Neverlang 2 – Componentised Language Development for the JVM

Walter Cazzola, Edoardo Vacchi

Neverlang 2⋆
Componentised Language Development for the JVM

Walter Cazzola and Edoardo Vacchi

Department of Computer Science,
Università degli Studi di Milano, Milano, Italy
{cazzola,vacchi}@di.unimi.it

Abstract. Traditional compiler development is non-modular. Although syntax extension and DSL embedding is making its way back in modern language design and implementation, componentisation in compiler construction is still an overlooked matter. Neverlang is a language development framework that emphasises modularity and code reuse. Neverlang makes extension, restriction and feature sharing easier, by letting developers define language components in distinct, independent units, that can be compiled independently and shared across different language implementations, even in their compiled form. The semantics of the implemented languages can be specified using any JVM-supported language. In this paper we will present the architecture and implementation of Neverlang 2, by the help of an example inspired by mobile devices and context-dependent behaviour. The Neverlang framework is already being employed successfully in real-world environments.

Keywords: Domain-Specific Languages, Language Design and Implementation, Composability and Modularity

1 Introduction and Motivations

Compilers are traditionally complex and monolithic entities that only experts can maintain and extend [4]. Even though parsers and compilers for existing programming languages are nowadays often available as source code, they are usually not meant for a developer to adapt or build upon. For instance, even today, the `javac` compiler still relies on a hand-coded LALR parser that, for any developer trying to experiment, represents a high barrier to entry. Although there is an effort to implement the `javac` parser using ANTLR [26], in the context of the OpenJDK project[1], purely generative tools such as ANTLR and the time-honoured `lex` and `yacc` do not really account for modularity and decomposability, making code reuse in compiler development still a challenge. This in turn often translates to duplicate efforts, such as re-implementing the parser for a whole language even when the change is relatively small. Because of this problem, language extensibility is an interesting problem that is currently under research. Microsoft has recently released Roslyn [24], a technology preview of a platform-level API

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

[1] `http://openjdk.java.net/projects/compiler-grammar/`

to control and extend (by way of AST manipulations) the compiler of C#, and in general any language compiled for the .NET platform. Scala is moving in a similar direction, bringing compiler structures and features at the API level to support reflection and meta-programming [25]. Still, enabling extensibility does not automatically make a language implementation modular, in that extending a language is not the same as sharing a feature across different language implementations. It is not infrequent for languages to have features in common: in recent years, functional programming languages have been cross-pollinating the object-oriented world, and lazy evaluation and higher-order functions are now available in traditionally object-oriented languages such as C#, Scala and Python. Development of new languages often implies to put together the same old concepts and constructs with a different syntax; a typical example of this are conditionals and loops. It follows that development of new languages could benefit from being able to reuse portions of existing compilers; even more if these portions could be shared in a precompiled form. Sharing tested, precompiled components across language implementations, could help minimising the effort and timing required for the development of a new DSL. Moreover, precompiled, runtime loadable components could enable new possibilities, such as hot deployment of new features on running programs.

Our proposed solution is Neverlang [8], a framework designed to assists developers in the implementation of domain-specific languages in a modular way. The last Neverlang implementation we presented had some limitations: the parser generator rebuilt the parse table from scratch for any change in the code base; semantic actions were woven into the AST using the AspectJ compiler, so they had to be rewoven most of the time. In our experience this process took a perceivable amount of time for each rebuilding process. Our older implementation also imposed the choice of the Java language to express semantic actions. Finally, many people brought to our attention that our implementation of the Neverlang compiler had not been written using Neverlang itself.

In this paper we are introducing Neverlang 2. In this new version, we integrated our own compiler generator, which generates and updates LALR parser on-the-fly. Other components can be compiled independently and shared in their pre-compiled form, and, once these components have been finalised, they do not need to be recompiled any more. Pre-compiled semantic actions can be expressed using any language that the JVM supports, and the AspectJ dependency has been completely dropped, favouring instead a manual method dispatching mechanism. The new Neverlang compiler has also been completely rewritten on top of the new runtime library, and thus it is completely self-hosted. We will describe in detail the architecture of Neverlang 2 by the help of a running example inspired by modern sensor-rich mobile devices. The new version of the framework is already successfully solving real-world problems:

1. a Neverlang-generated DSL is now being integrated in TheMatrix [16], a Java framework to query and manipulate Italian administrative databases to produce information on the prevalence of chronic disease and on standards of care across the country;
2. Neverlang is being employed in the implementation of a DSL for ERP software development;
3. the Neverlang compiler has been boostrapped, i.e., it has been developed using Neverlang itself.

```
Set ringer mode to silent between 11:00 PM and 7:00 AM
```

Listing 1: on{X} recipe adapted from `http://onx.ms/recipes/silentAtNight`

```
when time is between 11:00pm and 7:00am : turn ringer off.
```

Listing 2: A DSL using the Recipe DSL

Paper Outline. In Sect. 2 we will describe the running example that we will employ to show Neverlang's features. In Sect. 3 we will describe Neverlang and its architecture, including the incremental parser generator DEXTER [10]. In Sect. 4 we describe the implementation of our running example. In Sect. 5 we discuss the related work and in Sect. 6 we draw our conclusions and describe future work.

