Computer Languages, Systems & Structures 43 (2015) 1–40

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

Neverlang: A framework for feature-oriented language development

Edoardo Vacchi, Walter Cazzola *

Department of Computer Science, Università degli Studi di Milano, Italy

ARTICLE INFO

Article history:
Received 6 October 2014
Received in revised form
7 February 2015
Accepted 11 February 2015
Available online 20 February 2015

Keywords:
Domain specific languages
Language development
Modularity

ABSTRACT

Reuse in programming language development is an open research problem. Many authors have proposed frameworks for modular language development. These frameworks focus on maximizing code reuse, providing primitives for componentizing language implementations. There is also an open debate on combining feature-orientation with modular language development. Feature-oriented programming is a vision of computer programming in which features can be implemented separately, and then combined to build a variety of software products. However, even though feature-orientation and modular programming are strongly connected, modular language development frameworks are not usually meant primarily for feature-oriented language definition. In this paper we present a model of language development that puts feature implementation at the center, and describe its implementation in the Neverlang framework. The model has been evaluated through several languages implementations: in this paper, a state machine language is used as a means of comparison with other frameworks, and a JavaScript interpreter implementation is used to further illustrate the benefits that our model provides.

1. Introduction

The problem of extending programming languages through new constructs has never lost interest in the industry and in the research community. Modern general purpose programming languages are *multiparadigm*, 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 with pure object-orientation in mind nowadays tend to include functional constructs. This tendency to contamination between different programming styles can be read as the symptom of a need for more flexibility.

Traditionally, the design and implementation of a programming language is more of a *top-down* activity, where most of the time is spent on the design of a consistent set of features; *extensibility of the compiler*, although desirable, is not a strict requirement. But when today people speak about *language development*, they often mean developing a new programming language with *specific requirements* in mind. We could even dare to say that *problem-tailored* programming language development is more of a *bottom-up* activity, because, in some sense, the language specification *rises* from the problem that the developers are trying to solve. Intuitively, a top-down design phase is still important, because it is important that the

* Corresponding author.
E-mail addresses: vacchi@di.unimi.it (E. Vacchi), cazzola@di.unimi.it (W. Cazzola).

http://dx.doi.org/10.1016/j.cl.2015.02.001
1477-8424/

language consists of a coherent set of features; but in a domain-specific language (DSL [1–3]), this phase can be often reduced to a minimum: even more so, if it were possible to implement new languages using off-the-shelf components.

A technique to implement languages is *embedding*; this technique is part of the idiom of many modern programming languages such as Scala, Ruby, and Groovy which, in some sense, are following the lead of veterans such as LISP, the "*programmable programming language*" [4], and Smalltalk. *Embedded* DSLs [5] are really just a byproduct of choosing a particular API design style that Fowler and Evans dubbed a *fluent interface* [6]. Fluent APIs are often used to embed query languages within the body of a general purpose programming language (cf. Spring Data's Query DSL [7]) or to describe graphical user interfaces (cf. JavaFX's APIs [8]).[1] This technique has clear benefits: first of all it is easy to implement; second, it guarantees a high-degree of code reuse, because an embedded language is just a *library*. The main limit is that the expressivity of the language is inevitably dictated by the *host* programming language. *External DSLs*, on the other hand, are instead usually developed using dedicated toolsets, and work as stand-alone programming languages. The traditional route to external DSL development is to implement the front-end through parser generators such as good old `yacc`, ANTLR [11] or, more recently, parser combinators [12,13], and then implementing the semantics of the language. For this purpose, the variety of techniques ranges from attribute grammars [14,15] to simple syntax-directed translation [16] to term-rewriting [17]. The object of our research of the last few years 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.

In fact, empiric evidence shows that many general-purpose languages share similar syntax and similar semantics for the same concepts: for instance, C-like programming languages such as C++, Java, and C# etc. all share a similar syntax for `for` loops, if branches, variable declarations, etc. 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 our own framework, called 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 the model 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: Section 2 gives a brief overview of the background information. Section 3 presents the abstract model. Section 4 introduces the Neverlang implementation of this model. Section 5 presents a full example (a state machine language). Section 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, Section 7 briefly discusses related work and Section 8 draws the conclusions and describes the future work.

2. Background

A *context-free* grammar is a formal grammar where production rules are written as $A \to \omega$ where A is a nonterminal, and ω is a word of terminals and nonterminal.s. The generated language $L(G)$ of a grammar G is the set of all the words that can be *derived* from G. $L(G)$ is *empty* if $L(G) = \varnothing$ and, conversely, non-empty when it contains at least one word. In the following we will assume grammars that generate non-empty languages, and, although it is allowed in Neverlang, for simplicity, we will make the assumption that our grammars do not contain the empty word ε.

A *syntax-directed definition* [16] (SDD) is a technique to implement the semantics of context-free languages, in terms of their grammar. *Attribute grammars* [14] 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* an 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 [16].

[1] Literature has also shown how to support true language embedding through library-based, possibly type-driven language preprocessing [9,10].

Typically, SDTs are implemented *during parsing*, without building a parse tree. In this case, two important classes of grammars are [16]

- *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* [14,15] 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 [15]. 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. [18]). 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 Section 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 also be described in terms of program *transformations*; the Stratego [19] 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, Section 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 Section 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., [19–23]) 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., [24]) 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 DSLs, 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. [23]). As seen in Section 2, the earliest literature already established that languages can be described in terms of their syntax. It has also been shown (e.g., [25,26]) 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.

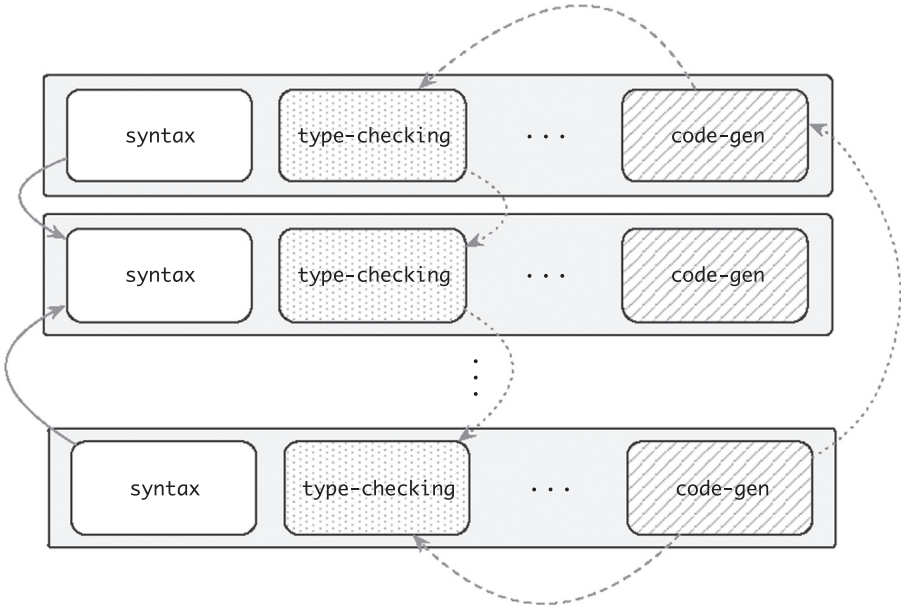


Fig. 1. A language implementation can be broken down over two dimensions: the dimension of syntactic constructs and the dimension of evaluation phases. Dependencies are represented as arrows from *required* to *provided* features. Dependencies may go beyond their phase and depend on other phases.

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. 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. 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 features of a language are the points at the *intersections* between the two dimensions. The *implementation* of a feature is what we call a *language component* [27,28].

Language components: *Modular language implementation approaches* enforce separation between *processing phases* and *concepts*. The dimension of processing phases represents the *separation* between *linguistic concerns* [29] 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. 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, consider a simple imperative language with a Java-like `while` loop construct[2]:

```
syntax:
      while ( ⟨ loop-condition ⟩ ) {
        ⟨ loop-body ⟩
      }
```

[2] 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.

