TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik

Technische Berichte
Technical Reports

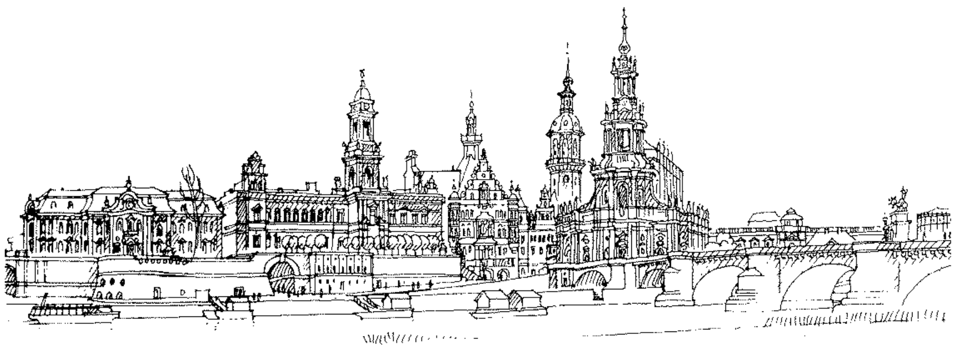
ISSN 1430-211X

TUD-FI10-01-Januar 2010

Sven Karol, Steffen Zschaler

Institut für Software- und Multimediatechnik

Providing Mainstream Parser Generators with Modular Language Definition Support



Technische Universität Dresden
Fakultät Informatik
D−01062 Dresden
Germany
URL: http://www.inf.tu−dresden.de/

Providing Mainstream Parser Generators with Modular Language Definition Support

Sven Karol[1] and Steffen Zschaler[2]

[1] Lehrstuhl Softwaretechnologie, Fakultät Informatik, Technische Universität Dresden
Sven.Karol@inf.tu-dresden.de
[2] Computing Department, Lancaster University
szschaler@acm.org

Abstract. The composition and reuse of existing textual languages is a frequently re-occurring problem. One possibility of composing textual languages lies on the level of parser specifications which are mainly based on context-free grammars and regular expressions. Unfortunately most mainstream parser generators provide proprietary specification languages and usually do not provide strong abstractions for reuse. New forms of parser generators do support modular language development, but they can often not be easily integrated with existing legacy applications. To support modular language development based on mainstream parser generators, in this paper we apply the *Invasive Software Composition* (ISC) paradigm to parser specification languages by using our Reuseware framework. Our approach is grounded on a platform independent metamodel and thus does not rely on a specific parser generator.

1 Introduction

Solving complex problems in software development commonly leads to decomposition into a set of less complex sub problems that can be solved more easily. Solutions to sub problems are encapsulated in components like functions, modules or classes on programming language level and may also be reused as partial solutions in other systems. A re-occurring complex problem in software development is parser construction to provide support for textual languages. Parser generators can substantially reduce complexity through lifting parser construction to a more declarative and abstract level [1]. Instead of implementing parsing methods manually, languages are declared by syntax specifications based on context-free grammars. In most cases, languages are described by a monolithic specification as a parser generator's input source. However, since syntax specifications for large languages can become quite complex, concepts for modularisation on the specification level are desirable. Related parts of syntax specifications then can be decomposed into modules to reduce complexity and to allow reuse of partial solutions in other related problems.

Because of these clear benefits, some recent approaches tackle the support for modular language definitions by using enhanced parsing techniques. We consider scannerless parsing and generalised parsing as the most prominent ones. Section 2 briefly discusses these techniques and compares them to mainstream parser generators.

In this work, we use the term *mainstream* to refer to the group of parser generators with classical strict separation of lexical analysis based on regular expressions and deterministic finite automata (DFA) as well as syntactic analysis based on context-free grammars and deterministic push down automata (PDA) (e.g. LL(k) and LR(k)). Examples are the JavaCC [13] and SableCC [22] parser generators. Mainstream parser generators often lack an appropriate module support. This is mainly due to the restricted grammar classes they can handle and the DFA based lexical analysis phase: Both easily cause ambiguities in composed languages. However, appropriate reuse abstractions can help to resolve or avoid these conflicts. Introducing them to mainstream parser generators would enable their users to specify more reliable grammars, to decompose huge languages into a composite of smaller sublanguages and to easily adapt or extend existing complex language definitions.

Implementing module support for each parser generator from scratch is an elaborate task. Hence, a more generic solution would be beneficial. The idea of this paper is to use a generic parser specification component model as a basis for a generic composition system for parser specification languages. The system provides appropriate composition operators to suffice most use cases, e.g. supporting extension, adaptation and refactorings of existing language specifications. To partially reach these goals, we implement an operator conceptually based on *grammar inheritance* [2]. In addition, we introduce a composition operator for transparent language embedding which is a common task when it comes to the composition of domain specific languages (DSLs). However, embedding operations for mainstream parser generators are problematic: Besides the fact that LL(k) and LR(k) grammar classes are not closed under unification, overlaps between token definitions usually foil modularity. To handle the latter case, we define a composition operator which generates lexer state automata which in fact are supported by many mainstream generators.

We use *Invasive Software Composition* (ISC) as realisation technique for our composition system. For the prototype implementation we apply our Reuseware framework [12]. Reuseware allows to create composition systems based on ISC for any language given by its component model and a form of concrete syntax either textual or graphical.

The paper is structured as follows: Section 2 gives an overview of the most prevalent concepts for modular language specifications. Section 3 discusses requirements for the component model and the composition system. In Section 4, a prototype implementation and applications of the system are presented. Subsequently, Section 5 discusses the usability of the system in comparison to others. Section 6 concludes the paper and gives an outlook.

2 State of the Art in Parser Generation

In contrast to mainstream parser generators, systems with a strong notion of modularity exist. The SDF2/SGLR [25] environment is based on the generalized LR (GLR) parsing technology which allows to generate a parser for any context-free grammar [24]. The basic idea of GLR is to extend the LR(k) formalism such that ambiguities do not result in a parsing error but instead to construct a *parse forest* covering all alternatives to continue the parse. From a compositional point of view, GLR is very practical since context-free grammars are closed under composition, which is not the case for the restricted LL(k) and LR(k) classes. SDF2/SGLR fully integrates lexical and syntactic analysis such that generated parsers directly consume characters from the input stream. This allows to assign equivalent strings with different meaning depending on the productions (i.e. the context) which produce the strings, which clearly is an advantage over mainstream parser generators whose users are often faced with lexical conflicts between token definitions that define non disjoint regular languages. These conflicts are normally solved manually by declaring a precedence order over token definitions or concepts that simulate context like *lexer states*. Thus scannerless parsing allows to compose languages from modules without bothering about intersections between token definitions. The scannerless parser generator $Rats!$ [10] provides strong module support. $Rats!$ can be used to generate backtracking packrat parsers that can cache and reuse intermediate results to converge to linear parsing time at the cost of increased memory consumption. The syntax definition language of $Rats!$ is derived from the parsing expression grammars (PEG) [8] formalism which goes beyond a declarative description of languages, specifying how to parse them. In contrast to context-free grammars derivation semantics, packrat parsers are based on a greedy approach. Thus, even if a grammar is ambigous a packrat parser will at worst find one valid derivation. This is a clear difference in comparison to the *parse forest* output of GLR parsers.

The concept of *context-aware scanning* [27] can be regarded as a compromise between strict separation and full integration of lexical and syntactic analysis phases. In *context-aware scanning*, the token set currently valid is determined by the tokens a parser expects by its look-ahead. In this way the lexer is always informed about context, but lexer and parser are only loosely coupled. Unfortunately, so far, there only seems to be a prototypical implementation by the authors of [27].

However, mainstream parser generators are still in widespread use. There are multiple reasons for this: Their concepts are very common and well known by users, thus the learning effort is relatively small. The generated components are usually fast and relatively compact. Moreover, especially for recursive descent LL(k), the generated parsers are similar to handwritten ones such that debugging is relatively easy, at least for developers that are used to it. Additionally, some look more like programs instead of grammars, e.g. in JavaCC and ANTLR [19] productions are similar to Java methods.

Many mainstream parser generators provide access to rich repositories of predefined grammars, even for complex languages. These grammars already contain solutions to problems related to the base formalisms. A composition system can be applied to enhance reuse of grammars from repositories and legacy software. Existing software based on mainstream parser generators can be developed further in a more modular fashion.

3 Ingredients of the Composition System

According to [3], a well defined composition system consists of three main building blocks:

- The `component model` describes structure, interface and kinds of components. Interfaces allow to hide implementation details and make relevant parts visible to the outside. Different kinds of components may occur on different levels of abstraction.

- The `composition technique` describes how components are brought together. This usually includes a set of composition operators based on a specific implementation technology.
- Composition programs are specified according to a `composition language`. This can be a dedicated language for composition purposes only or an extension to the component language. [11] proposes the latter as a viable solution for providing existing DSLs with module support.

