

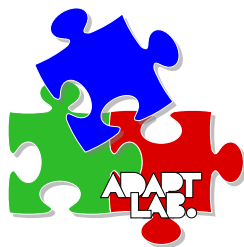
# Programming Languages *à la Carte*

Edoardo Vacchi

Id. Number: R09518

Scuola di Dottorato in Informatica  
PhD in Computer Science

Advisor: Prof. Walter Cazzola



**UNIVERSITÀ DEGLI STUDI DI MILANO**  
Computer Science Department  
ADAPT-Lab



# Contents

<b>1. Introduction</b>	<b>1</b>
<b>2. Background</b>	<b>7</b>
<b>3. Feature-Oriented Language Composition</b>	<b>11</b>
3.1. Conceptual Model . . . . .	12
3.1.1. Language Components . . . . .	13
3.1.2. Dependencies Between Components . . . . .	15
3.1.3. Globally-Scoped Components . . . . .	17
3.1.4. Composition Model . . . . .	18
<b>4. Neverlang</b>	<b>21</b>
4.1. The Neverlang Framework . . . . .	22
4.1.1. Defining Syntax and Semantics: Modules . . . . .	23
4.1.2. Mapping Semantics onto Syntax: Slices . . . . .	30
4.1.3. Combining Slices Together: Generating a Language . . . . .	35
4.2. Runtime Deployment of Semantic Actions and Tree Rewriting DSL . . . . .	38
4.3. Tools and Utilities . . . . .	39
4.4. Implementation . . . . .	43
4.4.1. Architecture . . . . .	44
4.4.2. Runtime and Execution . . . . .	49
4.4.3. DEXTER . . . . .	53
<b>5. Case Study: Evolution of a DSL through Composition</b>	<b>55</b>
5.1. A Simple State Machine DSL . . . . .	56
5.2. A Simple Imperative Language . . . . .	59
5.3. Guards and Actions: Composing the DSLs . . . . .	61
<b>6. Evaluation</b>	<b>67</b>
6.1. Feature-Oriented Language Implementation Across Tools . . . . .	67
6.1.1. LISA . . . . .	68
6.1.2. Silver . . . . .	70
6.1.3. Spoofox . . . . .	72
6.1.4. Xtext . . . . .	74
6.1.5. Summary . . . . .	76

## Contents

6.2. Extending a Real-World Language: neverlang.js . . . . .	80
6.2.1. Runtime Evolution for Dynamic Optimization . . . . .	83
6.3. The DESK Language . . . . .	84
6.4. Tracking Dependencies Through Variability Management . . . . .	89
<b>7. Related Work</b>	<b>93</b>
7.1. Extensible Parser Generators . . . . .	96
7.2. Variability Modeling of Language Families . . . . .	97
<b>8. Conclusions</b>	<b>99</b>
<b>A. Formal Composition Model</b>	<b>101</b>
A.1. Decomposition of Syntax Definitions . . . . .	101
A.2. Decomposition of Language Semantics . . . . .	103
<b>B. On The Relation Between LR Goto-Graphs</b>	<b>107</b>
B.1. Goto-Graphs and Growing Grammars . . . . .	109
B.1.1. Construction of $\varphi$ and $\Delta V$ . . . . .	112
B.1.2. Construction of $\psi$ and $\Delta E$ . . . . .	115
B.1.3. Construction of $\Gamma_{G'}$ from $\Gamma_G$ . . . . .	118
B.2. Goto-Graphs and Shrinking Grammars . . . . .	121
<b>C. Variability Model Inference</b>	<b>123</b>
C.1. Tag Generation . . . . .	123
C.2. Hierarchical Clustering . . . . .	124
C.3. Refinement Procedure . . . . .	125
C.4. Heuristics for Mining Constraints . . . . .	127

# 1

## Introduction

In *Nineteen Eighty-Four*, the *IngSoc* party imposes on the population an artificial language called *Newspeak* «not only to provide a medium of expression for the world-view and mental habits proper to the devotees of *IngSoc*, but to make all other modes of thought impossible»<sup>1</sup>.

Fiction aside, a highly-debated, fascinating hypothesis in linguistics is that the language we speak shapes the way we think. Since the 1930s, the notion that different languages may influence the cognition skills of the speakers has become associated to Edward Sapir and Benjamin Whorf, American linguists who studied how languages vary and conjectured the ways different tongues would affect their speakers. These ideas were initially met with enthusiasm and excitement, yet they were finally stroked by a substantial lack of evidence. But recently, a solid body of new empirical evidence has emerged [10], showing that language has influence even in the most fundamental dimensions of human experience, such as space, time, causality and relationships to others, affecting even memory and learning abilities.

*Programming languages* are quite different from natural languages. Natural languages can be ambiguous, informal and vague; they leave room for creativity and imagination. On the other hand, programming languages are a form of communication between humans and *machines*: they are necessarily *unambiguous*, *formal* and *precise*. Because of this inherent rigidity, not only do these artificial languages enforce strict rules on the way a program should be written, but also impose a *mental model* on the programmers. And, as programmers, we often feel like choosing a programming language over another *frees* our mind from cognitive burden. Edsger W. Dijkstra despised FORTRAN «for as a vehicle of thought it is no longer adequate: it wastes our brainpower, is too risky and therefore too expensive to use» [24]; Paul Graham, Eric S. Raymond and

---

<sup>1</sup>Orwell G., from the appendix "The Principles of Newspeak"

## 1. Introduction

```
val fixedIncomeTrade =  
  200 discount_bonds IBM for_client NOMURA on NYSE at 72.ccy(USD)  
val equityTrade =  
  200 equities GOOGLE for_client NOMURA on TKY at 10000.ccy(JPY)
```

**Listing 1.1:** A trading DSL in Scala (from [38]).

many others have defined learning Lisp «an enlightenment experience» [79, 41]. But even if languages influence our thought, the influence goes the other way, too. In fact, we are constantly looking for ways to *extend* and *enrich* our programming languages with new constructs and powerful abstractions, trying to *close the gap* between the way we are accustomed to think as human beings and the way machines have to be instructed. Modern general purpose languages are striving to provide more abstraction to programmers; mainstream design is progressively converging towards a hybrid between object-orientation and functional programming. Languages from both the communities cross-pollinate each other with features. Languages that were born as purely object-orientated, nowadays tend to include functional constructs.

However, even though the tendency to contamination is strong, programming language implementations hardly share any code. Even *close relatives*, with similar syntax and semantics are usually developed from scratch, using techniques that we would barely call modern; after all, modular programming dates back to the 1970s [75]. Yet, programming language development is often still a *top-down, monolithic* activity, where *extensibility of the compiler*, although desirable, is only an afterthought. Nevertheless, the rising trend of developing programming languages to target a particular *application domain*, to solve a particular problem of that domain, has been cause for a *shift* in the way computer languages are designed and implemented. The model of development where the software system is built from the ground up using a *little language* [7] has been dubbed *language-oriented programming* [107]. In language-oriented programming, *domain-specific languages* (DSLs) are developed to write programs of the application domain in a concise, problem-oriented way. Contrasting this trend to the traditional language development process, in this model language implementation is quite a *bottom-up* activity, where the specification *rises* from the problem that the developers need to solve. As a matter of fact, domain-specific languages are a natural part of our everyday workflow, because they are designed to simplify interaction with specific software systems; at the same time, a well-crafted DSL brings interaction with software even within the reach of *domain experts*, who may not be necessarily professional programmers. Mathematicians may write MATLAB programs; statisticians may use the SAS programming language; a hardware engineer could write in Verilog.

Different DSLs targeting different domains may be used to implement different concerns of the same software system. Therefore, *language composition, extension* and *reuse* are highly-researched topics. For instance, in electronic automation and communication protocols, the transitions between states are usually represented through *state machine languages*; it is not uncommon to combine these languages with a restricted

```
List<Person> persons = query.from(person)
    .where(
        person.firstName.eq("John"),
        person.lastName.eq("Doe"))
    .list(person);
```

**Listing 1.2:** A query using Spring Data's Query DSL [78] in Java.

*imperative programming language* to model actions that are performed when a transition fires. General purpose programming languages often *embed* other domain-specific languages to perform a number of tasks, such as querying databases (using SQL), sending marshaled data over the wire (using JSON) or pattern matching over a string (using regular expressions).

Language composition and extension can be achieved through *embedding* of a *guest* language in a *host* programming language or by implementing the language to be stand-alone through specific tooling. In the first case, the simplest form of embedding is through *quoting*: the foreign programming language is typically represented as a string; this has several downsides, the most obvious of which is the lack of a support for static verification from the host language tooling. A more advanced and modern form of embedding is the *fluent interface* [35]. A fluent interface is a particular API design style that makes syntactically valid lines of code of the *host* programming language *read* like a *foreign* language. Fluent APIs are often used to embed query languages within the body of a general purpose programming language (*cf.*, Spring Data's Query DSL [78], Listing 1.2) or to describe graphical user interfaces (*cf.* JavaFX's APIs [23]). DSL *embedding* through fluent interfaces has become part of the idiom in many modern programming languages such as Scala, Ruby and Groovy (*e.g.*, Listing 1.1). Because of this, embedding is generally the most convenient and widely adopted technique to implement a DSL. Recent research has also shown the benefits of employing the same technique to produce high-performance, code-generating DSLs [82]. However, this technique has its limits. First of all, the syntax of the embedded DSL is inevitably dictated by the *host* programming language. Second, since it may not be possible to *sandbox* the guest language environment, final users may inadvertently end up outside the boundaries of the DSL and in direct contact with the host language.

*External languages*, on the other hand, are usually implemented using *dedicated toolsets*, but they give a much finer-grained level of control to the language developer. There is no more hard limit on the way the syntax is defined, and the language implementation works stand-alone, without a host. The traditional route to this kind of language development is to first implement the front-end either by hand or through parser generators such as yacc, ANTLR [76] or, more recently, parser combinators [93, 66]. Then, the semantics of the language can be implemented using a variety of techniques, ranging from syntax-directed translation [2] and attribute grammars [58, 74] to term-rewriting [104], to model-based engineering [59, 47]. The downside with traditional toolsets is that they do not focus on code reuse.

*Modular language development* is the research branch that investigates tools and tech-

## 1. Introduction

niques to develop languages in a *componentized* way. Frameworks for modular language development are toolsets that enforce a modular implementation strategy on the *language developers*, enabling a high degree of code reuse even in the implementation of stand-alone programming languages. Many frameworks for modular language development have been proposed during the years (*e.g.*, [59, 63, 101, 56, 105, 98]), with different models for code reuse, usually inspired by the primitives provided by general-purpose programming languages, like *inheritance* [63] and *superimposition* [60]. An open research problem [43, 110, 60, 71, 98] is to apply *feature-oriented programming* to language development. Feature-oriented programming is a vision of programming in which individual *features* can be defined separately and then composed to build a variety of *software products*. In language development, *feature-orientation* means developing the features of a programming language *in isolation*, and then being able to transform an arbitrary set of said features into a consistent language implementation. Although most modular language development frameworks deal with the problem of language extensibility and componentization, the idea of realizing a programming language *starting from its features* is usually not a primary objective.

The Neverlang framework was originally born [16, 15, 14] to explore feature-oriented modularization in language design and implementation. In the last three years the Neverlang framework has been object of a large design overhaul [17, 18, 99, 19, 98] that shifted the implementation from a largely static, Java-oriented, code-generating toolset, to a highly-dynamic componentized language-agnostic framework for language processing. The object of our research has been geared towards realizing techniques and tools to implement *componentized language implementations* with the final “grand vision” of a world where general-purpose and domain-specific programming languages can be realized by composing together *linguistic features* the same way we combine together the pieces of a puzzle. And, just like each piece of a puzzle lives and exists on its own, each linguistic feature should be something that we can describe and implement in isolation and separately. The ultimate goal is to maximize reuse of syntactic and semantic definitions across different language implementations. To the point where end users may be even able to generate a language implementation by picking *features* from a curated list. Programming languages *à la carte*.

**Contribution** Most of our experience in feature-oriented definition of programming languages have been carried out using Neverlang. Our contribution with this work is

1. an abstract model for feature-oriented language implementation,
2. a description of our implementation of this model in Neverlang
3. showing that it can be supported by most of the existing tools for modular language implementation,
4. showing that the native implementation of this model strengthens the benefits of a modular language implementation.

**Organization** Chapter 2 gives a brief overview of the background information. Chapter 3 presents the abstract model. Chapter 4 introduces the Neverlang implementation



of this model. Chapter 5 presents a full example (a state machine language). Chapter 6 is devoted to evaluate the model in a variety of contexts: the state machine language is re-implemented in other frameworks to show how the model can be reproduced; the benefits of using this model are then showed by describing the experience of extending Neverlang's JavaScript implementation `neverlang.js`; a DESK language implementation is briefly given to exemplify the expressive power of the Neverlang framework; finally, this section describes the experience of modeling *variability* in programming language family, by automatically mining data from a collection of pre-implemented features. Finally, Chapter 7 briefly discusses related work and Chapter 8 draws the conclusions and describes the future work.

Appendix A is a more formal description of the Neverlang model, that puts in relation the Neverlang implementation with the conceptual model of Chapter 3. Appendix B reproduces part of the formal proofs [19] related to Neverlang's dynamically extensible parser generator, DEXTER. Appendix C is an excerpt from [98] where we give more detail on the variability modeling experience described in Chapter 6.



# 2

## Background

A *context-free* grammar is a tuple  $G = \langle \Sigma, N, P, S \rangle$ , where  $\Sigma$  is an alphabet of *terminal symbols*,  $N$  is an alphabet of *nonterminal symbols*,  $P$  is a set of *production rules* and  $S \in N$  is the *start symbol*. A production rule (or simply a *production*, or a *rule*) is written as  $A \rightarrow \omega$  where  $A \in N$ , and  $\omega \in (\Sigma \cup N)^*$ , with  $(\Sigma \cup N)^*$  being the transitive closure of set  $\Sigma \cup N$  with respect to symbol juxtaposition. The generated language  $L(G)$  of a grammar is the set of all the words that can be *derived* from a grammar  $G$ . A language for a grammar  $G$  is said to be *empty* if  $L(G) = \emptyset$  and, conversely, *non-empty* when it contains at least one sentence. In other word, there exists at least one *sentence*, (or *word*, or *program*) that can be expressed using the language represented by  $G$ . In the following, we will assume grammars that generate non-empty languages, and, for simplicity, we will make the assumption that our grammars do not contain the empty word  $\epsilon$ .

A *syntax-directed definition* [2] (SDD) is a technique to implement the semantics of context-free languages, in terms of their grammar. *Attribute grammars* [58] are a formalism introduced by Knuth to represent SDDs by associating information with a language construct by attaching *attributes* to the grammar symbols representing the construct. Attribute grammars specify the values of the attributes by associating *semantic rules* with the grammar productions. *Syntax-directed translation schemes* (SDTs) are sometimes described as complementary notation to attribute grammars. A syntax-directed translation scheme is a context-free grammar with *program fragments* embedded within production bodies, called *semantic actions*, with the purpose of *translating* a input program written in a given language into a *target language*; that is, SDTs are usually employed to implement *compilers*. Any SDT can be implemented by first building the parse tree that represents the input program, and then performing the actions in a left-to-right depth-first order, that is, during a *preorder* traversal [2]. Typically, SDTs are implemented *during parsing*, without building a parse tree. In this case, two important classes of grammars are

## 2. Background

- *L-Attributed Grammars*, a class of attribute grammars that can be incorporated in *top-down parsing*.
- *S-Attributed Grammars*, a class of attribute grammars that can be incorporated in both *top-down parsing and bottom-up parsing*. Any S-attributed grammar is also an L-attributed grammar.

However, L-attributed and S-attributed grammars are rather limited classes, and many interesting although simple languages cannot be defined using this translation scheme. The main benefit of implementing L-attributed and S-attributed grammars is that the *evaluation order* of the semantic rules is known *a priori*, because they impose constraints on the way semantic rules are defined. In fact, in *attribute grammars* we distinguish between the set of *synthesized* attributes, expressed only in terms of the attributes of the children of a nonterminal symbol, and *inherited* attributes, expressed in terms of the attributes of their ancestors or siblings. The S in *S-Attributed grammars* stands for *synthesized*: this class allows only *synthesized* attributes to be defined. It is the class that traditional parser generators such as yacc support. In *L-attributed grammars*, the *inherited* attributes can be evaluated in one single left-to-right pass.

By relaxing the constraints on attribute evaluation, the attribute grammar formalism becomes more general but also it leaves space for computations that *may not terminate*. In order to give guarantees on the evaluation of the attributes, attribute grammar implementations compute different kinds of *dependency graphs* [58, 74] between attributes and impose different sets of constraints; at the very least, each attribute should be *well-defined*: that is, for each node, an attribute should either be a *constant expression* or it should be defined in terms of *other well-defined attributes* on its parent or its siblings. Further constraints may be imposed to give more guarantees. For instance, one notable class is that of *Absolutely Noncircular Grammars*, which includes both L-Attributed and S-Attributed grammars and it has been shown to be powerful enough to represent many nontrivial programming languages [74]. It is therefore advisable that an attribute grammar implementation supports at least absolutely noncircular grammars.

A strict implementation of an attribute grammar is usually *pure*: that is, attributes should be defined in terms of other attributes, and the evaluation of such attributes should not produce *side-effects*. This gives a greater deal of flexibility to attribute grammar implementations, that may employ a number of techniques to optimize attribute evaluation such as *memoization* (cf. [83]). However, many implementations allow side-effects with varying degrees of control. When arbitrary, possibly *side-effectful* computations are allowed to take place within semantic *definitions*, then we speak more broadly of semantic *actions*. In such cases, automatic caching and memoization of attributes may not be supported, but implementations may overcome this limitation by giving users more control on which attributes are evaluated at a time (as we will see in Chapter 4 this is the case for Neverlang).

Syntax-directed translation through attribute definition is not the only technique to implement languages, though; for instance, languages can be also described in terms of program *transformations*; the Stratego [12] language implements this technique, rewriting *terms* that initially represent the parse tree up until the final representation of

a compiled program is reached.

In order to stress the generality of the approach, Chapter 3 describes a *conceptual* model of feature-oriented language definition without making explicit references to a particular model of language processing. In this model, evaluation phases of the language are modularized in terms of language constructs, in order to represent a language implementation in terms of its constructs. In Chapter 4 we will then delve into the details of our own implementation of this model; in our case the processing model can be modeled after SDDs, as a modular rendition of the *visitor pattern*.

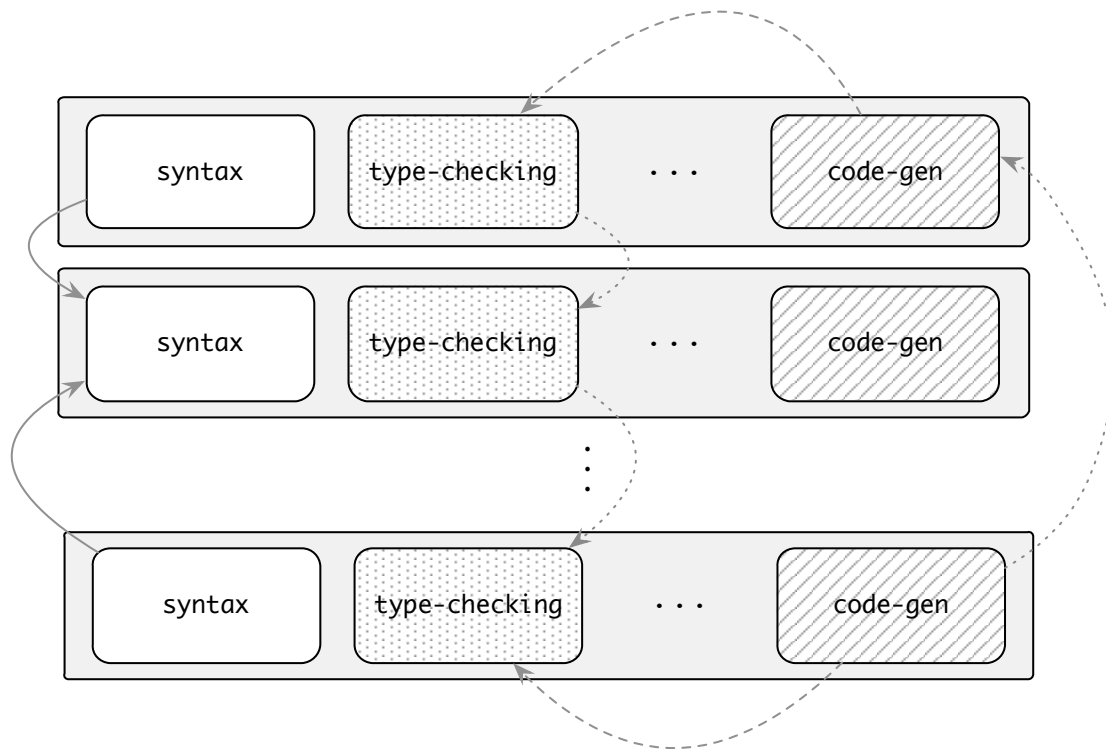


# 3

## Feature-Oriented Language Composition

Frameworks for modular language implementation (e.g., [18, 100, 63, 101, 12]) make componentized development *front and center*, by providing facilities to simplify the implementation of a language in modules that can be shared and reused. But the modularization of a language is not merely a matter of convenience: modular software implementation has been known to be good from the dawn of computer science (e.g., [75]) for a number of reasons; among the others, *component isolation*, which also enables work to be carried out in parallel by different teams of programmers; *modular reasoning*, which make it possible to concentrate on the implementation of the component of a system to be developed independently from the others. To a certain extent this is possible for language implementation as well, and it is very apparent in the development of *domain-specific languages*, where it is easier to map *features* of the language onto *concepts* of the problem domain. Our final objective aims at representing a language as a *collection* of independent *features* that can be easily used in conjunction, but that should be possible to implement without knowledge of one another (cf. *composable extensions* in Van Wyk *et al.* [101]). As seen in Chapter 2, The earliest literature already established that languages can be described in terms of their syntax. It has also been shown (e.g., [1, 32]) that such definitions can be logically *partitioned* into distinct *processing* or *evaluation phases*. During each phase the input program is subsequently analyzed and transformed up to the final phase, when it is finally executed, in the case of an interpreter, or code is generated, in the case of a compiler. For instance, at the time of writing, Scala 2.11's compiler `scalac` performs 25 compilation phases on each input program. This section gives an overview of the concepts behind modular language development, in order to stress the generality of the approach.

### 3. Feature-Oriented Language Composition



**Figure 3.1.:** A language implementation can be broken down over two dimensions: the dimension of syntactic constructs, and the dimension of evaluation phases. Dependencies are represented as arrows from required to provided features. Dependencies may go beyond their phase and depend on other phases.

## 3.1. Conceptual Model

A *syntax definition* of a language may be broken down with respect to language *constructs*. For instance, a *looping construct* may be defined and reasoned upon independently from a *conditional branch*, even though they still depend on a notion of *truth value*. All of these parts of a language together form a complete language implementation; thus, the modularization of both syntax and semantics is pivotal in the componentization of a language implementation. Therefore, there are at least *two dimensions* over which a language implementation can be broken down: the dimension of the *processing phases*, which, more broadly, includes the *syntax analysis* of the input program, and the dimension of the *syntactic constructs* (Fig. 3.1), which logically partitions a language implementation with respect to the constructs that it contains. We call *feature* of a language an abstract concept or construct, with its semantics; in some sense, then, the feature of a language the points at the *intersections* between the two dimensions. The *implementation* of a feature is what we call a *language component* [99, 98].



### 3.1.1. Language Components

*Modular language implementation approaches* enforce separation between *processing phases* and *concepts*. The dimension of processing phases represents the *separation* between *linguistic concerns* [103] that may crosscut or tangle, such as type-checking, code-generation, and so on. This kind of modularization suggests that componentization may be achieved by grouping together, as a self-contained bundle:

1. the *syntax definition* of a construct, such as the keywords that introduce it, and
2. the *sections* of the *evaluation phases* that *relate* to the *syntax definition*, implementing only the semantics that is relevant to that construct, in terms of *properties* of the feature. For instance, the concept of *truth value* may be seen as a *property* of the *looping construct* feature.

By creating a *bundle* of these smaller bricks, the syntax definition and the sequence of the relevant parts of the processing phases, we obtain a *language component* (Fig. 3.1): a higher-level unit of composition that represents the *implementation* of a concrete *feature* of our language. These components can be *shared* across language implementations, and *substituted* to define variants of the same language: for instance, a different implementation of the semantics may be given for the same syntactic construct. For example, let us now consider a simple imperative language with a Java-like `while` loop construct<sup>1</sup>:

```

syntax:
  while ( { loop-condition } ) {
    { loop-body }
  }

```

The *placeholders* in angle brackets represent parts of the syntax that are not defined locally, because they represent concepts that are logically distinct, albeit related.

**Definition 1.** A *placeholder* is a part of a syntax definition that is not defined in place.

Now, a language usually has to be processed through several phases. In this case, suppose these phases are *type checking* to verify the correctness of the construct and *code generation* to output compiled code; since we made the initial assumption that it is possible to modularize phases by breaking them down with respect to the syntax definitions that pertain to a given language construct, we can componentize these phases, with respect to the given looping constructs:

<sup>1</sup>The canonical syntax definition for a Java-like `while` loop would not include braces, and it would be in terms of a `{ statement }` which might possibly be a *block*. For the sake of conciseness and clarity we chose to imagine that a `while` loop is always followed by a braced block

### 3. Feature-Oriented Language Composition

```
type-checking:
  the type-of { loop-condition } should evaluate to a boolean value
  otherwise raise error: " { loop-condition } : bad type "
...
code-gen:
  cond := compiled-code-for { loop-condition }
  body := compiled-code-for { loop-body }
  compiled-code-for this := generate-object-code(cond, body)
```

In the example, the phase contains some bold-face words; these represent *properties* of the language that are being evaluated during that phase. For instance, the type-checking phase evaluates a *property* (in bold face in the code) named **type-of**. This property is bound to placeholder named { loop-condition }. The code-gen phase compiles the input language to object code; thus it expects to evaluate a *property* named **compiled-code-for**. The semantics of a phase can be therefore given in terms of such properties.

**Definition 2.** A *property* is a facet of a feature in an *evaluation phase*.

**Definition 3.** An *evaluation phase* is a definition of a linguistic *concern* in terms of the *properties* of a feature.

A *bundle* of the implementation of all such *phases* together with the syntax definition of the while loop, yields a *language component* for this construct. A *descriptor* for one such component may be:

```
define component while-loop:
  use syntax while
  use phase type-checking
  use phase code-gen
```

Such a bundle can be shared across different language implementation with similar requirements in terms of features and processing phases, thereby maximizing *reuse* of the feature, and minimizing code *duplication*. For instance, a different bundle, reusing the same syntax, but varied semantics, may be reused in an alternate version of the same language to produce a change in the way the language construct are processed and evaluated. *Different* bundles may still share the implementation of some phases. New phases can be deployed by just *extending* or *repackaging* the bundle. For instance, the language component for the interpreter of the while loop may reuse the syntax and the type-checking phases, but it would trade the code-gen phase for an evaluation phase, where the program would be actually executed. Moreover, the same semantics may apply even if the introducing keywords were different. For instance, in a Pascal-like language they would be «while { loop-condition } do begin { loop-body } end».

**Definition 4.** A *language component* is a self-contained bundle that implements a *feature* of a language by putting in relation a *syntax definition* with the related parts of a series of *evaluation phases*. A language component may be shared across different language

Linguistic Feature	A concept or an abstract construct of the language
Syntax Definition	Definition of the syntax for a language construct. A syntax definition may be defined in terms of other syntax definitions using <i>placeholders</i>
Syntactic Placeholder	A part of a syntax definition that is not defined in place. It constitutes a reference to a <i>class</i> of syntax definitions.
Evaluation Phase	Definition of the semantics of the language with respect to a particular concern (e.g., type-checking, code-generation, etc.) in terms of <i>properties</i> of a feature
Semantic Property	A facet of an evaluation phase that is bound to the implementation of a feature
Language Component	A self-contained component that implements a <i>feature</i> of the language by putting in relation a <i>syntax definition</i> with the relevant parts of the <i>evaluation phases</i> . In this sense, we can say that a language component <i>provides</i> the implementation of a feature. The language component may also <i>require</i> other language features to work.
Dependency	A feature that is <i>required</i> by a component, and that it is not defined within that same component. A dependency in a <i>language</i> is <i>unsatisfied</i> if there is a component that <i>requires</i> a feature that no components <i>provide</i> . A dependency may be <i>syntactic</i> if it is expressed within a <i>syntax definition</i> by <i>placeholders</i> , or <i>semantic</i> if it is expressed in an evaluation phase by <i>properties</i>
Globally-Scoped Component	A component that implements a concern that should be available to any <i>language component</i>
Language Implementation	A collection of <i>language components</i> where every dependency is satisfied

**Table 3.1.:** Summary of the informal definitions in this section.

implementations. Syntax definitions and evaluation phases may be shared across different language components.

Now that we have defined language components, we can say that a *language implementation* is a set of said components, and the evaluation of a program written in this language corresponds to the ordered execution of the evaluation phases. This realizes a model of language implementation *à la carte*.

One important detail must be still discussed to complete the description of the model, though; in order for the set of language components to qualify as a true language implementation, it is necessary that all the *dependencies* between these components are *satisfied*.

### 3.1.2. Dependencies Between Components

In the previous example, the *while* loop syntax definition includes *placeholders*, and processing phases includes *properties*. Placeholders represent in some sense *features* that the component *requires*, but it does not *provide* in itself. Similarly, *properties* represent

### 3. Feature-Oriented Language Composition

*facets* of the feature that the language processor should be able to evaluate. The *while* component provides syntax and semantics for the looping construct, but it implicitly *relies on* the definition of some parts of the syntax to be available (presumably in other components). The semantic phases implicitly *expected* that some properties were defined with respect to a given feature. For instance, the type-checking phase was trying to query a type-of property that was expected to be defined on the feature represented by the `{ loop-condition }` placeholder.

The fact that a component may not contain all of the logic that is needed to process the language is the very essence of modularization: each component is defined in such a way that some concern will be *eventually* implemented by some *other* component. This implicitly introduces a notion of *required* and *provided* feature in a language implementation.

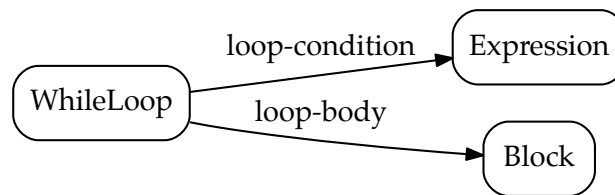
**Definition 5.** A component *requires* a feature if it contains a *placeholder* in its syntax definition, or it relies on the definition of a *property* that relates to a different feature. A component *provides* the feature it implements.

For instance, in the previous example the *while*-loop component *provided* an implementation for a looping construct (the *while* loop), and *required* the concepts of *loop condition* and *loop body* (Fig. 3.2). The type-checking phase required that the *loop condition* could be evaluated to a boolean type, and the code-gen phase required that the condition and the body were compilable down to machine code, so that the result of their compilation could be combined into the compilation of the loop construct itself. In a certain sense, the set of *provided* and *required* properties and placeholders define the *interface* (in a OOP-sense) of the language component. A complete language implementation should *satisfy* each requirement with the implementation of a feature.

**Definition 6.** A *language implementation* is a collection of *language components* where all the *dependencies* are *satisfied*.

**Definition 7.** The *dependencies* of a *language component* are *satisfied* in a *language implementation*, if, for all the *required* features of the component, there exist a component in the language implementation that *provides* that feature.

It is worth noticing that *more than one component may satisfy the same requirement*: if this does not introduce a contradiction, that is, two components provide a feature that is logically contradicting the other, then the two components represent an *alternative choice* in the language implementation. Therefore, imposing that a complete language implementation requires all of its requirements to be satisfied, does not prevent further language *extensions*. Again, for the case of the *while* loop, a language implementation *must* include the language component that satisfies all the requirements that the *while* loop has on the *loop condition* and *loop body* (e.g., the property *type-of* and *compiled-code-for* that the component expects to be able to query). Of course, the *loop body* may be implemented by several kinds of *statements* (e.g., function invocation, variable assignment, variable increment, etc.): each statement is not logically in conflict with the other, although they may appear in the same position in a program written



**Figure 3.2.:** Dependencies between syntax definitions of the language components in the `while` loop example.

using our language. On the other hand, different, alternative implementation for the *loop condition* may or may not be acceptable, depending on the language designer's choice; for instance, the C programming language expects the *loop condition* to be a numeric value, treating it as *true* if non-zero, and false otherwise. Another programming language (e.g., Java) may have similar syntax but enforce the existence of a *boolean* type.

Dependencies may occur *within* the same evaluation phase, or *across* different evaluation phases. In the first case, a *language component* depends on another because the implementation of an evaluation phase is *distributed* across different language components. In the second case, an *evaluation phase* depends on a value that is computed within a *different* evaluation phase: this is reflected in a dependency between components, because evaluation phases are distributed across them. Depending on the way phases are evaluated, dependencies across phases may also impose an *ordering relation* on the evaluation phases (cf. [1] on attribute grammars), because a property in a phase may be only referred to within the *same* phase or a *subsequent* phase.

### 3.1.3. Globally-Scoped Components

In a language implementation components may also need to invoke *support functions* (e.g., I/O, math or graphic libraries) and *ancillary data-structures* (e.g., symbol tables, function tables, etc.). These implement *features* of the language that *do not* have a direct representation in the *syntax* of the language; thus a modular language framework should include a form of component that encapsulates and *provides* support code to other language components. Similarly to the other language components, *globally-scoped components* should be easy to swap with alternate implementations, provided that the substitute component implements the same functions and structures (in OOP-terms, it implements the same interface). For instance, a thread-based model of tasks execution could be swapped with a distributed execution model without changing the syntax of the language (cf. the Linda-Python language in [14]), by swapping the component that implements the task execution model.

### 3. Feature-Oriented Language Composition

#### 3.1.4. Composition Model

Each component *provides* and (optionally) *requires* the implementation of a feature. Composition between components is therefore consequence of satisfying the constraints that are implied by such dependencies. Let us now see which forms of language composition apply to the model that we have described so far. In order to better discuss languages and language implementation frameworks, Erdweg *et al.* [29] have isolated and described five forms of language composition.

- *Language Extension* is the property of a framework to define reusable *components* that may extend a *base language*, independently from the choice of this base language.
- *Language Restriction* is the property of a framework to *restrict* a language implementation to a subset of its features.
- *Language Unification* is the property of a framework to merge together the implementation of two languages by the (optional) help of glue code only.
- *Extension Composition* describes the ability of a framework to compose together extensions (which may only implement subsets of a language).
- *Self-Extension* is the property of a *programming language* that make it possible to extend itself reflectively.

A language *extension* may be a new syntactic feature for a *base* programming language. For instance, Java 8 introduced *lambda* expressions; these were mapped onto the more general case of *functional interfaces* (single-member interfaces) [40, Sect. 9.8]. Language extensions are often defined in terms of *desugaring* towards the base language. Language *restriction* may be useful in education: Erdwed *et al.* suggest to forbidding *monads* and *type classes* in a beginner’s course on Haskell. In language *unification*, as opposed to *restriction* and *extension*, where a *dominant* language exists, the composed languages are composed in an «unbiased manner» [29], and the two languages can interact: one example of this kind of composition may be HTML and JavaScript. Finally, *self-extensible* programming languages are those where a language may be embedded within a host language, through support from the host language itself. Lisp may be regarded as one such language.

Our model supports the following forms of composition:

- ✓ *Language Extension*. In our model a language implementation corresponds to a *collection of language components* such that all their requirements are satisfied. An extension to one such language is a *collection of language components* that *provides* additional implementation that other components *require* (e.g., in the `while` example, additional implementations of the `{ loop-body }` place-holder).
- ✓ *Language Restriction*. Erdweg *et al.* [29] present language restriction as a useful functionality (e.g., in the education area) that can be easily simulated using language *extension* alone, by deploying an extension to the validation phase of the language that rejects any program using “restricted” constructs. Even though the model that we present may very well implement language restriction in the same

way, by redefining phases of the existing components, our model supports “real” language restriction by *unplugging* components from the language: in fact, being a language a set of components, in our model a restriction is a subset of the original collection where the restricted feature is not present.

- ✓ *Language Unification.* In general, because a language is a set of components, language unification would be the union of two sets of such components, plus, if needed, *glue code*, that is, code that “bridges” components that otherwise would not go well together. For instance, in our example, the `while` loop required a `{ loop-condition }`. Another language (e.g., an expression language) may provide a component for a `{ boolean-expression }`. The name of the placeholders does not match: glue code would be that code that adapts a `{ boolean-expression }` to satisfy the requirements of a `{ loop-condition }` in the `while` loop. These requirements may be purely syntactic, for instance the placeholders may have different names; but these requirements may be also semantic, for instance the semantic properties of `{ boolean-expression }` may be different from those required by the `while` loop component. The language framework should provide ways to adapt language components to suit these situations. The glue code might be implemented as additional components, or as directives that developers would configure.
- ✓ *Extension Composition.* In our model, the unit of reuse is the *language component*, which may implement language *extensions* or *parts* of a *base language*, depending on the point of view. It follows that extensions may compose. Erdweg *et al.* themselves do notice that if a framework supports *language unification*, then it also supports *extension composition*, which is the case.
- ✗ *Self-Extension.* This property does not apply here but only because this is a property of the *programming language* and not of the framework with which the language is being implemented. As noticed by Mernik in [63], the model itself does not prevent from implementing a self-extensible programming language.





# 4

## Neverlang

In Chapter 3 we presented a general model to represent language features as *language components*. The objective of such a model is to describe a language in terms of its features, by modularizing the implementation over both the axis of *evaluation phases* and the axis of *language constructs*. The result is a self-contained unit of composition, called the *language component*, that can be *reused*, *shared* or *substituted* in different language implementations. The final goal is to be able to implement a language by combining such components, making it easy to *extend* the language implementation by adding new features, *restricting* it for DSL purposes, *vary* it to accommodate different requirements in the application domain, and, in general, *evolving* the language implementation over time.

The problem of evolving a language implementation is known. In a typical interpreter or compiler implementation, each construct of the language is mapped onto an *abstract representation* often called an *abstract syntax tree* (AST), over which different language *processors* or *evaluators* dispatch the execution of procedures that implement the semantics of that construct. During the visit of this tree, a *language evaluator* or *language processor* maps each of its nodes onto the semantics of the constructs that the node represents, depending on its type. In the case of an *interpreter* the input program will then be *executed*, while, in the case of a *compiler*, the input program will be translated into a target language. As we saw in the previous sections, the semantics of a construct may be implemented as *several separate phases*; multiple phases enable to better modularize the implementation of the semantics of each construct. Nevertheless, for better modularity, even the definition of each construct would better be isolated from the definition of other constructs.

However, the *evolution* of a language implementation involves both the dimension of constructs (that may be represented by distinct data types) and the dimension of evaluation phases (data type processor): neither *functional* nor *object-oriented* program-

#### 4. Neverlang

ming languages can fully address the problem. In *functional programming* it is easy to vary the set of phase evaluators that pattern match on the different *cases* of a data type (e.g., all the types of *loops*, or all the types of *statement*). On the other hand, it is harder to vary the number of cases in a data type definition. The situation is known as *the expression problem*, after the term coined by Philip Wadler [106].

In *object-oriented programming* it is easy to *extend* a class, defining a new subtype, but, on the other hand, it is harder to add new language processors on the data type, because an idiomatic object-oriented program would implement such processors using the *interpreter* pattern, where the super-class define an abstract method `eval()`, implemented by the concrete subclasses. This problem can be partially addressed by implementing the so-called *visitor* pattern where an extra-level of indirection is introduced (through a visitor object) to make the semantics of each construct independent from the class that represents that construct; on the other hand, the *visitor* pattern make it, again, easier to add new operations and harder to add new types: it is effectively the object-oriented implementation of pattern matching [37].

It has been shown that a modular implementation of the visitor and interpreter patterns [71, 70, 112] can be achieved using constructs such as *traits* [84] to decouple the data type representation from the logic that implements the semantics of a construct, while still retaining all the good properties of object oriented programming, that is, the ability to extend the data type with new sub-types. Our rendition of the model in Chapter 3 can be seen as an implementation of a *modular visitor pattern*, which is the underlying execution model of the Neverlang framework.

### 4.1. The Neverlang Framework

In Chapter 3, high-level descriptions for *syntax definitions* and *evaluation phases* were discussed. In Neverlang, the syntax of the language is given as a *formal grammar*, and the semantics of the language is given as a syntax-directed definition (Chapter 2), in terms of *attributes* attached to the *nonterminals* of this grammar. In Neverlang, an *evaluation phase* is called a *role*; a role implements the semantics of the language with respect to the syntax definition of the *language constructs*. Both *roles* and *syntax* definitions are declared inside *modules*. *Language components* (Chapter 3) are defined by a construct called *slice*, which relates syntax definitions to roles imported from modules; globally-scoped components are called *endemic slices*; endemic slices may provide *libraries* or globally-accessible data-structures such as *symbol tables*. A construct called `language` declares the collection of slices that composes a language and the order in which *roles* should be executed. The syntactic definitions generate a syntax tree, which is then *visited*. Each visit constitutes an *evaluation phase*. Contrary to a traditional visitor pattern implementation, though, Neverlang's visitor is extensible both on the dimension of *processing phases* and on the dimension of new *language constructs*. In fact, slices compose semantics from different modules, making it possible to define new roles (processing phases) for the same linguistic construct; but slices can be added, removed or replaced to a language implementation at any time: therefore, the language

can evolve in any direction.