Table 1
Summary of the informal definitions in this section.

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 *placeholders*
Syntactic placeholder	A part of a syntax definition that is not defined in place. It constitutes a reference to a *class* of syntax definitions
Evaluation phase	Definition of the semantics of the language with respect to a particular concern (e.g., type-checking and code-generation) in terms of *properties* 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 *feature* of the language by putting in relation a *syntax definition* with the relevant parts of the *evaluation phases*. In this sense, we can say that a language component *provides* the implementation of a feature. The language component may also *require* other language features to work
Dependency	A feature that is *required* by a component, and that it is not defined within that same component. A dependency in a *language* is *unsatisfied* if there is a component that *requires* a feature that no components *provide*. A dependency may be *syntactic* if it is expressed within a *syntax definition* by *placeholders*, or *semantic* if it is expressed in an evaluation phase by *properties*
Globally scoped component	A component that implements a concern that should be available to any *language component*
Language implementation	A collection of *language components* where every dependency is satisfied

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.

A language is usually processed through several phases, such as *type checking* to verify the correctness of the construct and *code generation* to output compiled code. Initially, we assumed that it is possible to modularize phases by breaking them down with respect to the syntax definitions that pertain to a given language construct; thus, we can componentize these phases with respect to the given looping constructs:

```
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)
```

The example contains some bold-face words; these represent *properties* of the language that are being evaluated during the evaluation phase. The `type-checking` phase evaluates a *property* named **`type-of`**. This property is bound to a 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 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 such *phases* together with the syntax definition of the `while` loop yields *language component*. 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 implementations with similar requirements in terms of features and processing phases, maximizing *reuse* of the feature, and minimizing code *duplication*. For instance, a different bundle, may reuse the same syntax, with variated semantics, to implement an alternate version of the same language. *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. The same semantics

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

A *language implementation* can be therefore described by a set of *language 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*.

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 *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. 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.

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*. 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 exists a component in the language implementation that *provides* that feature.

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*. 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, and variable increment): each statement is not logically in conflict with the other, although they may appear in the same position in a program written 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. [25] on attribute grammars), because a property in a phase may be only referred to within the *same* phase or a *subsequent* phase.

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 and function tables). 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 [30]), by swapping the component that implements the task execution model.

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. [9] 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) [31, Section 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" [9], 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. [9] 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 [22], the model itself does not prevent from implementing a self-extensible programming language Table 1.

4. Neverlang

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* programming 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 variate the number of cases in a data type definition. The situation is known as *the expression problem*, after the term coined by Philip Wadler [32].

It has been shown that a modular implementation of the visitor and interpreter patterns [33–35] can be achieved using constructs such as *traits* [36] 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 Section 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

Listing 1. Complete EBNF grammar of the *Neverlang language*.

```
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 ("," 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+ Mapping?;
  Mapping ⇐ "mapping" "{" ( MappingDef ("," MappingDef)* )  "}"
  MappingDef ⇐ Integer " ⇒ " Integer;

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

// language
Language ⇐ "language" QualifiedId "{" LangSlices LangEndemic LangRoles LangRenames "}" ;
  LangSlices ⇐ "slices" ( QualifiedId | "bundle" "(" QualifiedId ")" )+ ;
  LangEndemic ⇐ "endemic" "slices" QualifiedId+ ;
  LangRoles ⇐ "roles" "syntax" ( LangVisitOp Id (LangVisitOpFull Id)* )? ;
  LangVisitOp ⇐ "<" | "<" "+"  ;
  LangVisitOpFull ⇐ 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 ⇐ [A-Za-z_-][A-Za-z0-9_-]* ;
Label ⇐ Id
SimpleTerminal ⇐ <quoted string> ;
RegexTerminal ⇐ <perl-like regex literal> ;
Integer ⇐ <integer number> ;
CodeBlock ⇐ <parsing delegated to translator plugins> ;
```

In Section 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 (Section 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* (Section 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 DSL 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 Section 4.1.1. The generated source code will be described in Section 4.3, 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 1 is the full grammar of the Neverlang language (EBNF operators were used for conciseness).

4.1.1. Defining syntax and semantics: modules

Listing 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 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 3. **`reference syntax`** for a Pascal-style `while` statement.

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

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 (Section 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*. The empty string "" denotes the empty word ε. 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 Section 3 the `while` loop could have been defined with a Java-like syntax, using braced blocks (Listing 2), or using a Pascal-like syntax (Listing 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 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*.

The *reference syntax* section contains a list of production rules between braces (see Listing 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 declares that the reference syntax definition is the one in module `javalang.WhileLoop` (Listing 2).

Listing 4. An example code generation role, generating Java bytecode in Jasmin syntax.[3]

```
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
      ''';
    }.
  }
}
```

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

[3] http://jasmin.sourceforge.net

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 5 shows an example for the `while` loop. A use case for tags will be discussed in Section 6.4.

Listing 5. **`provides`** and **`requires`** sections.

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

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 (Section 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 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`.[4] 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*[5] 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 (Listings 2 and 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.

The choice of *implicit* declaration is a convenience that is retained for compatibility with Neverlang's previous version. This is of course a double-edged sword, because users may mistype attribute names and obtain an exception at runtime. Work is being done to support explicit declaration of attribute names and their types.

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 6). We will see more on how actions are compiled in Section 4.3.

[4] 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.

[5] E.g., consider grammar **A**◃—**B**; **C**◃—**D**: rules 0, 1 may refer either 0, 1, but not 2; etc.

Listing 6. Using multiple languages in a module.

```
<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
    }.
  }
}
```

Labels: The reasons for the choice of a numbering scheme instead of a naming convention to indicate syntax definitions are mostly a matter of history. Neverlang's original implementation [37,38] followed the same convention, which was inspired 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 2 shows that the rule could be defined for label `While:`, which would then be resolved by the Neverlang compiler `nlgc` (Section 4.1.1) 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 are 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 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 and refactoring). Moreover, as we will see in Section 4.3.1, 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 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) 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.

Listing 7. Syntactic sugar to execute a post-order visit.

```
// 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 = ...
}.
```

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 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" [16]. 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* in the **`language`** descriptor (see Listing 12 and Section 4.1.3). You may also have noticed (Listing 4) that Neverlang's Java code blocks provide a special extended syntax for multi-line strings, deliberately reminiscent of Xtend's *template expressions* [39].

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 `$terminate`. To raise an error, a Java `RuntimeException` or an `Error` can be thrown as usual (e.g., see Listing 2). For more information on the implementation of the `$terminate` command see Section 4.3.2.

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 (cf. Linda-Python in [30]).

4.1.2. Mapping semantics onto syntax: slices

Listing 8. Slice implementing a bytecode-generating `while` loop feature for the Java language.

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

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 8, the `javalang.WhileLoopSlice` is being defined. The *reference syntax* from the `javalang.WhileLoop` module (Listing 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 2) and the `code-gen` role in the `javalang.WhileLoopCodeGen` module (Listing 4). Nonetheless, the same roles could still apply to the reference syntax in module `pascallang.WhileLoop` (Listing 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 [20]; the *Recipe* DSL is a language inspired by Microsoft's on{X},[6] 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*.

Listing 9. Remapping part of the `while` implementation onto the `do-while` syntax.

```
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 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.

```
// 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, ...
  }
}
```

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 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

[6] http://onx.ms

`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, see Section 4.3— 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 form of an experimental semantic action DSL; we will return on this later in Section 6.2.

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 `UnaryExpr⇐PostfixExpr;CastExpr⇐UnaryExpr;…` (Listing 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 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⇐B;B⇐C;C⇐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:

`role``(r){0.{$`**`0`**`.value=$`**`1`**`.value;}.}`

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⇐C` action `0`, with `B` as `A`, and `C` as `B`; similarly, on rule `C⇐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.

Listing 11. Endemic slice providing a `SymbolTable` interface with an implementation.

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

Endemic slices: In Section 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 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 [30], 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 `nlgi` tool (Section 4.3).

Listing 12. Extract of the **`language`** descriptor for a `Java` compiler.

