Language Interface

The complete type hierarchy for *ScalaLanguage* is given in Figure 3.2. *AbstractBaseLanguage* implements parts of the *IBaseLanguage* interface and *ScalaLanguage* further extends *AbstractBase-Language* and implements the rest of the required methods.

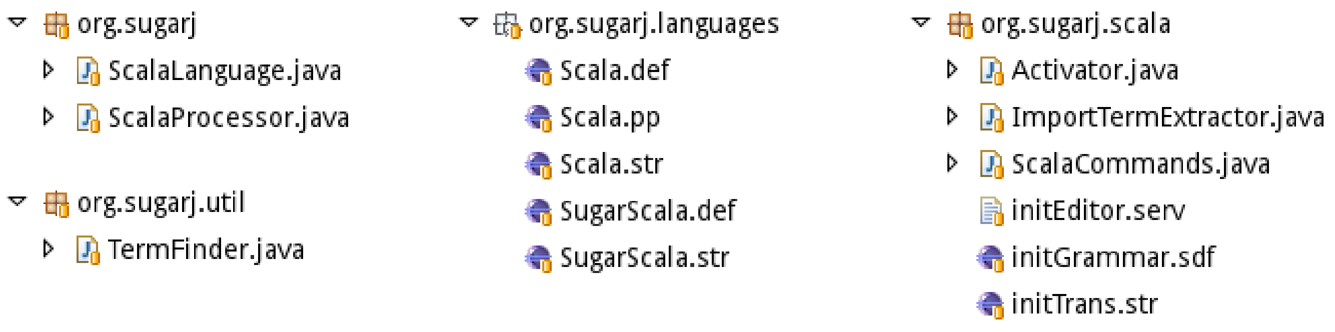


Figure 3.1.: SugarScala Project Layout

The language interface methods can be subdivided into four kinds: processor factory, configuration, initializer and predicate. The *createNewProcessor* method is the only method of the processor factory kind and is expected to create a valid processor for the language to be defined. In the case of *ScalaLanguage* this method simply instantiates a new *ScalaProcessor*. The methods of the configuration kind are *getBaseFileExtension*, *getBinaryFileExtension*, *getLanguageName*, *getSugarFileExtension*, *getVersion* and *getPackagedGrammars*. The methods of the initializer kind are *getInitEditor*, *getInitEditorModuleName*, *getInitGrammar*, *getInitGrammarModuleName*, *getInitTrans* and *getInitTransModuleName*. The methods of the predicate kind are *isBaseDecl*, *isExtensionDecl*, *isImportDecl* and *isPlainDecl*.

The base file extension is the file extension of the base language. So for *ScalaLanguage getBaseFileExtension* simply returns "scala", as base Scala source files have this file extension. The binary file extension is the file extension to which the base language compiles to. Scala compiles to *.class files, so *getBinaryFileExtension* simply returns "class". The language name is the human-readable name of the language, so *ScalaLanguage.getLanguageName* returns "Scala". The sugar file extension is the extension for files, which may contain code with syntactic sugar. For *ScalaLanguage* we choose "sugs" as sugar file extension. It is reasonably short, compared to more descriptive forms as "sugscala" or even "sugarscala" and still distinctive, which "ss" would not be, as it clashes with a scheme source file extension. The *getVersion* method simply returns a free-form version string, which may be updated during the development of the base language. The *getPackagedGrammars* is supposed to return a list of paths to SDF *.def files, which contain the base language definition as well as the form of syntactic extensions and rules for what a `ToplevelDeclaration` is. For *ScalaLanguage* this methods returns paths to two files: `Scala.def`, which contains the complete grammar for the Scala language, and `SugarScala.def`, which contains the rules for the syntactic extensions and `ToplevelDeclarations`. The `ToplevelDeclaration` rule is necessary for Sugar* to process any sugared file. The framework will process one ToplevelDeclaration at a time and then maybe adapt the further parsing accordingly.

The complete contents of `SugarScala.def` is given in Listing 3.1. The fully qualified name of the module is `org/sugarj/languages/SugarScala` and it uses the rules from the base language definition and predefined rules from `org/sugarj/languages/Sugar`. Every rule of the Scala grammar is prepended with "Scala" in the `Scala.def`, to avoid possible name clashes. The rules which may be used to parse `ToplevelDeclaration` are `PackageDeclarationSemi`, `ScalaTopStatSemi` and `ScalaExtension`. A `ScalaExtension` is composed of a head and a body. The head is the keyword "sugar", followed by a Scala id. The body of an extension is

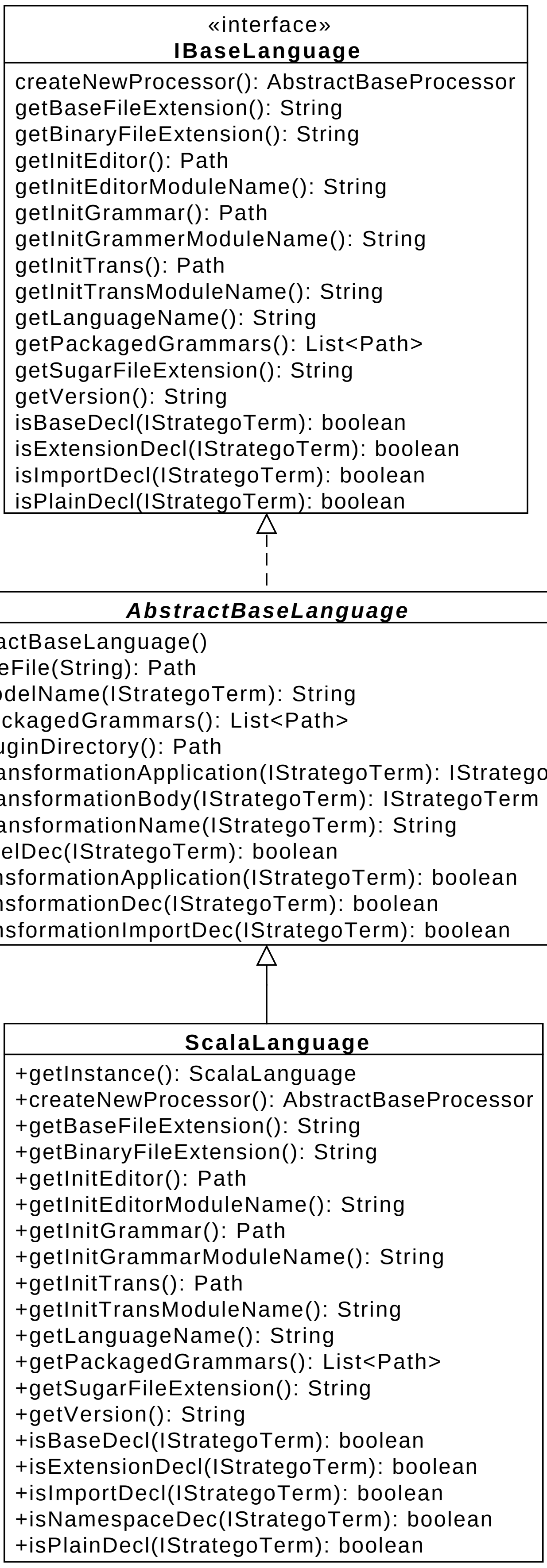

«interface»
IBaseLanguage
createNewProcessor(): AbstractBaseProcessor
getBaseFileExtension(): String
getBinaryFileExtension(): String
getInitEditor(): Path
getInitEditorModuleName(): String
getInitGrammar(): Path
getInitGrammerModuleName(): String
getInitTrans(): Path
getInitTransModuleName(): String
getLanguageName(): String
getPackagedGrammars(): List<Path>
getSugarFileExtension(): String
getVersion(): String
isBaseDecl(IStrategoTerm): boolean
isExtensionDecl(IStrategoTerm): boolean
isImportDecl(IStrategoTerm): boolean
isPlainDecl(IStrategoTerm): boolean
AbstractBaseLanguage
+AbstractBaseLanguage()
+ensureFile(String): Path
+getModelName(IStrategoTerm): String
+getPackagedGrammars(): List<Path>
+getPluginDirectory(): Path
+getTransformationApplication(IStrategoTerm): IStrategoTerm
+getTransformationBody(IStrategoTerm): IStrategoTerm
+getTransformationName(IStrategoTerm): String
+isModelDec(IStrategoTerm): boolean
+isTransformationApplication(IStrategoTerm): boolean
+isTransformationDec(IStrategoTerm): boolean
+isTransformationImportDec(IStrategoTerm): boolean
ScalaLanguage
+getInstance(): ScalaLanguage
+createNewProcessor(): AbstractBaseProcessor
+getBaseFileExtension(): String
+getBinaryFileExtension(): String
+getInitEditor(): Path
+getInitEditorModuleName(): String
+getInitGrammar(): Path
+getInitGrammarModuleName(): String
+getInitTrans(): Path
+getInitTransModuleName(): String
+getLanguageName(): String
+getPackagedGrammars(): List<Path>
+getSugarFileExtension(): String
+getVersion(): String
+isBaseDecl(IStrategoTerm): boolean
+isExtensionDecl(IStrategoTerm): boolean
+isImportDecl(IStrategoTerm): boolean
+isNamespaceDec(IStrategoTerm): boolean
+isPlainDecl(IStrategoTerm): boolean

Figure 3.2.: ScalaLanguage class hierarchy

```
definition
module org/sugarj/languages/SugarScala
imports org/sugarj/languages/Scala
        org/sugarj/languages/Sugar

exports
  context-free syntax
    ScalaPackageDeclarationSemi -> ToplevelDeclaration
    ScalaTopStatSemi            -> ToplevelDeclaration
    ScalaExtension              -> ToplevelDeclaration

    ScalaExtensionHead ScalaExtensionBody -> ScalaExtension {"ScalaExtension"}
    "sugar" ScalaId         -> ScalaExtensionHead {"ScalaExtensionHead"}
    "{" ExtensionElem* "}" -> ScalaExtensionBody {"ScalaExtensionBody"}
```

Listing 3.1: Extension Syntax and ToplevelDeclarations for ScalaLanguage

an arbitrary number of `ExtensionElem` inside of curly braces. `ExtensionElem` is a rule defined in `org/sugarj/languages/Sugar` and can either be Stratego or SDF rules. A more thorough explanation of syntax extensions is given in the case studies in this chapter.

The initializer methods come in pairs and are used to point to initial state files and the respective modules defined in these files. The editor initializer points to a *.serv file, which contains definitions for an Eclipse editor. The grammar initializer points to an *.sdf file, which is supposed to include the sugar module and a SugarJ common module. The trans initializer points to a *.str file, which is supposed to import all initial Stratego definitions for the language to define.

The predicate methods are used by the Sugar* framework to help identify Stratego terms. A base declaration is any top level declaration in the base language, for example class or object declarations in Scala. The predicate should thus return true if the Stratego term is a constructor of such a declaration. An extension declaration is a Stratego term which corresponds to a syntactic extension. For *ScalaLanguage* this is the `ScalaExtension` constructor. An import declaration denotes another entity to be imported and has a special role in Sugar*, as it may also be used to import a syntactic extension into a compilation unit. How this is achieved is explained in the description of the language processor. A plain declaration is used for less structured languages like LaTeX and can be ignored in the context of *ScalaLanguage*.

3.3.2 Processor Interface

The complete type hierarchy for *ScalaProcessor* is given in Figure 3.3. The responsibility of the processor is to process one compilation unit with possible syntactic extensions. For this, it must unparse the AST in form of *IStrategoTerms* to compilable base language source code, tell Sugar* how to call the base language compiler appropriately and also signal dependencies on other modules to the framework. The actual resolving of circular dependencies is handled by the framework and does not need to be taken care of in the processor. The processor is created by the framework with a call to *createNewProcessor* in the corresponding implementation of the language interface.

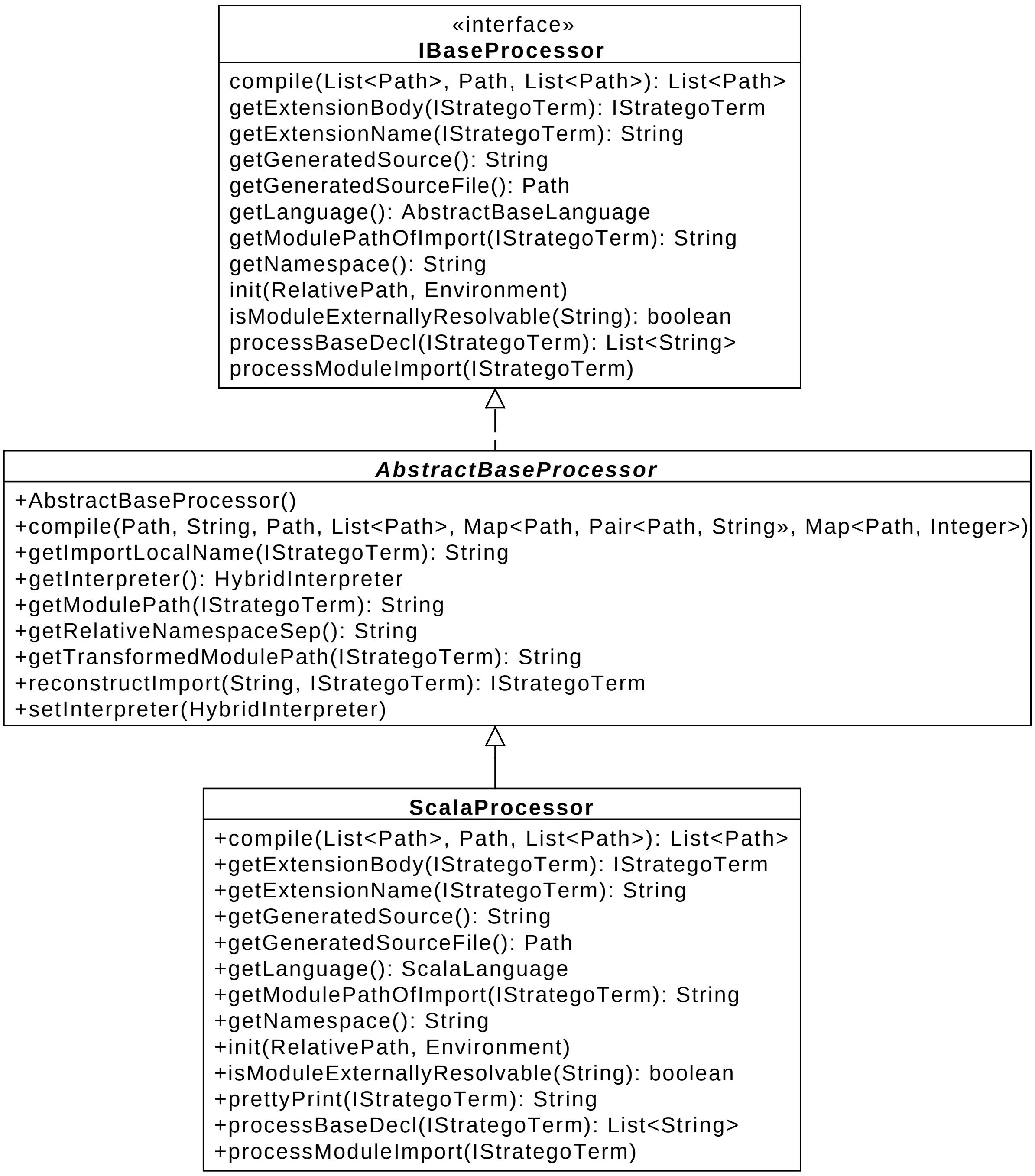

«interface»
IBaseProcessor
compile(List<Path>, Path, List<Path>): List<Path>
getExtensionBody(IStrategoTerm): IStrategoTerm
getExtensionName(IStrategoTerm): String
getGeneratedSource(): String
getGeneratedSourceFile(): Path
getLanguage(): AbstractBaseLanguage
getModulePathOfImport(IStrategoTerm): String
getNamespace(): String
init(RelativePath, Environment)
isModuleExternallyResolvable(String): boolean
processBaseDecl(IStrategoTerm): List<String>
processModuleImport(IStrategoTerm)
AbstractBaseProcessor
+AbstractBaseProcessor()
+compile(Path, String, Path, List<Path>, Map<Path, Pair<Path, String>», Map<Path, Integer>)
+getImportLocalName(IStrategoTerm): String
+getInterpreter(): HybridInterpreter
+getModulePath(IStrategoTerm): String
+getRelativeNamespaceSep(): String
+getTransformedModulePath(IStrategoTerm): String
+reconstructImport(String, IStrategoTerm): IStrategoTerm
+setInterpreter(HybridInterpreter)
ScalaProcessor
+compile(List<Path>, Path, List<Path>): List<Path>
+getExtensionBody(IStrategoTerm): IStrategoTerm
+getExtensionName(IStrategoTerm): String
+getGeneratedSource(): String
+getGeneratedSourceFile(): Path
+getLanguage(): ScalaLanguage
+getModulePathOfImport(IStrategoTerm): String
+getNamespace(): String
+init(RelativePath, Environment)
+isModuleExternallyResolvable(String): boolean
+prettyPrint(IStrategoTerm): String
+processBaseDecl(IStrategoTerm): List<String>
+processModuleImport(IStrategoTerm)

Figure 3.3.: ScalaProcessor class hierarchy

All methods of *IBaseProcessor* are callbacks and called by Sugar* at appropriate times. When to call some of the methods of the processor is determined with the help of the predicates in the corresponding language interface. The first method called after instantiation of a processor is *init*, which informs the processor of the path to the sugared compilation unit source file and the current environment. *ScalaProcessor* uses this information to determine the location of the to-be-created base language source file returned by *getGeneratedSourceFile*, which may be passed to the compiler in later steps.

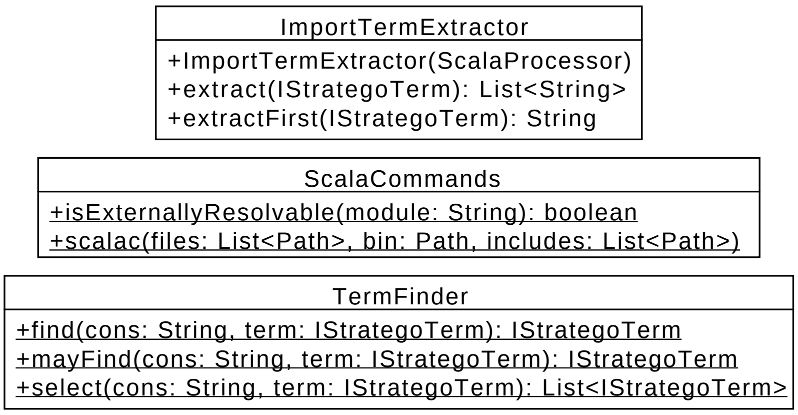

ImportTermExtractor
+ImportTermExtractor(ScalaProcessor)
+extract(IStrategoTerm): List<String>
+extractFirst(IStrategoTerm): String
ScalaCommands
+isExternallyResolvable(module: String): boolean
+scalac(files: List<Path>, bin: Path, includes: List<Path>)
TermFinder
+find(cons: String, term: IStrategoTerm): IStrategoTerm
+mayFind(cons: String, term: IStrategoTerm): IStrategoTerm
+select(cons: String, term: IStrategoTerm): List<IStrategoTerm>

Figure 3.4.: ScalaProcessor Helper Classes

Every identified base declaration is passed to *processBaseDecl*. The processor then needs to unparse this term and append the result to an internal buffer, which can be used at a call to *getGeneratedSource* to build the base source file. The unparsing is achieved with the helper method *prettyPrint*, which makes use of an *ATermCommands* helper class provided by the Sugar* framework. This helper class allows to parse pretty printing tables in the Box language format and then apply these pretty printing rules to Stratego terms. The pretty printing table for the Scala grammar is generated by Spoofax and included in the plug-in as `Scala.pp`.

The *processBaseDecl* method is also used to query and update context information from the passed terms. *ScalaProcessor* for example further extract namespace information from encountered package declaration terms, which can then be returned by *getNamespace*. The *processModuleImport* method is intended for the same unparsing and processing purpose, but it is not called everytime an import is encountered. The reason for this is, that import statements, resolving to syntactic extensions, should not show up in base source files. The compiler of the base language would not be able to resolve the import and fail to compile the desugared source files.

Syntactic extensions are handled by the processor with the methods *getExtensionBody* and *getExtensionName*, which simply extract body and name from an extension term, respectivelly. The actual interpretation of the extension definition and the adaption of the parser is handled by Sugar*. If the processed compilation unit only consists of imports or syntactic extension, the processor must take care to return the empty string on a call to *getGeneratedSource*. This is the signal for the framework to consider the compilation unit to be a pure syntactic extension and avoid a call to *processModuleImport* on other compilation units including the currently processed unit.

Another callback concerning modules is *isModuleExternallyResolvable*. On encountering an import declaration the framework tries to resolve the respective module in the active projects. If it fails to find a corresponding file, it uses the callback to determine if the module can be resolved outside from the project context. This is for example the case for the Scala standard library, which is in the classpath of the project, but can not be resolved as source file.

ScalaProcessor makes use of three helper classes: *ScalaCommands*, *ImportTermExtractor* and *TermFinder*. The contained methods and signatures are given in Figure 3.4. The *isModuleExternallyResolvable* method from *ScalaProcessor* is forwarded to the helper method *isExternallyResolvable*

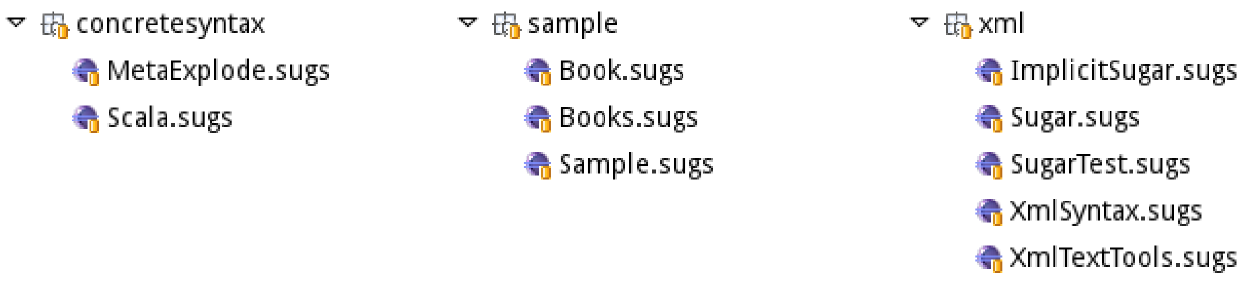


Figure 3.5.: Project Layout for the XML Syntax Extension

from *ScalaCommands*. This helper method determines the resolvability of a module by creating a source file only containing an import statement corresponding to the module in question. This file is then passed to the compiler with the current classpath and the result is checked against success or failure.

The *scalac* method creates a command line call from the passed arguments to the *scalac* script included in the standard Scala distribution. The *files* argument is the list of files to compile, *bin* is the path to the output directory and *include* is the classpath to use. The expected return value is a list of files which were actually created by the call. This list of files is obtained by matching the verbose output of the compiler against the regular expression `^\[wrote '[^']*' to (.*)\]$`. The implementation of *ScalaProcessor.compile* simply forwards to the *scalac* method.