In the following paragraphs we will present modules, roles and slices using the *Neverlang language* syntax: a domain-specific language that simplifies the implementation of these constructs in a convenient, uniform way. The *Neverlang language* is a DSL that compiles down Neverlang source files to Java and JVM-compatible source-code. The `nlgc` compiler is self-hosted and will be described in Sect. 4.3. The generated source code will be described in Sect 4.4, where the framework and its APIs are described in detail. These APIs have been designed to be easy to use even using a general-purpose JVM-supported programming language. The Neverlang language is just one of the possible front-ends to this API. For completeness, Listing 4.1 is the full grammar of the Neverlang language (EBNF operators were used for conciseness). For a more formal description of the Neverlang framework, see Appendix A.

#### 4.1.1. Defining Syntax and Semantics: Modules

A *module* is a basic container unit that groups different *roles* together, defined in terms of a *reference syntax* declaration. A module may hold any number of roles, but each module must at least include a *reference syntax* declaration.

**Reference Syntax.** The *reference syntax* section is the section of a module to define the syntax of a construct (Chapter 3). The *reference syntax* section either *defines* or *refers* to a set of *production rules* of a BNF grammar. When it *defines* production rules, it is a bracket-delimited *block* that contains a list of production rules; when it *refers* to another syntax, it is substituted by the clause **from** <modulename>, where *modulename* is the name of the module that contains the list of productions that is being referred.

In a production, unquoted identifiers represent *nonterminals* and quoted identifiers represent *terminals*. Special syntax for *patterns* is also provided, to represent classes of terminals such as identifiers or numbers. In this case, instead of quotes the traditional Perl-like syntax for regular expression literals is used. For instance the literal `/[a-z]+/` matches one or more alphabetic characters. Neverlang provides full support to Java's Pattern library.

The set of production rules in the reference syntax section represents the *concrete syntax* of a construct the semantic roles will be coded against. It is a *reference syntax*, though, because the roles that are defined in terms of this syntax are not required to be always bundled with this same syntax. The framework makes it possible to code against one *reference syntax* and then ship with a different *concrete syntax*, provided that a mapping between the two is possible. In Chapter 3 the `while` loop could have been defined with a Java-like syntax, using braced blocks (Listing 4.2), or using a Pascal-like syntax (Listing 4.3). Because one syntax definition is basically isomorphic to the other, modulo the terminal symbols, they can be easily swapped: coding against Java-like syntax really makes little difference compared to coding against Pascal-like syntax. In this sense, Neverlang's *reference syntax* can be seen as a sort of «abstract syntax with defaults». For instance, the production in Listing 4.2 can be thought of as representing a tree node `WhileLoop(LoopCondition, LoopBody)`. We will see more on the mapping between syntax and semantics later, when we will describe *slices*.

#### 4. Neverlang

```
CodeUnit ← Unit* ;
Unit ← Module | Slice | EndemicSlice | Language | Bundle;

// module
Module ← LangAnnot? "module" QualifiedId "{" ReferenceSyntax Role+ "}";
// module: reference syntax
ReferenceSyntax ← "reference" "syntax" ( SynFrom | SynDef );
SynFrom ← "from" QualifiedId;
SynDef ← "{" Provides? Requires? Production+ "}";
Provides ← "provides" TaggedNonterminals;
Requires ← "requires" TaggedNonterminals;
TaggedNonterminals ← "{" ( Nonterminal ":" Tag* )+ "}";
Production ← Nonterminal " ← " ( Nonterminal | Terminal )+ ";";

// module: roles
Role ← "role" "(" Id ")" LangAnnot? "{" SemanticActionDef* "}";
SemanticActionDef ← ( Integer | Label ":" ) <LangAnnot> CodeSection;
CodeSection ← ".{" CodeBlock "}" | "@{" CodeBlock "}" ;
LangAnnot ← "<" Id ">";

// slice
Slice ← "slice" QualifiedId "{" ConcreteSyntax ModuleImport+ "}";
ConcreteSyntax ← "concrete" "syntax" "from" QualifiedId;
ModuleImport ← "module" QualifiedId "with" "role" Id+ Remap?;
Mapping ← "mapping" "{" ( MappingDef ( "," MappingDef ) * ) "}";
MappingDef ← Integer " ⇒ " Integer;

//endemic slice
EndemicSlice ← "endemic" "slice" QualifiedId "{" Declare "}";
Declare ← "declare" "{" Declaration+ "}";
Declaration ← "static"? Id ( ":" LongId | ".{" CodeBlock "}" );

// language
Language ← "language" QualifiedId "{"
    LangSlices LangEndemic LangRoles LangRenames "}";
LangSlices ← "slices" ( QualifiedId | "bundle" "(" QualifiedId ")" )+ ;
LangEndemic ← "endemic" "slices" QualifiedId+ ;
LangRoles ← "roles" "syntax" ( "<" Id ( LangVisitOp Id ) * )? ;
LangVisitOp ← "<" | "<" "+" | ":" ;
LangRenames ← "rename" "{"
    Nonterminal " → " Nonterminal ( "," Nonterminal ) * ";" "}";

// bundle
Bundle ← "bundle" QualifiedId "{" LangSlices LangEndemic LangRenames "}";

// common lexemes
QualifiedId ← Id ( "." Id ) * ;
Nonterminal ← Id ;
Tag ← Id ;
Terminal ← SimpleTerminal | RegexTerminal ;
Id ← <unquoted identifier> ;
SimpleTerminal ← <quoted string> ;
RegexTerminal ← <perl-like regex literal> ;
Integer ← <integer number> ;
CodeBlock ← <parsing delegated to translator plugins> ;
```

Listing 4.1: Complete EBNF grammar of the Neverlang language.

```

module javalang.WhileLoop {
  reference syntax {
    While:
      WhileLoop ← "while" "(" LoopCondition ")" "{" LoopBody "}" ;
  }
  role (type-checking) {
    0 .{ // opt.: 'While:' or 'While[0]:' instead of '0'
      eval $1;
      if ( $1.type != Boolean.class ) // opt.: $While[1] instead of $1
        throw new Error("The type of LoopCondition should be a boolean value");
    }.
  }
}

```

**Listing 4.2:** *reference syntax* for the while statement and the type-checking. The name of the module and the left-hand nonterminal are generally not required to match.

```

module pascallang.WhileLoop {
  reference syntax {
    WhileLoop ← "while" LoopCondition "do" "begin" LoopBody "end";
  }
}

```

**Listing 4.3:** *reference syntax* for a Pascal-style while statement.

The *reference syntax* section contains a list of production rules between braces (see Listing 4.2). Each production may be optionally introduced by a *label*, that may be used in role definitions. Roles may also be defined in *different modules*, but new processing phases can be still described in terms of the same piece of syntax. In this case, the programmer should indicate that the roles in the module refer to a syntax definition that has been defined in a different model, using the **reference syntax from** clause. For instance, Listing 4.4, declares that the reference syntax definition is the one in module `javalang.WhileLoop` (Listing 4.2).

Concerns about readability could be raised: using the **reference syntax from** clause, the syntactic definition may not be present locally to a module where semantics is given. Nonetheless, the same could be said for any OOP language, where subclasses do not show the members that they are inheriting, unless these are overridden. The solution to this problem may be *better tooling*; we are currently working on IDE technologies that may assist users by providing visual clues about the syntax definition that has been referenced.

Additionally, this section can be decorated with *optional* metadata about the *intended meaning* of the syntax, using *tags*. The *provides* and *requires* sections may be the first statements in a **reference syntax**. Each line of the section is constituted by a *provided* nonterminal (on the left-hand side of a production) or a *required* nonterminal (on the right-hand side of a production), followed by a list of tags. Listing 4.5 shows an

#### 4. Neverlang

```
module javalang.WhileLoopCodeGen {
  reference syntax from clang.WhileLoop
  role (code-gen) {
    0 .{
      String labelLoop      = Utils.genUniqueLabelName();
      String labelExitWhile = Utils.genUniqueLabelName();

      eval $1;
      String comparatorCode = $1.code; // if* <labelExitWhile>

      eval $2;
      String bodyCode = $2.code;      // body of the loop

      // output
      $0.code = '''
        ${labelLoop}:
          ${bodyCode}
          ${comparatorCode} ${labelExitWhile}
          goto ${labelLoop}
        ${labelExitWhile}:
          nop
      ''';
    }.
  }
}
```

**Listing 4.4:** An example code generation role, generating Java bytecode in Jasmin<sup>1</sup> syntax.

```
reference syntax {
  provides {
    WhileLoop ← loop, statement;
  }
  requires {
    LoopCondition ← truth-value, boolean-expression, expression;
    LoopBody ← statement, statement-list;
  }
  WhileLoop ← "while" "(" LoopCondition ")" "{" LoopBody "}" ;
}
```

**Listing 4.5:** provides and requires sections..

example for the while loop. A use case for tags will be discussed in Sect 6.4.

**Roles.** A *role* section defines the part of a *processing phase* that pertains to the *reference syntax*. A processing phase is implemented as a *tree traversal* of the syntax tree that represents the input program. Each role in a module is identified by a *name*. The name of the role is *user-defined*, and names *do not* have a special meaning. Obviously it is advisable to choose meaningful names and follow general conventions; type checking phases may be usually called *type-checking*; code generating phases might be called *compilation*, or *code-gen*; *evaluation* phases that actually *execute* the program shall be

called evaluation, execution and so on. As seen for SDT (Chapter 2), the semantics is specified by *semantic actions*, a snippet of code that should be executed when a node of the syntax tree is being *evaluated* (visited). A role is therefore a collection of semantic actions pertaining to a given reference syntax. Thus, a *visit* of the tree is described by the collection of all the semantic actions of a role in all the slices that constitute a language.

A semantic action is represented by a *code block* enclosed within the delimiters `.{` and `}.` and introduced by a *number*. The mapping between nodes of the tree and semantic actions is given through these numbers: each *nonterminal* can be referred from a role using its ordinal position inside the *reference syntax* section, starting from 0. Thus, action number 0 will be executed when the 0-th nonterminal of the reference syntax will be visited, action number 1 when the visit will move to the 1-st nonterminal, etc. For instance, in Listing 4.2 `WhileLoop` is 0, `LoopCondition` is 1, and `LoopBody` is 2; thus the action from role `type-checking` is being attached to the root node `WhileStatement`<sup>2</sup>. Because of the reference/concrete syntax duality, terminals are excluded for this count. First, because, being a leaf, it does not make sense to descend into a terminal node, second, this scheme makes it easier to *remap* semantic actions onto different syntactic definitions, because it is independent from the naming of the nonterminals.

Inside actions, it is possible to access any other nonterminal *within the same rule*<sup>3</sup> using the same numbering scheme; in this case nodes are referred through their identifying number preceded by a dollar sign; it is possible to read and attach attributes to nonterminals using a familiar dot notation (Listing 4.2 and 4.4). The type of the attribute is defined implicitly at each use-site. For instance, `$0.foo = "hello"` defines `foo` as a `String` attribute with value "hello". Similarly, `String foo= $0.foo;` is pulling a `String` value from the `foo` attribute. Invalid attribute uses (*e.g.*, mismatched types or undefined values) will cause the system to raise an exception.

Attributes that are attached to nonterminals are similar to instance fields of a class. Each nonterminal may hold as many attributes as desired, and each attribute may be of any JVM type. Attributes are implicitly defined after the first assignment and, once they have been defined, they can be referred from any semantic action associated to any of the nonterminals in the same production.

Actions are written using a JVM language. The default is Java (with some minor syntactic extensions), but programmers may opt-in to use a different JVM language using *language annotations*. Each section of a module that contains code may be annotated to switch to an alternative language; this can be done on a per-module, per-role, or even per-action basis (Listing 4.6). We will see more on how actions are compiled in Sect. 4.4.

**Labels.** The reasons for the choice of a numbering scheme instead of a naming convention to indicate syntax definitions is mostly a matter of history. Neverlang's original implementation [16, 15] followed the same convention, which was inspired

<sup>2</sup>In this example we assume that identifiers are collected into a symbol table during the type-checking phase; in real language implementations, a separate phase may be introduced

<sup>3</sup>*e.g.*, consider grammar  $A \leftarrow B$ ;  $C \leftarrow D$ : rules 0, 1 may refer either 0, 1, but not 2; etc.

#### 4. Neverlang

```
<scala> // switches to the scala language on the whole module
module com.example.MultiLang {
  reference syntax from javalang.WhileLoop
  role(type-checking) {
    0 .{
      eval $1
      // if we use Scala, then we could model $1.type with Either
      val t: Either[Class[_],Error] = $1.type
      t match {
        case Right(type) if type == classOf[Boolean] => ...
        case Left(err) => ... // an error occurred...
      }
    }.
    1 <java> .{ /* switch back to Java here */ }
  }
  // in the template language, everything is a string, unless
  // it is inside {{ ... }}; the result is attached to $0.Text
  role(code-gen) <template> {
    0 @{ // pre-evaluates the child nodes, see paragraph "Driving the Visit"
      loop:
        {{ $1.code }}
        {{ $2.code }} exit
        goto loop
      exit:
        nop
    }.
  }
}
```

Listing 4.6: Using multiple languages in a module.

from venerable tools such as YACC. Since those days, Neverlang has undergone a major rewrite, but the basic principles and syntax remained faithful to the original implementation. The current incarnation of the Neverlang framework provides a way to *label* production rules in the *reference syntax* section.

Listing 4.2 shows that the rule could be defined for label `While:`, which would then be resolved by the Neverlang compiler `nlgc` (Sect. 4.3) as 0. Labels can also be used to refer to every nonterminal of a labeled production using the offset notation `While[n]`, counting from 0. However, since syntax sections are supposed to pertain to one single construct, they usually should not contain more than 2-3 productions at a time; this is the reason why sometimes it might be still more convenient to use the legacy numbering scheme, rather than labels. Of course, labels support should not be seen as an invite to write longer syntax sections, but rather, as a convenience to enhance code readability. Our guidelines for syntax definitions is to keep them short and small, so that they can be shared more easily across language implementations. In fact, as a syntactic definition gets large, it may become more specific to a particular



```

// code-gen, using the post-order shorthand
0 @{
  // eval $1, eval $2 are implied
  String labelLoop      = Utils.genUniqueLabelName();
  String labelExitWhile = Utils.genUniqueLabelName();
  String comparatorCode = $1.code;
  String bodyCode       = $2.code;
  $0.code = ...
}.
// using the eval-and-get shorthand
0 .{
  ...
  String comparatorCode = $1:code; // eval $1; then return $1.code
  String bodyCode       = $2:code; // eval $2; then return $2.code
  $0.code = ...
}.

```

**Listing 4.7:** *Syntactic sugar to execute a post-order visit.*

language implementation, hampering its reusability.

In any case, the planned work on IDE technologies should help in ruling out all the typical shortcomings of the numbering scheme (e.g., rule insertion, refactoring, etc.). Moreover, as we will see in Chapter 4.4.1, the Neverlang API is powerful enough that users may even define custom semantic action loading strategies for modules.

**Driving the Visit.** Users may explicitly descend into child nodes of the tree using the **eval \$N** statement—where **N** is the identifier of a child of the root node of the production that is currently being evaluated, or the root node itself. For instance, consider Listing 4.2, when the type-checking phase will be evaluated for the while loop, at some point, the visitor will descend into the node `WhileLoop(LoopCondition, LoopBody)`: this will trigger the execution of action 0. The first statement of this action is **eval \$1**, which triggers the visit of node 1 (`LoopCondition`). This will execute any action attached to the node of type `LoopCondition` to be executed. Once the visit terminates, control is returned to action 0, which then proceeds to test if the attribute type does not equal to `Boolean.class`, and so on. Similarly, action 0 in code-gen role (Listing 4.4) first visits nodes `LoopCondition` (**\$1**) and `LoopBody` (**\$2**), and then it pulls the attributes **\$1.code** and **\$2.code**, which are then used to generate the attribute **\$0.code** of the `WhileLoop` node, which represents the compiled bytecode (in Jasmin format) of the while loop.

Because the pattern of descending into the child nodes and then evaluating the root node might be frequent—in compilers it is often the norm—Neverlang supports some *syntactic sugar* to shorten the code in such situations (Listing 4.7). It is possible to mark a rule with the **@** modifier, which means “first descend, then execute”, that is, it makes the visit effectively “post-order” [2]. It is also possible to refer a nonterminal using the *eval-and-get* shorthand **\$1:attribute** which is compiled to an **eval \$1** statement and an attribute access **\$1.attribute**. It is also possible to mark an entire role as *post-order*

#### 4. Neverlang

```
slice javalang.WhileLoopSlice {  
  concrete syntax from javalang.WhileLoop // alt.: pascalang.WhileLoop  
  module javalang.WhileLoop with role type-checking  
  module javalang.WhileLoopCodeGen with role code-gen  
}
```

**Listing 4.8:** *Slice implementing a bytecode-generating while loop feature for the Java language.*

in the **language** descriptor (see Listing 4.1.3). You may also have noticed (Listing 4.4) that Neverlang’s Java code blocks provide a special extended syntax for multi-line strings, deliberately reminiscent of Xtend’s *template expressions* [8] (see also Sect 6.1.4).

Visits can also be terminated abruptly by *raising errors* or using a special Neverlang *signal*, useful to return from a procedure or break out of a loop: the command that terminates a visit abruptly is `$terminate`. To raise an error, a Java `RuntimeException` or an `Error` can be thrown as usual (*e.g.*, see Listing 4.2). For more information on the implementation of the `$terminate` command see Sect. 4.4.2.

Finally, Neverlang has experimental support for *suspending* and *resuming* the execution phase. In this case the statement `$suspend` interrupts the execution of the current visit, proceeds to the following (possibly, up to the last), and then automatically, when all the remaining visits are terminated, or —typically— programmatically, using the `$resume`; statement, it *resumes* execution from the suspended phase. The idea with the `$suspend`; and `$resume`; commands is to be able to *untangle* evaluation phases: for instance an interpreter may require type-related information that is only known at runtime. The *type-checking phase* could be partially executed statically, before evaluation and then *suspended* up to when this information is available at runtime (for a use case, *cf.* Linda-Python in [14]).

#### 4.1.2. Mapping Semantics onto Syntax: Slices

A *slice* contains the definition of a single, individually implemented component of the language. A component is defined in terms of the modules that contains the syntax definition that represents the language construct and the roles that implement its semantics. Each slice must import a *reference syntax* from a module, and may import as many roles as desired. Once used in a slice, the *reference syntax* is called a *concrete syntax*. For instance in Listing 4.8, the `javalang.WhileLoopSlice` is being defined. The *reference syntax* from the `javalang.WhileLoop` module (Listing 4.2) is used as the *concrete syntax* for the language component; the semantics that will be used are the type-checking role in `javalang.WhileLoopTCheck` (Listing 4.2) and the code-gen role in the `javalang.WhileLoopCodeGen` module (Listing 4.4). Nonetheless, the same roles could still apply to the reference syntax in module `pascalang.WhileLoop` (Listing 4.3), because the nonterminals for the C-like syntax are trivially mapped onto the nonterminals for the Pascal-like syntax.

Another interesting use case has been described in [18]; the *Recipe DSL* is a language

```

module javalang.DoWhileLoop {
  reference syntax {
    DoWhileLoop ← "do" "{" LoopBody "}" "(" LoopCondition ")" "while" ";";
  }
}
slice javalang.DoWhileLoopSlice {
  concrete syntax from javalang.DoWhileLoop
  module javalang.WhileLoopTCheck with role type-checking mapping {
    1 ⇒ 2, 2 ⇒ 1
  }
  module javalang.DoWhileLoop with role code-gen
}

```

**Listing 4.9:** Remapping part of the while implementation onto the do-while syntax.

inspired by Microsoft’s on{X}<sup>4</sup>, an application to control Android smartphones so that they can react to particular events with user-defined actions. These actions are developed through a JavaScript API, but pre-defined *recipes* can be shared, selected and deployed to the user’s phone through the application website. *Recipe* brings the idea further: it is a DSL whose syntax resemble natural language, to define rules of the form “when X happens, then do Y”. A different syntax definition may be used to *translate* the English keywords into other languages. For instance, the paper shows Italian. Using the *remapping* feature it would be even possible to support languages where the structure of the sentence is not *subject-verb-object*.

**Remapping.** When the mapping between two slices is non-trivial, there is still the chance to reuse (part of) the code without changes, by using the **mapping** feature. In this case the **module** statement is qualified with the optional **mapping** clause and a *mapping* between the nonterminals of the *reference syntax* and the nonterminals of the *concrete syntax* is given; the mapping is between the ordinal numbers that correspond to the nonterminals, following the same scheme that has been described for modules (Listing 4.9). For instance, although the compiled code for a do-while loop slightly differs from the generated code for a while loop because the LoopBody shall be evaluated at least once, type-checking can be reused verbatim, by remapping nonterminal 1 of WhileLoop onto nonterminal 2 of DoWhileLoop, and nonterminal 2 of WhileLoop onto nonterminal 1 of DoWhileLoop. The mapping is *local* to the role, and does not ‘stick’ between roles, unless explicitly declared: this means that in role code-gen the order of the nodes for DoWhileLoop will be the one that has been originally declared in the concrete syntax. A new code-gen role must be still written, but the type-checking phase will be reused.

Although named —and effectively implemented, as we will see in Sect. 4.4— in a different way, this operation has in practice the same effect of *rewriting* the tree node DoWhileLoop(LoopBody, LoopCondition) to a node WhileLoop(LoopCondition, LoopBody). Nonetheless, the *rewrite* operation is available in Neverlang as well, in the

<sup>4</sup><http://onx.ms>

#### 4. Neverlang

```
// without remapping
module Expr {
  reference syntax {
    UnaryExpr ← PostfixExpr;
    CastExpr ← UnaryExpr;
    MulExpr ← CastExpr;
    AddExpr ← MulExpr;
    ShiftExpr ← AddExpr;
    RelExpr ← ShiftExpr;
    ...
  }
  role(evaluation) {
    0.{ $0.value = $1.value; }.
    2.{ $2.value = $3.value; }.
    4.{ $4.value = $5.value; }.
    6.{ $6.value = $7.value; }.
    8.{ $8.value = $9.value; }.
    10.{ $10.value = $11.value; }.
    ...
  }
}
slice ExprSlice {
  concrete syntax from Expr
  module Expr with role evaluation
}

// same code, with remapping
module Expr {
  reference syntax {
    UnaryExpr ← PostfixExpr;
    CastExpr ← UnaryExpr;
    MulExpr ← CastExpr;
    AddExpr ← MulExpr;
    ShiftExpr ← AddExpr;
    RelExpr ← ShiftExpr;
    ...
  }
  role(evaluation) {
    0.{ $0.value = $1.value; }.
  }
}
slice ExprSlice {
  concrete syntax from Expr
  module Expr with role evaluation
  mapping {
    2 ⇒ 0, 3 ⇒ 1, 4 ⇒ 0, 5 ⇒ 1, ...
  }
}
```

**Listing 4.10:** Usage of the remapping feature to reuse code within the same module. On the left, the full, explicit version; on the right, the repetition has been replaced by remapping the same semantic action.

form of an experimental semantic action DSL; we will return on this later in Sect. 4.2.

The remapping feature is also useful to repeat an action over several productions, even within the same slice; for instance, consider the chain of expressions for C-like languages  $\text{UnaryExpr} \leftarrow \text{PostfixExpr}; \text{CastExpr} \leftarrow \text{UnaryExpr}; \dots$  (Listing 4.10): where many actions involve passing over values throughout the chain. Instead of rewriting the same semantic action assigning attributes along the chain over and over (on the left of Listing 4.10), you can use the mapping construct to do it for you (on the right). In general, consider some module  $M$  with **reference syntax**:

$$A \leftarrow B; B \leftarrow C; C \leftarrow D;$$

And suppose you want to pass on the attribute value from  $D$  to  $C$ ; then, in some role  $r$  of  $M$  you may write:

$$\text{role}(r) \{ 0 \text{ .}\{ \$0.\text{value} = \$1.\text{value}; \} \text{ .} \}$$

And, then, assuming  $A$  maps to  $0$ ,  $B$  maps to  $1$ , the second  $B$  maps to  $2$ , etc. the slice would read:

```
module M with role r mapping { 2 ⇒ 0, 3 ⇒ 1, 4 ⇒ 0, 5 ⇒ 1 }
```

which would mean to apply on rule  $B \leftarrow C$  action 0, with B as A, and C as B; similarly, on rule  $C \leftarrow D$  the action 0, will be executed with C instead of A and D in place of C.

On the one hand, the usage of this feature may hamper the reusability of a component, because it would depend on the way the syntactic module was originally written. Any change to that grammar production would cause any module that depend on the other to break. On the other hand, this may be true during the first phases of the development, when iterations on a definition may be frequent. But, in the case of a *stable* module, this should not occur often. In fact, it is good programming habit that a radical change in a code unit should correspond to releasing a *new version* of the code unit, so that backwards compatibility can be preserved. This is especially true for Neverlang, since components can be released even in their binary form. During the development phase, refactoring tools could limit the impact of this problem on user code.

**Endemic Slices.** In Chapter 3 the concept of *globally-scoped components* was introduced. Neverlang implements such components through **endemic slice**. The **declare** block of an endemic slice defines ancillary fields and methods that should be globally accessible from the code of any semantic action. Endemic slices are used to implement features in a language that do not have a direct syntactic counterpart. A typical example of this is the *symbol table* (see Listing 4.11 for an example). Although every compiler might manage its symbol table in its own particular way, this is a construct that is generally always present in some form. The information that we store in a symbol table must be consistent and accessible from all the components of the compiler. Therefore, in Neverlang, this component should be accessible from all slices that are used in the language, even if there is no syntactic construct inside the language to refer to it. An *endemic slice* declares the interface and the constructor of the implementation of a globally-accessible object that implements this concern. An endemic slice only declares an interface and a constructor, so that the programmer is free to use his favorite programming language and tools to implement the globally-accessible object. The endemic slice *imports* the implementation *inside* Neverlang, so that it is available to every component that may require it. The endemic slice can be substituted at will, by any compatible object that implements the same interface (for a use case, see Linda-Python [14], where a threaded execution model is substituted with a distributed, RMI-based execution model). Objects declared in an endemic slice are destroyed and recreated at each execution of the interpreter, that is, for each new input program, but its state, if any, “sticks” between evaluation phases. It is also possible to make an endemic slice “stick” across evaluations using the `static` modifier, before the name of the object in the **declare** section; in this case Neverlang will instantiate the object only once during the execution of the interpreter, so that the state may be preserved across the evaluation of different input programs. This may be useful in the creation of interactive interpreters using the `nlg` tool (Sect. 4.4).

#### 4. Neverlang

```
// javalang/SymbolTable.n1
endemic slice javalang.SymbolTable {
  declare {
    // invoke the empty constructor, put it in the $$SymbolTable object
    SymbolTable: javalang.utils.SimpleSymbolTable ;
    // alt. syntax: the block may contain an arbitrary Java expression
    // SymbolTable: .{ SymbolTableFactory.create() }.
  }
}
// SymbolTable.java
package clang.utils;
public interface SymbolTable {
  Object getValue(String name);
  Object getType(String type);
  void put(String name, String type, Object value);
  ...
}
// SimpleSymbolTable.java
package javalang.utils.
public class SimpleSymbolTable implements SymbolTable { ...}
// VarLookup.n1
module javalang.VarLookup {
  import { javalang.utils.*;}
  reference syntax {
    VarLookup ← Identifier;
  }
  role(type-checking) {
    0 @{
      String ident = $0.identifier;
      $0.type = $$SymbolTable.getType(ident);
    }.
  }
}
}
```

**Listing 4.11:** Endemic slice providing a `SymbolTable` interface with an implementation.

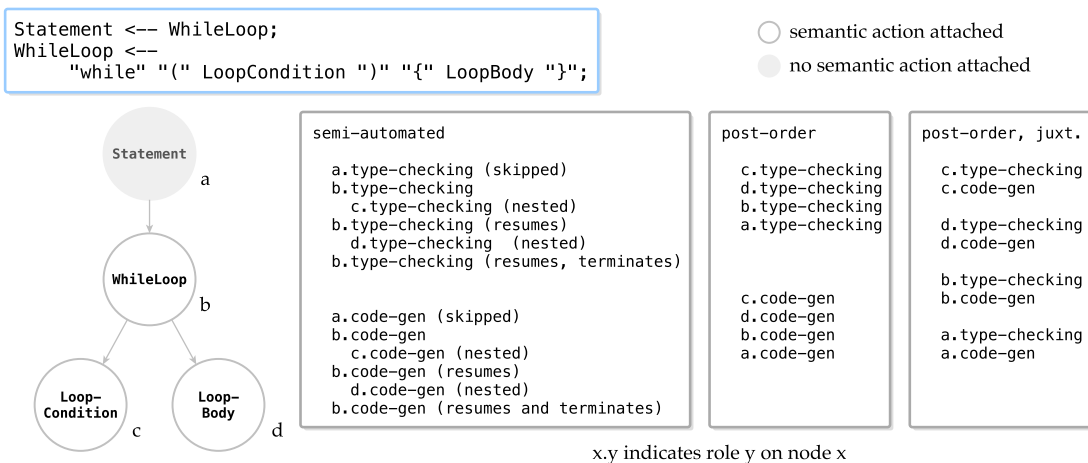
```

language javalang.Lang {
  slices
  ...
  javalang.WhileLoopSlice
  javalang.DoWhileLoopSlice
  javalang.Expression
  javalang.StatementList

  endemic slices
  ...
  javalang.SymbolTable

  roles syntax <+ type-checking <+ code-gen
  rename {
    ...
    LoopCondition → Expression;
    LoopBody → StatementList;
  }
}

```

Listing 4.12: Extract of the **Language** descriptor for a Java compiler.

**Figure 4.1.** Role execution order: the grayed node does not have semantic actions attached. For the sake of the example, we assume that all roles are evaluated in the same way (semi-automated or post-order).

### 4.1.3. Combining Slices Together: Generating a Language

Neverlang's **Language** descriptor lists all the *slices* that form the complete language implementation, including *endemic slices*. It also defines the sequence in which *roles* will be evaluated (Listing 4.12).

**Role Execution Order.** The order of execution is specified by the **roles** clause. The first role is always *syntax*, indicating that the first phase is *parsing*, followed by the sequence in which every role in the slices should be processed, separated by a delimiter.

#### 4. Neverlang

Figure 4.1 shows an example with the while loop. In the picture we are using the notation `node.role-name` to indicate the order of execution.

There are three kinds of pre-defined visiting strategies (all depth-first):

**Semi-automated** indicated by “<+”; the visitor automatically descends from the root into the children until a semantic action is attached; then the control is left to the action, which might or might not opt-in to use the **eval** statement (or one of the shorthands described in the previous paragraphs) to proceed with the visit. The **eval** statement can be used to perform arbitrary visiting strategies, where nodes may be even re-evaluated more than once. When **eval** is used, the execution of the child nodes is *nested*; that is, once the execution of the action for the child nodes has terminated, control is given back to the parent, which may eventually use **eval** again.

**Post-order** indicated with “<”, this strategy visits the tree depth-first, and executes the actions *after* the children have been evaluated (left-to-right). It is well-suited for L-attributed and S-attributed grammars (Chapter 2).

**Juxtaposition** indicated by “:”, when two roles are juxtaposed, the execution of roles is *interleaved*; that is, instead of executing one role per tree visit, all the roles that are juxtaposed will be executed “at once”: in other words, for each node all the actions of all the juxtaposed roles will be executed in sequence, as opposed to simple *semi-automated* and *post-order*, where each role corresponds to *one visit* of the tree. When two (or more) roles are juxtaposed, the execution strategy is the one indicated by the first left-hand non-juxtaposing role; e.g., with the **roles** clause:

```
roles syntax < ... < foo : bar < ...
```

then the execution is post-order, and bar is juxtaposed to foo; with the **roles** clause:

```
roles syntax < ... <+ foo : bar < ...
```

then the execution is semi-automated, and bar is juxtaposed to foo. Juxtaposition in combination with semi-automated at the time of writing is experimental.

For a use case of juxtaposition, see for instance the *Log Language* in [15, 14, 100], a language for log rotating, similar to the UNIX logrotate utility. In this language, each line is a log management operation (e.g, rename, backup, etc.). The utility, besides the execution phase where the log management operations are performed on the file system, includes two more evaluation phases, logging and permissions. The logging phase produces itself a log of the operations that are being executed, the permissions phase checks the file permissions of the files that are being modified. The default is a *post-order* execution, which causes each phase to be executed in sequence: first it logs all the operations that are going to be executed, then, for all the commands the permissions are verified, then all the commands are executed at once; by switching to the *interleaved* execution strategy, for each command in the input file the operation is first logged, then permissions are evaluated, and finally the operation is executed.



**Dependencies Between Slices.** As we saw in Chapter 3, each language component has *dependencies*. These dependencies should be satisfied when the components are combined together (Chapter 3). In particular, we saw that the *syntax definitions* provide and require other syntactic definitions, by way of *nonterminals* (syntactic dependencies), and *attribute definitions* provide and require other attribute definitions (semantic dependencies). In order for a language implementation to be consistent, both syntactic and semantic dependencies shall be satisfied. In a *slice*, syntactic dependencies are implied by the *concrete syntax*, while semantic dependencies derive from the roles. For instance, in the case of the while loop (Listings 4.2 and 4.4):

- the concrete syntax *provides* the `WhileLoop` nonterminal, and it *requires* at least one definition for the `LoopBody` and `LoopCondition` nonterminals. Because of the semantics of grammars, each nonterminal may admit more than one definition, but at least one is required.
- the type-checking role *requires* the `Class<?>` attribute type to be defined on nonterminal `$1`, which, in this case, resolves to `LoopCondition`
- the code-gen role *requires* the `String` attribute code to be defined on nonterminals `LoopCondition` and `LoopBody`, respectively, and *provides* a `String` attribute code on nonterminal `$0`, which in this case resolves to `WhileLoop`.

Chapter 3 also stated that these dependencies must be *satisfied* in a language implementation. Thus, the framework must enforce the resolution of such dependencies at composition time. In Neverlang, when one such dependency is left unsatisfied, the runtime throws an error. In [99, 98] we explored ways to track and resolve dependencies automatically, and present them to end users in a convenient way (see also Sect. 6.4 for further details): the objective is to enable end users to compose a working language implementation, where all the dependencies are satisfied, for any given set of slices, without writing code.

The *starting symbol* or *axiom* of the language, in Neverlang is always called `Program` by convention. In order to produce a *meaningful* language (that is, a non-empty language, see Chapter 2) at least one slice should *provide* the `Program` nonterminal. This can be done *explicitly*, by introducing a production of the form “`Program ← . . .`”, or implicitly, by using the **rename** feature, that has also a number of other uses.

**Rename.** A **language** descriptor may optionally include a **rename** section. This section declares a list of nonterminals that should be consistently renamed to other nonterminals. For instance, in our example, we always used the `LoopCondition` nonterminal to represent the condition of the while and do-while loops. This condition is usually represented by an `Expression`. Now, let us suppose that a slice `javaLang.Expression` is available and that it *provides* the nonterminal `Expression`. We may introduce a slice with the production `LoopCondition ← Expression`, but this slice would serve no meaningful purposes beside satisfying the requirements of the slice `javaLang.WhileSlice`. There is a better mechanism to achieve the same result, which is providing a **rename** mapping. In Listing 4.12, the `LoopCondition` is renamed to `Expression` in every production in which it occurs, causing, for instance, the production

```
While ← "while" "(" LoopCondition ")" "{" LoopBody "}"
```

#### 4. Neverlang

```
language javalang.alt.Lang {
  slices
  ...
  bundle ( javalang.bundles.Loops )
  ...
}
bundle javalang.bundles.Loops {
  slices
  ...
  javalang.WhileLoopSlice
  javalang.DoWhileLoopSlice
  ...
}
```

Listing 4.13: *javalang.Lang* using bundles.

in `javalang.WhileSlice` to become

```
While ← "while" "(" Expression ")" "{" LoopBody "}".
```

The same can be done with `LoopBody`, that could be renamed into `StatementList`, assuming that there exist a slice that *provides* such nonterminal. The **rename** feature can be also used to declare the starting symbol of the language, by renaming it to `Program`.

**Bundles.** A bundle is a collection of slices that together implement a sort of *macro-feature*. For instance, one might bundle together all the slices that implement the looping constructs for a language, all the slices that implement the conditional branches, etc. The role of the **bundle** construct is only one of convenience. A bundle is automatically expanded into the collection of slices that it contains; thus using a bundle in a language is completely equivalent to spell out its contents in the language descriptor. For instance (Listing 4.13), consider a bundle `javalang.bundles.Loops` containing the slices `javalang.WhileLoopSlice` and `javalang.DoWhileLoopSlice`, and suppose this bundle is added to a language `javalang.Lang`; this language is identical to the language that included directly the slices `javalang.WhileLoopSlice` and `javalang.DoWhileLoopSlice`.

## 4.2. Runtime Deployment of Semantic Actions and Tree Rewriting DSL

It is worth spending a few words of the runtime capabilities of the Neverlang system. The mapping between tree nodes and slices is maintained in a dynamic data structure, that is filled and looked up at runtime. This, for instance, makes it possible to *deploy new concerns of the language*, not only without having to rebuild the interpreter, but also with no need for *shutting it down* in the first place. For instance, the *Log Language* [15, 14, 100] is a use case for juxtaposition (Sect. 4.1.3). The logging the behavior of the utility is a classic example of a cross-cutting concern that users may want to deploy and undeploy at runtime [4]. The dynamic loading capabilities of Neverlang make it possible to

deploy such aspects of a language *natively*, at runtime, thereby allowing features to be enabled and disabled at will during program execution.

The same feature can be useful to *specialize* the semantics of the role that is being executed with *optimized* versions of the same semantic rule. The feature is currently being exploited in our JavaScript implementation (Sect. 6.2) to realize a feature-oriented rendition of the Truffle [111, 50] runtime system for Java-based AST interpreters. In this case, though, multiple semantic actions may be executed depending on *guards*. Each guard is an assertion that may trigger execution of a different action on a particular node, depending on a runtime-evaluated expression. For instance, the result of an integer expression may use ints internally up until an overflow exception is raised [111]; in this case a guard would activate and cause a different rule to be executed (for instance, a standard-compliant ECMAScript implementation would promote the Number value from an int to a double). The goal in Neverlang is to make it possible to implement such rules in different modules, so that programmers may develop different optimization strategies and choose which to include in their own language implementation. The Truffle system, among other things, rewrites the tree with specialized version of the nodes. The same feature is being developed for Neverlang. An API will be made available to rewrite nodes of the syntax tree, and a DSL will be integrated using Neverlang's native language plugin system, to simplify its use.

All of these features are currently being implemented and further investigated; we reserve to describe them in more depth in a future work.

### 4.3. Tools and Utilities

At the beginning of this section we recalled that the *Neverlang language* is only one of the ways developers can exploit the Neverlang APIs. The Neverlang compiler `nlgc` translates the Neverlang language into JVM-compatible source code, so that this API is exposed to a more concise interface. Other tools are also bundled with the Neverlang framework: the simple Neverlang launcher `nlg`, and the interactive read-eval loop `nlgi`. A small library of utility functions and classes is also provided. The reason this library is small, is that users are free to use standard libraries from the Java ecosystem. Table 4.1 shows a list of methods of this library. The `GenericMain` is used to implement custom launchers (instead of the default `nlg`), to collect attributes from subtrees into a `List<T>` (`AttributeList`) and to dump an AST to a Graphviz [28] picture.

**The Neverlang Compiler `nlgc`** In Sect. 4.4 we will show that it is easy to map the Neverlang language onto the Neverlang API, but the framework comes with a compiler called `nlgc` which automates the process. In most cases, `nlgc` generates *Java source files*. For instance, assuming that the source code presented in Chapter 4 were in a directory called `nlg-src`, a user would type:

```
$ nlgc -s src nlg-src/*.nl && javac -d bin src/**/*.java
```

#### 4. Neverlang

Launcher
<b>GenericMain</b>
GenericMain() <i>Implements a generic custom launcher (to implement a custom nlg command.)</i>
public void parseArgs() <i>Parse CLI arguments</i>
public void readFiles() throws IOException <i>Read files to the internal buffer.</i>
abstract List<String> processFiles() <i>Process files through language implementation</i>
Neverlang Runtime Library
<b>AttributeList</b>
static <T> java.util.List<T> collectFrom(ASTNode t, String attributeName) <i>Collects a typed List of T-valued attributes, implementing the bucket brigade operator [52]. Usage: List&lt;String&gt; attributes = AttributeList.collectFrom(\$n, "attributeName"). For an example see Chapter 5.</i>
<b>FileUtils</b>
String fileToString(String path) throws IOException <i>Loads a file into a String.</i>
void stringToFile(String contents, String path) throws IOException <i>writes a String to file</i>
<b>GraphvizAST</b>
GraphvizAST(ASTNode root) <i>Generates a Graphviz [28] representation of the given tree.</i>
String toString() <i>Returns a String representation of the tree in dot format.</i>

**Table 4.1.:** A summary of support library functions in Neverlang.

The command `nlgc` generates all the source files in the `src` directory, then `javac` compiles the source code into class files in the `bin` directory. For instance, in Sect. 4.1, we described how to write a while loop in Neverlang, and we have shown how to write a module (Listing 4.2, p.25), a slice (Listing 4.8, p.30) and a language descriptor (Listing 4.12, p.30). By invoking `nlgc` over these files the result would be a collection of Java source files. An example of what the generated source code looks like can be seen in the next section, in Listing 4.15 (p.47).