```
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;
  }
}
```

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 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. Fig. 2 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 (Section 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 [21,30,38], a language for log rotating, similar to the UNIX `logrotate` utility. In this language, each line is a log management operation (e.g., `rename` and `backup`). 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 Section 3, each language component has *dependencies*. These dependencies should be satisfied when the components are combined together (Section 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 2 and 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 `$1`, which, in this case, resolves to `LoopCondition`
- the `code-gen` role *requires* the `String` attribute `code` to be defined on nonterminals `LoopCondition` and `LoopBody`, respectively, and *provides* a `String` attribute `code` on nonterminal `$⊙`, which in this case resolves to `WhileLoop`.

Section 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 [27,28] we explored ways to track and resolve dependencies automatically, and present them to end users in a convenient way (see also Section 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 Section 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 12, the `LoopCondition` is renamed to `Expression` in every production in which it occurs, causing, for instance, the production

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

in `javalang.WhileSlice` to become

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

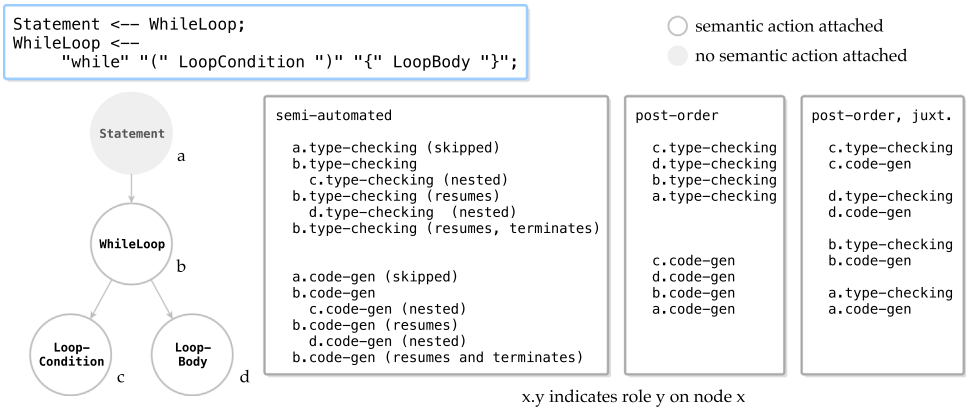


Fig. 2. Role execution order: the grayed node does not have semantic actions attached. For the sake of the example, we assume that all roles are evaluated in the same way (semi-automated or post-order).

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

Listing 13. `javalang.Lang` using bundles.

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

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 and all the slices that implement the conditional branches. 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 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. 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.

Listing 14. Translator Plugin for Java.

```
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});";
    ...
  }
}
```

The Neverlang Compiler nlgc: In Section 4.3 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*. 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 Section 4.1, we described how to write a `while` loop in Neverlang, and we have shown how to write a module (Listing 2), a slice (Listing 8) and a language descriptor (Listing 12). 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 15. The Neverlang language allows semantic actions to be written using a custom language as well, while still providing useful 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. In Section 4.1.1, Listing 6 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 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.[7] 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 a separate Neverlang instance, which will parse and possibly generate the source code in a target language.

Launching and interpreting interactively: Neverlang comes bundled with a convenient predefined launcher called `nlg`. The launcher instatiates a given language and invokes the `Language.eval(String)` API (Section 4.3) on each of the given file names.

The *read-eval loop* utility `nlgi` executes input programs interactively and it can be started by invoking it at a 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 [41]. Fig. 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.

[7] Moreover, this allows to include even language variants such as @Java's extended annotations [40].

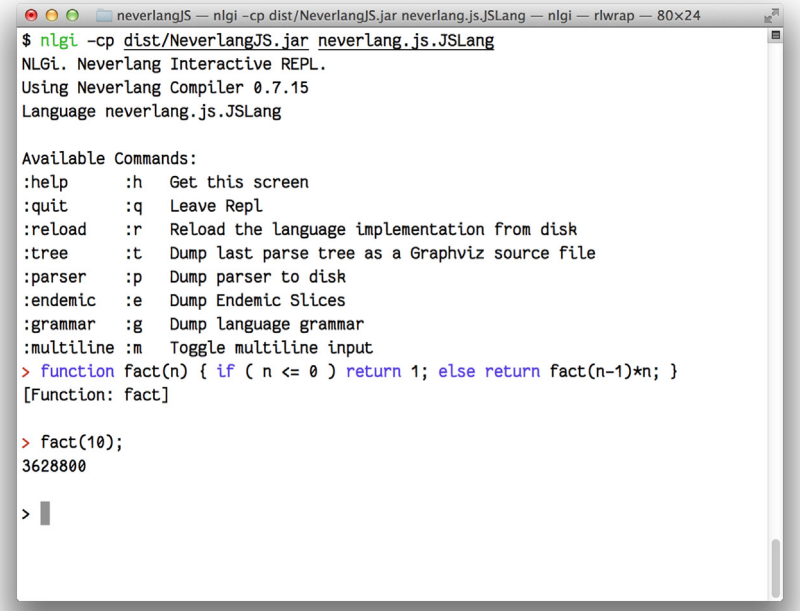


Fig. 3. `nlgi` executing a JavaScript program with neverlang.js (Section 6).

4.3. 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 [20]). The framework is designed so that its APIs are easy to use. The Neverlang language is only one of the possible *front-end*s to the Neverlang core. The Neverlang APIs can be used *directly*, exploiting the multi-language features of the JVM platform. For simplicity, the code examples in this section will always use Java, as it is the *lingua franca* of the JVM.

4.3.1. Architecture

In Neverlang modules, slices and the language descriptor are mapped onto *regular Java classes*. They are loaded by a Java `ClassLoader` through their *canonical class name*. The canonical class name reflects the dotted identifier that is conventionally used in 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 that load components *by name*, such as `importSlice()`. 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*.

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 [21]). 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.

Listing 15. Components in Section 4 as represented using Neverlang's APIs.

```
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")
    );
  }
}
```

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 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 Section 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

  $$<\texttt{module-name}>\texttt{\$role\$syntax}.$$

  For instance, the syntax for module `javalang.WhileLoop`in Listing 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

$$<\texttt{module-name}>\$\texttt{role}\$<\texttt{role-name}>\$<\texttt{N}>$$

where $<\texttt{N}>$ is the number of the nonterminal that is being hooked and $<\texttt{role-name}>$ 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 0 for the `type-checking`role of `javalang.WhileLoop`would be mapped onto

$$\texttt{javalang.WhileLoop\$role\$type\_checking\$}\odot$$

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). Fig. 4 shows a summary of all the classes that must be generated; Listing 15 shows a complete example of how the *Neverlang language* relates to the Neverlang API. Fig. 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 (i) reduces compile times (ii) simplifies *separate compilations* (iii) 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. This is particularly useful as the language implementation becomes large and complex (see the experience we conducted with neverlang.js Section 6.2). Plus, 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, 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*.

4.3.2. Runtime and execution

The Neverlang runtime is made of two main parts: the DEXTER [42,43] incremental parser generator and the component manager [20]. The component manager is responsible for loading `language`s, `slice`s and `module`s, 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*). 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. 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 $m: P \times \mathbf{R} \times \mathbb{N} \to SA_{\perp}$ where P is the set of productions for a grammar G, $\mathbf{R} = \{R_0, R_1, \ldots R_n\}$ is the set of all the roles for language L, and $\mathbb{N} = \{0, 1, \ldots n, \ldots\}$ is the set of natural numbers, and $SA_{\perp}$ 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

$$m(p, r, i) = \begin{cases} S_{r,i} & \text{if } S_{r,i} \text{ exists} \\ \perp & \text{otherwise} \end{cases} \tag{1}$$

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. 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 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. 6 represents the tree for the production `WhileLoop` (Listing 2), during the execution of the `type-checking` role. Because nonterminal `WhileLoop` is number 0 in slice `javalang.WhileLoop`, then:

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

where p is rule `WhileLoop` ⟵"`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 interface `SemanticAction`:

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

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. 6, the `Context` instance would contain node 0, 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")
```