2 Running Example

In recent years, mobile devices such as smartphones or tablets have become more and more accessible to the masses. Applications can interact with the great number of sensors of these devices to infer information about the user, and trigger specific actions depending on them. In particular, there are applications that enable users to define *custom actions* to take when a particular condition occurs. On the Android platform, for instance, there are Tasker[2] and Llama[3]. Some of these applications provide the end-user with a graphical user interface to specify the actions and the conditions. Microsoft's on{X}[4] enable users to share *real code snippets* written in JavaScript through a web application. Selected actions can then be synced and deployed to the device. Snippets are made available to programming-illiterate users using a natural-language description (a *recipe*) that can be partially customized. In Listing 1 there is one such recipe (simplified from one really available on the on{X} web site), to put a smartphone in silent mode when time happens to be between a particular, customizable range (in red).

Although the idea is nice, (a) it requires users to know JavaScript to define new actions and (b) code snippets cannot be written directly on the device, but only through the provided web interface. One might want to put the idea further by enabling users to write their own code snippets using a simplified, natural language-like DSL. In this case, users would be writing real code, except it would look similar to a recipe. In Listing 2 we show how the Recipe DSL might look like.

3 Neverlang 2 Architecture

Neverlang [8] is the framework that we developed to implement DSLs using a compositional approach. Our current implementation introduces a number of new features. The previous version of Neverlang employed AspectJ to weave executable code for

[2] `http://tasker.dinglisch.net`
[3] `http://kebabapps.blogspot.com`
[4] `http://onx.ms`

```
module recipe.lang.MainModule {
  import { neverlang.runtime.utils.* }
  role(syntax) {
    Program  <-  RuleList ;
    RuleList <-  Rule RuleList ; RuleList <-  Rule ;
    Rule     <-  "when" ConditionList ":" Action "." ;
    ConditionList <-  Condition ;
    ConditionList <-  Condition "and" ConditionList ;
   }
  role(evaluation) {
    0 .{ $0.rules   = AttributeList.collectFrom($1, "ruleObj"); }.
    7 .{
      List<Condition> conditions = AttributeList.collectFrom($8, "condition")
      $7.ruleObj = new RuleObj(conditions, $9.action);
    }.
  }
}
```

Listing 3: A Recipe program is a list of rules.

semantic actions inside an AST made of several Java class files. In fact, a typical implementation of the *visitor pattern* in an OOP context is to subclass each node of the AST and then invoke some `visit()` method on that node. However, this approach had two main drawbacks: (a) many source files had to be rewoven each time the smallest change affected the code base and (b) involving AspectJ in the building process of the generated compiler took a perceivable amount of time. In Neverlang 2 the AspectJ dependency has been dropped, the AST is generated on-the-fly using the DEXTER LALR parser generator [10] and a *component manager* now loads and dispatches the semantic actions, which now can be also written in any JVM-supported language. The added bonus is that now components can be compiled and possibly distributed separately. Finally the new Neverlang compiler `nlgc` has been bootstrapped. In this section we will briefly describe the concepts that Neverlang 2 has retained from the older version of the framework (for instance, the concepts of **`module`** and **`slice`**), and we will then detail the new architecture in depth.

3.1 Neverlang Components

In Neverlang, a single language component is defined in a *module*. Each module encodes a syntactic feature along with its semantics. For instance, in a C-like programming language a module can define the **`for`** looping construct or the **`if`** branch. C-like languages such as Java, JavaScript, PHP, etc. share most of their syntactic definitions. Writing a compiler or an interpreter using Neverlang, makes possible to share modules between implementations. Each module contains one or more of *roles*.

Modules and Roles. A `syntax` role is a portion of the language's formal grammar. For instance, Listing 3 shows part of the grammar for a Recipe program. Nonterminals are capitalised, and terminals (keywords) are between double quotes[5]. A Recipe program is a list of `Rules`, and each `Rule` is in the form:

```
Rule <- "when" ConditionList ":" Action "."          (1)
```

[5] Terminals can also be defined using regular expressions. In that case, literals are delimited by slashes; e.g., `/[a-z]+/` captures any non-zero-length word of lowercase alphabetic characters.

```
slice foo.bar.MySlice {
  module foo.bar.SomeModule with role syntax
  module baz.qux.AnotherModule with role evaluation type_check
}
```

Listing 4: A slice for a fictional programming language

that is, the `when` keyword, some condition to evaluate, a colon symbol, and some action to take when the condition evaluates to true. Conventionally, the module containing the `Program` nonterminal is considered the main module, and `Program` is always considered the start symbol of the grammar.

Any other **`role`** in the module is a *semantic* role, that is, a compilation phase. Every module can contain as many **`role`**s as needed. The order of evaluation is specified in a separate configuration file. Each **`role`** contains several code sections, introduced by a number. A code section introduced by a number N, binds the corresponding code section to the evaluation of the N-th nonterminal in the syntactic role[6]. Nonterminals are numbered from left to right and from top to bottom. For instance, in Listing 3, 0 would be the first nonterminal in the first production (`Program`), and 7 would be the first nonterminal in the third production (`Rule`). Thus, 0 binds the Java code between the delimiters "`.{`" and "`}.`" to be evaluated when visiting the the 0-th nonterminal in the syntax role, and 7 binds code to the `Rule` nonterminal. Code sections can also refer to nonterminals using the `$N` notation and associate custom attributes to them using a familiar dot-notation. For example, the code

```
$7.rule = new RuleObj(conditions, act);
```

creates an attribute for nonterminal **`$7`** called `rule`, which contains an instance of the class `Rule`. We will see what this code does in more detail in Sect. 4.