One core goal for the Neverlang runtime was to have as few dependencies as possible. Thus, the Neverlang runtime has been written in pure Java 6, and the *default language* for semantic actions of the Neverlang language is Java as well. In Sect. 4.4 we will see that the Neverlang API is a JVM API, therefore, it is possible to use it from *any* programming language of the JVM, and, consequently, even semantic actions can be written using the JVM language of choice. The Neverlang language allows semantic actions to be written using a custom language as well, while still providing useful

```

public class JavaTranslatorPlugin extends TranslatorPlugin {
    public JavaTranslatorPlugin() {
        language = "java";
        fileExtension = "java";
        fileTemplate = "public class {0} implements SemanticAction '{'\n"+
            "    public void apply(Context $ctx) '{'\n{1}\n    }'\n'";
        attributeRead = "$ctx.node({1}).getValue(\"{1}\")";
        attributeWrite = "$ctx.node({1}).setValue(\"{2}\", {3});";
        ...
    }
}

```

Listing 4.14: Translator Plugin for Java.

syntactic sugar to drive the visit and access the attributes (the `eval` keyword, the dot-notation for accessing attributes, etc.). This is provided by way of *translator plugins*. When multiple programming languages are used, `nlgc` generates source files for each given target language. Each file can be then compiled using the native platform tools. For instance, `scala` source files would be compiled using the `scalac` compiler.

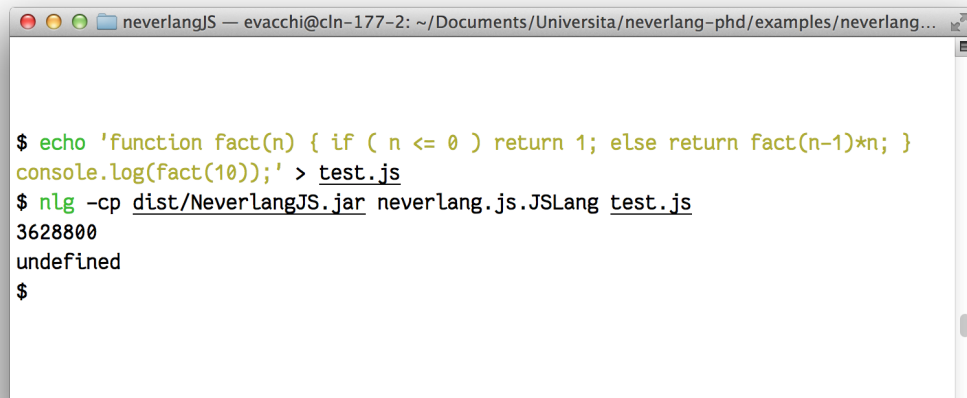
A translator plugin hooks into the code-generation process of the Neverlang compiler to analyze and manipulate the input source of a semantic action, and desugar the Neverlang language shorthands into the regular API calls that we presented in this section. The developer can then hint at the system that semantic actions are being written in a different language. Listing 4.6 (p.28) showed how to switch between language plugins in a module written using the *Neverlang language*.

Translator plugins in their *simplest form* describe how the occurrences of each shorthand should be rewritten into API calls, and in their *extended form*, they may parse and verify (e.g. type check) the *entire* code block. Listing 4.14 shows how the *simple* Java translator plugin is defined using Java itself to implement it. Code for the Scala plugin is similar. The plugin itself, again, can be written using any JVM-supported language.

Currently we have implemented *simple* support for Java and Scala, plus the `template` plugin, which is suitable for code generation. These plugins *do not* actually parse the code block, which is rather reproduced *verbatim*; desugaring occurs through simple source code transformations (pattern matching). Although this is a very simplistic approach, it is also very convenient, because translator plugins never become outdated: new language releases can be supported from the day one. Compare this to the alternative solution of fully-parsing the entire block of code, and how, for instance, it made instantly obsolete all the tools that were written in the pre-generics age of Java. With this approach, the Neverlang language supports all the most recent Java features, including Java 7's lambdas. Besides, type checking and other verifications will be executed at the time the generated source files will be compiled using the target language compiler (e.g., for java `javac`).

Nevertheless, *extended* plugins can be implemented as well; this is especially preferable when the code block hosts a *custom DSL*. In this case, the code block can be passed into

#### 4. *Neverlang*



```
neverlangJS — evacchi@cln-177-2: ~/Documents/Universita/neverlang-phd/examples/neverlang...
$ echo 'function fact(n) { if ( n <= 0 ) return 1; else return fact(n-1)*n; }
console.log(fact(10));' > test.js
$ nlg -cp dist/NeverLangJS.jar neverlang.js.JSLang test.js
3628800
undefined
$
```

**Figure 4.2.:** *nlg* executing a JavaScript program with *neverlang.js* (Section 6).

a separate *Neverlang* instance, which will parse and possibly generate the source code in a target language. For instance, the experimental tree rewriting DSL (Sect. 4.2) is implemented as an extended plugin.

**The *Neverlang* Launcher *nlg*** In the next section, we will see that in *Neverlang*, a language implementation is a subclass of the *Language* class of the *Neverlang* API. In order to use the language implementation you should instantiate this class (*e.g.*, see Listing 4.16, p.50). Because this is routine, *Neverlang* comes bundled with a convenient predefined launcher called *nlg*. The *nlg* tool expects the canonical name of the language class to be given on the command line, and at least one input source file name (if no file name is given, the launcher will enter the interactive mode, that we will describe in the next paragraph). The launcher will automatically instantiate the given language and invoke the *Language.eval(String)* API (Sect. 4.4) on each of the given files. This is particularly useful during the first phases of the development of a language, or to quickly test if the language loads as expected. It is still recommended to roll your own custom launcher, by subclassing the bundled *GenericMain* class. For instance, we implemented a JavaScript interpreter that we dubbed *neverlang.js* (Sect. 6.2). Figure 4.2 shows the JavaScript interpreter invoked through *nlg*, with

```
$ nlg neverlang.js.JSLang <input-program>
```

with an input program that computes the factorial 10!. We will see more on *neverlang.js* in Chapter 6.

**The *Neverlang* Interactive Read-Eval Loop *nlgi*** *Neverlang* comes bundled with an *read-eval loop* utility to interactively input programs at a prompt. The utility can

```

neverlangJS — nLgi -cp dist/NeverLangJS.jar neverlang.js.JSLang — nLgi — rlwrap — 80x24
$ nLgi -cp dist/NeverLangJS.jar neverlang.js.JSLang
NLGi. Neverlang Interactive REPL.
Using Neverlang Compiler 0.7.15
Language neverlang.js.JSLang

Available Commands:
:help      :h  Get this screen
:quit      :q  Leave Repl
:reload    :r  Reload the language implementation from disk
:tree      :t  Dump last parse tree as a Graphviz source file
:parser    :p  Dump parser to disk
:endemic   :e  Dump Endemic Slices
:grammar   :g  Dump language grammar
:multiline :m  Toggle multiline input
> function fact(n) { if ( n <= 0 ) return 1; else return fact(n-1)*n; }
[Function: fact]

> fact(10);
3628800

>

```

**Figure 4.3:** *nLgi* executing a JavaScript program with *neverlang.js* (Chapter 6).

be started by invoking it at command line with the language name as a parameter. The *nLgi* tool provides commands to interact with the language; it is possible to dump the contents of the endemic slices, show the complete grammar, print the attributes of the tree, and dump the AST and the goto-graph of the parser to the screen using Graphviz [28]. Figure 4.3 shows *neverlang.js* launched as an interactive console, evaluating an interactively-defined factorial function, as you can see, the input source code is also automatically colorized, depending on the grammar of the language.

## 4.4. Implementation

The Neverlang framework runs on the Java Virtual Machine. Core data structures, support and utility classes are written in Java, bearing as few dependencies as possible. In fact, Neverlang depends on no other library or technology besides pure JDK 1.6, which makes it even compile and run on Android’s Dalvik VM (see also [18]). It is also fully forward-compatible with Java 7 and Java 8, and, for what concerns semantics spec-

## 4. *Neverlang*

ification, it is virtually compatible with any JVM programming language. Section 4.1 presented how the *Neverlang* framework represents the concepts that we described in Chapter 3 using the syntax of the *Neverlang language*, a convenient DSL that is used to describe the *Neverlang* components, and that can be automatically translated into JVM-source files using the compiler tool `nlgc` (Sect. 4.3). Nevertheless, the framework is designed so that its APIs are easy to use. Thus, the *Neverlang* language is only one of the possible *front-ends* to the *Neverlang* core. The *Neverlang* APIs can be used *directly*, exploiting the multi-language features of the JVM platform. Therefore, it is possible to interface with *Neverlang* using any programming language that runs on the JVM, such as Java, Scala, Groovy, JavaScript, JRuby, Jython only to name a few. This makes it possible to easily extend the ways programmers may interact with the foundations of *Neverlang*. For instance, even though the main, suggested way to write *Neverlang* components is still the *Neverlang* language, developers may write slices and modules *directly*, using Java or their favorite JVM programming language. In fact, the only assumption in the *Neverlang* framework is that language components have been compiled down to JVM class files that implement particular interfaces. As long as this contract is respected, the *Neverlang* framework will happily load the language components, regardless of the way they were generated.

This kind of flexibility makes it possible to build interesting interfaces on top. Future directions may involve interactive development environments that hook directly into *Neverlang*'s core. A first example of these capabilities is the *Neverlang* read-eval-loop `nlgc` (Sect. 4.3), which gives the language developer an interactive console for debugging and playing with language implementations. Further developments involve the implementation of a GUI for automatic variability management (see Sect. 6.4) and a full assisted development environment.

For simplicity, the code examples in this section will always use Java, as it is the *lingua franca* of the JVM, and because it is the language the *Neverlang* API is written in. Nevertheless, there is no limitation on the choice of languages that may use the *Neverlang* API, as long as a suitable JVM implementation is available. A summary of the complete API is given in Table 4.2 for reference. Details on how this API is used will be given in the next section.

### 4.4.1. Architecture

The *Neverlang* framework API is conceptually very simple; modules, slices and the language descriptor are mapped onto *regular Java classes*. Therefore, they can be loaded by a Java `ClassLoader` through their *canonical class name*. The canonical class name reflects the dotted identifier that is conventionally used in given module, slice, bundle and language declarations in the *Neverlang* language. For instance, the declaration

```
slice com.example.MySlice { ... }
```

would generate a class `MySlice` in package `com.example`. Class loading is internally used by all the APIs (Table 4.2) that load components *by name*, such as `importSlice()`.



Core
Language
<code>importSlice(String sliceName)</code> <i>Imports the slice with the given canonical class name</i>
<code>importSlices(String... sliceNames)</code> <i>Imports the given list of slices</i>
<code>importEndemicSlice(String sliceName)</code> <i>Imports an endemic slice with the given canonical class name</i>
<code>importEndemicSlices(String... sliceNames)</code> <i>Imports the given list of endemic slices</i>
<code>declare(Role rs, Role... rs)</code> <i>Declares the given role definitions</i>
interface SemanticAction
<code>apply(Context \$ctx)</code> <i>Executes the semantic action on the given Context</i>
interface Component (implemented by Module and Slice)
<code>SemanticAction getAction(String role, int pos)</code> <i>Returns the semantic action at the given position. Slices delegate to the imported modules.</i>
<code>Syntax getSyntaxDef()</code> <i>Returns the syntax definition associated to this component</i>
Syntax
<code>TaggedNonterminal[] getProvides()</code> <i>Returns an array of the provided nonterminal, along with their (optional) tags (Sect. 6.4)</i>
<code>TaggedNonterminal[] getRequires()</code> <i>Returns an array of the required nonterminal, along with their (optional) tags (Sect. 6.4)</i>
Module
<code>protected declareSyntax()</code> <i>Declares that the modules define a syntax section</i>
<code>protected syntaxFrom(String fromModule)</code> <i>Declares from which module the reference syntax is imported from</i>
<code>protected declareRole(String roleName, int... ids)</code> <i>Declares a role and the number identifiers (nonterminals) for which semantic actions have been defined</i>
Slice
<code>protected importSyntax(String fromModule)</code> <i>Declares from which module the concrete syntax is imported from</i>
<code>protected importRoles(String fromModule, String role, int... ids)</code> <i>Declares from which module the role role is imported, and which actions are being imported</i>

Table 4.2.: A summary of the Neverlang Runtime API.

#### 4. Neverlang

API clients extend these classes and initialize their fields in their constructors using the provided methods. The runtime system, using the public API of these classes, loads into memory all the required components, instantiating the language implementation. Thus, a language implementation in Neverlang is not an opaque executable, but a collection of components that JVM languages can interact with, by querying a rich API. This API does not only drive the execution of the language processor, but also it may be employed to retrieve information on the loaded components, making it even possible to substitute and unload components *at runtime*.

The presentation of the API that follows represents the *default* implementation of the Neverlang framework. However, nothing prevents users from defining their own alternative implementations of the given interfaces, choosing different strategies for class loading and decomposition: as we will see, modules are decomposed in such a way that different language targets are possible for semantic actions. Implementations are free to bring this further: for instance, it may be possible to adopt the scripting API of the JVM, and define components with JavaScript (using the *Rhino* or *Nashorn* interpreter), Groovy, JRuby, Jython, Clojure, etc.; in this case not only the components could be *loaded* at runtime, but they could be even *defined dynamically* at runtime.

**Language** A *language descriptor* is a class that extends the `Language` class. A `Language` subclass must declare in its constructor (using the `importSlice()` method) which slices it imports, the order in which roles are executed, and the renames. The `importSlice()` directive expects the canonical name of an implementor of the `Component` interface to be given; both `Slice` and `Module` implement this interface, thus a language may import both slices and modules; if a module name is given, then the module also represents the slice that declares the syntax and the roles that it contains. For instance, if a module `com.example.MyModule` declares a **reference syntax** and some role (e.g., type-checking), then it also represents a slice with the same name that contains its *reference* syntax as a *concrete* syntax, and the corresponding implementation for role type-checking. This is a convenience that is generally used during the first stages of the development; as new roles will require to be introduced, slices may be a better fit (e.g., see [100]). Incidentally, the `Language` class inherits from `Bundle`, since `Language` is a slice container as well (it follows that languages can be used as bundles).

**Slices** *Slices* are subclasses of `Slice` and implement the `Component` interface; in their constructors they declare the modules from which they import their syntax and semantic roles, using the `importSyntax(String moduleName)` and `importRole(String roleName)` API. *Endemic slices*, do not extend the `Slice` class because they behave in a different way: they do not import roles or syntax from modules, but rather they declare a singleton object. Nonetheless, they extend the `EndemicSlice` class and invoke a different API to instantiate the globally accessible resource that they implement.

**Modules** A module is a complex component made of several classes: one class inherits from the `Module` class, and it declares whether it is referencing a syntax definition from

```

package javalang;
public class WhileLoop extends Module {
    public WhileLoop() {
        declareSyntax();
        declareRole("type_check", 0);
    }
    public class WhileSlice extends Slice {
        public WhileSlice() {
            importSyntax("clang.WhileLoop");
            importRoles("clang.WhileLoop", "type_checking", 0);
        }
    }
    public class WhileLoop$role$syntax extends Syntax {
        public WhileLoop$role$syntax() {
            declareProductions(
                p(nt("WhileLoop"), "while", "(", nt("LoopCondition"), ")", nt("LoopBody"))
            );
        }
    }
    public class WhileLoop$role$type_check$0 implements SemanticAction {
        public void apply(Context $ctx) {
            $ctx.eval($ctx.node(1));
            if ( $ctx.node(1).getValue("type") != Boolean.class )
                throw new Error("The type of LoopCondition should be a boolean value");
        }
    }
    public class Lang extends Language {
        public Lang() {
            importSlices(
                ...
                "javalang.WhileLoopSlice",
                ...
            );
            importEndemicSlices(
                "javalang.SymbolTable"
            );
            declare( // syntax is implied
                role(PREORDER, "type_checking"),
                role(PREORDER, "code_gen")
            );
        }
    }
}

```

**Listing 4.15:** *Components in Chapter 4 as represented using Neverlang's APIs.*

#### 4. Neverlang

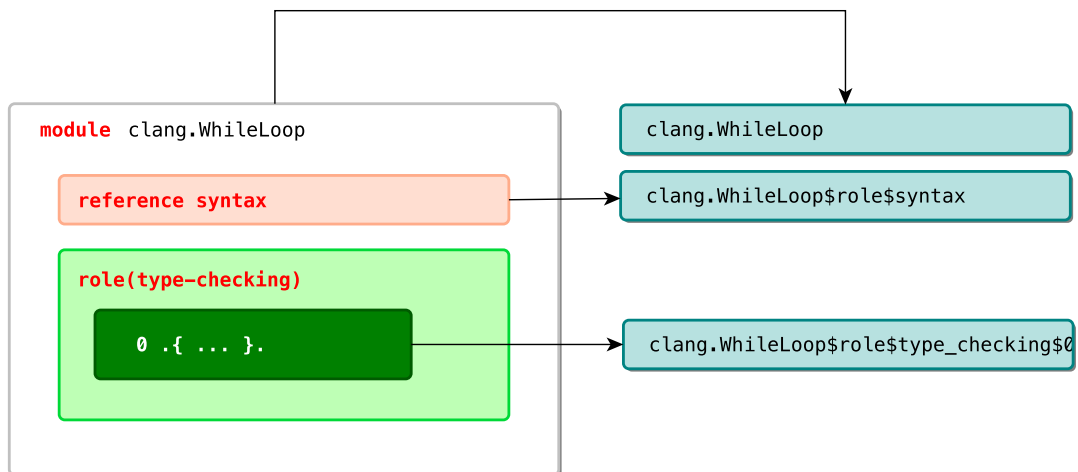


Figure 4.4.: How a module is broken down into several classes.

a different module, or if the syntax is being defined within the module itself; then it declares which roles are being defined, and which nonterminals will be hooked into, using the numbering scheme described in Chapter 4. Then:

- if the module comes with a syntax definition, another class, extending the Syntax class should be implemented; by convention this class shall be named

`<module-name>$role$syntax`.

For instance, the syntax for module `javalang.WhileLoop` in Listing 4.2 would be named `javalang.WhileLoop$role$syntax`.

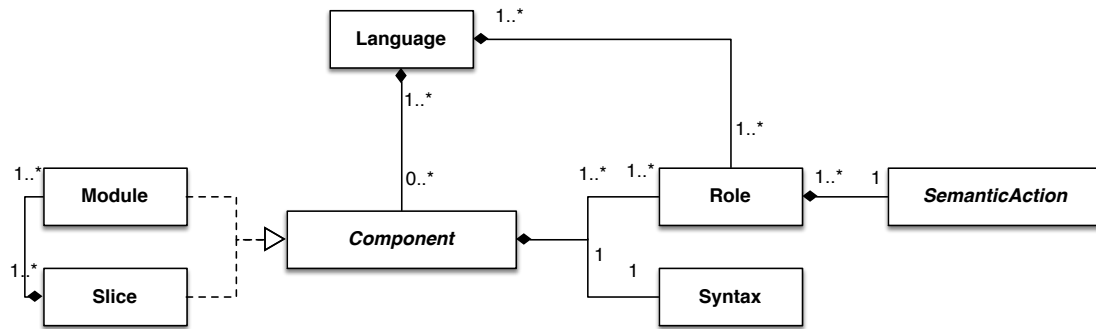
- for each role, and for each nonterminal being hooked into, a new class, extending the SemanticAction interface should be defined. By convention, such classes would be named by convention this class shall be named

`<module-name>$role$<role-name>$<N>`

where `<N>` is the number of the nonterminal that is being hooked and `<role-name>` is the role identifier; any “-” in the role identifier is replaced with “\_” to make it a valid Java identifier (e.g., `type-checking` becomes `type_checking`). For instance, rule `o` for the `type-checking` role of `javalang.WhileLoop` would be mapped onto

`javalang.WhileLoop$role$type_checking$0`.

The reason for this complex decomposition is to allow each semantic action to be written using a different language of the JVM. When the method `Module.getAction()` will be invoked, at runtime, the required action is loaded from disk and returned. The class `javalang.WhileLoop` must invoke in its constructors the APIs to declare all its sub-components (the class defining the syntax, and the classes defining the semantic actions for each role). Figure 4.4 shows a summary of all the classes that must be generated; Listing 4.15 shows a complete example of how the *Neverlang language* relates



**Figure 4.5.:** Relations between classes and interfaces in Neverlang (interface names are in italics).

to the Neverlang API. Figure 4.5 shows the relations between classes and interfaces.

Because each semantic action is compiled as a different class file, a different programming language can be used, provided that it can compile to a JVM class file. This fine-grained decomposition of the compiled modules makes it possible to achieve a finer-grained *compilation model*, that

1. reduces compile times
2. simplifies *separate compilations*
3. enables to ship, distribute and share language components as pre-compiled binaries.

In fact, a change in one module requires to recompile only *that* module from source (specifically, it would actually require recompilation only for *those sections* that have been modified). Compare this to conventional compiler generation techniques, that, being usually based on source generation, often require a large part (if not all) of the source code to be recompiled anew. This approach streamlines the compiler-generation process by making it possible to compile only those components that really *need* to be rebuilt. Of course, this possibility becomes particularly useful when the language implementation becomes large and complex (for instance, compare the experience we conducted with neverlang.js Sect. 6.2). Moreover, pre-compiled Neverlang components can be bundled together in jars to distribute them conveniently, and they can also be shared and imported by different languages independently.

Finally, please recall that users are free to write alternative `Module` implementations, with different loading strategies for semantic actions. For instance, a scripting language would make it possible to *define* custom behavior even at *runtime*.

#### 4.4.2. Runtime and Execution

The Neverlang runtime is made of two main parts: the DEXTER [17, 19] incremental parser generator and the component manager [18]. The component manager is responsible for loading languages, slices and modules, and for dispatching the correct semantic action to the node of the syntax tree that is being visited in the correct phase (described in a *role*).

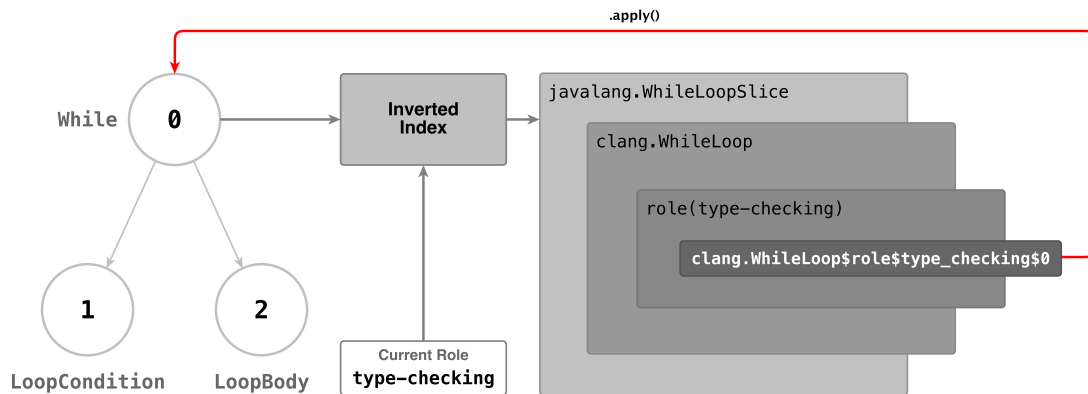
#### 4. Neverlang

```

import neverlang.runtime.*;
public class Launcher {
    public static void main(String[] args) {
        Language lang = new javalang.Lang();
        lang.eval(neverlang.utils.FileUtils.fileToString(args[0]));
    }
}

```

**Listing 4.16:** The easiest way to write a launcher for a neverlang language processor. The quickest way to launch it is using the `nlg` tool (Sect. 4.3).



**Figure 4.6.:** Method Dispatching.

Once all the components have been defined and compiled into class files using the regular platform tools (`javac`, `scalac`, etc.), and a Language subclass has been implemented, it is possible to execute the language processor, by invoking its `Language.eval(String)` method (Listing 4.16). This is when the *component manager* kicks in.

**The Component Manager** The *component manager* is Neverlang’s core. It implements the componentized *visitor* pattern and it loads and unloads the language components into memory. When the Language subclass is instantiated, the component manager loads the *slices* from disk, then it queries them for the *modules* they require. For each production in each syntax definition, an *inverted index* is populated to map each grammar production into the components that implement its semantics. For a given triplet  $(p, r, i)$ , where  $p$  is a production,  $r$  is a role, and  $i \in \mathbb{N}$  is an integer number, there is *at most one* semantic action that may be executed at a time. In particular, for a language  $L$  with a grammar  $G$ , consider the mapping

$$m : P \times \mathbf{R} \times \mathbb{N} \rightarrow SA_{\perp} \quad (4.1)$$

where  $P$  is the set of productions for a grammar  $G$ ,  $\mathbf{R} = \{R_0, R_1, \dots, R_n\}$  is the set of all the roles for language  $L$ , and  $\mathbb{N} = \{0, 1, \dots, n, \dots\}$  is the set of natural numbers, and

```
public interface SemanticAction { void apply(Context $ctx); }
```

Listing 4.17: Semantic Action interface.

$SA_{\perp}$  is the set of all the semantic actions, in all the roles of  $\mathbf{R}$ , plus the undefined action  $\perp$ . Let us also indicate with  $S_{r,i}$  the action hooked to the  $i$ -th nonterminal in role  $r$  of the slice  $S$ . Then, for all  $p \in P, r \in \mathbf{R}, i \in \mathbb{N}$  the mapping  $m$  is defined as:

$$m(p, r, i) = \begin{cases} S_{r,i} & \text{if } S_{r,i} \text{ exists} \\ \perp & \text{otherwise} \end{cases} \quad (4.2)$$

The reason why  $p$  is needed in the definition is that each slice contains (imports) a syntax definition, and the index  $i$  refers to a nonterminal in the syntax definition; then the triplet  $(p, r, i)$  maps to *at most* one slice, which is the slice that contains production  $p$ , role  $r$  and the semantic action hooked at index  $i$ . Currently, only one rule can be hooked at the same  $i$ . This is reasonable because, for each role, we want to execute only *one* semantic action per node<sup>5</sup>. Because the index is populated dynamically during the bootstrap phase, *it is possible to grow it and shrink it at runtime*, by removing, adding and updating its contents.

**Action Dispatching** When an input program must be processed, the `eval()` method in the Language class passes the input text to the parser. If the parsing process terminates unsuccessfully (the input program is syntactically incorrect), a `ParsingException` is raised. Otherwise, a syntax tree is generated. Each node of the tree is an instance of the `ASTNode` class. Each node is given a *tag*, which is the *grammar production* that it represents.

Once the parsing has terminated successfully, the component manager begins the *visiting* process, starting from the first semantic role. For instance, in the case of the example language `javalang.Lang` (Listing 4.12), the first role would be type-checking. For each node, the component manager reads the *tag* and it queries the inverted index  $m$ . Back to our example, Fig. 4.6 represents the tree for the production `WhileLoop` (Listing 4.2), during the execution of the type-checking role. Because nonterminal `WhileLoop` is number 0 in slice `javalang.WhileLoop`, then:

$$m(p, \text{type-checking}, 0) = \text{javalang.WhileLoop}\$role\$type\_checking\$0$$

where  $p$  is rule `WhileLoop`  $\leftarrow$  `"while"("LoopCondition ")""{"LoopBody "}`". If the value  $m(p, r, i) \neq \perp$ , then the semantic action  $S_{r,i}$  is executed.

**Action Execution** Each semantic action implements the single-method interface named `SemanticAction` (Listing 4.17); this is what Java 8 now calls a *functional interface*<sup>6</sup>

<sup>5</sup>the  $m$  function may be *allowed* to return a *set* of semantic actions: but then the system should be able to *choose* among them; we are using this idea to implement the *Truffle*-inspired dispatching mechanism (Sect. 6.2.1)

<sup>6</sup><http://docs.oracle.com/javase/tutorial/java/java00/lambdaexpressions.html>

#### 4. Neverlang

and it is often called a single-abstract method (or SAM) type. When the semantic action  $S_{r,i}$  must execute, its `apply(Context)` method is invoked. `Context` is a data class that contains a reference to the node that is being visited, a reference to the current `Language` instance, a reference to the role that is being executed. Executing  $S_{r,i}$  means invoking the `SemanticAction.apply(Context)` method of the corresponding class with a valid `Context` instance. In the case of Fig. 4.6, the `Context` instance would contain node `o`, role type-checking and a reference to the current `javalang.Lang` instance. The body of the action supposedly interacts with the `Context` instance:

- it drives the visit of the tree (and the consequent evaluation of its children or siblings), using the `eval(ASTNode)` to descend into a given child node, and also `evalAndReturn(ASTNode, String)` to descend into a child node, and return the value of one of its attributes. It also provides the `suspend()` and `resume()` methods to suspend and resume the role that is currently executing. Invoking these methods correspond, in the Neverlang language, to the syntactic sugar: `eval $N, $N:attribute, $suspend, $resume;`
- it accesses nodes by nonterminal id using `Context.node(int)`; e.g., `ctx.node(0)` returns the node for `WhileLoop`, `ctx.node(1)` returns the node for `LoopCondition`, `ctx.node(2)` returns the node for `LoopBody`, etc. In the Neverlang language these correspond to the short form `$N`;
- it reads and writes attributes using the idioms

```
T value = ctx.node(i).getValue("attrName")
```

where `T` is the expected type for `$i.attrName` and

```
ctx.node(i).setValue("attrName", value).
```

In the Neverlang language, they correspond to the attribute access `$N.attribute`.

- it provides access endemic slices: `ctx.singleton("EndemicName")` corresponds to `$$EndemicName`.

Listing 4.15 shows how the semantic action `o` for the type-checking phase in module `javalang.WhileLoop` (Listing 4.2) may be written in Java by only using Neverlang's native APIs.

**A Short Note on Signaling Termination** The visit of a language processor may require sometimes to be terminated abruptly, for instance to make an interpreter *return* from a procedure or to break out of a loop. In Chapter 4 we showed that Neverlang provides special keywords to support these situations. The `$terminate` command signals to the system that a visit should immediately terminate. These special control-flow operations require to unwind the call stack up to a given point (e.g., in the case of function call, up until to the point where the function has been invoked, and in the case of a loop, up to its root node). Neverlang implements such signals using a well-known technique that involves using “fast” JVM exceptions to signal termination, and `try/catch` statements to capture the signal: the technique consists in overriding the `fillInStackTrack()` method with a `NOP`, and preallocating such exceptions as singletons, which makes throwing



exceptions up to 400% times faster<sup>7</sup>, the technique is most notably used in the Scala compiler<sup>8</sup>, in the Ruby interpreter [87], in the Truffle [111] interpreter implementation system for Java, and in the Truffle-based interpreter for the R programming language FastR [53]. In Neverlang, JVM exceptions have been exploited to implement control-flow statements such as `break`, `return`; literature has also shown that JVM exceptions can be even useful to implement even more powerful control flow abstractions, such as forms of continuations [90].

For instance, consider Listing 4.18: suppose that we are implementing an interpreter for the language of our running example, and suppose we want to introduce support for the `break` statement inside the `while` loop. The interpreter may define an evaluation role, where the input program is executed. The module `BreakStatement` implements the syntax and semantics (role evaluation), for the `break` statement: that is, it throws a singleton instance of the class `CFlowSignal`, called `Break`. The semantics of the evaluation role of the `while` loop is to iterate up until the attribute `booleanCondition` evaluates to `false` (notice the usage of `Context.evalAndReturn()`); the `try/catch` blocks handle the signaling and break out of the loop or continue, depending on which signal (if any) is thrown during the evaluation of `LoopBody`. This technique (or a variation thereof) is used in the JavaScript interpreter (Sect. 6.2) to implement `break`, `continue`, to return from function calls and to handle exceptions.

#### 4.4.3. DEXTER

In order to support componentization and runtime composability, we also developed DEXTER: the Dynamically EXTensible Recognizer [17, 19]. In the bootstrap phase, productions are read from the `Syntax` subclasses, and they are fed to the DEXTER parser generator. DEXTER implements an in-memory LR parser generator. The generated parsers can be incrementally extended (*grown*) or restricted (*shrunk*) by adding and removing grammar productions on-the-fly. In fact, the **syntax** role of a module is a straight translation from the Neverlang DSL to a series of Java API calls to the DEXTER component (compare Listing 4.2 to Listing 4.15). The DEXTER parser generator implements an algorithm that bears some resemblance to those described in [48] and [46]. The algorithm *updates* the LR(o) DFA, which is the basis for many interesting parsers of the LR family, such as GLR and LALR. The DEXTER component includes an extensible regex-based lexer that allows to define lexemes at runtime. This subcomponent is called LEXTER. Lexemes are defined inline in a production both when they are constant keywords and when they are patterns. In the Neverlang language (Sect. 4.1), patterns are delimited by slashes, while keywords are delimited by quotes; in the Java API the distinction is made by using different classes. Further information on DEXTER and the underlying formal model that proves the correctness of the updated parsers can be found in [19]. An excerpt from this paper is reproduced in Appendix B.

<sup>7</sup>See [http://blogs.atlassian.com/2011/05/if\\_you\\_use\\_exceptions\\_for\\_path\\_control\\_dont\\_fill\\_in\\_the\\_stac/](http://blogs.atlassian.com/2011/05/if_you_use_exceptions_for_path_control_dont_fill_in_the_stac/) and <http://www.javaspecialists.eu/archive/Issue129.html>.

<sup>8</sup>see <http://www.scala-lang.org/files/archive/api/2.11.x/#scala.util.control.Breaks>, <http://www.tzavellas.com/techblog/2010/09/20/catching-throwable-in-scala/>

#### 4. Neverlang

```
public enum CFlowSignal extends neverlang.runtime.BaseCFlowSignal {
    public static final CFlowSignal Break    = new CFlowSignal();
    public static final CFlowSignal Continue = new CFlowSignal();
    private BreakSignal(){}
}
// semantic action for Break statement:
// BreakStatement <-- "break";
public class BreakStatement$role$syntax extends Syntax {
    public void apply(Context ctx) { throw CFlowSignal.Break; }
}
// semantic action for WhileLoop role evaluation:
public class WhileLoop$role$evaluation$0 implements SemanticAction {
    public void apply(Context ctx) {
        // eval LoopCondition
        // and return attribute LoopCondition.booleanCondition
        while(ctx.eval(ctx.evalAndReturn(1, "booleanCondition"))) {
            try {
                ctx.eval(ctx.node(2)); // evaluate body
                ctx.eval(ctx.node(1)); // re-evaluate condition
            } catch (CFlowSignal sig) {
                if (sig == Break)    break;
                if (sig == Continue) continue;
            }
        }
    }
}
```

Listing 4.18: Example CFlow Signals Handling through Exceptions.

# 5

## Case Study: Evolution of a DSL through Composition

Domain-specific languages are computer languages designed to tackle problems that are tied to a particular problem-domain. Studies pointed out [62] that up to 80% of a software system lifetime is spent on maintenance and evolution activities, and DSLs are no exception: continual evolution of DSL implementations is often difficult because it is generally unplanned and unanticipated. Componentized language development leads to language implementations that can be easily extended and evolved.

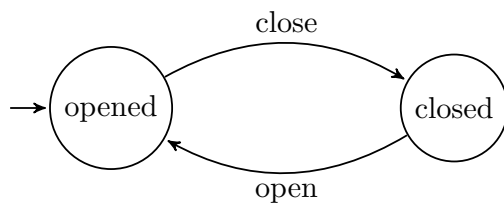
This section is devoted to show a simple but complete usage example of Neverlang. The example is the same state machine domain-specific language in [97]. Just like in Tratt's paper, the DSL will be *evolved* through *extension*; but, in our case, we will show that the same kind of language extension can be achieved in Neverlang using *language components*. In Sect. 6.1 the same experience will be discussed, comparing other language frameworks. The source code listings in this section may have been edited for readability, the full working example with source code can be downloaded at

<http://neverlang.di.unimi.it/vacchi/examples.tgz>.

Auxiliary Classes	
Transition	A data class of three fields: <i>to</i> , <i>from</i> , and <i>event</i>
GuardedTransition	A subclass of <i>Transition</i> that supports guards and actions
TransitionTable	Maps a state into the transitions that leaves that state

**Table 5.1:** *Auxiliary Classes for the State Machine DSL.*

## 5. Case Study: Evolution of a DSL through Composition



```
// door.sm
state machine Door {
  state opened state closed
  transition from opened to closed:
    close
  transition from closed to opened:
    open
}
```

Figure 5.1.: Door State Machine.

```
// sm.Program
Program ← "state" "machine" Identifier  "{" StateList TransitionList "}";
// module sm.State
State ← "state" Identifier;
// sm.Transition
Transition ← "transition" "from" Identifier "to" Identifier ":" Identifier;
// sm.StateList
StateList ← State StateList;
StateList ← State;
// sm.TransitionList
TransitionList ← Transition TransitionList ;
TransitionList ← Transition;
```

Listing 5.1: Grammar of the State Machine DSL.

### 5.1. A Simple State Machine DSL

Although often represented graphically, state machines are also conveniently represented in the form of *text* using a DSL. In this example, the DSL translates the state machine defined in the input program into compilable source code. The first version of the language only supports defining a list of states and a list of transitions between states. Each state is indicated through its Identifier; a transition is a triplet of Identifiers that represent, respectively, the name of the state from which the transitions leave, the name of the one where the transition goes, and a name for the transition itself. The first declared state is by convention also the *initial* state of the machine. Figure 5.1 shows the state machine for a door, along with the code that describes it; this machine loops indefinitely.

Listing 5.1 shows the grammar of the language as broken down into the *reference syntax* section of 6 modules, each of which represents a *syntactic feature* of the DSL (the Neverlang syntax has been omitted for conciseness). Three *evaluation phases*, in Neverlang, *roles*, (Sect. 4) have been defined: collect-states, validation, code-gen. Each role represents a different *concern* in the domain-specific language. The collect-states role collects the list of states in a Java Set<State>; the validation verifies that no undefined states were used in transitions, that are put in their own TransitionTable:

```

module sm.base.Program {
  reference syntax form sm.Program
  role(collect-states) {
    2 .{
      // pulls a List<State> of attributes called "state"
      // defined on each nonterminal "State"
      // in StateList ← State StateList; StateList ← State;
      // notice that generics are supported
      List<State> states = AttributeList.collectFrom($2, "state");
      $0.initialState = states.get(0);
      $2.stateSet = new java.util.HashSet<State>(states);
    }.
  }
  role(validate) {
    3 .{
      // same for attribute "transition" of nonterminal "Transition"
      List<Transition> transitions = AttributeList.collectFrom($3,"transition");
      Set<State> states = $2.stateSet;
      // validate each transition
      for (Transition t: transitions) {
        if (!states.contains(t.from()) || !states.contains(t.to()))
          throw new Error(''Undefined states in transition ${t.name()}'');
      }
      // all transitions are valid, proceed to fill the table
      // usage of a stateful table here is for instructional purposes;
      // an attribute $3.transitionTable would have worked as well.
      $$TransitionTable.addAll(transitions);
    }.
  }
  role (code-gen) { ... }
}

```

Listing 5.2: A snippet from module *sm.base.Program*.

it may raise an error if an undefined state is encountered; the code-gen role generates compilable (Java) source code implementing the state machine. Listing 5.2 shows a few lines of code from the collect-states and validation phases for module *sm.base.Program*, which implements the semantics for the syntax defined in *sm.Program*. Notice that attributes are pulled from *StateList* and *TransitionList* using a Neverlang API (*AttributeList.collectFrom()*), which basically implements the bucket brigade operator [52]. This frees users from defining specific, repetitive semantics for rules of the form  $XList \leftarrow X XList$ ;  $XList \leftarrow X$ . Thus, the modules *sm.StateList* and *sm.TransitionList* can be used as they are in the language implementation, without additional semantic definitions. The usage of this API is the preferred way to deal with such cases. Please notice that pulling up states and doing the analysis here is not idiomatic in attribute grammars, where you would rather pass the list of states down the tree so that each transition would perform the validation. Of course, this is possible

## 5. Case Study: Evolution of a DSL through Composition

```
module sm.base.Program {
  // ...
  role(code-gen) {
    0.{
      String className = "StateMachine_" + $1.identifier;
      StringBuilder sb = new StringBuilder();
      Set<State> states = $2.stateSet;

      // for each state, construct a "case" clause
      for (State s: states) {
        String caseStr = '''
          case ( "${ s.name() }" ):
            System.out.println("${s.name()}");
          ''';

        // there will be only one transition per state
        // otherwise the automaton would be non-deterministic!
        for (Transition t: $$TransitionTable.values()) {
          if (t.from().equals(s)) {
            String nextStateString = '''nextState = "${ t.to().name() }";''';
            caseStr += nextStateString;
          }
        }
        caseStr += "break;";
        sb.append(caseStr);
      }
      State initialState = $0.initialState;
      String code = /* ... generate the class body ... */;
      $0.Text = code;
      $0.className = className;
      System.out.println(code);
    }.
  }
}
```

Listing 5.3: A snippet from the code-gen role in `sm.base.Program`.

in Neverlang as well.