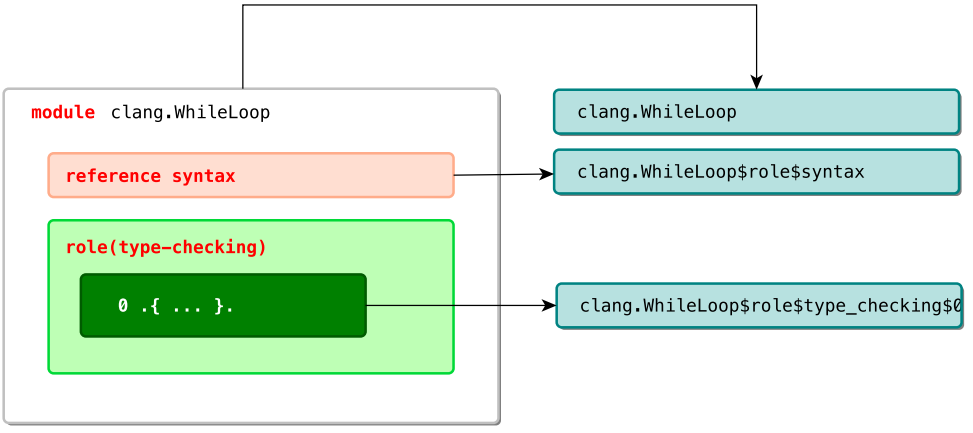


Fig. 4. How a module is broken down into several classes.

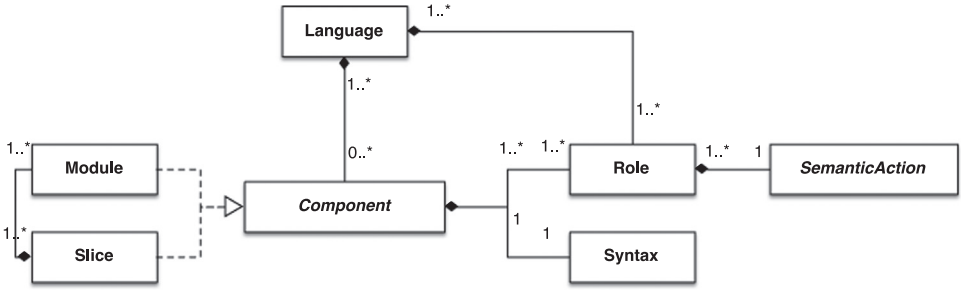


Fig. 5. Relations between classes and interfaces in Neverlang (interface names are in italics).

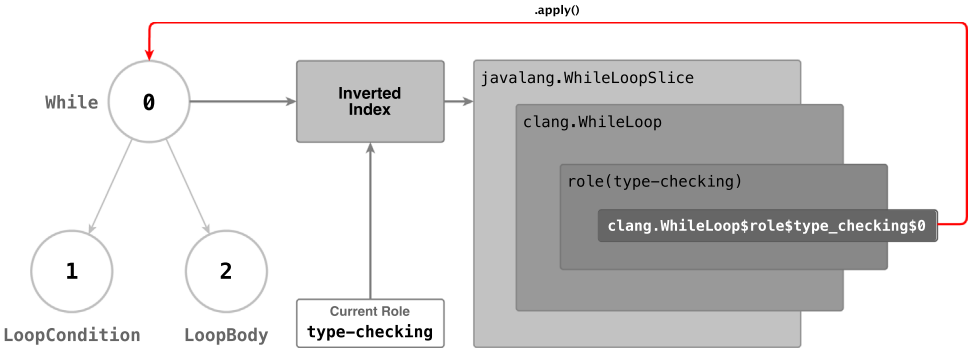


Fig. 6. Method dispatching.

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 15 shows how the semantic action 0 for the `type-checking` phase in module `javalang.WhileLoop` (Listing 2) may be written in Java by only using Neverlang's native APIs.

4.3.3. DEXTER

In order to support componentization and runtime composability, we also developed DEXTER: the Dynamically EXTEnsible Recognizer [42,43]. 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 2 to Listing 15). The DEXTER parser generator implements an algorithm that bears some resemblance to those described in [44,45]. The algorithm *updates* the LR(0) 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 (Section 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 [43].

4.3.4. Conflicts and syntactic composition

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. However, many notable subclasses of LR, such as LALR, in general are *not closed under composition* (see e.g., [46]). However, this can be tackled in many ways; for instance, a known subset of LALR (1) is closed under composition; it has been shown [46,47] that targeting this subset is practical if *context-aware scanning* is employed: this is the subset that is accepted by the Copper parser generator [46]. Currently DEXTER does not directly address the problem of composition, and therefore an error is raised if two language components introduce a syntactic conflict in the language. The language developer is then able to fix the problem by editing the grammar. Remapping can be used to reuse the code in most situations. Indeed, this is an area where further work may be useful. Nevertheless, the Neverlang framework is not in itself tightly coupled to DEXTER, and alternative parser generators may be employed in the future. A possibility that is being investigated is to support Copper's subset of LALR.

5. Case study: evolution of a DSL through composition

DSLs are computer languages designed to tackle problems that are tied to a particular problem-domain. Studies pointed out [48] 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 shows a simple but complete usage example of Neverlang. The example is the same state machine DSL in [49]. 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 Section 6.1 the same experience will be discussed, comparing other language frameworks. The source code listings may have been edited for readability, the full

working example with source code can be downloaded from http://neverlang.di.unimi.it/comlan14/examples.tgz.

Listing ~16. Grammar of the State Machine DSL.

```
// 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;
```

5.1. A simple state machine DSL

Listing 17. A snippet from module `sm.base.Program`.

```
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) { ... }
}
```

The state machine defined in the input program will be translated 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 `Identifier`s 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. Fig. 7 shows the state machine for a `door`, along with the code that describes it; this machine loops indefinitely.

Listing 16 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* (Section 4) have been defined: `collect-states`, `validation`, `code-gen`. Each role represents a different *concern* in the DSL. 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`: 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 17 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 implements the bucket brigade operator [50]. The usage of this API is the preferred way to deal with such cases. 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 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. 7) 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 Section 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 2.

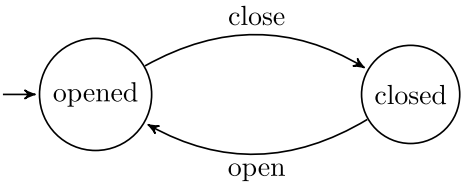
The semantic action for the `Program` nonterminal in the `code-gen` phase produces compilable code for the state machine (Listing 18). The generated source code is a simple Java program with a `while` loop that switches over the possible states of the machine, setting the variable `nextState` when a transition exists.

Listing 18. Compiled code for the Door state machine.

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

The `language` descriptor (Section 4.1) `sm.base.Lang` lists all of the slices constituting the base language. The `nlgc` tool (Section 4.1.1) produces the source code that interfaces with the Neverlang API (Section 4.3. The code is compiled bt `javac`.[8] The language implementation may be executed using `nlg` or `nlgi` (Section 4.2), *called* from a regular JVM program (Section 4.3.2), and *reused* across different language implementations without any change. The input program in Fig. 7 produces the compilable source code in Listing 18 (support APIs are provided to automatically generate an output file on disk).

[8] Of course, as seen in Section 4, if any other language is used in the semantic actions, users will have to invoke the language-specific compiler.




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

Fig. 7. Door state machine.

Table 2
Auxiliary classes for the state machine DSL.