The helper classes *ImportTermExtractor* and *TermFinder* are used to implement the *getModulePathOfImport* method. This method expects a Stratego term representing any kind of scala import constructor and returns the expected path to the module to import. As Scala has a variety of possible import terms the *TermFinder* is used to find subterms representing module names in the given term and the *ImportTermExtractor* then extracts the fully qualified name out of the term.

3.4 Case Study: XML

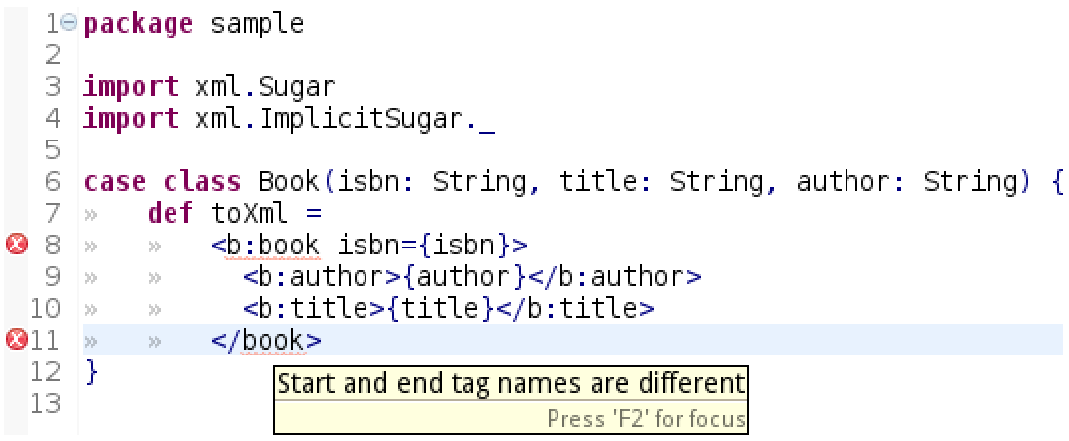
The Scala grammar created in this work does intentionally not have support for embedded XML expressions – opposed to native Scala code. This first case study shows how support for embedded XML expressions in Scala can be achieved with SugarScala. The Eclipse project layout of the XML extension is given in Figure 3.5.

The project is split into three packages: `concretesyntax`, `sample` and `xml`. The `concretesyntax` package contains definitions for easier handling of the actual syntactic extensions. The `sample` package contains example uses of the syntactic extension. Inside the `xml` package are the actual syntactic extension and the necessary term rewritings.

To give a short recap on XML the basic grammar for an XML element is given in Listing 3.2. An element is either in the open form, having a start and end tag, or in the closed form, only consisting of one tag. A tag always begins with a less-than-sign and ends with a greater-than-sign. An end tag has a slash after the less-than-sign and a closed tag has a slash before the greater-than-sign. Start and closed tags can have an arbitrary number of attributes after the tag name. An attribute consists of an attribute name, followed by an equals-sign and a value surrounded

```
elem      = sTag, {elem | ? CDATA ?}, eTag
          | emptyElem
sTag      = "<", name, {attr}, ">"
eTag      = "</", name, ">"
name      = [ns], id
emptyElem = "<", name, {attr} "/>"
ns        = id, ":"
attr      = name, "=", attrVal
attrVal   = '"', ? val ?, '"'
```

Listing 3.2: Sketched out XML element syntax



```
package sample

import xml.Sugar
import xml.ImplicitSugar._

case class Book(isbn: String, title: String, author: String) {
	def toXml =
		<b:book isbn={isbn}>
		  <b:author>{author}</b:author>
		  <b:title>{title}</b:title>
		</book>
}

```

Start and end tag names are different
Press 'F2' for focus

Figure 3.6.: Eclipse editor using the XML extension, showing a modified sample/Book.sugs

by quotation marks. Every name can have a namespace, which is an id prepended before the name, followed by a colon. In between start and end tags can be an arbitrary number of nested elements or character data.

A showcase of the features of the XML extension is given in Figure 3.6 which shows an Eclipse editor using the XML extension. The sample file consists of a package declaration, imports for the XML extension and a *Book* case class with a *toXml* method. As one can see the XML syntax is used in a position where a Scala expression is expected. Furthermore the attribute value for the ISBN and the contents of the author and title elements are replaced by Scala blocks. Additionally the mismatch of the start and end tag names is detected and marked as an error.

The desugared code for the sample file in Figure 3.6 is given in Listing 3.3. XML elements with contents are expressed by the *scala.xml.Elem* class. The signature for the apply method of the companion object used for the desugaring is:

```
def apply(prefix: String, label: String, attributes: MetaData,
          scope: NamespaceBinding, minimizeEmpty: Boolean, child: Node*): Elem
```

The prefix is the namespace part of the tag name. The label is simply the tag name. The list of attributes is not represented by the *List* type in Scala, but is rather of type *scala.xml.MetaData*. *MetaData* is a common supertype of unprefixed and prefixed attributes, which are represented by two different classes, *scala.xml.UnprefixedAttribute* and *scala.xml.PrefixedAttribute*, respectively. The parameters for the *UnprefixedAttribute* constructor are the name of the attribute, the value of the attribute and the remaining attributes, again represented as *MetaData*, in that order.

```
package sample

import xml.ImplicitSugar._

case class Book(isbn: String, title: String, author: String) {
  def toXml =
    scala.xml.Elem("b", "book",
      new scala.xml.UnprefixedAttribute("isbn", { isbn }, scala.xml.Null),
      scala.xml.TopScope,
      false,
      scala.xml.Elem("b", "author", scala.xml.Null,
                     scala.xml.TopScope, false, { author }),
      scala.xml.Elem("b", "title", scala.xml.Null,
                     scala.xml.TopScope, false, { title }))
}
```

Listing 3.3: Desugared Code for Book Example

```
1  package concretesyntax
2
3  import org.sugarj.languages.Scala
4  import org.sugarj.languages.Stratego
5
6  import concretesyntax.MetaExplode
7
8  sugar Scala {
9    context-free syntax
10     "|[" ScalaExpr "]|"     -> StrategoTerm {cons("ToMetaExpr")}
11     ":${" StrategoTerm "}" -> ScalaExpr    {cons("FromMetaExpr")}
12     ":${" StrategoTerm "}" -> ScalaNoLExpr {cons("FromMetaExpr")}
13 }
```

Listing 3.4: concretesyntax/Scala.sugs

The empty list of *MetaData* is represented by the class *scala.xml.Null*. The *NamespaceBinding* represents the namespace the element is in, which could be introduced by a special *xmlns* attribute. But as there is none all the elements are part of the toplevel namespace scope which is represented by the *scala.xml.TopScope* object. The *minimizeEmpty* flag can be set to signal that the element may be printed as an empty element, as long as it does not have any nested children. The list of children for an element is given as the *child* variable arguments parameter.

The contents of `concretesyntax/Scala.sugs` is given in Listing 3.4. Lines 1-6 are regular Scala syntax, but none of the imports refers to valid Scala types. Line 3 imports the Scala grammar SDF rules, including `ScalaExpr` and `ScalaNoLExpr`. Line 4 imports the Stratego grammar rules, including `StrategoTerm`. Line 6 imports the local `concretesyntax/MetaExplode.sugs`, which is taken from the StrategoXT repository and contains Stratego rules matching on *ToMetaExpr* and *FromMetaExpr* constructors. Lines 8-13 describe a syntactic extension, as defined in Listing 3.1.

The syntactic extension consists of new SDF rules which simplify the writing of desugarings on the Scala SDF AST, as they define the concrete syntax of Scala [4]. The first rule in line 10 allows to use a Scala expression as a `StrategoTerm`, which will be transparently transformed by the rules from `MetaExplode.sugs` to the corresponding AST. The rules defined in lines 11 and 12 describe the opposite direction and allow a `StrategoTerm` to be transparently transformed to a Scala expression.

The core syntactic extension for XML is described in `xml/Sugar.sugs` and given in Listings 3.5 and 3.6. It makes use of Scala and Stratego syntax elements, as well as concrete syntax and thus imports `concretesyntax.Scala`. Another important aspect of the XML syntax extension is actual XML syntax and corresponding rewriting rules, which are imported from `xml.XmlSyntax` and `xml.XmlTextTools`. Both of these files are copied from the Stratego repository and did not require any modification. The actual syntax extension is given from line 8 to 18 and is only a small part of the whole extension. The rest of the file describes the tree rewritings to achieve the desugaring to base Scala syntax.

The changes to existing lexical rules are the rejection of XML commentary start and end brackets as Scala operators. Additionally they are added to the keywords as they also resemble perfectly fine identifiers with which they should not be ambiguous. The changes to context-free syntax enable the interweaving of Scala syntax with XML syntax. `Element` and `AttValue` are the imported SDF rules for XML elements and attribute values, respectively. Lines 15 and 16 allow an XML element to appear everywhere, where a Scala expression could be. Lines 17 and 18 allow to use Scala blocks, including the curly braces, instead of elements or attribute values.

The Stratego rules declared as desugarings are applied innermost on the parsed AST, until no more changes can be made to the tree. So *desugar-element*, *desugar-empty-element* and *desugar-text* are entry points for the desugaring, whereas *desugar-attrs*, *desugar-prefix* and *desugar-attrval* are helper rules. Strategy *scala-multiline-quote* is a helper strategy to wrap Strings without quotes into Scala multiline strings. The rule *constraint-error* is special, as it is used by Sugar* to highlight errors in the Eclipse editor.

The rule *desugar-element* matches against the *Element* constructor. This constructor takes four parameters: the name of the start tag, a list of attributes, a list of children and the name of the end tag. A tag name in the imported XML syntax always is a *QName* constructor, nested in an *ElemName* constructor. The *QName* constructor takes two parameters: a namespace prefix and the actual name. The *desugar-element* rule is defined twice – one matches against *Element* without children, the other matches against *Element* with children.

The first case for *Element* without children is the simpler of both and can make use of the concrete syntax defined in Listing 3.4. As one can see by comparing Figure 3.6 and Listing 3.3, an *Element* needs to be desugared to an application of the *scala.xml.Element* object. Because of the concrete syntax the application can be expressed in regular Scala syntax. As the arguments need further desugaring, the concrete syntax is again used to fall back to Stratego expressions, except for the *TopScope* and **false** arguments.

The prefix is optional in the grammar and is thus either *None()* or *Some(Prefix(p))*. The helper rule *desugar-prefix* has two definitions for these two cases. *None* is desugared to the Scala **null**, which is represented by the *Null()* constructor in the AST. The *p* of prefix is a Stratego string. If wrapped in double quotes, it denotes a Scala string in the AST.

The *attrs* pattern variable binds against a possibly empty list of *Attribute* constructors. The *Attribute* constructors takes two parameters: an attribute name in form of *AttrName* constructors and a value. The *AttrName* always has an already known *QName* as only child. The desugaring of the attributes list is achieved with the *desugar-attrs* rule, which distinguishes three cases:

```
package xml

import concretesyntax.Scala
import xml.XmlSyntax
import xml.XmlTextTools

sugar Sugar {
  lexical syntax
    "<!--"  -> SCALA-KEYWORD
    "<!--"  -> SCALA-BRACKET-OP {reject}
    "-->"   -> SCALA-KEYWORD
    "-->"   -> SCALA-SUM-OP {reject}

  context-free syntax
    Element -> ScalaExpr {prefer}
    Element -> ScalaNoLExpr {prefer}
    ScalaBlockExpr -> Element {prefer}
    ScalaBlockExpr -> AttValue

  desugarings
    desugar-element
    desugar-empty-element
    desugar-text

  strategies
    scala-multiline-quote = double-quote; double-quote; double-quote

  rules
    desugar-element:
      Element(ElemName(QName(prefix, name)), attrs, [], _) ->
        |[ scala.xml.Elem(
          :${<desugar-prefix> prefix},
          :${quotedName},
          :${<desugar-attrs> attrs},
          scala.xml.TopScope,
          false) ]|
        where quotedName := <quote(!'"')> name

    desugar-element:
      Element(ElemName(QName(prefix, name)), attrs, children, _) ->
        AppExpr(
          Path(["scala", "xml", "Elem"])
        , ArgumentExprs(
            Some(
              Exprs(
                <flatten-list>
                [ <desugar-prefix> prefix
                , <double-quote> name
                , <desugar-attrs> attrs
                , Path(["scala", "xml", "TopScope"])
                , False()
                , children ]))))
```
Listing 3.5: xml/Sugar.sugs

```
    desugar-empty-element:
      EmptyElement(ElemName(QName(prefix, name)), attrs) ->
        |[ scala.xml.Elem(
          :${<desugar-prefix> prefix},
          :${quotedName},
          :${<desugar-attrs> attrs},
          scala.xml.TopScope,
          true) ]|
        where quotedName := <quote(!'"')> name

    desugar-text:
      Text(chardata) ->
        |[ scala.xml.Text(:${str}) ]|
        where str := <scala-multiline-quote> <chardata2string> Text(chardata)

    constraint-error:
      Element(sname, _, _, ename) ->
        [(sname, "Start and end tag names are different"),
         (ename, "Start and end tag names are different")]
      where <not(structurally-equal)> (sname, ename)

    desugar-attrs:
      [] -> |[ scala.xml.Null ]|

    desugar-attrs:
      [Attribute(AttrName(QName(None(), name)), attrval) | rst] ->
        |[ new scala.xml.UnprefixedAttribute(
          :${<double-quote> name},
          :${<desugar-attrval> attrval},
          :${<desugar-attrs> rst}) ]|

    desugar-attrs:
      [Attribute(AttrName(QName(Some(Prefix(prefix)), name)), attrval) | rst] ->
        |[ new scala.xml.PrefixedAttribute(
          :${<double-quote> prefix},
          :${<double-quote> name},
          :${<desugar-attrval> attrval},
          :${<desugar-attrs> rst}) ]|

    desugar-prefix:
      None() -> Null()

    desugar-prefix:
      Some(Prefix(p)) -> <double-quote> p

    desugar-attrval:
      BlockExpr(blk) -> BlockExpr(blk)

    desugar-attrval:
      DoubleQuoted(c) -> <double-quote> <xml-attr-value2string> DoubleQuoted(c)
}
```

Listing 3.6: xml/Sugar.sugs cont.

1. The list is empty. This case desugars to the Scala object *scala.xml.Null*.
2. The head of the list is an attribute without a prefixed name. This case desugars to a new initialization of a *scala.xml.UnprefixedAttribute*.
3. The head of the list is an attribute having a prefixed name. This case desugars to a new initialization of a *scala.xml.PrefixedAttribute*.

All cases make use of concrete syntax to avoid writing the AST directly. The name and a possible prefix just need to be turned to strings by wrapping them in double quotes. The attribute value is handled by the helper rule *desugar-attrval*. The rest of the attribute list is recursively handled by the *desugar-attrs* rule.

The attribute values can either be Scala block expressions, denoted by *BlockExpr*, or valid XML values wrapped in double quotes, which are represented by a *DoubleQuoted* constructor. The *BlockExpr* does not need any desugaring and is thus just returned. The *DoubleQuoted* is processed by the imported helper rule *xml-attr-value2string*, which turns the value to a Stratego string and is then wrapped with *double-quote* to be a Scala string.

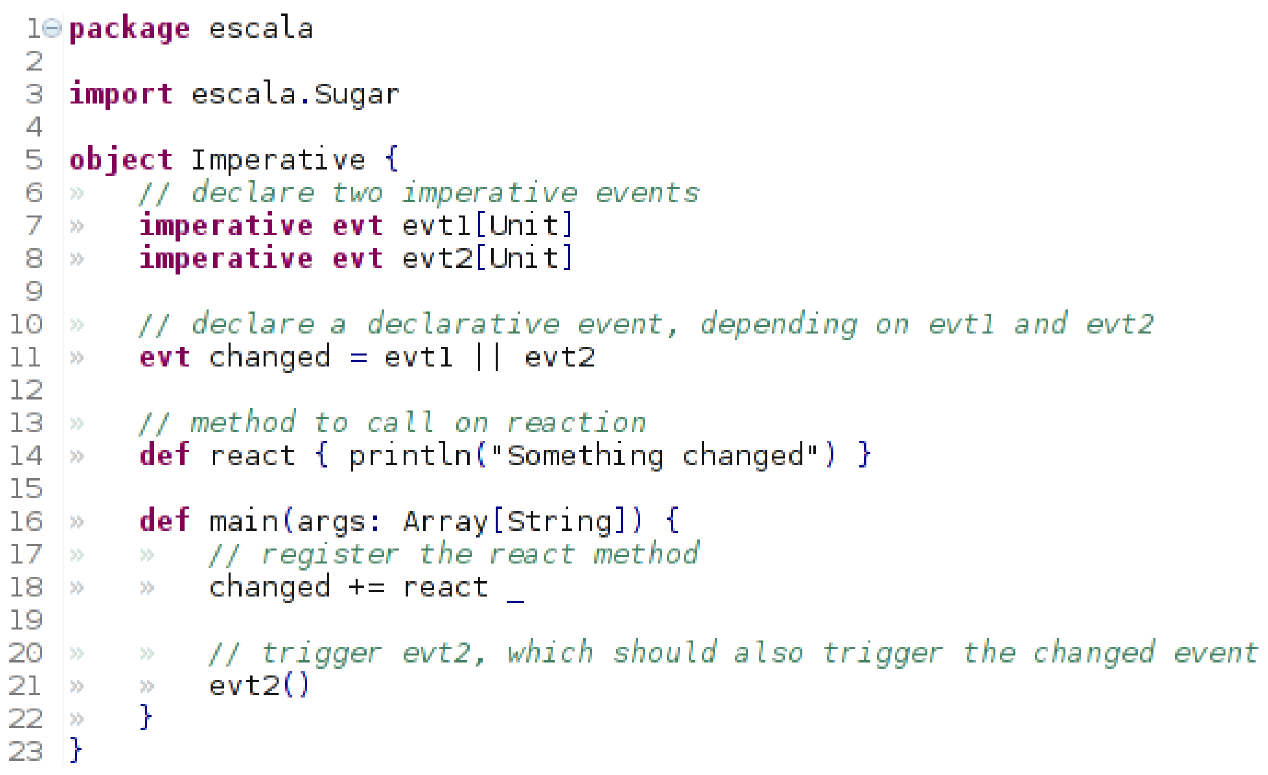
The case with children for *desugar-element* can not make use of the concrete syntax. This is because the number of children is arbitrary and must be appended to the list of expressions inside of the argument expression. For this purpose the AST is written down directly, as it allows to make use of the *flatten-list* rule to flatten the list of expressions. This way the list of the form [prefix, name, attrs, scope, flag, [c1, c2, ...]] is transformed to [prefix, name, attrs, scope, flag, c1, c2, ...], which is the desired effect.

The rule *desugar-empty-element* is defined analogously to *desugar-element* in the case without children. The only difference is, that the rules matches against the *EmptyElement* constructor, which does not even have a parameter for children. Additionally the *minimizeEmpty* flag is set to true, as empty elements should be rendered as such and not in the open element form.

The rule *desugar-text* matches against *Text* constructors, which represent char data and can also be used as children to open elements. Text elements in Scala are represented by the *scala.xml.Text* class and can be easily instantiated by the companion object from simple strings. As these strings may contain linebreaks, the rule makes use of the imported *chardata2string* rule to transform the text into a Stratego string and then applies the *scala-multiline-quote* rule to wrap the String into a Scala multiline string form.

The *constraint-error* rule is used by the Spoofax framework for static analysis and marking of errors in the editor. The rule matches on *Element* constructors in which the term structure of the start tag name and the end tag name are not equal. This is only true if the start tag name and end tag name are not the same. In this case the element is transformed into a list of pairs. The first argument of the pair is the term to mark, and the second argument of the pair is the error message to attach to the mark.

This XML case study shows how Scala can be syntactically extended with a domain specific language. It does not make any use of the XML parsing capability built into the Scala compiler, but still appears to be seamlessly integrated into the language. XML elements can be used instead of expressions and Scala blocks can be used as an interpolation mechanism for nested elements or attributes. Additionally static analysis can be used to detect mismatching opening and closing tags.



```
 1 package escala
 2 
 3 import escala.Sugar
 4 
 5 object Imperative {
 6     // declare two imperative events
 7     imperative evt evt1[Unit]
 8     imperative evt evt2[Unit]
 9 
10     // declare a declarative event, depending on evt1 and evt2
11     evt changed = evt1 || evt2
12 
13     // method to call on reaction
14     def react { println("Something changed") }
15 
16     def main(args: Array[String]) {
17         // register the react method
18         changed += react _
19 
20         // trigger evt2, which should also trigger the changed event
21         evt2()
22     }
23 }
```

Figure 3.7.: Editor showing an example EScala file

The extension describes the entry points by providing additional productions and constructors for already existing sorts of the Scala grammar. These additional productions are again only composed of imported sorts from the XML grammar which could be reused without any modification. The Stratego language is used to transform the newly introduced nodes from the XML grammar to nodes of the Scala grammar. As the unparsing rules for the Scala grammar are provided as part of SugarScala, the Scala syntax tree is unparsed to valid Scala syntax which can then be passed to the Scala compiler for the actual compilation.

3.5 Case Study: EScala

EScala is an extension to Scala which allows declarative object-oriented events [16]. It is originally developed as a modification to the Scala Compiler v2.9.0. Unfortunately this modification is not compatible with the current Scala compiler v2.10.3. Large parts of the functionality of EScala are however implemented in a Scala library which can be used apart from the compiler modification. This case study shows how SugarScala allows to use the functionality of the EScala library with the familiar syntax from the compiler modification.

Figure 3.7 shows an example EScala file, opened in an Eclipse editor with SugarScala enabled. The import in line 3 is needed to actually activate the EScala syntactic extension. The syntax extensions used are the two new template statements **imperative evt** and **evt**. The **imperative evt** statement, as used in line 7, declares a new imperative event with an identifier and a type parameter clause. The type parameter clause defines the type of arguments the event expects to be triggered with. A *Unit* type parameter defines an event which does not take any arguments.