Slices and Languages. Once the language has been broken into separate modules, it can be composed together using the **`slice`** and the **`language`** constructs. The **`language`** construct composes together the modules that the developer selects using slices. A **`slice`** imports roles from (possibly) different modules, and it encapsulates a *feature* of the language. Listing 4 shows an example of the syntax: the slice is importing the syntax role for one module, and two semantic roles from a different module. For instance, with respect to our running example, the Recipe language needs at least one slice to define the time condition and one slice to define the action of turning the ringer on or off. Slices can be used to bind semantic roles from one module to the syntax defined in another, so they constitute a powerful mechanism to reuse code in compiler development. We will demonstrate this feature further by supporting a form of localisation in our Recipe language (Sect. 4).

3.2 The Neverlang Compiler

An important part of Neverlang, beside its own DSL, is obviously the Neverlang compiler. The new Neverlang compiler `nlgc` has been developed using the Neverlang run-

[6] It follows that syntactic roles are mandatory for semantic rules to make sense.

time (Sect. 3.3) to bootstrap the system. As a result, Neverlang today is completely self-hosted.

The `nlgc` tool acts like a translator from the Neverlang DSL into JVM-supported languages. Each **`slice`** and each **`language`** component is translated into a separate Java file. Modules are broken into several files: one Java class that explicitly declares every role in the module, one Java class containing the translation of the `syntax` role, and then one compile unit for each semantic action binding in each semantic role. For instance, the module `recipe.lang.MainModule` in Listing 3 is translated into 4 independent but logically related classes:

1. `recipe.lang.MainModule`, which lists each sub-component
2. `recipe.lang.MainModule$role$syntax`, which describe the syntactic part of the module
3. `recipe.lang.MainModule$role$evaluation$0`, because a semantic action has been bound to the 0-th nonterminal in the `evaluation` role
4. `recipe.lang.MainModule$role$evaluation$7`, because a semantic action has been bound to the 7-th nonterminal in the `evaluation` role

As we will see in Sect. 3.3 most of the class loading and method dispatching is performed automatically by the Neverlang runtime. Modules and slices have very few interdependencies and thus, they can be compiled *separately*. A change in one module requires to recompile only *that* module from source. 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 possible to compile only those components that really need to be rebuilt. Of course, this possibility becomes particularly useful when the compiler becomes large and complex. Moreover, pre-compiled Neverlang components can be bundled together in jars for convenience of distribution, and they can also be shared and imported by different languages independently.

Full JVM support. We said that `nlgc` translates the Neverlang DSL into JVM-supported languages. In most cases, this means that it generates Java source files. One core goal for the Neverlang 2 runtime was to have very few system requirements. Thus, the Neverlang 2 runtime has been written in Java, and the default language for semantic actions is Java as well. But semantic actions can be implemented using *any* language supported by the Java Virtual Machine, provided that a *translator plug-in* is available. The developer can then hint at the system that semantic actions are being written in a different language. Listing 5 shows an example of the syntax.

The new Neverlang compiler translates each semantic action into a class that implements the simple `SemanticAction` interface:

```
public interface SemanticAction { public void apply(ASTNode n); }
```

As mandated by the syntax-directed translation technique, a semantic action can attach arbitrary attributes to any nonterminal, that Neverlang refers with the dollar notation. This really translates to attaching attributes to the node of an AST: in Listing 3 the `condition` attribute will be attached to the root of any subtree of the AST which has

`Rule` at its root and the nodes `"when"`, `ConditionList`, `":"`, `Action`, `"."` as its children (more on this in Sect. 3.3)[7].

A translator plug-in describes how the occurrences of a nonterminal reference (in dollar notation) in a semantic action should be translated into the internal representation (a call to `n.nchild(int)`). Currently we have implemented support for Java and Scala. Listing 6 shows how this is done for Java. Code for Scala is similar. The plugin itself can be written in any JVM-supported language.

```
<jruby> // switch to jruby in global scope
module foo.bar.Multilang {
  role(syntax) { ... }
  role(role1) { 0 <scala> .{ ... }. } // scala for this action only
  role(role2) <jython> { ... }        // jython is default for this role
}
```

Listing 5: Any language running on the JVM can be supported.

```
public class JavaTranslatorPlugin extends TranslatorPlugin {
  public JavaTranslatorPlugin() {
    language = "java";
    fileExtension = "java";
    fileTemplate  = "public class {0} implements SemanticAction '{'\n"+
                    "  public void apply(ASTNode n) '{'\n{1}\n  '}'\n'}'";

    // when $N is the root of the subtree
    rootAttributeWrite  = "n.setValue(\"{1}\", {2});";
    rootAttributeRead   = "n.getValue(\"{1}\")";
    // when $N refers to a child node
    childAttributeWrite = "n.ntchild({0}).setValue(\"{1}\", {2});";
    childAttributeWrite = "n.ntchild({0}).getValue(\"{1}\")";
  }
}
```

Listing 6: Translator Plug-in for Java

3.3 The Neverlang 2 Runtime

The Neverlang 2 runtime is made of two main parts: the DEXTER [10] incremental parser generator and the component manager. A compiler written using Neverlang implements the well-known syntax directed translation mechanism [1], and implements an adaptive visitor pattern [8]. 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*).

