Neverlang Tutorial

Walter Cazzola, Edoardo Vacchi *Università degli Studi di Milano*

In this tutorial we will show how to implement a simple domain-specific language using Neverlang. The objective of this tutorial is to show that Neverlang makes possible to define a complete language implementation in a modular way. Each component can be compiled *separately* and *independently* from the others. The Neverlang framework is built on top of the Java runtime. It fits nicely in the Java ecosystem, using the tools that are native to this platform, and providing an environment that is easy and comfortable to use to the Java developer.

1 LogLang

LogLang is a simple DSL that describes tasks for a log rotating tool, similar to the `logrotate` Unix utility. A program written in LogLang is a list of *tasks* that should be performed on log files. Each task is a list of file system commands such as file *removal* or *renaming*.

This is how an input file for the LogLang interpreter should look like.

```
task DoSomething {
     backup "/foo/bar.txt" "/backup/bar.bak"
     rename "/foo/bar.txt" "/foo/bar.txt.old"
     remove "/faz.dat"
}
```

Listing 1: *The input file for our first LogLang iteration*

Each task is identified by a name, and each command is followed by one or more paths, enclosed within quotes. For instance, the `backup` command is followed by two string literals, because the backup copy of the source file should be placed on a different destination path. On the other hand, the `remove` command operates only on the path of the source file.
We will now show how to implement the interpreter for this script using Neverlang.

1.1 The Neverlang Architecture

Neverlang implements the syntax-directed translation technique [1] in a modular way. In the syntax-directed translation technique, the compiler (or the interpreter) for a language generates a parse tree for an input program, and then it visits this tree several times, attaching attributes to the nodes. Each visit is called a *compilation phase*. The final result is the execution of the program in the case of an interpreter, or executable code in the case of a compiler.

In Neverlang, each feature is defined in a separate `module`. Each module contains a section that defines the syntax for the feature, and it may contain one or more `role` definitions. A role is the part of a *compilation phase* that is bound to a particular syntactic feature. Consequently, each role contains only those semantic actions that should be performed when that part of the syntax is recognized by the interpreter.

Each feature is then described by a `slice`. A slice ties together related syntax definitions and roles, possibly coming from different modules.

Finally, the `language` descriptor lists every slice that the language implementation requires, and defines an order for the roles. Because every slice represents a feature, and every role represents a phase, the language descriptor actually describes the full language implementation.

1.2 Implementing LogLang v.1

The input file for the first version of our language (Listing 6) includes the definition for a `task` as a list of commands, and only three built-in commands: `backup` that copies a file to a long-term storage location, **`rename`**, that moves a file, and `remove`, that unlinks the file.

```
module neverlang.examples.loglang.Task {

        reference syntax {
             [TASK]     Task  ⇽ "task" Identifier "{" CmdList "}";
                               CmdList ⇽ Cmd CmdList ;
                               CmdList ⇽ Cmd ;
        }
        role (execution) {
                  [TASK](1)  .{ System.out.println("executing task "+ $TASK[1].ident
        }

}
```

Listing 2: *The Task module*

```
module neverlang.examples.loglang.Backup {
        imports {
                neverlang.examples.loglang.utils.*;
        }
        reference syntax {
                [BKP] Backup ⇽ "backup" String String;
                           Cmd     ⇽ Backup;
        }
        role(execution) {
                [BKP] .{
                          String src  = $BKP[1].string;
                          String dest = $BKP[2].string;

                          $$FileOp.backup(src, dest);
                }.
        }
}
```

Listing 3: *The Backup module*

Module The `reference syntax` section defines the syntax for our feature, represented as a BNF grammar. Unquoted identifiers represent nonterminals, while quoted strings are terminals. By convention, nonterminals always begin with a capital letter. There is also a way to define patterns using regular expressions that we will not show here. You may have noticed that there nonterminals that are not defined within this grammar. This is because they are expected to be defined in other modules. For instance, the `String` nonterminal should resolve to the definition for the quoted string literal that we use in the `backup` command. The second line is saying that the first grammar rule describes a command. All of these missing nonterminal definitions should be resolved at the end, when write the `language` descriptor.

The same module may define one or more semantic roles that refer that syntax definition. In this case, we are defining a role `execution` that should be executed whenever the `backup` command is encountered in an input file. The binding between actions and the syntax definition is made using numbers. Nonterminals are numbered from left to right, and from top to bottom, starting from 0. In this case, we are hooking a semantic action to nonterminal number 0 (that is, `Backup`). Inside the action we can refer to other logically-related nodes (the children) using the same numbering scheme. Inside the action, numbers must be prefixed using the `$` symbol. The familiar dot-notation can be used to set and retrieve the value of an attribute of a node. In this case we are referring to the attribute `string` of nonterminal number 1 and 2 (that is `String`, with capital 'S').