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

5.2. A simple imperative language

Listing 19. Relevant parts of the grammar for the Action Language.

```
// 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 20. Example code for `Sum` in the Action Language. Code generation uses the `template` syntax.

```
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 }} }.
  }
}
```

Executable UML models include the specification of an *action language* [51] that can be used for many purposes, such as expressing actions and guards in a state machine model. 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 `javalang.Lang` language that we used as our running example in Section 4. It is easy to see that by combining the

syntax definitions in Listing 19 with the `WhileLoop` definition, 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 (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 20 shows the module for the `Sum` definition. The `code-gen` phase uses the `template` syntax (see Section 4.2). 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. A `VarTable` keeps track of the defined variables, thus one should include an endemic slice for this purpose.

5.3. Guards and actions: composing the DSLs

Listing 21. Code for the `Vending Machine` in Fig. 8. Code for `drinks` is omitted, since it mirrors the `candies` side.

```
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 22. A detail from `GuardedTransition`.

```
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) { ... }
}
```

The slices of `al` may be used to introduce *guards* and *actions* in our state machine DSL. Fig. 8 and Listing 21 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 implementation, 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 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`. The syntactic definition would be similar to the one in `Transition.nl` (Listing 16, p. 37), but it 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.

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

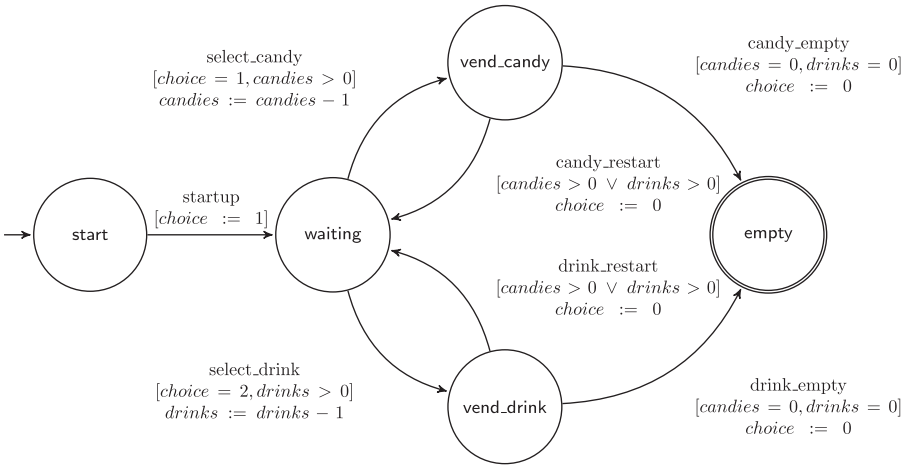


Fig. 8. Vending machine.

Listing 23. Slices included in the state machine language with guards and transitions.

```
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
}
```

Finally, Listing 23 shows all the slices that have been included in the complete 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.

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 implementations, using *language components*; moreover, 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. Each change does not require any editing on the existing source code, but rather consists in the creation of *new modules* and *new slices*. The original, pre-compiled implementations can be left untouched on disk.

6. Evaluation

The experiences that follow evaluate our model of language implementation. The state machine DSL of the previous section will be used to compare Neverlang to other modular language implementation frameworks, with the objective to show that the model is general enough to be reproducible using different tools. In the following the benefits of a native implementation of this model are shown: for instance, 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 at *runtime*. Finally the expressive power of Neverlang will be discussed by the help of an implementation of the DESK language [15].

6.1. Feature-oriented language implementation across tools

In this section 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 experiments were conducted using LISA 2.2, Silver r1230 (hg), Spoofax 1.2.0.0-s41399, Xtext 2.5.3. Full source code of the examples can be found at http://neverlang.di.unimi.it/comlan14/examples.tgz.

A full comparison of the features of the tools is shown in Table 4.

Feature-oriented language composition can be achieved if the language framework of choice provides facilities to modularize the language implementation both on the 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. This rules out the Xtext framework [52,53], which is severely limited in the way language components can be defined: for instance, only single-inheritance is permitted in syntactic definitions.

The other 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 [54], syntax definitions can be separated from semantics using the inheritance mechanism. Multiple inheritance makes it possible to componentize parts of the language into language components. Silver's grammars [23] are able 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 (Section 4.1.2), but it is in fact more powerful, because it supports *rewriting* the tree.[9] Neither in Silver or LISA there is a formal mechanism to separate between evaluation phases, but these can be nevertheless defined separately through design patterns (e.g., choosing a naming convention for the attributes, or modularize the implementation accordingly). Spoofax [55] is designed to separate concerns to the extent that each of the major evaluation phases (*name binding*, *code generation*, etc.) can be implemented using a different DSL. These DSLs are provided by the framework, but users are free to define their own evaluation phases using the native Stratego programming language.

Considering the taxonomy in Erdweg et al. [9], we may say that all the surveyed modular tools are able to support both semantic and syntactic 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. But, again, 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 [9] 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,[10] 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. The obvious drawback of this technique is that, still, entire modules have to be substituted, even when only a small computation has to be changed. There is indeed a trade-off between module size and the ability to substitute small computations; as modules become smaller, module management becomes more and more complex. But this kind of complexity could be managed through specific tooling (cf. Section 6.4 for dependencies and variability management).

As we already mentioned in Section 3.1, Mernik [22] have observed that self-extension is a property of a *language*, rather than a property of a language *framework*. For instance, SugarJ [56] 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 3, we used the usual metrics of lines-of-code (LOC) (as found e.g., in [57]) 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).

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

[9] Neverlang is currently adding support to a tree-rewrite DSL, though (Section 6.2).

[10] http://releases.strategoxt.org/strategoxt-manual/unstable/manual/chunk-chapter/rules-and-strategies.html#id3317807.

(Section 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.

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. According to the Sputnik test suite[11] our implementation covers about the 70% of the specification. This score may not result very high at a first glance; really, it is justified because a large part of the test cases cover the implementation of JavaScript's *built-in libraries*. Implementing these libraries is possible, but it constitutes a time-consuming activity, that we have planned 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. Our implementation of the *semantics* of the language is however complete: the interesting parts of the language are supported (e.g., *closures*, *higher-order functions* and the *prototype chain*), to the point that many of the built-in libraries may be even implemented *within* the language; the subset of the built-in libraries that is currently available is able to run browser-unrelated benchmarks in the Google Octane suite[12] without modifications.[13]

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.[14] 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 (see the paragraph at the end of this subsection on runtime evolution). A breakdown of the slices that constitute the neverlang.js implementation is presented in Table 5.

Extending JavaScript: *A classroom experience*: Our JavaScript implementation consists of 73 slices (Table 5) that correspond roughly to the same number of modules, 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. 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 Section 4.3.1, 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 extensions. Because of the didactic nature of the experiment, only a few extensions actually conflicted (cf. Section 4.3.4): 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: (i) language components can be developed separately, by different programmers, thereby allowing multiple teams to realize new features for a language implementation in parallel; (ii) features can be shipped as pre-compiled components; (iii) pre-compiled components can be composed onto the core language implementation at any time, possibly at runtime; (iv) independently developed features can be tested together without touching the core language implementation in an automated fashion.

[11] http://test262.ecmascript.org/
[12] https://developers.google.com/octane/benchmark
[13] In order to keep the grammar simple, semi-colon insertion was disregarded.
[14] Preliminary tests, mainly conduced on Octane's `crypto.js`, showed a 30 s execution time for neverlang.js, while Rhino takes about 6 s.

Table 3
Summary of the sizes of the different implementations.

