

Neverlang 2 Reference Manual

Edoardo Vacchi

May 10, 2013

Contents

1	Getting Started With Neverlang	3
1.1	Introduction	3
1.2	Introduction to Neverlang	3
1.2.1	Modules and Slices	3
1.2.2	Grammars and parse trees	3
2	The Neverlang Language	5
2.1	Modules	5
2.1.1	Reference Syntax	6
2.1.2	Roles	7
2.1.3	Semantic Actions	8
2.1.4	The imports section.	11
2.1.5	The eval keyword	11
2.2	Slices	13
2.2.1	Endemic Slices	15
2.3	Language	17
2.3.1	Roles	17
3	Neverlang Tools	19
3.1	The Neverlang Compiler <code>nlgc</code>	19
3.2	The Neverlang Interactive REPL <code>nlgi</code>	21
A	Installing Neverlang	23
A.1	Setting up the environment	23
A.1.1	Configuring Ant	23
A.2	Examples	24

Preface

Chapter 1

Getting Started With Neverlang

1.1 Introduction

1.2 Introduction to Neverlang

1.2.1 Modules and Slices

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 a syntax definition and one or more of *roles*.

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.

1.2.2 Grammars and parse trees

Each module defines a portion of a grammar. When modules are composed together to obtain a language implementation, each grammar portion is merged with the other, and the resulting full grammar feeds a parser generator.

DEXTER is the parser generator for Neverlang. DEXTER is an extended LR(0) parser generator, which supports the empty word ε , (specified in Neverlang as the empty string ""). This apparently scarce power has been shown

empirically to be enough in many cases. An experimental lookahead generation procedure is available to generate LALR lookahead sets, and convert the extended LR(0) automaton in a LALR(1) DFA¹

Input evaluation. When an input string is given to a Neverlang language implementation, the input is first given to the generated parser, which returns a parse tree. The parse tree is then visited *depth-first* as mandated by the syntax-directed translation technique (see [1] for reference). However, the visit can be either carried out in postorder (default) or in preorder.

Postorder. Semantic actions are evaluated *after* the subtree rooted at the current node is visited. This is the default in Neverlang.

Preorder. Semantic actions are evaluated *before* the subtree rooted at the current node is visited.

¹If `l` is your `Language` instance, then use `l.getDexter().update()` to generate the lookahead sets.

Chapter 2

The Neverlang Language

In this chapter we will describe how the Neverlang language ...

2.1 Modules

```
module com.example.HelloName {  
  reference syntax {  
    Program ← HelloWorld;  
    HelloWorld ← Hello Name;  
    Hello ← "hello";  
    Name ← /[a-zA-Z]+/;  
  }  
  role(print-name) {  
    2.{  
      String userName = $4.name;  
      System.out.print("Hello, ");  
      System.out.println(userName);  
    }.  
    6.{  
      $6.name = #0.text;  
    }.  
  }  
}
```

Listing 1: HelloName Module

Modules are used to describe the syntax of a language feature. They usually include a portion of the grammar of the language (the *reference syntax* section) and a series of *roles*, that is, compilation phases, that contain

executable code. Each module contains *at least* the mandatory *reference syntax* section. This section is mandatory because semantic roles refer to the productions defined in the grammar.

2.1.1 Reference Syntax

The *reference syntax* section *defines* or *refers* a portion of a formal grammar. This syntax definition is called a *reference syntax* because it is not mandatory for slices to actually use the syntax that is being defined. Rather, the grammar serves only as a reference for the semantic roles that define semantic rules in the *role* sections.

Grammar Definitions. If it defines a grammar, then this section embeds a list of production rules enclosed within ‘{’ and ‘}’. Each production rule is ended by a semicolon (;) (see Listing 1)

```

⟨production⟩ ::= ⟨nonterminal⟩ ‘<.’ ⟨symbol-list⟩ ‘;’

⟨symbol-list⟩ ::= ⟨symbol⟩ ⟨symbol-list⟩ | ⟨symbol⟩

⟨symbol⟩ ::= ⟨nonterminal⟩
           | ⟨keyword-terminal⟩
           | ⟨regex-terminal⟩

⟨nonterminal⟩ ::= ⟨id⟩

⟨keyword-terminal⟩ ::= ⟨string-literal⟩

⟨regex-terminal⟩ ::= ⟨regex-literal⟩

⟨identifier⟩ ::= [a-zA-Z_-]+

⟨string-literal⟩ ::= ‘”’ any character except a double quote ‘”’

⟨regex-literal⟩ ::= ‘/’ any character except a slash, unless escaped ‘/’

```

Listing 2: Grammar for reference syntax

Identifiers. Identifiers are words containing only alphabetic characters, underscores, and possibly dashes. Dashes, however, are internally converted to underscores.