The Component Manager. When a Neverlang-generated compiler or interpreter is started, the Neverlang 2 component manager kicks in. To ensure quick loading and interpreting of the directives contained inside the `language`, `slice` and `module` constructs, all these components need to be pre-compiled into JVM class files. Compilable source files are translated from Neverlang source files using the `nlgc` tool. These compilable source files can then be given as input to their corresponding native compilers (i.e.,

[7] Attributes are implemented as a map *attribute* → *value* attached to each node. Setting or getting is implemented as a method call. See Listing 6.

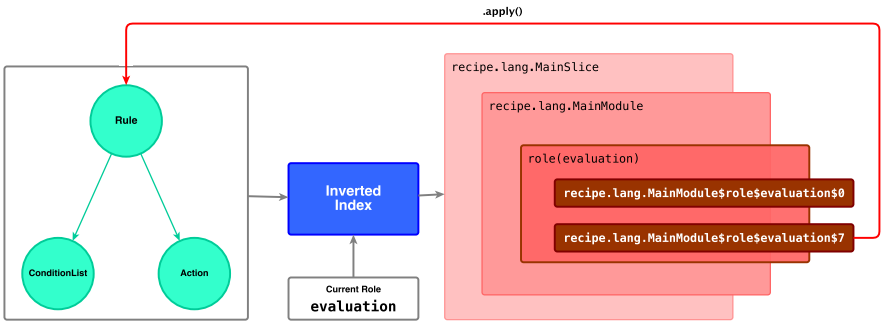

Fig. 1. The Component Manager and the method dispatching procedure.

`.java` files will be sent to `javac`, `.scala` source files might be compiled using `scalac`, and so on) to generate their own class files.

The compiled **`language`** component implements the *main* interpreter class (a subclass of `Language`), and includes the public API to interact with the interpreter. Using a Neverlang-generated compiler is usually as easy as instantiating this class and then invoking the method `Language.eval(String source)` on a source string, which returns an evaluated AST. The `Language` subclass then loads every declared **`slice`**-related class, which in turn causes every **`module`**-related class to be loaded. Each semantic action is then loaded on-demand, only when the AST visiting procedure requires them to become available (in other words, if certain syntax is never used in an input file, the corresponding semantic action will never be loaded into memory).

Internally, when an input file is given to the generated compiler, the parser constructs an AST. Then, for each role that has been defined in the **`language`** construct, the tree is visited. For each node, if a semantic action has been defined, it should be executed. The component manager is responsible for this form of *method dispatching*: it executes the correct semantic action by invoking its `apply()` method on that node. For instance, if current role is `evaluation`, then, each time a node n containing a nonterminal N, pertaining to a production p is being evaluated, the component manager queries an inverted index to retrieve the slice s from which p has been imported. Then, the corresponding semantic action `sa` of the `evaluation` role imported from s (if any) is applied to the node (`sa.apply(`n`)`). In the process, new modules and new semantic actions might be caused to be loaded from disk. For instance, consider rule (1): the corresponding AST (if we ignore terminals) is like the one in Fig. 1. When the component manager visits the root node of this subtree (labeled with `Rule`), it queries the inverted index for the corresponding semantic action for the `evaluation` role. In this case there is one binding to nonterminal 7. The inverted index returns the right semantic action object, possibly loading the class file from disk, and it then invokes the `apply()` method on the root node.

Incremental Generation of LALR Parsers: DEXTER. In order to support componentisation and runtime composability, we developed DEXTER: the Dynamically EXTEnsible Recognizer. DEXTER builds a LALR parser from the production in the modules, parses the input that is given to the compiler and it constructs the AST that

the component manager visits. DEXTER implements an in-memory LALR parser generator that can be incrementally extended (*grown*) or restricted (*shrunk*) by adding and removing grammar productions on-the-fly[8] 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. For instance, the production in (1) becomes

```
p(nt("Rule"), /* <- */ "when", nt("Condition"), ":", nt("Action"))
```

where `p` is a method that takes a nonterminal (the left-hand side of a production rule), and then a list of symbols (the right hand side of a rule), and it returns a `Production` instance; `nt` is a method that returns a nonterminal symbol object instance, and string literals are converted to terminal symbol instances. The DEXTER parser generator implements an algorithm that bears some resemblance to those described in [20] and [19]. The algorithm *updates* the LR(0) DFA, which is the basis for many interesting parsers of the LR family, such as GLR and of course LALR, the one adopted in DEXTER. 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, whether they are keywords or patterns. Patterns are delimited by slashes, while keywords are delimited by quotes. See Listing 9 for an example of both.