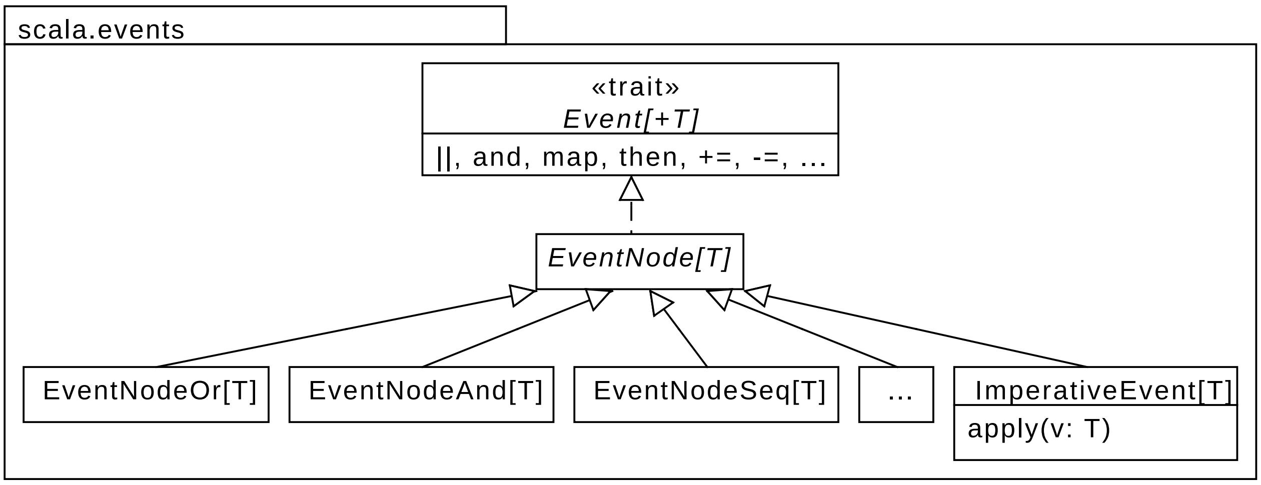


Figure 3.8.: Type hierarchy for EScala events

```scala
package escala

object Imperative {
  import scala.events.EventsLibConversions._

  lazy val evt1 = new scala.events.ImperativeEvent[Unit]
  lazy val evt2 = new scala.events.ImperativeEvent[Unit]

  lazy val changed = evt1 || evt2

  def react { println ( "Something changed" ) }

  def main (args: Array[String]) {
    changed += react _
    evt2()
  }
}
```

Listing 3.7: Desugared code for Figure 3.7

Imperative events can be triggered in the code in a similar way to applying functions, as can be seen in line 21. The **evt** statement in line 11 is used for declarative events, which are defined on behalf of already existing events. For this purpose the statement takes arbitary expressions on its right hand side, which can combine already existing events with operators. Apart from combination operators, events also have operators to register or unregister event handlers. Line 18 registers the *react* function on the declarative *changed* event, which is then called each time when *evt1* or *evt2* are triggered.

The type hierarchy for events in EScala is sketched out in Figure 3.8. *Event* is the base trait of the events and *EventNode* a common abstract implementation. All events thus have common operators defined which can be used for combination and reaction purposes. Additionally all types are generalized over *T*, which is the argument type of the event. The different operations yield a corresponding event type with the appropriate handling logic. Only imperative events can be triggered (applied) directly.

```
 1 package escala
 2
 3 import escala.Sugar
 4
 5 class Figure {
 6   // declare event on observable method
 7   evt moved = afterExec(moveBy)
 8
 9   // define observable method
10   observable def moveBy(dx: Int, dy: Int) = {
11     // moving...
12   }
13
14   // method to call on moved event
15   def afterMoved(args: (Int, Int), ret: Unit) = println("Moving...")
16
17   // register method with moved event
18   moved += afterMoved _
19 }
```

Listing 3.8: EScala observable method example

Listing 3.7 shows the desugared code for Figure 3.7. The import for `escala.Sugar` does not show up in the desugared code, as it refers to a syntactic extension and not a valid Scala module. But instead a new import statement is included at top of the template of *Imperative*, which imports implicit conversions for the EScala library. These are used to implicitly convert the events to appropriate types and thus provide a nicer API. The **imperative evt** statements are desugared to lazy value definitions of instantiations of *scala.events.ImperativeEvent*. The declarative **evt** statement is simply desugared to a lazy value definition with the right-hand-side reused. The `||` is definied as unary method in *scala.events.ImperativeEvent* can thus be used as infix operator.

Apart from imperative and declarative events, EScala also has support for observable methods. An example use is given in Listing 3.8. The *moved* event is declared by a new *afterExec* expression in line 7. The corresponding observable method *moveBy* is defined in line 10 and marked with the new **observable** modifier. The reaction function *afterMoved* is defined in line 15 and registered in line 18. Reaction functions to observable methods always require two arguments: the first is the arguments for the observed method and the second is the return value of the observed method. Multiple arguments are expressed with tuple types.

The desugared code for the observable method example is given in Listing 3.9. Again the import of `escala.Sugar` does no longer appear, but the implicits are imported on top of the template for *Figure*. The *afterExec* expression is desugared to a simple select of *after* on the argument, as can be seen in line 6. The desugaring of observable methods is twofold:

1. The method signature and definition are taken as they are, but the name is synthesized (line 8). The synthetization is leaned on the way Scala handles synthetic names, by making use of dollar signs and a descriptive label.

2. A new lazy value with the original name is defined. It is bound to an application of *scala.events.Observable[T, U]* taking the method as argument (line 10). The *Observable* wraps around the method definition and has the val *after* of type *ImperativeEvent* as member,

```
package escala

class Figure {
  import scala.events.EventsLibConversions._

  lazy val moved = moveBy.after

  def $escala_syn$moveBy(dx: Int, dy: Int) = {}

  lazy val moveBy = scala.events.Observable($escala_syn$moveBy)

  def afterMoved(args: (Int, Int), ret: Unit) = println("Moving...")

  moved += afterMoved _
}
```

Listing 3.9: Desugared code for Listing 3.8

among others. Additionally *Observable[T, U]* extends $(T) \Rightarrow U$, so together with the imported implicits it can be handled like any other mod from a usage point of view.

The EScala syntactic extension is defined in `escala/Sugar.sugs` and is given in Listings 3.10 and 3.11. Lines 6 to 24 describe the new syntax. The imperative and declarative events, as well as the observable method are defined as new template statements. Each production has its own node constructor, so they can be easily distinguished and desugared later on. The *afterExec* and *beforeExec* expressions are added as new `ScalaExprs`, again with their own respective constructors. For simplicity reasons the syntax inside the parentheses is expected to be a `ScalaId`, but it could as well be an arbitary `ScalaNoLExpr`. In the same manner, *afterExec* and *beforeExec* can not be used as `ScalaNoLExpr`, inside parentheses, with this extension. The new keywords are added as such to the lexical syntax so they can not be ambiguous with identifiers.

The desugaring for EScala is conceptually a bit different from the desugaring for XML. The reason for this is the need to conditionally add the import for the implicits to templates containing EScala nodes. The Stratego rules declared as desugarings are applied innermost, as stated before. But they are not applied innermost one after another in the order they are declared, but rather is each of the rules tried at each possible position in the tree. So if the desugaring rules were all declared as desugarings, the EScala nodes would all have been already desugared at the point, where the desugaring process encounters the first template. This way it would be impossible to detect the use of EScala nodes on the template level. To encounter this circumstance the EScala syntactic extension defines only one rule as entry point for the desugaring, *desugar-escala*.

The rule *desugar-escala*, defined in line 30, makes use of the implicit conditionals used in the Stratego language. So first it tries to apply the rule *desugar-escala-implicits-import*, and only if that succeeds, it will use the rule *desugar-escala-nodes* innermost from the current position. But the rule *desugar-escala-implicits-import* will only match on *TemplateBody* applications. So if this rule does not match, then *desugar-escala* will fail on the current node. This way it is possible to prepend the implicits import in the template but still apply the node desugarings innermost.

Apart from only matching *TemplateBody* applications, the *desugar-escala-implicits-import* defined in line 34 has two additional requirements:

```
sugar Sugar {
  context-free syntax
    "imperative" "evt" ScalaId ScalaTypeParamClause
      -> ScalaTemplateStat {cons("ImperativeEvt")}

    "evt" ScalaId ScalaTypeParamClause? "=" ScalaExpr
      -> ScalaTemplateStat {cons("Evt")}

    "observable" "def" ScalaFunDef
      -> ScalaTemplateStat {cons("ObservableDef")}

    "afterExec"  "(" ScalaId ")" -> ScalaExpr {cons("AfterExec")}
    "beforeExec" "(" ScalaId ")" -> ScalaExpr {cons("BeforeExec")}

  lexical syntax
    "afterExec"  -> SCALA-KEYWORD
    "beforeExec" -> SCALA-KEYWORD
    "observable" -> SCALA-KEYWORD
    "evt"        -> SCALA-KEYWORD
    "imperative" -> SCALA-KEYWORD

  desugarings
    desugar-escala

  rules
    desugar-escala =
      desugar-escala-implicits-import
      ; innermost(desugar-escala-nodes)

    desugar-escala-implicits-import:
      TemplateBody([|body]) -> TemplateBody([import|body])
      where import := |[ import scala.events.EventsLibConversions._; ]|
        ; <not(elem)> (import, body)
        ; <uses-escala> body

    uses-escala =
      oncetd(
        ?AfterExec(_)
        + ?BeforeExec(_)
        + ?ImperativeEvt(_, _)
        + ?Evt(_, _, _)
        + ?ObservableDef(_))

    escala-syn = ?name; <strcat>("$escala_syn$", name)

    desugar-escala-nodes =
      desugar-after-exec
      <+ desugar-before-exec
      <+ desugar-imperative-evt
      <+ desugar-evt
```

Listing 3.10: EScala Syntactic Extension

```

    desugar-after-exec:
      AfterExec(name) -> DesignatorExpr(name, Id("after"))

    desugar-before-exec:
      BeforeExec(name) -> DesignatorExpr(name, Id("before"))

    desugar-imperative-evt:
      ImperativeEvt(name, type) ->
        |[ lazy val :${name} = new scala.events.ImperativeEvent :${type} ]|

    desugar-evt:
      Evt(name, type, expr) -> |[ lazy val :${name} = :${expr} ]|

    desugar-observable-def:
      [obs_def|rst] ->
        <flatten-list> <map(desugar-observable-def-proc <+ id)> [obs_def|rst]
      where
        <oncetd(?ObservableDef(_))> [obs_def|rst]

    desugar-observable-def-proc:
      TemplateStatSemi(
        ObservableDef(
          ProcDef(
            FunSig(Id(name), typeParamClause, paramClause),
            blk)),
        semi)
      ->
      [TemplateStatSemi(
        FunDefDef(
          ProcDef(
            FunSig(
              Id(<escala-syn> name),
              typeParamClause,
              paramClause),
            blk)),
        semi)
      ,TemplateStatSemi(
        |[ lazy val :${name} = scala.events.Observable(:${<escala-syn> name}) ]|,
```

Listing 3.11: EScala Syntactic Extension cont.

```
sugar Scala {
  context-free syntax
    "|[" ScalaTemplateStat "]|"     -> StrategoTerm {cons("ToMetaExpr")}
    "|[" ScalaTemplateStatSemi "]|" -> StrategoTerm {cons("ToMetaExpr")}
    ":${" StrategoTerm "}" -> ScalaExpr            {cons("FromMetaExpr")}
    ":${" StrategoTerm "}" -> ScalaNoLExpr         {cons("FromMetaExpr")}
    ":${" StrategoTerm "}" -> ScalaTemplateStat    {cons("FromMetaExpr")}
    ":${" StrategoTerm "}" -> ScalaId              {cons("FromMetaExpr")}
    ":${" StrategoTerm "}" -> ScalaTypeParamClause {cons("FromMetaExpr")}
}
```

Listing 3.12: Concrete syntax for EScala – concretesyntax/Scala.sugs

1. The import statement must not already be in the template body.
2. The body must contain at least one EScala node.

The former is checked with the predefined *elem* rule on the body. The latter uses a matching-only helper rule *uses-escala*. Also noteworthy is the actual definition of the import statement with the help of concrete syntax. The semicolon makes sure that the import statement can be unambiguously parsed as a terminated statement. So where the XML concrete syntax only defines Scala expressions, the EScala concrete syntax also makes use of template statements and other Scala rules. The adapted `concretesyntax/Scala.sugs` is given in Listing 3.12 and is defined analogous to Listing 3.4.

The helper rule *uses-escala* matches against any of the defined EScala applications *AfterExec*, *BeforeExec*, *ImperativeEvt*, *Evt* or *ObservableDef* with the *oncetd* strategy. This strategy traverses the AST in a top-down fashion and succeeds if the given rule can be successfully applied at least once. So the rule *uses-escala* succeeds, if any of the EScala nodes is used in the subtree of the current node.

The rule *desugar-escala-nodes* simply tries each of the rules *desugar-after-exec*, *desugar-before-exec*, *desugar-imperative-evt*, *desugar-evt* and *desugar-observable-def* from left to right. All of the mentioned rules, except for the last rule *desugar-observable-def*, are straight-forward translations. Rule *desugar-after-exec* desugars the node *AfterExec(name)* to a *DesignatorExpr* with *name* as the left-hand-side and the id "after" as the right-hand-side. The rule *desugar-before-exec* is defined analogous. Rules *desugar-imperative-evt* and *desugar-evt* desugar the respective application to a **lazy val** definition, as already described. With the help of the concrete syntax the rules can be expressed intuitively.

The desugaring of *ObservableDef* applications is achieved with the help of the *desugar-observable-def* rule. As in this case one statement must be desugared into two statements, the rule matches on lists of statements. If the list contains an *ObservableDef* application, then the helper rules *desugar-observable-def-proc* are mapped over the list. These are two rules with the same name, but different match criteria. One of these rules matches on *ProcDef* statements, the other on *FunDef* statements (omitted in the listing). As the map should not fail – and with it the whole rule – on non-*ObservableDef* applications, the alternative to the *desugar-observable-def-proc* is the never-failing *id*. If one of the rules matches on a statement in the list, it will be transformed into a list of two statements: The first with the respective **def**, but a synthetic name; the second with a lazy value definition having the original name, and being bound to *scala.events.Observable* wrapped around the synthetic name, as explained before. As the desired result is not a nested list, the predefined *flatten* rule is used to assure a flat list. The synthetization of the name is achieved with the *escala-syn* rule, which simply prepends the string "$escala_syn$" on the name.

The EScala case study shows that SugarScala allows to syntactically integrate Scala language extensions without much effort – supposed these extensions come in form of Scala libraries. The respective invocations of these libraries can be hidden with syntactic sugar and let them appear to be almost seamlessly integrated into the base language. This syntactic sugar can be defined anywhere in the Scala grammar, not only at expression or application position, as can be seen with the **evt** or **observable** statements. If the language library further provides smart *implicit* declarations, these can be transparently imported to avoid providing otherwise necessary type information. Implicits are a large benefit for SugarScala given that desugarings can not access the type system of the base language and thus can not be defined type-dependent or derive necessary type information.

SugarScala however fails to replace the EScala compiler extension to full extent. The original EScala does not require methods to be marked with **observable** to use them in **afterExec** and **beforeExec** expressions in the same class. The use of **observable** would only be necessary to allow methods to be observable between different types. But SugarScala needs a syntactical distinction between regular methods and observable methods to know which need to be rewritten. The introduction of new syntax allows to introduce a new node in the AST, which can then be easily detected and transformed. Otherwise the necessary static analysis to find the referenced method would need to be reimplemented in form of complicated Stratego rules.

Another shortcoming is the already mentioned inability to access or modify the type system. The **observable** seems to be a modifier for method definitions but it is in fact a new statement which only resembles method definitions. A user unaware of the extension may assume that the **observable def** is really a method definition which can be overridden in subclasses, as other method definitions can. This however will not work as the desugaring will transform the statement into a **lazy val** definition of type *Observable* with the same name. The EScala compiler extension additionally allows to override events in subclasses and reference the parent event with *super*. This semantical change of *super* requires access to the type system and can thus simply not be expressed with SugarScala.

3.6 Discussion

The XML and EScala case studies show how SugarScala allows to extend regular Scala with support for domain specific languages on the syntactical level: The SDF Scala grammar is extended by adding new arbitrary productions for existing sorts. With the new productions the parser may then create new constructor applications in the resulting AST. The AST with the new applications must be transformed with the help of Stratego to an AST only consisting of applications from the original grammar. This task can be simplified with the help of concrete syntax. The desugared AST can then finally be unparsed and the resulting source code is passed to the Scala compiler.

This is a purely syntactical approach because it is not possible to define desugarings conditional on information derived from static analysis. It may be possible to handle type information with Stratego in SugarScala as can be seen with SugarFomega [21], but we did not investigate in this direction due to the complexity of the Scala language. What however is possible is the definition of simple syntactic analysis as can be seen with the detection of mismatching start and end tags for the XML case study.

The lack of static analysis is a drawback for usability of SugarScala. Compilation errors are not indicated in the opened Eclipse editor, but are printed by the Scala compiler on the Sugar* console. As the Scala compiler only sees the unparsed AST the user might be confronted with error messages referring to code he has not written himself but is the result of desugarings.

The error reporting and recovery in general have room for improvement, as well as the usability. Syntax errors are simply reported as such by the JSGLR parser, as it does not have an understanding of the semantics of the syntax. Failing to desugar all new application constructors of an extension may silently inhibit that the unparsed code is passed to the compiler. The Sugar* parser may need some time to resolve and parse dependencies of the currently edited file, which may lead to a noticeable delay between typing and updated syntax highlighting and static analysis results.

SugarScala however appears functional and can be used for prototyping syntactic DSL embeddings in Scala without the need to modify the Scala compiler.

4 Integration into the Scala Compiler

The previous chapters discuss support for syntactic extensibility for Scala with the help of the Sugar* framework. The facilities used in this framework are the composition of grammars expressed in SDF as well as term rewritings and analysis with the help of the Stratego language. The desugared terms are then finally unparsed and passed to the base language compiler to generate byte- or machine code. The framework however does not provide any means to access the type system of the base language which decreases the possibilities for the static analysis. From a theoretical point of view the type system of the base language could be reimplemented in Stratego, or a type-annotated output from the compiler could be reparsed and the information could be used to enrich the current term representation. But from a practical standpoint it is just not feasible to reimplement a complex type system as Scala‘s in Stratego. Passing parts of the unparsed AST to the Scala compiler and making use of the *XPrint* flag seems more feasible. But this output needs to be parsed again and the gained information must be re-integrated into the existing tree, which bears new sources of challenges and errors.

The approach discussed in this chapter is the direct integration of syntactic extensibility into the Scala compiler. Once being in the compiler, all its provided facilities can be used and the type information can be accessed directly in its internal representation. There would be no need to reimplement the type system and no error-prone parsing of converted type information output would be necessary.

4.1 Scala Compiler Overview

The Scala compiler is itself written in Scala. Its source code resides under `src/compiler` in the Scala code repository. The main entry point for the executable compiler is the class *scala.tools.nsc.Main*, where *nsc* is an abbreviation for new Scala compiler. *Main* extends the abstract type *scala.tools.nsc.Driver*, in which the actual *main* method is defined. Both *Driver* and *Main* are responsible to parse the command line arguments to the compiler, prepare the actual compilation environment represented by a *scala.tools.nsc.Global* class and actually start the compilation. *Global* can be considered the root of the compiler and imports and combines large parts of the remaining code of the compiler sources. Apart from the compilation environment and helpers, it defines and configures the different compilation phases and describes the compilation process and the switching of phases.

Phases active in the Scala compiler v2.10.3, according to the sources, are *parser*, *namer*, *typer*, *inlineclasses*, *pickler*, *refchecks*, *uncurry*, *specialize*, *explicitouter*, *erasure*, *posterasure*, *lambdalift*, *flatten*, *mixin*, *cleanup*, *icode*, *inliner*, *inlinerExceptionHandlers*, *closelim*, *dce* and *jvm*. The first compilation phase is *parser*, which has the responsibility to parse a source file and construct the initial AST. All following phases then sequentially analyze, transform or annotate the current AST, including checking for constraints or issuing compilation warnings. In the context of syntactic extensibility, the *parser* phase is the most interesting. So the remainder of this chapter will ignore the other phases and instead focus on the *parser* phase.

```scala
class Global /* ... */ {
  // ...
  // phaseName = "parser"
  object syntaxAnalyzer extends {
    val global: Global.this.type = Global.this
    val runsAfter = List[String]()
    val runsRightAfter = None
  } with SyntaxAnalyzer
  //...
}
```

Listing 4.1: Object syntaxAnalyzer definition in *Global*

```scala
// ...
abstract class SyntaxAnalyzer extends SubComponent with Parsers
               with MarkupParsers with Scanners with JavaParsers with JavaScanners {
  val phaseName = "parser"

  def newPhase(prev: Phase): StdPhase = new ParserPhase(prev)

  class ParserPhase(prev: scala.tools.nsc.Phase) extends StdPhase(prev) {
    // ...
    def apply(unit: global.CompilationUnit) {
      import global._
      informProgress("parsing " + unit)
      unit.body =
        if (unit.isJava) new JavaUnitParser(unit).parse()
        else if (reporter.incompleteHandled) new UnitParser(unit).parse()
        else new UnitParser(unit).smartParse()
    // ...
    }
  }
}
```

Listing 4.2: Original *SyntaxAnalyzer*

The definition of the parser phase in *Global* is given in Listing 4.1. It is of type *SyntaxAnalyzer* and implemented as an object member with the name *syntaxAnalyzer*. The compiler environment and the predecessors of the parser phase are provided in an early definition block to the object definition. The explicit passing of the *Global* instance is a common pattern used in the compiler and can be seen often. The code for the abstract *SyntaxAnalyzer* is given in Listing 4.2. Being a subcomponent of the compiler it extends the corresponding abstract class *SubComponent*, but also mixes in parsing and scanning utilities from *Parsers MarkupParsers*, *Scanners*, *JavaParsers* and *JavaScanners*. The definition of the inner class *ParserPhase* shows the general concept of the phases in the Scala compiler. Each phase is applied to a compilation unit, which is expressed nicely in the code. During this application, the abstract syntax tree for the compilation unit is checked or modified. In the case of the parser phase, the tree needs to be initial created by parsing the source file for the compilation unit. Unsurprisingly the definition of *apply* is basically an assignment to *unit.body*. The parsing itself is delegated to *UnitParser* or *JavaUnitParser*, depending on whether the compilation unit is Java or Scala.

```scala
// ...
abstract class SyntaxAnalyzer extends SubComponent with SGLRParsers with Parsers
               with MarkupParsers with Scanners with JavaParsers with JavaScanners {
  val phaseName = "parser"

  def newPhase(prev: Phase): StdPhase = new ParserPhase(prev)

  class ParserPhase(prev: scala.tools.nsc.Phase) extends StdPhase(prev) {
    // ...
    def apply(unit: global.CompilationUnit) {
      import global._
      informProgress("parsing " + unit)
      unit.body =
        if (unit.isJava) new JavaUnitParser(unit).parse()
        else new SGLRUnitParser(unit).parse()
    // ...
    }
  }
}
```

Listing 4.3: Modified *SyntaxAnalyzer*

As this work focuses on syntactic extensibility for Scala the *JavaUnitParser* will not be further explained. The *UnitParser* however is the entry point for the actual parsing. It has the responsibility to turn the Scala source code associated with the compilation unit into the AST on which the whole remaining compilation relies. *UnitParser* is a nested type of the *Parsers* trait in `scala/tools/nsc/ast/parser/Parsers.scala`. It is an extension of *SourceFileParser*, which further is an extension of *Parser*. In *Parser* resides the actual parsing logic, the other mentioned extensions are mainly used for configuration purposes. Unfortunately *Parser* does not only translate the scanned tokens into a naive first version of the AST, but has more responsibilities. One obvious additional responsibility for the parser is syntax error reporting. In cases where the parser just is not able to produce an AST it fails with a descriptive error message including position information. Another nice usability feature is the emisson of warnings in cases of error-prone or deprecated syntactical constructs.