The `TransitionTable` implementation may be provided to the language using an *endemic slice*. The `TransitionTable` may be defined as a map between states and a list of states between which a transition exists. For instance, in the door state machine (Fig. 5.1) the opened state should return the set `{closed}`. Obviously in a deterministic state machine only *one* transition should leave from each state; thus, the validation role may also check that, for each declared state, the size of its entry in the table is less or equal to 1. A `Transition` may be implemented as a custom Java data class, that modules would import. Similarly, the `TransitionTable` and `StateSet` companion classes for the corresponding endemic slices should be written, as described in Sect. 4.1.2; we will omit the source code for these components, since they are trivial to write. A summary of the support classes is shown in Table 5.1.

The semantic action for the `Program` nonterminal in the code-gen phase (Listing 5.3) produces compilable code for the state machine (Listing 5.4). In this case, the generated source code is a simple Java program with a while loop that switches over the possible states of the machine, setting the variable `nextState` when a transition exists.

```
String nextState = "opened"; // initial state
while (true) {
  switch (nextState) {
    case "opened" :
      System.out.println("transition close");
      nextState = "closed";
      break;
    ...
  }
}
```

**Listing 5.4:** Compiled code for the Door state machine.

In order to use the code-generating language processor for this language, a language descriptor (Sect. 4.1) `sm.base.Lang` will be defined. Then it is possible to invoke the `nlgc` tool (Sect. 4.3) on the implemented modules and slices to produce the source code interfacing with the Neverlang API (Sect. 4.4). The code will be compiled using `javac`<sup>1</sup>. At this point, the generated class files will be executable as a self-contained language implementation using `nlg` or `nlgc` (Sect. 4.3), *callable* from a regular JVM program (Sect. 4.4.2), and *reusable* across different language implementations without any change. For instance, to execute the input program in Fig. 5.1, one would simply write at a prompt:

```
$ nlg sm.base.Lang door.sm
```

producing the compilable source code in Listing 5.4 (support APIs are provided to automatically generate an output file on disk).

## 5.2. A Simple Imperative Language

Executable UML models include the specification of an *action language* [61] that can be used for many purposes, such as expressing actions and guards in a state machine model. For instance the ALF specification<sup>2</sup>, describes a stand-alone, imperative programming language with a Java-like syntax. In this example, let us suppose that we already have an implementation of a suitable language for this purpose, that is a simple, imperative programming language with support for variables and expressions like the `java.lang.Lang` language that we used as our running example in Sect. 4. For instance, it is easy to see that by combining the syntax definitions in Listing 5.5 with the `whileLoop` definition that we have seen in the previous sections, we would have enough components to define a simple Turing-complete programming language. As you can guess from the grammar, `al` (Action Language) supports only two types: numbers

<sup>1</sup>of course, as seen in Sect. 4, if any other language is used in the semantic actions, users will have to invoke the language-specific compiler as well; *e.g.*, for Scala, it would be `sca1ac`

<sup>2</sup>see <http://www.omg.org/spec/ALF/1.0.1/>

## 5. Case Study: Evolution of a DSL through Composition

```
// module al.BoolExpr
BoolExpr ← BoolOperand;
BoolExpr ← BoolExpr "&&" BoolOperand;
BoolExpr ← BoolExpr "||" BoolOperand;
// module al.RelExpr
RelExpr ← RelOperand ;
RelExpr ← RelExpr "<" RelOperand;
RelExpr ← RelExpr ">" RelOperand ;
...
RelExpr ← RelExpr "==" RelOperand ;
// module al.VarDef
VarDef ← Identifier "==" Expr;

// module al.Sum
SumExpr ← Term;
SumExpr ← SumExpr "+" Term;
SumExpr ← SumExpr "-" Term;
// al.Term
Term ← Const;
Term ← VarLookup;
// module al.VarLookup
VarLookup ← Identifier;
// module al.Const
Const ← /[0-9]+/;
// omissis: al.Statement
// omissis: al.StatementList
```

**Listing 5.5:** Relevant parts of the grammar for the Action Language.

```
module al.Sum {
  reference syntax {
    [st] SumExpr ← Term;
    [sp] SumExpr ← SumExpr "+" Term;
    [sm] SumExpr ← SumExpr "-" Term;
  }
  role (code-gen) <template> {
    [st] .{ {{ $st[1].Text }} }.
    [sp] .{ {{ $sp[1].Text }} + {{ $sp[2].Text }} }.
    [sm] .{ {{ $sm[1].Text }} - {{ $sm[2].Text }} }.
  }
}
```

**Listing 5.6:** Example code for Sum in the Action Language. Code generation uses the template syntax.

(integers) and booleans. For simplicity, variables can be only assigned integer values, and undeclared variables are assigned the default value  $-1$ .

Two roles are defined: `validate` and `code-gen`. The `code-gen` role generates Java source code, thus it is compatible with the `code-gen` role of the state machine DSL. Listing 5.6 shows the module for the Sum definition. The `code-gen` phase uses the template syntax (see Sect. 4.3). For instance, compare the valid `al` program to compute the factorial of  $n$  in Listing 5.7 (on the left) with the Java/C source code (on the right) that the `code-gen` phase produces. The `validate` role keeps track of the used variables, so that the `code-gen` role may declare them at the top of the listing. You may also be able to see that it would be easy to extend the language with a module `VarDecl` to declare variables: the `validate` role of the `VarLookup` module could then raise an error when users attempt to use an undeclared identifier. In this case a `VarTable` would keep track of the defined variables, thus one should include an endemic slice for this purpose.



```

// al program to compute n!
n := 4; i := n; acc := 1;
while (i > 0) {
  acc := acc * i;
  i := i - 1;
}

int n = -1, i = -1, acc = -1;
n = 4; i = n; acc = 1;
while (i > 0) {
  acc = acc * i;
  i = i - 1;
}

```

Listing 5.7: al language and the result of the code-gen role.

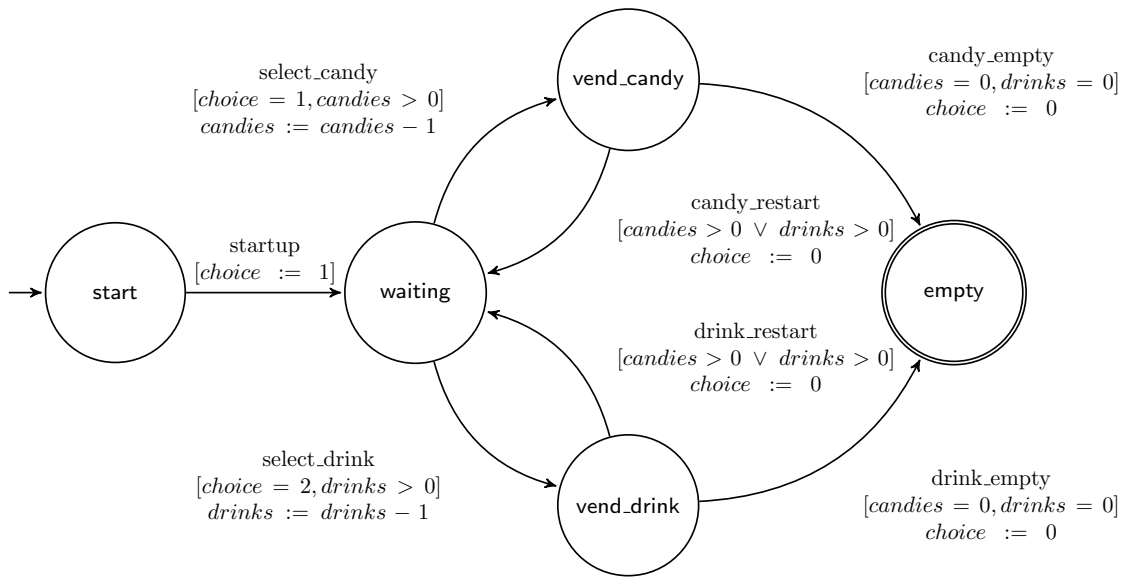


Figure 5.2.: Vending Machine.

### 5.3. Guards and Actions: Composing the DSLs

The al language would be useful to introduce an interesting feature in our original DSL: *guards* and *actions*. Figure 5.2 and Listing 5.8 show the state chart of a *vending machine*. The machine vends drinks and candies, depending on an initial choice, which is an integer value—that is, 1 for candies, 2 for drinks, and 0 for neither. Once a candy or a drink has been vended, the machine resets the choice to 0, and it goes back to the initial waiting state, unless both candies and drinks are unavailable, in which case the machine goes to the empty state. The example requires us to introduce the concepts of *variable*, *guard* and *action* to transitions: the *guard* is a *boolean expression* that causes a transition to fire only when it evaluates to true, an *action* is a sequence of statements of the action language that are executed when a transition fires, and a *variable* is an identifier that is associated with an integer value. All these concepts can be described in terms of components of the al language.

A *guarded transition* is almost the same as a simple Transition of the base implemen-

## 5. Case Study: Evolution of a DSL through Composition

```
state machine VendingMachine {
  state start state waiting state vend_candy state vend_drink state empty
  transition from start to waiting : startup { choice := 1; }
  transition from waiting to vend_candy :
    select_candy [ choice = 1 && candies > 0 ] { candies := candies - 1; }
  transition from vend_candy to waiting :
    candy_restart [ candies > 0 || drinks > 0 ] { choice := 0; }
  transition from vend_candy to empty :
    candy_empty [ candies = 0 && drinks = 0 ] { choice := 0; }
  ...
}
```

**Listing 5.8:** Code for the Vending Machine in Fig. 5.2. Code for drinks is omitted, since it mirrors the candies side..

```
module sm.ext.GuardedTransition {
  reference syntax {
    Transition ← "transition" "from" Identifier "to" Identifier ":"
                Identifier GuardAction;

    [ga] GuardAction ← Guard Action;
    [g]  GuardAction ← Guard;
    [a]  GuardAction ← Action;

    [gg] Guard ← "[" BoolExpr "]";
    [aa] Action ← "{" SMActionList "}";
    ...
  }
  role (code-gen) {
    0.{
      String from = $1.identifier;
      String to   = $2.identifier;
      String identifier = $3.identifier;
      String guard  = $4.guard;
      String action  = $4.action;

      $$TransitionTable.addTransition(from, to,
                                     GuardedEvent.of(identifier, guard, action));
    }.
    [ga] .{
      $ga[0].guard = $ga[1].Text;
      $ga[0].action = $ga[2].Text;
    }.
    [g] .{
      $g[0].guard = $g[1].Text;
      $g[0].action = null;
    }.
    ...
  }
}
```

**Listing 5.9:** Implementation of the GuardedTransition..

tation, but it is followed by a guard—a boolean expression between brackets—and/or by an action—a sequence of assignments. In a state machine with guards, a transition *fires* only when its guard evaluates to true; therefore, now multiple transitions may

```

int choice = -1, candies = -1, drinks = -1;
String nextState = "waiting"; // initial state
while (true) {
    switch (nextState) {
        ...
        case "vend_candy" :
            // transition from vend_candy to waiting: candy_restart
            if (candies > 0 || drinks > 0) {
                choice = 0;
                System.out.println("transition candy_restart");
                nextState = "waiting";
                break;
            }
            // transition from vend_candy to empty: candy_empty
            if (candies == 0 || drinks == 0) {
                choice = 0;
                System.out.println("transition candy_empty");
                nextState = "empty";
                break;
            }
            ...
        }
    }
}

```

Listing 5.10: Compiled code for the Vending Machine..

leave the same state. This extension can be realized 1) by adding a new component to the language that implements a transition with a guard and an action, *alongside* the original “simple” transition, and 2) modifying the code-gen phase so that multiple transitions leaving the same state can be accounted for.

The new module is called `sm.ext.GuardedTransition` (Listing 5.9) and it uses a companion Java class `GuardedEvent`. As you can see, the syntactic definition is similar to the one in `Transition.nl` (Listing 5.1, p.56), this does not introduce conflicts because of the new `GuardAction` nonterminal, that appears only when the transition statement is followed by a guard, an action or both. During the code-gen phase the new transition is added to the `TransitionTable`. This transition contains the generated code for the guard expression and the assignment statements in the action body: the code-gen role from the `al.BoolExpr` slice and the `al.StatementList` would pass on the generated code through the code attribute defined on their nonterminals. Listing 5.10 shows a part of the generated source code for the vending machine in Listing 5.8.

The new slices can be introduced alongside the old ones; only one substitution is required: the code-gen phase in the `Program` slice must now be aware that more than one transition may leave a state, and that guards and actions should be printed out (Listing 5.11).

Finally, Listing 5.12 shows all the slices that have been included in the complete

## 5. Case Study: Evolution of a DSL through Composition

```
0.{
  ...
  for (State s: states) {
    ...
    for (Transition t: localTs) {
      String nextStateString =
        '''nextState = "${ t.to().name() }";''';
      Event evt = t.event();
      if (evt instanceof GuardedEvent) {
        GuardedEvent ge = (GuardedEvent) evt;
        if (ge.guardText() != null) {
          caseStr += '''if ( ${ge.guardText()} ) ''';
        }
        if (ge.actionText() != null) {
          caseStr += '''{
            ${ge.actionText()}
            ${nextStateString}
          }''';
        } else {
          caseStr += nextStateString;
        }
      } else {
        caseStr += nextStateString;
      }
    }
    ...
  }
  ...
}.
```

**Listing 5.11:** A snippet from the code-gen role in *sm.ext*.Program..

language implementation:

- the *sm.\** package contains *sm.StateList* and *sm.TransitionList*, simple syntactic definitions where no additional semantics has been defined.
- the *sm.base.\** package denotes slices defined for the basic state machine language
- the *al.\** package denotes slices defined by the imperative language, that were used in the guard/action language
- the *sm.ext.\** package denotes the slices that were explicitly (re)defined for the extended state machine language with guards and actions.

**Observations** This example showed how to implement a DSL as a collection of components, each representing a *concept* or *feature* of the language. The component-based model, however, shows that it is possible to improve code reuse of pre-defined features, possibly coming from different languages. The model makes it possible to reuse pre-packaged bundles of syntax and evaluation phases across different language imple-

```

language sm.ext.Lang {
  slices
    sm.base.State           al.Term           sm.ext.ProgramSlice
    sm.base.Transition      al.VarLookup     sm.ext.GuardedTransition
    sm.StateList            al.SumExpr
    sm.TransitionList       al.RelExpr
    sm.base.Identifier      al.BoolExpr

  endemic slices
    sm.base.SMBuilder al.VarTable

  roles syntax < collect-states < validate < translate
}

```

**Listing 5.12:** Slices included in the state machine language with guards and transitions.

mentations, using *language components*; nevertheless, it makes it possible to easily reuse syntax and evaluation phases in different language components, both making it easier to produce variants of the same DSL or to reuse the same components into language implementations that have different requirements. For instance, in this example we produced a state machine that generated compilable Java source code. The code-gen phase could be easily traded with a different implementation, generating code for an alternative target language; for instance, Sect. 4.1 showed how to generate bytecode using Jasmin and the template translator plugin. Another alternative processing phase may generate a graphic representation using Graphviz's dot syntax [28]; finally the state machine could be *interpreted*. Each of these changes do not require any editing on the existing source code, but they rather consist in the creation of *new modules* and then wiring together the desired *roles* by defining *new slices*. The original, pre-compiled implementations can be left untouched on disk. This development model has further implications: independent *aspects* of the language could be deployed separately in distinct implementations of the same language. For instance, *logging* and *tracing* concerns may be packaged only with the *development* version of a DSL; or they could be packaged together and deployed dynamically *on demand*. They could even be *loaded dynamically* at runtime.



# 6

## Evaluation

We will now present a series of experiences directed towards the evaluation of our model of language implementation. First, we will once again use the implementation of the componentized state machine DSL of Sect. 5 to compare a selection of other modular language implementation frameworks, to show that the model is general enough to be reproducible using different tools. Then, by the help of the experiences that we carried out through Neverlang, we will show the benefits that a native implementation of this model would carry. We show that language extension is simplified by a feature-oriented language implementation; this experience has been carried out by first implementing an interpreter for a real-world programming language (JavaScript). Language components can be *compiled separately*, they can be redistributed as pre-compiled artifacts, and they can be *loaded dynamically*; not only is it possible to reuse code, but extensions can be implemented in isolation and loaded on demand; making it possible to *evolve* a language implementation even *atruntime*. In the last subsection we will discuss the expressive power of Neverlang through the implementation of the DESK language [74].

### 6.1. Feature-Oriented Language Implementation Across Tools

In Chapter 3 we presented a model of feature-oriented language composition. The basic idea is to be able to encode features of a language as *self-contained language components*, that can be reused across different language implementations. In Chapter 5 we presented a simple, but full example of the model in action, using Neverlang. In order to evaluate this model, we tried to reproduce the same experience in other frameworks for modular language development. We will not go into the details of implementing the state machine DSL itself, but we will use this language as a way to discuss the features of each framework and the way the model fits in their design. The

## 6. Evaluation

```
language StateMachine {
  lexicon {
    keyword    state | machine
    identifier [a-zA-Z_][a-zA-Z0-9_]*
    ...
  }
  attributes sm.model.StateMachine STATEMACHINE.sm;
  ...
  rule statemachine {
    STATEMACHINE ::= state machine #identifier
                  #lbrace STATE_LIST TRANSITION_LIST #rbrace

    compute {
      STATEMACHINE.sm = newStateMachine(#identifier.value(),
        STATE_LIST.states, TRANSITION_LIST.transitions);
    };
  }
  method StateMachineFactory {
    import sm.model.*;
    import java.util.List;
    public StateMachine newStateMachine(String id, List states, List
      transitions) {
      return new StateMachine(id, states, transitions);
    }
    ...
  }
}
language GuardedStateMachine extends StateMachine, ActionLang {...}
```

**Listing 6.1:** A snippet from the State Machine language implementation in LISA.

experience were conducted using LISA 2.2, Silver r1230 (hg), Spoofox 1.2.0.0-s41399, Xtext 2.5.3. Full source code of the examples can be found at

<http://neverlang.di.unimi.it/vacchi/examples.tgz>.

At the end of this section we will summarize the results of this experience.

### 6.1.1. LISA

The LISA [63, 65] modular language development framework achieves *language composition* through an extension to *attribute grammars* (Sect. 2), supporting *multiple inheritance* of language specifications. It is possible to inherit language definitions from several specifications and to compose them into one language. For instance, the *state machine language* with guards and actions can be seen as the composition of two language specifications: *StateMachine* and *ActionLanguage*, with the code for guarded transitions being the glue between the two.



```

// StateMachine          // ActionLanguage          // GuardedStateMachine
rule StateMachine      rule Term                rule START
rule State              rule VarLookup           rule GuardedTransition
rule Transition         rule SumExpr            rule GuardedTransition
rule StateList          rule RelExpr            rule GuardedTransition
rule TransitionList    rule BoolExpr           rule GuardedTransition

```

**Listing 6.2:** Rules sections defined for the State Machine DSL.

In Lisa, *syntax definitions* are represented in the **rule** section of a **language** specification (Listing 6.1). LISA's **rule** section can be seen as a combination of a syntax definition with its semantics (Sect. 3). A **rule** section may contain one or more productions and programmers may refer *attributes* on the nonterminals of such productions; such sections are usually small and define semantically-related portions of the language. As such, we may say that they concisely encode *language components* (Chapter 3). In fact, they can be substituted (*overridden*) and composed (by way of language inheritance). The implementation of the state machine language with guards and transition followed approximately the partitioning scheme used for Neverlang in Sect. 5 (see Listing 6.2). Dependencies between such components are implicitly given by the syntax definitions (relations between productions) and the attributes that are defined and used between portions of attribute grammars, that is, **rule** sections.

LISA does not provide a first-class construct to partition rules into *evaluation phases* (Neverlang's roles); however, it *is* possible to achieve this kind of separation of concerns by logically partitioning the attributes with respect to the concerns they pertain to. For instance, the most simplistic approach may be to resort to a *naming convention*, such as prefixing with a phase "identifier" those attributes that relate to a particular phase (e.g., `validation_attribName`), which is perfectly fine for simple languages like the State Machine DSL. In this case, one attribute may be used to validate the state machine, or an error could be thrown; another attribute could be used to accumulate the generated source code (for the equivalent of a code-gen role in Neverlang), which in LISA is done through concatenation of strings; since there is no dedicated syntax for this purpose, we moved the bulk of the code generation to the `toString()` method of the support classes (Sect. 5) for convenience.

A better way to factorize the evaluation phases would be to exploit LISA's multiple inheritance, and define evaluation phases as separate languages that *inherit* from a base language, which would define the syntax. Because multiple inheritance is allowed, it is conceptually possible to partition attribute definitions into phases. Then, for each phase, one would define a different **language** specification.

LISA's model is able to achieve a similar degree of componentization of the conceptual model of Sect. 3, although using a different strategy, which is to be ascribed to the use of inheritance. Because of this model, code reuse may be harder in some situations, and easier in others. For instance, inheritance natively supports language *extension*; thus **rule** sections in a language specification cannot be "cherry-picked" in a way self-contained language components would allow, but only *inherited* in bulk. Suppose

## 6. Evaluation

```
grammar sm:statemachine;
import sm:transition;
import sm:transitionlist;
...
terminal State_t 'state' ;
terminal Machine_t 'machine' ;
...
synthesized attribute name :: String;
nonterminal StateMachine_c with sname, stateList, transitionList;
concrete production statemachine_c
s::StateMachine_c ::= State_t Machine_t
  id::Identifier_t LBrace_t
  sl::StateList
  tl::TransitionList RBrace_t
{
  s.stateList = sl.stateList;
  s.sname = id.lexeme;
  s.transitionList = tl.transitionList;
}
```

**Listing 6.3:** A portion of the Silver grammars for the State Machine DSL..

that a **rule** section of language  $L_a$   $S_1$  requires nonterminal  $X$  in **rule** section  $S_2$ , and suppose that  $L_b$  inherits from  $L_a$ , but only wants to use  $S_1$  and not  $S_2$ ; the only way to disallow rules in  $S_2$  is *semantically* through the technique described by Erdweg *et al.* [29] to implement language restrictions through extension. Thus, the language processor would *grow* in size and complexity without any real benefit. The solution in this case would be to factor a language as a collection of smaller “sub-languages” only containing one **rule** section each. The only limit to this model would be that it is still not possible to separate the definition of the syntax from the definition of the semantics.

In Sect. 3 we also assumed that languages may have visibility on globally-accessible resources, declare new data structures, support libraries, etc. This role is played by LISA’s **method** section. In this section a **language** may define data structures and operation on said data structures that should be globally available throughout all the **rule** sections of the input source.

### 6.1.2. Silver

Silver [101] is a statically-typed domain-specific language for the implementation of *modular* attribute grammars. A module in Silver is simply called a **grammar**. Grammars define both the (abstract or concrete) syntax and the semantics of a part of a language. The framework encourages the development of self-contained language extensions called *composable* extensions, which are those extensions that can be used in conjunction with other language extensions typically designed without knowledge of one another.

```

// base state machine // action language // extended state machine
sm:base:statemachine al:term sm:guards:main
sm:base:state al:varlookup sm:guards:guardedtransitions
sm:base:transition al:sumexpr
sm:base:statelist al:relexpr
sm:base:transitionlist al:boolexpr

```

Listing 6.4: Silver grammars for the State Machine DSL.

In Silver a *language component* may be implemented through a collection of grammars. A grammar in Silver is equivalent to a *module* in a general purpose programming language. A language is the composition of several grammars: a grammar, usually named `Main` contains a **parser** section that specifies the grammars that should compose into the full language implementation, yielding the language processor.

Grammars define **production** rules, terminal and nonterminal symbols, and the attributes that may occur on said nonterminals. Each **production** declaration introduces a code section which contains the attribute definitions of the nonterminals. For instance, Listing 6.3 is a detail from the implementation of the state machine DSL.

Even though it seems like Silver's grammars are supposed to be self-contained, in that the syntax of a construct and its semantics (as attributes) form a single code section (similarly to LISA) it is still possible to inject extra-attributes, and therefore extra-semantics onto the same production using **aspect productions**. Aspect productions may be defined in separate modules. Therefore, although there is no formal support for a concept of *compilation phase*, it is certainly possible to use aspect productions to factorize the semantics in such a way. In Silver, it is also possible to define **abstract** productions as opposed to **concrete** productions. A concrete production is the concrete syntax definition of a language construct; abstract productions represent an abstract data structure, that can be constructed during the processing of the parse tree to *map* the concrete syntax onto an abstract representation, or to implement a data structure. For instance, in our state machine DSL, abstract productions would be used to implement the equivalent of the `Transition` class in Table 5.1.

Extra support code can be declared in functions. In a certain sense, functions and **abstract** productions may play the role of the *globally-scoped components* that a language may refer; that is, ancillary data structures or code routines that are used throughout the entire language implementation. However, while in the model the idea is that these components may be easily swapped with alternative implementations, in Silver these components must be **imported**, thus creating a *dependency* between the two modules. This is required in order for the Silver compiler to be able to type-check correctly: in fact, as we said at the beginning, one of Silver's peculiar features is *static typing*. On the other hand, this means that you cannot easily *substitute* the imported module with a different implementation, unless you provide another module with the same exact name. In Neverlang this is in part possible thanks to its architecture, based on *module* and *slice* composition; on the other hand, in Neverlang static checking is not currently performed, even though it could be implemented *on top* of the current framework. Silver performs this kind of checks, but because it currently does not support type

## 6. Evaluation

```
Start.StateMachine =  
    < state machine <ID> { <State*> <Transition*> } >  
State.State = < state <ID> >  
Transition.Transition = < transition from <ID> to <ID> : <ID> >
```

**Listing 6.5:** *Syntax definitions in SDF3.*

classes or a construct alike<sup>1</sup> it is unfortunately not possible to abstract away from the actual implementation of the module. Nevertheless, the issue can be overcome by resorting to abstract syntax definitions, and remapping the concrete syntax onto an abstract representation.

The state machine example in Silver, from the syntactic point of view, can be factored out similarly to the base example in Chapter 5. Yielding the grammars in Listing 6.4. From the semantic point of view, similarly to the LISA example, we resorted to simple naming conventions on attributes to separate validation and code generation. The validation phase can be implemented using an attribute to report validity of the input program, or raising an error using the `error(msg)` built-in function. Code generation can be done through string concatenation of an attribute (in Silver, conventionally `pp` for *pretty printing*), but Silver makes available utility functions to pretty print and generate output code; for instance it supports string interpolation through the template string syntax. The separation could be also made more flexible through **aspect productions**: in this case, one would define one module for the syntax of each construct, and then, for each one of them, a module implementing each evaluation phase (validation and code generation). It is worth noticing, though, that, since Silver is a *purely functional* language, *stateful* data structures would not actually fit within the system's programming model: so, although support libraries are supported, this limit is by design.

Finally, Silver also supports *attribute forwarding*, this feature makes it possible to define attributes on a node in terms of the attributes of *another subtree*. For instance, the `GuardedTransition` could be partially defined in this way (see the Sect. 6.1.5 for more details.)

### 6.1.3. Spoofox

Spoofox [56] is an Eclipse-based language workbench that implements a complete tool suite to implement DSLs and code generators. The Spoofox workbench, besides compilers for an input DSL, also generates the platform tooling for language editing, i.e., an Eclipse plugin with syntax highlighting, auto-completion, integrated error reporting, etc.

In the implementation of the state machine language the syntax was defined using the SDF3 syntax specification language; SDF3 makes it possible to define the syntax of a language, and the tree the concrete syntax should map to, by way of a template language

<sup>1</sup>«When Silver begins to support type classes (it doesn't yet, but probably will)» [https://code.google.com/p/silver/wiki/Reference\\_Function](https://code.google.com/p/silver/wiki/Reference_Function)

```

to-java:
  GuardedTransition(s1,s2,t, Guard(guard), Action(action)) -> $<
    if (<guard'>) {
      <action''>
      nextState = "<s2>";
      break;
    }
  >
with
  guard'   := <to-java> guard
; action' := <map(stmt-to-java)> action
; action'' := <concat-strings> action'

```

**Listing 6.6:** *to-java* rule for term `GuardedTransition(state, state, id, guard, action)`.

(Listing 6.5). SDF2, which uses a variant of a BNF-like syntax, is also available. In this template language, instead of using grammar productions, the syntax is specified by way of syntax snippets, where *placeholders* act as nonterminals (similarly to the examples in Chapter 3). Syntax definitions written in SDF map the concrete syntax onto an *abstract representation* called *term*, which are then analyzed and manipulated throughout the evaluation phases. Because Spoofox is a Workbench, each evaluation phase may be triggered by specific user actions; in particular syntax checking and coloring is performed as you type, program analysis (name resolution etc.) is done on save; code generation is performed by clicking a button.

Spoofox provides DSLs for simplifying the description of common language processing tasks (or *concerns* [103]). For instance, the state machine DSL's *validation* phase can be written using the built-in *name binding language*, *NaBL*. This domain-specific language does actually more than just error reporting, though; it highlights incorrect references in an input program (see Fig. 6.1), it makes it possible to *jump* from the use-site of an identifier to its corresponding declaration site, etc. Other DSLs currently under development are *TS* for type analysis and *DynSem* for specification of executable operational semantics to generate interpreters [103]. Nevertheless, it is possible to specify custom processing phases using the Stratego/XT [12] program transformation language and platform, which is the core of the platform itself, although the underlying runtime system is the JVM<sup>2</sup>. For instance, the editor provides predefined hooks for source code generation, but the code generation phase is written using Stratego.

The Stratego language is itself a *declarative, dynamically-typed* programming language for *term rewriting*. In the case of languages, a term can be seen as a node in an abstract syntax tree. For instance the node `State(ID("Initial"))` may be generated from the syntax definition of `State` in Listing 6.5, and the input program in Listing 5.8, and it would denote the state called `Initial`.

Stratego programs are made of *rules* and *strategies*. A *rule* defines a *transformation*, that is, a partial function that maps terms into terms; rules are defined by way of

<sup>2</sup>The original implementation was in C

## 6. Evaluation

```
grammar ex.StateMachines with org.eclipse.xtext.common.Terminals
generate stateMachines "http://www.StateMachines.ex"
Model:
    statemachine=StateMachine;
StateMachine:
    'state' 'machine' name=ID '{'
        states+=State+ transitions+=Transition+ '}';
State: 'state' name=ID;
Transition:
    'transition' 'from' from=[State] 'to' to=[State] event=Event;
```

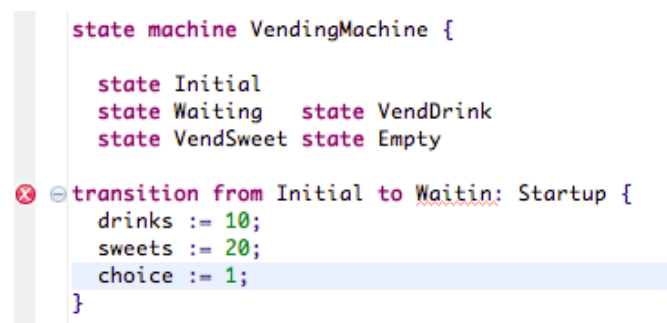
Listing 6.7: State Machine grammar in Xtext.

pattern matching over the form of a term. In a language implementation, a *strategy* can be seen as a way to describe the order of a visit on a syntax tree; rules are the transformations that are iteratively applied over matching nodes of the tree up until the visit has completed. For instance, in Listing 6.6 strategy map is being applied to rule `stmt-to-java` (not shown here) with action as input term. In this case, action is a list of statements of the action language, that the `stmt-to-java` rule *transforms* into strings.

Although feature-orientation is not the main purpose of Spoofox, it has been shown [60] that it may be used to model a feature-oriented language implementation. However, this experience has been carried out by *refactoring* an existing language implementation into components *after* it had been already implemented. In any case, a high-level of reuse can be easily achieved by writing *smaller* modules, that describe single linguistic features. This may require some discipline on the developer's behalf, but there is no technological limitation that would prevent it.

### 6.1.4. Xtext

The Eclipse foundation has endorsed for long the Xtext language workbench [27, 9]. Xtext is a framework that covers all the aspects of a complete language infrastructure,



```
state machine VendingMachine {
    state Initial
    state Waiting    state VendDrink
    state VendSweet state Empty
    transition from Initial to Waitin: Startup {
        drinks := 10;
        sweets := 20;
        choice := 1;
    }
}
```

The screenshot shows a code editor with a red 'x' icon and a tooltip indicating a reference error. The error message is "transition from Initial to Waitin: Startup". The word "Waitin" is underlined with a red squiggly line, indicating it is an unresolved reference. The rest of the code is highlighted in light blue.

Figure 6.1.: Spoofox highlighting a reference error in the Vending Machine example.

```

class StateMachinesGenerator implements IGenerator {
    ...
    def compile(StateMachine sm) {
        val tm = sm.transitionsToMap.compile
        val vars = if (vardefs.empty) ""
                    else "int " + vardefs.join(" = -1, ") + " = -1;"
        return '''
        public class «sm.fullyQualifiedName» {
            public static void main(String[] args) {
                «vars»
                String nextState = "«sm.states.get(0).name»";
                while(true) {
                    switch (nextState) {
                        «tm»
                    }
                }
            }
        }
        ''';
    }
    ...
}

```

Listing 6.8: State Machine body code generation in Xtend.

starting from the parser, up to a full Eclipse IDE integration (syntax highlighting, error reporting). Xtext is included here mostly because it is widely known among researchers and practitioners as the go-to tool to implement domain-specific languages in Java. In particular, Xtext comes with a fully-featured reusable expression language called Xbase, complete even of *lambda expressions*, that programmers can use as the basis to implement their own domain-specific Java-compatible domain-specific language. Using Xbase, programmers may easily map domain-specific language constructs onto a Java model, and enable interoperability between the custom DSL and a real Java codebase, which is the main objective of the Xtext platform. The Xbase expression language is not stand-alone (it cannot be run in its own interpreter or compiler) but it comes as a single, large grammar that can be reused in other language by using Xtext's *inheritance* mechanism.

The syntax of the state machine language is indicated through an ANTLR-like syntax (this is not by chance, since ANTLR [76] is the underlying parser generator), with extensions. For instance, a nonterminal between square brackets actually indicates a reference *by name* to a concept defined by another production rule. You can see an example in Listing 6.7: the [State] nonterminal in the rule for definition of Transition points to the State nonterminal; but what this really means is that users may write **transition from a to b**: t and then a and b will automatically “point to” declarations of the form **state a** and **state b**; in other words, clicking the identifier would



## 6. Evaluation

Framework	N.Rules	N. Components	LOC	Support Code	Total Size	Runtime Lib
<b>Neverlang</b>	37 rules (grammar+terminal defs)	12 slices 2 endemic slices	412 lines (including section declarations; ~150 lines of Java 7 code)	7 Java classes (6 support + 1 endemic impl.)	276K	784K
<b>LISA</b>	26 rules (+ lexeme defs)	4 languages (12 rule sections) 3 method sects.	202 lines (externalized in classes for convenience)	6 Java classes	256K	1584K
<b>Silver</b>	34 rules (+ terminal defs)	9 grammars	232 lines	(functions, included in LOC count)	221K	1020K
<b>Spoofax</b>	21 rules + library lexemes	1 SDF module, 1 NaBL module (name resolution / validation), 1 stratego module (code-gen)	221 lines	(auto generated)	1400K (uncompressed)	(Size of the Plugin)
<b>Xtext</b>	16 rules (EBNF) + library terminals	1 Xtext module (grammar+name resolution / validation) 1 Xtend source file (code-gen)	178 lines	(auto generated)	736K (uncompressed)	(Size of the Plugin)

**Table 6.1.:** Summary of the sizes of the different implementations.

make the editor jump to the reference, and using an undeclared reference would make the editor highlight the error. This is similar to what Spoofax does with the name binding DSL, but embedded within the grammar specification. This feature was used to implement the *validation* phase of the state machine language.

Further language processing can be done using Java or the Xtend language, editing the classes that Xtext automatically generates based on the grammar specification. The Xtend language is an Xtext-implemented superset of Java that enhances the Java language with a lot of syntactic sugar: for instance it supports higher-order functions, filtering, folding and mapping over collections before Java 8 was even released. The Xtend language is particularly suitable for code generation, as it comes with a convenient string interpolation syntax (Listing 6.8). We used this feature to implement the code generation phase of the state machine DSL.

Unfortunately, from a *language composition* perspective, the Xtext platform is limited to *single-inheritance*. Programmers have long realized that, in object-oriented programming, single inheritance is not expressive enough to factor out common features: such features must either be forced into a common parent (where they do not belong), or they must be duplicated in the components that should share them [85, 25]. This Xtext limit restricts severely the way grammars can be reused and shared. It is possible for different languages to interoperate, usually by defining a mapping over common data structures (*cf.* [9]), but not to easily combine the syntax of many different grammars, because only one can be inherited at a time. It is therefore easy to see that, although Xtext is perfectly capable of implementing a simple example like the state machine DSL, it will quickly not scale on larger examples, where many components are involved. For instance, because of single-inheritance it is not possible to inherit from both the basic state machine DSL and the action language grammars and introduce the `GuardedTransition` construct (Chapter 5). Trying to achieve a *finer-grained* composition model, factorizing grammars into *language components* would be even more complicated.

### 6.1.5. Summary

*Feature-oriented* language composition can be achieved if the language framework of choice provides facilities to modularize the language implementation both on the



## 6.1. Feature-Oriented Language Implementation Across Tools

	Neverlang	LISA	Silver	Spoofax	Xtext
Lexical Definition	Yes, In Productions	Yes	Yes	Yes	Yes
Syntax Rules	BNF-like	BNF	BNF	SDF2, SDF3	EBNF
Abstract Syntax Support	Reference / Concrete Syntax, Rewriting DSL under development	Support for Expression and Priority-based conflict resolution during parsing	Yes	Yes (Terms)	Yes (Mapping onto Model)
Attribute Definition	Implicit in Semantic Actions	Explicit clause	Explicit clause	Tree-rewriting based	Mapping onto Model
Higher-Order Attributes	Attributes can be of any arbitrary JVM type	Attributes can be of any JVM type	Production-valued	N/A	Attributes can be of any JVM type
Supported Languages for Semantic Actions	Java (up to 8+), Scala, Template, Tree Rewriting DSL, support for custom DSLs and other JVM languages supported through language plugins	Java, but currently no support for generics	Silver, Java through <i>foreign</i> keyword	Stratego, Custom DSLs for predefined phases, Java through extensions	Xtend, Java, JVM languages
Special Syntax for Code Generation	Template, Java String interpolation	No	String Templates	StrategoString Quotations	Xtend
Multiple Rule Evaluation Strategies	Yes, pre-order (with eval, allowing arbitrary visits), post-order	Yes	Yes, attribute-driven	Yes; custom strategies may be user-defined	N/A
<b>Language Composition Model</b>					
Composition Model	Language Components (slices, modules, roles)	Multiple Inheritance of Language Specs	Grammars (Modules)	Modules	Single Inheritance of Grammars
Separation Between Syntax Definition and Semantics Specification	Reference Syntax / Roles	Inheritance and overriding	Selective imports, abstract and aspect productions	Selective imports and abstract syntax definitions	Model mapping
Separation Between Evaluation Phases	Roles	Inheritance or naming conventions (no formal notion of phase)	Selective imports and naming conventions; aspect productions may be used	Selective imports and predefined DSLs (name resolution, etc.)	References for name binding; hooks for code generation. No formal notion of evaluation phase
Self-Contained Language Components	Slices	Inheritance+ Factorization	Imports+ Factorization	Imports+ Factorization	N/A (single inheritance)
Reuse of Semantics through Syntax Rewriting	Remapping, Renaming, Tree Rewriting DSL	N/A	Attribute Forwarding	Tree-rewriting based	N/A
Language Extension	Yes	Yes	Yes	Yes	Yes
Language Restriction	Yes	Through Extension or Refactoring	Through Extension or Refactoring	Through Extension or Refactoring	Through Extension or Refactoring
Language Unification	Yes	Yes	Yes	Yes (modulo refactoring)	No
Extension Composition	Yes	Yes	Yes	Yes	No
Self-Extension	No	No	No	No	No
<b>Platform/APIs</b>					
Runtime Loading of Components / Evaluation Phases / Actions	Yes	No	No	Yes, dynamic loading of strategies	No
Supported Languages for Interacting with the Platform	Any JVM language, Neverlang language	Java, JVM languages	Silver, Java	Java, Stratego	Xtend, Java, JVM languages
<b>Generated Artifacts</b>					
Parsing Backends	DEXTER, custom drop-in replacement can be written	Yes, several: LR, LL, LALR, etc.	Copper	LR, JSGLR	ANTLR
Separate/Incremental Compilations	Yes	No	Yes	No	Only for semantics
Pre-Compiled Language Components	Yes	No	Yes	No	Only for semantics
IDE Generation	No	No	No	Yes, Eclipse-based	Yes, Eclipse-based
<b>Utilities</b>					
Interactive Interpreter	Nlgi	No	No	StrategoShell, Eclipse tools	No
Language Workbench	Support for Sublime Text, Vim and other text editors	Custom editor with support tools	Support for Emacs and other text editors	Yes, Eclipse-based	Yes, Eclipse-based
Debugging Tools	Interactive interpreter, or regular Java tooling	Yes, through support tools in the editor	N/A	Yes, through Eclipse support tools	Yes, through Java / Eclipse

Table 6.2.: Summary of the differences between the tools.