The default strategy of visit is the so-called post-order, that is, first the children of the tree are visited, and *then* the corresponding rule is executed. Even though we will not show it here, it is also possible to visit the tree in pre-order. In that case *first* the rule is executed, then the visit proceeds to the children. Because in this case we are doing post-order, then we can expect other attributes to have been set during the depth-first visit. In this case, we expect the `string` attributes to be set.

The $$-prefixed identifier is used in Neverlang to refer a singleton instance of an object that provides some services. In this case, it is a library that provides file system manipulation primitives. The implementation for this object is defined in a separate component that is called an `endemic slice`. We will not show the details on how to implement an endemic slice in this tutorial, but you can find more on that in Neverlang's documentation.

Modules can be compiled to source files using the `nlgc` tool. If you do not specify otherwise, the tool generates regular Java source files. Once the source files are generated, you can then compile them using `javac`. Once your modules are finalized and compiled, *you will not need to re-compile them anymore*, because the framework is designed to be modular at the class-file level. Components can be pre-compiled, packaged and distributed as bytecode.

```
$ nlgc -s out Backup.nl
Backup.nl
neverlang/examples/loglang/Backup$role$syntax.java
neverlang/examples/loglang/Backup$role$execution$0.java
neverlang/examples/loglang/Backup.java
$ javac -d bin out/neverlang/examples/loglang/*.java
```

It is worth noticing that the tool generates Java source files by default, but semantic actions can be written using any language supported by the Java Virtual Machines that compiles into bytecode. Neverlang comes bundled with support for Scala, but any JVM language that compiles into bytecode can be supported, by writing a plugin, using a very simple extension mechanism. If you are interested in the details, we refer you to [2, 3].

Slice The slice construct ties together every component that is used in the implementation of a feature. In this case, both the syntax definition and the semantic role are defined within the same module. Therefore, the slice looks like Listing 4.

```
slice neverlang.examples.loglang.v1.BackupSlice {
    concrete syntax from neverlang.examples.loglang.Backup
    module neverlang.examples.loglang.Backup with role execution
}
```

Listing 4: *BackupSlice*

You may have noticed that modules define a `reference syntax`; on the other hand, slices declare a `concrete syntax`. In Neverlang, the syntax definition is a component like any other. Therefore, although a semantic role may refer to a particular syntax definition, the feature implementation may actually use a different concrete syntax coming from a different module. For instance, a language implementation may use English for its keywords, but then it could be possible to localize the interpreter by choosing a different, localized concrete syntax in the slice.

The first line of the slice construct always declares a concrete syntax. Every subsequent line is a `role` declaration. The order in which roles are executed in specified in `language` descriptor.

Even slices can be compiled separately by the other components, and modules are not required to be compiled contextually to the slices that use them.

```
$ nlgc -s out BackupSlice.nl
BackupSlice.nl
neverlang/examples/loglang/v1/BackupSlice.java
$ javac -d bin out/neverlang/examples/loglang/*.java
```

```
language neverlang.examples.loglang.v1.LogLang {
  slices
          neverlang.examples.loglang.TaskSlice
          neverlang.examples.loglang.v1.RenameSlice
          neverlang.examples.loglang.v1.RemoveSlice
          neverlang.examples.loglang.v1.BackupSlice
          neverlang.examples.loglang.Main
          neverlang.commons.SimpleTypes
  endemic slices
          neverlang.examples.loglang.FileOpEndemic
  roles syntax < terminal-evaluation < execution
}
```

Listing 5: *The language definition for the first iteration of LogLang*

Language The language descriptor lists every slice that the language uses and defines the order in which roles should be executed. You may notice that `neverlang.commons.SimpleTypes` is a slice that comes bundled with Neverlang, that defines common literals. In particular it defines quoted strings, that we used in the commands of our language. This slice also defines a role called `terminal-evaluation` that we therefore execute prior to the `execution` role that we defined in our commands. This way, we ensure that the phase that evaluates the terminals defined inside `SimpleTypes` will be executed before `execution`.