But apart from reporting the parser also combines desugaring, synthetization and normalization, which makes it particularly hard to extend directly. An example for desugaring is the transformation of left associative infix expressions to priority-correctly nested function applications. So `a + b * c` is desugared to `a.$plus(b.$times(c))` and not `a.$plus(b).$times(c)`. Synthetization is a bit more elaborate and includes generation of default constructors for templates, which are created dependendly on the defined class parameters. An example for normalization is the transformation of lead-in package statements to nested package declarations, which wrap around the whole compilation unit.

4.2 Extension Approach

This bag of responsibilities is all mixed in the code and does not allow for easy extensibility of the parser. So instead of modifying the current parser or other existing code in the Scala compiler, we chose to replace it altogether. The entry point for the modification is the definition

```scala
trait SGLRParsers {
  // ...
  class SGLRUnitParser(unit: global.CompilationUnit) {
    def parse(): Tree = {
      val stratego_term = parser.parse(unit.source)
      val wrapped_term = Term(stratego_term)(unit.source)
      val iScalacAST = toTree(wrapped_term)
      val fullyTransformed = ToFullScalacASTTransformer.transform(iScalacAST)
      fullyTransformed
    }
  }
  // ...
}
```

Listing 4.4: Implementation of SGLRUnitParser

of *SyntaxAnalyzer*. The changed code is given in Listing 4.3. The case distinction between Java or non-Java code is kept intact. But in case of non-Java code instead of *UnitParser* a new *SGLRUnitParser* is instantiated and used to parse the compilation unit.

The implementation of SGLRUnitParser is given in Listing 4.4. The new approach to parse a compilation unit is split into four phases:

1. Derivation of a first parse tree with a SGLR parser
2. Wrapping of the derived parse tree into a better data structure for easier processing
3. Translation of the wrapped tree into an unfinished native Scala AST
4. Finishing the native Scala AST with further transformations, so it can be processed by the later phases

4.2.1 Derivation of a Parse Tree With a SGLR Parser

As the name already suggests *SGLRUnitParser* utilizes a SGLR parser to parse the source code. More precisely the JSGLR parser from the Spoofax project is used together with the corresponding term library. Both are included as Java archives and added to the dependencies of the project as `org.spoofax.jsglr_1.2.0.*.jar` and `org.spoofax.terms_1.2.0.*.jar`, respectively. The Scala wrapper around the JSGLR Java library is given in Listing 4.5. The SGLR parser expects two parameters for initialization (l. 11). The first is the tree builder to use and the second is the precompiled parse table. The tree builder used is one provided from the Spoofax library and is expected to return *IStrategoTerms* (l. 10 and l. 15ff). The parse table is created from the Scala grammar developed in this work with the help of the Spoofax Language Workbench. It is provided as resource in a linked JAR and loaded with the help of the Spoofax *ParseTableManager* (l. 5), which is itself wrapped in a simple Scala object of the same name (l. 23ff).

4.2.2 Wrapped Datastructure

The *IStrategoTerms* yielded from the parser are then again wrapped in Scala case classes. The resulting tree types are given in Figure 4.1 The abstract base class for the wrapped terms is *Term*.

```
trait SGLRParsers {
  // ...
  val scala_tbl_stream =
    getClass.getResourceAsStream("/scala/tools/nsc/sugar/Scala.tbl")
  val scala_tbl = ParseTableManager.loadFromStream(scala_tbl_stream)
  val parser = SGLRParser
  // ...

  object SGLRParser {
    val tb = new TreeBuilder
    val sglr = new SGLR(tb, scala_tbl)

    def parse(source: SourceFile): IStrategoTerm = {
      sglr.parse(new FileReader(source.file.file), source.file.name,
                 "CompilationUnit") match {
        case v: IStrategoTerm => v
        case unexp =>
          throw new RuntimeException(s"Expected IStrategoTerm, but got ${unexp}")
      }
    }
  }

  object ParseTableManager {
    val ptm = new org.spoofax.jsglr.io.ParseTableManager

    def loadFromStream(stream: InputStream) = ptm.loadFromStream(stream)
    def loadFromFile(file: File) = ptm.loadFromStream(new FileInputStream(file))
  }
  // ...
}
```

Listing 4.5: Scala wrapper around the Spoofax JSGLR parser

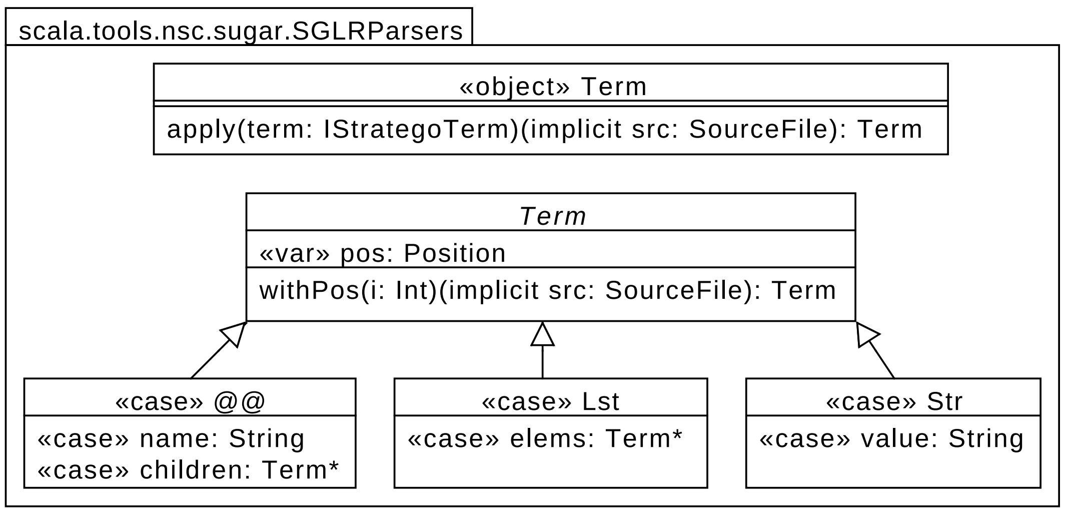


Figure 4.1.: Wrapped *Term* type hierarchy

Every *Term* has a position, which holds the offset to the term in the source file. The position is not part of the case variables for term, as it is mutable and may be changed with the helper method *withPos* at any time. It is extracted from the underlying *IStrategoTerm*, which also provides exact range and column information of the parse result. This information is currently not used, but could be in later versions.

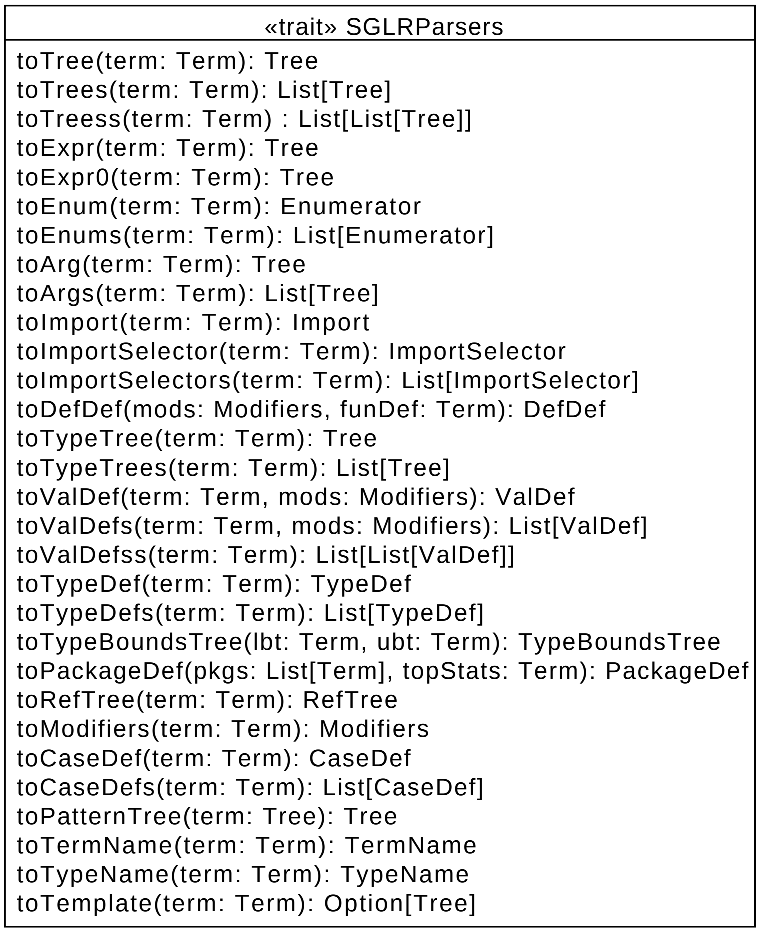
The subclasses of *Term* are *@@*, *Lst* and *Str*. These wrap around *StrategoAppl*, *StrategoList* and *StrategoString*, respectively. The operator style for the applications is used to allow a nicer syntax for later pattern matching and term rewriting. Every application has a name, which is equal to the constructor name of the node and an arbitary amount of children. A *Lst* only has an arbitrary amount of children, without a constructor name. A *Str* is an atomar value in a *Term* tree and is simply mapped to a Java/Scala *String*. The actual wrapping from *IStrategoTerm* to *Term* is defined in the *apply* method of the companion object *Term*. The implicit *SourceFile* is used as reference for the position information, which is also applied by the companion object.

With help of the companion object nested *IStrategoTerms* can be easily translated from cumbersome Java API into an idiomatic Scala algebraic data type. Having a Scala case class representation of the derived parse tree allows a functional pattern matching style processing and transformation. This simplifies the code for further processing and makes analysis and modification easier.

4.2.3 Transformation to an Unfinished Native Scala AST

The wrapped parse tree in form of the *Term* data type can not be used by the remaining compiler phases, as they expect the native Scala compiler AST to work with. For this purpose the *SGLRParsers* trait defines a couple of transformation methods from *Term* to *Tree*, which are shown in Figure 4.2. All this methods take one or more *Term* arguments, pattern match over the arguments and then either call one another or return a value corresponding with the method name. The entry point for the transformation is the general *toTree* method returning the corresponding general *Tree* type. Adding an "s" to the method name denotes a list of the type, which can repeated for nested lists. This can be seen with the *toTreess* method, which returns *List[List[Tree]]*. The nested lists of *Tree* are for example used to represent method arguments which can come in multiple argument lists with each having multiple arguments. But also more specialized methods like *toImportSelector* exist, which can be utilized everytime a *Term* is expected to transform to an *ImportSelector*. The native Scala AST often expects certain types for child-nodes in the tree, for which these specialized methods help avoiding type conversions. They also help to structure the transformation logic and provide better maintainability for the code.

To illustrate the basic approach and excerpt of *toTree* is given in Listing 4.6. As already mentioned, the body of the method is a large pattern match against the *term* argument. The usage of the operator name @@ for applications allows to use infix notations for application patterns. As a large amount of matches are against applications this improves the overall readability of the patterns. All parse trees derived with the Scala SDF grammar created in this work have *CompilationUnit* as root node. The first of the two arguments to this constructor is a list of package statements, the second is a list of other toplevel statements.



```
«trait» SGLRParsers
toTree(term: Term): Tree
toTrees(term: Term): List[Tree]
toTreess(term: Term) : List[List[Tree]]
toExpr(term: Term): Tree
toExpr0(term: Term): Tree
toEnum(term: Term): Enumerator
toEnums(term: Term): List[Enumerator]
toArg(term: Term): Tree
toArgs(term: Term): List[Tree]
toImport(term: Term): Import
toImportSelector(term: Term): ImportSelector
toImportSelectors(term: Term): List[ImportSelector]
toDefDef(mods: Modifiers, funDef: Term): DefDef
toTypeTree(term: Term): Tree
toTypeTrees(term: Term): List[Tree]
toValDef(term: Term, mods: Modifiers): ValDef
toValDefs(term: Term, mods: Modifiers): List[ValDef]
toValDefss(term: Term): List[List[ValDef]]
toTypeDef(term: Term): TypeDef
toTypeDefs(term: Term): List[TypeDef]
toTypeBoundsTree(lbt: Term, ubt: Term): TypeBoundsTree
toPackageDef(pkgs: List[Term], topStats: Term): PackageDef
toRefTree(term: Term): RefTree
toModifiers(term: Term): Modifiers
toCaseDef(term: Term): CaseDef
toCaseDefs(term: Term): List[CaseDef]
toPatternTree(term: Tree): Tree
toTermName(term: Term): TermName
toTypeName(term: Term): TypeName
toTemplate(term: Term): Option[Tree]
```

Figure 4.2.: List of methods used for the transformation of *Term* to *Tree*

The first pattern in l.5 matches against this *CompilationUnit* constructor and passes the arguments to the helper method *toPackageDef*, which wraps the toplevel statements in package declarations according to the provided package statements. The method *toPackageDef* uses *toTrees* on the toplevel statements, which then again maps *toTree* on the list of statements. Method *toTrees* is needed as it also handles other cases, for example import statements with multiple imports which then would again transform to *List[Tree]* rather than only one *Tree*.

The next pattern in l.8 matches against *TopStatSemi* and is a simple unwrapper. The actual semi information is discarded and *toTree* is recursively called on the actual statement.

The patterns in lines 10 to 25 match deeper than only one constructor application and are counted among the most complicated patterns used in the transformation. The reason for this is the rather complex nature of template definitions. Every toplevel template definition must have a name and may have annotations, modifiers and a template body. The parents of the template are defined as part of the template body. A trait definition may additionally have type parameters, as it can be polymorphic. A class definition may additionally have constructors annotations, constructors access modifiers and value parameters. Objects in the native Scala AST are represented by *ModuleDef* nodes, whereas classes and traits are represented by *ClassDef* nodes. The distinction between traits and classes are the used modifiers for the template and the constructor. Trait templates are always abstract and additionally have the special trait modifier.

```
 1 trait SGLRParsers {
 2   // ...
 3   def toTree(term: Term): Tree = term match {
 4     // --- Top Level Statements ---
 5     case "CompilationUnit" @@ (Lst(pkgs@_*), topStats) =>
 6       toPackageDef(pkgs.toList, topStats)
 7
 8     case "TopStatSemi" @@ (topStat, _) => toTree(topStat)
 9
10     case "TopTmplDef" @@
11         (annots, mods, "Object" @@ ("ObjectDef" @@ (name, body))) =>
12       IObjectDef(toModifiers(mods, annots), toTermName(name), toTemplate(body))
13
14     case "TopTmplDef" @@
15         (annots, mods, "Class" @@ ("ClassDef" @@
16           (morphism, constrAnnots, accessMods, classParamClauses, tplOpt))) =>
17       IClassDef(toModifiers(mods, annots), toTypeName(morphism),
18                 toTypeDefs(morphism), toModifiers(accessMods),
19                 toValDefss(classParamClauses), toTemplate(tplOpt))
20
21     case "TopTmplDef" @@
22         (annots, mods, "Trait" @@ ("TraitDef" @@ (id, typeParams, tplOpt))) =>
23       IClassDef(toModifiers(mods, annots) | Flags.TRAIT | Flags.ABSTRACT,
24                 toTypeName(id), toTypeDefs(typeParams),
25                 Modifiers() | Flags.TRAIT, ListOfNil, toTemplate(tplOpt))
26     // ...
27     // --- Some and None ---
28     case "Some" @@ (t) => toTree(t)
29     case @@("None")    => EmptyTree
30     case _             => toExpr(term)
31   }
32   // ...
33 }
```

Listing 4.6: Excerpt from *toTree*, illustrating the basic approach

This trait modifier is attached to the template and additionally used as access modifier for the constructor. The different typed arguments for the intermediate template representations are obtained by using the specialized helper methods *toModifiers*, *toTypeName*, *toTemplate*, *toTypeDefs* and *toValDefss* for the terms in the respective positions.

Optional nodes in the tree are represented by *Some* and *None* constructors. For this purpose some transformation methods match against these constructors to unwrap the values from *Some* or provide a fall back value for *None* as does *toTree* in l.28f. The default case for *toTree* is to try *toExpr*. Most other transformation methods will throw a descriptive exception instead.

A special case for the tree transformation are expressions – more precisely the handling of wildcards in expressions. Consider the following Scala expression: `_ + _`. The SGLR parser will derive the term `InfixExpr(WildcardExpr(), "+", WildcardExpr())` from this expression. Wildcards in expressions have the property of implicitly spanning a new closure for the scope of the wildcard. Each extra wildcard in the same scope will however just be a new parameter for the closure. So the expected transformed result of the above expression is `(x$1, x$2) => x$1.$plus(x$2)`, an anonymous function with two parameters, where the types of *x$1* and *x$2* need to be inferable

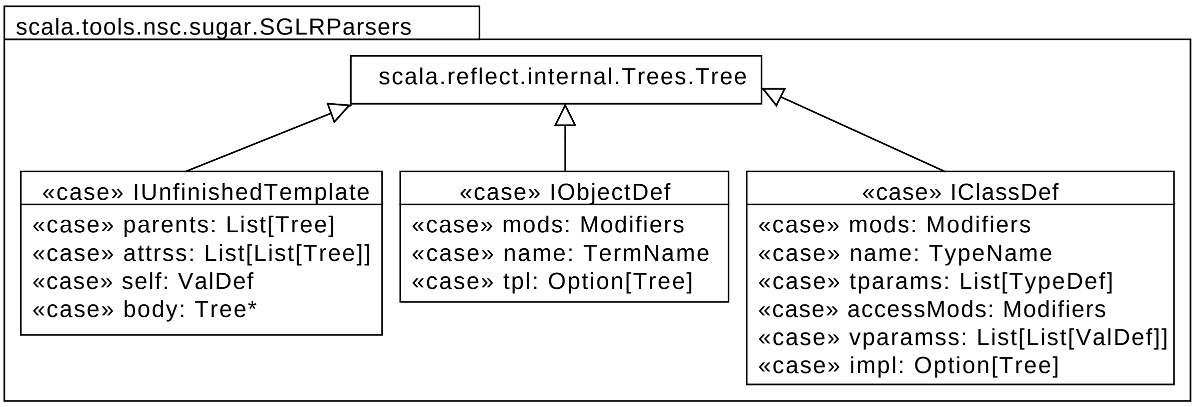

scala.tools.nsc.sugar.SGLRParsers
scala.reflect.internal.Trees.Tree
«case» IUnfinishedTemplate
«case» parents: List[Tree]
«case» attrss: List[List[Tree]]
«case» self: ValDef
«case» body: Tree*
«case» IObjectDef
«case» mods: Modifiers
«case» name: TermName
«case» tpl: Option[Tree]
«case» IClassDef
«case» mods: Modifiers
«case» name: TypeName
«case» tparams: List[TypeDef]
«case» accessMods: Modifiers
«case» vparamss: List[List[ValDef]]
«case» impl: Option[Tree]

Figure 4.3.: New intermediate representations for templates

from the context of the expression. A more elaborate example is the expression `_ op {_ + _}`. The curly braces denote a new scope, so this expressions needs to be transformed to nested anonymous functions. The expected result is `(x$1) => x$1.op(((x$2, x$3) => x$2.$plus(x$3)))`.

The need to handle closures in expressions yields two functions to handle expressions, *toExpr* and *toExpr0*. The former is used for expressions which introduce a new scope and contains the logic to store the currently seen placeholder parameters to create a fresh environment. The latter is used for expressions which should keep the current environment of placeholder parameters. It is also used by *toExpr* after the placeholder housekeeping logic is executed.

The actual logic to handle the encountered wildcards with their respective scope makes use of a mutable trait-wide *placeholderParams* variable of type *List[ValDef]*. A call to *toExpr* saves this list in a local variable and resets the trait-wide one to create a fresh environment. The input term is then forwarded to *toExpr0*, which contains the pattern matching and actual transformation logic. If *toExpr0* encounters a wildcard, it will prepend it on the trait-wide *placeholderParams*. After *toExpr0* has finished, *toExpr* will wrap the result in an anonymous function if any wildcards were encountered. Before *toExpr* returns, it will restore the *placeholderParams*. This same approach is used in the original Scala parser and the resulting code is copied in large parts, but of course adapted to the new *Term* data type.

The result of the transformation with *toTree* may be unfinished in the sense that it may contain nodes which are not recognized by other phases. The reason for this is the introduction of new children *IUnfinishedTemplate*, *IObjectDef* and *IClassDef* for *Term*, which are shown in Figure 4.3. They are prefixed with "I" for "Intermediate" and represent nodes which need further synthetization before they can be considered valid native Scala AST nodes. The idea behind this approach is the separation of two concerns:

1. Transforming the source code into a tree representation
2. Enriching this tree with implicit or derivable information

The second concern is especially elaborate in the context of templates, which lead to the decision to introduce the mentioned new nodes and defer the actual synthetization to a later step. As an immediate benefit the first concern is easier to express and comprehend. With regard to transformations on the *Term* structure it also eases the introduction of templates as part

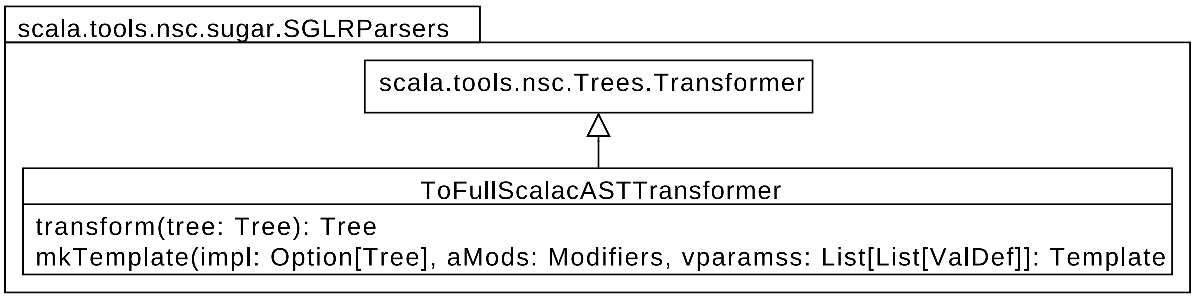

scala.tools.nsc.sugar.SGLRParsers
scala.tools.nsc.Trees.Transformer
ToFullScalacASTTransformer
transform(tree: Tree): Tree
mkTemplate(impl: Option[Tree], aMods: Modifiers, vparamss: List[List[ValDef]]): Template

Figure 4.4.: Signature of the transformer from intermediate *Tree* nodes to the native Scala AST

of desugarings, as the desugarer does not bear the responsibility to provide the synthesized information and can better focus on the logical abstraction.

4.2.4 Finishing the Native Scala AST for Further Processing

The finishing of the Scala AST with intermediate nodes is achieved with help of *ToFullScalacAST-Transformer*, which is shown in Figure 4.4. It extends the compiler-provided *Transformer* class, which contains the logic to recursively visit each node in the native Scala AST. The *transform* method pattern matches on the provided tree node and may transform the node in case of a match. If the node does not match the *Transformer* continues the visitation without the need of the implementor to explicitly define how to continue. The *mkTemplate* method is a helper method to actually achieve the transformation of the intermediate nodes from Figure 4.3. It makes heavy use of the provided helper method *Template* in `scala.tools.nsc.ast.Trees`.