In *Invasive Software Composition,* components are fragments given by a form of abstract syntax or metamodel. To enable extensibility and variability of components, ISC uses the concepts *hook* for extension points and *slot* for variation points. A concrete hook in a fragment is a position that can be safely extended by a compatible fragment. In contrast, slots are placeholders for missing parts of an underspecified component that can be safely bound to a compatible fragment. Fragments are woven based on invasive composition operators. *Extend* and *bind* are the most elementary ones that can be used to write composition programs or to define more complex composition operators (*composers*).

In this paper, we use the Reuseware composition framework [12]. Reuseware is based on the Eclipse Modelling Framework (EMF) [6] and allows to apply the ideas of *Invasive Software Composition* (ISC) [3] to any language given by its metamodel and a form of syntax either textual or graphical.

In the following we conceptually discuss the main parts of the composition system for syntax definitions.

3.1 Component Model

Component models in invasive software composition are fragment based, thus given by a form of abstract syntax or metamodel. To create a generic component model for syntax definitions in mainstream parser generators, we investigated the common concepts of these languages.

A syntax definition normally consists of at least two sections for lexical analysis and syntactic analysis. Lexical analysis is a pre-processing phase and introduces regular expressions and tokens into parser specification languages. Tokens are considered as terminal symbols during syntactic analysis and their value range is determined by a regular language specified by a regular expression. Since the valuations of tokens are not necessarily disjoint, parser specifications normally allow to specify precedence in token definitions. Another supplementary concept to better handle these decision problems is the concept of lexer states as it is supported by tools like JavaCC. The concept is simple: Every lexer state describes a state a scanner can be in and contains a set of tokens being valid in that state. Transitions between lexer states are caused by recognised tokens which have been marked with a follow state in the parser specification. Syntactic analysis is usually based on context-free grammars and Extended Backus Naur Form (EBNF).

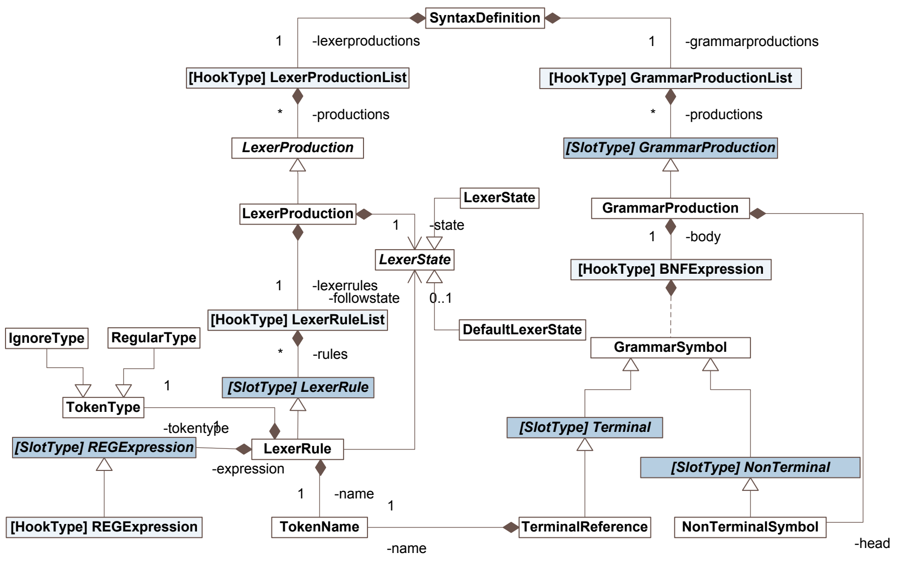


Fig. 1. Language metamodel overview

From these general observations we derived a *fragment component model* which is specified by the basic metamodel depicted in Figure 1 and the actual *fragment box* hierarchy specified in Listing 1. Fragment boxes provide means for accessing fragment components, i.e., they manage a fragment's *hooks* and *slots*, provide access methods for child boxes, ensure type-safe composition and manage persistency.

The component model is generic in the sense that it does not support generator specific constructs. Instead, the model provides concepts that are very likely to occur in mainstream parser specifications. To apply the composition system to a specific parser generator, a mapping to the component model has to be created. We have created such mappings for JavaCC and partially for SableCC. In this paper, we will focus specifically on the JavaCC realisation, which will be discussed in Sect. 4.

The concepts `LexerProductionList` and `GrammarProductionList` correspond to the above mentioned parts of a syntax definition. Every lexer production describes exactly one lexer state. Therefore, all tokens defined by lexer rules in a production are only acceptable for a scanner if it has currently the state defined by the production. The initial state of a scanner is given by the first occurring production in the sequence of lexer productions. The order of lexer rules implies the precedence of defined tokens in context of a lexer production. Furthermore, a token can be of the type $normal$ or $ignore$, which declare whether it can occur as a terminal on the right hand side of syntactical productions. A nonterminal is allowed to occur once on the left hand side of one single production ($definition$) and in any EBNF expression in any production ($reference$).

To emphasise where extension points and variation points may occur, we marked several model elements as `HookType` or `SlotType`. A `HookType` is a container pro-

```
componentmodel SyntaxDefinitionBoxology
fragmentnamespace <http://www.reuseware.org/ syntaxdefinition >

boxtype SyntaxDefinitionBox  hiding  SyntaxDefinition  provides
    hook LexerProductionHook :  lexerproductions
    hook GrammarProductionHook : grammarproductions

boxtype LexerProductionBox hiding  LexerProduction  provides
    hook LexerRuleHook :  lexerrules

boxtype GrammarProductiofragmentnamespacenBox hiding GrammarProduction provides
    hook ExpressionHook : body

boxtype LexerRuleBox hiding LexerRule provides
    hook ExpressionHook :  expression

slottype  GrammarProduction binds TerminalReference by name
slottype  REGExpression binds REGExpression by name
slottype  LexerRule binds LexerRule by name
slottype  Terminal  binds  TerminalReference  by name
slottype  NonTerminal binds NonTerminalSymbol by name
```

Listing 1. Componentmodel specification for syntax definitions

viding a collection that can be extended with additional elements, e.g., a composer may append new lexer productions to the corresponding sequence. Instances of `HookTypes` are always part of a component's implicit interface. In contrast, `SlotTypes` contribute to the explicit interface: Variation points of components have to be declared by its developer.

For parser specifications we propose different slot types. A common scenario is the import of complex lexer rules and regular expressions from a library, e.g. notations for different data types. Variation points for grammar productions, terminals and nonterminals serve to define templates of grammar productions. A reasonable use case for grammar templates are expression languages. Stacking of the language's operators according to their precedence is an acceptable way to avoid ambiguities, at least in the concerned part of the grammar. Since many kinds of expressions basically share a similar syntactic structure, e.g. boolean- and arithmetic expressions in infix notation, this can be factored out as a template. A small example for this is depicted in listing 2. The first part shows an EBNF based production serving as a template for infix binary expressions. It contains a variation point for the operator (*Op*) and the two operands (*L1* and *L2*). Note that the recursion of *L1* will cause instantiated expressions to be represented right associative in syntax trees. In the second part the template is imported twice and instantiated such that a simple boolean expression language is created. The priority levels are connected through binding *L2* and *L1* appropriately. The resulting EBNF productions are shown in the listing's third part.

```
[NonTerminal:L1]::=[NonTerminal:L2][Terminal:Op][NonTerminal:L1] | [NonTerminal:L2]
```

```
import fragment expression as level1
import fragment expression as level2
bind level1 : L1−>Or, Op−>"|", L2−>And
bind level2 : L1−>And, Op−>"&", L2−>Value
```

```
Or ::= And "||" Or | And
And ::= Value "&" And | Value
```

Listing 2. Instanciation of an expression template as boolean expression

In ISC components are fragments, i.e. in our case instances of parts of the component model. The set of valid fragment types can mainly be derived from the set of `Slot-` and `HookTypes` defined in the component model specification (see Listing 1). Additionally, we see syntax definitions as valid fragments. Note that a module may contain one or more fragments.

3.2 Composition Technique and Composition Operators

Composition systems are only as powerful as their composition operators are. Lämmel [15] defines a set of basic operators for grammar adaption, reflecting a grammar developer's potential actions when manually customising an existing grammar through editing. A syntax composition system should not be restrictive in performing these actions when reusing fragments from a module. Since the underlying formalisms of mainstream parser generators constrain valid compositions, the system also has to support the user, either through manually resolving or automatically (at least partially) avoiding conflicts. The first case resembles *whitebox* composition while the latter is about *blackbox* reuse. In the following we briefly discuss the composers realized in our system.

Import allows to directly include fragments from a declared module. In contrast to the example of Listing 2, we use import as an *in-place bind* operator. Thus the operator may occur in place of a variation point replacing itself with the imported fragment.