4 Neverlang 2 in Action

In this section we describe the implementation of (part of) the Recipe DSL. We will first show how to support the example in Listing 2, then we will see how Neverlang 2 makes easier to extend the DSL. We will also show that it is possible to change its syntax while still leaving the semantic code unaffected.

4.1 Implementing the Recipe DSL

In Sect. 2, we showed a short snippet from our DSL Recipe. The interpreter for our language will obviously need an internal engine to execute actions when the specified conditions are met. We will not discuss the implementation details of this engine, since they would anyway depend on the particular software platform in use. We will concentrate on the development of the actual language interpreter. In Sect. 3 we described how Neverlang puts together the separate components that make up an interpreter (or a compiler). Our language needs at least three slices. One slice should define the general look of the DSL. A Recipe program is a list of rules, so we expect the first slice to describe this. A rule is supposed to express some truth condition, and then some action to take when the condition holds. As conditions and actions describe single features of our language, it will make sense to write one slice for each one of them. The main step will be to define three modules; then we will write one slice for each one of them, so that the component manager will be able to put them together.

Main Module. In Listing 3 we defined that a `Rule` in our language will be always in the form specified in (1). That is, the keyword `when`, a single condition or a list of

[8] The result of the computation can be still cached to disk for performance, though.

```
slice recipe.lang.MainSlice {
  module recipe.lang.MainModule with role syntax evaluation
}
```

Listing 7: The `MainSlice` slice for Recipe

```
language recipe.lang.Recipe {
  slices recipe.lang.MainSlice recipe.lang.TimeRangeConditionSlice
      recipe.lang.RingerActionSlice
  roles evaluation
}
```

Listing 8: The `language` definition for Recipe.

conditions separated by the and keyword, a colon symbol, and then the action to execute. You may notice that no action has been attached to nonterminals 2, 4, 10, 12. Action 0 contains a reference to `RuleList` (`$1`). Because collecting a list of attributes is a common usage pattern, Neverlang 2 makes available a library method for this purpose. In this case, the built-in library method `AttributeList.collectFrom($1,"ruleObj")` visits the AST subtree rooted at `RuleList` and collects the values attached to the attribute `ruleObj` of each `Rule` node, which is set in action 7. The same method call appears in the action 7, where it collects `"condition"` attributes in a `ConditionList`. It follows that modules that specify conditions shall fill this attribute. In fact, were this attribute not found, the system would raise an exception. For the sake of simplicity, in our example, conditions can be only connected by and.

Conditions and Actions. Conditions could be encapsulated into `Condition` object instances that would provide a `boolean check()` method that evaluates to true when the specified condition holds. The "time range" condition has been implemented in Listing 9. In this module, the condition is encapsulated into a `TimeRangeCondition` object (that would be a subclass of `Condition`). The starting and ending times are captured using a *terminal pattern* (described in Sect. 3.3): the *hash* notation `#N` references N-th pattern. In this case, numbering is per-rule, starting from 0. Therefore `#0` references the pattern in the second rule[9]. The predefined property `#N.text` contains the matched text.

[9] The pattern captures any possible date in the form `HH:MMam/pm`. Of course it might match malformed times such as `27:99pm`; in that case the `TimeRangeCondition` constructor could raise an exception

```
module recipe.lang.TimeRangeConditionModule {
  role(syntax) {
     Condition <- "time" "is" "between" Time "and" Time ;
     Time      <- /[0-9]{1,2}:[0-9]{2}(am|pm)/ ;
  }
  role(evaluation) {
    0 .{ $0.condition = new TimeRangeCondition($1.time, $2.time); }.
    3 .{ $3.time = #0.text; }.
  }
}
```

Listing 9: Time condition

```
module recipe.lang.RingerActionModule {
  role(syntax) {
     Action <- "turn" "ringer" OnOff ;
     OnOff  <- "on" ;  OnOff  <- "off" ;
  }
  role(evaluation) {
    0 .{ $0.action = new RingerAction($1.ringerIsOn); }.
    2 .{ $2.ringerIsOn = true;  }.
    3 .{ $3.ringerIsOn = false; }.
  }
}
```

Listing 10: Ringer action

```
String script = "when time is between 11:00pm and 7:00am : turn ringer off.";
ASTNode tree = new Recipe().eval(script);
List<RuleObj> rules = tree.getValue("rules");
RecipeSys.registerRules(rules);
```

Listing 11: Retrieving the result of the evaluation of the input string.

The "toggle ringer" action could be encapsulated into a subclass of a generic `Action` object that could provide a method `perform()` implementing the logic (in this case, interfacing with the system ringer and turning it on or off). In Listing 10 a `RingerAction` object is instantiated with a boolean representing the ringer status.

Slices and language. We can now define three slices (one for each module) like that in Listing 7, and then add all of them to the `language` definition for Recipe (Listing 8). Once everything has been passed through `nlgc` and compiled using `javac`, we can already use the `Recipe` object. When the `Recipe.eval(source)` method is invoked on the input string in Listing 2, it puts on the root node of the AST (`Program`) an attribute `ruleList`, which contains a one-element list of `RuleObj` instances (Listing 3). Then each rule can be passed on to the system that will put them into effect at the right time (Listing 11).