The motivation for this extra step can best be illustrated with a minimal class example. Consider a compilation unit with the source file contents given in the following:

```
class Foo
```

It only contains the definition of the class *Foo*, which does not provide a template body. Not even a package declaration is provided for the compilation unit. Naturally one would expect a simple AST from this simple definition. The unexpected actual result created by the Scala parser is the following:

```
package <empty> {
  class Foo extends scala.AnyRef {
    def <init>() = {
      super.<init>();
      ()
    }
  }
}
```

As part of the normalization performed in the parser all definitions of a compilation unit are wrapped in a package declaration, even if there is no package statement at the beginning of a compilation unit. In this case a special name <empty> is used for the package declaration. The implicit fact that every non-value class has *scala.AnyRef* as parent is also made explicit by the parser. Even if the original source code did not provide a template body for *Foo* at all, the parser

	real[s]	user[s]	system[s]
scalac	5.212	10.415	0.192
sugsc	6.789	12.775	0.324
slowdown	+30.24%	+22.66%	+68.75%

Table 4.1.: Comparison of time needed to compile the test bench of 84 files averaged over five runs — scalac with optimized parser vs. sugsc with SGLR parser

adds the default template body containing the default constructor and the required call to super inside of the constructor. To emphasize once more: all this is information is added by the *parser*, not by the namer or typer or another compilation phase. Omitting this information would lead to errors in the later compilation phases. Fortunately the needed logic is partly provided as helper methods, values or objects. But to keep the transformation and synthetization concerns separate, the new intermediate types are introduced. This way the syntactical body- and parentless class definition can simply be transformed to *IClassDef* with *impl* set to *None*. The synthetization is then performed in this final finishing step.

4.3 Discussion

The integration of syntactic extensibility into the compiler is still in a early stage at the time of this writing. More precisely the current state does not even allow syntactic extensibility but is an incomplete prototypical replacement of the original Scala parser. The focus of this prototype is to produce the same native Scala AST as the original parser but use an extensible SGLR parser as backend. This SGLR parser uses a parse table derived from an extensible SDF grammar which in return allows easy extensibility of the parser. So the effortless addition of desugaring capabilities in later iterations seems promising. Moreover the integration in the compiler renders it possible to access the Scala type system which may allow type-depend desugarings in the future.

The focus on extensibility with the SGLR parser however comes with a cost. As the SGLR parser is not optimized for speed, opposed to the original Scala parser, the compilation time increases by roughly 30% on the test bench. The average timings over five test runs are shown in Table 4.1. Moreover the SGLR parser does not have the same error recovery capability as the native parser. Where the native parser produces a human-readable error message with position information, the SGLR parser simply fails with a cryptic parse error. Warnings on deprecated syntax or errors on semantically wrong but syntactically correct parse trees could be produced by the SGLR approach, but where neglected in focus of compatible parse results.

To assess the compatibility of the SGLR parser with the original parser the produced results are compared on two levels. The first level is the output created by passing the flag *-Xprint:parser* to the compilers. This causes the compiler to print the derived AST after the parser phase of the compilation. The output is then captured for both runs and compared using *sdiff* from GNU diffutils. Equal AST output is an indicator for equal behaviour, but not all AST information really shows up in the output. For this purpose the result is also compared on a lower level, which is the produced JVM bytecode. The resulting class files are disassembled with the most verbose invocation[1] of *javap* and then again compared with *sdiff*. This assures that the code compiled

[1] Using the flags -s -c -p

from the SGLR AST shows exactly the same behaviour as the code compiled from the original parser AST for the test bench.

The test bench currently consists of 84 hand-crafted Scala files each trying to test a certain syntactical construct. These files reach from different versions of *HelloWorld* (with/without package statement, explicit main as procedure, explicit main as definition, extension of App, ...), over different template and statement constructs to expressions, including closure constructs. All this source files contain valid syntax – no negative tests are formulated. We unfortunately ran out of time and could thus not complete the transformation from our AST to the native Scala AST, but all sample files can be correctly compiled. In summary our approach seems promising to be able to completely replace the Scala parser for research purposes in the future.

5 Related Work

Scala Macros [7] are a new experimental language feature of Scala 2.10. They enable compile-time metaprogramming with respect to the rich syntax and static type system of Scala. A Scala Macro definition resembles a regular function definition, but is distinguished by the keyword **macro** at the beginning of the function body. The body is then expected to be a qualifier to the macro implementation, rather then an arbitrary Scala block. A macro application is syntactically equal to a function application. But the arguments to the macro are not the objects resulting from an evaluation of the argument expressions, but the abstract syntax trees of the expressions in argument position. The macro implementation can then, conditionally on the passed abstract syntax and the context it was applied in, rewrite the abstract syntax tree.

To spare a macro implementor from the error-prone burden to manually construct abstract syntax trees with the help of classes from the compiler-library a macro called *reify* is provided as part of the macro feature. This macro works opposite to the evaluation function and allows to turn typed Scala expressions into Scala abstract syntax trees, promoting it to a quasiquoting-similar utility. Furthermore are all macros which only construct abstract syntax trees with help of *reify* hygienic [18], as the implemention of *reify* takes care of hygiene.

Together with string interpolation Scala Macros allow a better embedding of external DSLs [6]. String interpolation in Scala allows to prepend strings with arbitrary identifiers. A corresponding implicit method to that identifier can then be used to process the provided string, split into static and interpolated parts. Using a macro in that context allows to process the string at compile time. Provided the string represents a deeply embedded language this further allows static analysis with help of the Scala type system.

Quasiquotes for Scala [28] is another example for a Scala extension making use of the combination of macros and string interpolation. Where *reify* only works with typed Scala expressions, a quasiquote, denoted by the *q* string interpolation prefix, allows to create untyped Scala abstract syntax trees, even including definitions and values. Quasiquotes aim to be a drop-in replacement for *reify*, as they can be used for a wider range of syntax and further minimize the need to manually construct or rewrite Scala abstract syntax trees with the compiler-library.

Yet another approach to increase the extensibility of Scala is Scala-Virtualized [26]. The aim of the virtualization is to make Scala a better host for embedded DSLs. For this purpose it introduces infix methods, expresses control flow statements as method calls and provides source code context information through *implicit* argument expressions.

However non of these approaches allows to extend Scala directly with arbitrary syntax similar to the build-in XML support because they are all bound to the unextensible Scala parser integrated into the Scala compiler. The nearest one can get with these approaches to syntactically embed XML is to wrap the XML into an interpolated string. The direct use of XML elements in an expression position is however not possible to realize, as XML uses a completey different syntax compared to Scala.

SugarScala instead builds on the extensibility of the parser and thus allows any language expressible as a context free grammar to be an extension of Scala on the syntactical level, as has been shown in the XML case study. But the actual deep embedding of XML is still expressed in form of a Scala library and not in SugarScala. So the relation of SugarScala to Scala Macros, String Interpolation and Scala-Virtualized is not competetive, but rather cooperative. It can be considered another stage on top of a staging compilation process which allows to seemlessly embed DSLs with their native syntax into Scala.

Other work does not aim on the general extensibility of Scala but has its focus on one particular DSL or submodule of Scala. Ozma [10] extends Scala with features from the Oz language [29] for concurrent and distributed systems. Akin to that is SubScript [30] which extends Scala with the algebra of communicating processes. Burak Emir has extended the pattern matching of Scala to allow better XML processing similar to the XPath [8] and XQuery [2] languages [11]. Garcia, Izmaylova and Schupp have extended Scala with the capability to formulate database queries similar to Microsoft's LINQ [15, 23]. REScala [27] builds on EScala and further extends Scala with reactive functional programming capabilities.

Presumably more work on the extension of Scala exists but the point is the large interest for the extension of Scala with any kind of domain specific language capability. SugarScala can help with the embedding of syntax for these domain specific languages, if at least for early prototyping or feasability studies.

6 Conclusion and Future Work

We have argued that Scala should have an extensible syntax to allow DSL embedding similar to its built-in support for XML. For this purpose we have crafted a modular and extensible context-free grammar for Scala in SDF2. Based on this grammar we have developed SugarScala as an instantiation of the Sugar* framework. We have further presented two extensions for SugarScala in the case studies for XML and EScala. Finally we have investigated a possibility to integrate our work in the Scala compiler to allow for built-in syntactic extensibility in the future.

The grammar could be further tuned to be more concise and it has room for improvement in terms of performance and precision. We however diligently tested it for quality and quantity and it has proven to be usable for the XML and EScala case studies. We could additionally show that the ASTs resulting from the grammar can be correctly transformed to the native Scala AST for a selected subset of Scala source files.

SugarScala is in an experimental state and has some issues concerning performance, stability and usability, but we could successfully utilize it for the XML and EScala case studies. It does neither allow access to the Scala type system nor does it provide static analysis for Scala. Nevertheless can SugarScala be used for quick prototyping of syntactic DSL embedding without the need to modify the Scala compiler. One could even argue that the lack of access to the Scala type system is a benefit because it requires the DSL implementor to provide a library-based embedding of the DSL with a good API which can be used with just simple desugarings.

We achieved the integration in the Scala compiler by replacing a call to the *parse* method of *UnitParser* in the parser phase with a call to a method of our new trait *SGLRUnitParser*. This new trait utilizes the JSGLR parser of the Spoofax project in combination with the generated parse table of our grammar to parse the contents of provided compilation unit sources. We then transform the resulting AST of our grammar to the native Scala AST of the compiler for further use in the remaining compilation process.

We compare the textual representation of the resulting AST as well as the disassembled resulting byte-code of our implementation with the respective results of the native Scala compiler to assert correctness of our implementation. The transformation is not yet completely defined for all nodes in our AST, but we were successful to construct the correct native Scala AST for few chosen examples, including general template definitions and implicit expression closures. We interpret this success as indication of the feasibility of the approach.

Future work could finish the transformation of the AST and provide a complete replacement of the current hard-to-extend Scala parser. The next steps then would be to allow user-defined syntax extensions together with the necessary transformations conceptually similar to the Sugar* framework. These extensions would however have the possibility to access all compiler facilities, including the type system, and could thus be more elaborate than current SugarScala extensions. Additionally could the transformations be based on Scala with arbitrary libraries, e.g. Kiama[1], as alternative to Stratego.

[1] http://code.google.com/p/kiama/

Bibliography

[1] O. Ben-Kiki, C. Evans, and B. Ingerson. Yaml ain't markup language (yaml™) version 1.1. *Working Draft 2008-05*, 11, 2001.

[2] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. Xquery 1.0: An xml query language. *W3C working draft*, 12, 2003.

[3] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Science of Computer Programming*, 72(1–2):52 – 70, 2008. <ce:title>Special Issue on Second issue of experimental software and toolkits (EST)</ce:title>.

[4] M. Bravenboer and E. Visser. Concrete syntax for objects: Domain-specific language embedding and assimilation without restrictions. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 365–383, New York, NY, USA, 2004. ACM.

[5] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (xml). *World Wide Web Journal*, 2(4):27–66, 1997.

[6] E. Burmako. Scala macros: Let our powers combine! In *Proceedings of the 4th Annual Scala Workshop*, 2013.

[7] E. Burmako and M. Odersky. Scala macros, a technical report. In *Third International Valentin Turchin Workshop on Metacomputation*, page 23, 2012.

[8] J. Clark, S. DeRose, et al. Xml path language (xpath), 1999.

[9] D. Crockford. The application/json media type for javascript object notation (json). 2006.

[10] S. Doeraene. Ozma: Extending scala with oz concurrency. *Université Catholique de Louvain, Belgium, Masterarbeit*, 2011.

[11] B. Emir. Extending pattern matching with regular tree expressions for xml processing in scala. *Master's thesis, RWTH Aachen*, 2003.

[12] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. Sugarj: library-based syntactic language extensibility. *ACM SIGPLAN Notices*, 46(10):391–406, 2011.

[13] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. Layout-sensitive generalized parsing. In *Software Language Engineering*, pages 244–263. Springer, 2013.

[14] S. Erdweg and F. Rieger. A framework for extensible languages. In *Proceedings of the 12th international conference on Generative programming: concepts & experiences*, pages 3–12. ACM, 2013.

[15] M. Garcia, A. Izmaylova, and S. Schupp. Extending scala with database query capability. *Journal of Object Technology*, 9(4):45–68, 2010.

[16] V. Gasiunas, L. Satabin, M. Mezini, A. Núñez, and J. Noyé. Escala: modular event-driven object interactions in scala. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD ’11, pages 227–240, New York, NY, USA, 2011. ACM.

[17] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism sdfreference manual. *SIGPLAN Not.*, 24(11):43–75, Nov. 1989.

[18] D. Herman and M. Wand. A theory of hygienic macros. In S. Drossopoulou, editor, *Programming Languages and Systems*, volume 4960 of *Lecture Notes in Computer Science*, pages 48–62. Springer Berlin Heidelberg, 2008.

[19] L. C. Kats, R. Vermaas, and E. Visser. Integrated language definition testing: enabling test-driven language development. In *ACM SIGPLAN Notices*, volume 46, pages 139–154. ACM, 2011.

[20] L. C. Kats and E. Visser. The spoofax language workbench: rules for declarative specification of languages and ides. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA ’10, pages 444–463, New York, NY, USA, 2010. ACM.

[21] F. Lorenzen and S. Erdweg. Modular and automated type-soundness verification for language extensions. ICFP, 2013.

[22] S. McIntosh, B. Adams, and A. Hassan. The evolution of ant build systems. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 42–51, 2010.

[23] E. Meijer, B. Beckman, and G. Bierman. Linq: Reconciling object, relations and xml in the .net framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’06, pages 706–706, New York, NY, USA, 2006. ACM.

[24] M. Odersky. The scala language specification, version 2.8. *EPFL Lausanne, Switzerland*, 2009.

[25] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the scala programming language. Technical report, Citeseer, 2004.

[26] T. Rompf, N. Amin, A. Moors, P. Haller, and M. Odersky. Scala-virtualized: linguistic reuse for deep embeddings. *Higher-Order and Symbolic Computation*, pages 1–43, 2013.

[27] G. Salvaneschi, G. Hintz, and M. Mezini. Rescala: Bridging between object-oriented and functional style in reactive applications. In *Proceedings of the 13th International Conference on Aspect-Oriented Software Development, AOSD*, volume 14, 2013.

[28] D. Shabalin, E. Burmako, and M. Odersky. Quasiquotes for scala, a technical report. 2013.

[29] G. Smolka. The oz programming model. In J. Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 324–343. Springer Berlin Heidelberg, 1995.

[30] A. van Delft. Subscript: Extending scala with the algebra of communicating processes. *Scala Days*, 2012, 2012.

[31] M. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Trans. Softw. Eng. Methodol.*, 5(1):1–41, Jan. 1996.

[32] M. G. Van Den Brand, J. Scheerder, J. J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized lr parsers. In *Compiler Construction*, pages 143–158. Springer, 2002.

[33] E. Visser. Scannerless generalized-lr parsing. Technical report, Technical Report, 1997.

[34] E. Visser. *Syntax definition for language prototyping*. Eelco Visser, 1997.