Framework	N.Rules	N. Components	LOC	Support Code
Neverlang	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.)
LISA	26 rules (+ lexeme defs)	4 languages (12 rule sections) 3 method sects.	202 lines (externalized in classes for convenience)	6 Java classes
Silver	34 rules (+ terminal defs)	9 grammars	232 lines	(functions,included in LOC count)
Spoofax	21 rules + library lexemes	1 SDF module, 1 NaBL module (name resolution/validation), 1 stratego module (code-gen)	221 lines	(auto generated)
Xtext	16 rules (EBNF) + library terminals	1 Xtext module (grammar+name resolution/validation) 1 Xtend source file (code-gen)	178 lines	(auto generated)

Runtime evolution for dynamic optimization: In Neverlang slices and roles are pre-compiled components that can be *deployed* and *undeployed* at runtime. We are currently investigating how to exploit this feature to bring it one step further. Inspired by the Truffle [58,59] 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 [59] 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 Section 4.3.2 we described how actions are resolved. The key idea is that the component manager may to return a *set* of semantic actions; then the runtime system may *choose* 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 [15] to show an example of an *absolutely non-circular attribute grammar* (Section 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 Section 2). Full source code is available at http://neverlang.di.unimi.it/comlan14/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. The execution of a valid DESK program evaluates the expression and prints its value.

Table 4
Summary of the differences between the 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/ConcreteSyntax, 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 currrently no support for generics	Silver, Java through *foreign* 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 abritrary visits), post-order	Yes	Yes, attribute-driven	Yes; custom strategies may be user-defined	N/A
Language Composition Model					
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
Platform/APIs					
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
Generated Artifacts					
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
Utilities					
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 5
Summary of neverlang.js by feature bundle.

Bundle	Slices	LOC	Rules
Core			
Language core	11	277	24
Expressions			
Arithmetic	3	128	9
Boolean	3	92	5
Relational	2	137	10
Conditional	1	32	2
Bitwise	5	216	17
Typing (typeof, instanceof)	2	65	2
Function call	2	113	9
Construct call	1	56	3
Types			
String	1	21	2
Number	1	24	3
Boolean	1	23	3
RegExp	1	23	2
Object	4	189	13
Array	3	131	9
Function (definition)	2	100	11
This resolution	1	17	1

Bundle	Slices	LOC	Rules
Statements			
Block Statement	1	32	4
Cflow			
If Statement	1	45	3
Switch Statement	1	102	12
(Loop Statements)	*1*	*19*	*1*
While Statement	1	50	2
For loop	1	57	7
For-each loop	1	113	10
(NoIn expressions integration)	*11*	*305*	*26*
Interrupt: break	1	22	2
Interrupt: continue	1	22	2
Interrupt: return	1	30	3
Exception throwing + handling	2	122	8
Variables			
Variable assignment	5	226	21
Variable resolution	1	24	2
Endemic Slices			
Symbol Table		230	

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*.

Total Slices	**73**
Total LOC	**3043**
Total Rules	**228**

Table 6
List of JavaScript extensions.

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

Listing 24. DESK grammar.

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

Listing 25. Neverlang descriptor for the DESK language.

```
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 26. `Program` in DESK.