Adapt provides *whitebox* reuse abstractions and is based on *grammar inheritance* [2,16]. It allows the adaptation of existing language specifications at a very fine level of granularity. Several kinds of merge rules can be applied. *Remove* and *refine* allow to delete or add a certain alternative to an EBNF expression or a regular expression of the base specification. Furthermore, *override* permits replacements of complete expressions. To handle conflicts among regular expressions in lexical rules we provide an additional kind of rule: *Merge before* inserts a lexical rule before a specific lexical rule occurring in the base specification.

```
/**
* This function computes faculty of a number.
* @param number — the value for which to compute faculty
* @return number!
*/
public int fac(int number) // do computation
```

Listing 3. A Java method with doclet comment

A usage of refinements will lead to an enrichment of the languages which are subject of adaption while removing and replacing may cause non-monotonic changes with respect to base languages.

Several use cases for grammar inheritance can be found in connection with the Polyglot [18] framework which provides an extensible compiler frontend for Java.

A formal definition of the operator as we use it here can be found in the Appendix B.

Refactor allows to rename or remove all of a grammar symbol's occurrences in a module. This is essentially useful for avoiding naming conflicts in composed fragments or for removing unwanted parts of a reused language.

Embed is an invasive composer for embedding existing languages from syntax defintion modules into a host language. Although grammar inheritance is a powerful and useful mechanism to allow adaptations of existing language specifications and therefore introduces a notion of modularity into language specifications, it seems not to be adequate in some cases: Users need to have an exact understanding of the context-free language and the specification they want to adapt. However, there are use cases in which a kind of *blackbox* reuse would be more adequate to provide modularity, in particular when a language is composed of embedded sublanguages.

In many cases textual languages and especially programming languages can be regarded as composed languages. For instance, the common object-oriented language Java may be regarded syntactically as a composite of a host language which contains the language's basic concepts (e.g. compilation units, method bodies, statements etc.) and some additional *hooked in* sublanguages like expression languages (e.g. arithmetic or boolean expressions). Furthermore, Java allows a special kind of static annotations in comments (called doclets) whose syntactic structure can also be regarded as a sublanguage. External tools like XDoclet [26] and Javadoc [20] then make use of this additional meta-information for code generation or generation of documentation. Listing 3 shows an example of a simple Java method with an annotated doclet comment which can be processed by Javadoc. The doclet comment language is delimited from the rest of Java through $/**$ and $*/$. Hence, doclets are *embedded* into the Java language. A composition operator which embeds sublanguages into a host language automatically would allow to pull the doclet language specification out of the Java specification into a separate module.

```
**In** $m_{host}$ //A host language module
**In** $m_{embed}$ //The module to be embedded
**In** $name$ //The unique module name
**In** $ttype$ //The preferred transition token type
**In** $guard_{in}, guard_{out}$ //Regular expressions for valid guard literals
**Out** $m_{comp}$ //The composed result
I. Syntactic embedding
    Create the sequence of syntactic productions in $m_{comp}$ such that
    I.1 $m_{comp}$ contains all productions of $m_{embed}$ and $m_{host}$
    I.2 $m_{comp}$ contains a production $name ::= startsymbol(m_{embed})$
        if $ttype = ignore$ or
        otherwise $name ::= name\_in\ startsymbol(m_{embed})\ name\_out$
    I.3 $startsymbol(m_{comp}) = startsymbol(m_{host})$
II. Lexical embedding
    Create the new lexer state automaton in $m_{comp}$ such that
    II.1 $m_{comp}$ contains all lexer states of $m_{host}$ and $m_{embed}$
    II.2 both modules have their own namespace in $m_{comp}$
    II.3 $hom$ maps $m_{host}$ and $m_{embed}$ to $m_{comp}$
    II.4 each state in $hom(m_{embed})$ has a host lexer transition with
         $name = name\_out$, $tokentype = ttype$, $regex = guard_{out}$ and
         $state = startstate(hom(m_{host}))$
    II.5 each state in $hom(m_{host})$ has a lexer transition with $name = name\_in$,
         $tokentype = ttype$, $regex = guard_{in}$ and $state = startstate(hom(m_{embed}))$
```

Listing 4. Description of the embed operator in pseudo code

One may argue that embedding sublanguages into host languages means just copying productions from one grammar into another. In the broadest sense and in theory this is true, however, in practice, additional conditions need to be considered. Parser specifications do not only contain grammar productions but also lexical rules which define terminal symbols as tokens. Copying just one specification into another most likely causes conflicts between lexical rules in the composite. Reconsider the above example. The string `number` frequently occurs inside the doclet comment and in parts of the method declaration. In the doclet's context `number` is regarded as a piece of text while it actually is an identifier in Java.

In Section 2 we discussed *scanner-less parsing* and *context-aware scanning* as prevalent concepts for avoiding lexical conflicts. Realizing these non declarative concepts in a mainstream parser generator require changing its implementation. Scannerless parsing even requires appropriate disambiguation constructs in the specification language [23]. Hence, for our composition system, we decided to build up on *lexer states* as a more declarative but maybe less powerful technology that is common to many mainstream parser generators, e.g., JavaCC and SableCC.

Listing 4 shows an abstract description of the embed operator. *embed* creates a new syntax definition by extending it with the rules and productions from the participating modules in a structure preserving manner. In Listing 4, structure preservation is ensured by the homomorphic mapping hom. The lexer state automata defined by lexer productions of the modules are then connected via *in* and *out* transitions as configured by the user through $guard_{in}$ and $guard_{out}$. Lexer states of different modules are separated by namespaces. Users can also select the appropriate token type for guard tokens. Ignored tokens have the advantage not to influence the context-free grammar's structure such that tools do not have to cope with newly introduced symbols. However, then they are not available for firstset computations in the parser generator which can be problematic in some cases.

An additional formal definition of the *embed* operator for multiple arguments can be found in Appendix C.

3.3 Composition Language

A very important part of composition systems is the language composition programs are written in. To exemplify the usability of ISC for grammars in Listing 2, a very simple composition language supporting the importing of fragments and binding of slots was chosen. However, this language is a general composition language which does not provide any domain specific abstractions (e.g., it does not include the complex composers of the previous section). To define a more specialized and usable composition language, we decided to extend the metamodel of Figure 1 with concepts representing composer calls. Such concepts can be rewritten by Reuseware by traversing the fragment components and executing the composers defined in Section 3.2.

To avoid redundancy with Figure 1, we explain our composition language briefly here:

- *Refactor* operations occur at the syntax definition root.
- *Import* may occur in place of grammar productions, regular expressions, or lexer rules.
- A call to *Adapt* has to be placed in the root of a syntax definition. The merge rules *refine*, *override* and *remove* are combined with grammar productions. *Merge before* additionally occurs in lexer rules.
- *Embed* only occurs in place of grammar productions.

Section 4 will show some concrete example composition programs.

4 Prototype and Applications

In this section, we show an application of our composition system to syntax specifications that can be used with the JavaCC parser generator. While composition takes place on the model level, a concrete syntax mapping closes the gap between the fragment based component model and the concrete specification language jj of JavaCC[3]. In the following, we shortly discuss the mapping between our composition system and JavaCC. Afterwards, we present some examples creating the Java and AspectJ [14] like toy languages μJava and μAspectJ from modules.

[3] See `https://javacc.dev.java.net/doc/javaccgrm.html` to get an overview of the complete jj specification language. In this paper, we consider a subset of jj abstracting from semantic actions and other generator-specific constructs.

```
<DEFAULT>
TOKEN:{
    <opor:" || ">:DEFAULT |
    <opand:"&&">:DEFAULT
}
void Or():{} { And() <opor> Or() | And() }
void And():{} { Value() <opand> And() | Value() }
```

Listing 5. Expressions of Listing 2 in jj syntax

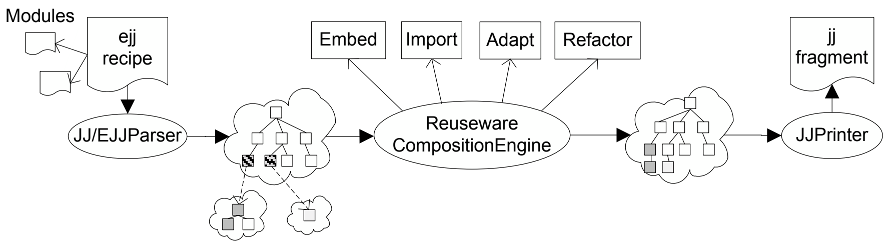


Fig. 2. Composition execution

4.1 Implementation

To support specification languages of mainstream parser generators, parts of their specification languages have to be mapped to corresponding concepts in the fragment component model. For instance, concepts for syntactic productions have to be mapped to the type `GrammarProduction` in the model. To load and store jj fragments, we use *text-to-model* and *model-to-text* transformations. Therefore, a variety of textual concrete syntax mapping approaches exist [9]. In Reuseware, such transformations are realised with the help of the EMFText [7] tool which allows to create mappings to automatically generate parser, printer and a text based editor.