A Full SDF Grammar

```
%%% Scala-Annotations.sdf %%%
module Scala-Annotations

imports
Scala-Expressions
Scala-Types

exports

context-free syntax
"@" SimpleType ArgumentExprsSeq? -> Annotation                     {"Annotation",
                                          layout("2.last.line == 3.first.line")}

"@" SimpleType ArgumentExprs? -> ConstrAnnotation           {"ConstrAnnotation",
                                          layout("2.last.line == 3.first.line")}

Annotation                -> AnnotationSeq
Annotation AnnotationSeq -> AnnotationSeq                        {"AnnotationSeq",
                                  layout("2.first.line - 1.last.line < num(2)")}


%%% Scala-BasicDeclsDefs.sdf %%%
module Scala-BasicDeclsDefs

imports
Scala-ClassesObjects
Scala-Expressions
Scala-Identifiers
Scala-Types

exports

context-free syntax
%% Extracted to avoid optional lexical syntax:
%%     (":" Type)?
":" Type      -> Typed                                                  {"Typed"}
":" ParamType -> ParamTyped                                        {"ParamTyped"}

%% §4 - Basic Declarations and Definitions
PatVarDef       -> Def
TmplDef         -> Def


%% §4.1 - Value Declarations and Definitions
"val" ValDcl                        -> Dcl                          {"ValDclDcl"}
{Id ","}+ ":" Type                  -> ValDcl                          {"ValDcl"}
"val" PatDef                        -> PatVarDef                    {"ValPatDef"}
{Pattern2 ","}+ Typed? "=" Expr     -> PatDef                          {"PatDef"}
```

```
%% §4.2 - Variable Declarations and Definitions
"var" VarDcl               -> Dcl                              {"VarDclDcl"}
"var" VarDef               -> PatVarDef                        {"VarPatDef"}
{Id ","}+ ":" Type         -> VarDcl                              {"VarDcl"}
{Id ","}+ ":" Type "=" "_" -> VarDef              {prefer, "WildcardVarDef"}
PatDef                     -> VarDef


%% §4.3 - Type Declarations and Type Aliases
"type" TypeDcl                                        -> Dcl       {"TypeDclDcl"}
Id TypeParamClause? LowerBoundType? UpperBoundType? -> TypeDcl      {"TypeDcl"}
"type" TypeDef                                        -> Def       {"TypeDefDef"}
Id TypeParamClause? "=" Type                          -> TypeDef      {"TypeDef"}


%% §.4.4 - Type Parameters
"[" {VariantTypeParam ","}+ "]" -> TypeParamClause           {"TypeParamClause"}

Annotation* TypeParam     -> VariantTypeParam               {"VariantTypeParam"}
Annotation* "+" TypeParam -> VariantTypeParam           {"PlusVariantTypeParam"}
Annotation* "-" TypeParam -> VariantTypeParam            {"NegVariantTypeParam"}

Id  TypeParamClause? LowerBoundType?
  UpperBoundType? TypeViewBound* TypeContextBound* -> TypeParam    {"TypeParam"}
"_" TypeParamClause? LowerBoundType?
  UpperBoundType? TypeViewBound* TypeContextBound* -> TypeParam
                                                            {"WildcardTypeParam"}

">:" Type -> LowerBoundType                                    {"LowerBoundType"}
"<:" Type -> UpperBoundType                                    {"UpperBoundType"}
"<%" Type -> TypeViewBound                                      {"TypeViewBound"}
":" Type  -> TypeContextBound                                {"TypeContextBound"}


%% §4.6 - Function Declarations and Definitions
"def" FunDcl                         -> Dcl                        {"FunDclDcl"}
FunSig ":" Type                      -> FunDcl                        {"FunDcl"}
"def" FunDef                         -> Def                        {"FunDefDef"}
FunSig Typed? "=" Expr               -> FunDef                        {"FunDef"}
Id TypeParamClause? ParamClauses?    -> FunSig                        {"FunSig",
                                     layout("3.first.line - 1.last.line < num(2)")}
%% Annotations are allowed for function type parameters according to
%% scalac v2.10.0*, % so TypeParamClause is used here instead of
%% FunTypeParamClause
%% "[" {TypeParam ","}+ "]"         -> FunTypeParamClause {"FunTypeParamClause"}

ParamClause                     -> ParamClauses
"(" "implicit" {Param ","}+ ")" -> ParamClauses           {"ImplicitParamClause"}
ParamClause ParamClauses        -> ParamClauses                  {"ParamClauses",
                                layout("2.first.line - 1.last.line < num(2)")}

"(" {Param ","}* ")" -> ParamClause                               {"ParamClause"}

Annotation* Id ParamTyped? Assignment? -> Param                         {"Param"}
```

```
Type      -> ParamType
"=>" Type -> ParamType                                               {"ByNameParam"}
Type "*"  -> ParamType                                             {"RepeatedParam"}

%% §4.6.3 - Procedures
FunSig                 -> FunDcl                                         {"ProcDcl"}
FunSig "{" Block "}" -> FunDef                                          {"ProcDef",
                                  layout("2.first.line - 1.last.line < num(2)")}

%% §4.7 - Import Clauses
"import" {ImportExpr ","}+ -> Import                                      {"Import"}

StableId                     -> ImportExpr                          {"ImportExpr"}
StableId "." "_"             -> ImportExpr                  {"WildcardImportExpr"}
StableId "." ImportSelectors -> ImportExpr                 {"SelectorsImportExpr"}

"{" {ImportSelector ","}+ "}"         -> ImportSelectors     {"ImportSelectors"}
"{" {ImportSelector ","}+ "," "_" "}" -> ImportSelectors
                                              {"ImportSelectorsWithWildcard"}
"{" "_" "}"                           -> ImportSelectors
                                              {"OnlyWildcardImportSelectors"}

Id          -> ImportSelector                                  {"ImportSelector"}
Id "=>" Id  -> ImportSelector                            {"MappedImportSelector"}
Id "=>" "_" -> ImportSelector                          {"WildcardImportSelector"}


%%% Scala-ClassesObjects.sdf %%%
%% §5 Classes and Objects
module Scala-ClassesObjects

imports
Scala-BasicDeclsDefs
Scala-Expressions
Scala-Identifiers
Scala-Types
Scala-Whitespace

exports

context-free syntax
%% §5.1 - Templates
EarlyDefs? ClassParents TemplateBody? -> ClassTemplate         {"ClassTemplate"}
EarlyDefs? TraitParents TemplateBody? -> TraitTemplate         {"TraitTemplate"}

Constr WithAnnotType*    -> ClassParents                        {"ClassParents"}
AnnotType WithAnnotType* -> TraitParents                        {"TraitParents"}

"with" AnnotType         -> WithAnnotType                      {"WithAnnotType"}

Id Typed? "=>"       -> SelfType                                    {"SelfType"}
"this" ":" Type "=>" -> SelfType                                {"ThisSelfType"}
"_" ":" Type "=>"    -> SelfType                            {"WildcardSelfType"}
```

```
"{" SelfType TemplateStatSemi* "}" -> TemplateBody      {"SelfTypeTemplateBody",
                                                                          prefer}
"{" TemplateStatSemi* "}"          -> TemplateBody              {"TemplateBody"}

%% §5.1.1 - Constructor Invocations
AnnotType ArgumentExprsSeq? -> Constr                                 {"Constr"}

%% §5.1.6 - Early Definitions
"{" EarlyDefSemi* "}" "with" -> EarlyDefs                          {"EarlyDefs"}

EarlyDef SEMI -> EarlyDefSemi                   {longest-match, "EarlyDefSemi"}
EarlyDef EOL  -> EarlyDefSemi                                  {"EarlyDefSemi",
                                        enforce-newline, longest-match, prefer}
EarlyDef EOB  -> EarlyDefSemi            {longest-match, avoid, "EarlyDefSemi"}

AnnotationSeq? Modifier* PatVarDef -> EarlyDef                      {"EarlyDef"}


%% §5.2 - Modifiers
LocalModifier  -> Modifier
AccessModifier -> Modifier
"override"     -> Modifier                                  {"OverrideModifier"}

"abstract" -> LocalModifier                                 {"AbstractModifier"}
"final"    -> LocalModifier                                    {"FinalModifier"}
"sealed"   -> LocalModifier                                   {"SealedModifier"}
"implicit" -> LocalModifier                                 {"ImplicitModifier"}
"lazy"     -> LocalModifier                                     {"LazyModifier"}

"private" AccessQualifier?   -> AccessModifier               {"PrivateModifier"}
"protected" AccessQualifier? -> AccessModifier             {"ProtectedModifier"}

"[" Id "]"      -> AccessQualifier                           {"AccessQualifier"}
"[" "this" "]" -> AccessQualifier                              {"ThisQualifier"}


%% §5.3 - Class Definitions
"class" ClassDef -> TmplDef                                            {"Class"}

Morphism ConstrAnnotation*
  AccessModifier? ClassParamClauses? ClassTemplateOpt -> ClassDef   {"ClassDef",
                                  layout("4.first.line - 1.last.line < num(2)")}

Id                  -> Morphism
Id TypeParamClause  -> Morphism                                    {"Polymorph"}

ClassParamClause                             -> ClassParamClauses
"(" "implicit" {ClassParam ","}+ ")" -> ClassParamClauses
                                            {prefer, "ImplicitClassParamClause"}
ClassParamClause ClassParamClauses     -> ClassParamClauses
            {layout("2.first.line - 1.last.line < num(2)"), "ClassParamClauses"}

"(" {ClassParam ","}* ")" -> ClassParamClause               {"ClassParamClause"}
```

```
Annotation* Id ":" ParamType Assignment?                 -> ClassParam
                                                                {"ClassParam"}
Annotation* Modifier* "val" Id ":" ParamType Assignment? -> ClassParam
                                                             {"ValClassParam"}
Annotation* Modifier* "var" Id ":" ParamType Assignment? -> ClassParam
                                                             {"VarClassParam"}

"extends" ClassTemplate -> ClassTemplateOpt            {"ClassClassTemplateOpt"}
"extends" TemplateBody  -> ClassTemplateOpt         {"TemplateClassTemplateOpt"}
                        -> ClassTemplateOpt            {"EmptyClassTemplateOpt"}
TemplateBody            -> ClassTemplateOpt


%% §5.3.1 - Constructor Definitions
"this" ParamClauses "=" ConstrExpr -> FunDef                  {"ThisExprFunDef"}
"this" ParamClauses ConstrBlock    -> FunDef                 {"ThisBlockFunDef",
                                       layout("3.first.line - 2.last.line < num(2)")}

"this" ArgumentExprsSeq -> SelfInvocation                     {"SelfInvocation"}
"this" BlockExpr        -> SelfInvocation                {"BlockSelfInvocation",
                                       layout("2.first.line - 1.last.line < num(2)")}

SelfInvocation -> ConstrExpr
ConstrBlock    -> ConstrExpr

"{" SelfInvocation ";" BlockStatSemi* "}"  -> ConstrBlock        {"ConstrBlock"}
"{" SelfInvocation "}"                     -> ConstrBlock        {"ConstrBlock"}
"{" SelfInvocation BlockStatSemi+ "}"      -> ConstrBlock        {"ConstrBlock",
                                       layout("3.first.line - 2.last.line > num(0)")}

%% §5.3.2 - Case Classes
"case" "class" ClassDef   -> TmplDef                               {"CaseClass"}

%% §5.3.3 - Traits
"trait" TraitDef -> TmplDef                                            {"Trait"}

Id TypeParamClause? TraitTemplateOpt -> TraitDef                    {"TraitDef"}

"extends" TraitTemplate -> TraitTemplateOpt            {"TraitTraitTemplateOpt"}
"extends" TemplateBody  -> TraitTemplateOpt         {"TemplateTraitTemplateOpt"}
                        -> TraitTemplateOpt            {"EmptyTraitTemplateOpt"}
TemplateBody            -> TraitTemplateOpt

%% §5.4 - Object Definitions
"case" "object" ObjectDef -> TmplDef                              {"CaseObject"}
"object" ObjectDef        -> TmplDef                                  {"Object"}

Id ClassTemplateOpt -> ObjectDef                                   {"ObjectDef"}
```

```
%%% Scala-Expressions.sdf %%%
module Scala-Expressions

imports
Scala-Annotations
Scala-BasicDeclsDefs
Scala-ClassesObjects
Scala-Literals
Scala-PatternMatching
Scala-Types
Scala-Whitespace

exports

lexical syntax
-> EOC

context-free restrictions
EOC -/- ~[c]


%%%%% With Layout %%%%%%
context-free priorities
{
  Literal                 -> Expr                                          {prefer}
  "_"                     -> Expr                                  {"WildcardExpr"}
  Expr ArgumentExprs      -> Expr                                       {"AppExpr",
                                      layout("1.last.line == 2.first.line")}
  Expr BlockExpr          -> Expr                                  {"BlockAppExpr",
                              layout("2.first.line - 1.last.line < num(2)")}
  "(" {NoLExpr ","}* ")"  -> Expr                                     {"TupleExpr"}
  Expr TypeArgs           -> Expr                               {"TypeApplication"}
  Path                    -> Expr
}

> {
  "new" ClassTemplate  -> Expr                        {prefer, "NewClassExpr"}
  "new" TemplateBody   -> Expr                     {prefer, "NewTemplateExpr"}
  Expr "_"             -> Expr                      {left, "EtaExpansionExpr",
                                      layout("1.last.line == 2.first.line")}
  BlockExpr            -> Expr
}

> PREFIX Expr             -> Expr                                    {"PrefixExpr",
                              prefer, layout("1.last.line == 2.first.line")}
> PREFIX Expr             -> Expr                                    {"PrefixExpr",
                              prefer, layout("1.last.line == 2.first.line")}
> {
  Expr SPECIAL-OP Expr         -> Expr                         {left, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
  Expr RASSOC-SPECIAL-OP Expr -> Expr                         {right, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
}
```

```
> {
  Expr MULT-OP Expr         -> Expr                              {left, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
  Expr RASSOC-MULT-OP Expr -> Expr                              {right, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
}
> {
  Expr SUM-OP Expr         -> Expr                               {left, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
  Expr RASSOC-SUM-OP Expr -> Expr                               {right, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
}
> {
  Expr COLON-OP Expr         -> Expr                             {left, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
  Expr RASSOC-COLON-OP Expr -> Expr                             {right, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
}
> {
  Expr CMPR-OP Expr         -> Expr                              {left, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
  Expr RASSOC-CMPR-OP Expr -> Expr                              {right, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
}
> {
  Expr BRACKET-OP Expr         -> Expr                           {left, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
  Expr RASSOC-BRACKET-OP Expr -> Expr                           {right, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
}
> {
  Expr AMPERSAND-OP Expr         -> Expr                         {left, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
  Expr RASSOC-AMPERSAND-OP Expr -> Expr                         {right, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
}
> {
  Expr CIRCUMFLEX-OP Expr         -> Expr                        {left, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
  Expr RASSOC-CIRCUMFLEX-OP Expr -> Expr                        {right, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
}
> {
  Expr BAR-OP Expr         -> Expr                               {left, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
  Expr RASSOC-BAR-OP Expr -> Expr                               {right, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
}
> {
  Expr LETTER-OP Expr         -> Expr                            {left, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
  Expr RASSOC-LETTER-OP Expr -> Expr                            {right, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
}
```

```
> Expr ASSIGN-OP Expr      -> Expr                             {left, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
> Expr Id -> Expr  {avoid, layout("1.last.line == 2.first.line"), "PostfixExpr"}
> Expr Id -> Expr  {avoid, layout("1.last.line == 2.first.line"), "PostfixExpr"}
> Expr "match" "{" CaseClause+ "}" -> Expr                             {"MatchExpr"}
> Expr ArgumentExprs "=" Expr -> Expr         {non-assoc, "AccessAssignmentExpr"}
<0> > Expr "." Id "=" Expr     -> Expr     {non-assoc, "DesignatorAssignmentExpr"}
<0> > {
  Bindings "=>" Expr                           -> Expr          {prefer, "FunExpr"}
  Id "=>" Expr                                 -> Expr        {prefer, "IdFunExpr"}
  "implicit" Id "=>" Expr                      -> Expr {prefer, "ImplicitFunExpr"}
  "_" "=>" Expr                                -> Expr {prefer, "WildcardFunExpr"}
  "if" "(" NoLExpr ")" Expr                    -> Expr           {prefer, "IfExpr"}
  "if" "(" NoLExpr ")" Expr "else" Expr        -> Expr               {"IfElseExpr"}
  "if" "(" NoLExpr ")" Expr ";" "else" Expr    -> Expr               {"IfElseExpr"}
  "while" "(" NoLExpr ")" Expr                 -> Expr                {"WhileExpr"}
  "try" Expr                                   -> Expr          {prefer, "TryExpr"}
  "try" Expr "catch" Expr                      -> Expr             {"TryCatchExpr"}
  "try" Expr "finally" Expr                    -> Expr           {"TryFinallyExpr"}
  "try" Expr "catch" Expr "finally" Expr       -> Expr     {"TryCatchFinallyExpr",
                                                                            avoid}
  "do" Expr ";" "while" "(" Expr ")"           -> Expr              {"DoWhileExpr"}
  "do" Expr "while" "(" Expr ")"               -> Expr              {"DoWhileExpr"}
  "for" "(" EnumeratorSemi+ ")" Expr           -> Expr                  {"ForExpr"}
  "for" "{" EnumeratorSemi+ "}" Expr           -> Expr                  {"ForExpr"}
  "for" "(" EnumeratorSemi+ ")" "yield" Expr -> Expr               {"ForYieldExpr"}
  "for" "{" EnumeratorSemi+ "}" "yield" Expr -> Expr               {"ForYieldExpr"}
  "throw" Expr                                 -> Expr                {"ThrowExpr"}
  "return" Expr?                               -> Expr              {"ReturnExpr",
                                          layout("1.last.line == 2.first.line")}
  Id "=" Expr                                -> Expr {non-assoc, "AssignmentExpr"}
  Expr Ascription                            -> Expr     {avoid, "AscriptionExpr"}
}

context-free priorities
  Expr "." Id   -> Expr                                  {avoid, "DesignatorExpr"}
  > PREFIX Expr -> Expr                                              {"PrefixExpr"}


%%%%% NO Layout %%%%%%
context-free priorities
{
  Literal                    -> NoLExpr                                   {prefer}
  "_"                        -> NoLExpr                           {"WildcardExpr"}
  NoLExpr ArgumentExprs      -> NoLExpr                                {"AppExpr"}
  NoLExpr BlockExpr          -> NoLExpr                           {"BlockAppExpr"}
  "(" {NoLExpr ","}* ")"     -> NoLExpr                              {"TupleExpr"}
  NoLExpr TypeArgs           -> NoLExpr                        {"TypeApplication"}
  Path                       -> NoLExpr
}
```

```
> {
  "new" ClassTemplate -> NoLExpr                        {prefer, "NewClassExpr"}
  "new" TemplateBody  -> NoLExpr                     {prefer, "NewTemplateExpr"}
  NoLExpr "_"         -> NoLExpr                     {left, "EtaExpansionExpr"}
  BlockExpr           -> NoLExpr
}

> PREFIX NoLExpr             -> NoLExpr                            {"PrefixExpr"}
> PREFIX NoLExpr             -> NoLExpr                            {"PrefixExpr"}
> {
  NoLExpr SPECIAL-OP NoLExpr         -> NoLExpr                {left, "InfixExpr"}
  NoLExpr RASSOC-SPECIAL-OP NoLExpr -> NoLExpr                {right, "InfixExpr"}
}
> {
  NoLExpr MULT-OP NoLExpr         -> NoLExpr                   {left, "InfixExpr"}
  NoLExpr RASSOC-MULT-OP NoLExpr -> NoLExpr                   {right, "InfixExpr"}
}
> {
  NoLExpr SUM-OP NoLExpr         -> NoLExpr                    {left, "InfixExpr"}
  NoLExpr RASSOC-SUM-OP NoLExpr -> NoLExpr                    {right, "InfixExpr"}
}
> {
  NoLExpr COLON-OP NoLExpr         -> NoLExpr                  {left, "InfixExpr"}
  NoLExpr RASSOC-COLON-OP NoLExpr -> NoLExpr                  {right, "InfixExpr"}
}
> {
  NoLExpr CMPR-OP NoLExpr         -> NoLExpr                   {left, "InfixExpr"}
  NoLExpr RASSOC-CMPR-OP NoLExpr -> NoLExpr                   {right, "InfixExpr"}
}
> {
  NoLExpr BRACKET-OP NoLExpr         -> NoLExpr                {left, "InfixExpr"}
  NoLExpr RASSOC-BRACKET-OP NoLExpr -> NoLExpr                {right, "InfixExpr"}
}
> {
  NoLExpr AMPERSAND-OP NoLExpr         -> NoLExpr              {left, "InfixExpr"}
  NoLExpr RASSOC-AMPERSAND-OP NoLExpr -> NoLExpr              {right, "InfixExpr"}
}
> {
  NoLExpr CIRCUMFLEX-OP NoLExpr         -> NoLExpr             {left, "InfixExpr"}
  NoLExpr RASSOC-CIRCUMFLEX-OP NoLExpr -> NoLExpr             {right, "InfixExpr"}
}
> {
  NoLExpr BAR-OP NoLExpr         -> NoLExpr                    {left, "InfixExpr"}
  NoLExpr RASSOC-BAR-OP NoLExpr -> NoLExpr                    {right, "InfixExpr"}
}
> {
  NoLExpr LETTER-OP NoLExpr         -> NoLExpr                 {left, "InfixExpr"}
  NoLExpr RASSOC-LETTER-OP NoLExpr -> NoLExpr                 {right, "InfixExpr"}
}
> NoLExpr ASSIGN-OP NoLExpr      -> NoLExpr                    {left, "InfixExpr"}
> NoLExpr Id -> NoLExpr                                     {avoid, "PostfixExpr"}
> NoLExpr Id -> NoLExpr                                     {avoid, "PostfixExpr"}
> NoLExpr "match" "{" CaseClause+ "}" -> NoLExpr                    {"MatchExpr"}
> NoLExpr ArgumentExprs "=" NoLExpr    -> NoLExpr       {"AccessAssignmentExpr",
                                                                      non-assoc}
```

```
<0> > NoLExpr "." Id "=" NoLExpr    -> NoLExpr      {"DesignatorAssignmentExpr",
                                                                     non-assoc}
<0> > {
  Bindings "=>" NoLExpr                -> NoLExpr                {prefer, "FunExpr"}
  Id "=>" NoLExpr                      -> NoLExpr              {prefer, "IdFunExpr"}
  "implicit" Id "=>" NoLExpr           -> NoLExpr        {prefer, "ImplicitFunExpr"}
  "_" "=>" NoLExpr                     -> NoLExpr        {prefer, "WildcardFunExpr"}
  "if" "(" NoLExpr ")" NoLExpr         -> NoLExpr                 {prefer, "IfExpr"}
  "if" "(" NoLExpr ")" NoLExpr
    "else" NoLExpr                     -> NoLExpr                     {"IfElseExpr"}
  "if" "(" NoLExpr ")" NoLExpr
    ";" "else" NoLExpr                 -> NoLExpr                     {"IfElseExpr"}
  "while" "(" NoLExpr ")" NoLExpr      -> NoLExpr                      {"WhileExpr"}
  "try" NoLExpr                        -> NoLExpr                {prefer, "TryExpr"}
  "try" NoLExpr "catch" NoLExpr        -> NoLExpr                   {"TryCatchExpr"}
  "try" NoLExpr "finally" NoLExpr      -> NoLExpr                 {"TryFinallyExpr"}
  "try" NoLExpr "catch" NoLExpr
    "finally" NoLExpr                  -> NoLExpr   {avoid, "TryCatchFinallyExpr"}
  "do" NoLExpr ";" "while" "(" NoLExpr ")"      -> NoLExpr          {"DoWhileExpr"}
  "do" NoLExpr "while" "(" NoLExpr ")"          -> NoLExpr          {"DoWhileExpr"}
  "for" "(" EnumeratorSemi+ ")" NoLExpr         -> NoLExpr              {"ForExpr"}
  "for" "{" EnumeratorSemi+ "}" NoLExpr         -> NoLExpr              {"ForExpr"}
  "for" "(" EnumeratorSemi+ ")" "yield" NoLExpr -> NoLExpr         {"ForYieldExpr"}
  "for" "{" EnumeratorSemi+ "}" "yield" NoLExpr -> NoLExpr         {"ForYieldExpr"}
  "throw" NoLExpr                               -> NoLExpr            {"ThrowExpr"}
  "return" NoLExpr?                             -> NoLExpr           {"ReturnExpr"}
  Id "=" NoLExpr                           -> NoLExpr {non-assoc, "AssignmentExpr"}
  NoLExpr Ascription                       -> NoLExpr     {avoid, "AscriptionExpr"}
}

context-free priorities
NoLExpr "." Id   -> NoLExpr                                {avoid, "DesignatorExpr"}
> PREFIX NoLExpr -> NoLExpr                                            {"PrefixExpr"}


%%% Repeat Priorities in context-free syntax for correct PPTable generation %%%
context-free syntax
"_"                                         -> Expr               {"WildcardExpr"}
Expr ArgumentExprs                          -> Expr                    {"AppExpr",
                                         layout("1.last.line == 2.first.line")}
Expr BlockExpr                              -> Expr               {"BlockAppExpr",
                                    layout("2.first.line - 1.last.line < num(2)")}
"(" {NoLExpr ","}* ")"                      -> Expr                  {"TupleExpr"}
Expr TypeArgs                               -> Expr            {"TypeApplication"}
"new" ClassTemplate                         -> Expr      {prefer, "NewClassExpr"}
"new" TemplateBody                          -> Expr   {prefer, "NewTemplateExpr"}
Expr "_"                                    -> Expr    {left, "EtaExpansionExpr",
                                         layout("1.last.line == 2.first.line")}
PREFIX Expr                                 -> Expr        {prefer, "PrefixExpr"}
Expr SPECIAL-OP Expr                        -> Expr           {left, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
Expr MULT-OP Expr                           -> Expr           {left, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
Expr SUM-OP Expr                            -> Expr           {left, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
```

```
Expr COLON-OP Expr                                 -> Expr           {left, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
Expr CMPR-OP Expr                                  -> Expr           {left, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
Expr BRACKET-OP Expr                               -> Expr           {left, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
Expr AMPERSAND-OP Expr                             -> Expr           {left, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
Expr CIRCUMFLEX-OP Expr                            -> Expr           {left, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
Expr BAR-OP Expr                                   -> Expr           {left, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
Expr LETTER-OP Expr                                -> Expr           {left, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
Expr RASSOC-SPECIAL-OP Expr                        -> Expr          {right, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
Expr RASSOC-MULT-OP Expr                           -> Expr          {right, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
Expr RASSOC-SUM-OP Expr                            -> Expr          {right, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
Expr RASSOC-COLON-OP Expr                          -> Expr          {right, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
Expr RASSOC-CMPR-OP Expr                           -> Expr          {right, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
Expr RASSOC-BRACKET-OP Expr                        -> Expr          {right, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
Expr RASSOC-AMPERSAND-OP Expr                      -> Expr          {right, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
Expr RASSOC-CIRCUMFLEX-OP Expr                     -> Expr          {right, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
Expr RASSOC-BAR-OP Expr                            -> Expr          {right, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
Expr RASSOC-LETTER-OP Expr                         -> Expr          {right, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
Expr ASSIGN-OP Expr                                -> Expr           {left, "InfixExpr",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
Expr Id                                            -> Expr        {avoid, "PostfixExpr",
                                                      layout("1.last.line == 2.first.line")}
Expr "match" "{" CaseClause+ "}" -> Expr                                   {"MatchExpr"}
Expr ArgumentExprs "=" Expr      -> Expr      {non-assoc, "AccessAssignmentExpr"}
Expr "." Id "=" Expr             -> Expr {non-assoc, "DesignatorAssignmentExpr"}
Bindings "=>" Expr               -> Expr                           {prefer, "FunExpr"}
Id "=>" Expr                     -> Expr                         {prefer, "IdFunExpr"}
"implicit" Id "=>" Expr          -> Expr                   {prefer, "ImplicitFunExpr"}
"_" "=>" Expr                    -> Expr                   {prefer, "WildcardFunExpr"}
"if" "(" NoLExpr ")" Expr        -> Expr                            {prefer, "IfExpr"}
"if" "(" NoLExpr ")" Expr
  "else" Expr                    -> Expr                                {"IfElseExpr"}
"if" "(" NoLExpr ")" Expr ";"
  "else" Expr                    -> Expr                                {"IfElseExpr"}
"while" "(" NoLExpr ")" Expr            -> Expr                          {"WhileExpr"}
"try" Expr                              -> Expr                   {prefer, "TryExpr"}
"try" Expr "catch" Expr                 -> Expr                       {"TryCatchExpr"}
"try" Expr "finally" Expr               -> Expr                     {"TryFinallyExpr"}
"try" Expr "catch" Expr "finally" Expr  -> Expr   {avoid, "TryCatchFinallyExpr"}
```

```
"do" Expr ";" "while" "(" Expr ")"      -> Expr             {"DoWhileExpr"}
"do" Expr "while" "(" Expr ")"          -> Expr             {"DoWhileExpr"}
"for" "(" EnumeratorSemi+ ")" Expr         -> Expr             {"ForExpr"}
"for" "{" EnumeratorSemi+ "}" Expr         -> Expr             {"ForExpr"}
"for" "(" EnumeratorSemi+ ")" "yield" Expr -> Expr        {"ForYieldExpr"}
"for" "{" EnumeratorSemi+ "}" "yield" Expr -> Expr        {"ForYieldExpr"}
"throw" Expr                               -> Expr           {"ThrowExpr"}
"return" Expr?                             -> Expr          {"ReturnExpr",
                                         layout("1.last.line == 2.first.line")}
Id "=" Expr                              -> Expr   {non-assoc, "AssignmentExpr"}
Expr Ascription                          -> Expr       {avoid, "AscriptionExpr"}
Expr "." Id                              -> Expr       {avoid, "DesignatorExpr"}
PREFIX Expr                              -> Expr                {"PrefixExpr"}

"_"                                    -> NoLExpr              {"WildcardExpr"}
Literal                                -> NoLExpr                      {prefer}
NoLExpr ArgumentExprs                  -> NoLExpr                   {"AppExpr"}
NoLExpr BlockExpr                      -> NoLExpr              {"BlockAppExpr"}
"(" {NoLExpr ","}* ")"                 -> NoLExpr                 {"TupleExpr"}
NoLExpr TypeArgs                       -> NoLExpr           {"TypeApplication"}
"new" ClassTemplate                    -> NoLExpr     {prefer, "NewClassExpr"}
"new" TemplateBody                     -> NoLExpr  {prefer, "NewTemplateExpr"}
NoLExpr "_"                            -> NoLExpr  {left, "EtaExpansionExpr"}
PREFIX NoLExpr                         -> NoLExpr        {prefer, "PrefixExpr"}
NoLExpr SPECIAL-OP NoLExpr             -> NoLExpr          {left, "InfixExpr"}