```
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);
    }.
  }
}
```

The attribute grammar in [15] has been converted into a Neverlang compiler. In Listing 24 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 25) defines three roles: `collect-constants`, `evaluation`, `code-gen`. The first role has *post-order* semantics (Section 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 26 the `Program`module is shown. This module contains the starting symbol of the grammar, as defined in [15]. The `evaluation`role uses the *semi-automated* evaluation strategy (Section 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 `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.

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[15] provides `ImmutableMap`. Another point that is worth mentioning is that our DESK implementation is a pedantic translation of Paakki's original, in a way that is non-idiomatic in Neverlang; if the DESK language in a way that is more Neverlang-friendly, 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, but we believe 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 [16,11].

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 (Section 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*— Section 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 (Section 4) to explicitly require attribute evaluation, while retaining conciseness. On 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. 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

Each language component in itself represents a *feature* of a language (Section 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*. 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 [27,28], 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 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 DSL, 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* [60–62] and generate the language implementation *automatically*. Mixing domains would also be allowed: for instance state machine features may be combined with an action language, as described in Section 5.

[15] https://github.com/google/guava

Part of the information is inferred directly from the dependency graphs that can be constructed from internal properties of the language components (Section 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 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. In the Neverlang case, these would be stored as fields of the objects that represent slices and modules (see Section 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 (Section 4.3). Tags are then fed into a *hierarchical clustering algorithm*. By manipulating the *dendogram* 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 Section 5. This language family included different kind of states, and extra transition types. Other experiments involved a simple imperative language. A similar experiment is now being conducted on neverlang.js (Section 6.2), where variants can be constructed using the slices of the full neverlang.js interpreter (e.g., a purely functional, stateless variant, and an imperative-only version).

7. Related work

Section 6 summarized which features of LISA, Silver, Spoofax were useful to implement a feature-oriented implementation of the state machine language. These frameworks provide further features to simplify language implementation.

AspectLISA [63,64] 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 it is harder to separate attribute definition from grammar definitions, and it is not possible to define an *abstract syntax*. LISA's main target language is a Java subset. 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 evaluate attribute grammars, with several choices on the kind of parser generator to use (e.g., LL, LR, and LALR) and the evaluation strategy for the attribute grammar (among the others, Lenic Tree Walk Evaluator, Katayama Evaluator, L-Attributed Evaluator, Visit Pattern Evaluator, etc.).

Silver [46,65] supports the definition of *abstract productions*, *verifiable* composition of modules, raising errors if a language extension is not *well-defined* [65] and *aspect productions*, (a different feature from LISA's). It also supports *attribute forwarding*, which can be described as a macro-like system to rewrite the semantics of a construct in terms of the semantics written for a different construct (e.g., by remapping attributes, and nonterminals; in Neverlang the simpler mechanisms of *remapping* and *renaming* are present). The generated parser uses a variant of LALR with context-aware scanning [47], and it implements an algorithm for *verifiable* composition of deterministic parsers [46]. These features together make easier to compose LALR grammars and give strong 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 interfacing with Java/JVM code for Neverlang is a key point, 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. Most notably, Silver is the only tool, among the surveyed, to support separate compilations,[16] Neverlang aside.

The Spoofax [55] language workbench internally uses Stratego, a dynamically typed declarative DSL 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[17]; interoperability with Java code is possible, but it is not within the main objectives of the project. Separate compilations are unfortunately not supported yet [66].

The JastAdd [67] is a Java-based attribute grammar system. JastAdd programmers define a grammar and the attributes and the system scaffolds pre-built AST Java classes. Aspects can be used to statically inject semantics into attributes. Attributes are implemented as Java methods that support parameters. JastAdd also supports *reference attributes* (similar to Silver's higher-order 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 [68] 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*

[16] http://melt.cs.umn.edu/publications/silver/silver-reference.pdf
[17] https://strategoxt.org/Stratego/StrategoJ

inheritance, similarly to LISA, all the rules of a parent grammar are inherited; *overriding* of rules is also possible. Semantics is given through visitors.

Jetbrains' Meta-Programming System (MPS) [69] for DSL implementation 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 main drawback with projectional editors is a steeper learning curve [70]. The system supports language modularization and inheritance-based code reuse.

Lightweight Modular Staging (LMS) [71] uses Scala's embedded DSL idiom to implement compilers. LMS 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.

As seen in Section 6, Xtext [52] language workbench is severely limited because it only supports single inheritance. Xsemantics [53] 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 also want to cite EMFText [72], another EMF-based tool [73] (like Xtext) that supports modular language implementation using syntax imports.

Neverlang's DEXTER is a modular LR parser generator. In the literature, authors have tried to address composability in LR parser, since LR is known not to be closed under composition (see Section 4.3.4); in particular, Silver targets a safe subset of LALR(1) using Copper [46], Bravenboer and Visser [74] describe an algorithm to compose together different grammar portions and obtain an LR(0) goto-graph. [75] describes an approach to componentized LR parsing. In [43] we proved the existence of a relation between goto-graphs, and presented an algorithm to transform the goto-graph of a grammar into the goto-graph of an extended grammar. Finally, with respect to extensible parsers, recent work by Reis et al. [76] has shown an extension to PEGs that may be employed to define extensible parsers, called *Adaptable PEG*.

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 and we showed that a native implementation of this model gives a greater range of possibilities. Separate compilations make it possible to redistribute pre-compiled components. In neverlang.js (Section 6.2) we showed that language extensions can be developed, and tested and integrated in parallel and in isolation. The variability management experience (Section 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 [77] 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.[18] The same implementation of Neverlang's compiler `nlgc` (Section 4.1.1) and of the interpreter for a complete, modern programming language (neverlang.js, Section 6.2) are a testament to the strengths of Neverlang and its underlying model. We are also completing the implementation of a modular Java language pre-processor, in the style of Polyglot [78], SugarJ [56], and ableJ [46]. Other experiments involved experimenting with a simple implementation of the Logo programming language. Moreover, because the core API is entirely Java 6 compatible, we were able to successfully port the entire Neverlang implementation onto Android (an example use case would be the *Recipe* language described in [20]), where its dynamic loading capabilities could be useful to separately distribute *plugins* for a core language implementation.

Nevertheless, in Neverlang there is still much room for improvement. Future work will concentrate both on its expressive power and ease of use. Runtime evolution and a DSL for tree rewriting are already in development, and thanks to the architecture of the framework they should be available soon. The language plugin system makes it possible to support new programming languages for semantic actions, and the dynamic mapping between parse trees and semantic actions would simplify support to dynamic dispatching of alternative actions.

Composition between language components is also a matter of establishing a set of guidelines: Section 3 gave an overview of the principle of *dependency* between language components. Even though Neverlang provides *renaming* and *remapping* capabilities (Section 4), guidelines on naming and factorization of language components are still an open problem that affects any modular language implementation tool. Our plan is to explore this problem more through field studies and further experiences.

Acknowledgments

This work has been partially supported by the MIUR project CINA: Compositionality, Interaction, Negotiation, Autonomicity for the future ICT society.

[18] http://www.mate.it/index.php/en/tyl

References

[1] Mernik M, Heering J, Sloane AM. When and how to develop domain specific languages. ACM Comput Surv 2005;37(4):316–44.
[2] Fowler M. Language workbenches: the killer-app for domain specific languages?. Martin Fowler's Blog. URL: ⟨http://www.martinfowler.com/articles/languageWorkbench.html⟩, May 2005.
[3] Ghosh D. DSLs in action. Manning Publications Co; 2010.
[4] Foderaro J. LISP: introduction. Commun ACM 1991;34(9):27 (special issue on Lisp).
[5] Fowler M, Parsons R. Domain specific languages. Reading, MA, USA: Addison Wesley; 2010.
[6] Fowler M. Fluent interface. Martin Fowler's Blog. URL: ⟨http://martinfowler.com/bliki/FluentInterface.html⟩, May 2005.
[7] Pollack M, Gierke O, Risberg T, Brisbin J, Hunger M. Spring data. O'Reilly Media; 2012.