The usage of jj and *model-to-text* transformations is due to JavaCC's support for lexer states and its close similarity to the metamodel from Section 3.1. Listing 5 shows a syntax definition in jj syntax equivalent to the instantiated expression template in Listing 2. It contains a lexical production defining the lexer state $DEFAULT$, which always is the name of the initial state in JavaCC. The production contains two lexical rules for the expression operators, both declaring a reflective transition to $DEFAULT$. Syntactic productions are defined in the remainder of the specification. As an extension to jj, ejj is a textual syntax for composition programs based on the extended metamodel mentioned in Section 3.3. Consider Figure 2 which gives a rough overview on how compositions are executed by our system. It first takes an *ejj* specification as composition program.

```
 1  REUSING SyntaxDefinition IN /doclet.jj  AS doclet;
 2  REUSING GrammarProduction IN /expressiontemplate.ejj AS expression;
 3  {
 4   TOKEN:{
 5     <INT:"int">|
 6     // ...  snip  ...//
 7     <IDENT:(["a"−"z"]|"_")(["a"−"z","0"−"9"]|"_")∗>|
 8     <INTEGER_VALUE:(["1"−"9"](["0"−"9"])∗)|"0">
 9   }
10
11   SKIP:{ <WS:"␣"|"\t"|"\n"|"\r"|"\f"> }
12
13   void CompilationUnit():{}{ ( PackageDeclaration() )? (
             ImportDeclaration() | Comment() )∗ ( ClassDeclaration ()) }
14
15   @ Comment(){ "/∗∗": doclet :"∗/" }
16
17   void PackageDeclaration():{}{ <PACKAGE> QualifiedName() <
             SEMICOLON> }
18
19   // ...  snip  ...//
20
21   void Statement():{}{( ExpressionStatement() | WhileStatement() |
             IfStatement() | ReturnStatement() ) }
22
23   // ...  snip  ...//
24
25   IMPORT expression<expression−>'Expression()'.jj,op−>'Operator1()'.jj,
             term−>'Expression2()'.jj>
26
27   void Term():{}{ Atom() | <LBRACK> Expression() <RBRACK> }
28
29   // ...  snip  ...//
30
31   void Operator1():{}{ <LT> | <GT> | <EQOP> }
32
33   // ...  snip  ...//
34  }
```

Listing 6. Embedding doclets

```
 1  {
 2   SKIP:{ < default_doclet :"/∗∗">: doclet_default  }
 3
 4   TOKEN:{
 5     <INT:"int">|
 6     // ...  snip  ...//
 7   }
 8
 9   SKIP:{ <WS:"␣"|"\t"|"\n"|"\r"|"\f"> }
10
11   < doclet_default >
12   SKIP:{ < doclet_default_host :"∗/">: DEFAULT }
13
14   < doclet_default >
15   TOKEN:{
16     <PARAM:"@"((["a"−"z","A"−"Z"]|"_")(["a"−"z","A"−"Z","0"−"9"]|
               "_")∗)>|
17     <TEXT:(~["@","∗"])+>
18   }
19
20   < doclet_default >
21   SKIP:{ <WSD:"∗"> }
22
23   void CompilationUnit():{}{ ( PackageDeclaration() )? (
             ImportDeclaration() |Comment() )∗ ( ClassDeclaration ()) }
24
25   void Comment():{}{ doclet() }
26
27   // ...  snip  ...//
28
29   void Expression():{}{ Expression2() ( Operator1() Expression() )? }
30
31   // ...  snip  ...//
32
33   void doclet():{}{ DocletStatement() }
34
35   // ...  snip  ...//
36  }
```

Listing 7. Composition result

The specification contains references to *jj* conformant modules as subjects to be composed. The *ejj* parser creates model instances for *ejj* programs and each referenced module to make them available to the Reuseware composition engine. The composition engine is aware of all defined composition operators and interprets the composition programs by traversing the model and executing the composers. After composition, the resulting model is forwarded to a printer and stored as textual component reusable as a module itself.

4.2 Constructing μJava

In the following two subsections we show how a doclet language could be embedded into a subset of the Java programming language (called μJava) and how the composed language may be extended to obtain an AspectJ like aspect language (called μAspectJ) containing an embedded pointcut language.

Listing 6 shows an excerpt from our μJava definition in ejj syntax. The first two lines declare references to the doclet definition (namespace doclet) and to a generic expression production (namespace expression). Lines 4-11 define the host language's lexical productions. Lines 13-34 contain syntactic productions in jj syntax and composer calls. A call to *embed* can be found in line 15. It is declared by the @ keyword. There are several things to be observed: First, the doclet module is treated as *blackbox*, since it is just referred by its name. Second, doclets will be surrounded by $/**$ and $*/$ in the composed language. And finally, the doclet definition can be referenced by the nonterminal $Comment()$ as in line 13. A call to *import* can be found in line 25. It includes a grammar production from the *expression* module and fills the variation points *expression*,*op* and *term* with appropriate nonterminals.

Listing 7 shows the resulting specification produced by the Reuseware composition engine. Lexer states have been included and manipulated by *embed* with ignore tokens. Lines 2-9 belong to the host language's default state (implicitly named $DEFAULT$ in JavaCC) while the doclet language's default state (named $< doclet\_default >$) is declared in lines 11-21. Line 2 and line 12 define transitions between μJava and doclet language and have been introduced by *embed* for the arguments $/**$ and $*/$. A grammar production for $Comment()$ can be found in line 25 while the doclet production sequence actually begins in line 33. The result corresponding to the mentioned *import* call can be found in line 29.

4.3 Extending μJava to μAspectJ

In this section we use our system to extend the resulting specification from last section to obtain μAspectJ – a small aspect oriented language with AspectJ like syntax. Deriving a complete compiler for AspectJ is out of the scope of this paper since this work has already been done by the *abc* project [4] which uses Polyglot [18] for extending Java. Moreover, in [5] a scannerless solution based on SDF2/SGLR is presented.

Here we use *adapt* to slightly modify μJava. In combination we use *embed* to embed a small pointcut language into the specification. The actual composition program is contained in Listing 8. The EXTEND keyword declares a specification to reuse and extend a base specification and belongs to the *adapt* composer call. In this context, lines 7-12 introduce new aspect related keywords to the basic μJava specification in front of the identifier (IDENT) definition since this definition would subsume them normally. The aspect syntax is declared by a newly introduced production for the nonterminal $AspectDeclaration()$ in line 18. To make aspects available in compilation units the original $CompilationUnit()$ production is replaced to allow aspect declarations as alternative to class declarations.

Pointcuts are embedded in lines 23-24. In comparison to the embedding of doclets no single literals but regular expressions are used to declare *in* and *out* transitions to the pointcut lexer state automaton. That leads to a more convenient notation for transitions between host language and pointcut language, since $< in >$ and $< out >$ can be used instead of $pointcut$ and ;. Furthermore, pointcuts are connected to the original μJava type declarations. This is done by replacing the declared variation point $TypeSlot$ with $Type()$.

In contrast to μJava we will not discuss the resulting specification here. Instead we glance at a concrete instance of the composed aspect language shown in Listing 9. It contains an aspect with several pointcuts, a standard Java method, a doclet comment and an advice. Recall that the pointcut definitions in lines 5,11,13 and the doclet comment in lines 7-10 belong to embedded languages. In our case, the main difference between doclet comments and pointcuts is that the former was embedded independently, hence introduces no references to the host language. In contrast, pointcuts are connected to the host language via a reference to $Type()$ replacing $TypeSlot$.

```
REUSING SyntaxDefinition IN /java—comments.jj AS java;
REUSING ReuseSyntaxDefinition IN /pointcuts.ejj  AS pointcuts ;

EXTEND java{

  TOKEN : {
    MERGE BEFORE IDENT <ASPECT:"aspect">|
    MERGE BEFORE IDENT <BEFORE_:"before">|
    MERGE BEFORE IDENT <AFTER:"after">|
    MERGE BEFORE IDENT <AROUND:"arround">|
    MERGE BEFORE IDENT <DP:":">|
    MERGE BEFORE IDENT <RET:"returning">
  }

  OVERRIDE void CompilationUnit():{}{
   ( PackageDeclaration () )? ( ImportDeclaration () |Comment() )∗ (
            ClassDeclaration () | AspectDeclaration () )    }

  void AspectDeclaration ():{}{
   <ASPECT> <IDENT> ( ExtendList() )? AspectBody() }

    // ...  snip  ...//

  @ PointcutDeclaration (){
    " pointcut " | "<in>" : pointcuts <TypeSlot—>'Type()'.jj> :  ";" | "
             <out>" }
}
```

Listing 8. Extending μJava

```
1   package tests ;
2
3   aspect TestAspectUnit{
4
5       pointcut classes () : within(<out>babylon. util . SharedList<in>) ||
                within(<out>babylon. util .LinkedMap<in>);
6
7       /∗∗
8        ∗ Pointcut for selecting all methods returning integer numbers.
9        ∗ <out> and <in> declare lexerstate  transitions .
10       ∗/
11      pointcut methods(): execution(<out>int<in> ∗(..) );
12
13      pointcut pointsOfInterest (): methods() && classes();
14
15      public int fac(int number){
16          int result = 1;
17          while(number > 0){
18              result = result ∗ number;
19             number = number — 1;
20          }
21          return result ;
22      }
23
24      after () returning(int result ): pointsOfInterest () {
25          System.out. println ("Faculty␣of␣" + result + "␣is␣" + fac(
                    result ));
26      }
27  }
```