NoLExpr MULT-OP NoLExpr                -> NoLExpr          {left, "InfixExpr"}
NoLExpr SUM-OP NoLExpr                 -> NoLExpr          {left, "InfixExpr"}
NoLExpr COLON-OP NoLExpr               -> NoLExpr          {left, "InfixExpr"}
NoLExpr CMPR-OP NoLExpr                -> NoLExpr          {left, "InfixExpr"}
NoLExpr BRACKET-OP NoLExpr             -> NoLExpr          {left, "InfixExpr"}
NoLExpr AMPERSAND-OP NoLExpr           -> NoLExpr          {left, "InfixExpr"}
NoLExpr CIRCUMFLEX-OP NoLExpr          -> NoLExpr          {left, "InfixExpr"}
NoLExpr BAR-OP NoLExpr                 -> NoLExpr          {left, "InfixExpr"}
NoLExpr LETTER-OP NoLExpr              -> NoLExpr          {left, "InfixExpr"}
NoLExpr RASSOC-SPECIAL-OP NoLExpr      -> NoLExpr         {right, "InfixExpr"}
NoLExpr RASSOC-MULT-OP NoLExpr         -> NoLExpr         {right, "InfixExpr"}
NoLExpr RASSOC-SUM-OP NoLExpr          -> NoLExpr         {right, "InfixExpr"}
NoLExpr RASSOC-COLON-OP NoLExpr        -> NoLExpr         {right, "InfixExpr"}
NoLExpr RASSOC-CMPR-OP NoLExpr         -> NoLExpr         {right, "InfixExpr"}
NoLExpr RASSOC-BRACKET-OP NoLExpr      -> NoLExpr         {right, "InfixExpr"}
NoLExpr RASSOC-AMPERSAND-OP NoLExpr    -> NoLExpr         {right, "InfixExpr"}
NoLExpr RASSOC-CIRCUMFLEX-OP NoLExpr   -> NoLExpr         {right, "InfixExpr"}
NoLExpr RASSOC-BAR-OP NoLExpr          -> NoLExpr         {right, "InfixExpr"}
NoLExpr RASSOC-LETTER-OP NoLExpr       -> NoLExpr         {right, "InfixExpr"}
NoLExpr ASSIGN-OP NoLExpr              -> NoLExpr          {left, "InfixExpr"}
NoLExpr Id                             -> NoLExpr       {avoid, "PostfixExpr"}
NoLExpr Id                             -> NoLExpr       {avoid, "PostfixExpr"}
NoLExpr "match" "{" CaseClause+ "}"    -> NoLExpr                 {"MatchExpr"}
NoLExpr ArgumentExprs "=" NoLExpr      -> NoLExpr     {"AccessAssignmentExpr",
                                                                   non-assoc}
NoLExpr "." Id "=" NoLExpr             -> NoLExpr  {"DesignatorAssignmentExpr",
                                                                   non-assoc}
Bindings "=>" NoLExpr                  -> NoLExpr           {prefer, "FunExpr"}
Id "=>" NoLExpr                        -> NoLExpr         {prefer, "IdFunExpr"}
"implicit" Id "=>" NoLExpr             -> NoLExpr   {prefer, "ImplicitFunExpr"}
```

```
"_" "=>" NoLExpr                         -> NoLExpr    {prefer, "WildcardFunExpr"}
"if" "(" NoLExpr ")" NoLExpr                        -> NoLExpr    {prefer, "IfExpr"}
"if" "(" NoLExpr ")" NoLExpr "else" NoLExpr         -> NoLExpr        {"IfElseExpr"}
"if" "(" NoLExpr ")" NoLExpr ";" "else" NoLExpr     -> NoLExpr        {"IfElseExpr"}
"while" "(" NoLExpr ")" NoLExpr                     -> NoLExpr         {"WhileExpr"}
"try" NoLExpr                                  -> NoLExpr               {"TryExpr"}
"try" NoLExpr "catch" NoLExpr                  -> NoLExpr          {"TryCatchExpr"}
"try" NoLExpr "finally" NoLExpr                -> NoLExpr        {"TryFinallyExpr"}
"try" NoLExpr "catch" NoLExpr
  "finally" NoLExpr                            -> NoLExpr   {"TryCatchFinallyExpr"}
"do" NoLExpr ";" "while" "(" NoLExpr ")"         -> NoLExpr           {"DoWhileExpr"}
"do" NoLExpr "while" "(" NoLExpr ")"             -> NoLExpr           {"DoWhileExpr"}
"for" "(" EnumeratorSemi+ ")" NoLExpr            -> NoLExpr               {"ForExpr"}
"for" "{" EnumeratorSemi+ "}" NoLExpr            -> NoLExpr               {"ForExpr"}
"for" "(" EnumeratorSemi+ ")" "yield" NoLExpr -> NoLExpr          {"ForYieldExpr"}
"for" "{" EnumeratorSemi+ "}" "yield" NoLExpr -> NoLExpr          {"ForYieldExpr"}
"throw" NoLExpr                                  -> NoLExpr             {"ThrowExpr"}
"return" NoLExpr?                                -> NoLExpr            {"ReturnExpr",
                                          layout("1.last.line == 2.first.line")}
NoLExpr Ascription                       -> NoLExpr       {avoid, "AscriptionExpr"}
Id "=" NoLExpr                           -> NoLExpr   {non-assoc, "AssignmentExpr"}
NoLExpr "." Id                           -> NoLExpr       {avoid, "DesignatorExpr"}
PREFIX NoLExpr                           -> NoLExpr                  {"PrefixExpr"}


%%%%% Common %%%%%%
context-free syntax
"(" NoLExprs? ")"                              -> ArgumentExprs   {"ArgumentExprs"}
"(" (NoLExprs ",")? NoLExpr ":" "_" "*" ")" -> ArgumentExprs
                                               {prefer, "SequenceArgumentExprs"}

ArgumentExprs                     -> ArgumentExprsSeq
ArgumentExprs ArgumentExprsSeq -> ArgumentExprsSeq           {"ArgumentExprsSeq",
                                          layout("1.last.line == 2.first.line")}

{NoLExpr ","}+ -> NoLExprs                                              {"Exprs"}

":" InfixType    -> Ascription                                 {"TypeAscription"}
":" Annotation+ -> Ascription                            {"AnnotationAscription"}
":" "_" "*"      -> Ascription                             {"SequenceAscription"}

Pattern1 "<-" Expr Guard?       -> Generator                        {"Generator"}
"val" Pattern1 "<-" Expr Guard? -> Generator                        {"Generator"}

"if" NoLExpr -> Guard                                                   {"Guard"}

Enumerator EOL   -> EnumeratorSemi                             {"EnumeratorSemi",
                                         enforce-newline, longest-match, prefer}
Enumerator SEMI -> EnumeratorSemi               {longest-match, "EnumeratorSemi"}
Enumerator EOP   -> EnumeratorSemi       {longest-match, avoid, "EnumeratorSemi"}
Enumerator EOB   -> EnumeratorSemi       {longest-match, avoid, "EnumeratorSemi"}
```

```
Generator                     -> Enumerator
Guard                         -> Enumerator
Pattern1 "=" Expr             -> Enumerator                       {"ValDef"}
"val" Pattern1 "=" Expr       -> Enumerator                       {"ValDef"}

"case" Pattern Guard? "=>" CaseBlock -> CaseClause            {"CaseClause"}

"{" CaseClause+ "}" -> BlockExpr                          {"CaseBlockExpr"}
"{" Block "}"       -> BlockExpr                              {"BlockExpr"}

Block              -> CaseBlock
CaseBlockStatSemi* -> CaseBlock                      {avoid, "CaseBlock"}

BlockStatSemi      -> CaseBlockStatSemi
BlockStat EOC      -> CaseBlockStatSemi          {avoid, "EOCBlockStatSemi"}

BlockStatSemi* ResultExpr -> Block                        {prefer, "Block"}
BlockStatSemi*            -> Block                                {"Block"}

Bindings "=>" Block                      -> ResultExpr   {"BindingsResultExpr"}
"implicit" Id ":" CompoundType "=>" Block -> ResultExpr  {"ImplicitResultExpr"}
Id (":" CompoundType)? "=>" Block        -> ResultExpr     {"SimpleResultExpr"}
"_" (":" CompoundType)? "=>" Block       -> ResultExpr   {"WildcardResultExpr"}

BlockStat EOL   -> BlockStatSemi                          {"BlockStatSemi",
                                        enforce-newline, longest-match, prefer}
BlockStat SEMI  -> BlockStatSemi                 {longest-match, "BlockStatSemi"}
BlockStat EOB   -> BlockStatSemi          {longest-match, avoid, "BlockStatSemi"}
SEMI            -> BlockStatSemi

Import                              -> BlockStat
Expr                                -> BlockStat
Annotation* LocalModifier* TmplDef  -> BlockStat     {prefer, "TmplDefBlockStat"}
Annotation* Def                     -> BlockStat                 {"DefBlockStat"}
Annotation* "implicit" Def          -> BlockStat         {"ImplicitDefBlockStat"}
Annotation* "lazy" Def              -> BlockStat             {"LazyDefBlockStat"}

TemplateStat SEMI -> TemplateStatSemi        {longest-match, "TemplateStatSemi"}
TemplateStat EOL  -> TemplateStatSemi                     {"TemplateStatSemi",
                                        enforce-newline, longest-match, prefer}
TemplateStat EOB  -> TemplateStatSemi {longest-match, avoid, "TemplateStatSemi"}
SEMI              -> TemplateStatSemi

AnnotationSeq? Modifier* Def -> TemplateStat                 {"DefTemplateStat"}
AnnotationSeq? Modifier* Dcl -> TemplateStat                 {"DclTemplateStat"}
Import                       -> TemplateStat              {"ImportTemplateStat"}
Expr                         -> TemplateStat                {"ExprTemplateStat"}

Id (":" Type)?  -> Binding                                          {"Binding"}
"_" (":" Type)? -> Binding                                  {"WildCardBinding"}

"(" {Binding ","}* ")" -> Bindings                                 {"Bindings"}

"=" Expr -> Assignment                                           {"Assignment"}
```

```
%%% Scala-Identifiers.sdf %%%
module Scala-Identifiers

exports

lexical syntax
[A-Z] | [\$]    -> UPPER
[a-z]           -> LOWER
UPPER | LOWER   -> LETTER
[0-9]           -> DIGIT
LETTER | DIGIT -> ID-REST

[\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~] -> OP-CHAR

%% OPerators by precedence from low to high
OP-CHAR* [\=]                  -> ASSIGN-OP
VAR-PLAIN-ID | CONST-PLAIN-ID -> LETTER-OP
[\|] OP-CHAR*                  -> BAR-OP
[\^] OP-CHAR*                  -> CIRCUMFLEX-OP
[\&] OP-CHAR*                  -> AMPERSAND-OP
([\<] | [\>]) OP-CHAR*         -> BRACKET-OP
([\=] | [\!]) OP-CHAR*         -> CMPR-OP
[\:] OP-CHAR*                  -> COLON-OP
([\+] | [\-]) OP-CHAR*         -> SUM-OP
([\*] | [\/] | [\%]) OP-CHAR* -> MULT-OP
[\#\?\@\\\~] OP-CHAR*          -> SPECIAL-OP

%% right associative operators
%% TODO: Make RASSOC-LETTER-OP actually work
%%      Won't do at the current state, as of restrictions on OP
(VAR-PLAIN-ID | CONST-PLAIN-ID) [\:] -> RASSOC-LETTER-OP
[\|] OP-CHAR* [\:]                   -> RASSOC-BAR-OP
[\^] OP-CHAR* [\:]                   -> RASSOC-CIRCUMFLEX-OP
[\&] OP-CHAR* [\:]                   -> RASSOC-AMPERSAND-OP
([\<] | [\>]) OP-CHAR* [\:]          -> RASSOC-BRACKET-OP
([\=] | [\!]) OP-CHAR* [\:]          -> RASSOC-CMPR-OP
[\:] OP-CHAR* [\:]                   -> RASSOC-COLON-OP
([\+] | [\-]) OP-CHAR* [\:]          -> RASSOC-SUM-OP
([\*] | [\/] | [\%]) OP-CHAR* [\:]   -> RASSOC-MULT-OP
[\#\?\@\\\~] OP-CHAR* [\:]           -> RASSOC-SPECIAL-OP

[\=] OP-CHAR* [\=] -> ASSIGN-OP {reject}
"="  -> ASSIGN-OP  {reject}
"<=" -> ASSIGN-OP  {reject}
">=" -> ASSIGN-OP  {reject}
"!=" -> ASSIGN-OP  {reject}
"="  -> CMPR-OP    {reject}
"=>" -> CMPR-OP    {reject}
":"  -> COLON-OP   {reject}
"<-" -> BRACKET-OP {reject}
"<:" -> BRACKET-OP {reject}
"<:" -> RASSOC-BRACKET-OP {reject}
"<%" -> BRACKET-OP {reject}
">:" -> BRACKET-OP {reject}
```

```
">:" -> RASSOC-BRACKET-OP {reject}
"#"  -> SPECIAL-OP {reject}
"@"  -> SPECIAL-OP {reject}

%% ASSIGN-OP is exception and thus more important than the other ops
ASSIGN-OP -> LETTER-OP     {reject}
ASSIGN-OP -> BAR-OP        {reject}
ASSIGN-OP -> CIRCUMFLEX-OP {reject}
ASSIGN-OP -> AMPERSAND-OP  {reject}
ASSIGN-OP -> BRACKET-OP    {reject}
ASSIGN-OP -> CMPR-OP       {reject}
ASSIGN-OP -> COLON-OP      {reject}
ASSIGN-OP -> SUM-OP        {reject}
ASSIGN-OP -> MULT-OP       {reject}
ASSIGN-OP -> SPECIAL-OP    {reject}

%% Right associative identifiers have higher priority than
%% their left associative counterparts
RASSOC-LETTER-OP     -> LETTER-OP     {reject}
RASSOC-BAR-OP        -> BAR-OP        {reject}
RASSOC-CIRCUMFLEX-OP -> CIRCUMFLEX-OP {reject}
RASSOC-AMPERSAND-OP  -> AMPERSAND-OP  {reject}
RASSOC-BRACKET-OP    -> BRACKET-OP    {reject}
RASSOC-CMPR-OP       -> CMPR-OP       {reject}
RASSOC-COLON-OP      -> COLON-OP      {reject}
RASSOC-SUM-OP        -> SUM-OP        {reject}
RASSOC-MULT-OP       -> MULT-OP       {reject}
RASSOC-SPECIAL-OP    -> SPECIAL-OP    {reject}

OP-CHAR+ -> OP

LOWER                            -> IVAR-ID
(IVAR-ID | IVAR-ID-USS) ID-REST  -> IVAR-ID
(IVAR-ID | IVAR-ID-USS) [\_]     -> IVAR-ID-USS
IVAR-ID-USS OP                   -> IVAR-ID-OP

[\_]                                 -> ICONST-ID
UPPER                                -> ICONST-ID
(ICONST-ID | ICONST-ID-USS) ID-REST  -> ICONST-ID
(ICONST-ID | ICONST-ID-USS) [\_]     -> ICONST-ID-USS
ICONST-ID-USS [\_]                   -> ICONST-ID-USS
ICONST-ID-USS OP                     -> ICONST-ID-OP

IVAR-ID     -> VAR-ID
IVAR-ID-USS -> VAR-ID-USS
IVAR-ID-OP  -> VAR-ID-OP

ICONST-ID     -> CONST-ID
ICONST-ID-USS -> CONST-ID-USS
ICONST-ID-OP  -> CONST-ID-OP

(VAR-ID   | VAR-ID-USS   | VAR-ID-OP)   -> IVAR-PLAIN-ID
(CONST-ID | CONST-ID-USS | CONST-ID-OP) -> ICONST-PLAIN-ID
```

```
OP               -> IPLAIN-ID
IVAR-PLAIN-ID    -> IPLAIN-ID
ICONST-PLAIN-ID  -> IPLAIN-ID

IVAR-PLAIN-ID    -> VAR-PLAIN-ID
ICONST-PLAIN-ID  -> CONST-PLAIN-ID
IPLAIN-ID        -> PLAIN-ID

"abstract"  -> KEYWORD
"case"      -> KEYWORD
"catch"     -> KEYWORD
"class"     -> KEYWORD
"def"       -> KEYWORD
"do"        -> KEYWORD
"else"      -> KEYWORD
"extends"   -> KEYWORD
"false"     -> KEYWORD
"final"     -> KEYWORD
"finally"   -> KEYWORD
"for"       -> KEYWORD
"forSome"   -> KEYWORD
"if"        -> KEYWORD
"implicit"  -> KEYWORD
"import"    -> KEYWORD
"lazy"      -> KEYWORD
"macro"     -> KEYWORD
"match"     -> KEYWORD
"new"       -> KEYWORD
"null"      -> KEYWORD
"object"    -> KEYWORD
"override"  -> KEYWORD
"package"   -> KEYWORD
"private"   -> KEYWORD
"protected" -> KEYWORD
"return"    -> KEYWORD
"sealed"    -> KEYWORD
"super"     -> KEYWORD
"this"      -> KEYWORD
"throw"     -> KEYWORD
"trait"     -> KEYWORD
"try"       -> KEYWORD
"true"      -> KEYWORD
"type"      -> KEYWORD
"val"       -> KEYWORD
"var"       -> KEYWORD
"while"     -> KEYWORD
"with"      -> KEYWORD
"yield"     -> KEYWORD
"_"         -> KEYWORD
":"         -> KEYWORD
"="         -> KEYWORD
"=>"        -> KEYWORD
"<-"        -> KEYWORD
"<:"        -> KEYWORD
"<%"        -> KEYWORD
```

```
">:"          -> KEYWORD
"#"           -> KEYWORD
"@"           -> KEYWORD

KEYWORD   -> VAR-PLAIN-ID   {reject}
KEYWORD   -> CONST-PLAIN-ID {reject}
KEYWORD   -> PLAIN-ID       {reject}

"-"           -> MINUS-PREFIX
"+"           -> PREFIX
MINUS-PREFIX  -> PREFIX
"~"           -> PREFIX
"!"           -> PREFIX

"`" ~[\`]+ "`" -> FANCY-ID
FANCY-ID       -> PLAIN-ID

lexical restrictions
OP -/- [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]

ASSIGN-OP       -/- [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]
LETTER-OP       -/- [a-zA-Z0-9]
BAR-OP          -/- [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]
CIRCUMFLEX-OP   -/- [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]
AMPERSAND-OP    -/- [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]
BRACKET-OP      -/- [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]
CMPR-OP         -/- [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]
COLON-OP        -/- [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]
SUM-OP          -/- [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]
MULT-OP         -/- [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]
SPECIAL-OP      -/- [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]

RASSOC-LETTER-OP       -/- [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]
RASSOC-BAR-OP          -/- [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]
RASSOC-CIRCUMFLEX-OP   -/- [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]
RASSOC-AMPERSAND-OP    -/- [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]
RASSOC-BRACKET-OP      -/- [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]
RASSOC-CMPR-OP         -/- [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]
RASSOC-COLON-OP        -/- [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]
RASSOC-SUM-OP          -/- [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]
RASSOC-MULT-OP         -/- [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]
RASSOC-SPECIAL-OP      -/- [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]

VAR-ID    -/- [A-Za-z0-9\$\_]
CONST-ID  -/- [A-Za-z0-9\$\_]

VAR-ID-USS    -/- [A-Za-z0-9\$\_] \/ [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]
CONST-ID-USS  -/- [A-Za-z0-9\$\_] \/ [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]
```

```
"abstract" -/- [a-zA-Z0-9\$\_]
"case"     -/- [a-zA-Z0-9\$\_]
"catch"    -/- [a-zA-Z0-9\$\_]
"class"    -/- [a-zA-Z0-9\$\_]
"def"      -/- [a-zA-Z0-9\$\_]
"do"       -/- [a-zA-Z0-9\$\_]
"else"     -/- [a-zA-Z0-9\$\_]
"extends"  -/- [a-zA-Z0-9\$\_]
"false"    -/- [a-zA-Z0-9\$\_]
"final"    -/- [a-zA-Z0-9\$\_]
"finally"  -/- [a-zA-Z0-9\$\_]
"for"      -/- [a-zA-Z0-9\$\_]
"forSome"  -/- [a-zA-Z0-9\$\_]
"if"       -/- [a-zA-Z0-9\$\_]
"implicit" -/- [a-zA-Z0-9\$\_]
"import"   -/- [a-zA-Z0-9\$\_]
"lazy"     -/- [a-zA-Z0-9\$\_]
"macro"    -/- [a-zA-Z0-9\$\_]
"match"    -/- [a-zA-Z0-9\$\_]
"new"      -/- [a-zA-Z0-9\$\_]
"null"     -/- [a-zA-Z0-9\$\_]
"object"   -/- [a-zA-Z0-9\$\_]
"override" -/- [a-zA-Z0-9\$\_]
"package"  -/- [a-zA-Z0-9\$\_]
"private"  -/- [a-zA-Z0-9\$\_]
"protected"-/- [a-zA-Z0-9\$\_]
"return"   -/- [a-zA-Z0-9\$\_]
"sealed"   -/- [a-zA-Z0-9\$\_]
"super"    -/- [a-zA-Z0-9\$\_]
"this"     -/- [a-zA-Z0-9\$\_]
"throw"    -/- [a-zA-Z0-9\$\_]
"trait"    -/- [a-zA-Z0-9\$\_]
"try"      -/- [a-zA-Z0-9\$\_]
"true"     -/- [a-zA-Z0-9\$\_]
"type"     -/- [a-zA-Z0-9\$\_]
"val"      -/- [a-zA-Z0-9\$\_]
"var"      -/- [a-zA-Z0-9\$\_]
"while"    -/- [a-zA-Z0-9\$\_]
"with"     -/- [a-zA-Z0-9\$\_]
"yield"    -/- [a-zA-Z0-9\$\_]

"=" -/- [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]
":" -/- [\!\#\%\&\*\+\-\/\:\<\=\>\?\@\\\^\|\~]
"_" -/- [a-zA-Z0-9\_\$]

PREFIX -/- [\+\-\~\!]
MINUS-PREFIX -/- [0-9]

FANCY-ID -/- [a-zA-Z0-9\_\$]

context-free syntax
PLAIN-ID -> Id {"Id"}
```

```
%%% Scala-Literals.sdf %%%
module Scala-Literals

imports
Scala-Identifiers

exports

lexical syntax
DECIMAL-NUMERAL ("L" | "l")? -> INTEGER
HEX-NUMERAL ("L" | "l")?     -> INTEGER
OCTAL-NUMERAL ("L" | "l")?   -> INTEGER                          {prefer}
"-" INTEGER                  -> INTEGER

[0] | NON-ZERO-DIGIT DIGIT* -> DECIMAL-NUMERAL
"0x" HEX-DIGIT+             -> HEX-NUMERAL
[0] OCTAL-DIGIT+            -> OCTAL-NUMERAL
[0-9]       -> DIGIT
[1-9]       -> NON-ZERO-DIGIT
[0-7]       -> OCTAL-DIGIT
[0-9A-Fa-f] -> HEX-DIGIT

DIGIT+ "." DIGIT+ EXPONENT-PART? FLOAT-TYPE? -> FLOATING-POINT
"." DIGIT+ EXPONENT-PART? FLOAT-TYPE?        -> FLOATING-POINT
DIGIT+ EXPONENT-PART                         -> FLOATING-POINT
DIGIT+ FLOAT-TYPE                            -> FLOATING-POINT
DIGIT+ EXPONENT-PART FLOAT-TYPE              -> FLOATING-POINT
"-" FLOATING-POINT                           -> FLOATING-POINT

("E" | "e") ("+" | "-")? DIGIT+ -> EXPONENT-PART
"F" | "f" | "D" | "d"           -> FLOAT-TYPE

"'" PRINTABLE "'"       -> CHAR
"'" CHAR-ESCAPE-SEQ "'" -> CHAR

[\32-\126]        -> PRINTABLE
[\0-\127] / [\"]  -> CHAR-NO-DOUBLE-QUOTE

"\\b" | "\\t" | "\\n" | "\\f" |
"\\r" | "\\\"" | "\\'" | "\\\\" -> CHAR-ESCAPE-SEQ
"\\" [0-1]? [0-9]? [0-9]        -> CHAR-ESCAPE-SEQ
"\\" DIGIT? DIGIT? DIGIT        -> CHAR-ESCAPE-SEQ
UNICODE-ESCAPE                  -> CHAR-ESCAPE-SEQ

"\\" "u" HEX-DIGIT HEX-DIGIT HEX-DIGIT HEX-DIGIT -> UNICODE-ESCAPE

"\"" STRING-ELEMENT* "\""          -> STRING
"\"\"\"" MULTI-LINE-CHARS "\"\"\"" -> STRING

[\32-\126] / [\"\\]   -> STRING-ELEMENT
CHAR-ESCAPE-SEQ       -> STRING-ELEMENT

PLAIN-ID "\"" PROCESSED-STRING-ELEMENT* "\""                 -> PROCESSED-STRING
PLAIN-ID "\"\"\"" MULTI-LINE-PROCESSED-STRING-ELEMENT "\"\"\""
                                                             -> PROCESSED-STRING
```

```
[\32-\126] / [\"\$\\] -> PROCESSED-STRING-ELEMENT
CHAR-ESCAPE-SEQ       -> PROCESSED-STRING-ELEMENT
PROCESSING            -> PROCESSED-STRING-ELEMENT

"$$"                         -> PROCESSING
"${" PROCESSING-ELEMENT* "}" -> PROCESSING
"$" IPLAIN-ID                -> PROCESSING

"{" PROCESSING-ELEMENT* "}" -> PROCESSING-ELEMENT
[\0-\127] / [\{\}\"]        -> PROCESSING-ELEMENT
PROCESSED-STRING            -> PROCESSING-ELEMENT
STRING                      -> PROCESSING-ELEMENT

([\"]? [\"]? CHAR-NO-DOUBLE-QUOTE)* [\"]* -> MULTI-LINE-CHARS

([\"]? [\"]? IMULTI-LINE-PROCESSED-STRING-ELEMENT)* [\"]*
                                          -> MULTI-LINE-PROCESSED-STRING-ELEMENT

PROCESSING         -> IMULTI-LINE-PROCESSED-STRING-ELEMENT
[\0-\127] / [\"\$] -> IMULTI-LINE-PROCESSED-STRING-ELEMENT

[\"]       -> DOUBLE-QUOTE
[\"] [\"] -> DOUBLE-DOUBLE-QUOTE

"'" IPLAIN-ID -> SYMBOL

lexical restrictions
INTEGER        -/- [0-9a-zA-Z]
FLOATING-POINT -/- [0-9a-zA-Z]
Literal        -/- [0-9a-zA-Z]
"true"         -/- [0-9a-zA-Z]
"false"        -/- [0-9a-zA-Z]
DOUBLE-QUOTE        -/- [\"]
DOUBLE-DOUBLE-QUOTE -/- [\"]

context-free syntax
INTEGER          -> Literal                                    {"Int"}
FLOATING-POINT   -> Literal                                  {"Float"}
CHAR             -> Literal                                   {"Char"}
STRING           -> Literal                                 {"String"}
SYMBOL           -> Literal                                 {"Symbol"}
"null"           -> Literal                                   {"Null"}
PROCESSED-STRING -> Literal                        {"ProcessedString"}
BooleanLiteral   -> Literal

"true"  -> BooleanLiteral                                     {"True"}
"false" -> BooleanLiteral                                    {"False"}
```

```
%%% Scala-Macros.sdf %%%
module Scala-Macros

imports
Scala-BasicDeclsDefs
Scala-Identifiers
Scala-Types

exports

context-free syntax
FunSig (":" Type)? "=" "macro" StableId TypeArgs?         -> FunDef {"MacroDef"}


%%% Scala-PatternMatching.sdf %%%
module Scala-PatternMatching

imports
Scala-Literals
Scala-Types

exports

lexical syntax
PLAIN-ID -> INFIX-PATTERN-OP
"|"      -> INFIX-PATTERN-OP                                               {reject}

context-free priorities
{
  "_"                                           -> SimplePattern    {"WildcardPattern"}
%%VarId                                          -> SimplePattern    {"VariablePattern"}
  Literal                                       -> SimplePattern     {"LiteralPattern"}
  StableId "(" Patterns ")"                     -> SimplePattern {"ConstructorPattern"}
  "(" Patterns ")"                              -> SimplePattern       {"TuplePattern"}
  StableId "(" (Patterns ",")? (VAR-PLAIN-ID "@")? "_" "*" ")"
                                                -> SimplePattern    {"PatternSequence"}
  StableId                                      -> SimplePattern
}

> {
  Pattern3 INFIX-PATTERN-OP Pattern3 -> Pattern3          {left, "InfixPattern",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
  SimplePattern                      -> Pattern3
}

> {
  VAR-PLAIN-ID "@" Pattern3 -> Pattern2                             {"PatternBinder"}
  Pattern3                  -> Pattern2
}

> {
  VAR-PLAIN-ID ":" Type -> Pattern1                                  {"TypedPattern"}
  "_" ":" Type          -> Pattern1                          {"WildcardTypedPattern"}
  Pattern2              -> Pattern1
}
```

```
> {
  Pattern1              -> Pattern                                    {longest-match}
  Pattern "|" Pattern -> Pattern                            {"DisjunctPattern", left}
}

context-free syntax
{Pattern ","}* -> Patterns {"Patterns"}

%%% Repeat priorities in context-free syntax for correct PPTable generation %%%
context-free syntax
"_"                                  -> SimplePattern        {"WildcardPattern"}
Literal                              -> SimplePattern         {"LiteralPattern"}
StableId "(" Patterns ")"            -> SimplePattern     {"ConstructorPattern"}
"(" Patterns ")"                     -> SimplePattern           {"TuplePattern"}
StableId "(" (Patterns ",")?
  (VAR-PLAIN-ID "@")? "_" "*" ")"    -> SimplePattern        {"PatternSequence"}
Pattern3 INFIX-PATTERN-OP Pattern3   -> Pattern3             {left, "InfixPattern",
   layout("1.last.line == 2.first.line && 3.first.line - 2.last.line < num(2)")}
VAR-PLAIN-ID "@" Pattern3            -> Pattern2               {"PatternBinder"}
VAR-PLAIN-ID ":" Type                -> Pattern1                {"TypedPattern"}
"_" ":" Type                         -> Pattern1        {"WildcardTypedPattern"}
Pattern "|" Pattern                  -> Pattern         {left, "DisjunctPattern"}


%%% Scala.sdf %%%
module Scala

imports
Scala-Annotations
Scala-BasicDeclsDefs
Scala-ClassesObjects
Scala-Expressions
Scala-Identifiers
Scala-Literals
Scala-Macros
Scala-PatternMatching
Scala-TopLevelDefinitions
Scala-Types
Scala-Whitespace

exports

context-free start-symbols
Annotation
Block
CompilationUnit
Dcl
Def
Expr
Id
Import
NoLExpr
Path
Pattern
```

```
StableId
TemplateBody
TemplateStat
TmplDef
Type
TypeParamClause


%%% Scala-TopLevelDefinitions.sdf %%%
%% Chapter 9 - Top-Level Definitions
module Scala-TopLevelDefinitions

imports
Scala-BasicDeclsDefs
Scala-ClassesObjects
Scala-Identifiers
Scala-Whitespace

exports

context-free syntax
-> CompilationUnit                                          {"EmptyCompilationUnit"}
PackageDeclarationSemi* TopStatSemi+ -> CompilationUnit         {"CompilationUnit"}

TopStat EOL  -> TopStatSemi                                             {"TopStatSemi",
                                                  enforce-newline, longest-match, prefer}
TopStat SEMI -> TopStatSemi                                             {"TopStatSemi"}
TopStat EOF  -> TopStatSemi                               {longest-match, "TopStatSemi"}
TopStat EOB  -> TopStatSemi                        {longest-match, avoid, "TopStatSemi"}

AnnotationSeq? Modifier* TmplDef -> TopStat                              {"TopTmplDef"}
Import                           -> TopStat
Packaging                        -> TopStat
PackageObject                    -> TopStat

"package" QualId EOL  -> PackageDeclarationSemi                {"PackageDeclaration",
                                                                       enforce-newline}
"package" QualId SEMI -> PackageDeclarationSemi                {"PackageDeclaration"}

{Id "."}+ -> QualId                                                          {"QualId"}

"package" QualId "{" TopStatSemi+ "}" -> Packaging                        {"Packaging",
                                  layout("3.first.line - 2.last.line < num(2)")}

"package" "object" ObjectDef -> PackageObject                         {"PackageObject"}


%%% Scala-Types.sdf %%%
module Scala-Types

imports
Scala-Annotations
Scala-BasicDeclsDefs
Scala-Identifiers
Scala-Whitespace
```

```
exports

%% §3.1 Paths
context-free syntax
Id              -> Path                                                    {prefer}
{PathElem "."}+ -> Path                                                    {"Path"}

PLAIN-ID                              -> PathElem
"this"                                -> PathElem                          {"This"}
"super" ClassQualifier? "." PLAIN-ID  -> PathElem                         {"Super"}

PLAIN-ID                -> StableIdElem
"this"                  -> StableIdElem                             {"StableThis"}
"super" ClassQualifier? -> StableIdElem                            {"StableSuper"}

Id                                 -> StableId
{StableIdElem "."}+ "." PLAIN-ID -> StableId                          {"StableId"}

"[" PLAIN-ID "]" -> ClassQualifier                              {"ClassQualifier"}

PREFIX -> Path {reject} %% not 100% sure this is correct

context-free restrictions
StableId -/- [\.] . [\33-\126] / [\{\,\_]


%% §3.2 - Value Types
context-free syntax
%% §3.2.1 - Singleton Types
Path "." "type"     -> SimpleType                              {"SingletonType"}

%% §3.2.2 - Type Projection
SimpleType "#" Id   -> SimpleType                             {"TypeProjection"}

%% §3.2.3 - Type Designators
StableId            -> SimpleType                                       {"Type"}

%% §3.2.4 - Parameterized Types
SimpleType TypeArgs -> SimpleType                          {"ParameterizedType"}
"[" {Type ","}+ "]" -> TypeArgs                                     {"TypeArgs"}

%% §3.2.5 - Tuple Types
"(" {Type ","}+ ")" -> SimpleType                                  {"TupleType"}

%% §3.2.6 - Annotated Types
SimpleType             -> AnnotType
SimpleType Annotation+ -> AnnotType                                {"AnnotType",
                                          layout("1.last.line == 2.first.line")}
```

```
%% §3.2.7 - Compound Types
AnnotType                    -> CompoundType
Refinement                   -> CompoundType
AnnotType Refinement         -> CompoundType                         {"RefinedType"}
AnnotType With+ Refinement? -> CompoundType                          {"CompoundType",
                                   layout("3.first.line - 2.last.line < num(2)")}

"with" AnnotType -> With                                                    {"With"}

"{" RefineStatSemi* "}" -> Refinement                                 {"Refinement"}

RefineStat EOL  -> RefineStatSemi                                 {"RefineStatSemi",
                                          enforce-newline, longest-match, prefer}
RefineStat SEMI -> RefineStatSemi                {longest-match, "RefineStatSemi"}
RefineStat EOB  -> RefineStatSemi         {longest-match, avoid, "RefineStatSemi"}

Dcl             -> RefineStat
"type" TypeDef -> RefineStat                                      {"TypeRefineStat"}

%% §3.2.8 - Infix Types
InfixType -> Type

CompoundType                   -> InfixType
InfixType PLAIN-ID InfixType -> InfixType                       {left, "InfixType",
                                   layout("3.first.line - 2.last.line < num(2)")}

%% §3.2.9 - Function Types
FunctionArgTypes "=>" Type  -> Type                    {prefer, "FunctionType"}

InfixType                   -> FunctionArgTypes
"(" {ParamType ","}* ")" -> FunctionArgTypes        {prefer, "FunctionArgType"}

%% §3.2.10 - Existential Types
InfixType ExistentialClause -> Type {"ExistentialType"}

"forSome" "{" ExistentialDclSemi+ "}" -> ExistentialClause {"ExistentialClause"}

ExistentialDcl SEMI -> ExistentialDclSemi  {longest-match, "ExistentialDclSemi"}
ExistentialDcl EOL  -> ExistentialDclSemi                 {"ExistentialDclSemi",
                                          enforce-newline, longest-match, prefer}
ExistentialDcl EOB  -> ExistentialDclSemi  {longest-match, "ExistentialDclSemi"}

"type" TypeDcl -> ExistentialDcl                               {"ExistentialType"}
"val"  ValDcl  -> ExistentialDcl                                {"ExistentialVal"}

"_" TypeBounds?     -> SimpleType                          {avoid, "WildcardType"}

">:" Type           -> TypeBounds                               {"LowerTypeBound"}
"<:" Type           -> TypeBounds                               {"UpperTypeBound"}
">:" Type "<:" Type -> TypeBounds                       {"LowerAndUpperTypeBound"}
```

```
%%% Scala-Whitespace.sdf %%%
%% Part of the definitions and restrictions are taken from
%% bobd91's JavaScript syntax
%% (https://github.com/bobd91/sugarjs/blob/master/
%%        language-libraries/javascript/src/org/sugarj/languages/JavaScript.def)

module Scala-Whitespace

exports

lexical syntax
[\ \t\n\r]        -> LAYOUT
"//" ~[\n]* [\n] -> LAYOUT
BLOCK-COMMENT     -> LAYOUT

"/*" BLOCK-COMMENT-PART* "*/" -> BLOCK-COMMENT
~[\/\*]        -> BLOCK-COMMENT-PART
ASTERISK       -> BLOCK-COMMENT-PART
SLASH          -> BLOCK-COMMENT-PART
BLOCK-COMMENT  -> BLOCK-COMMENT-PART

    -> EOL
    -> EOF
    -> EOB
    -> EOP
";" -> SEMI

[\n] -> NL
[\*] -> ASTERISK
[\/] -> SLASH

lexical restrictions
ASTERISK -/- [\/]
SLASH    -/- [\*]

context-free restrictions
LAYOUT? -/- [\ \t\n\r]
LAYOUT? -/- [\/].[\*]
LAYOUT? -/- [\/].[\/]
EOF -/- ~[]
EOB -/- ~[\}]
EOP -/- ~[\)]
```