4.2 Extending the DSL

The Recipe interpreter can be wrapped up in a package and possibly deployed on the target device. Now, suppose that we want to extend the DSL to support location-based conditions. For instance, mobile devices may allow users to indicate a particular location as *home*. Our original recipe turned off the phone ringer when time was in a customizable range. However, during this time frame the phone user might be away from home, maybe even in a noisy place: in this case, the action should *not* be triggered, because we would prefer the ringer to be on. We would like to add a new feature: a location-dependent condition, "`my position is <location-name>`", that should be evaluated together with the time range condition. We want our Recipe script to turn the ringer off not only when time is in the given range, but when we are also at home (Listing 12). In Listing 13 is the Neverlang code that implements the new feature. Extending the DSL will be as easy as defining a simple slice (similar to that in Listing 7) and adding it to the `language` construct (Listing 8).

```
when my position is home
   and time is between 11:00pm and 7:00am : turn ringer off.
```

Listing 12: Extended Recipe DSL with location support.

```
module recipe.lang.LocationModule {
  role(syntax) {
     Condition ⇐ "my" "location" "is" PredefinedLocation ;
     PredefinedLocation ⇐ "home" ;
  }
  role(evaluation) {
    0 .{ $0.condition = new LocationCondition($1.location); }.
    2 .{ $2.location  = RecipeSys.getHomeLocation(); }.
  }
}
```

Listing 13: Location condition

4.3 Localisation

Slices are not just a way to advertise a feature to the component manager. Their real power is to allow to pick features from different modules and mix them together. In this example, we will pick the `evaluation` role defined in the previous `module`s, and apply it to a different (although similar) syntax definition. In particular, we will show that we can *localise* our DSL into another language, with very little effort. In Listing 14 we are showing a Recipe script that has the same meaning as the one in Listing 2 but written in Italian. In Listing 15 we are showing two modules with only one syntax definition each: the first redefines the syntax for the time range condition, and the second redefines the ringer action. As you can see, in both cases no semantic action is specified. In fact, we can reuse the semantic actions we defined in the English modules, because the syntax did not change the order of the nonterminals. Therefore, the action that will be performed when visiting a certain nonterminal will be the same as if the script were written in English. The change does not require the developer to recompile any of the older modules, which are unaffected by the change. In this case, the only components which need compiling are the affected slices and the new modules. Of course, this example is only meant to show that slices give great flexibility to language developers, and not to provide a compelling example for localisation, which is an entirely different matter. In this case, the syntax of the Italian language does not affect the way nonterminals are ordered, but this could very well happen, even in non-natural languages: we are currently working on a solution to this kind of problem (more in Sect. 6). A deeper discussion on how Neverlang supports DSL evolution can be found in [9].

5 Related Work and Discussion

MontiCore [22] is a framework for language composition and extension that provides grammar inheritance and rewriting mechanisms additionally to modularisation features. However the underlying parser generator is still traditional (ANTLR [26]), in the sense that parser-related code has to be recompiled from scratch most of the time the user updates the grammar. The Rats! [17] packrat parser generator makes possible to share and

```
quando l'orario è tra 11:00pm e 7:00am : spegni la suoneria.
```

Listing 14: Localised Recipe DSL.

reuse parser components by organising grammar fragments into *modules*. The Rats! module system makes possible to extend and programmatically rewrite existing grammar fragments in a way that resembles a grammar-tailored inheritance system. For instance, a Rats! module can import rules from another module, substitute symbols, and add new productions. However, Rats! is a traditional parser generator that generates Java code. We employed Rats! in the earlier versions of Neverlang, but, of course, the code-generation approach makes impossible separate compilations and sharing precompiled components. Neverlang 2's DEXTER dynamic LALR parser generator does not yet support the same variety of operations on a grammar, but we are currently investigating in this direction. Given the dynamic and in-memory nature of DEXTER-generated parsers, we are confident that implementing features such as namespacing and symbol substitution would require only little effort, while adding rules is already possible.

As seen in [8], the JastAdd [18] compiler construction system was similar to Neverlang's first implementation in that it separates compilation aspects and implemented the AST using the traditional OOP style, generating all the required Java classes that were injected with methods and fields using AOP. The newest Neverlang architecture presents a runtime-generated AST; code is no more injected by way of weaving: instead, the component manager (Sect. 3.3) performs method dispatching depending on the AST node contents. This choice dispenses Neverlang 2 from needing a weaver, and greatly reduced the time to generate and compile the resulting compiler. Moreover, now Neverlang 2 components can be compiled independently and only when needed, and semantic actions can be expressed in any language supported by the JVM. On the other hand JastAdd does not focus on code reuse, it does not separates components nor optimises for pre-compiled code reuse, and it only supports Java.