Even in this case, modules and slices can be pre-compiled, and the language descriptor can be compiled separately, provided that the other class files are on the class path.

```
$ nlgc -s out LogLang.nl
LogLang.nl
neverlang/examples/loglang/LogLang.java
$ javac -d bin out/neverlang/examples/loglang/*.java
```

Executing the interpreter Neverlang generates simple, readable Java source files. For instance, this is how the language descriptor looks like, once compiled:

```
package neverlang.examples.loglang.v1;
import neverlang.runtime.*;
public class LogLang extends Language {
  public LogLang() {
    importSlices(
     "neverlang.examples.loglang.TaskSlice",
     "neverlang.examples.loglang.v1.RenameSlice",
     "neverlang.examples.loglang.v1.RemoveSlice",
     "neverlang.examples.loglang.v1.BackupSlice",
     "neverlang.examples.loglang.Main",
     "neverlang.commons.SimpleTypes"
    );
    importEndemicSlices(
     "neverlang.examples.loglang.FileOpEndemic"
    );
    declare(
     role(Role.Flags.POSTORDER, "terminal_evaluation"),
     role(Role.Flags.POSTORDER, "execution")
    );
  }
}
```

As you can see, the generated language descriptor is nothing but a subclass of the `Language` class found in the `neverlang.runtime` package. This class can be instantiated by user code, and it provides an API to parse and execute program files. However, in order to test language implementations immediately, the Neverlang framework provides two bundled tools: the `nlgi` interactive interpreter and the `nlg` laucher.

The `nlg` launcher requires the user to provide the canonical class name for a language implementation and a series of input files. The launcher will then interpret or compile the given input file using the language implementation provided by the specified class. Optionally, it is possible to specify a classpath using the `-cp` command line switch. In this tutorial, we will omit this detail.

```
$ nlg neverlang.examples.loglang.v1.LogLang ../tests/loglang-input-1.txt
executing task DoSomething
`\color{blue}{backup: /foo/bar.txt --> /backup/bar.bak}`
`\color{blue}{rename: /foo/bar.txt --> /foo/bar.txt.old}`
`\color{blue}{unlink: /faz.dat}`
```

1.3 Implementing LogLang v.2

In a new iteration of the LogLang interpreter, we want to introduce the `merge` command, that was previously unavailable.

```
task DoSomething {
    backup "/foo/bar.txt" "/backup/bar.bak"
    rename "/foo/bar.txt" "/foo/bar.txt.old"
    merge  "/baz/qux1.txt" "/baz/qux2.txt"
    remove "/faz.dat"
}
```

Listing 6: *The input file for the second LogLang iteration*

In fact, if we try to execute this input file with our interpreter we will get a syntax error.

```
$ nlg neverlang.examples.loglang.v1.LogLang ../tests/loglang-input-2.txt
`\color{red}{4:5: Unexpected identifier: 'merge'. Expected: symbol '\}'.}`
```

The `merge` command has a similar syntax to the one for the `backup` command, and it is really easy to implement.

We can now write a new language descriptor that includes the new components.

As you will probably notice, the slice names that we used in the previous version are again found in this version. This is because we are *actually* using the pre-compiled slices that composed the previous iteration.

The new version of the interpreter obviously supports both the new input file and the old one.

```
module neverlang.examples.loglang.Merge {
        imports {
                neverlang.examples.loglang.utils.*;
        }
        reference syntax {
                [MRG]        Merge <- "merge" String String;
                                Cmd           <- Merge;
        }

        role(execution) {
                [MRG] .{
                        String src  = $MRG[1].string;
                        String dest = $MRG[2].string;

                        $$FileOp.merge(src, dest);
                }.
        }
}
```

```
slice neverlang.examples.loglang.v2.MergeSlice {
    concrete syntax from neverlang.examples.loglang.Merge
    module neverlang.examples.loglang.Merge with role execution
}
```

Listing 7: *Merge implementation.*

```
language neverlang.examples.loglang.v2.LogLang {

  slices neverlang.examples.loglang.v1.BackupSlice
         neverlang.examples.loglang.v1.RemoveSlice
         neverlang.examples.loglang.v1.RenameSlice
         neverlang.examples.loglang.v2.MergeSlice // new feature
         neverlang.examples.loglang.Task
         neverlang.examples.loglang.Main
         neverlang.commons.SimpleTypes

  endemic slices neverlang.examples.loglang.FileOpEndemic

  roles syntax < terminal-evaluation < execution

}
```

Listing 8: *The language definition for the second iteration of LogLang*

```
$ nlgc -s out *.nl
[ output omitted ]
$ javac -d bin out/**/*.java
$ nlg neverlang.examples.loglang.v2.LogLang ../tests/loglang-input-2.txt
executing task DoSomething
`\color{blue}{backup: /foo/bar.txt --> /backup/bar.bak}`
`\color{blue}{rename: /foo/bar.txt --> /foo/bar.txt.old}`
`\color{blue}{merge: /baz/qux1.txt + /baz/qux2.txt}`
`\color{blue}{unlink: /faz.dat}`
```

1.4 Implementing LogLang v.3

In the third version of our interpreter we want to show off another feature that is peculiar to Neverlang. Using slices, it is possible to define in separate modules further processing phases, but, again, in separate components. The original pre-compiled units do not need to be modified. In this version of the interpreter, we want to add a permission check phase, in which file permissions are checked. If the tool runs at an insufficient level of permissions, the execution of the program is aborted. Moreover, we want to add logging capabilities to the utility, so that each time a task is performed, we can keep track of the operations in a transcript.