B Additional Plots for Grammar Evaluation

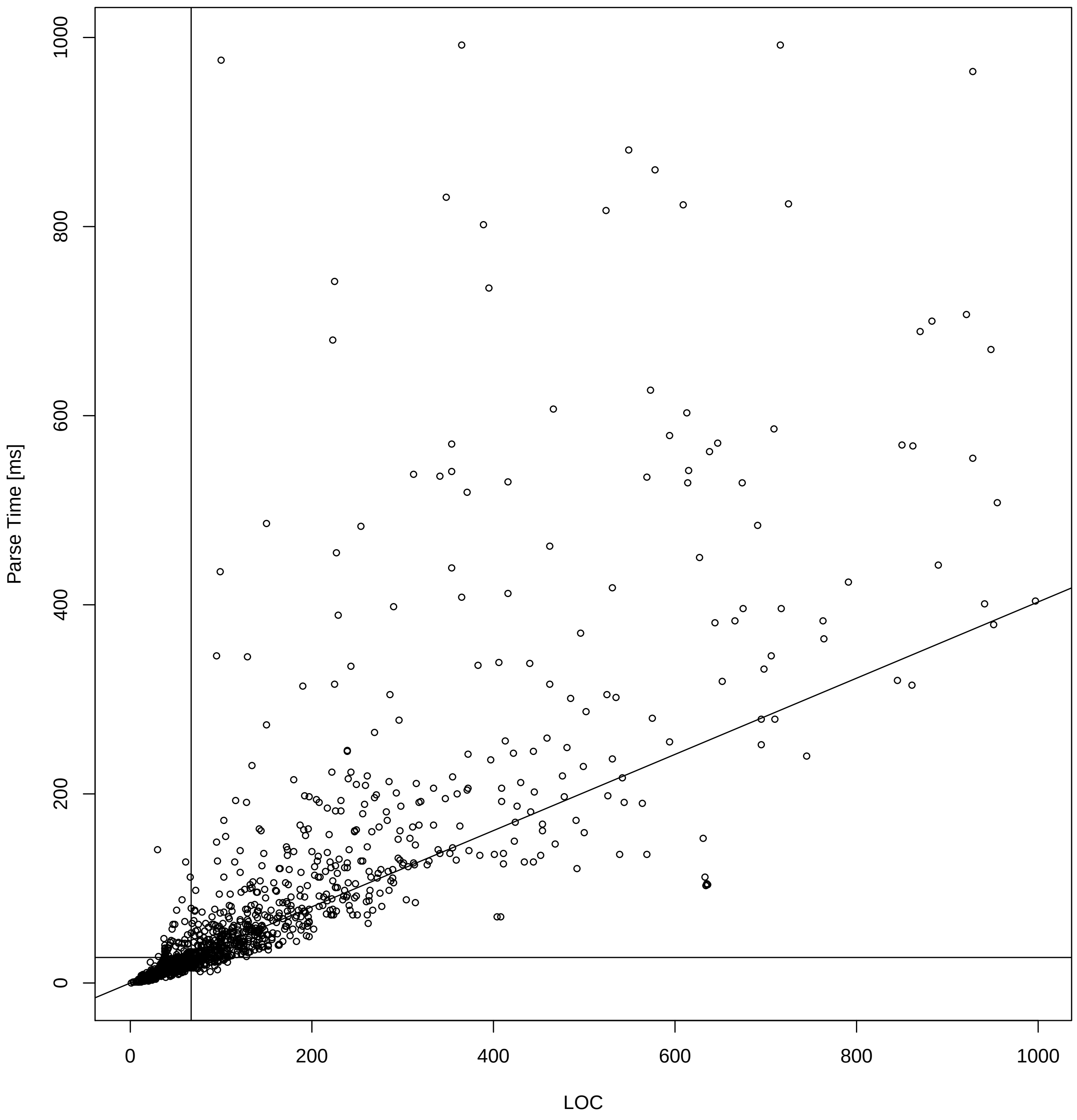


Figure B.1.: Parse times for the 1471 successfully parsed files with LOC ⩽ 1000 and parse time ⩽ 1s — lines mark medians and resulting gradient through origin

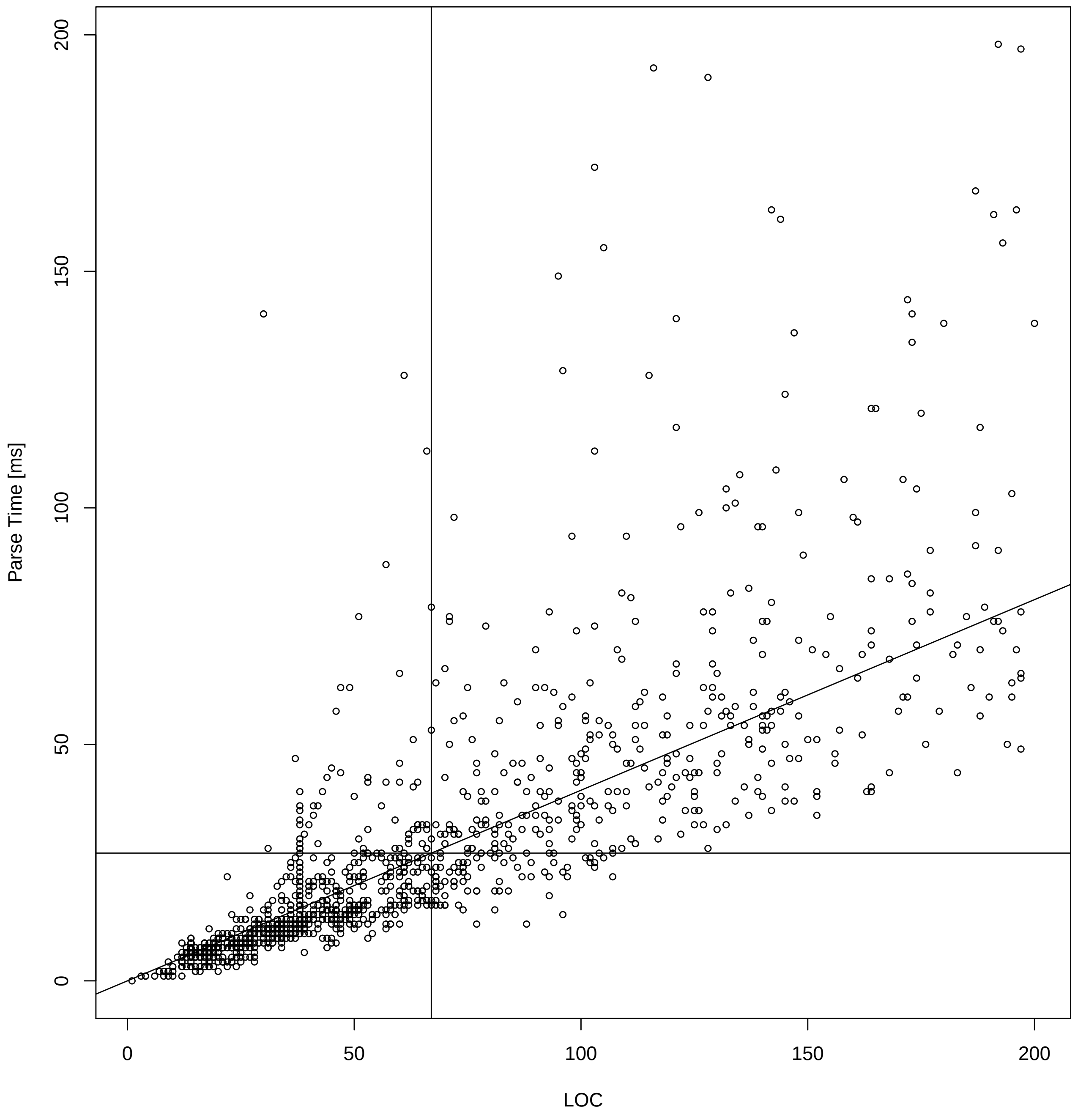


Figure B.2.: Parse times for the 1196 successfully parsed files with LOC $\leqslant$ 200 and parse time $\leqslant$ 200ms — lines mark medians and resulting gradient through origin