Listing 9. An aspect

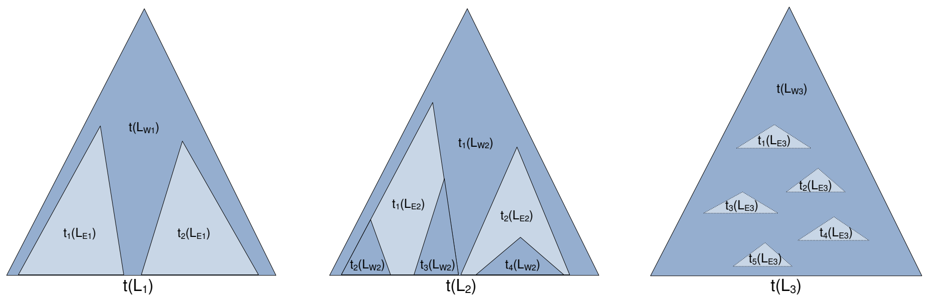
Naturally, if a type occurs in an actual pointcut definition (e.g. lines 5 and 11) it is necessary to switch to the host language's lexer state automaton and back, since terminals required by $Type()$ are not contained in the pointcut language but in the host language. This could have been avoided with some redundancy by manually duplicating the token definitions for the terminals demanded by $Type()$ in the pointcut lexer state automaton. Clearly, such an action may introduce unwanted conflicts among token definitions only solvable by an extra lexer state. Hence, in such cases it could be better to use *import* instead of *embed*.

The next section discusses the usability and possible improvements of the composition system.

5 Usability and Future Work

The examples of the last section showed that the syntax composition system provides a sufficient set of composition operators to extend the Java syntax appropriately. Altogether, our approach combines invasive software composition and parser specifications in a very beneficial way for mainstream parser generators. The *adapt* composer allows to extend and adjust grammars in a flexible and nearly unrestricted way. On the one hand, this flexibility is especially important if language extensions make more complex adjustments in a grammar necessary, such as factoring out a common prefix to avoid overlapping firstsets which is a likely transformation to make grammars manageable for LL(k) parser generators. On the other hand, in combination with interfaces given by *slots* and *hooks, adapt* is essential for adding grammar connectors as a kind of glue between language modules.

In contrast, *embed* is much more specific. It provides black-box reuse abstractions by generating special glue which connects lexer state DFAs of the modules and the host language and thus, avoids introducing new conflicts between token definitions. However, there are some obvious limitations of the *embed* operator. Its need for special



(a) sublanguage which does not refer to the host language

(b) little cohesion between sublanguage and host language

(c) strong cohesion between sublanguage and host language

Fig. 3. Syntax trees of composed languages

guard tokens declaring transitions between lexer state automata forces to enrich the language with perhaps unwanted keywords. In general, this is not the case for scanner-less parsing and context-aware scanning. These techniques make use of the parser's state information. Lexer states can only simulate context, i.e., the guards mark beginning and ending of the embedded language's input. The input tokens then are created according to the context of the starting nonterminal of the embedded language. However, lexer state based context simulation is sufficient in several use cases, especially if the cohesion between host language and embedded language is low. Figure 5 shows three hypothetical syntax trees $t(L_i)$ representing different levels of syntactic cohesion between a host language L_{Wi} and an embedded sublanguage L_{Ei}. Note that L_i is the language composed from L_{Wi} and L_{Ei}. In the first example (Figure 3(a)) subtrees $t_i(L_{E1})$ contain nodes which belong to constructs of L_{E1} only, as it would be the case with the doclet language. For L_1 like cases, lexer state based *embedding* can be applied with similar results to the context-aware technologies. In contrast, Figure 3(b) and Figure 3(c) contain examples for syntax trees of embedded languages which refer to the host language and may contain subtrees whose root represents constructs of the host language. The pointcut language from Section 4.3 could be L_2 since it actually is a relatively complex language providing own leaf nodes in syntax trees. The problem here is the needed context simulation forcing users to recurringly declare transitions from L_{E2} to L_{W2} and back. It depends on the language designer to make this transparent in the created language. In case of L_3, the benefit gained from embedding L_{E3} from a module into L_{W3} is even more questionable, since trees $t_i(L_{E3})$ are rather flat. Additionally, every path in a $t_i(L_{E3})$ leads to a subtree which belongs to L_{W3} at some point. This is due to the fact that L_{E3} does not provide any leaf nodes, hence does provide no terminals in its (partial) grammar.

There are several starting points for future work and improvements of the presented composition system:

– A more intelligent implementation of *embed* could analyse the grammar to derive the guards automatically if possible.

– The component model is relatively small and lacks a support of semantic actions, which usually consist of platform dependent program fragments. To integrate them into the generic component model, an appropriate representation has to be found.
– To support multiple mainstream parser generators, a more advanced composition system should provide means for extending a platform independent core by adding generator specific plugins, languages for semantic actions and composition operators. To some degree, such a system may also allow to compose specifications written in different parser specification languages.

6 Conclusion

In this paper we presented a generic composition system for syntax specifications of mainstream parser generators. The system supports a set of composition operators that are tailored to the features of these parser generators. It helps to resolve conflicts in grammars that usually would have been resolved through directly editing the specifications to be composed. Naming conflicts can be avoided by using the *refactor* operator. *Import* easily allows to instantiate and reuse single constructs and templates from a repository. By using the *adapt* composer, syntax definitions can be adapted and extended at a very fine level of granularity. This especially helps when legacy systems are about to be developed further and allows to create extensions as modules. *Embed* introduces abilities to embed languages from modules and avoids to create lexical conflicts. It generates lexer states to separate the token sets of the participating languages. However, it has some restrictions because lexer states can only simulate context.

As a proof of concept we implemented the composition system on the basis of the Reuseware framework. The resulting system includes the generic component model, a generic composition language extending the component model and implementations of the discussed composition operators. To apply the composition system to a mainstream parser generator, we chose the JavaCC specification language. A corresponding subset was mapped to our component model. Composition programs can be written in an extended syntax based on the selected subset.

In the future we aim to apply our composition system to more mainstream parser generators. We also want to extend the component model to support more constructs like semantic actions.

References

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullmann. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1985.
2. Mehmet Aksit, René Mostert, and Boudewijn Haverkort. Compiler generation based on grammar inheritance. Technical report, University of Twente, July 1990.
3. Uwe Aßmann. *Invasive Software Composition*. Springer, 1 edition, 2003.
4. Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: an extensible AspectJ compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 87–98, New York, NY, USA, 2005. ACM.

5. Martin Bravenboer, Éric Tanter, and Eelco Visser. Declarative, formal, and extensible syntax definition for aspectj. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 209–228, New York, NY, USA, 2006. ACM.
6. EMF. Eclipse modeling framework project. http://www.eclipse.org/modeling/emf/.
7. EMFText. http://www.emftext.org.
8. Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. *SIGPLAN Not.*, 39(1):111–122, 2004.
9. Thomas Goldschmidt, Steffen Becker, and Axel Uhl. Classification of Concrete Textual Syntax Mapping Approaches. In *Proc. of ECMDA-FA*, volume 5095 of *LNCS*. Springer, 2008.
10. Robert Grimm. Better extensibility through modular syntax. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 38–51, New York, NY, USA, 2006. ACM.
11. Jakob Henriksson. *A Lightweight Framework for Universal Fragment Composition*. PhD thesis, Technischen Universität Dresden, 2008.
12. Jakob Henriksson, Florian Heidenreich, Jendrik Johannes, Steffen Zschaler, and Uwe Aßmann. Extending grammars and metamodels for reuse: the Reuseware approach. *IET Software*, 2(3):165–184, 2008.
13. JavaCC. https://javacc.dev.java.net/doc/docindex.html.
14. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.
15. Ralf Lämmel. Grammar adaptation. In *FME '01: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, pages 550–570, London, UK, 2001. Springer-Verlag.
16. M. Mernik, M. Lenič, Enis Avdičaušević, and Viljem Žumer. Multiple attribute grammar inheritance. *Second Workshop on Attribute Grammars and their Applications*, March 1999.
17. Marjan Mernik, Viljem Žumer, Mitja Lenič, and Enis Avdičaušević. Implementation of multiple attribute grammar inheritance in the tool LISA. *SIGPLAN Not.*, 34(6):68–75, 1999.
18. Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. *Proceedings of the 12th International Conference on Compiler Construction*, LNCS 2622:138–152, April 2003.
19. Terence Parr and Russell Quong. Antlr: A predicated-LL(k) parser generator. *Software – Practice and Experience*, 25(7):789–810, July 1995.
20. Mark Pollack. Code generation using Javadoc. *JavaWorld.com*, August 2002.
21. PPG. A parser generator for extensible grammars. http://www.cs.cornell.edu/Projects/polyglot/ppg.html.
22. SableCC. http://sablecc.org/.
23. J. Scheerder, J. J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized lr parsers. In *Compiler Construction (CCŠ02)*, pages 143–158. Springer-Verlag, 2002.
24. Masaru Tomita. *Generalized L.R. Parsing*. Kluwer Academic Publishers, Norwell,USA, 1991.
25. Eelco Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997.
26. Craig Walls and Norman Richards. *XDoclet in Action*. Manning Publications, 2003.
27. Eric R. Van Wyk and August C. Schwerdfeger. Context-aware scanning for parsing extensible languages. *Sixth International Conference on Generative Programming and Component Engineering (GPCE'07)*, October 2007.