In order to do this, we can define new modules: one for each role and each feature that we want to implement. For instance, we can define `BackupPermCheck` for permissions and `BackupLogging` for logging.

```
module neverlang.examples.loglang.BackupPermCheck {
    imports {
        neverlang.examples.loglang.utils.*;
    }
    reference syntax from neverlang.examples.loglang.Backup
    role(permissions) {
        [BKP] .{
            String f1 = $BKP[1].string;
            String f2 = $BKP[2].string;
            if (! $$Permission.canRead(f1) || ! $$Permission.canWrite(f2)
                throw new Error("cannot perform backup operation: wrong p
            }
        }.
    }
}
```

Listing 9: *The new Permission Check module for the third iteration of LogLang*

As you may notice, the `reference syntax` section is shortened so that it points to the definition found in the original module. This is required to resolve the numbers in the role definitions to the correct nonterminals. You may also notice that new endemic slices are being used. We will need to indicate them in our new language definition.

```
module neverlang.examples.loglang.BackupLogging {
    imports {
        java.util.logging.Logger;
    }
    reference syntax from neverlang.examples.loglang.Backup
    role(logging) {
        [BKP] .{
            $$Logger.info("Backup: "+$BKP[1].string+" --> "+$BKP[2].string);
        }.
    }
}
```

Listing 10: *The new Logging module for the third iteration of LogLang*

```
slice neverlang.examples.loglang.v3.BackupSlice {
    concrete syntax from neverlang.examples.loglang.Backup
    module neverlang.examples.loglang.Backup with role execution
    module neverlang.examples.loglang.BackupPermCheck with role permissions
    module neverlang.examples.loglang.BackupLogging with role logging
}
```

Listing 11: *BackupSlice3*

The new slice for `backup` reflects that new roles and new modules are now involved. You will also notice that even in this case, we are still using the pre-compiled modules that we used in the previous iterations.

```
language neverlang.examples.loglang.v3.LogLang {

  slices neverlang.examples.loglang.v3.BackupSlice
         neverlang.examples.loglang.v3.RemoveSlice
         neverlang.examples.loglang.v3.RenameSlice
         neverlang.examples.loglang.v3.MergeSlice
         neverlang.examples.loglang.Task
         neverlang.examples.loglang.Main
         neverlang.commons.SimpleTypes

  endemic slices neverlang.examples.loglang.FileOpEndemic
                 neverlang.examples.loglang.PermEndemic
                 neverlang.examples.loglang.LoggerEndemic

  // roles syntax < terminal-evaluation < logging < permissions < execution
  roles syntax < terminal-evaluation < logging : permissions : execution
}
```

Listing 12: *Third version of LogLang*

The new language definition includes the endemic slices that implement the permission check operations and the logging library. In this example we are specifying that logging should be performed before every other file-related operation. Then, we execute the permission checks, and finally, if everything went right, we can proceed with executing the task. The commented line is defining an alternate execution order. We will get back to that in a moment, but, first, we will compile this version of the language.

Using Ant for build automation. In this version of the language, we will use `ant` to automate the build. Neverlang comes bundled with an `ant` task that the Neverlang developer can use to automate builds. In this case, Neverlang `.nl` source files are usually put in a `nlg-src` directory, and generated source files go inside the `src` directory.

To use this task, we just add a task definition at the top of our build script:

```
        <taskdef name="nlgc"
              ↪    classname="neverlang.tools.ant.NlgcTask">
            <classpath>
              <pathelement path="${classpath}"/>
              <pathelement location="path/to/Neverlang.jar"/>
            </classpath>
        </taskdef>
```