## 6. Evaluation

```
grammar simple:abstractsyntax;
nonterminal Stmt with errors;
abstract production while
s::Stmt ::= c::Expr b::Stmt
{
  s.errors <- if isBoolean(c.type) then []
               else [err(locUnknown(), "Expression \""
                    ++ show(100,c.pp) ++
                    "\" must be of type Boolean.\n")];
}
abstract production dowhile
s::Stmt ::= body::Stmt cond::Expr
{
  forwards to while(cond, body);
}
abstract production repeatuntil
s::Stmt ::= body::Stmt cond::Expr
{
  forwards to dowhile(body, not(cond));
}
abstract production not ...
```

Listing 6.9: Type checking and attribute forwarding in Silver.

dimension of *language constructs* and on the dimension of *semantic concerns* of the language implementation. This capability *requires* the framework to support non-trivial modularization capabilities. With the notable exception of Xtext, all the surveyed tools are capable of achieving a feature-oriented componentization of a language, although they are generally focused on *code-reuse* rather than providing a mechanism to specifically implement languages in a feature-oriented way. For instance, in all the tools, to a different extent, it is possible to separate the implementation of the semantics from the definition of the syntax. Every tool makes it possible to provide *libraries* of functions that can be shared among components. The degree of freedom in separating language concerns changes for each tool. For instance, in LISA, *multiple inheritance* can be used to separate syntax definitions from multiple evaluation phases (Sect. 6.1.1), but this obviously imposes a strict parent-child relation between components. Silver's grammars (Sect. 6.1.2), together with its concept of aspect productions (not to be confused with LISA's, which provides a similarly-named concept, with a different meaning) may be used to achieve the same kind of componentization. Silver also supports *attribute forwarding*: with this feature the attributes of a node may be defined in terms of the attributes of another node. This feature bears some similarity to Neverlang's *remapping* feature (Sect. 4.1.2), but it is in fact more powerful, because it supports *rewriting* the tree<sup>3</sup>. In Listing 6.9 we show how type-checking for the do-while loop can be described in terms of the while loop. Notice that the repeat-until loop can be also described

<sup>3</sup>Neverlang is currently adding support to a tree-rewrite DSL, though (Sect. 4.2)

in terms of the do-while loop, where the additional node `not` decorates the condition: this is not really necessary for type checking, but it may be useful in other situations, (e.g., code generation).

Spoofax (Sect. 6.1.3) is designed to separate concerns to the extent that each of the major evaluation DSL phases (*name binding*, *code generation*, etc.) can be implemented using a different DSL.

Considering the taxonomy in Erdweg *et al.* [29], all the surveyed modular tools are able to support language extension, restriction, unification and extension composition. It is worth noticing that Neverlang is the only tool that supports *by design* real language restriction, through slice removal. Nevertheless, this can be achieved in the other cases through extension or, if necessary, through refactoring. Language unification is also possible, because all of the tools are able to compose language specifications; in particular, even though according to [29] Spoofax would not be able to perform semantics unification, we argue that the Stratego language's module system allows to define rules and strategies across different modules<sup>4</sup>, and therefore, such a kind of composition is indeed possible.

On the one hand, one might raise the concern that the finer-grained componentization described in this section, inspired by the Neverlang model, may not be *idiomatic* in each framework. But, on the other hand, even if this were true, it would not disprove that the surveyed tools are powerful enough to achieve these results.

As we already mentioned in Sect. 3.1.4, Mernik [63] have observed that self-extension is a property of a *language*, rather than a property of a language *framework*. For instance, SugarJ [30] is an extension to the Java language on top of Java, SDF and Stratego which supports *syntactic* self-extensibility.

A comparison of the examples in terms of code size is shown in Table 6.1, we used the usual metrics of lines-of-code (LOC) (as found e.g., in [60]) and number of components. In terms of lines of code, size is comparable. The Neverlang implementation may appear bigger because of the way Neverlang language definition introduces **module** and **slice** declarations. With respect to number of productions, number of modules and roles, the size is comparable to the other implementations; moreover, it is worth recalling that Neverlang supports alternate JVM languages for the semantics, in which case line count might drop considerably (e.g., consider the boilerplate needed to iterate over a collection in Java 7 compared to Scala or Java 8: in this implementation we used Java 7). A full comparison of the features of the tools is also shown in Table 6.2.

The conclusion of this experience is that none of these tools really centers around the idea of feature-oriented language implementation, but most of them can be retargeted for this purpose through design patterns. This proves that the model (Sect. 3) is general enough to be supported by other tools. In the next section we will show the particular benefits that a native implementation of this model provides.

---

<sup>4</sup><http://releases.strategoxt.org/strategoxt-manual/unstable/manual/chunk-chapter/rules-and-strategies.html#id3317807>

## 6.2. Extending a Real-World Language: neverlang.js

JavaScript is a dynamic, general-purpose programming language that has been recently gaining wider and wider consideration. In order to evaluate the capabilities of the Neverlang framework, we decided to realize a feature-oriented implementation for this language. In our JavaScript interpreter it is possible to plug and unplug features to realize multi-purpose dialects of the original language. The main goal of this experience was not to compete with state-of-the-art JS interpreter implementations. We chose to implement JavaScript because it is a rather simple programming language, and, lately, there has been a lot of buzz around it. We believe that implementing the JavaScript programming language represents evidence that Neverlang is powerful enough to implement not only toy languages, but also real-world general purpose programming languages.

A breakdown of the ECMAScript 3 coverage is in Fig. 6.2: the titles on the left represent the section of the ECMAScript 3 specification, the percentages represent the passed test cases of the Sputnik test suite<sup>5</sup>. Our implementation covers about the 70% of the specification. This result deserves an explanation. The ECMAScript specification is very large, but the largest part of the spec does not revolve around the syntax and the semantics of the programming language; rather, it concentrates on specifying the behavior of the *built-in libraries* that should be found in a fully-compliant implementation. Implementing these libraries is possible, but it constitutes a time-consuming activity, that we plan to complete in a later phase. Unfortunately, the Sputnik test suite assumes that *all* of the built-in libraries are available, causing some tests to fail. However, our implementation of the *semantics* of the language is complete: the interesting parts of the language are supported (*e.g.*, *closures*, *higher-order functions*, the *prototype chain*, etc.), to the point that many of the built-in libraries may be even implemented *within* the language. Still, with the subset of the built-in libraries that is currently available we were able to run browser-unrelated benchmarks in the Google Octane suite<sup>6</sup> without modifications<sup>7</sup>.

Performance-wise, preliminary tests have shown that we were able to make neverlang.js up to *only one order of magnitude* slower than the Rhino JavaScript implementation<sup>8</sup>. Considering that the implementation's main goal was *modularity* and not *performance*, this result is quite promising. Moreover, we are already trying to address this issue through specific efforts (*e.g.*, Truffle-like optimizations, see Sect. 6.2.1).

A breakdown of the slices that constitute the neverlang.js implementation is presented in Table 6.3.

**Extending JavaScript: a Classroom Experience** Our JavaScript implementation consists of 73 slices (Table 6.3) that correspond roughly to the same number of modules,

---

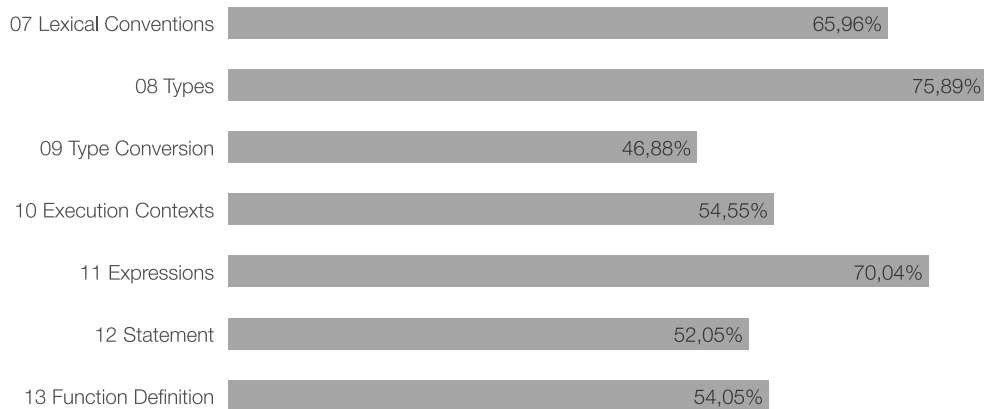
<sup>5</sup><http://test262.ecmascript.org/>

<sup>6</sup><https://developers.google.com/octane/benchmark>

<sup>7</sup>in order to keep the grammar simple, semi-colon insertion was disregarded

<sup>8</sup>preliminary tests, mainly conducted on Octane's `crypto.js`, showed a 30 second execution time for neverlang.js, while Rhino takes about 6 seconds.

## 6.2. Extending a Real-World Language: neverlang.js



**Figure 6.2.:** Coverage of the ECMAScript 3 standard in neverlang.js.

for a total number of 3043 lines of code, plus around 64 Java classes of support code (mostly, related to the supported parts of the built-in objects). Because we intended the language to be used in a short course on modular language implementation, we intentionally kept it simple. For instance, only one *role* (evaluation) has been currently implemented. This short course consisted of only three 4-hours lessons. At the end of the short course, students were handed a full pre-compiled, pre-packaged implementation of the JavaScript interpreter, and were required to implement a different *language extension*. Each extension consisted of a new language construct, with varying levels of complexity. Each student would have provided the implementation of his/her extension as i) a collection of Neverlang source files ii) a pre-compiled *jar* with the extension as a bundle and iii) a collection of test cases for the developed extension. A summary of the implemented extensions can be found in Table 6.4. Each extension has been developed in isolation from the others. Students were provided with a copy of the source code, exclusively for *reference and documentation purposes*. Students were *not* allowed to modify the source code of the reference implementation directly, but rather to realize *new components*. The objective was to see the effectiveness of Neverlang as a tool to develop separate language extensions. Grading consisted in first verifying that the provided source code was actually compiling. Then, an automated script loaded each student-provided jar file, introducing the new components in the base interpreter.

As seen in Sect. 4.4.1, in Neverlang a language implementation is a JVM object instance. The public method `importSlice(sliceName)` can be invoked at *any time* during the life-time of a language, making it possible to introduce and substitute slices at runtime. The students' extensions were tested using this Neverlang feature. In order to verify the correct execution, each language extension was first introduced independently from the other, and tested in isolation; then we proceeded to verify the interactions between the extensions by testing all the possible  $2^{14}$  combinations of such

## 6. Evaluation

Bundle	Slices	LOC	Rules	Bundle	Slices	LOC	Rules
<b>Core</b>				<b>Statements</b>			
Language core	11	277	24	Block Statement	1	32	4
<b>Expressions</b>				<b>Cflow</b>			
Arithmetic	3	128	9	If Statement	1	45	3
Boolean	3	92	5	Switch Statement	1	102	12
Relational	2	137	10	<i>(Loop Statements)</i>	1	19	1
Conditional	1	32	2	While Statement	1	50	2
Bitwise	5	216	17	For loop	1	57	7
Typing (typeof, instanceof)	2	65	2	For-each loop	1	113	10
Function call	2	113	9	<i>(NoIn expressions integration)</i>	11	305	26
Construct call	1	56	3	Interrupt: break	1	22	2
<b>Types</b>				Interrupt: continue	1	22	2
String	1	21	2	Interrupt: return	1	30	3
Number	1	24	3	Interrupt: throwing + handling	2	122	8
Boolean	1	23	3	<b>Variables</b>			
RegExp	1	23	2	Variable assignment	5	226	21
Object	4	189	13	Variable resolution	1	24	2
Array	3	131	9	<b>Endemic Slices</b>			
Function (definition)	2	100	11	Symbol Table		230	
This resolution	1	17	1				

Italicized features depend on other features: *loop statements* require at least one actual loop implementation (e.g., `while`, `for`, etc.), *No-In expressions* are part of the ECMAScript spec and depend on the definition of *for-each*.

<b>Total Slices</b>	<b>73</b>
<b>Total LOC</b>	<b>3043</b>
<b>Total Rules</b>	<b>228</b>

**Table 6.3.:** Summary of *neverlang.js* by feature bundle.

extensions. Because of the didactic nature of the experiment, only a few extensions actually conflicted: in particular the students that implemented tuples and pattern matching chose a similar syntax for the same feature, causing the parser generator to generate an error if such extensions were introduced at the same time.

The relevance of this experiment is to show that Neverlang's rendition of our feature-based model of language implementation emphasizes its good properties, when they are brought to their natural extreme:

1. language components can be developed separately, by different programmers, thereby allowing multiple teams to realize new features for a language implementation in parallel
2. features can be shipped as pre-compiled components
3. pre-compiled components can be composed onto the core language implementation at any time, possibly at runtime
4. independently-developed features can be tested together without touching the core language implementation in an automated fashion

Extension Name	LOCs
Function Type Annotations	225
Catch Guards	80
Class-Based Single Inheritance	314
Dictionary Comprehension	79
Destructuring Assignment	73
Tuple Literal	91
List concat operator	91
Lambda Expressions	76
Named Arguments in Functions	78
List Sum Operator (Vector Sum)	41
Pipe Forward Operator	92
Immutable References	31
List Comprehension	81
Syntax for Pattern Matching	191

Table 6.4.: List of JavaScript Extensions.

### 6.2.1. Runtime Evolution for Dynamic Optimization

In Sect. 4.2 we saw that slices and roles are pre-compiled components that can be *deployed* and *undeployed* at runtime. One simple consequence is that, as we saw in the previous paragraph, testing of extra-features for a base language implementation is as simple as invoking the method of a class. The `importSlice()` method (Sect. 4.4) makes it possible to load new slices, and makes the feature immediately available to the complete language implementation. However, we are currently investigating how to exploit this feature to bring it one step further. Inspired by the Truffle [111, 50] runtime system, we saw an occasion to exploit this capability to perform modular *runtime-optimizations* on our JavaScript interpreter. The Truffle JavaScript implementation, among other things, optimizes code paths by rewriting tree nodes using specialized versions. *Guards* are installed on the code bodies that implement the semantics of AST-based hand-written interpreters, and the rewriting occurs when a guard fires. It has been shown [50] that a Truffle-based JavaScript interpreter implementation is close performance-wise to highly-optimized interpreters such as Google's V8. The Truffle project uses Java and Java annotations to achieve this impressive results. We are currently trying to reproduce similar techniques in Neverlang by introducing *tree rewriting* capabilities and *guards*. In Sect. 4.4.2 we described how actions are resolved. The function  $m$  maps the triplet  $(p, r, i)$ , where  $p$  is a production,  $r$  a role, and  $i$  a numeric index, onto *one* slice  $S_{r,i}$ . The key idea was to allow  $m$  to return a *set* of semantic actions; then the runtime system *chooses* which rules should be executed, depending on the guards. Although the work is still in its infancy, the results are promising: in an initial implementation of this technique, we measured that avoiding boxing of primitive values through rewriting resulted in a (up to) 20× speedup. Further results on this matter will be reported in a separate work.

### 6.3. The DESK Language

DESK is a *simple desk calculation language* described in [74] to show an example of an *absolutely non-circular attribute grammar* (Chapter 2). Obviously, we are aware that implementing the DESK language *does not* constitute proof that Neverlang is able to handle *any* non-circular attribute grammar; nonetheless, we believe that showing that Neverlang is able to implement DESK constitutes at least *evidence* that the framework is able to handle non-trivial cases (see Chapter 2). Full source code is available at

<http://neverlang.di.unimi.it/vacchi/examples.tgz>

In DESK, programs are of the form

**PRINT** { *expression* } **WHERE** { *definitions* }

where { *expression* } is an arithmetic expressions and defined constants, and { *definitions* } is a sequence of constant definitions of the form

{ *constant-name* } = { *number* }

Each constant occurring in { *expression* } must be defined in { *definitions* } and, for each constant, only one definition may be given. A valid DESK program may be

**PRINT**  $x + y + 1$  **WHERE**  $x = 1, y = 2$

The original DESK definition only includes addition as a valid expression; nevertheless, the DESK language includes many central features of a real programming language:

- declaration of named entities (constants)
- use of declared entities
- conditions on the declaration and use of such entities
  - an entity cannot be redeclared
  - only declared entities can be referenced by name

In Paakki's work, DESK is compiled into an assembly code for a simple one-register machine:

LOAD	$n$	load value $n$ into the register
ADD	$n$	add value $n$ to the register
PRINT	$\emptyset$	prints the contents the register
HALT	$\emptyset$	halts the machine

The execution of a valid DESK program evaluates the expression and prints its value. For instance, the code generated for the previous DESK program would be:

LOAD	1	( $x$ )
ADD	2	( $y$ )
ADD	1	
PRINT	$\emptyset$	
HALT	$\emptyset$	



```

// desk.Program                                // ConstDef
Program ← "PRINT" Expression ConstPart;      ConstDefList ← ConstDefList ", "
// desk.Expression                               ConstDef;
Expression ← Expression "+" Factor;          ConstDefList ← ConstDef;
Expression ← Factor;                          ConstDef ← ConstName "=" Number;
//desk.Factor                                    //desk.Tokens (not shown in Pakki)
Factor ← ConstName;                           ConstName ← /[a-zA-Z]+/;
Factor ← Number;                               Number ← /[0-9]+/;
// desk.ConstPart
ConstPart ← "";
ConstPart ← "WHERE" ConstDefList;

```

Listing 6.10: DESK grammar.

```

language desk.Lang {
  slices desk.Program  desk.Expression  desk.ConstPart
          desk.ConstDef  desk.Factor     desk.Tokens
  roles
    syntax                // parse input
    < collect-constants // map constants into values (numbers ↦ ints)
    <+ evaluation        // prepare envs and evaluate the expression with env
    < code-gen           // generate and output code
}

```

Listing 6.11: Neverlang descriptor for the DESK language.

The attribute grammar in [74] has been converted into a Neverlang compiler. In Listing 6.10 we show the DESK grammar with respect to the way we have defined language components. In our implementation we chose to define 5 modules, plus the one for defining lexer tokens for constants, `ConstName` and numbers, `Number`; Paakki makes no distinction between evaluation phases; in Neverlang it is easier to reason in terms of **roles**. Our implementation (Listing 6.11) defines three roles: `collect-constants`, `evaluation`, `code-gen`. The first role has *post-order* semantics (Chapter 4) because it maps lexer tokens into their corresponding values (*e.g.*, it maps into an int value the token for the token for `Number`): the relevant actions are attached to the leaves of the syntax tree, thus it makes sense to evaluate these first. The evaluation role performs the majority of the work. In Listing 6.12 the `Program` module is shown. This module contains the starting symbol of the grammar, as defined in [74]. The evaluation role uses the *semi-automated* evaluation strategy (Sect. 4.1.3), thus, the developer is given full control on how and when the child nodes should be evaluated. In particular, in the DESK language, the visit should start from the `ConstPart` nonterminal, and *then* proceed to the `Expression`. Neverlang is able to do this, because it is possible to **eval** the second left-hand side nonterminal *before* the first left-hand side nonterminal, using the command **eval** command. In this case, the label P was assigned to the production in the **reference syntax** section. So, we can write:

## 6. Evaluation

```
module desk.Program {
  reference syntax {
    P: Program ← "PRINT" Expression ConstPart;
  }
  role (evaluation) {
    P: .{
      eval $P[2];
      // pull ConstPart.envs and push it into Expression.envi
      // notice that attributes "stick" between phases
      $P[1].envi = $P[2].envs;
      // print out the environment (not in Pakki)
      eval $P[1];
      System.out.println($P[1].envi);
    }.
  }
  role (code-gen) {
    P: .{
      Boolean constPartOk = $P[2].ok;
      String code = $P[1].code;
      code += constPartOk? "PRINT 0\nHALT 0\n" : "HALT 0\n";
      $P.code = code;
      // print out the generated code
      System.out.println(code);
    }.
  }
}
```

Listing 6.12: Program.

```
eval $p[2];
```

to proceed to evaluate the ConstPart nonterminal. Once control is returned to this semantic action (that is, the recursive visit of the ConstPart subtree has terminated), it is possible to proceed to the Expression subtree. It is possible to pass down a value (a *inherited* attribute) by assigning it before **eval** is invoked. Then, we can proceed to evaluate Expression using:

```
eval $P[1];
```

Finally, the code-gen role generates the assembly code using the attributes that were computed during the execution of the evaluation role, and the code attribute in Expression and its descendants, computed during the code-gen phase: code generation is a role that is, again, a good candidate for simple *post-order* visit.

Listing 6.13 shows the full DESK source code, excluding the less interesting Tokens module (about 30 lines of code), for the sole sake of space (of course, this is available in the online version of the code).

```

module desk.Expression {
  reference syntax {
    e1: Expression ← Expression "+" Factor;
    e2: Expression ← Factor;
  }
  role (evaluation) {
    e1:.{
      // Expression_2.envi =
      // Expression_1.envi
      $e1[1].envi = $e1.envi;
      eval $e1[1];
      // Factor.envi = Expression_1.envi
      $e1[2].envi = $e1.envi;
      eval $e1[2];
    }.
    e2:.{
      // Factor
      $e2[1].envi = $e2.envi;
      eval $e2[1];
    }.
  }
  role (code-gen) {
    e1:.{
      Boolean factorOk = $e1[2].ok;
      String code = $e1[1].code;
      code += factorOk?
        "ADD " + $e1[2].value : "HALT 0";
      $e1.code = code + "\n";
    }.
    e2:.{
      Boolean factorOk = $e2[1].ok;
      String code = (factorOk?
        "LOAD " + $e2[1].value
        : "HALT 0") + "\n";
      $e2.code = code;
    }.
  }
}

module desk.Factor {
  reference syntax {
    F: Factor ← ConstName;
    NF: Factor ← Number;
  }
  role (evaluation) {
    F: @{
      String name = $F[1].name;
      java.util.Map<String,Integer> e = $F.envi;
      $F.ok = e.containsKey(name);
      $F.value = e.get(name);
    }.
    NF: @{
      $NF.ok = true;
      $NF.value = $NF[1].value;
    }.
  }
}

module desk.ConstPart {
  reference syntax {
    Empty:ConstPart ← "";
    Const:ConstPart ← "WHERE" ConstDefList;
  }
  role(evaluation) {
    Empty : .{
      $Empty.ok = true;
      $Empty.envs = new java.util
        .HashMap<String,Integer>();
    }.
    Const: @{
      $Const.ok = $Const[1].ok;
      $Const.envs = $Const[1].envs;
    }.
  }
}

module desk.ConstDef {
  reference syntax {
    CDL1: ConstDefList ← ConstDefList ", "
      ConstDef;
    CDL2: ConstDefList ← ConstDef;
    CD: ConstDef ← ConstName "=" Number;
  }
  role(evaluation) {
    // In idiomatic Neverlang you call
    // AttributeList.collectFrom();
    CDL1: .{
      eval $CDL1[2]; // eval ConstDef
      String name = $CDL1[2].name;
      Integer value = $CDL1[2].value;
      // eval ConstDefList_2
      eval $CDL1[1];
      java.util.Map<String, Integer> e =
        $CDL1[1].envs;
      Boolean cd1Ok = $CDL1[1].ok;
      $CDL1.ok = cd1Ok && !e.containsKey(name);
      e.put(name, value);
      // ConstDefList_1.envs =
      // ConstDefList_2+(name, value)
      $CDL1.envs = e;
    }.
    CDL2: .{
      eval $CDL2[1]; // eval ConstDef
      $CDL2.ok = true;
      String name = $CDL2[1].name;
      Integer value = $CDL2[1].value;
      java.util.Map<String, Integer> e =
        new java.util.HashMap<>();
      e.put(name, value);
      $CDL2.envs = e;
    }.
    CD: @{
      $CD.name = $CD[1].name;
      $CD.value = $CD[2].value;
    }.
  }
}

```

Listing 6.13: DESK: Expression, Factor, ConstPart and ConstDef.

## 6. Evaluation

The size of the full DESK implementation is less than 180 lines of code. The apparent verbosity is due to our choice of employing built-in Java data structures to keep the code base small, and without external dependencies. For instance, a `java.util.Map` is used here for the *environment*. But `java.util.Map` is a *stateful* data structure. Nevertheless, because of the way the language is defined (the environment is first filled, and then read), this does not introduce unintended side-effects. Of course, it is always possible to rely on third-party libraries: for instance Google’s Guava library<sup>9</sup> provides `ImmutableMap`. Another point that is worth mentioning is that the DESK implementation in Listing 6.13 is a pedantic translation of Paakki’s original. However, in Neverlang, at least two things are usually done in a different way:

1. the *environment* would be probably implemented using a stateful *endemic slice* (Sect. 4.1.2)
2. collecting attributes on the `ConstDef` nonterminal would be done through the library function `AttributeList.collectFrom()` (Table 4.1, p.40).

If the DESK language were rewritten using these techniques, the code base would result even tighter.

**Observations** The implementation of the DESK language in Neverlang certainly *does not* represent a formal proof for Neverlang’s expressive power. However, we believe that it constitutes strong *evidence* that Neverlang should be practical enough to implement *non-trivial* attribute grammars. If anything, it shows that Neverlang is more powerful than simpler tools such as Yacc and ANTLR, which are limited to L-attributed or S-attributed grammars [2, 76].

This power comes in some cases at the cost of being explicit about how the visit of the tree is conducted (using the `eval` command) and about the way attributes are partitioned into roles. Tools that implement proper attribute grammars do not require attribute evaluation to be triggered explicitly; in the most simplistic case, attribute evaluation is triggered at their use-site. For instance, the rule `A.val = B.val + C.val` for a production  $A \rightarrow BC$  would cause the evaluation of attributes `B.val`, `C.val` which would, in turn, cause the evaluation of any other attribute they may be defined in term of. In fact, one strategy to implement an attribute grammar is to map attributes onto *functions*; attribute grammar frameworks may then employ caching and memoization techniques (Chapter 2) to avoid recomputing attributes more than once, when they produce the same results. However, memoization may be hindered if the language framework admits *impure* computations. This is sometimes unavoidable, for instance when *I/O* has to be performed. Neverlang’s approach is more *explicit*, in that (unless the visit is *post-order*—Sect. 4) it requires users to explicitly signal where attributes are being evaluated.

On the one hand, being explicit may feel a little inconvenient, because it places the burden of choice on the end users. In Neverlang this is addressed by providing syntactic sugar (Sect. 4) to explicitly require attribute evaluation, while retaining conciseness. On

---

<sup>9</sup><https://github.com/google/guava>

the other hand, this gives users *more control* over what is being evaluated: attributes may be explicitly re-evaluated if the programmer knows that the value of an attribute should have changed; likewise, the programmer may choose not to do so when a pre-computed attribute retains a valid value. This may be a plus for developers that need this kind of finer-grained control. For instance, Lex Spoon, co-author of the book *Programming in Scala* [69] with Martin Odersky, has observed<sup>10</sup> that in attribute grammar systems performance may be harder to predict.

The biggest downside is that components coming from different sources may not play well together because they expect different evaluation orders. In fact, delegating the computation of the evaluation order to automatic machinery (as it usually happens with more traditional attribute grammar evaluation systems) would relieve the developers from needing to think of this aspect in the first place, and, in the end simplify the combination of components coming from different authors.

All in all, choosing one strategy over the other is a matter of *trade-offs*. Neverlang's choice was to trade a bit of convenience in favor of giving users control; in AG evaluation systems users are relieved from the burden of choice, but, on the other hand, they have less power over the way the language is evaluated.

## 6.4. Tracking Dependencies Through Variability Management

We already discussed the way language components represent *features* of a language. A language implementation is a collection of language components, but, as we saw in Chapter 3, an arbitrary collection of language components is not necessarily a language implementation. Each language component in itself represents a *feature* of a language (Chapter 3), which, by itself, does not constitute a self-contained language definition. This is why each component may have *dependencies*. Dependency-tracking is a concern that is not directly related to the composition model of a language framework, but it is nonetheless induced by the way the language framework conceives *components*. In our model, we defined *provided* and *required* features: these dependencies may be represented in the syntax (*placeholders* or *nonterminals*) or in the semantics (*properties* or *attributes*). In a language implementation we want that every dependency, of both these kinds, to be *satisfied*, otherwise we would end up with an incomplete, inconsistent language implementation. These dependencies are usually not tracked *automatically*: the framework may warn the user that a dependency has not been satisfied and raise a compile-time or run-time error. However, the framework usually *does not* provide users with suggestions about how these missing dependencies may be satisfied to complete the language implementation.

In previous work [98, 99], we have researched a way to *mine* data from pre-compiled language components that not only allowed to represent the relationships between components, but also could be employed to provide users with a readable representation of these dependencies, thereby allowing even end users to compose a language implementation from an *arbitrary selection* of pre-compiled language components. Language

<sup>10</sup><http://blog.lexspoon.org/2011/04/practical-challenges-for-attribute.html>

## 6. Evaluation

Bundle	Language Variants							
	Core	Calc.JS	Imperative Only (no obj)	Functional.JS	Simple string manipulation	Logical calculator	Single command	No Functions
Language core	✓		✓	✓	✓	✓	✓	✓
<b>Expressions</b>								
Arithmetic	✓		✓	✓	✓		✓	✓
Boolean			✓	✓		✓	✓	✓
Relational			✓	✓			✓	✓
Conditional			✓	✓		✓	✓	✓
Bitwise			✓	✓			✓	✓
Typing (typeof, instanceof)							✓	✓
Function call				✓	✓*		✓*	
Construct call								✓**
<b>Types</b>								
String			✓	✓	✓		✓	✓
Number	✓		✓	✓			✓	✓
Boolean			✓	✓		✓	✓	✓
RegExp				✓	✓		✓	✓
Object							✓	✓
Array				✓			✓	✓
Function (definition)				✓				
This resolution								✓
<b>Statements</b>								
Block Statement			✓					✓
<b>Cflow</b>								
If Statement			✓				✓	✓
Switch Statement			✓					✓
(Loop Statements)			✓				✓	✓
While Statement			✓				✓	✓
For loop			✓				✓	✓
For-each loop							✓	✓
(NoIn expressions integration)							✓	✓
Interrupt: break			✓					✓
Interrupt: continue			✓					✓
Interrupt: return				✓				
Exception throwing + handling			✓				✓	✓
<b>Variables</b>								
Variable resolution	✓		✓	✓	✓	✓	✓	✓
Variable assignment	✓		✓			✓		✓
<b>Endemic Slices</b>								
Symbol Table	✓		✓	✓	✓	✓	✓	✓

\* Allow Only Built-in Functions

\*\* Allow Only Built-in Constructors

**Table 6.5.:** Summary of a few possible *neverlang.js* language variants.

components could be grouped by *language families*: by collecting all the components that belong to a particular domain, users would be allowed to pick the features of a domain-specific language, realizing a *variant* that is member of that family. For instance, it would be possible to represent a family of *state machine* languages in terms of the possible feature that a state machine language could include. End users may select the features they want from a *variability model* [77, 49, 21] and generate the language implementation *automatically*. Mixing domains would be also allowed: for instance state machine features may be combined with an action language, as described in Chapter 5.

Part of the information is inferred directly from the dependency graphs that can be constructed from internal properties of the language components (Chapter 3). We have then tried to *infer* automatically a variability model by further mining information from our language components. For instance, language components may be *tagged* by

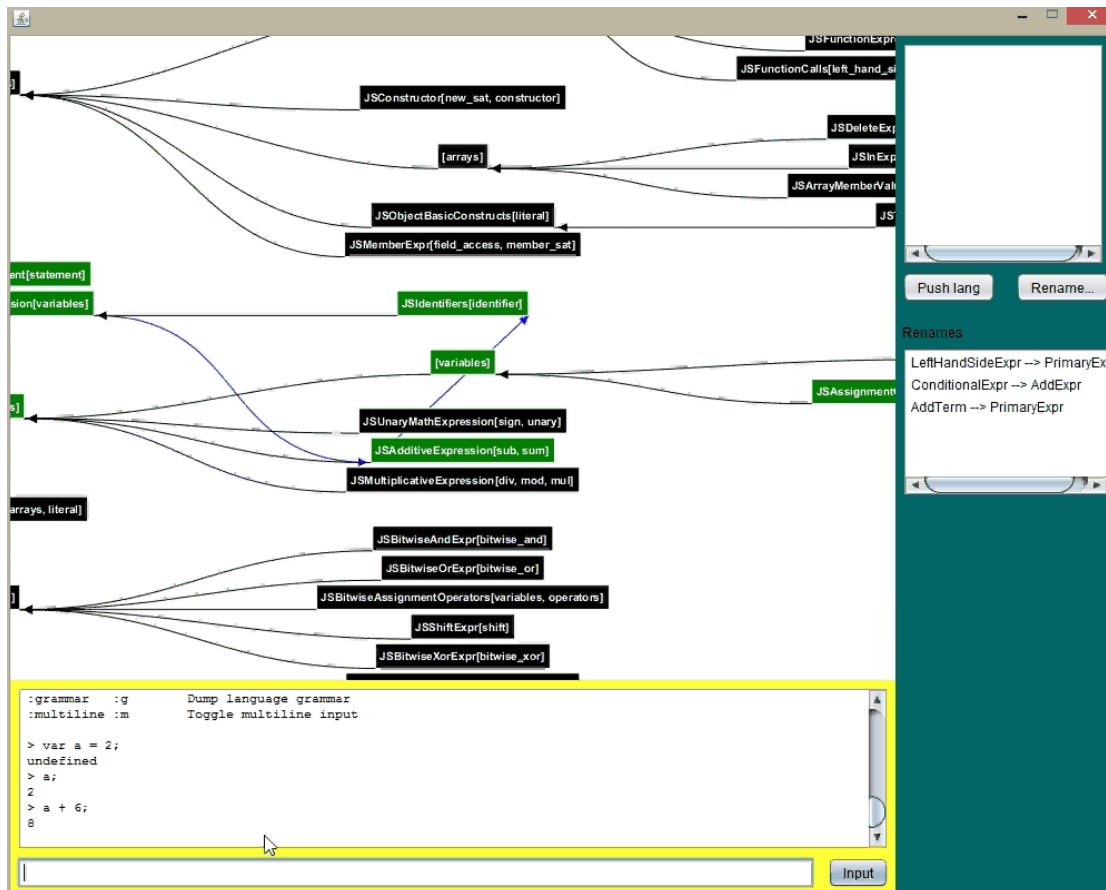


Figure 6.3.: Selecting the neverlang.js variants through an interactive GUI.

language developers with *keywords*. The while loop implementation may be tagged with the keywords `loop`, `statement`; transition with guards may be tagged with the keywords `transition`, `guard`, `action`, etc. These tags may be bundled with the language component. In the Neverlang case, these would be stored as fields of the objects that represent slices and modules (see Chapter 4). This metadata can be later extracted for further processing. In the Neverlang implementation this metadata can be extracted from the pre-compiled components through the framework’s APIs (Sect. 4.4). Tags are then fed into a *hierarchical clustering algorithm*. By manipulating the *dendrogram* resulting from the clustering procedure, we are then able to present the features and their relationships through a tree-like structure that is a snapshot of the given set of language components. End users are then able to pick features by selecting components of this tree (the variability model), and the engine automatically resolves the dependencies and combines the components into the language implementation.

The experiment has been carried out using different languages. One experiment mined data from a *family of state machine languages*, similar to the one described in Chapter 5. This language family included different kind of states, and extra transition

## 6. *Evaluation*

types. Other experiments involved a simple imperative language. A similar experiment is now being conducted on `neverlang.js` (Sect. 6.2). Table 6.5 shows possible language variants that can be constructed using the slices of the full `neverlang.js` interpreter. The features are grouped by bundles (on the left); the columns identify a language variant. In a working prototype, it is now possible to select features of `neverlang.js` and test live the resulting language variant through the embedded `nlgj` console (Fig 6.3). An excerpt from the experience conducted in [99, 98] can be found in Appendix C.



# 7

## Related Work

Chapter 6 gave a brief overview of three of the most relevant modular language implementation frameworks, with respect to a running example. These frameworks, however, provide further functionalities to simplify language implementation.

LISA's extension AspectLISA [80] supports AOP-like constructs to hook into productions through pattern-matching and inject attributes at multiple sites at once. The template construct makes it possible to perform a sort of macro-expansion of repetitive rules (*e.g.*, the bucket brigade pattern, to collect lists of attributes, which Neverlang implements through library functions 4). However, in LISA there is no way to separate attribute definition from grammar definitions, and, in particular, it is not possible to define an *abstract syntax* to code against, creating a tighter coupling between syntax definitions and language semantics, which may hinder reuse of semantics over different language implementations. It is possible to rely on external libraries to create an internal representation, as we did in the state machine example, but then this internal representation is completely unaware of the surrounding framework. LISA's main target language is Java and it uses Java for attribute definition, although it does not currently support generics; this is clearly only a limit with the way the tool parses input files, which could be extended in the future. LISA generates Java source files which are in turn compiled into class files; the generated files are mostly meant to be run standalone, but it is possible to invoke user-provided code; in this case, there is no limit on the Java language version users may write support code into (or the usage of the Java language itself, for that matter: class files may be written using any JVM language); in the implementation of the example, it was possible to reuse many of the support classes that were developed for the Neverlang example. Unfortunately, there is no support for separate compilations, nor does the tool support language extension to be performed from pre-compiled binaries; the language input files have to be provided as source code. LISA supports the most extensive number of techniques to parse and

## 7. Related Work

evaluate attribute grammars, with several choices on the kind of parser generator to use (e.g., LL, LR, LALR, etc.) and the evaluation strategy for the attribute grammar (among the others, Lencic Tree Walk Evaluator, Katayama Evaluator, L-Attributed Evaluator, Visit Pattern Evaluator, etc.).

Silver [54, 86] supports a way to define *abstract productions* that programmers may employ to define new data structures; in fact, the way Silver defines and uses productions make them feel like *cases* of *algebraic data structures*; to the point that Silver supports *pattern matching* on productions: all features that reveal the authors' intentions to make Silver feel as much as possible as a functional programming language, while keeping it close to the *grammar* metaphor. Even the concept of *attribute* can be roughly equated to that of *function* and *let-binding*. Silver supports both *higher-order attributes* and *reference attributes*. Higher-order attributes are trees that have not yet been given any inherited attributes and thus their synthesized attributes are not yet defined; such trees can be constructed and passed around. Reference attributes are references to another tree that is already decorated and attributes can be read off of it. Silver's most important feature to support language extension is, however, *attribute forwarding*. With forwarding it is possible to describe the attributes of a node in terms of the attributes of another, known subtree. In Sect. 6.1.5, for instance, we have shown how to describe the *do-while* and *repeat-until* loop constructs in terms of the *while* loop. Silver supports *verifiable* composition of modules, raising errors if a language extension is not *well-defined* [54]. It also includes *aspect productions*, which, in spite of the name, indicate a different feature from LISA's (see Section 6.1.2). The generated parser uses a variant of LALR with context-aware scanning [102], and it implements an algorithm for *verifiable* composition of deterministic parsers [86]. These features together make it easier to compose LALR grammars and give stricter guarantees on the generation of a deterministic parser. Silver source files are compiled down to Java and it is possible to reuse Java libraries, but while the interfacing between Neverlang and Java/JVM code is nearly seamless, the Silver system is meant to be self-contained; thus, although it *is* possible to interface with Java code, the endorsed way to define extra support code is to use Silver itself. In fact, the choice of a purely-functional programming language with a peculiar syntax could be an obstacle to bring Silver within the reach of a broader audience.

Spoofox [56] language workbench uses Stratego [12] as its main programming language, but provides different DSLs to deal with different *concerns* [103] of a programming language. This rather radical approach on the one hand make it possible for the Spoofox workbench to generate not only a compiler for the language that is being implemented, but also to use the DSLs as a means to plug language analysis processes directly in the generated IDE plugin. The usage of these DSLs is not mandatory, but highly-recommended; otherwise using Stratego directly would often require programmers to resort to design patterns to obtain the same results. After all, elevating design patterns and APIs to first-class constructs of a domain-oriented language is really one of the primary reasons for implementing a DSL [64, 36, 38]. The danger with this approach is to put too much cognitive burden on the language developer, who is then required to understand *several languages* instead of one. On the other hand, the Stratego language is a dynamically-typed declarative domain-specific language for *term rewriting*,

with a unique syntax. The JVM implementation compiles the DSLs into Stratego source files and Stratego files into an internal high-level format interpreted by the Stratego/J execution engine<sup>1</sup>; interoperability with Java code is possible, but it is not within the main objectives of the project. Separate compilations are unfortunately not supported yet [31].

The JastAdd [44] compiler construction system follows a completely different approach. It is in fact an attribute grammar system, but it has chosen to completely embrace the Java programming language. JastAdd programmers define a grammar and the attributes that should decorate such grammar, and then the system scaffolds pre-built AST Java classes that programmers may then complete with the implementation. Aspects can be used to statically inject semantics into attributes. Attributes are implemented as Java methods, and they support parameters. JastAdd also includes *reference attributes*. Aspects can be used to separate concerns such as *evaluation phases*. Aspects can be factorized in such a way that it is possible to define pre-compiled *language components*, but JastAdd is a code-generating tool, thus it is not possible to further extend components without editing the original source code.