Nonterminals. Conventionally represented as capitalized identifiers in *Camel-Case*, although the underscore character ‘_’ is accepted by the parser.

The special nonterminal ‘_’ is used to designate patterns or keywords that should be internally ignored. E.g.:

```
// Ignore single line comments that begin with '#'
_ ← /#.*/
```

When you define a rule with ‘_’, you should always define a rule for whitespace, as well. Otherwise, the whitespace will be treated as significant.

```
// Ignore any kind of whitespace
_ ← /\s/
```

Keyword Terminals. A keyword terminal is a token that is found verbatim in the input string. For instance the **if** keyword. A keyword terminal is a quoted string. It is not possible to escape the double quote character (it would not really make sense, since “” usually is not a keyword in a language.)

The special keyword "" represents the empty word ε and it is used to indicate that a keyword may be optional (as in “ $\langle A \rangle ::= \text{‘keyword’} \mid \varepsilon$ ”)

Regex Terminals. A regex terminal is a token that represents a *class* of tokens in the input string. For instance, an identifier or an integer can be only defined as a regex. The syntax for regexes is the same for Java native regex engine. The regex literal is enclosed in slashes. If you need to put a literal slash in your regex, then you can escape it by prepending a backslash as in “\”¹.

Referring a Grammar in a Distinct Module. Sometimes it is useful to define a module with only semantic roles. In this case, the programmer should indicate that the roles in the module refer to some syntax definition that is found in another module. When it refers a formal grammar, the list of production is substituted by the keyword **from** followed by the name of another module where the formal grammar is actually defined.

For instance, code in Listing 3 instructs Neverlang that the current module is referring to the syntax in Listing 1.

2.1.2 Roles

A role is introduced by the keyword **role** and an *identifier* in parenthesis. For instance Listing 1 contains one single role called `print-name`. Each role defines one or more *semantic actions*. Each semantic action is introduced by a number, and the code for the semantic action is enclosed between the delimiters ‘.{' and ‘}.’.

¹yes, it looks funny

```

module com.example.AnotherModule {
  reference syntax from com.example.HelloName
  role(do-stuff) {
    1 .{
      $1.someValue = "an arbitrary string value";
      SomeType obj = $0.someObj;
    }.
  }
}

```

Listing 3: Using **reference syntax from**.

The number refers a nonterminal in the *reference syntax*. In Listing 1, nonterminal 0 is `Program`, 1 is `Hello`, 2 is `Name`, 3 is `Hello`, and so on. Please notice that this numbering does not stop at the production boundaries, but it is *global* for each nonterminal occurring in the grammar. This number is called the **hook** of a semantic action, and it represents the nonterminal to which the semantic action should be attached.

2.1.3 Semantic Actions

Code in the semantic action is a piece of regular Java code with only one extension; it is possible to refer to a nonterminal by number using the $\$K$ syntax, where K is an integer number, that is, the index of a nonterminal. We call this a **nonterminal reference**. In particular, let be J the index of the left-hand nonterminal of a production, and L be the last nonterminal of the same production, then $J \leq K \leq L$. For instance, consider Listing 3; since the *hook* is 1, then it is possible to use $\$0$, $\$1$, $\$2$ inside the same rule. We call this the **current rule**, and we say that 0, 1, 2 are *reachable* from the current hook (or the current action).

As mandated by the syntax-directed translation technique, it is possible to attach, write and read **attributes** from any nonterminal that can be referred within a semantic action using a dot-notation syntax. The type is automatically inferred from the variable to which the value is being assigned.

Languages in Semantic Actions. Neverlang does not force the programmer to use Java for writing semantic actions. Developers can implement translator plugins that mainly define the way nonterminal references are translated. Neverlang comes bundled with two language plugins: `java` (default), and `scala`. It is possible to switch to one or the other by annotating a module in various position as in Listing 4. If the module is annotated at the top, then the default language for the whole module becomes the one in

```

<scala>
module com.example.MultiLang {
  ...
  role(do-stuff) <java> {
    2 <scala> .{
      val str : String = $4.text;
      System.out.println(str);
    }.
    6 <template> .{
      I am free to write anything, even expressions: {{ 10+10 }} !
    }.
  }
}

```

Listing 4: Using different languages.

angle brackets. It is possible to specify a default that is local to a particular role, or switch to a particular language only in a single action, by annotating the code section.