A Foundations

In this section we will define syntax definitions on the basis of context-free grammars and deterministic finite automata (DFA). The definition reflects the needs of parser generators for such specifications and considers the concept of lexer states for lexical disambiguation. It also uses the concepts of regular expressions in context-free grammars.

Definition 1 (Context-free Grammar (CFG)). *A context free grammar is a 4-tupel $G = (N, T, P, S)$ with N a set of nonterminals, T a set of terminals with $T \cap N = \emptyset$, $P : N \times Regexp_{T \cup N}$ the production relation and $S \subseteq N$ the set of start symbols.*

Definition 2 (Lexer Grammar (LG)). *A lexer grammar is a CFG with $P : N \times Regexp_T$ and S=P. P is ordered and elements of P are called token definitions.*

Definition 3 (Deterministic Finite Automaton (DFA)). *A deterministic finite automaton $A = (Q, \delta, q_0, Q_F, E)$ is a 5-tupel with Q a set of states, $\delta : Q \times E \to Q$ the transition function, $q_0 \in Q$ the initial state, $Q_F \subseteq Q$ the set of accepting states and the set E of edge names.*

Definition 4 (Syntax Definition with Lexer States). *A syntax definition $D = (G_S, G_L, I, A, Q_L)$ consists of a context-free grammar G_S, a lexer grammar G_L with $T_S = N_L - I$, $I \subseteq N_L$ a set of ignored tokens, a lexer state DFA A with $Q_F = Q$ and $E = N_L$ and Q_L a set of lexer productions with $Q_L : Q \to \mathcal{P}(N_L)$.*

The set I marks tokens that are not to be used in the CFG and thus are igored by the parser. In the remainder of this paper, we distinguish different types of tokens by annotating them with $normal$ or $ignore$. Lexer productions are used to assign token definitions with states in the lexer state DFA which is used by the scanner component at runtime to determine the currently valid set of tokens.

To stay close to the component model in Section 3.1 and the actual prototype implementation discussed in Section 4, we use a similar notation for syntax definitions which can be mapped to Definition 4.

Definition 5 (Notation for Syntax Definitions). *A syntax definition $D := (g_L : G_L, g_S : G_S)$ consists of a context-free grammar g_S of type G_S and a sequence of lexer productions g_L of type G_L with $G_L := [q_1 : Q_L, \dots, q_i : Q_L, \dots, q_n : Q_L]$ and $Q_L := (g_R : G_R, s_A : Q)$. g_R denotes a sequence of token definitions with $G_R := [r_1 : R, \dots, r_j : R, \dots, r_m : R]$ and*

$$R := (terminal : T \to regex : Regexp_T, ttype : I, s_F : Q).$$

$terminal$ is a terminal of type T and $ttype$ denotes the token type, s_F is a lexer state of type Q and $regex$ a regular expression. A grammar of type $G_S := [p_1 : P, \dots, p_i : P, \dots, p_n : P]$ is defined by a sequence of syntactical productions of the form

$$P := (nonterminal : N \to regex : Regexp_{N \cup T}).$$

The following example defines the concrete syntax of a simple language for arithmetic expressions like $3 - 4 \div 5$.

Example 1 The syntax definition D_{Exp} describes the concrete syntax of simple arrithmetical expression allowing the operators $-$ and $\div$.

$$
\begin{aligned}
D_{Exp} =&\ (G_{L_{Exp}}, G_{S_{Exp}}) \\
G_{L_{Exp}} =&\ [(G_{R_{Exp}}, default)] \\
G_{R_{Exp}} =&\ [(minus \to {''}{-}{''}, normal, default), \\
&\ (div \to {''}{\div}{''}, normal, default), \\
&\ (num \to {''}[0-9]+{''}, normal, default) \\
&\ (wse \to {''}\backslash n|\backslash r|\ {''}, ignore, default)] \\
G_{S_{Exp}} =&\ [(E \to T(-T)*), \\
&\ (T \to F(\div F)), \\
&\ (F \to num)]
\end{aligned}
$$

E represents the start symbol of the context-free grammar since its production $E \to T(-T)*$ occurs first in the sequence of syntactical productions. $\diamondsuit$

To define the composition operators we will use some elementary notations and operations. Syntax definitions consist of sequences of grammar and lexer productions and sequences of token definitions. Therefore, operations based on lists can be helpful to manipulate the whole data structure. Thus in the following our notation for lists and some simple operations on lists are defined.

Lists can be declared by the keyword $List$ and $List\ E$ declares a list with entries of type E. For example $List\ P$ declares a list of syntactical productions which may also be abbreviated with G_S. An entry type E could again be a list, so that $List\ G_S$ declares a list of lists with entries of type P, e.g., a list of context-free grammars. We use three different representations for lists which later get used in the operator definition.

- $\emptyset$ - denotes an empty list
- $[e_1|e_2, \ldots]$ - denotes a list of arbitrary length with first element e_1 and rest $e_2, \ldots$
- $[e_i, \ldots, e_j]$ - denotes a list with elements from e_i to e_j of length $l = 1 + (j - i)$ and $i, j \in \mathbb{N} \wedge i < j$

Furthermore, we use $\_ \oplus \_$ to concatenate lists and $\_ \ominus \_$ to filter lists [4].

To enable a usage of $\_ \oplus \_$ with regular expressions, these are treated as lists of regular expressions. A list of partial expressions is equivalent to their unification, e.g., a regular expression $a|bc|(e)*$ is equivalent to $[a, bc, (e)*]$.

Beyond manipulation operators, some constructors are needed. For every possible type $\mathcal{T}$, which may be used in a syntax definition, we introduce a standard constructor $\mathcal{T}(arg_1, \ldots, arg_n)$ to construct an element of type $\mathcal{T}$.

The next sections introduce the adapt composition operator for extending syntax definitions and embed composition operator for embedding context-free languages specified by syntax definitions.

[4] It takes two lists l_1 and l_2 as arguments and removes all elements from l_1 which also occur in l_2.

B Adapt Composer Definition

The *adapt* composer realises a grammar inheritance relation between different syntax definitions. The basic idea is to reuse productions and token definitions from an existing base grammar and to a adapt it to a new usage context by *replacing*, *refining*, *extending* or *removing* parts of the grammar.

Grammar inheritance is often used by systems providing concepts for modular grammars, e.g., the parser generator PPG [21] or LISA [17], an attribute grammar based compiler framework.

B.1 Merge Relations

The adapt composer maps a base definition $D_{\mathcal{B}} = (G_{S_{\mathcal{B}}}, G_{L_{\mathcal{B}}})$ and an extending definition $D_{\mathcal{E}} = (G_{S_{\mathcal{E}}}, G_{L_{\mathcal{E}}})$ to a new syntax definition $D_{\mathcal{M}} = (G_{S_{\mathcal{M}}}, G_{L_{\mathcal{M}}})$, with:

$$G_{S_{\mathcal{M}}} = merge(G_{S_{\mathcal{B}}}, G_{S_{\mathcal{E}}}, SMR_{\mathcal{B} \triangleleft \mathcal{E}})$$

$$G_{L_{\mathcal{M}}} = merge(G_{L_{\mathcal{B}}}, G_{L_{\mathcal{E}}}, LMR_{\mathcal{B} \triangleleft \mathcal{E}})$$

Users may use four different basic merge operations to specify how and what parts of the base definition should be adapted. Those merge operations can be applied to the context-free grammar and token definitions by providing a *syntax merge relation (SMR)* and a *lexical merge relation (LMR)*.

$refine$ Allows to append new alternatives to regular expressions in grammar productions or token definitions.
$override$ Allows to replace regular expressions grammar productions or token definitions.
$remove$ Allows to remove alternatives in regular expressions.
add Allows to add new grammar productions or token definitions.

In the following we define the merge function for the context-free grammar part of syntax definitions. Afterwards, we define the merge function for token definitions and lexer productions.