It is also worth mentioning MontiCore [59] a framework for language composition and extension that provides grammar inheritance and rewriting mechanisms additionally to modularization features. MontiCore uses a combined grammar format for concrete and abstract syntax and it supports *grammar inheritance* and *rule inheritance*. In the case of *grammar inheritance*, similarly to LISA, all the rules of a parent grammar are inherited; *overriding* of rules is also possible. UML-like associations describe semantics.

We also want to mention JetBrains' Meta-Programming System (MPS) [105], a language workbench for DSL implementation; MPS stands out because, unlike the others, it is a *projectional editor*, which means users are actually *editing an AST* through context menus, autocompletion and keyboard shortcuts instead of just typing in text. The authors claim that, after a training period, users become much faster at writing code. The system supports language modularization and inheritance-based code reuse.

Lightweight Modular Staging (LMS) [82] uses Scala's embedded DSL idiom to implement compilers. LMS indeed provides the means to modularize the syntax of a program to generate code, and therefore it can be used to implement the work that we described. Moreover, syntactic composition in this case would be easier than in Silver because in LMS merging language components only consists in using APIs coming from different libraries instead of merging parse tables. The downside is that syntax of programs is limited to the constraints imposed by Scala's compiler.

Xtext [27] is a framework and language workbench for model-based development of DSLs that tightly integrates with EMF [92]. The framework makes possible to reuse existing grammars and existing meta-models to implement other languages. It uses ANTLR to generate the parser, and it may compile to source code or bytecode. It supports single inheritance between grammars; therefore, although it is not possible to just "plug" a feature inside another language, it is possible to extend an existent language with the feature. The downside is that the reuse of the feature is limited to

---

<sup>1</sup><https://strategoxt.org/Stratego/StrategoJ>

## 7. Related Work

inheritors of the base grammar. Xsemantics [9] is a DSL that can be used in combination with Xtext to formally define and verify the semantics of compilation phases such as type-checking. We want also to cite EMFText [47], another EMF-based tool that supports modular language implementation using syntax imports.

Computer language implementation can be also achieved through *self-extensible* programming languages, where *self-extension* is intended here with the meaning in [29]. For instance the SugarJ language [30] is an extension to Java that supports syntactic extensions, and it is implemented using SDF and Stratego. Compile-time meta-programming systems such as Template Haskell [89] and Converge [96] use quasi-quoting to denote constructs that must be evaluated at compile-time. DSLs can be embedded by directly manipulating the AST of the host language. The Wyvern programming language [72] uses *type-specific languages* to allow language embedding (parsing and elaboration) in a type-safe way. An in-depth comparison of language embedding through macro-systems and source-to-source translators can be found in [96].

### 7.1. Extensible Parser Generators

The DEXTER parser generator (Sect. 4.4.3) is an LR parser generator that we developed for Neverlang. The DEXTER generator generates and maintains the LR(o) goto-graph in memory and updates it on-the-fly as new rules are added, or old rules are removed from the grammar.

Several authors have dealt with the problem of evolving the parser of a language from different perspectives. The problem of performing syntax extensions is known to be nontrivial. Traditionally, the problem of parsing a context-free language is a matter of choosing between the LL and the LR family [57], but these techniques alone do not deal with the problem of evolution and extension.

Modern programming languages such as Scala [66] contributed to a new wave of interest in *parser combinator* (e.g., [51]) libraries. Parser combinators are higher order functions that can be composed together to form a parser for a complete language. Parser combinators, PEGs [34], and in general any technique based on recursive descent (e.g., [108, 81]) are known not to be trouble-free. For instance, a naïve implementation of an ambiguous context-free grammar requires exponential time and space. Other authors investigated alternative parsing algorithms, to allow more flexibility in grammar definitions. The metafront tool [11] uses a novel parsing algorithm called *specificity parsing*; the main problem with this algorithm is that the class of recognized languages is not fully characterized. On the other hand, Earley's parsing algorithm [26] is well-known, and it is able to handle any (possibly ambiguous) context-free grammar, although at the cost of a non-linear computational complexity [42]. The dynamic nature of the algorithm makes possible to implement *reflective grammars* [91], that is grammars that allow language extension at parse time. The algorithm is also being employed in the SPARK toolkit for DSLs generation in Python [5], but the author reports concerns in term of speed. A proposed solution to this problem (e.g., [6]) involves a pre-computation phase similar to those for LL and LR parsers, but the cost

is losing the dynamic properties that make evolution easier.

LR parsers are known to be an efficient family of bottom-up parsers that is guaranteed to run in linear time for any deterministic context-free language. The main arguments against this family are that LR parsers are not easy to write by hand, and that many notable subclasses of LR, such as LALR, are in general not closed under composition [86]. The first problem can be addressed using an LR parser generator, such as YACC, that takes (E)BNF grammars as their input. There are also parsers in the LR family that are known to be able to parse *any* context-free grammar, such as GLR. Scannerless Generalized LR parsers (GLR) [95] have been showed to be closed under composition, they are able to parse any context-free grammar, and are generally more efficient than Earley for programming languages that are close to LR [13]. However, for nondeterministic grammars, they may generate multiple different parse trees, and therefore execute different actions for the same input text. Another notable class of the LR family, LALR(1), has been showed to be practical and efficient with respect both to parsing and to composition, by introducing *context-aware scanning* [102]. Both LALR and GLR parsers are based off the same formalism, that is, LR(o) goto-graphs. The conclusion is that studying a method to extend and restrict LR(o) goto-graph leads to a working foundation for reasoning about the other classes. [13] describe an algorithm to compose together different grammar portions and obtain an LR(o) goto-graph. First, the input grammars are translated into  $\epsilon$ -NFAs, then they are compose together, and then the result is converted into the LR(o) goto-graph. The algorithm has been implemented in MetaBorg and in the OCaml project *dyppen* [73]. This intermediate representation, however, can be avoided, by applying the updating procedure directly on the LR(o) goto-graph. Early literature on incremental extension of LR(o) goto-graphs [46, 48] showed promising results. However, these work did not really provide formal proofs of their results, favoring a largely empirical explanation of the software tools that the authors implemented. In [19] we filled the void by completing these works with a formal proof of the existence of a relation between extended and restricted LR(o) goto-graphs using the technique of reduction over path lengths. An excerpt of the main matter of this paper can be found in Appendix B.

## 7.2. Variability Modeling of Language Families

In Sect. 6.4 we discussed how variability management can be employed to represent and resolve dependencies between language components in Neverlang. The variability model is automatically inferred from *tags* on the slices, and refined through information from the structural dependencies between components (Chapter 3 and Appendix A). A detailed discussion of the experience can be found in [99, 98]. An excerpt from [98] can be found in Appendix C.

Many authors have addressed the problem of recovering a feature model from various kinds of artifacts. She *et al.* [88] showed how to reverse engineer a feature model starting from *feature descriptions* (written in natural language) and static analysis of source code. Davril *et al.* [22] presented a fully automated approach for constructing

## 7. Related Work

feature models from publicly available product descriptions (e.g., as found in SoftPedia and CNET). Alves *et al.* [3] and Niu *et al.* [67] use clustering techniques to infer a tree structure. Ferrari *et al.* [33] considered natural language documents. Weston *et al.* [109] extract feature models from the requirements description in natural language.

In [99] the domain knowledge necessary to build the variability model was provided by a *domain expert*, while in [98] we used a *semantic network*. In general *tags* can be employed to describe a feature, although additional knowledge may be required to infer further constraints (such as conflicting features). Some work has applied variability management to language implementation. Although we used Neverlang, other modular language implementation frameworks can be employed to implement a similar kind of approach. Cengarle *et al.* [20] use MontiCore [59] to describe variations of a base language. Haugen *et al.* [43] have used CVL to model possible DSL variations. White *et al.* [110] use feature modeling to improve reusability of features among a language family. Liebig *et al.* [60] have used Spoofox to generate a family of languages from the *Mobl* DSL. The authors do not start from a set of pre-defined components, but rather they componentize an already existing language and develop the variability model to support it. Therefore, relations between language components are imposed by the developers as they implement them. In our approach we have discovered the relations between the components using information that we extract directly from the implemented language components. Thus, our objectives were quite different: we wanted to help users finding implicit or explicit relations between *existing* components. The result makes it possible for end-users to configure their own DSL.

# 8

## Conclusions

Modular language implementation is the first step towards bringing language implementation to a wider audience. The model we presented can be easily implemented using many of the already existing tools. However, we showed that a native implementation of this model gives a greater range of possibilities. For instance, separate compilations not only do enable users to distribute reusable components without bundling the original source code, but also to extend and build upon the original language implementation leaving it untouched. The `neverlang.js` experience (Sect. 6.2) showed that it is possible to develop language extensions in parallel and in isolation, and test and integrate them with the core implementation. The variability management experience (Sect. 6.4) has shown that multiple language components can be mined to simplify the definition of language variants from pre-built artifacts.

The Neverlang framework has been successfully employed in real-world projects. TheMatrix [39] is a Java framework to query and manipulate Italian's national healthcare databases to produce statistics on the prevalence of chronic diseases and estimate the standards of care across the country. The Tyl Language is an experimental business-oriented DSL for the development of ERP software<sup>1</sup>. The same implementation of Neverlang's compiler `nlgc` (Sect. 4.3) and of the interpreter for a complete, modern programming language (`neverlang.js`, Sect. 6.2) are a testament to the strengths of Neverlang and its underlying model. We have also realized a modular Java language pre-processor, in the style of Polyglot [68], SugarJ [30], and ableJ [86]. Other experiments involved experimenting with a simple implementation of the Logo programming language. Because the core API is entirely Java 6 compatible, we were also able to successfully port the entire Neverlang implementation onto Android (an example use case would be the *Recipe* language described in [18]), where its dynamic loading capabilities could be useful to separately distribute *plug-ins* for a core language implementation.

<sup>1</sup><http://www.mate.it/index.php/en/tyl>

## 8. Conclusions

Nevertheless, in Neverlang there is still much room for improvement. Future work will concentrate both on its expressive power and ease of use. In Sect. 4.2 and Sect. 6.2 we mentioned runtime evolution and a DSL for tree rewriting. These extensions are already in development, and thanks to the architecture of the framework they should be ready soon. The language plugin system makes it possible to easily support new programming languages for semantic actions, and the dynamic mapping between parse trees and semantic actions should make easy to support dynamic dispatching of alternative actions.

From a methodological point of view, future efforts should be also geared towards establishing a set of guidelines so that users may define language components (slices) that are easy to combine. In Sect. 3 we gave an overview of the principle of *dependency* between language components, and in the following sections a few examples were given where reliable language composition was an effect of *naming conventions*. Even though Neverlang provides *renaming* and *remapping* capabilities (Sect. 4), guidelines on naming and factorization of language components are still an open problem that affects any modular language implementation tool. State-of-the-art static analysis techniques (e.g., [55]) may be also employed to further simplify component development. Our plan is to explore this problem in more depth through field studies and further experiences.





## Formal Composition Model

In Chapter 3 high-level description of the model of a feature-oriented, componentized language implementation framework were given. Many existing frameworks may fit this model. In the following we will try and give a formal description of the Neverlang implementation using formal grammars for syntax definition and attribute grammars for semantics specification, because they are the models that this implementation fits better. Not all the frameworks may implement this abstract model to the letter: for instance, they may not use BNF grammars for syntax or they may not implement the semantics using attribute grammars (e.g., Spoofox uses SDF [45] for grammars and tree rewriting for semantics).

### A.1. Decomposition of Syntax Definitions

Let us consider a grammar  $G = \langle \Sigma, N, P, S \rangle$ . Then  $P$  can be partitioned in  $n > 0$  sets  $P_0, P_1, \dots, P_{n-1}$ . A syntax definition of a *language component* can be seen in fact as the set of a partition of the grammar  $G$  of the entire language implementation. For instance, if  $G$  were the grammar for a Java-like programming language, there would be a partition  $P_i$  ( $i \in 0, 1, \dots, n-1$ ) describing the syntax of the **while** loop (Sect. 3), one for the **if** branch and so on. It follows that for any choice of a number  $k \leq n$  of partitions of  $P$ , it is always possible to define a subset of such language by a grammar

$$G' = \langle \Sigma, N, P_{j_0} \cup P_{j_1} \cup \dots \cup P_{j_k}, S \rangle,$$

where, for all  $t$ ,  $P_{j_t} \in \{P_0, P_1, \dots, P_{n-1}\}$ . It also follows that *any language* can be represented as the union of sets of productions over the alphabets of symbols  $\Sigma$  and  $N$  with an axiom  $S$ . Any syntax definition in a *language component* may be seen as one such set of productions.

## A. Formal Composition Model

**Non-empty Languages** In Chapter 3 one form of dependency between components is implied by the syntax definitions, because *placeholders* represent a feature that is *required*; these dependencies must be *satisfied* so that the resulting language is *meaningful*, that is, non-empty Chapter 2; thus, although the set  $P$  of productions in the tuple  $G = \langle \Sigma, N, P, S \rangle$  can be seen as the union of an arbitrary collection of  $k$  production sets  $\mathbf{P} = \{P_0, P_1, \dots, P_k\}$ , we are interested only in those grammars that generate a *non-empty* languages (Chapter 2). For instance, let be  $P_B \ni \{A \rightarrow B\}$  one set in  $\mathbf{P}$ . Then there must be some set  $P_A \ni \{X \rightarrow \omega_1 A \omega_2\}$  of  $\mathbf{P}$ , that is a set that contains a production where  $A$  occurs in its left-hand part, and it must be possible to derive  $A$  from  $S$ ; that is,  $S \Rightarrow_G^* A$ . Similarly, there must be  $P_B \ni \{B \rightarrow \beta\}$  in  $\mathbf{P}$ , such that  $A \Rightarrow_G^* w$ , with  $w \in L(G)$ ; and thus,  $S \Rightarrow_G^* w$ . In other words, for each set  $P_k$  of productions, we can define a set  $r(P_k) \subseteq N$  of *required* nonterminals and a set  $p(P_k) \subseteq N$  of *provided* nonterminals [99].

**Definition 8 (Provide and Require Sets).** Let be  $P$  a set of productions consisting of  $m$  productions and let the  $j$ -th production be

$$X_{j0} \rightarrow X_{j1}X_{j2}\cdots X_{jn_j}$$

- The *provide* set  $p(P)$  is the set of all the *left-hand* nonterminals of the rules in  $P$ :

$$p(P) = \{X_{j0} \mid j = 0, 1, \dots, m\}$$

- The *require* set is the set of all the *right-hand* nonterminals of all the rules in  $P$ :

$$r(P) = \{X_{ji} \mid j = 0, 1, \dots, |P|, 0 < i \leq n_j\}$$

- The *restricted require* set is the set of all the right-hand nonterminals that are *not* in the provide set of  $P$

$$\bar{r}(P) = r(P) \setminus p(P)$$

Then, given a set of productions  $P_k$  with a production  $A \rightarrow B$ , we can say that  $P_k$  is *providing*  $A$ , and it *requires*  $B$ .

**Definition 9 (Sat Relation).** Let be  $\mathbf{P}$  a collection of disjoint sets of productions, over the alphabets  $\Sigma$  of terminal symbols and  $N$  of nonterminal symbols. Let be  $P_k, P_j \in \mathbf{P}$ ,  $X \in N$ , the *satisfies* relation is the set  $\mathbf{Sat} \subseteq \mathbf{P} \times \mathbf{P} \times N$  and it is  $(P_k, P_j, X) \in \mathbf{Sat}$  (“ $P_k$  satisfies  $P_j$  with  $X$ ”) if and only if  $p(P_k) \cap r(P_j) \ni \{X\}$ , that is if  $P_k$  *provides* at least a nonterminal  $X$  that  $P_j$  *requires*.

The *satisfies* relation induces the definition of an *edge-labeled directed graph* where the vertex set is the set of all the grammar partitions, the edge set is the set of pairs  $(P_k, P_j)$  in a triple in  $\mathbf{Sat}$ , and the labels of the edges are the nonterminals in the same triple.

**Definition 10 (Dependency Graph).** The *dependency graph* for  $\mathbf{P}$  is the tuple  $D = \langle \mathbf{P}, E \rangle$ , where  $E = \{(P_k, P_j) \mid (P_k, P_j, X) \in \mathbf{Sat}\}$ , with a function  $\ell : E \rightarrow N$  such that  $\ell(d) = X$  for each  $d = (P_k, P_j) \in D$ , such that  $(P_k, P_j, X) \in \mathbf{Sat}$ .

## A.2. Decomposition of Language Semantics

Attribute grammars (see Chapter 2) describe the relations between attributes that are attached to nonterminal symbols in the tree representation of a program written in some given language. The semantics of a programming language can be described by as a sequence of *compilation phases*; with respect to attribute grammars, Adams [1], has shown that, if a *temporal ordering* exist, compilation phases can be implemented as a *sequence of attribute grammars*. Each attribute grammar implementing a phase would then communicate with other phases through attributes. As see in the formal model, a type checking phase may define a *type* attribute that would be used in the *code generation* phase. Attribute that are reused across phases realize what we called *inter-phase* dependencies in Chapter 3. In Chapter 3 we also gave a definition for *intra-phase* dependencies: these dependencies rise from the modularization of a *processing phase* along the axis of language constructs.

Let us consider some attribute grammar  $AG = \langle G, A^s, A^i, V, \sigma \rangle$  for a given grammar  $G = \langle \Sigma, N, P, S \rangle$ :

- $A^s(X)$  and  $A^i(X)$  are the attribute sets for any nonterminal  $X$  of  $G$ ,
- $V$  is a function assigning to each attribute  $X.\alpha$  a *domain*  $V(X.\alpha)$ , which may be written alternatively as  $V_{X,\alpha}$ , and
- $\sigma$  is the mapping assigning each production  $p = X_{p0} \rightarrow X_{p1}X_{p2}\dots X_{pn_p}$  a finite set  $\sigma(p)$  of *semantic rules*, and each semantic rule maps values of certain attributes of  $X_{p0}, X_{p1}, \dots, X_{pn_p}$  into the value of some attribute  $X_{pj}$ ,  $0 \leq j \leq pn_p$ .

Suppose that the productions  $P$  of a grammar  $G = \langle \Sigma, N, P, S \rangle$  has been partitioned into a set  $\mathbf{P}$  of  $n$  production sets. Then, for each partition  $P_k \in \mathbf{P}$ , let us denote with  $N_k \subseteq N$  the subset of the nonterminal alphabet  $N$  that contains only the nonterminals that occur in productions of  $P_k$ , and consider the tuple  $R_k = \langle P_k, A_k^s, A_k^i, V, \sigma_k \rangle$ :

- $A_k^s = \{A^s(X) \mid X \in N_k\}$ ; thus  $A_k^s$  is, the subset of *synthesized* attributes of only those nonterminals that occur in  $P_k$
- $A_k^i = \{A^i(X) \mid X \in N_k\}$ ; that is, the subset of the *inherited* attributes of only those nonterminals that occur in  $P_k$
- $\sigma_k$  is the restriction of  $\sigma$  to the sole productions in  $P_k$  (undefined otherwise).

Then, it is easy to see that, for all  $k = 0, 1, \dots, n - 1$  the original attribute grammar  $AG$  can be obtained by the respective union of all the components in each tuple  $R_k$ . The tuple  $R_k$  is the part of the *evaluation phase*  $R$  that pertains to the syntax definition in  $P_k$ . Because a language implementation consists of *many* phases, and thus, many attribute grammars, let us consider then a collection of attribute grammars  $\mathbf{AG} = \{AG_1, AG_2, \dots, AG_r\}$ , each representing a phase for the language specified by the grammar  $G$ . Then for each  $P_k$ , for each  $AG_j$ ,  $0 \leq j \leq r$  there is a role  $R_{kj}$ . Finally, consider the tuple  $S_k = \langle P_k, \mathbf{R}_k \rangle$  where  $\mathbf{R}_k = \{R_{k0}, R_{k1}, \dots, R_{kr}\}$ , in other words,  $\mathbf{R}_k$  is a collection of roles for the production set  $P_k$ . This is a *language component*.

## A. Formal Composition Model

**Abstract and Concrete Syntax** Now, consider some role  $R_{kj}$  with syntax  $P_k$ . Now let be  $S_k = \langle \bar{P}_k, \mathbf{R}_k \rangle$  a slice, with  $R_{kj} \in \mathbf{R}_k$ . The definition of role *does not* mandate that  $\bar{P}_k \equiv P_k$ . In fact, it is assumed that it may be  $\bar{P}_k \neq P_k$ . However, in this case, it is expected that there exist one mapping  $\varphi : \bar{P}_k \rightarrow P_k$  such that  $\forall p \in \bar{P}_k : \varphi(p) = q, q \in P_k$ . In particular, let be  $X_0, X_1, \dots, X_n$  the nonterminals in production  $p$ , then  $\varphi$  will map each  $X_i$  onto nonterminals  $Y_0, Y_1, \dots, Y_n$  in production  $q$ . In other words, roles may be defined in terms of a set of productions that differs from the set of productions in the slice, provided that a mapping between the set of productions of the slices and the set of productions for the roles exists.

**Relations Between Evaluation Phases** In Chapter 3 we saw that dependencies between language components are also introduced by *properties* of the *features*. In the case of attribute grammar, these properties are represented by *attributes* of the nonterminals. In the previous sections we gave the definition for **Sat** relation between two sets of productions  $P_k$  and  $P_j$  and the set of nonterminals  $N$  (Def. 9). We might now be tempted to formalize a relation **RSat** between roles  $R_k = \{P_k, A_k^s, A_k^i, V, \sigma_k\}$ ,  $R_j = \{P_j, A_j^s, A_j^i, V, \sigma_j\}$ , and the attributes that are defined and used within the semantic actions of these roles. Let us assume, without loss of generality, that the production sets are singletons defined as follows:  $P_k = \{X_0 \rightarrow X_1 X_2 \dots X_n\}$  and  $P_j = \{Y_0 \rightarrow Y_1 Y_2 \dots Y_m\}$ . Now, let us assume that  $P_k$  satisfies  $P_j$  with  $X_0$ , and thus it is  $(P_k, P_j, X_0) \in \mathbf{Sat}$ . By definition it is then  $Y_t = X_0$  for some  $0 \leq t \leq m$ .

In this case we cannot define one-sided *provides* and *requires* sets as we did for production sets: in fact  $R_k$  may *provide synthesized* attributes to  $R_j$ , and  $R_j$  may *provide inherited* attributes to  $R_k$ , and similarly they may both *require* attributes to be present. Of course, the relation is *satisfied* when the attributes *required* by one role are *provided* by the other. Now, consider Knuth's dependency graph [58, p.134], where there is an edge  $(Y.\beta, X.\alpha)$  if and only if the semantic rule for the value of attribute  $Y.\beta$  depends directly on (*requires*) the value of attribute  $X.\alpha$  (the attribute that  $X$  *provides*). In these cases we can say that  $X.\alpha$  *satisfies*  $Y.\beta$ . We can then define the set  $\mathbf{ASat} = \{(X, Y, \alpha) \mid \exists Y.\beta \in A(Y) : X.\alpha \text{ satisfies } Y.\beta\}$ . As the reader may have realized, the formal definition for the relation **RSat** is somewhat more complicated than the one for **Sat**. We will just give an informal definition of the **RSat** relation between *roles*  $R_k, R_j$  as the collection of all the pairs  $(R_k, R_j)$  such that:

- $(P_k, P_j, X) \in \mathbf{Sat}$
- $(X, Y, \alpha) \in \mathbf{ASat}, Y \in N_j, X.\alpha \notin A_j(X) \implies X.\alpha \in A_k^s(X)$  that is, whenever an attribute  $X.\alpha$  *satisfies* some attribute of a nonterminal  $Y$  found in the productions of  $P_j$  and that attribute is not already in  $A_j(X)$ , then  $X.\alpha$  must be a *synthesized* attribute of  $X$  in  $R_k$ .
- $(X', X, \beta) \in \mathbf{ASat}, X' \in N_k, X'.\beta \notin A_k(X') \implies X'.\beta \in A_j^i(X)$   
that is, whenever an attribute  $Y.\beta$  *satisfies* some attribute of a nonterminal  $X$  found in the productions of  $P_k$ , and that attribute is not already in  $A_k(X')$ , then  $Y.\beta$  must be an *inherited* attribute of  $Y$  in  $R_j$ .

This definition only captures *part* of the problem; in particular, with attribute grammars not only do we require for these properties to be satisfied, but also that computations of the attributes *terminate* correctly (*non-circularity*—*e.g.*, see [74]). A more extensive discussion of modularity and *verifiability* of language extensions in attribute grammars would go beyond the extent of this work, where AGs are only used as a possible model of the idea; for further details on AG-based verifiable language extensions we refer to [55]. It follows that, in order to realize a working language implementation, not only should the language be non-empty, but also both dependencies *within* phases and *across* phases (Sect. 3.1.2) should be satisfied, that is, there should be no references to undefined attributes.

**Language Implementation** We can finally give a definition of a *language implementation* as a collection of  $m$  slices  $S_0, S_1, \dots, S_m$ , where the generated language is *non-empty* for a given axiom, where all the syntactic and semantic dependencies are *satisfied*, and for which the *roles* defined in each  $S_k$  are executed in a given *sequence*; in particular, for any permutation of roles, their evaluation in sequence should be equivalent to evaluate in the same order the corresponding attribute grammars. In other words, if  $\mathbf{R}_k = \{R_{k0}, R_{k1}, \dots, R_{kr}\}$  are the roles for  $S_k$ , and  $AG_0, AG_1, \dots, AG_k$  are the respective attribute grammars of which each  $R_{ki}$  is a portion, if the roles are evaluated in the order  $(R_{k0}, R_{k1}, \dots, R_{kr})$ , then the evaluation should be equivalent to evaluating the sequence of attribute grammars  $(AG_0, AG_1, \dots, AG_k)$ .



# B

## On The Relation Between LR Goto-Graphs

The DEXTER parser generator is the component that performs the syntax analysis in Neverlang. DEXTER generates LR parse tables on the fly using an algorithm described in [19]. This algorithm *transforms* the LR *goto-graph* [2] of a given grammar  $G$  into the *goto-graph* of a grammar  $G'$ , if  $G'$  can be described in terms of  $G$ , plus a set of *new productions*  $Q$ . In this appendix we have reproduced, for reference, the relevant section of the full journal paper. The proofs assume knowledge of formal grammars and the LR parsing technique.

**The Goto-Graph.** The *augmented* grammar of a given grammar  $G_0 = \langle \Sigma, N, S, P \rangle$  is the tuple

$$G = \langle \Sigma \cup \{\$, \}, N \cup \{S'\}, S', P \cup \{S' \rightarrow S\$\} \rangle$$

with  $\$ \notin \Sigma$ . We will call  $S' \rightarrow S\$,$  where  $S' \notin N$ , the *starting production*.

An *LR( $o$ ) item* (*item* for short) is a dotted production rule, e.g.,  $A \rightarrow \alpha \cdot X\beta$ . A generic item will be denoted by  $\zeta, \zeta', \zeta'',$  etc. In the following, we may say that symbol  $X$  is “dotted in  $\zeta$ ” if it is preceded by  $\cdot$  in the item  $\zeta$ . We say the dot “ $\cdot$ ” to be *leftmost* in  $\zeta$  if  $\zeta = A \rightarrow \cdot \omega$  (for some  $A, \omega$ ) and we say the dot to be *rightmost* if  $\zeta = A \rightarrow \omega \cdot$ . A rule with a rightmost “ $\cdot$ ” is said to be a *reduction candidate* or more simply a *candidate*. The item  $S' \rightarrow \cdot S\$,$  will be called “initial item”. For convenience we also define the following expressions:

$$\text{next}(A \rightarrow \alpha \cdot X\beta) \triangleq A \rightarrow \alpha X \cdot \beta \quad \text{and} \quad \text{prev}(A \rightarrow \alpha X \cdot \beta) \triangleq A \rightarrow \alpha \cdot X\beta$$

with  $\text{next}(\zeta) = \zeta$  when  $\zeta$  is candidate, and  $\text{prev}(\zeta) = \zeta$  when the dot in  $\zeta$  is leftmost. Now, let  $G = \langle \Sigma, N, S, P \rangle$  be a grammar and let  $I$  be a set of items. The *closure* of  $I$  (with respect to  $G$ ) is defined as the smallest set  $\text{CLOSURE}_G(I)$  such that:

1.  $I \subseteq \text{CLOSURE}_G(I)$

## B. On The Relation Between LR Goto-Graphs

$$2. B \rightarrow \alpha \cdot A\beta \in \text{CLOSURE}_G(I), A \rightarrow \gamma \in G \implies A \rightarrow \cdot \gamma \in \text{CLOSURE}_G(I)$$

In other words, for all productions  $B \rightarrow \alpha \cdot A\beta \in I$ , then  $\text{CLOSURE}_G(I)$  is the set obtained by first enriching  $I$  with all the items  $A \rightarrow \cdot \gamma$  (provided  $A \rightarrow \gamma \in G$ ), and then enriching iteratively the obtained set until no more items can be added.

A *kernel item* is either the initial item  $S' \rightarrow \cdot S\$$  of an augmented grammar or an item whose dot is non-leftmost. Any other item is called *nonkernel*. For each set of items  $I$  we denote with  $K(I)$  the subset of  $I$  that contains all of its kernel items. Notice that it is always  $\text{CLOSURE}_G(K(I)) = I$ , thus, for each pair of sets  $I, J$  it is  $K(I) = K(J)$  if and only if  $I = J$ .

The particular collection of sets of LR(o) items of a given grammar that is be used to drive the pushdown automaton of an LR parser is called the *canonical* LR(o) collection; we will indicate this collection with the symbol  $\mathcal{I}$ . The set  $\mathcal{I}$  is the result of the iterative application of the  $\text{CLOSURE}$  function. We call a set of items  $I$  *state* when  $I \in \mathcal{I}$ . The algorithm to generate the LR(o) set of states  $\mathcal{I}$  from an augmented grammar is described in [2]. the procedure uses the  $\text{GOTO}$  function, defined as follows: let  $\mathcal{I}$  be the set of LR(o) states for grammar  $G$ , and let  $I \in \mathcal{I}$  and  $X \in (\Sigma \cup N \cup \{\epsilon\})$  then

$$\text{GOTO}_G(I, X) \triangleq \text{CLOSURE}_G(\{A \rightarrow \alpha X \cdot \beta \mid A \rightarrow \alpha \cdot X\beta \in I\}). \quad (\text{B.1})$$

Because of the algorithm definition, we call the closure of set  $I = \{S' \rightarrow \cdot S\}$  the *initial state* of the canonical LR(o) collection of states, and we usually indicate it with  $I_0$ .

It is quite common to represent the canonical LR(o) set of states  $\mathcal{I}$  as a graph. Informally, the set of vertices of this graph corresponds to  $\mathcal{I}$ , and the set of edges is the set of pairs  $(I, J)$ , with  $I, J \in \mathcal{I}$  and such that  $\text{GOTO}_G(I, X) = J$  for some symbol  $X$ .

**Definition 11** (Goto-Graph). Let  $\mathcal{I}$  be the LR(o) set of states for a grammar  $G$ , then the goto-graph is a tuple  $\Gamma_G = \langle V, E \rangle$  where  $V$  is the set of vertices and  $E$  is the set of edges. Each vertex  $q$  corresponds to one and only one state  $I \in \mathcal{I}$ . For each  $q \in V$ , we denote with  $\ell_G(q)$  the kernel  $K(I)$  of the corresponding state. Then  $E$  is the subset of  $V \times V$  such that for all pairs  $(p, q) \in E$ :

1.  $\ell_G(p) = K(I), \ell_G(q) = K(J)$  and  $I, J \in \mathcal{I}$
2.  $\text{GOTO}_G(I, X) = J$  with the symbol  $X \in (\Sigma \cup N \cup \{\epsilon\})$ .

We may also write  $\ell_G(p, q)$  to refer to the label of the edge  $(p, q)$ ; if this label is  $X$  then  $\ell_G(p, q) = X$ ; if  $(p, q) \notin E$  then we might write  $\ell_G(p, q) = \perp$ . For simplicity, we may also write  $\delta(p, X) = q$  when  $\text{GOTO}_G(I, X) = J$  and  $\delta(p, X) = \perp$  when  $\text{GOTO}_G(I, X) = \emptyset$ . To denote the edge  $(p, q)$  with label  $X$  we will also use the notation  $p \xrightarrow{X} q$ . A sequence  $\ell_G(p, q) = X, \ell_G(q, r) = Y$  could be also written  $p \xrightarrow{X} q \xrightarrow{Y} r$ .

As you may have noticed, because  $I = \text{CLOSURE}_G(K(I))$ , each vertex  $q \in V$  can be labeled with the kernel  $K(I)$  of some state  $I$ . It follows from the definition of LR(o) states that  $\ell_G(q) = \ell_G(p)$  if and only if  $q \equiv p$ . Finally, since the LR(o) collection of states admits a notion of *initial state*, we can also define a notion of *starting vertex* for the goto-graph.



**Definition 12** (Starting Vertex). Let be  $\Gamma_G = \langle V, E \rangle$  the goto-graph for  $G$  and let be  $q_0 \in V$  the vertex such that the starting production  $S' \rightarrow S\$ \in \ell_G(q_0)$ ; then  $q_0$  is the *starting vertex* of the goto-graph.

Let us now define two operations over grammars called *growth* and *shrinkage*. The result of these operations applied to an input grammar  $G$  and a production  $A \rightarrow \omega$  is a new (possibly identical) grammar  $G'$ .

**Definition 13** (Growth). Let  $G = \langle \Sigma, N, S, P \rangle$  be a grammar and  $A \rightarrow \omega$  a production, and let us denote with  $\sigma(A \rightarrow \omega)$  the set of terminal symbols and with  $\nu(A \rightarrow \omega)$  the set of non-terminal symbols of the new rule; then grammar

$$G' = \langle \Sigma \cup \sigma(A \rightarrow \omega), N \cup \nu(A \rightarrow \omega), S, P \cup \{A \rightarrow \omega\}, \rangle.$$

is the *growing* grammar for  $G$ , and we will denote it with  $G \oplus \{A \rightarrow \omega\}$ .

**Definition 14** (Shrinkage). Let  $G = \langle \Sigma, N, S, P \rangle$  be a grammar and  $A \rightarrow \omega$  a production; then

$$G' = \langle \Sigma, N, P \setminus \{A \rightarrow \omega\}, S \rangle.$$

is the *shrinking* grammar for  $G$ , and we will denote it with  $G \ominus \{A \rightarrow \omega\}$ .

Finally, we make the following generalizations: let  $Q$  be a non-empty set of productions such that  $Q = \{A_1 \rightarrow \omega_1, A_1 \rightarrow \omega_2, \dots, A_n \rightarrow \omega_n\}$  then we define

$$G \oplus Q \triangleq G \oplus \{A_1 \rightarrow \omega_1\} \oplus \{A_1 \rightarrow \omega_2\} \oplus \dots \oplus \{A_n \rightarrow \omega_n\}. \quad (\text{B.2})$$

If  $Q \subseteq P$  we can define  $G \ominus Q$  similarly.

## B.1. Goto-Graphs and Growing Grammars

In this section we will find a relation between the graph of a given grammar  $G$  and the graph of a corresponding growing grammar  $G' = G \oplus Q$ . This relation will help us to define a procedure to *augment* the graph of  $G$  in such a way that the result is equivalent to the graph of the growing grammar  $G'$ . We will then describe the opposite procedure, to obtain the graph of a shrinking grammar  $G \ominus Q$ , starting from a given  $G$ . Before we carry on with the details, it is useful to introduce a notion of *path*.

**Definition 15** (Path). Let  $\Gamma_G = \langle V, E \rangle$ ; we call *path* a string  $\alpha = X_0 X_1 \dots X_k$  of symbols of  $(\Sigma \cup N \cup \{\epsilon\})$  such that:

$$q_0 \xrightarrow{X_0} q_1 \xrightarrow{X_1} q_2 \xrightarrow{X_2} \dots \xrightarrow{X_{k-2}} q_{k-1} \xrightarrow{X_{k-1}} q_k$$

where  $q_0$  is the starting vertex by convention, and  $q_1, q_2, \dots, q_k$  is any sequence of vertices of  $V$  for which the condition holds. We can say that path  $\alpha$  *reaches*  $q_k$  or that  $q_k$  is *reachable through*  $\alpha$ .

## B. On The Relation Between LR Goto-Graphs

The length of a path is generally the length of the word  $\alpha$ , unless the last symbol  $X_{k-1} = \varepsilon$ ; in that case (and only in that case) the length of the path is  $|\alpha| + 1$ . In fact, because of the way LR(o) states are constructed, if there is one  $\varepsilon$  on a path, it is always the last symbol of the path: if  $p \xrightarrow{\varepsilon} q$ , it is not possible that there is some  $\bar{X}$  such that  $p \xrightarrow{\varepsilon} q \xrightarrow{\bar{X}} r$ , as it is generally assumed that rules containing  $\varepsilon$  are always of the form  $Z \rightarrow \varepsilon$ , with  $Z$  being a nonterminal. Thus,  $\ell(q) = \{Z \rightarrow \cdot \varepsilon\}$ , which implies that there cannot be any  $\bar{X}$  such that  $q \xrightarrow{\bar{X}} r$ : in fact, the case  $p \xrightarrow{\varepsilon} q \xrightarrow{\bar{X}} r$  would only be possible if the grammar contained a rule of the form  $Z \rightarrow \omega \varepsilon \bar{X} \omega'$  with  $\omega, \omega' \in \Sigma \cup N$ ; but then the rule would be written as  $Z \rightarrow \omega \bar{X} \omega'$ .

The degenerate 0-length path is admissible only in those graphs  $\Gamma_G$  where  $V \equiv \{q_0\}$ . For instance, this is the case for a grammar with one sole production of the form  $S \rightarrow A$ , that is a grammar where  $A$  is a useless nonterminal. It is easy to see that in these cases the language generated by  $G$  is empty: this is not to be confused with those grammars  $G$  whose generated language is the sole word  $\varepsilon$ : in this case there will be at least one path  $q_0 \xrightarrow{\varepsilon} q_1$ , for some  $q_1 \in V$ .

For the sake of simplicity, in the following we will assume grammars not to include productions of the form  $Z \rightarrow \varepsilon$ , therefore, for all paths  $\alpha$  the length of a path will be the length of the word  $|\alpha|$ .

We shall now consider a grammar  $G = \langle \Sigma, N, P, S \rangle$  and its growing grammar  $G' = G \oplus Q$ . For the sake of simplicity, we will suppose that the production set  $Q$  is always a singleton<sup>1</sup>; the results can be easily generalized to the case when the cardinality  $m$  of  $Q$  (denoted by  $|Q|$ ) is greater than 1 by considering the  $m$  singleton sets (one set for each rule of  $Q$ ) and the chain seen in (B.2).

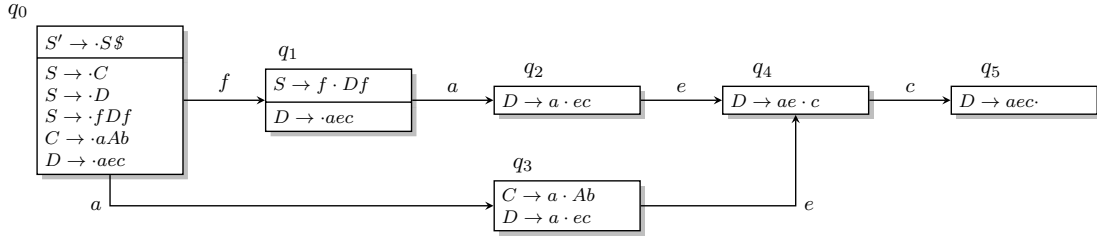
**The Graph of  $G'$  as an Augmented  $\Gamma_G$ .** Earlier work *e.g.* [48] has proven by counterexamples that given the goto-graph  $\Gamma_G$  for  $G$  and a growing grammar  $G' = G \oplus Q$  it is not always true that  $\Gamma_G$  is a subgraph of  $\Gamma_{G'}$ . In fact, due to what has been called a *splitting phenomenon* [48], one vertex in  $\Gamma_G$  might correspond to more than one vertex in  $\Gamma_{G'}$ . Consider the next example.

*Example 1 (Splitting).* The following is the grammar  $G$  presented in [48].

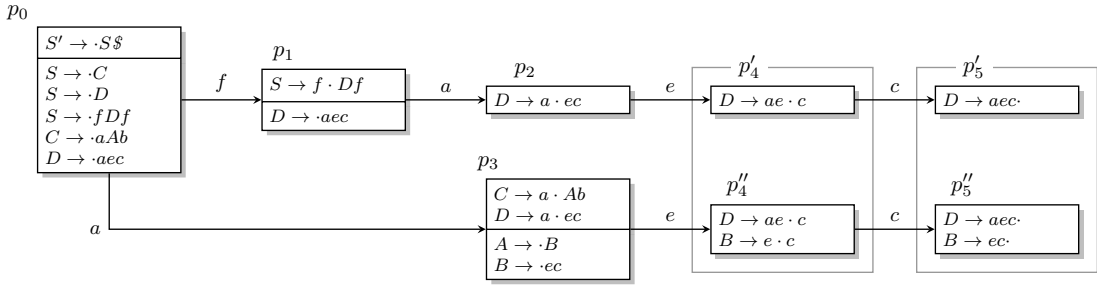
$$S \rightarrow C \mid D \mid fDf, C \rightarrow aAb, D \rightarrow aec, B \rightarrow ec \quad (\text{B.3})$$

We are showing the relevant vertices and edges of this goto-graph in Fig. B.1. Please notice that  $B$  is an unreachable nonterminal and  $A$  is useless. If we consider  $G' = G \oplus Q$ , with  $Q = \{A \rightarrow B\}$ , here  $B$  becomes reachable and  $A$  is no longer useless. In the new graph  $\Gamma_{G'}$  (Fig. B.2) that we can obtain by applying the canonical algorithm [2], we could informally say that some vertices have *split*. In particular, while it is easy to find a bijection between  $q_0, q_1, q_2, q_3$  and  $p_0, p_1, p_2, p_3$ , respectively, it is harder to decide whether  $q_4$  is related to  $p'_4$  or  $p''_4$ . In fact, in a certain sense we could even say that  $q_4$  is related to *both*  $p'_4$  and  $p''_4$ .