The <template> language plugin. Beside <java>, and <scala>, another special plugin is available: <template>. The template plugin injects automatically the whole content of the semantic action into a string attribute of the current *hook*. For instance the semantic action hooked to nonterminal 2 causes the string “I am free to write anything, even expressions: 20 !” to be assigned to **\$2**.text. As you may have noticed, text between {{ and }} can be any regular Java expression, but it may even contain nonterminal references².

Attribute Types. Attributes are *typed*, but they are internally stored in a hash map `String → Object` therefore casting is actually done internally. The casting mechanism relies on Java’s internal type inference mechanism.

```

$0.aString = "foo";
String theString = $0.aString;

```

However, no type check is done on the actual type of the attribute, therefore it is up to the programmer to assign and pass around values to variables that have the correct type. In case of mismatched type you will get a `ClassCastException`.

²As substitution of expressions is performed using a pattern matching procedure similar to that described above, similar pathological cases apply (e.g., “begin {{ “bad ” string” }} end”)

```

$0.anArray = new String[] { "foo", "bar" } ;
Object o = $0.anArray; // no complaint
CharSequence charseq = $0.anArray; // ClassCastException

```

Accessing an undefined value causes the system to throw an exception.

```
String foo = $0.anUndefinedValue; // throws an UndefinedAttributeException
```

Internal Implementation. Nonterminal references are translated into invocations to methods of the class `ASTNode`. The translation is carried out by way of *simple pattern matching*. In the case of the assignment, the pattern captures everything between (including) the nonterminal reference and a semicolon. This is particularly apparent in cases such as:

```

$0.aStringVal = "A pathological case; don't do this";
$0.anIntVal    = /* Another pathological case; too bad */ 15;

```

which will get translated to the syntactically incorrect:

```

.setValue("aStringVal", "A pathological case); don't do this";
.setValue("anIntVal", /* Another pathological case); too bad */ 15;

```

This limitation is unavoidable, unless we resort to actually parse the Java code inside the semantic actions. We chose not to, to be able to embed arbitrary languages inside the semantic actions without needing to be able to actually parse it (e.g., Scala).



Caveats

- Java versions prior to 8 are not sufficiently smart to infer the type in the following case

```

$0.aStringValue = "10";
int i = Integer.parseInt($0.aStringValue); // type error

```

You are supposed to either cast the field explicitly, call `$0.aStringValue.toString()` or hint the compiler by first “extracting” the attribute value to a variable.

- Because the internal hash map contains object references, primitive types are automatically boxed (e.g, `ints` become `Integers`). Boxed types are reference types; thus, as opposed to primitive types, they are nullable, and therefore partially incompatible with their unboxed equivalents.

```

$0.anIntVal = 10;
int myInt = $0.anIntVal; // the compiler will complain
Integer myInt = $0.anIntVal; // the compiler is happy

```

2.1.4 The `imports` section.

At the beginning of the module definition, before the reference syntax section you can insert the optional `imports` section.

```
module a.b.C {  
  imports {  
    java.util.List;  
    neverlang.utils.*;  
  }  
  ...  
}
```

Listing 5: The `imports` section.

This section works more or less like Java imports. Each line contains either a qualified type name or a qualified package name followed by a “*”, in case you want to import every type in that package. When you import a type from a package, then you can use its name unqualified in every semantic action (whichever the language you are using for the action: the translator plugin takes care of everything). For instance, the example in Listing `reflst:module-imports` is importing the `List` interface, but not any of its siblings (such as, for instance, `ArrayList`), and all of the types inside the `neverlang.utils` package.

There is no equivalent for the `import static` directive.