B.2 Adapting Grammar Productions

Syntax merge relations are defined as follows.

$$SMR \,:\, P_{\mathcal{E}} \times P_{\mathcal{B}} \times \{refine, override, remove\}$$
$$SMR \,:\, P_{\mathcal{E}} \times \{add\}$$

A SMR has to be specified by the user and is required to be left-unique such that extending grammar production can be mapped to multiple merge operations. In the following the actual $merge$ function is specified. $merge$ traverses the list of base grammar productions and creates the merged production list. It delegates merge execution for *refine*, *override* and *remove* to the $manipulate$ function which traverses the extending grammar's productions and merges them with the current base production as specified in the SMR. Finally, all extension grammar productions annotated with *add* are appended to the new production list.

$$merge : G_S \times G_S \times SMR \to G_S$$

$$merge(g_S, \emptyset, smr) = g_S$$

$$merge(\emptyset, g_S, smr) = g'_S \text{ with } g'_S \subseteq g_S \wedge \forall p_k \in g'_S : (p_k, add) \in smr$$

$$merge([p_1|p_2, \ldots], g_S, smr) =$$

$$\begin{cases} [p_1] \oplus merge([p_2, \ldots], g_S, smr) \\ \qquad \text{if } \forall p \in g_S : (p, p_1, refine) \notin smr \wedge \\ \qquad\qquad (p, p_1, remove) \notin smr \wedge \\ \qquad\qquad (p, p_1, override) \notin smr \\ manipulate(p_1, g_S, smr) \oplus merge([p_2, \ldots], g_S, smr) \\ \qquad \text{otherwise} \end{cases}$$

$$manipulate : P \times G_S \times SMR \to P$$

$$manipulate(p, \emptyset, smr) = p$$

$$manipulate(p, [p_1|p_2, \ldots], smr) =$$

$$\begin{cases} manipulate((p.nonterminal \to p_1.bnfex), [p_2|\ldots], smr) \\ \qquad \text{if } (p_1, p, override) \in smr \\ manipulate((p.nonterminal \to p.bnfex \oplus p_1.bnfex), [p_2|\ldots], smr) \\ \qquad \text{if } (p_1, p, refine) \in smr \\ manipulate((p.nonterminal \to p.bnfex \ominus p_1.bnfex), [p_2|\ldots], smr) \\ \qquad \text{if } (p_1, p, remove) \in smr \\ manipulate(p, [p_2|\ldots], smr) \\ \qquad \text{otherwise} \end{cases}$$

The following example shows the merge of two grammars.

Example 2 Let G_b be the base grammar and G_e an extending grammar:

$$G_b = [p_1, p_2] \text{ with } \begin{array}{l} p_1 = A \to aB \\ p_2 = B \to b \end{array} \quad G_e = [p_3, p_4, p_5] \text{ with } \begin{array}{l} p_3 = A \to aA \\ p_4 = B \to bC|bB \\ p_5 = C \to c|cC \end{array}$$

Obviously, $L(G_b) = \{ab\}$. Now we want to extend G_b with G_e to obtain G_r, such that $L(G_r) = \{a^n b^m c^o | (n, m, o \in \mathbb{N}) \wedge (n, m, o > 1)\}$.
With $SMR_r = \{ (p_5, add), (p_1, p_3, refine), (p2, p4, override)\}$ G_r results as follows:

$$G_r = merge(G_b, G_e, SMR_r) = [A \to aB|aA, \\ B \to bC|bB, \\ C \to c|cC] \qquad \Diamond$$

B.3 Adapting Token Definitions

Merging lexer grammars is very similar to the merge of context-free grammars. However, we have to consider the order of token definitions and their contexts w.r.t. different lexer states. The order of token definitions is important, because scanners generated by mainstream parser generator use this information for disambiguation of overlapping token definitions, e.g., overlapping keywords and identifiers.

Therefore, we introduce two additional kinds of merge operations.

$combine$ Triggers a merge of lexer productions (lexer states).
$before$ Allows to insert new token definitions at a specific position in the base definition.

Similar to SMR, a lexer merge relation LMR maps token definitions R and lexer productions Q_L to merge operations.

$$LMR\ :\ Q_{L\mathcal{E}} \times Q_{L\mathcal{B}} \times \{combine\}$$
$$LMR\ :\ Q_{L\mathcal{E}} \times \{add\}$$

LMR for token definitions:

$$LMR\ :\ R_{\mathcal{E}} \times R_{\mathcal{B}} \times \{refine, override, remove, before\}$$
$$LMR\ :\ R_{\mathcal{E}} \times \{add\}$$

LMR is left-unique.

In the following we define $merge$ for lexer productions. $merge$ traverses the list of base lexer productions and merges the token definitions of those productions which are annotated with $combine$ by delegating to $merge_{G_R}$. Extension productions annotated with add are appended to the new lexer production list, i.e., the corresponding lexer states are added to the lexer state DFA.

$$merge\ :\ G_L \times G_L \times LMR \rightarrow G_L$$

$$merge(g_L, \emptyset, lmr) = g_L$$

$$merge(\emptyset, g_L, lmr) = g'_L \text{ with } g'_L \subseteq g_L \wedge \forall q_k \in g'_L : (q_k, add) \in lmr$$

$$merge([q_1|q_2,\ldots], g_L, lmr) =$$

$$\begin{cases} Q_L(merge_{G_R}(q_1.g_R, q.g_R, lmr), q_1.s_A \oplus merge([q_2,\ldots], g_L, lmr) \\ \qquad \text{if } q \in g_L \wedge (q, q_1, combine) \in lmr \\ [q_1] \oplus merge([q_2,\ldots], g_L, lmr) \\ \qquad \text{otherwise} \end{cases}$$

In the following we define the $merge_{G_R}$ function. The actual execution of merge operations is again delegated to the $manipulate$ function.

$$merge_{G_R}\ :\ G_R \times G_R \times LMR \rightarrow G_R$$

$$merge_{G_R}(g_R, \emptyset, lmr) = g_R$$

$$merge_{G_R}(\emptyset, g_R, lmr) = g'_R \text{ with } g'_R \subseteq g_R \wedge \forall r_k \in g'_L : (r_k, add) \in lmr$$

$$merge_{G_R}([r_1|r_2,\ldots], g_R, lmr) =$$

$$\begin{cases} [r_1] \oplus merge_{G_R}([r_2,\ldots], g_R, lmr) \\ \qquad \text{if } \forall r \in g_R : (r, r_1, refine) \notin lmr \wedge \\ \qquad\quad (r, r_1, remove) \notin lmr \wedge \\ \qquad\quad (r, r_1, before) \notin lmr \wedge \\ \qquad\quad (r, r_1, override) \notin lmr \\ manipulate([r_1], g_R, lmr) \oplus merge_{G_R}([r_2,\ldots], g_R, lmr) \\ \qquad \text{otherwise} \end{cases}$$

$$manipulate : G_R \times G_R \times LMR \to G_R$$

$$manipulate(g_R, \emptyset, lmr) = g_R$$

$$manipulate([r_1,\ldots,r_i], [r_{i+1}|r_{i+2},\ldots], lmr) =$$

$$\begin{cases} manipulate(\\ \qquad [r_1,\ldots,(r_i.terminal \to r_{i+1}.regex, r_{i+1}.ttype, r_{i+1}.s_F)], \\ \qquad [r_{i+2}|\ldots], \\ \qquad lmr) \\ \quad \text{if } (r_{i+1}, r_i, override) \in lmr \\ manipulate(\\ \qquad [r_1,\ldots,(r_i.terminal \to r_i.regex \oplus r_{i+1}.regex, r_i.ttype, r_i.s_F)], \\ \qquad [r_{i+2}|\ldots], \\ \qquad lmr) \\ \quad \text{if } (r_{i+1}, r_i, refine) \in lmr \\ manipulate(\\ \qquad [r_1,\ldots,(r_i.terminal \to r_i.regex \ominus r_{i+1}.regex, r_i.ttype, r_i.s_F)], \\ \qquad [r_{i+2}|\ldots], \\ \qquad lmr) \\ \quad \text{if } (r_{i+1}, r_i, remove) \in lmr \\ manipulate(\\ \qquad [r_1,\ldots,r_{i+1},r_i], \\ \qquad [r_{i+2}|\ldots], \\ \qquad lmr) \\ \quad \text{if } (r_{i+1}, r_i, before) \in lmr \\ manipulate([r_1,\ldots,r_i], [r_{i+2}|\ldots], lmr) \\ \quad \text{otherwise} \end{cases}$$

C Embed Composer Definition

We begin our discussion with an example of the steps needed to embed a sublanguage into a host language. The concept of lexer states will be used to separate the sets of tokens of the participating languages.