<sup>1</sup>We disregard the case when  $Q$  is empty since it is explicitly excluded by our definition of growing grammar (Def. 13).



**Figure B.1.:** Representation of a portion of  $\Gamma_G$  for the grammar  $G$  in Example 1..



**Figure B.2.:** Representation of a portion of  $\Gamma_{G'}$  for the grammar  $G'$  in Example 1..

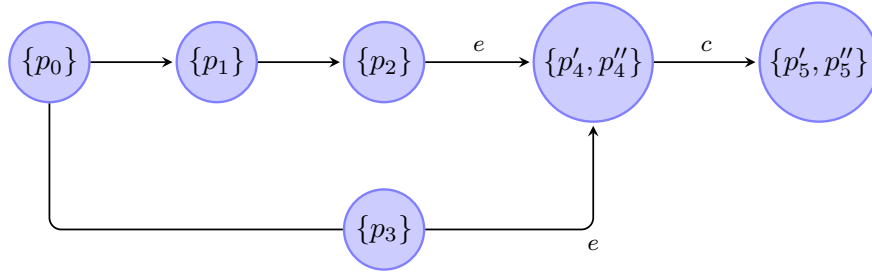
The conclusion is that the relation between  $\Gamma_G$  and  $\Gamma_{G'}$  is nontrivial, and that, in general, we cannot say that, for any set of productions  $Q$  and for any grammar  $G$ ,  $\Gamma_G$  is just a subgraph of  $\Gamma_{G \oplus Q}$ . Thus, it is not possible to find a simple mapping  $\varphi : V \rightarrow V'$ , but, more precisely, because of the splitting phenomenon that we just observed it might be possible to find some

$$\varphi : V \rightarrow \wp(V') \quad (\text{B.4})$$

In general, if  $q \in V$  we will expect  $\varphi(q)$  to be a singleton. In other words we usually expect  $q$  to correspond to one and only one vertex  $p$  of  $\Gamma_{G'}$ . For instance, in our example,  $q_0$  corresponds to  $p_0$  and  $q_1$  corresponds to  $p_1$ , therefore we could pose  $\varphi(q_0) \triangleq \{p_0\}$  and  $\varphi(q_1) \triangleq \{p_1\}$ . Then, by visually comparing Fig. B.1 and B.2, it is tempting to conjecture that there might be a way to map  $q_4$  onto the set of  $\{p'_4, p''_4\}$ , and similarly map  $q_5$  onto  $\{p'_5, p''_5\}$ . If this were possible, then, these would be the cases when a vertex *has split*. In Sect. B.1.1 we will find the mapping  $\varphi$  and give a formal definition of *split vertex*.

Our final goal is to find some graph  $\hat{\Gamma}$  isomorphic to  $\Gamma_G$ , so that, given  $\Gamma_G$  and the set  $Q$  of new productions, it is possible to define  $\Gamma_{G'}$  in terms of  $\hat{\Gamma}$ , a set of new vertices and a set of new edges. Now, let be  $\hat{\Gamma} = \langle \hat{V}, \hat{E} \rangle$ : we can define the sets  $\hat{V}$  and  $\hat{E}$  in terms of the mapping  $\varphi$ ; because for each  $q \in V$  the image  $\varphi(q)$  is a *set* of vertices of  $V$ , then the set of vertices for  $\hat{\Gamma}$  can be defined as a *collection of sets* of vertices in  $V$ ; the set of edges  $\hat{E}$  would be then a set of pairs of elements of  $\hat{V}$ . Thus, it would be natural to pose  $\hat{V} \subseteq \wp(V)$  and  $\hat{E} \subseteq \hat{V} \times \hat{V}$ ; in particular, we want  $\hat{V} = \varphi(V)$  and therefore

## B. On The Relation Between LR Goto-Graphs



**Figure B.3.:** The portion of graph  $\hat{\Gamma}$  that relates the portion of  $\Gamma_{G'}$  in Fig. B.2 to the portion of  $\Gamma_G$  in Fig. B.1..

$\hat{E} \subseteq \varphi(V) \times \varphi(V)$ ; that is,  $\hat{E}$  would contain edges between images of  $V$  through  $\varphi$ . For instance, let us impose  $\varphi(p_4) \triangleq \{p'_4, p''_4\}$  and  $\varphi(p_5) \triangleq \{p'_5, p''_5\}$  and let them be vertices in  $\hat{\Gamma}$  (Fig. B.3). In  $\Gamma_{G'}$  we have  $p'_4 \xrightarrow{c} p'_5$  and  $p''_4 \xrightarrow{c} p''_5$ . Therefore we would like  $\hat{\Gamma}$  to contain edge  $\varphi(p_4) \xrightarrow{c} \varphi(p_5)$ , because this edge is easy to trace back to edge  $p_4 \xrightarrow{c} p_5$  in  $\Gamma_G$ ; moreover, edge  $(\varphi(p_4), \varphi(p_5))$  should be in  $\hat{\Gamma}$  because we know that  $p'_4 \xrightarrow{c} p'_5$  and  $p''_4 \xrightarrow{c} p''_5$  in  $\Gamma_{G'}$ . This could be expressed by a function

$$\psi : E \rightarrow \wp(E') \quad (\text{B.5})$$

that maps edges in  $E$  onto collection of edges in  $E'$ , in such a way that edges between vertices like  $p_4$  and  $p_5$  are mapped into the set of edges between  $p'_4, p'_5$  and  $p''_4, p''_5$ ; in other words, we want that  $\psi(\varphi(p_4), \varphi(p_5)) = \{(p'_4, p'_5), (p''_4, p''_5)\}$ . In Sect. B.1.2 we will find this mapping  $\psi$  and we will describe the construction for  $\hat{\Gamma}$ .

Finally, in Sect. B.1.3, we will describe a construction to obtain  $\Gamma_{G'}$  (modulo one isomorphism) by augmenting  $\Gamma_G$ . In particular we will find a set  $\bar{V} \subseteq V$  and a set  $\bar{E} \subseteq E$  such that for some sets  $\Delta\bar{V}$  and  $\Delta\bar{E}$ , the graph  $GG$ :

$$GG = \langle \bar{V} \cup \Delta\bar{V}, \bar{E} \cup \Delta\bar{E} \rangle$$

is isomorphic to  $\Gamma_{G'}$ .

### B.1.1. Construction of $\varphi$ and $\Delta V$

The mapping  $\varphi$  can be constructed inductively.

- First, we define a family of functions  $\varphi_n$ : each of these functions maps any vertex on a  $r$ -length path ( $r \leq n$ ) in  $\Gamma_G$  into a collection of vertices of  $\Gamma_{G'}$ ;
- we then define  $\varphi$  in terms of this family of functions

Let us define a family of sets  $V_n \subseteq V$ . Each  $V_n$  contains each vertex of  $V$  that is reachable in  $\Gamma_G$  through every path with length at most  $n$  (see Def. 15). For instance, in Fig. B.1,  $V_0 = \{q_0\}$ ,  $V_1 = \{q_1, q_3\}$ ,  $V_2 = \{q_2, q_4\}$ , etc. If the graph is *acyclic*, then there exists a

longest finite path of length  $k$  in  $\Gamma_G$ , and we can write:

$$V = \bigcup_{n=0}^k V_n \quad (\text{B.6})$$

where  $V_0 \triangleq \{q_0\}$  (by the definition of paths). However, if  $\Gamma_G$  is *cyclic*, then there are infinite possible paths; therefore (B.6) becomes:

$$V = \lim_{k \rightarrow \infty} \bigcup_{n=0}^k V_n \quad (\text{B.7})$$

We can now define the family of applications:

$$\varphi_n : \bigcup_{r=0}^n V_r \rightarrow \wp(V')$$

Each of these functions maps any vertex  $q$  on any  $r$ -length path, with  $r \leq n$  to a collection of vertices of  $\Gamma_{G'}$ . Let us now suppose that  $q, q', q_0 \in V$  and  $p, p', p_0 \in V'$ , where  $q_0$  is the starting vertex for  $\Gamma_G$ , and  $p_0$  is the starting vertex for  $\Gamma_{G'} = \langle V', E' \rangle$ . We can then proceed to construct  $\varphi$  inductively as follows:

$$\begin{aligned} \varphi_0(q_0) &\triangleq \{p_0\} \\ \varphi_n(q) &\triangleq \{p \mid \exists p' \in V', q' \in V : p' \in \varphi_{n-1}(q'), p' \xrightarrow{X} p, q' \xrightarrow{X} q, \text{ for some } X\} \end{aligned} \quad (\text{B.8})$$

In other words, we impose  $q_0$  to map to the singleton set  $\{p_0\}$ ; in fact, since  $q_0$  is the starting vertex, it can never split. Then, the image of  $q \in V$  on a  $r$ -length path ( $r \leq n$ ) is defined in terms of  $\varphi_{n-1}$  as the collection of all those vertices  $p$  such that:

- there is an edge  $p' \xrightarrow{X} p$  in  $\Gamma_{G'}$
- $p'$  was in the image of a vertex  $q'$  on a path not longer than  $n - 1$
- there is an edge  $q' \xrightarrow{X} q$  in  $\Gamma_G$

For instance, with reference to Fig. B.1,  $\varphi_1(q_1) = \{p_1\}$  because  $q_1$  can be reached from  $q_0$  through an  $r$ -length path, where  $r \leq 1$ , that is, the 1-length path  $f$ , and  $p_1$  can be reached through the same 1-length path  $f$  from  $p_0$  (with  $p_1$  in Fig. B.1). Also,  $\varphi_3(q_4) = \{p'_4, p''_4\}$  ( $p'_4, p''_4$  in Fig. B.1); in fact,  $q_4$  can be reached from  $q_0$  through *two*  $r$ -length paths from  $q_0$ , with  $r \leq 3$ , and said paths are  $ae$  and  $fae$ ;  $p'_4$  and  $p''_4$  can be reached through those same paths, respectively.

Now, let us consider  $\varphi_k$  for some  $k > 0$ . By definition, it is:

$$\varphi_k : \bigcup_{r=0}^k V_r \rightarrow \wp(V').$$

If the graph is *acyclic*, then there exist some finite  $\bar{k}$  such that  $V = V_0 \cup V_1 \cup \dots \cup V_{\bar{k}}$ ; thus, we can write:

$$\varphi_{\bar{k}} : V \rightarrow \wp(V') \quad (\text{B.9})$$

## B. On The Relation Between LR Goto-Graphs

and pose

$$\varphi \triangleq \varphi_{\bar{k}} \tag{B.10}$$

When the graph is *cyclic*, paths are infinite in number, but set  $V$  is still finite; so, there will still be a finite  $\bar{k}$  such that (B.10) holds. Consequently, even in this case  $\varphi \triangleq \varphi_{\bar{k}}$ .

*Example 2.* Consider a generic grammar  $G$  and its goto-graph  $\Gamma_G$ , of which Fig. B.1 is a partial representation. Now consider a growing grammar  $G'$  and its goto-graph  $\Gamma_{G'}$ , for which the splitting phenomenon described by Horspool [48] takes place. Suppose that the resulting  $\Gamma_{G'}$  is portrayed in Fig. B.2. We want to find a mapping between the set  $V = \{q_0, q_1, q_2, q_3, q_4, q_5\}$  of vertices of  $\Gamma_G$  and the set  $V' = \{p_0, p_1, p_2, p_3, p'_4, p''_4, p'_5, p''_5\}$  of vertices of  $\Gamma_{G'}$ . If we consider the construction in this section, the mapping  $\varphi$  is defined as follows:  $\varphi(q_i) = \{p_i\}$  for  $i = 1, 2, 3$ ; moreover  $\varphi(q_4) = \{p'_4, p''_4\}$  and  $\varphi(q_5) = \{p'_5, p''_5\}$ .

We shall now prove that the definition of  $\varphi$  is *well-posed*. This would hold true only if we could guarantee that no vertex of  $\Gamma_G$  is ever mapped onto an empty set. Otherwise, there would some vertices of  $\Gamma_G$  that could not be put in relation with any vertex of  $\Gamma_{G'}$ . This can only occur when there is a path in  $\Gamma_G$  that is *not* also in  $\Gamma_{G'}$ . We shall now prove (Theorem B.1.1) that this can never happen. We will see that  $\varphi$  is well-posed as a simple consequence (Corollary B.1.1).

**Theorem B.1.1.** Let be  $G' = G \oplus Q$ ; then every path in  $\Gamma_G$  is also in  $\Gamma_{G'}$ .

*Proof.* The theorem can be proven by induction over the length  $n$  of a path. The 0-length path (Def. 15) is the one where the starting vertex coincides with the last vertex. By definition, it is  $q_0 \in V$  and  $p_0 \in V'$ .

Now, by contradiction, suppose for  $n = 1$  that there is one path  $X$  in  $\Gamma_G$  that is not also in  $\Gamma_{G'}$ . Then for some  $q$ ,  $q_0 \xrightarrow{X} q$  but there is no  $p$  such that  $p_0 \xrightarrow{X} p$ . But then  $\delta(p_0, X) = \perp$ , which would mean that some rule  $A \rightarrow X\alpha$  is in  $G$  but not in  $G'$ : but this is impossible, because  $G' = G \oplus Q$ .

Now consider any  $n$ -length path, with  $n > 1$ . By the inductive hypothesis path  $\alpha = X_1 X_2 \dots X_{n-1}$  is both in  $\Gamma_G$  and  $\Gamma_{G'}$  and

$$p_0 \xrightarrow{X_1} p_1 \xrightarrow{X_2} \dots \xrightarrow{X_{n-1}} p_{n-1} \xrightarrow{X_n} p.$$

Then again, by contradiction, let us suppose that there is no  $p$  such that  $p_{n-1} \xrightarrow{X_n} p$ . But then there is an edge  $q_{n-1} \xrightarrow{X_n} q$  in  $\Gamma_G$  that is not in  $\Gamma_{G'}$ , which can only happen if some rule  $Z \rightarrow \omega_1 X_n \omega_2$  is in  $\Gamma_G$ , but not in  $\Gamma_{G'}$ , which is a contradiction since  $G' = G \oplus Q$ .  $\square$

**Corollary B.1.1.** The image of any vertex of  $\Gamma_G$  is non-empty.

*Proof.* Because of the inductive definition of  $\varphi$ , the corollary is in turn proven by induction. We posed  $\varphi_0(q_0) = \{p_0\}$  by definition (B.8); then obviously  $\varphi(q_0)$  is non-empty.

Now, for  $n > 0$ , consider a  $(n - 1)$ -length path, and let  $q$  be the last vertex of this path. By the inductive hypothesis there is at least one  $p \in V'$  such that  $p \in \varphi(q)$ . Now suppose that  $q \xrightarrow{X} q'$ . If  $\varphi(q')$  were empty, then there would be no  $p' \in V'$  such that  $p' \in \varphi(q')$ . But this would contradict Theorem B.1.1, and therefore  $\varphi(q')$  must be non-empty, too.  $\square$

The concept of *split vertex* that we introduced in Sect. B.1 will be now described more formally with the following definition.

**Definition 16 (Split vertices).** If  $q \in V$  and  $|\varphi(q)| > 1$  we say that  $q$  has *split* (in  $\Gamma_{G'}$ ) or that  $q$  is a *split vertex*.

The set of the *split vertices* (Def. 16) will be:

$$V_S \triangleq \{q \in V \mid |\varphi(q)| > 1\} \quad (\text{B.11})$$

We can also define the set

$$V'_S \triangleq \{p \in V' \mid \exists q_s \in V_S : p \in \varphi(q_s)\} = \bigcup_{q_s \in V_S} \varphi(q_s). \quad (\text{B.12})$$

The collection  $\Delta V$  of vertices is the collection of all those vertices of  $\Gamma_{G'}$  that are not an image of any vertex in  $\Gamma_G$ . In symbols:

$$\Delta V \triangleq \{p \in V' \mid \forall q \in V : p \notin \varphi(q)\} = V' \setminus \bigcup_{q \in V} \varphi(q) \quad (\text{B.13})$$

Since our initial objective was to define a graph  $\hat{\Gamma} = \langle \hat{V}, \hat{E} \rangle$  that relates  $\Gamma_G$  to  $\Gamma_{G'}$ , we can now pose the set of vertices  $\hat{V} \triangleq \varphi(V)$ . The relation  $\varphi(V) \subseteq \mathcal{P}(V)$  holds as expected at the beginning.

### B.1.2. Construction of $\psi$ and $\Delta E$

In this section we will define the set of edges  $\hat{E}$  for  $\hat{\Gamma}$ .

- We will define the function  $\psi$  to relate each edge in  $\Gamma_G$  to a (possibly non-singleton) collection of edges of  $\Gamma_{G'}$ ;
- we will define a set  $\hat{E} \subseteq \hat{V} \times \hat{V}$  using  $\psi$
- we will finally prove this definition to be well-posed.

Let us define  $\psi : E \rightarrow \mathcal{P}(E')$  by:

$$\psi(e = (q, q')) \triangleq \{(p, p') \in E' \mid p \in \varphi(q), p' \in \varphi(q')\} \quad (\text{B.14})$$

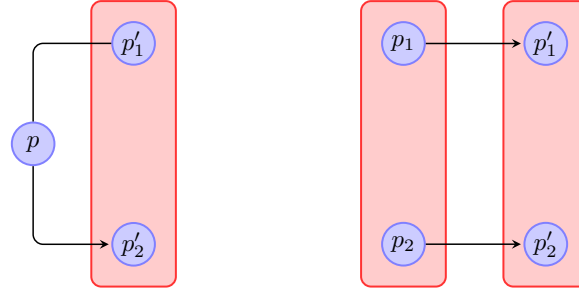
In other words, the image of  $(q, q') \in E$  is the collection of all those edges in  $E'$  between a vertex in the image of  $q$  and a vertex in the image of  $q'$ ; that is, that particular subset of  $\varphi(q) \times \varphi(q')$  that is contained in  $E'$ . Please notice that because of Theorem B.1.1, it is always  $\psi(e) \neq \emptyset$ , when  $e \in E$ .

*Example 3.* With respect to Example 2,  $\varphi(q_4) = \{p'_4, p''_4\}$ . In  $\Gamma_G$  there is one edge  $(q_4, q_5)$ , while in  $\Gamma_{G'}$  there are two edges  $(p'_4, p'_5)$  and  $(p''_4, p''_5)$ . The mapping  $\psi$  defines a relation between them all. In fact you can easily verify:

$$\psi((q_4, q_5)) = \{(p'_4, p'_5), (p''_4, p''_5)\}.$$

We would like now  $\hat{E}$  to be such that when  $(q, q') \in E$  also  $(\varphi(q), \varphi(q')) \in \hat{E}$ . For instance, we would like  $(\varphi(q_4), \varphi(q_5)) \in \hat{E}$ .

## B. On The Relation Between LR Goto-Graphs



**Figure B.4.:** Possible inconsistencies in the definition of  $\ell(\varphi(q), \varphi(q'))$ . On the left,  $\{(p, p'_1), (p, p'_2)\}$ ; on the right,  $\{(p_1, p'_1), (p_2, p'_2)\}$ .

Let us pose  $\hat{E}$  as follows:

$$\hat{E} \triangleq \{(\varphi(q), \varphi(q')) \mid \exists p \in \varphi(q), \exists p' \in \varphi(q'), (p, p') \in \psi((q, q'))\} \quad (\text{B.15})$$

That is, for each edge  $(\varphi(q), \varphi(q')) \in \hat{E}$  there is at least one edge between an element  $p \in \varphi(q)$  and an element  $p' \in \varphi(q')$ , and that edge  $(p, p') \in \psi((q, q'))$ .

Now we have potentially all the elements to define  $\hat{\Gamma} = \langle \hat{V}, \hat{E} \rangle$ . However, we still need to guarantee that every path in  $\Gamma_G$  is also in  $\hat{\Gamma}$  (Theorem B.1.3). In order to prove this, we need to give define properly the labels of the edges in  $\hat{E}$ . In particular, we want:

$$\ell(\varphi(q), \varphi(q')) = X \text{ when } q \xrightarrow{X} q', \text{ and } p \xrightarrow{X} p', \forall (p, p') \in \psi((q, q')).$$

However, given our definition of  $\psi$ , we might be concerned that in some situations the choice of that label  $X$  is not unique. In particular, if  $(q, q')$  is an edge, then the choice for  $\ell(\varphi(q), \varphi(q'))$  might not be unique (Fig. B.4):

- when  $\psi((q, q')) \ni \{(p, p'_1), (p, p'_2)\}$ ; in fact it should be  $\ell_{G'}(p, p'_1) \neq \ell_{G'}(p, p'_2)$ .

Otherwise the choice between  $p \xrightarrow{X} p'_1$  and  $p \xrightarrow{X} p'_2$  would be nondeterministic.

- when  $\psi((q, q')) \ni \{(p_1, p'_1), (p_2, p'_2)\}$ ; in fact, it *might* be  $\ell_{G'}(p_1, p'_1) \neq \ell_{G'}(p_2, p'_2)$ .

Let us see that neither of these can ever occur by proving the following theorem.

**Theorem B.1.2.** If  $q \xrightarrow{X} q'$  is in  $\Gamma_G$ , then, for all  $p' \in \varphi(q')$ , it is always  $p \xrightarrow{X} p'$ , for any edge  $(p, p')$  of  $\Gamma_{G'}$ , with  $p \in \varphi(q)$

*Proof.* Let us call  $e$  the generic edge  $(q, q') \in E$ . Consider the set  $\psi(e)$  and the collection

$$\ell(\psi(e)) = \{X \mid p \xrightarrow{X} p', (p, p') \in \psi(e)\}.$$

If any edge to a vertex  $p' \in \varphi(q')$  had always the same label  $X$ , then  $\ell(\psi(e))$  would be the singleton set  $\{X\}$ . Now, consider a vertex  $q'$  of  $\Gamma_G$ ; then, for some  $X \in (\Sigma \cup N \cup \{\varepsilon\})$ , and because of the definition of the goto-graph (Def. 11):

$$\forall q \text{ s.t. } (q, q') \in E : \ell_G(q, q') = X. \quad (*)$$



That is, any edge to  $q'$  of  $\Gamma_G$  has the same label  $X$ . But then  $\ell(\psi(e)) = \{X\}$ , because of the definition of  $\varphi$ .  $\square$

In the proof,  $(*)$  follows from the definition of goto-graph. In fact, in the goto-graph we have that  $q \xrightarrow{X} q'$  (which can also be written as  $\delta(q, X) = q'$ ) if and only if there are two states  $I, J \in \mathcal{I}$  such that  $\text{GOTO}_G(I, X) = J$  (Def. 11). Because  $\text{GOTO}$  is defined in terms of  $\text{CLOSURE}$ , if there is some state  $K \in \mathcal{I}$  such that  $\text{GOTO}_G(K, Y) = J$  then it must be  $Y \equiv X$ . Therefore, if there is some other edge  $(r, q')$ , for some  $r \in V$ , then it is always  $\delta(r, X) = q$ , that is,  $\ell_G(r, q) = X$ ; but then  $(*)$  holds. For a concrete example of this, consider vertices  $q_2, q_3, q_4$  in Fig. B.1; the label of the edges between this nodes is always  $e$ .

From Theorem B.1.2, it follows that we can always assign some single label  $X$  to  $\ell(\varphi(q), \varphi(q'))$ , so we can now legitimately write:

$$\varphi(q) \xrightarrow{X} \varphi(q').$$

We can finally prove that the definition of  $\hat{E}$  is well-posed.

**Theorem B.1.3.** A path is in  $\hat{\Gamma}$  if and only if it is in  $\Gamma_G$ .

*Proof.* Every path is in  $\hat{\Gamma}$  is also in  $\Gamma_G$  by construction. Let us then prove that when a path is in  $\Gamma_G$  it is also in  $\hat{\Gamma}$ , by induction on the length of the paths. For the 0-length path, the property is trivially true. Now, consider a path of length  $n > 0$  in  $\Gamma_G$  and  $\hat{\Gamma}$ , such that  $q$  and  $\varphi(q)$  is the last vertex on these paths, respectively; then for each  $q \xrightarrow{X} q'$  there must also be  $\varphi(q) \xrightarrow{X} \varphi(q')$ : in fact, let us suppose by contradiction that there is one edge  $q \xrightarrow{X} q'$  but there is no edge  $\varphi(q) \xrightarrow{X} \varphi(q')$ . Then, because of (B.14) and (B.15),  $\psi((q, q')) = \emptyset$ ; in other words,  $\forall p \in \varphi(q)$  and  $\forall p' \in \varphi(q')$  it is  $(p, p') \notin E'$ . But then there would be paths in  $\Gamma_G$  that are not also in  $\Gamma_{G'}$ , which would contradict Theorem B.1.1.  $\square$

We can now define *split edges* and describe another notable set of edges, similarly to what we did previously for vertex sets.

**Definition 17** (Split edges). If  $e \in E$  and  $|\psi(e)| > 1$  we say that  $e$  has split (in  $\Gamma_{G'}$ ) or that  $e$  is a *split edge*.

The *split edges* (Def. 17) will be the set:

$$E_S \triangleq \{e \in E \mid |\psi(e)| > 1\} \quad (\text{B.16})$$

Finally, the collection  $\Delta E$  of new edges is the set of all those edges in  $E'$  that are not the image of edges in  $E$ :

$$\Delta E \triangleq \{e' \in E' \mid \forall e \in E : e \notin \psi(e')\} = E' \setminus \bigcup_{e \in E} \psi(e) \quad (\text{B.17})$$

## B. On The Relation Between LR Goto-Graphs

### B.1.3. Construction of $\Gamma_{G'}$ from $\Gamma_G$

We will now describe how to obtain a graph  $\mathbf{G} = \langle V_G, E_G \rangle$  that is isomorphic to  $\Gamma_{G'}$ , starting from the given  $\Gamma_G$ .

**Theorem B.1.4.** Let  $\Gamma_G = \langle V, E \rangle$  and  $\Gamma_{G'} = \langle V', E' \rangle$ , with  $G' = G \oplus Q$ , for some  $Q$ . Then, there is a graph

$$\mathbf{G} = \langle V_G = (V \setminus V_S) \cup \Delta\bar{V}, E_G = (E \setminus E_S) \cup \Delta\bar{E} \rangle$$

where  $\Delta\bar{V} \subseteq V'$ ,  $\Delta\bar{E} \subseteq E'$ , and there is an isomorphism  $\varphi_G : V_G \rightarrow V'$  such that

$$(r, r') \in E_G \iff (\varphi_G(r), \varphi_G(r')) \in E' \quad (\text{B.18})$$

*Proof.* We first pose  $\bar{V} \triangleq V \setminus V_S$  and  $\bar{E} \triangleq E \setminus E_S$ . We also introduce the mapping  $\bar{\varphi} : \bar{V} \rightarrow V'$  as a restricted version of  $\varphi$  (B.10):

$$\bar{\varphi}(q) = p \iff \varphi(q) = \{p\}$$

Please notice that when  $V_S, E_S$  (see (B.11) and (B.16)) are empty, then  $\bar{V} \equiv V$  and  $\bar{E} \equiv E$ . Now, let us call  $\Delta\bar{V} \triangleq V'_S \cup \Delta V$ . Similarly, when  $V'_S$  is empty  $\Delta\bar{V} \equiv \Delta V$ . We can then write:

$$V_G \triangleq (V \setminus V_S) \cup (V'_S \cup \Delta V) = \bar{V} \cup \Delta\bar{V} \quad (\text{B.19})$$

Now, we want to express similarly  $E_G$ , that is:

$$E_G \triangleq \bar{E} \cup \Delta\bar{E}$$

$\bar{E}$  has been defined as the set of all those edges of  $\Gamma_G$  that are not split. Now, let us describe  $\Delta\bar{E}$ . The description of this set is not as simple as the one for  $\Delta\bar{V}$ , since it is supposed to be a collection of pairs that relate vertices of  $\bar{V}$  with new vertices in  $\Delta\bar{V}$ , in a way that makes  $\mathbf{G}$  isomorphic to  $\Gamma_{G'}$ . Let us define  $\Delta\bar{E}$  as follows:

$$\Delta\bar{E} \triangleq E_{\text{old}} \cup E_{\text{bdg}} \cup E_{\text{new}} \quad (\text{B.20})$$

- $E_{\text{old}}$  is the set of all those edges that were *not* in  $\Gamma_G$  but are between vertices that were *already* in  $\Gamma_G$  (modulo  $\varphi$ );
- $E_{\text{bdg}}$  is the set of the *bridging* edges; that is, all those edges between vertices of  $\Gamma_G$  (modulo  $\varphi$ ) and *new* vertices of  $\Gamma_{G'}$  (including split vertices), and vice versa;
- $E_{\text{new}}$  is the set of the edges between vertices that are completely new to  $\Gamma_{G'}$  (including split vertices)<sup>2</sup>

In symbols:

$$\begin{aligned} E_{\text{old}} &\triangleq \{(q, q') \notin E \mid q, q' \in \bar{V}, (\bar{\varphi}(q), \bar{\varphi}(q')) \in \Delta\bar{E}\} \\ E_{\text{bdg}} &\triangleq \{(q, p) \mid q \in \bar{V}, p \in \Delta\bar{V}, (\bar{\varphi}(q), p) \in E'\} \cup \{(p, q) \mid p \in \Delta\bar{V}, q \in \bar{V}, (p, \bar{\varphi}(q)) \in E'\} \\ E_{\text{new}} &\triangleq \{(p, p') \mid p, p' \in \Delta\bar{V}, (p, p') \in E'\} \subseteq E' \end{aligned}$$

---

<sup>2</sup>therefore  $V'_S \subseteq (E_{\text{bdg}} \cup E_{\text{new}})$

We then pose the following isomorphism  $\varphi_G : V_G \rightarrow V'$ :

$$\varphi_G(r) \triangleq \begin{cases} \bar{\varphi}(r), & r \in \bar{V} \\ r, & \text{otherwise} \end{cases}$$

- If  $(r, r') \in \bar{E}$  then  $(\varphi_G(r), \varphi_G(r')) \equiv (\bar{\varphi}(r), \bar{\varphi}(r')) \in E'$ ;
- If  $(r, r') \in E_{\text{old}}$  then  $(\varphi_G(r), \varphi_G(r')) \equiv (\bar{\varphi}(r), \bar{\varphi}(r')) \in \Delta E$ ;
- If  $(r, r') \in E_{\text{bdg}}$  then either:
  - $(\varphi_G(r), \varphi_G(r')) \equiv (\bar{\varphi}(r), r') \in \Delta E$  or  $(\varphi_G(r), \varphi_G(r')) \equiv (r, \bar{\varphi}(r')) \in \Delta E$ ;
- If  $(r, r') \in E_{\text{new}}$  then  $(\varphi_G(r), \varphi_G(r')) \equiv (r, r') \in \Delta E$ ;

Then, by construction, the (B.18) holds.  $\square$

The previous theorem describes the structure of a graph that isomorphic to  $\Gamma_{G'} = \langle V', E' \rangle$  starting from elements of  $\Gamma_G = \langle V, E \rangle$ . We called this graph  $\mathbf{G} = \langle V_G, E_G \rangle$ , but, from now on, we will assume the following:

**Remark B.1.1.** Because of Theorem B.1.4, without loss of generality, we can always assume that it is always  $\bar{V} = V \cap V', \bar{E} = E \cap E'$ . That is, from now on, we will always assume that  $V' \equiv V_G, E' \equiv E_G$ . In light of this, we can also assume:

$$\Gamma_{G'} \equiv \mathbf{G} \tag{B.21}$$

Until now, we put aside any consideration about the labels of the vertices on purpose. We will now see how labels change between a goto-graph  $\Gamma_G$  and the graph of its growing grammar  $\Gamma_{G'}$ . Heering [46] made similar observations in their early work on lazy construction of LR parsers. We can now restate these observations in light of the previous proofs. Intuitively, with the growth of the graph, labels grow as well. We will proceed by cases, first considering vertices that do not split, and then the case of split vertices.

**Theorem B.1.5 (Labels in  $\bar{V}$ ).** Let  $G = \langle \Sigma, N, S, P \rangle$  and  $G' = \langle \Sigma', N', S, P' \rangle$  two grammars such that  $G' = G \oplus Q$  where  $Q = \{A \rightarrow \omega\}$ ; let  $\Gamma_G = \langle V, E \rangle$  be the subgraph of  $\Gamma_{G'} = \langle V', E' \rangle$ , with  $\bar{V} = V \cap V', \bar{E} = E \cap E'$ ; then, for all  $q \in \bar{V}$ , it is always  $\ell_G(q) \subseteq \ell_{G'}(q)$ .

*Proof.* Let us again proceed by induction on the length of a path. On the zero-length path, we consider the starting vertex  $q_0$  (Def. 12): because of the definition of  $\ell$  as the kernel of an LR(o) set of items, it is easy to see that  $\ell_G(q_0) \equiv \ell_{G'}(q_0)$ , as it will only contain the initial item  $S' \rightarrow \cdot S \$$ . Therefore, it is also true that  $\ell_G(q_0) \subseteq \ell_{G'}(q_0)$ .

Now, for all  $q' \in \bar{V}$  on a  $(n-1)$ -length path, with  $n > 0$ , let us assume that the inductive hypothesis holds, and consider  $q$  such that  $(q', q) \in \bar{E}$ . Now, because of the ways goto-graphs are constructed (Def. 11):

$$\begin{aligned} \forall \zeta \in \ell_G(q) : \text{prev}(\zeta) \in \text{CLOSURE}_G(\ell_G(q')) \\ \forall \zeta' \in \ell_{G'}(q) : \text{prev}(\zeta') \in \text{CLOSURE}_{G'}(\ell_{G'}(q')) \end{aligned}$$

but, for the inductive hypothesis  $\ell_G(q') \subseteq \ell_{G'}(q')$ : then it is also  $\ell_G(q) \subseteq \ell_{G'}(q)$ .  $\square$

## B. On The Relation Between LR Goto-Graphs

The theorem above formally proves a simple intuitive observation: if the grammar has grown, then the label of a vertex can only grow; in particular it will grow if the closure changes. Split vertices are a special case, that we treat separately in this remark.

**Remark B.1.2 (Labels of the Split Vertices).** If  $q \in V_S$ , for each  $p \in \varphi(q)$ ,  $\ell_{G'}(p)$  is at least the same as  $\ell_G(q)$ ; in particular:

$$\ell_G(q) \subseteq \bigcap_{p \in \varphi(q)} \ell_{G'}(p) \quad \text{and} \quad \ell_G(q) \subseteq \bigcup_{p \in \varphi(q)} \ell_{G'}(p). \quad (*)$$

*Proof.* Let us first consider all those non-split vertices that lead to the split vertex  $q$  for some symbol  $X$ ; that is, all those  $q_i$  such that  $q_i \xrightarrow{X} q$  for  $i = 0, 1, \dots, n$ , for some  $n$ . Then, because of the definition of GOTO and CLOSURE (at the beginning of the appendix):

$$K(\text{GOTO}_G(\ell_G(q_i), X)) = \ell_G(q), \text{ for all } i = 0, 1, \dots, n$$

But, then this also means that there is a set of items  $L$  that is common to all labels  $\ell_G(q_i)$ , or, in symbols:

$$L \subseteq \bigcap_{i=0,1,\dots,n} \ell_G(q_i)$$

and this set  $L$  is such that  $K(\text{GOTO}_G(L, X)) = \ell_G(q)$ . Because every  $q_i$  is non-split, we already know that  $\ell_G(q_i) \subseteq \ell_{G'}(q_i)$ , then it is also:

$$L \subseteq \bigcap_{i=0,1,\dots,n} \ell_{G'}(q_i).$$

But then, by definition of  $\varphi(q)$ , for each  $p \in \varphi(q)$  there is at least one  $q_i$  such that  $q_i \xrightarrow{X} p$ , and it is  $\ell_{G'}(p) \supseteq K(\text{GOTO}_{G'}(L, X))$ . This proves the first inequality in (\*) for the case when every  $q_i$  is non-split. It is easy to see that the second inequality holds as well: in fact, if the relation did not hold true, then the labels of each  $p \in \varphi(q)$  would all coincide, but then, by definition of goto-graph  $\varphi(q) \equiv \{p\}$  which would mean  $q \notin V_S$ .

Now, let us consider some edge  $p \xrightarrow{X} p'$ , where  $p \in \varphi(q)$  and  $p' \in \varphi(q')$  such that  $q \xrightarrow{X} q'$ , where  $q, q'$  are both split vertices. By induction, for all  $p \in \varphi(q)$  there is a set  $L$  such that  $L \subseteq \ell_{G'}(p)$  and such that  $L \subseteq \ell_G(q)$ : then, for all  $p' \in \varphi(q')$  there is a set  $L'$  such that  $L' \subseteq \ell_{G'}(p')$  and such that  $L' \subseteq \ell_G(q')$ , and this set is  $L' = \text{GOTO}_{G'}(L, X)$ . This proves the left-hand inequality in (\*); the right-hand inequality, again, holds as well, otherwise  $q'$  would not be a split vertex.  $\square$

We can finally enunciate the following theorem.

**Theorem B.1.6 (Relation between labels).** Let be  $q \in V$ , then  $\forall p \in \varphi(q)$  it is  $\ell_G(q) \subseteq \ell_{G'}(p)$ .

The proof follows from Theorem B.1.5 and Remark B.1.2.

## B.2. Goto-Graphs and Shrinking Grammars

The theorem that follows will prove that the operation of growth can be inverted. The graph of a shrinking grammar can be obtained from the initial grammar by removing vertices and edges. Even in this case, split vertices require a special treatment. In the case of the growth operation, we were removing the set  $V_S$  and added in the set  $V'_S$ , that contained all the edges  $\varphi(q)$  such that  $q \in V_S$ . In this case, we will *remove* all those vertices  $p \in V'_S$  that are split in  $\Gamma_G$  and then we will add all the vertices in  $V_S$ . These operations can always be done, because any grammar  $G'$  can be seen as the growing grammar of some  $G = G' \ominus Q$ , for some  $Q$ . In light of Theorem B.1.4 and Remark B.1.1, we can then enunciate the theorem as follows.

**Theorem B.2.1.** Let  $\Gamma_{G'} = \langle V', E' \rangle$ , then there are  $\Delta\bar{V}, V_S, \Delta\bar{E}, E_S$  such that

$$\Gamma_G = \langle V' \setminus \Delta\bar{V} \cup V_S, E' \setminus \Delta\bar{E} \cup E_S \rangle. \quad (\text{B.22})$$

*Proof.* Let be  $\Gamma_{G'} = \langle V', E' \rangle$ . Because of Theorem B.1.4, we know that there are  $\bar{V} \subset V$  and  $\bar{E} \subset E$  such that

$$\langle (V \setminus V_S) \cup \Delta\bar{V}, (E \setminus E_S) \cup \Delta\bar{E} \rangle = \Gamma_{G \oplus Q} = \Gamma_{G' \ominus Q \oplus Q} = \Gamma_{G'}$$

for some sets  $\Delta\bar{V}, \Delta\bar{E}$ . The construction of these sets has been described in the correspondent proof. It is obviously  $V' = (V \setminus V_S) \cup \Delta\bar{V}$ ,  $E' = (E \setminus E_S) \cup \Delta\bar{E}$ . We can then derive the following expressions:

$$V = V' \setminus \Delta\bar{V} \cup V_S, \quad E = E' \setminus \Delta\bar{E} \cup E_S$$

The (B.22) follows. □

For the sake of completeness, we enunciate the following corollary, about the the vertex labels: in this case they *shrink*.

**Corollary B.2.1.** If  $q$  is a vertex of  $\Gamma_{G \ominus Q}$ , then  $\ell_G(q) \supseteq \ell_{G \ominus Q}(q)$ .

The proof of the corollary follows trivially from theorems B.1.6 and B.2.1.



# C

## Variability Model Inference

Appendix A gave formal definitions for the dependencies between *language components*, that were described in Chapter 3. Each dependency can be seen as a *logical constraint*. For instance, when we write  $A \leftarrow BCD$ , this can be read as *A requires B and C and D*; in other words,  $A \rightarrow B \wedge A \rightarrow C \wedge A \rightarrow D$ . Similarly, when we write  $X \leftarrow A$ , what we really mean is that *X requires at least one feature that provides a valid A definition*; thus, if we had two components, respectively providing  $A \leftarrow B$  and  $A \leftarrow C$ , what we *logically* mean, is that *A may be rewritten either to B or C*, that is  $A \rightarrow B \vee A \rightarrow C$ . Besides these simple observations, Neverlang makes it possible to *tag* (Sect. 4.1 and 6.4) language components with additional metadata describing their purpose. In Sect. 6.4 we briefly mentioned that this additional pieces of information can be mined to automatically infer a *variability model*. In this section we give a summary of the experience, that is fully described in [98]; in this paper, the tags are automatically inferred from a *semantic network*, but the algorithm works also for user-provided *tags*. The running example that we use is, again, a family of state machine languages; the implemented components are indicated in Table C.1.