Listing 13: *Task definition for the Ant build script*

A typical Ant build script for Neverlang first compiles every Neverlang source file to Java source code and then compiles all the Java source files to bytecode. This build script (Listing 14) also generates a jar file in the `dist/` directory. We can launch the generated interpreter with the usual command.

```
$ nlg neverlang.examples.loglang.v3.LogLang ../tests/loglang-input-1.txt
INFO: Backup: /foo/bar.txt --> /backup/bar.bak
INFO: Rename: /foo/bar.txt --> /foo/bar.txt.old
INFO: Remove: /faz.dat
`\color{green}{canRead: /foo/bar.txt}`
`\color{green}{canWrite: /backup/bar.bak}`
`\color{green}{canRead: /foo/bar.txt}`
`\color{green}{canWrite: /foo/bar.txt.old}`
`\color{green}{canWrite: /faz.dat}`
executing task DoSomething
`\color{blue}{backup: /foo/bar.txt --> /backup/bar.bak}`
`\color{blue}{rename: /foo/bar.txt --> /foo/bar.txt.old}`
`\color{blue}{unlink: /faz.dat}`
```

As you can see, the output of each role has been colored so that it is easy to spot which is being executed. As we expected, first, the logging role prints on the screen information about the command that is going to be executed, then the permission check phase (in green) is performed, and finally the actual actions (in blue) are executed.

Interleaved execution We want to finally show another order of execution for roles. In fact, strict sequentiality is not the only way to execute a compilation phase. For instance, we might want the execution of a role to be interleaved to another. In the interleaved execution, instead of executing one role at a time for each traversal of the tree, we execute in sequence every action that is attached to a node. In our case, previously we made the interpreter execute a full tree traversal for each of the logging, permissions and execution phases. If we use the alternate ':' syntax, we specify to Neverlang that we want the execution of the roles to be interleaved. Therefore, if we uncomment the last line and comment the second last, and we re-compile the language descriptor, we obtain an interleaved execution of the phases, that you can easily spot, by looking at the colored lines.

```
$ nlg neverlang.examples.loglang.v3.LogLang ../tests/loglang-input-1.txt
executing task DoSomething
INFO: Backup: /foo/bar.txt --> /backup/bar.bak
`\color{green}{canRead: /foo/bar.txt}`
`\color{green}{canWrite: /backup/bar.bak}`
`\color{blue}{backup: /foo/bar.txt --> /backup/bar.bak}`
INFO: Rename: /foo/bar.txt --> /foo/bar.txt.old
`\color{green}{canRead: /foo/bar.txt}`
`\color{green}{canWrite: /foo/bar.txt.old}`
`\color{blue}{rename: /foo/bar.txt --> /foo/bar.txt.old}`
INFO: Remove: /faz.dat
`\color{green}{canWrite: /faz.dat}`
`\color{blue}{unlink: /faz.dat}`
```

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, Reading, Massachusetts, 1986.

[2] Walter Cazzola and Edoardo Vacchi. Neverlang 2: Componentised Language Development for the JVM. In Walter Binder, Eric Bodden, and Welf Löwe, editors, *Proceedings of the 12th International Conference on Software Composition (SC'13)*, Lecture Notes in Computer Science 8088, pages 17–32, Budapest, Hungary, 19th of June 2013. Springer.

[3] Edoardo Vacchi and Walter Cazzola. Neverlang: A Framework for Feature-Oriented Language Development. *Computer Languages, Systems & Structures*, 43(3):1–40, October 2015.

```
<project name="LogLangV3" default="dist" basedir=".">
  <description> LogLang v3 build file </description>

  <property name="src" location="src"/>
  <property name="nlg-src" location="nlg-src"/>
  <property name="build" location="build"/>
  <property name="dist"  location="dist"/>

  <taskdef name="nlgc" classname="neverlang.tools.ant.NlgcTask">
    <classpath>
      <pathelement path="${classpath}"/>
      <pathelement location="dist/Neverlang.jar"/>
    </classpath>
  </taskdef>

  <target name="pre-compile">
      <nlgc destdir="${src}" >
          <fileset id="nlg-src" dir="${nlg-src}" includes="**/*.nl"
                   ↪   />
      </nlgc>
  </target>

  <target name="init"> <mkdir dir="${build}"/> </target>

  <target name="compile" depends="init,pre-compile"
          ↪   description="compile the sources" >
    <!-- Compile the java code from ${src} into ${build} -->
    <javac includeantruntime="true" srcdir="${src}"
      destdir="${build}" classpath="${classpath}"/>
  </target>

  <target name="dist" depends="compile" description="generate the
          ↪   distribution" >
    <mkdir dir="${dist}"/>
    <jar jarfile="${dist}/LogLangV3.jar" basedir="${build}"/>
  </target>

  <target name="clean"
        description="clean up" >
    <!-- Delete the ${build} and ${dist} directory trees -->
    <delete dir="${build}"/>
    <delete dir="${dist}"/>
  </target>
</project>
```

Listing 14: *Ant* `build.xml` *file*