Example 3 In this example we embed a slightly modified D_{Exp} (see Example 1) into a rudimentary programming language. For this, we replace $(F \rightarrow num)$ with $(F \rightarrow num|Other)$. The host language D_{Host} defines a program to consist of a sequence of methods which themselves contain a block as a sequence of variable declarations and assignments.

$$\begin{aligned}
D_{Host} = {} & ([(G_{R_{Host}}, default)], G_{S_{Host}}) \\
G_{R_{Host}} = {} & [(begin \rightarrow {}''begin'', normal, default), \\
& (end \rightarrow {}''end'', normal, default), \\
& (var \rightarrow {}''var'', normal, default), \\
& (ident \rightarrow {}''[0-9a-zA-Z]+'', normal, default) \\
& (wsp \rightarrow {}''\backslash n|\backslash r|\ '', ignore, default)] \\
G_{S_{Host}} = {} & [(Program \rightarrow Method+), \\
& (Method \rightarrow ident\ begin\ Block\ end), \\
& (Block \rightarrow Declaration|Assignment), \\
& (Declaration \rightarrow var\ ident\ ), \\
& (Assignment \rightarrow var\ Expression), \\
& (Other \rightarrow ident)]
\end{aligned}$$

$Expression$ is a placeholder to later refer to the embedded expression language. On the other hand, $(Other \rightarrow ident)$ will provide a connection from D'_{Exp} to D_{Host}. Hence, the composition interface between host language and sublanguage consists of Other and Expression[5]. For an embedding of the grammar it seems to be enough to append all productions in $G_{S_{Exp}}$ to $G_{S_{Host}}$ and to glue them together by introducing an additional production $(Expression \rightarrow E)$. Hence the resulting production sequence may be constructed as follows:

[5] In ISC, *Other* and *Expression* can be modelled as slots which can be bound with glue nonterminals of a grammar connector. Grammar connectors can be added by using the adapt composer.

$$
\begin{aligned}
G_{S_{Res}} &= G_{S_{Host}} \oplus [(Expression \rightarrow E)] \oplus G'_{S_{Exp}} \\
&= [(Program \rightarrow Method{+}), \\
&\quad (Method \rightarrow ident\ begin\ Block\ end), \\
&\quad (Block \rightarrow Declaration | Assignment), \\
&\quad (Declaration \rightarrow var\ ident\ ), \\
&\quad (Assignment \rightarrow var\ Expression), \\
&\quad (Other \rightarrow ident), \\
&\quad (Expression \rightarrow E), \\
&\quad (E \rightarrow T(-T)*), \\
&\quad (T \rightarrow F(\div F)), \\
&\quad (F \rightarrow num | Other)]
\end{aligned}
$$

Embedding lexer productions is slightly more complex. To avoid conflicts between the participating token sets it is necessary to structurally preserve lexer automata defined in $G'_{L_{Exp}}$ and $G_{L_{Host}}$. Both definitions declare a state $default$ which should be mapped to different states in the resulting specification. Since the resulting DFAs are disjoint, we need to introduce special edges in lexer productions by adding new token definitions with highest priority. These rules then refer to the former $default$ states. In the following we construct $G_{L_{Res}}$ by combining $G'{}_{R_{Host}}$ and $G''{}_{R_{Exp}}$, by retaining the state $default$ in D_{Host} and renaming $default$ in D_{Exp} to $expression$ and by adding transitions in and out between both states.

$$
\begin{aligned}
G_{L_{Res}} &= [(G'{}_{R_{Host}}, default), (G''{}_{R_{Exp}}, expression)] \\
G'{}_{R_{Host}} &= [(in \rightarrow {''}\%{''}, ignore, expression)] \oplus G_{R_{Host}} \\
G''{}_{R_{Exp}} &= [(out \rightarrow {''}\%{''}, ignore, default), \\
&\quad (minus \rightarrow {''}-{''}, normal, expression), \\
&\quad (div \rightarrow {''}\div{''}, normal, expression), \\
&\quad (num \rightarrow {''}[0-9]+{''}, normal, expression), \\
&\quad (wse \rightarrow {''}\backslash n | \backslash r |\ {''}, ignore, expression)]
\end{aligned}
$$

Transitions between *expression* and *default* are now triggered by occurrences of $\%$ in a concrete program. $\Diamond$

embed realises the steps discussed in the example above for one or more language modules and lexer state DFAs.

Beside a host definition given by the first argument and a list of syntax definitions to be embedded, *embed* takes different types of mappings as arguments:

$GNM\ :\ G_S \rightarrow N$ A name mapping associates a fresh nonterminal to every grammar module. In the resulting specification, they may be referred by the associated nonterminal. For instance, in example 3 $Expression$ refers to the former start symbol E in $G'_{L_{Exp}}$.

$GRM : G_L \rightarrow Regexp_T$ This mapping associates a regular expression to the lexer production sequences involved. In this way value ranges for the transitions between token sets are defined. In the example this would be the case for $\%$.

The operator *embed* consists of a helper function $embed_L$ embedding lexer productions and a helper function $embed_S$ embedding grammar productions:

$$embed((g_{L_W}, g_{S_W}), [(g_{L_1}, g_{S_1}), \ldots, (g_{L_n}, g_{S_n})], gnm, grm) =$$

$$D(\ embed_L(g_{L_W}, [g_{L_1}, \ldots, g_{L_n}], grm), \\ embed_S(g_{S_W}, [g_{S_1}, \ldots, g_{S_n}], gnm)\ \ )$$

Both functions will be defined in the following subsections. $embed_S$ will be discussed first.

C.1 Embedding Grammar Productions

To embed language modules into a host language, the productions of all sublanguages have to be appended to the host language's production sequence. Additionally, for every sublanguage a nonterminal is introduced, such that the embedded languages can be referenced by referring to that nonterminal. In general, we assume the sets of nonterminals of all participating languages to be disjoint, except those parts which belong to the component interface.

$embed_S$ takes a host grammar, a list grammar modules to be embedded and a user-specified GNM mapping.

$$embed_S\ :\ G_S \times List\, G_S \times GNM \rightarrow G_S$$

$$embed_S(g_S, \emptyset, gnm) = g_S$$

$$embed_S(g_S, [g_{S1} = [p_1|p_2 \ldots]|g_{S2}, \ldots], gnm) =$$
$$g_S \oplus (gnm(g_{S1}) \rightarrow p_1.nonterminal) \oplus embed_S(g_{S1}, [g_{S2}, \ldots], gnm)$$

C.2 Embedding Lexer Productions

An embedding of lexer productions has to preserve the lexer state DFAs of the modules involved and has to reasonably integrate all lexer automata into a more complex automaton. The result DFA shall allow to switch between the partial automata via explicitly distinguished transitions in composed languages. The functionality of constructing such a complex automaton and how to bring lexer production sequences together is realised by $embed_L$ (to increase readability, the mapping GRM will be left out in all signatures).

$$embed_L\ :\ G_L \times List\, G_L \rightarrow G_L$$

$$embed_L(g_L, \emptyset) = g_L$$

$$embed_L(g_L, list_{G_L}) =$$
$$addtransitions_L(g_L, list_{G_L}) \oplus combine(g_L, list_{G_L}, \emptyset)$$

New edges are added by $addtransitions$ while $combine$ traverses the list of language modules and triggers $addtransitions$ for each module.

Overall, edges between the host language and each module and between the modules are added. In the following both functions are defined.

$$combine\ :\ G_L \times List\ G_L \times List\ G_L \to G_L$$

$$combine(g_L, \emptyset, list_{G_L}) = \emptyset$$

$$combine(g_L, [g_{L1}|g_{L2}, \ldots], list_{G_L}) =$$

$$\begin{aligned} &addtransitions_L(unique\_states(g_{L1}), [g_L] \oplus list_{G_L} \oplus [g_{L2}, \ldots]) \\ &\qquad \oplus combine(g_L, [g_{L2}, \ldots], list_{G_L} \oplus [g_{L1}]) \end{aligned}$$

$unique\_states$ uniquely maps lexer states in the lexer grammars to corresponding states in the composed DFA.

$addtransitions$ implements the addition of transitions to the token set of a language. It introduces token definitions with highest priority to the productions q_i of a given production sequence g_L. Tokens introduced that way are of the type $ignore$, since they are relevant for lexical disambiguation but not for syntax analysis.

$$addtransitions_L\ :\ G_L \times List\ G_L \to G_L$$

$$addtransitions_L(\emptyset, list_{G_L}) = \emptyset$$

$$addtransitions_L([q_1|q_2, \ldots], list_{G_L}) =$$

$$\begin{aligned} &Q_L(addtransitions_R(q_1.g_R, list_{G_L}), q_1.s_A) \\ &\qquad \oplus addtransitions_L([q_2, \ldots], list_{G_L}) \end{aligned}$$

$$addtransitions_R\ :\ G_R \times List\ G_L \to G_R$$

$$addtransitions_R(g_R, \emptyset) = g_R$$

$$addtransitions_R(g_R, [g_{L1}|g_{L2}, \ldots]) =$$

$$\begin{aligned} &(gtm(g_{L1}) \to grm(g_{L1}), ignore, unique\_states(g_{L1}).q_1.s_A) \\ &\qquad \oplus addtransitions_R(g_R, [g_{L2}, \ldots]) \end{aligned}$$

Here, gtm creates a unique terminal name as edge name for the given language.