[8] Dea C, Heckler M, Grunwald G, Pereda J, Phillips S. JavaFX 8. Apress; 2014.
[9] Erdweg S, Giarrusso PG, Rendel T. Language composition untangled. In: Proceedings of LDTA'12, Tallinn, Estonia. ACM; 2012.
[10] Karakoidas V, Mitropoulos D, Louridas P, Spinellis D. A type-safe embedding of SQL into Java using the extensible compiler framework. Comput Lang Syst Struct http://dx.doi.org/10.1016/j.cl.2015.01.001.
[11] Parr TJ, Quong RW. ANTLR: a predicated-LL(k) parser generator. Softw Pract Exp 1995;25(7):789–810.
[12] Swierstra SD. Combinator parsers: from toys to tools. In: ENTCS, vol. 41 (1), 2000. p. 38–59.
[13] Moors A, Piessens F, Odersky M. Parser combinators in scala, CW-491. Belgium: Katholieke Universiteit Leuven; February 2008.
[14] Knuth DE. Semantics of context-free languages. Math Syst Theory 1968;2(2):127–45.
[15] Paakki J. Attribute grammar paradigms: a high-level methodology in language implementation. ACM Comput Surv 1995;27(2):196–255.
[16] Aho AV, Sethi R, Ullman JD. Compilers: principles, techniques, and tools. Reading, MA, USA: Addision; 1986.
[17] Visser E, Benaissa Z-e-A. A core language for rewriting. In: ENTCS, vol. 15, 1998. p. 422–41.
[18] Saraiva J, Swierstra SD, Kuiper M. Functional incremental attribute evaluation. In: Proceedings of CC'00. Lecture notes in computer science, vol. 1781. Berling, Germany; Springer; 2000. p. 279–94.
[19] Bravenboer M, Kalleberg KT, Vermaas R, Visser E. Stratego/XT 0.17. A language and toolset for program transformation. J Sci Comput Program 2008;72(1–2):52–70.
[20] Cazzola W, Vacchi E. Neverlang 2: componentised language development for the JVM. In: Proceedings of SC'13. Lecture notes in computer science, vol. 8088, Budapest, Hungary. Springer; 2013. p. 17–32.
[21] Vacchi E, Olivares DM, Shaqiri A, Cazzola W. Neverlang 2: a framework for modular language implementation. In: Proceedings of modularity'14, Lugano, Switzerland. ACM; 2014. p. 23–6.
[22] Mernik M. An object-oriented approach to language compositions for software language engineering. J Syst Softw 2013;86(9):2451–64.
[23] Van Wyk E, Bodin D, Gao J, Krishnan L. Silver: an extensible attribute grammar system. Sci Comput Program 2010;75(1–2):39–54.
[24] Parnas DL. On the criteria to be used in decomposing systems into modules. Commun ACM 1972;15(12):1053–8.
[25] Adams SR. Modular grammars for programming language prototyping [Ph.D. thesis]. UK: University of Southampton; 1991.
[26] Farrow R, Marlowe TJ, Yellin DM. Composable attribute grammars: support for modularity in translator design and implementation. In: Proceedings of POPL'92. USA: ACM; 1992. p. 223–34.
[27] Vacchi E, Cazzola W, Pillay S, Combemale B. Variability support in domain-specific language development. In: Proceedings of SLE'13. Lecture notes in computer science, vol. 8225, Indianapolis, USA. Springer; 2013. p. 76–95.
[28] Vacchi E, Cazzola W, Combemale B, Acher M. Automating variability model inference for component-based language implementations. In: Proceedings of SPLC'14, Firenze, Italy. ACM; 2014. p. 167–76.
[29] Visser E. Separation of concerns in language definition. In: Companion proceedings of modularity'14, Lugano, Switzerland. ACM; 2014. p. 1–2.
[30] Cazzola W. Domain-specific languages in few steps: the Neverlang approach. In: Proceedings of SC'12. Lecture notes in computer science, vol. 7306, Prague, Czech Republic. Springer; 2012. p. 162–77.
[31] Gosling J, Joy B, Steele Jr. GL, Bracha G, Buckley A. The Java language specification. Java SE 8 ed.Pearson Education; 2014.
[32] Wadler P. The expression problem. Java Genericity Mailing List, November 1998.
[33] Oliveira BCDS, van der Storm T, Loh A, Cook WR. Feature-oriented programming with object algebras. In: Proceedings of ECOOP'13. Lecture notes in computer science, vol. 7920, Montpellier, France. Springer; 2013. p. 27–51.
[34] Oliveira BCDS. Modular visitor components: a practical solution to the expression families problem. In: Proceedings of ECOOP'09. Lecture notes in computer science, vol. 5653, Genoa, Italy. Springer; 2009. p. 269–93.
[35] Zenger M, Odersky M. Independently extensible solutions to the expression problem. In: Proceedings of FOOL'12, Long Beach, CA, USA, 2005.
[36] Schärli N. Traits — composing classes from behavioral building blocks [Ph.D. thesis]. Bern, Switzerland: Universität Bern; February 2005.
[37] Cazzola W, Speziale I. Sectional domain specific languages. In: Proceedings of DSAL'09, Charlottesville, Virginia, USA. ACM; 2009. p. 11–4.
[38] Cazzola W, Poletti D. DSL evolution through composition. In: Proceedings of RAM-SE'10, Maribor, Slovenia. ACM; 2010.
[39] Bettini L. Implementing domain-specific languages with Xtext and Xtend. PACKT Publishing Ltd; 2013.
[40] Cazzola W, Vacchi E. @Java: bringing a richer annotation model to Java. Comput Lang Syst Struct 2014;40(1):2–18.
[41] Ellson J, Gansner E, Koutsofios L, North SC, Woodhull G. Graphviz—open source graph drawing tools. In: Proceedings of GD'01. Lecture notes in computer science, vol. 2265, Vienna, Austria. Springer; 2001. p. 483–4.
[42] Cazzola W, Vacchi E. DEXTER and Neverlang: a union towards dynamicity. In: Proceedings of ICOOOLPS'12, Beijing, China. ACM; 2012.
[43] Cazzola W, Vacchi E. On the incremental growth and shrinkage of LR goto-graphs. ACTA Inform 2014;51(7):419–47.
[44] Horspool RN. Incremental generation of LR parsers. J Comput Lang 1990;15(4):205–23.
[45] Heering J, Klint P, Rekers J. Incremental generation of parsers. IEEE Trans Softw Eng 1990;16(12):1344–51.
[46] Schwerdfeger AC, Van Wyk ER. Verifiable parse table composition for deterministic parsing. In: Proceedings SLE'09. Lecture notes in computer science, vol. 5969, Dublin, Ireland. Springer; 2009. p. 184–203.
[47] Van Wyk E, Schwerdfeger A. Context-aware scanning for parsing extensible languages. In: Proceedings of GPCE'07, Salzburg, Austria. ACM; 2007. p. 63–72.
[48] Mens T, Wermelinger M. Separation of concerns for software evolution. J Maint Evol 2002;14(5):311–5.
[49] Tratt L. Evolving a DSL implementation. In: Proceedings of GTTSE'07. Lecture notes in compute science, vol. 5235, Braga, Portugal. Springer; 2008. p. 425–41.
[50] Jullig RK, de Remer F. Regular right-part attribute grammars. In: Proceedings CC'84, Montreal, Canada. ACM; 1984. p. 171–8.
[51] Mellor SJ, Tockey S, Arthaud R, Leblanc P. An action language for UML: proposal for a precise execution semantics. In: Proceedings of "UML" '98. Lecture notes in computer science, vol. 1618, Mulhouse, France. Springer; 1998. p. 307–18.
[52] Efftinge S, Völter M. oAW xText: a framework for textual DSLs. In: Proceedings of ESE'06, vol. 32, Esslingen, Germany, 2006.
[53] Bettini L. Implementing Java-like languages in Xtext with Xsemantics. In: Proceedings of SAC'13, Coimbra, Portugal. ACM; 2013. p. 1559–64.
[54] Mernik M, Lenič M, Avdičaušević E, Žumer V. LISA: an interactive environment for programming language development. In: Proceedings of CC'02. Lecture notes in computer science, vol. 2304, Grenoble, France. Springer; 2002. p. 1–4.
[55] Kats LCL, Visser E. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In: Proceedings of OOPSLA'10, Reno, Nevada, USA. ACM; 2010. p. 444–63.
[56] Erdweg S, Rendel T, Kästner C, Ostermann K. SugarJ: library-based syntactic language extensibility. In: Proceedings OOPSLA'11, Portland, Oregon, USA. ACM; 2011. p. 391–406.
[57] Liebig J, Daniel R, Apel S. Feature-oriented language families: a case study. In: Proceedings of VaMoS'13, Pisa, Italy. ACM; 2013.

[58] Würthinger T, Wimmer C, Woß A, Stadler L, Duboscq G, Humer C, et al. One VM to rule them all. In: Proceedings of Onward!'13, Indianapolis, IN, USA. ACM; 2013. p. 187–204.
[59] Humer C, Wimmer C, Wirth C, WößA, Würthinger T. A domain-specific language for building self-optimizing AST interpreters. In: Proceedings of GPCE'14, Västerås, Sweden. ACM; 2014. p. 123–32.
[60] Pohl K, Metzger A. Variability management in software product line engineering. In: Proceedings of ICSE'06, Shanghai, China. ACM; 2006. p. 1049–50.
[61] Hubaux A, Classen A, Mendonça M, Heymans P. A preliminary review on the application of feature diagrams in practice. In: Proceedings of VaMoS'10, Linz, Austria, 2010. p. 53–9.
[62] Chen L, Babar MA. A systematic review of evaluation of variability management approaches in software product lines. J Inf Softw Technol 2011;53(4):344–62.
[63] Henriques PR, Varanda Pereira MJ, Mernik M, Lenič M, Gray J, Wu H. Automatic generation of language-based tools using the LISA system. IEE Proc — Softw 2005;152(2):54–69.
[64] Rebernak D, Mernik M, Wu H, Gray JG. Domain-specific aspect languages for modularising crosscutting concerns in grammars. IET Softw 2009;3(3):184–200.
[65] Kaminski T, Van Wyk E. Creating and using domain-specific language features. In: Proceedings of GlobalDSL'13, Montpellier, France. ACM; 2013. p. 18–21.
[66] Erdweg S, Rieger F. A framework for extensible languages. In: Proceedings of GPCE'13, Indianapolis, IN, USA. ACM; 2013. p. 3–12.
[67] Hedin G, Magnusson E. JastAdd — an aspect-oriented compiler construction system. Sci Comput Program 2003;47(1):37–58.
[68] Krahn H, Rumpe B, Völkel S. MontiCore: a framework for compositional development of domain specific languages. Int J Softw Tools Technol Transf 2010;12(5):353–72.
[69] Völter M, Pech V. Language modularity with the MPS language workbench. In: Proceedings of ICSE'12, Zürich, Switzerland. IEEE; 2012. p. 1449–50.
[70] Völter M, Siegmund J, Berger T, Kolb B. Towards user-friendly projectional editors. In: Proceedings of SLE'14. Lecture notes in computer science, vol. 8706, Västerøas, Sweden. Springer; 2014. p. 41–61.
[71] Rompf T, Odersky M. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In: Proceedings of GPCE'10, Eindhoven, The Netherlands. ACM Press; 2010. p. 127–36.
[72] Heidenreich F, Johannes J, Karol S, Seifert M, Wende C. Model-based language engineering with EMFText. In: Proceedings of GTTSE'11. Lecture notes in computer science, vol. 7680, Braga, Portugal. Springer; 2011. p. 322–45.
[73] Steinberg D, Budinsky D, Paternostro M, Merks E. EMF: eclipse modeling framework. Reading, MA, USA: Addison-Wesley; 2008.
[74] Bravenboer M, Visser E. Parse table composition: separate compilation and binary extensibility of grammars. In: Software language engineering. Lecture notes in computer science, vol. 5452. Springer; 2009. p. 74–94.
[75] Wu X, Bryant BR, Gray J, Mernik M. Component-based LR parsing. Comput Lang Syst Struct 2010;36(1):16–33.
[76] Reis LVS, Di Iorio VO, Bigonha RS. An on-the-fly grammar modification mechanism for composing and defining extensible languages. Comput Lang Syst Struct http://dx.doi.org/10.1016/j.cl.2015.01.002.
[77] Gini R, Coppola M, Ryan PB, Righetti G, Peri I, Berni R, et al. 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 MIE'12, Pisa, Italy, 2012.
[78] Nystrom N, Clarkson MR, Myers AC. Polyglot: an extensible compiler framework for Java. In: Proceedings of CC'03. Lecture notes in computer science, vol. 2622, Warsaw, Poland. Springer; 2003. p. 138–52.