### C.1. Tag Generation

We already have *logical information* (*implies* constraints) between slices, coming from the dependency graph. The implications can be exploited to organize slices into a hierarchy, but are not sufficient: slices are merely symbols. Intuitively user intervention is needed to further refine the meaning of each slice and to hierarchically organize slices. The *semantics* is what is still missing, i.e., the relations that occur between the domain concept that each slice represents.

Slices are lexically mapped onto domain concepts: each slice is associated to a set of terms, or *tags* that describe what it represents conceptually. These sets of tags are

### C. Variability Model Inference

StateChartDef StateChartBody StateDefList	Outer container of the state machine body Body of the state machine List of states
SimpleState	Syntax for simple state
StartState FinalState	Syntax for pseudostate start Syntax for pseudostate final
InnerCompositeStates OuterCompositeStates	Specific definitions for Inner semantics Specific definitions for Outer semantics
MultiTriggerForkDef SingleTriggerForkDef	Syntax for Fork with Multi Trigger Syntax for Fork with Single Trigger
Transition TransitionDefList TransitionAction	Definition of a transition List of transitions Body of a transition
Trigger Guard Effect	Trigger of a transition Guard of a transition Effect of a transition
Join Fork ForkTransition ForkLeftTransition ForkRightTransition	Join pseudostate implementation Fork pseudostate implementation Fork Transition in the Single Trigger case Left Transition in the Multi Trigger case Right Transition in the Multi Trigger case

**Table C.1.:** Slices for the SM language family.

automatically generated from an *initial* set of words. This initial set of words  $T_0$  is given in the modules (Sect. 4.1). These tags can be then *enriched* with additional words coming from a dictionary or a *semantic network* [98]. Now, let be  $W$  a set of words. Then, if slice  $S \in \mathbf{S}$  (where  $\mathbf{S}$  is an arbitrary set of slices), *provides* nonterminal  $X$ , we can define the initial set  $T_0(S) = \{w_0, w_1, \dots, w_n\}$ . By way of a dictionary or a semantic network, we can enrich  $T_0$  to a set of tags  $T(S) \supseteq T_0(S)$ .

## C.2. Hierarchical Clustering

Once tags have been generated for each language component, a relevant tree structure—that will serve as the basis for the variability model—should be derived. The idea is to clusterize our language components by applying an agglomerative hierarchical clustering algorithm [94]. The algorithm generates a binary tree (called *dendogram*).

Specifically, we reasoned by analogy with documental collections where each document can be seen as a set of words. Each word is associated with a frequency  $f$ , i.e., the number of occurrences of the word in the document. Our set of words  $T(S)$  for slice  $S$  can be seen as a document where each word only occurs once. The clustering algorithm works by comparing clusters using a *similarity measure*. A reasonable measure for similarity between slices is the *Jaccard similarity* [94]:

$$J(S_1, S_2) = \frac{|T(S_1) \cap T(S_2)|}{|T(S_1) \cup T(S_2)|} \quad (\text{C.1})$$

For instance, consider Fig. C.1 (the tagged language components are the leaves of



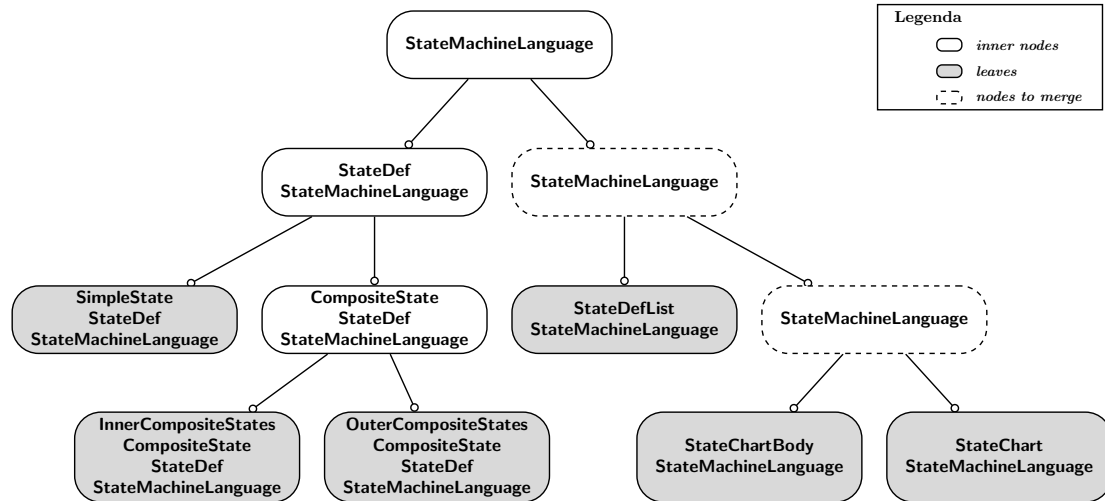


Figure C.1.: (Part of) Dendrogram for the State Machines..

the tree): the Jaccard similarity between  $S$  and  $S'$ , with  $T(S) = \{\text{InnerCompositeStates, CompositeState, StateDef, StateMachineLanguage}\}$  and  $T(S') = \{\text{OuterCompositeStates, CompositeState, StateDef, StateMachineLanguage}\}$  would be  $J(S, S') = .6$ , and the distance can be defined as  $d(S, S') = 1 - J(S, S') = 0.4$ , that is the *complement* of the similarity measure. Within this framework, we can recursively define a cluster as:

1. the singleton set  $\{T(S)\}$ , with  $S$  being a slice
2. the set  $\{c_1, c_2\}$  where  $c_1$  and  $c_2$  are clusters

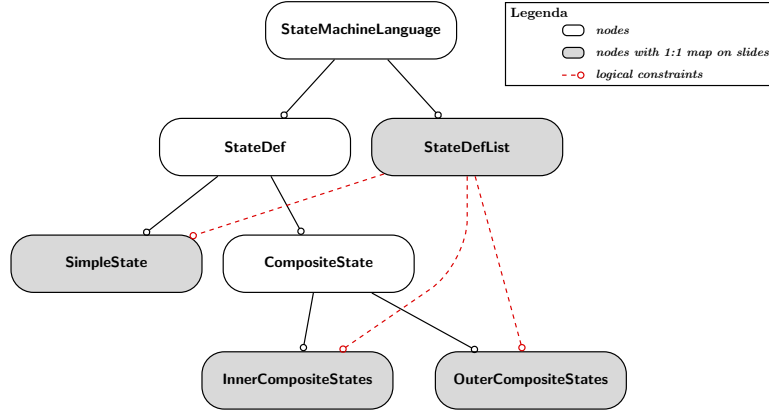
The output of this process is a dendrogram where all the leaves are clusters on one element of the form  $\{T(S)\}$ , that are therefore easily mapped to single slices. Each cluster contains the slices that are closer to each other, with respect to the Jaccard similarity measure. In some sense, then, each cluster contains the slices that are closer to each other *semantically*: in fact, the more two sets  $T(S), T(S')$  overlap, the closer the measure will be to 1. Figure C.1 shows the result of the hierarchical clustering on a small subset of slices for the state machine language example.

### C.3. Refinement Procedure

In this first approximation, the tree (Fig. C.1) has many nodes labeled in the same way and, as a variability model, it is poorly structured. We now described how to merge nodes and compute labels.

**Merging nodes.** To reduce the number of choices, we merge nodes according to the chosen distance measure  $d$ . If the distance between a pair of nodes is zero, then they must be merged. Once the nodes have been merged, they can be labeled with the tags contained in the clusters. The result of the clustering procedure is a binary tree

### C. Variability Model Inference



**Figure C.2.:** Structure of the tree with respect to the features that represent states..

$H = \langle C, E \rangle$ , where  $C$  is a set of clusters and the set of edges is  $E = \{(c, c') \mid c' \in c\}$ . In particular, there is a subset  $C_S \subset C$  that is the collection of 1-element clusters, i.e.,  $C_S = \{\{T(S)\} \mid S \in \mathbf{S}\}$ ;  $C_S$  is the set of the tree *leaves*. For each cluster  $c$  we define:

$$\tau(c) = \begin{cases} T(s) & c \in C_S \\ \tau(c_1) \cap \tau(c_2) & c = \{c_1, c_2\} \end{cases} \quad (\text{C.2})$$

Intuitively, the  $\tau$  function flattens the word sets found in each cluster, and computes their intersection. E.g., consider a cluster  $\bar{c} = \{\{\text{StateChart}, \text{StateMachineLanguage}\}, \{\text{StateChartBody}, \text{StateMachineLanguage}\}\}$ , then  $\tau(\bar{c}) = \{\text{StateMachineLanguage}\}$  (cf. Fig. C.1). The  $\tau$  function and the Jaccard similarity are used to find parent/child pairs that can be merged. For each parent/child pair  $(c, c') \in E$  we compute the similarity value:

$$J(\tau(c), \tau(c'))$$

where  $J$  is still the Jaccard similarity. When the similarity value is 1 (the distance is 0), then parent and child can be merged, i.e., children of  $c'$  may become children of  $c$ , and node  $c'$  may be removed from the tree.

**Labeling nodes.** The result of the merging procedure is a non-binary tree where nodes are still unlabeled. Using again the  $\tau$  function we can now define a labeling strategy  $\ell$ , i.e., the labeling where parent and child labels do not overlap:

$$\ell(c) = \begin{cases} \ell(c) & c \text{ is root} \\ \tau(c) \setminus \ell(c') & (c', c) \in E. \end{cases} \quad (\text{C.3})$$

The result of the entire procedure applied to Fig. C.1 can be seen in Fig. C.2. Red, dashed edges are *implies* constraints that can be inferred from the *dependency graph* (Appendix A.)

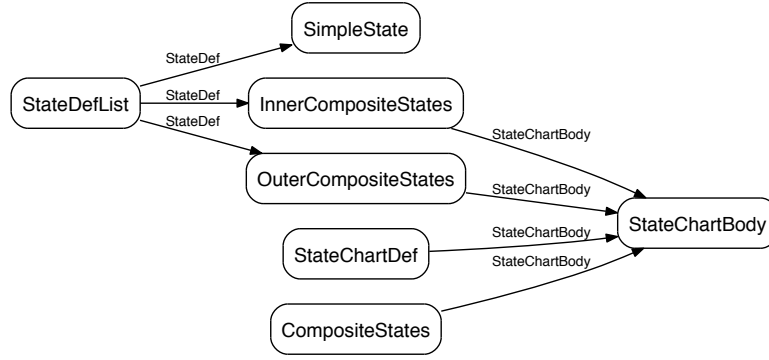


Figure C.3.: The dependency graph for a subset of the state machine language family.

## C.4. Heuristics for Mining Constraints

The dependency graph  $D = \langle \mathbf{P}, E \rangle$  (Def. 10) represents implication relations that we can extract from the syntax in the language components. Let us now consider an alternate definition of dependency graph, and let us call it  $DG$ .

**Definition 18** (Dependency Graph (alternate)). Let be  $D = \langle \mathbf{P}, E \rangle$  the dependency graph between syntactic definition of a language component. Then  $D$  induces a dependency graph  $DG = \langle \mathbf{S}, E_S \rangle$  between *language components*, and  $\mathbf{S}$  is a set of language components,

$$E_S = \{ (S, S') \mid (P, P') \in E, P \text{ is syntax for } S, P' \text{ is syntax for } S' \}.$$

Figure C.3 shows the dependency graph for a subset of the slices that we are considering for the state machine example. There are two additional opportunities for mining other kinds of constraints (beyond binary implications).

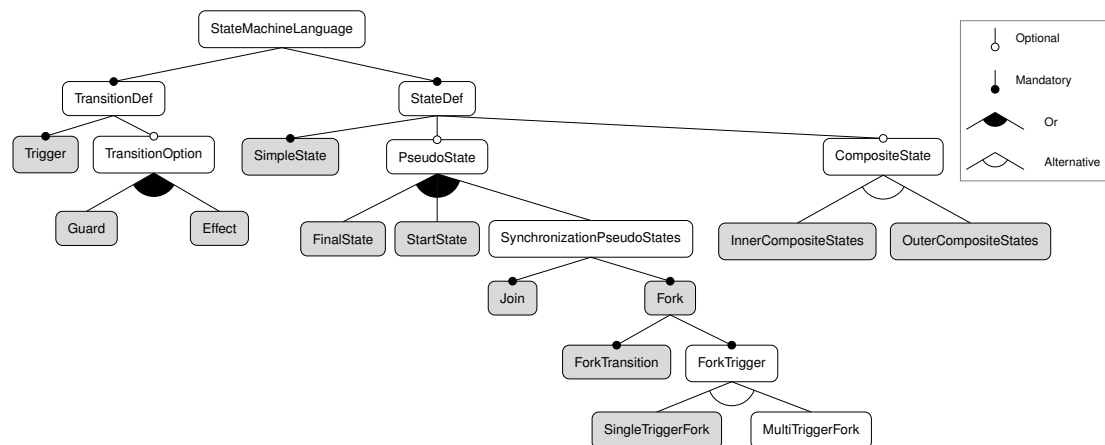
**Implication and disjunctions.** For each vertex  $S$  in the dependency graph  $DG$  there exist a pair  $(S, S') \in DG$  if and only if  $S'$  satisfies a dependencies of  $S$  (Appendix A). In particular,  $S'$  satisfies a dependency of  $S$  when there is some production  $B \rightarrow \beta$  in the syntax definition  $S'$  and  $S$  contains some production  $A \rightarrow \omega B \omega'$ . In this case, we can say that  $S'$  *provides*  $B$  and that  $S$  *requires*  $A$ . The relation between slices  $S, S'$  can be therefore interpreted as the logic constraint:

$$s \rightarrow s' \tag{C.4}$$

However, consider the case when there are multiple slices  $S'_i$  satisfying  $S$ , for  $i = 0, 1, \dots, n$ . For instance, each slice might contain a production of the form  $B \rightarrow \beta_i$ , where  $\beta_0 \neq \beta_1 \neq \dots \neq \beta_n$ . Although one might then be tempted to write the set of formulas:

$$S \rightarrow S'_0, \quad S \rightarrow S'_1, \quad \dots, \quad S \rightarrow S'_n$$

### C. Variability Model Inference



**Figure C.4.:** Final generated variability model. In grey, features that are mapped 1:1 to a slice..

this would be incorrect. In fact, in a grammar, rules of the form  $B \rightarrow \beta_i$  represent *choices* between possible rewritings, rather than constraints that shall hold *at the same time*.<sup>1</sup> Consequently, in this case we can infer the constraint:

$$S \rightarrow (S'_0 \vee S'_1 \vee \dots \vee S'_n) \quad (\text{C.5})$$

For instance, in Fig. C.3, the collection of equally-labeled outbound edges can be expressed using the formula:

$$\begin{aligned} \text{StateDefList} \rightarrow & \quad (\text{C.6}) \\ & (\text{SimpleState} \vee \text{InnerCompositeStates} \vee \text{OuterCompositeStates}). \end{aligned}$$

As a general rule, if there is one and only one slice  $S'$  that satisfies  $S$ , then the logic constraint in (C.4) encodes the mandatory requirement that, when  $S$  is in the language, then also  $S'$  shall be included; otherwise, if there are  $n$  slices  $S'_i$  that satisfy  $S$ , then the logic constraint (C.5) encodes the requirement that, when  $S$  is in the language, *at least one* slice  $S'_i$  shall be included. By reasoning in the same way for each slice in set  $\mathbf{S}$  and for each pair in set  $E_{\mathbf{S}}$ , we obtain a collection of *crosscutting constraints* that we can pair with the tree that we have as a result of the clustering procedure.

**Conflicting components.** Another kind of constraints can be inferred by looking at conflicts between components. So far we did not consider logical *exclusion* between features; we may say that two language features are in conflict if introducing both of them leads to an incorrect language implementation. In particular, two *language components* that define different semantics for the same keyword cannot coexist at the same time in the same language implementation; in this case, the conflict can be

<sup>1</sup>Recall that each production expresses a rewriting of the symbol on the left with all the symbols on the right; the logic constraints above would mean that  $B$  must be rewritten to *every*  $\beta_i$  *at the same time*, which clearly does not make sense.

#### *C.4. Heuristics for Mining Constraints*

detected at the level of the language framework. In the case of Neverlang, we consider that two slices are in conflict when they define the same syntax with different semantics. For further details, see [98].



## Bibliography

- [1] Stephen Robert Adams. *Modular Grammars for Programming Language Prototyping*. Ph.d. thesis, University of Southampton, Southampton, UK, 1991.
- [2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, Reading, Massachusetts, 1986.
- [3] Vander Alves, Christa Schwanninger, Luciano Barbosa, Awais Rashid, Peter Sawyer, Paul Rayson, Christoph Pohl, and Andreas Rummeler. An Exploratory Study of Information Retrieval Techniques in Domain Analysis. In Klaus Pohl and Birgit Geppert, editors, *Proceedings of the 12<sup>th</sup> International Software Product Line Conference (SPLC'08)*, pages 67–76, Limerick, Ireland, September 2008. IEEE.
- [4] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An Overview of Caesar]. *Transaction on Aspect-Oriented Software Development*, 1(1):135–173, March 2006.
- [5] John Aycock. The Design and Implementation of SPARK, a Toolkit for Implementing Domain-Specific Languages. *Journal of Computing and Information Technology*, 10(1):55–66, 2004. Special Issue on Domain Specific Languages.
- [6] John Aycock and R. Nigel Horspool. Directly-Executable Earley Parsing. In Reinhard Wilhelm, editor, *Proceedings of the 10th International Conference on Compiler Construction (CC'01)*, LNCS 2027, pages 229–243, Genova, Italy, April 2001. Springer.
- [7] Jon Bentley. Programming Pearls: Little Languages. *Communications of the ACM*, 29(8):711–721, August 1986.
- [8] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. PACKT Publishing Ltd, August 2013.
- [9] Lorenzo Bettini. Implementing Java-like Languages in Xtext with Xsemantics. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC'13)*, pages 1559–1564, Coimbra, Portugal, March 2013. ACM.
- [10] Lera Boroditsky. How language shapes thought. *Scientific American*, 304(2):62–65, 2011.

## Bibliography

- [11] Claus Brabrand and Michael I. Schwartzbach. The Metafront System: Safe and Extensible Parsing and Transformation. *Science of Computer Programming*, 68:2–20, August 2007.
- [12] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A Language and Toolset for Program Transformation. *Journal Science of Computer Programming*, 72(1-2):52–70, June 2008.
- [13] Martin Bravenboer and Eelco Visser. Parse Table Composition: Separate Compilation and Binary Extensibility of Grammars. In *Software Language Engineering*, LNCS 5452, pages 74–94. Springer, 2009.
- [14] Walter Cazzola. Domain-Specific Languages in Few Steps: The Neverlang Approach. In Thomas Gschwind, Flavio De Paoli, Volker Gruhn, and Matthias Book, editors, *Proceedings of the 11<sup>th</sup> International Conference on Software Composition (SC'12)*, Lecture Notes in Computer Science 7306, pages 162–177, Prague, Czech Republic, 31st of May-1st of June 2012. Springer.
- [15] Walter Cazzola and Davide Poletti. DSL Evolution through Composition. In *Proceedings of the 7<sup>th</sup> ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'10)*, Maribor, Slovenia, on 23rd of June 2010. ACM.
- [16] Walter Cazzola and Ivan Speziale. Sectional Domain Specific Languages. In *Proceedings of the 4<sup>th</sup> Domain Specific Aspect-Oriented Languages (DSAL'09)*, pages 11–14, Charlottesville, Virginia, USA, on 3rd of March 2009. ACM.
- [17] Walter Cazzola and Edoardo Vacchi. DEXTER and Neverlang: A Union Towards Dynamicity. In Eric Jul, Ian Rogers, and Olivier Zendra, editors, *Proceedings of the 7<sup>th</sup> Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'12)*, Beijing, China, 11th of June 2012. ACM.
- [18] Walter Cazzola and Edoardo Vacchi. Neverlang 2: Componentised Language Development for the JVM. In Walter Binder, Eric Bodden, and Welf Löwe, editors, *Proceedings of the 12<sup>th</sup> International Conference on Software Composition (SC'13)*, Lecture Notes in Computer Science 8088, pages 17–32, Budapest, Hungary, 19th of June 2013. Springer.
- [19] Walter Cazzola and Edoardo Vacchi. On the Incremental Growth and Shrinkage of LR Goto-Graphs. *ACTA Informatica*, 51(7):419–447, June 2014.
- [20] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within Modeling Language Definitions. In Andy Schürr and Bran Selic, editors, *Proceedings of the 12<sup>th</sup> International Conference on Model Driven Engineering Languages and Systems (MoDELS'09)*, LNCS 5795, pages 670–684, Denver, CO, USA, October 2009. Springer.



- [21] Lianping Chen and Muhammad Ali Babar. A Systematic Review of Evaluation of Variability Management Approaches in Software Product Lines. *Journal of Information and Software Technology*, 53(4):344–362, April 2011.
- [22] Jean-Marc Davril, Edouard Delfosse, Negaar Hariri, Mathieu Acher, Jane Cleland-Huang, and Patrick Heymans. Feature Model Extraction from Large Collections of Informal Product Descriptions. In Luciano Baresi and Mira Mezini, editors, *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'13)*, pages 290–300, Saint Petersburg, Russia, August 2013. ACM.
- [23] Carl Dea, Mark Heckler, Gerrit Grunwald, José Pereda, and Sean Phillips. *JavaFX 8: Introduction by Example*. Apress, June 2014.
- [24] Edsger Wybe Dijkstra. The Humble Programmer. *Communications of the ACM*, 15(10):859–866, October 1972.
- [25] Stéphanne Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. Traits: A Mechanism for Fine-Grained Reuse. *ACM Transactions on Programming Languages and Systems*, 28(2):331–388, March 2006.
- [26] Jay Earley. An Efficient Context-Free Parsing Algorithm. *Communications of the ACM*, 13(2):94–102, February 1970.
- [27] Sven Efftinge and Markus Völter. oAW xText: A Framework for Textual DSLs. In *Proceedings of the EclipseCon Summit Europe 2006 (ESE'06)*, volume 32, Esslingen, Germany, November 2006.
- [28] John Ellson, Emden Gansner, Lefteris Koutsofios, StephenC. North, and Gordon Woodhull. Graphviz— open source graph drawing tools. In Petra Mutzel, Michael Jünger, and Sebastian Leipert, editors, *Graph Drawing*, volume 2265 of *Lecture Notes in Computer Science*, pages 483–484. Springer Berlin Heidelberg, 2002.
- [29] Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel. Language Composition Untangled. In Anthony Sloane and Suzana Andova, editors, *Proceedings of the 12th Workshop on Language Description, Tools, and Applications (LDTA'12)*, Tallinn, Estonia, March 2012. ACM.
- [30] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. SugarJ: Library-Based Syntactic Language extensibility. In *Proceedings of the 26th ACM SIGPLAN Conference on Object-Oriented Programming (OOPSLA'11)*, pages 391–406, Portland, Oregon, USA, October 2011. ACM.
- [31] Sebastian Erdweg and Felix Rieger. A Framework for Extensible Languages. In Christian Kästner, editor, *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences (GPCE'13)*, pages 3–12, Indianapolis, IN, USA, October 2013. ACM.

## Bibliography

- [32] Rodney Farrow, Thomas J. Marlowe, and Daniel M. Yellin. Composable Attribute Grammars: Support for Modularity in Translator Design and Implementation. In Ravi Sethi, editor, *Proceedings of the 19th ACM Symposium on Principles of Programming Languages (POPL'92)*, pages 223–234, Albuquerque, NM, USA, January 1992. ACM.
- [33] Alessio Ferrari, Giorgio O. Spagnolo, and Felice Dell'Orletta. Mining Commonalities and Variabilities from Natural Language Documents. In *Proceedings of the 17th International Software Product Line Conference (SPLC'13)*, pages 116–120, Tokyo, Japan, September 2013. ACM.
- [34] Bryan Ford. Parsing Expression Grammars: a Recognition-Based Syntactic Foundation. In Neil D. Jones and Xavier Leroy, editors, *Proceedings of the 31<sup>st</sup> ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'04)*, pages 111–122, Venice, Italy, January 2004. ACM.
- [35] Martin Fowler. Fluent Interface. Martin Fowler's Blog, May 2005.
- [36] Martin Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? Martin Fowler's Blog, May 2005.
- [37] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, Reading, Ma, USA, 1995.
- [38] Debasish Ghosh. *DSLs in Action*. Manning Publications Co., December 2010.
- [39] Rosa Gini, Massimo Coppola, Patrick B. Ryan, Giacomo Righetti, Iacopo Peri, Roberto Berni, Paul Avillach, Preciosa M. Coloma, Gianluca Trifirò, Gayo D'Allo, Johan van der Lei, Miriam C.J.M. Sturkenboom, and Martijn J. Schuemie. Frameworks for Data Extraction and Management from Electronic Healthcare Databases for Multi-Center Epidemiologic Studies: a Comparison among EU-ADR, MATRICE, and OMOP Strategies. In *Proceedings of the 24th European Medical Informatics Conference (MIE'12)*, Pisa, Italy, August 2012.
- [40] James Gosling, Bill Joy, Guy L. Steele, Jr, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition*. Pearson Education, June 2014.
- [41] Paul Graham. Beating the averages, April 2003.
- [42] Dick Grune and Ceriel J. H. Jacobs. *Parsing Techniques*. Monographs in Computer Science. Springer, second edition, 2008.
- [43] Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, Gøran K. Olsen, and Andreas Svendsen. Adding Standardized Variability to Domain Specific Languages. In Klaus Pohl and Birgit Geppert, editors, *Proceedings of the 12<sup>th</sup> International Software Product Line Conference (SPLC'08)*, pages 139–148, Limerick, Ireland, September 2008. IEEE.

- [44] Görel Hedin and Eva Magnusson. JastAdd — An Aspect-Oriented Compiler Construction System. *Science of Computer Programming*, 47(1):37–58, April 2003.
- [45] Jan Heering, Paul R. H. Hendricks, Paul Klint, and Jan Rekers. The Syntax Definition Formalism SDF —Reference Manual—. *SIGPLAN Notices*, 24(11):43–75, November 1989.
- [46] Jan Heering, Paul Klint, and Jan Rekers. Incremental Generation of Parsers. *IEEE Transactions on Software Engineering*, 16(12):1344–1351, 1990.
- [47] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Model-Based Language Engineering with EMFText. In Ralf Lammel, João Saraiva, and Joost Visser, editors, *Proceedings of the International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'11)*, LNCS 7680, pages 322–345, Braga, Portugal, July 2011. Springer.
- [48] R. Nigel Horspool. Incremental Generation of LR Parsers. *Journal of Computer Languages*, 15(4):205–223, 1990.
- [49] Arnaud Hubaux, Andreas Classen, Marcílio Mendonça, and Patrick Heymans. A Preliminary Review on the Application of Feature Diagrams in Practice. In David Benavides, Don S. Batory, and Paul Grünbacher, editors, *Proceedings of the 4<sup>th</sup> International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'10)*, pages 53–59, Linz, Austria, January 2010.
- [50] Christian Humer, Christian Wimmer, Christian Wirth, Andreas Wöß, and Thomas Würthinger. A domain-specific language for building self-optimizing ast interpreters. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences, GPCE 2014*, pages 123–132, New York, NY, USA, 2014. ACM.
- [51] Graham Hutton. High-Order Functions for Parsing. *Journal of Functional Programming*, 2(3):323–343, 1992.
- [52] Richard K. Jullig and Frank DeRemer. Regular right-part attribute grammars. *SIGPLAN Not.*, 19(6):171–178, June 1984.
- [53] Tomas Kalibera, Petr Maj, Floreal Morandat, and Jan Vitek. A Fast Abstract Syntax Tree Interpreter for R. In Erez Petrank and Dan Tsafir, editors, *Proceedings of the 10th International Conference on Virtual Execution Environments (VEE'14)*, pages 89–102, Salt Lake City, UT, USA, March 2014. ACM.
- [54] Ted Kaminski and Eric Van Wyk. Creating and Using Domain-Specific Language Features. In Benoît Combemale, Walter Cazzola, and Robert B. France, editors, *Proceedings of the 1st Workshop on the Globalization of Domain Specific Languages (GlobalDSL'13)*, pages 18–21, Montpellier, France, July 2013. ACM.

## Bibliography

- [55] Ted Kaminski and Eric Van Wyk. Modular Well-Definedness Analysis for Attribute Grammars. In Krzysztof Czarnecki and Görel Hedin, editors, *Proceedings of the 5th International Conference on Software Language Engineering (SLE'13)*, Lecture Notes in Computer Science 7745, pages 352–371, Dresden, Germany, September 2013. Springer.
- [56] Lennart C. L. Kats and Eelco Visser. The Spoofox 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, Reno, Nevada, USA, October 2010. ACM.
- [57] Donald E. Knuth. On the Translation of Languages from Left to Right. *Information and Control*, 8(6):607–639, December 1965.
- [58] Donald E. Knuth. Semantics of Context-Free Languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.
- [59] Holger Krahn, Bernhard Rumpe, and Steven Völkel. MontiCore: A Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer*, 12(5):353–372, September 2010.
- [60] Jörg Liebig, Rolf Daniel, and Sven Apel. Feature-Oriented Language Families: A Case Study. In Philippe Collet and Klaus Schmid, editors, *Proceedings of the 7<sup>th</sup> International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'13)*, Pisa, Italy, January 2013. ACM.
- [61] Stephen J. Mellor, Stephen Tockey, Rodolphe Arthaud, and Philippe Leblanc. An Action Language for UML: Proposal for a Precise Execution Semantics. In Jean Bézivin and Pierre-Alain Muller, editors, *Proceedings of the first Workshop on The Unified Modeling Language («UML»'98)*, LNCS 1618, pages 307–318, Mulhouse, France, June 1998. Springer.
- [62] Tom Mens and Michael Wermelinger. Separation of Concerns for Software Evolution. *Journal of Maintenance and Evolution*, 14(5):311–315, 2002.
- [63] Marjan Mernik. An Object-Oriented Approach to Language Compositions for Software Language Engineering. *Journal of Systems and Software*, 86(9):2451–2464, September 2013.
- [64] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and How to Develop Domain Specific Languages. *ACM Computing Surveys*, 37(4):316–344, December 2005.
- [65] Marjan Mernik and Viljem Žumer. Incremental Programming Language Development. *Computer Languages, Systems and Structures*, 31(1):1–16, April 2005.

- [66] Adriaan Moors, Frank Piessens, and Martin Odersky. Parser Combinators in Scala. CW Report 491, Katholieke Universiteit Leuven, Leuven, Belgium, February 2008.
- [67] Nan Niu and Steve Easterbrook. On-Demand Cluster Analysis for Product Line Functional Requirements. In Klaus Pohl and Birgit Geppert, editors, *Proceedings of the 12<sup>th</sup> International Software Product Line Conference (SPLC'08)*, pages 87–96, Limerick, Ireland, September 2008. IEEE.
- [68] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An Extensible Compiler Framework for Java. In *Proceedings of the 12<sup>th</sup> International Conference on Compiler Construction (CC'03)*, LNCS 2622, pages 138–152, Warsaw, Poland, April 2003. Springer.
- [69] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Aritma Press, 2008.
- [70] Bruno C. d. S. Oliveira. Modular Visitor Components: A Practical Solution to the Expression Families Problem. In Sophia Drossopoulou, editor, *Proceedings of the 23<sup>rd</sup> European Conference on Object-Oriented Programming (ECOOP'09)*, Lecture Notes in Computer Science 5653, pages 269–293, Genoa, Italy, July 2009. Springer.
- [71] Bruno C. d. S. Oliveira, Tijs van der Storm, Alex Loh, and William R. Cook. Feature-Oriented Programming with Object Algebras. In Giuseppe Castagna, editor, *Proceedings of the 27th European Conference on Object-Oriented Programming (ECOOP'13)*, Lecture Notes in Computer Science 7920, pages 27–51, Montpellier, France, July 2013. Springer.
- [72] Cyrus Omar, Darya Kurilova, Ligia Nistor, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Safely composable type-specific languages. In Richard Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, volume 8586 of *Lecture Notes in Computer Science*, pages 105–130. Springer Berlin Heidelberg, 2014.
- [73] Emmanuel Onzon. *dypgen User's Manual*, March 2012.
- [74] Jukka Paakki. Attribute Grammar Paradigms: A High-Level Methodology in Language Implementation. *ACM Computer Survey*, 27(2):196–255, June 1995.
- [75] David L. Parnas. On the Criteria to Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.
- [76] Terence J. Parr and Russell W. Quong. ANTLR: A Predicated-LL(k) Parser Generator. *Software—Practice and Experience*, 25(7):789–810, July 1995.
- [77] Klaus Pohl and Andreas Metzger. Variability Management in Software Product Line Engineering. In Leon J. Osterwell, H. Dieter Rombach, and Mary Lou Soffa, editors, *Proceedings of the 28<sup>th</sup> International Conference on Software Engineering (ICSE'06)*, pages 1049–1050, Shanghai, China, May 2006. ACM.

## Bibliography

- [78] Mark Pollack, Oliver Gierke, Thomas Risberg, Jon Brisbin, and Michael Hunger. *Spring Data*. O'Reilly Media, Incorporated, October 2012.
- [79] Eric S Raymond. How to become a hacker. *Database and Network Journal*, 33(2):8–9, 2003.
- [80] Damijan Rebernak, Marjan Mernik, Hui Wu, and Jeff G. Gray. Domain-Specific Aspect Languages for Modularising Crosscutting Concerns in Grammars. *IET Software*, 3(3):184–200, June 2009.
- [81] Lukas Renggli, Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. Practical Dynamic Grammars for Dynamic Languages. In *Proceedings of the 4<sup>th</sup> Workshop on Dynamic Languages and Applications (DYLA'10)*, Málaga, Spain, June 2010.
- [82] Tiark Rompf and Martin Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *Proceedings of the 9<sup>th</sup> International Conference on Generative Programming and Component Engineering (GPCE'10)*, pages 127–136, Eindhoven, The Netherlands, October 2010. ACM Press.
- [83] João Saraiva, S. Doaitse Swierstra, and Matthijs Kuiper. Functional Incremental Attribute Evaluation. In David A. Watt, editor, *Proceedings of the 9th International Conference on Compiler Construction (CC'00)*, Lecture Notes in Computer Science 1781, pages 279–294, Berlin, Germany, March 2000. Springer.
- [84] Nathanael Schärli. *Traits — Composing Classes from Behavioral Building Blocks*. Phd thesis, Universität Bern, Bern, Switzerland, February 2005.
- [85] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable Units of Behaviour. In Luca Cardelli, editor, *Proceedings of the 17<sup>th</sup> European Conference on Object-Oriented Programming (ECOOP'03)*, Lecture Notes in Computer Science 2743, pages 248–274, Darmstadt, Germany, July 2003. Springer.
- [86] August C. Schwerdfeger and Eric R. Van Wyk. Verifiable Parse Table Composition for Deterministic Parsing. In Mark G. J. van den Brand, Dragan Gasevic, and Jeffrey G. Gray, editors, *Proceedings of the 2<sup>nd</sup> International Conference on Software Language Engineering (SLE'09)*, LNCS 5969, pages 184–203, Dublin, Ireland, June 2009. Springer.
- [87] Pat Shaughnessy. *Ruby under a Microscope: An Illustrated Guide to Ruby Internals*. No Starch Press, Incorporated, November 2013.
- [88] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wąsowski, and Krzysztof Czarnecki. Reverse Engineering Feature Models. In Harald Gall and Nenad Medvidović, editors, *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, pages 461–470, Waikiki, Honolulu, Hawaii, May 2011. IEEE.

- [89] Tim Sheard and Simon Peyton Jones. Template Meta-Programming for Haskell. *ACM Sigplan Notices*, 37(12):60–75, December 2002.
- [90] Lukas Stadler, Christian Wimmer, Thomas Würthinger, Haspeter Mössenböck, and John Rose. Lazy Continuations for Java Virtual Machines. In Christian W. Probst, editor, *Proceedings of the 7th International Symposium on Principles and Practice of Programming in Java (PPPJ'09)*, pages 143–152, Calgary, Alberta, Canada, August 2009. ACM.
- [91] Paul Stansifer and Mitchell Wand. Parsing Reflective Grammars. In Claus Brabrand and Eric Van Wyk, editors, *Proceedings of the 11th Workshop on Language Descriptions, Tools and Applications (LDTA'11)*, pages 10:1–10:7, Saarbrücken, Germany, March 2011. ACM.
- [92] Dave Steinberg, Dave Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, December 2008.
- [93] S. Doaitse Swierstra. Combinator Parsers: From Toys to Tools. *Electronic Notes in Theoretical Computer Science*, 41(1):38–59, January 2000.
- [94] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Addison-Wesley, Reading, MA, USA, March 2006.
- [95] Masaru Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1985.
- [96] Laurence Tratt. Domain Specific Language Implementation Via Compile-Time Meta-Programming. *ACM Transaction on Programming Languages and Systems*, 30(6):31:1–31:40, October 2008.
- [97] Laurence Tratt. Evolving a DSL Implementation. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Proceedings of the International Summer School on Generative and Transformational Techniques in Software Engineering II (GTTSE'07)*, LNCS 5235, pages 425–441, Braga, Portugal, April 2008. Springer.
- [98] Edoardo Vacchi, Walter Cazzola, Benoît Combemale, and Mathieu Acher. Automating Variability Model Inference for Component-Based Language Implementations. In Patrick Heymans and Julia Rubin, editors, *Proceedings of the 18th International Software Product Line Conference (SPLC'14)*, pages 167–176, Florence, Italy, 15th–19th of September 2014. ACM.
- [99] Edoardo Vacchi, Walter Cazzola, Suresh Pillay, and Benoît Combemale. Variability Support in Domain-Specific Language Development. In Martin Erwig, Richard F. Paige, and Eric van Wyk, editors, *Proceedings of 6<sup>th</sup> International Conference on Software Language Engineering (SLE'13)*, Lecture Notes on Computer Science 8225, pages 76–95, Indianapolis, USA, 27th–28th of October 2013. Springer.

## Bibliography

- [100] Edoardo Vacchi, Diego Mathias Olivares, Albert Shaqiri, and Walter Cazzola. Neverlang 2: A Framework for Modular Language Implementation. In *Proceedings of the 13th International Conference on Modularity (Modularity'14)*, pages 23–26, Lugano, Switzerland, 22nd–25th of April 2014. ACM.
- [101] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: an Extensible Attribute Grammar System. *Science of Computer Programming*, 75(1-2):39–54, January 2010.
- [102] Eric Van Wyk and August Schwerdfeger. Context-Aware Scanning for Parsing Extensible Languages. In Charles Consel and Julia L. Lawall, editors, *Proceedings of the 6th International Conference on Generative Programming and Component Engineering (GPCE'07)*, pages 63–72, Salzburg, Austria, October 2007. ACM.
- [103] Eelco Visser. Separation of concerns in language definition. In *Proceedings of the Companion Publication of the 13th International Conference on Modularity, MODULARITY '14*, pages 1–2, New York, NY, USA, 2014. ACM.
- [104] Eelco Visser and Zine-el-Abidine Benaissa. A Core Language for Rewriting. *Electronic Notes in Theoretical Computer Science*, 15:422–441, September 1998.
- [105] Markus Völter and Vaclav Pech. Language Modularity with the MPS Language Workbench. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*, pages 1449–1450, Zürich, Switzerland, June 2012. IEEE.
- [106] Philip Wadler. The Expression Problem. Java Genericity Mailing List, November 1998.
- [107] Martin P. Ward. Language Oriented Programming. *Software - Concept and Tools*, 15(4):147–161, 1994.
- [108] Alessandro Warth. *Experimenting with Programming Languages*. PhD thesis, University of California at Los Angeles, Los Angeles, CA, USA, 2009.
- [109] Nathan Weston, Ruzanna Chitchyan, and Awais Rashid. A Framework for Constructing Semantically Composable Feature Models from Natural Language Requirements. In John mc Gregor and Dirk Muthig, editors, *Proceedings of the 13th International Software Product Line Conference (SPLC'09)*, pages 211–220, San Francisco, CA, USA, August 2009. ACM.
- [110] Jules White, James H. Hill, Jeff Gray, Sumant Tambe, Aniruddha Gokhale, and Douglas C. Schmidt. Improving Domain-specific Language Reuse with Software Product-line Configuration Techniques. *IEEE Software*, 26(4):47–53, July-August 2009.
- [111] Thomas Würthinger, Christian Wimmer, Andreas Woß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to Rule Them All. In Robert Hirschfeld, editor, *Proceedings of the 2013*



*ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!'13)*, pages 187–204, Indianapolis, IN, USA, October 2013. ACM.

- [112] Mathias Zenger and Martin Odersky. Independently Extensible Solutions to the Expression Problem. In *Proceedings of the 12th International Workshop on Foundations of Object-Oriented Languages (FOOL'12)*, Long Beach, CA, USA, January 2005.