Several tools deal with the problem of DSL embedding [11], where a host language embeds another language for specific purposes (e.g., SQL or XML literals). For instance, Metafront [5] and Metaborg [6] (part of the Stratego/XT toolset) are tools designed to perform syntactic transformation between programming languages typically to extend a host programming language with an embedded DSL. However, because the problem they are trying to solve is rather different than achieving modularity in the development of one programming language, as in Neverlang, these tools do not really take into account feature or component sharing. Their related literature made still for an interesting read during the development of the DEXTER extensible parser; in particular, in [7] the authors discuss an algorithm for LR parser extension, which is different from DEXTER's, though. In fact, [7] updates the LR(0) ε-DFA, while DEXTER, more similarly to [20,19], applies the updating procedure on the actual LR(0) DFA. This direct updating approach makes possible to avoid an otherwise required additional transformation step, that is, from ε-DFA to LR(0) DFA. This in turn cuts down on the requirement of keeping the intermediate representation available for any subsequent update. SugarJ [15] uses Stratego to provide syntactic transformation to Java programs in the form of library bundles.

```
module recipe.lang.TimeRangeConditionIt {
  role(syntax) {
     Condition ⇐ "l'orario" "è" "tra" Time "e" Time ;
     Time      ⇐ /[0-9]{2}:[0-9]{2}(am|pm)/ ;
  }
}
slice recipe.lang.TimeRageConditionSlice {
  module recipe.lang.TimeRangeConditionIt with role syntax
  module recipe.lang.TimeRangeConditionModule with role evaluation
}
module recipe.lang.RingerActionIt {
  role(syntax) {
     Action ⇐ OnOff "la" "suoneria";
     OnOff  ⇐ "spegni" ;  OnOff  ⇐ "accendi" ;
  }
}
slice recipe.lang.RingerActionSlice {
  module recipe.lang.RingerActionIt with role syntax
  module recipe.lang.RingerActionModule with role evaluation
}
```

Listing 15: Localising the Recipe DSL using Neverlang 2.

The xText [14] project is a framework and language workbench for the model-based development of DSLs that tightly integrates with EMF [28]. The framework makes possible to reuse existing grammars and existing meta-models to implement other languages, but it is really meant for model-driven development and therefore is conceptually different from Neverlang. It uses ANTLR to generate the parser. The framework includes xBase [13] a «partial programming language» that can be used as a base for other DSLs, and xSemantics, a DSL for writing «type systems, reduction rules and in general relation rules for languages implemented in xText» [3]. MPS [29] is another language workbench with similar objectives as xText, but it is backed and developed by the JetBrains software company.

LISA [23] is a language workbench and compiler generator that uses inheritance to compose grammars. Similarly to Neverlang it uses attribute grammars to express a language, but it bases on the concept of inheritance to extend and compose syntax and semantics attached the rules. It even includes AOP-like constructs to hook into nonterminals add possibly inject cross-cutting behaviour. Even though inheritance and this kind of AOP construct enable to both layer new semantic actions on top of the others and to override a behaviour altogether, they do not really make possible to define distinct compilation phases. Moreover, LISA is a more traditional compiler generator, in the sense that it outputs Java code using a traditional parser generator; semantic actions are expressed in a Java dialect.

The SPARK toolkit [2] for DSL implementation has similar goals to Neverlang, but it is Python-based. The most interesting part of SPARK is the somewhat curious choice for the parser generator, Earley [12], which is justified by the target audience for the project, that includes users that do not have a background in parser and compiler definition: Earley parsers can handle any context-free grammar, even ambiguous ones. Nevertheless, this comes at the cost of a higher computational complexity than, for instance, LALR. Beside this, SPARK takes a more traditional approach in the definition

of the components of a compiler, it does not really account for modularisation or feature sharing and, of course, it is limited to Python.

For completeness, we want to mention π [21], an experimental programming language where the only construct is the *pattern*, i.e., a mapping between a syntax definition and its intended semantic interpretation. Programs written using this language can extend their own syntax and express the new semantics inline. From that point on, the programmer can employ the new constructs anywhere in a program. Something similar can be found, from a more parsing-related perspective, in [27]; in fact, they both employ Earley parsing as well. These proposals differ from Neverlang in that they are more close to metaprogramming techniques and reflective systems; on the other hand, Neverlang is not really a programming language in itself, but rather a framework to define new languages.

6 Conclusions and Future Work

In this paper we described Neverlang 2 and its architecture. The strengths of this new versions are the modular implementation system, which makes possible feature sharing, even when components have been pre-compiled, and the full JVM support, that makes possible to implement the DSL semantics using any language supported by the Java platform.

We are currently working on extending our implementation to make it more robust, with respect to composition. For instance, namespacing and symbol importing may be useful to avoid name clashes when composing grammar fragments. Similarly, symbol renaming could be supported to compose syntax roles while carefully avoiding unexpected behaviour. Programmatic symbol renumbering could be also a way to reuse semantic actions in modules even when keywords in the syntax role occur in a different order (cf. Sect. 4.3). We are currently working on a way to carry on this kind of transformation in a semi-automatic way, by providing a mapping between abstract syntax trees in the composition phase. We are also planning to support layering of roles, that is, not only evaluating distinct roles as distinct phases, but also being able to group roles as part of the same phase (e.g., decorating the `evaluation` role with a `logging` role).

Nevertheless, we believe that Neverlang's current feature set is already promising. In order to stress-test the Neverlang framework, our lab has already implemented a reusable exception handling mechanism, and we are currently developing a modularised Java compiler. The project is already being employed to develop real-world DSLs both in the research and in the industry area: the development of a query DSL for TheMatrix [16] which is in the final testing phase at the time of writing, and the development of a new DSL for ERP software implementation.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, Reading, Massachusetts, 1986.
2. J. Aycock. The Design and Implementation of SPARK, a Toolkit for Implementing Domain-Specific Languages. *J. of Computing and Inform. Tech.*, 10(1):55–66, 2004.

3. L. Bettini. Implementing Java-like Languages in xText with xSemantics. In *Proc. of SAC'13*, pp. 1559–1564, Coimbra, Portugal, Mar. 2013. ACM.
4. J. Bosch. Delegating Compiler Objects. In *Proc. of CC'96*, LNCS 1060, pp. 326–340, Linköping, Sweden, Apr. 1996. Springer.
5. C. Brabrand and M. I. Schwartzbach. The Metafront System: Safe and Extensible Parsing and Transformation. *J. Science of Computer Programming*, 68:2–20, 2007.
6. M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A Language and Toolset for Program Transformation. *J. Science of Comp. Progr.*, 72(1-2):52–70, 2008.
7. M. Bravenboer and E. Visser. Parse Table Composition: Separate Compilation and Binary Extensibility of Grammars. In *Proc. of SLE'09*, LNCS 5452, pp. 74–94. 2009.
8. W. Cazzola. Domain-Specific Languages in Few Steps: The Neverlang Approach. In *Proc. of SC'12*, LNCS 7306, pp. 162–177, Prague, Czech Republic, June 2012.
9. W. Cazzola and D. Poletti. DSL Evolution through Composition. In *Proc. of RAM-SE'10*, Maribor, Slovenia, June 2010. ACM.
10. W. Cazzola and E. Vacchi. DEXTER and Neverlang: A Union Towards Dynamicity. In *Proc. of ICOOOLPS'12*, Beijing, China, June 2012. ACM.
11. T. Dinkelaker, M. Eichberg, and M. Mezini. An Architecture for Composing Embedded Domain-Specific Languages. In *Proc. of AOSD'10*, pp. 49–60, Saint-Malò, France, 2010.
12. J. Earley. An Efficient Context-Free Parsing Algorithm. *Commun. ACM*, 13(2):94-102, 1970.
13. S. Efftinge, M. Eysholdt, J. Köhnlein, S. Zarnekow, R. von Massow, W. Hasselbring, and M. Hanus. xBase: Implementing Domain-Specific Languages for Java. In *Proc. of GPCE'12*, pp. 112–121, Dresden, Germany, Sept. 2012. ACM.
14. S. Efftinge and M. Völter. oAW xText: A Framework for Textual DSLs. In *Proc. of the EclipseCon Summit Europe 2006 (ESE'06)*, volume 32, Esslingen, Germany, Nov. 2006.
15. S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. SugarJ: Library-Based Syntactic Language extensibility. In *Proc. of OOPSLA'11*, pp. 391–406, Portland, OR, USA, Oct. 2011.
16. R. Gini. Frameworks for Data Extraction and Management from Electronic Healthcare Databases for Multi-Center Epidemiologic Studies: a Comparison among EU-ADR, MATRICE, and OMOP Strategies. Keynote, Nov. 2012.
17. R. Grimm. Practical Packrat Parsing. TR2004-854, NYU, New York, NY, USA, 2004.
18. G. Hedin and E. Magnusson. JastAdd — An Aspect-Oriented Compiler Construction System. *Science of Computer Programming*, 47(1):37–58, Apr. 2003.
19. J. Heering, P. Klint, and J. Rekers. Incremental Generation of Parsers. *IEEE Trans. Softw. Eng.*, 16(12):1344–1351, 1990.
20. R. Horspool. Incremental Generation of LR Parsers. *J. Comp. Lang.*, 15(4):205-223, 1990.
21. R. Knöll and M. Mezini. π: A Pattern Language. In *OOPSLA'09*, pp. 503–522, 2009.
22. H. Krahn, B. Rumpe, and S. Völkel. MontiCore: A Framework for Compositional Development of Domain Specific Languages. *J. SW Tools for Techn. Transfer*, 12(5):353–372, 2010.
23. M. Mernik and V. Žumer. Incremental Programming Language Development. *Computer Languages, Systems and Structures*, 31(1):1–16, Apr. 2005.
24. K. Ng, M. Warren, P. Golde, and A. Hejlberg. The Roslyn Project: Exposing the C# and VB Compiler's Code Analysis. White paper, Microsoft, Oct. 2011.
25. M. Odersky. Reflection and Compilers. Keynote at `Lang.NEXT`, Apr. 2012.
26. T. J. Parr and R. W. Quong. ANTLR: A Predicated-LL(k) Parser Generator. *Software—Practice and Experience*, 25(7):789–810, July 1995.
27. P. Stansifer and M. Wand. Parsing Reflective Grammars. In *Proc. of LDTA'11*, pp. 10:1–10:7, Saarbrucken, Germany, Mar. 2011. ACM.
28. D. Steinberg, D. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, Dec. 2008.
29. M. Völter and V. Pech. Language Modularity with the MPS Language Workbench. In *Proc. of ICSE'12*, pp. 1449–1450, Zürich, Switzerland, June 2012. IEEE.