2.1.5 The `eval` keyword

As we saw in Sect 1.2.2, the default evaluation strategy for semantic actions is *postorder*: in other words, each rule is evaluated **after** the subtree rooted at the given hook has been already visited. When the visit is in *preorder*, the action is evaluated *before* the subtree is actually visited. Most importantly, in Neverlang the choice to proceed with the visit is left to programmer. This is the reason because when the tree is being visited in *preorder*³, Neverlang makes available the `eval` keyword. The programmer can write

```
eval $N;
```

to tell Neverlang to *immediately* proceed with the visit to nonterminal *N*. That is, current semantic action is interrupted, and the corresponding subtree is visited. Once the visit of the subtree completes, then the execution

³this is not currently enforced: the developer may write `eval` even if the tree is being descended in *postorder*

```

module com.example.TestEval {
    ...
    role(test-eval) {
        0 @{
            System.out.println("Yay, my children are evaluated automatically!");
        }.
        2 .{ eval $4; }. // skip eval'ing $3
        3 .{ System.out.println("I will be executed anyway."); }.
        5 .{ System.out.println("Nobody loves me."); }.
        6 .{ $6.text = "yay, this is fun !"; }. // no children, skip eval
    }
}

```

Listing 6: The eval directive.

returns to the point where it left⁴.

Usual scoping rules for N apply: it must be reachable from the current action. Since no enforcement is done on this, one can also enter evaluation loops, forcing the system to re-visit a node (this is by design: suppose you want to interpret a loop construct).

Example 1 Consider the following grammar:

reference syntax { $A \leftarrow B C$; $B \leftarrow \text{"b"}$; $C \leftarrow \text{"c"}$; } }

then, if no semantic action is attached to 0, the visit automatically proceeds as follows: 0, 1, 3, 2, 4. Conversely, if a semantic action is attached to 0, then the evaluation stops, and does not proceed to 1, 3, 2 and 4, unless eval \$1 or eval \$2 occurs in the semantic action hooked at 0.



Caveats

Consider Example 1. If you hook nonterminal 1 and the action is actually executed, then also any action attached to 2 will execute. Although possibly counterintuitive, the reason is quite simple: in the parse tree, 1 and 2 are actually the same node!

The @ shorthand.

You will find often that even if the evaluation is supposed to be mainly in preorder, sometimes it would be handy to resort a postorder semantics

⁴This is actually implemented by inserting a call to a method that causes the visiting procedure recurse immediately.

for a particular nonterminal. Luckily, it is easy to simulate a postorder in a preorder, by adding a simple preamble to the semantic action: you should `eval` each nonterminal in the subtree rooted at the hook. You can do this manually, or by instructing Neverlang to insert automatically these `eval` directives automatically by using the semantic action delimiters '@{' and '}' (see Listing 6).

2.2 Slices

Slices are used to pick features from different modules and join them together, but also to expose features to the language construct. Slices contain only one section. The first statement is required to be about the syntax of the slice. Every other statement describes the semantic actions that the slice is exposing.

```
slice com.example.HelloNameSlice {
  concrete syntax from com.example.HelloName
  module com.example.HelloName with role print-name
  module com.example.AnotherModule with role do-stuff
}
```

Listing 7: A simple slice.

The slice contains

- **concrete syntax from** <qualified.module.Name>
- one ore more module statements of the form:

```
module <some.other.module.Name> with role <role-name-1> <role-name-2> ...
```

role names can be one ore more for each module. The order is not relevant here.

For instance, Listing 7 describes a slice that is joining together the role `print-name` from `com.example.HelloName` and role `do-stuff` from the module called `com.example.AnotherModule`. The first statement declares the *concrete syntax* for the slice. The concrete syntax is defined as a reference syntax inside some module. In this case, `com.example.HelloName`. The module from which the concrete syntax is imported is not required to occur in the other statements.

Reference Syntax vs. Concrete Syntax. You may have notice that modules call the syntax section a *reference*, while in slices the syntax is *concrete*. Why is that? Modules define a syntax portion that may or may

not be actually used inside a slice. In fact, one may define a syntax section in module M and refer it within some role R , but then a slice S is free to choose a syntax compatible with M ⁵ from another module M' and apply the role R to the reference syntax of M' . Then the reference syntax in M becomes the *concrete syntax* for slice S (although R has been defined with the syntax in M as its reference implementation).

Tips & Tricks

Modules are slices too. What if your module M contains all of the pieces (syntax and roles) that you need to define a slice? Do you really need to write a slice just to declare the reference syntax in M to be concrete and all of the roles contained in M to be imported in a language? You do not. In fact, every module defines an *implicit slice* that declares all of the components contained in the module. For instance, module `com.example.HelloName` in Listing 1 implicitly defines a slice called the same. The result is the same as if you write a slice like the one in Listing 8^a.

^aNever really define a slice with the same name of a module: the generated Java source files would be overwritten!

```
slice com.example.HelloName {
  concrete syntax from com.example.HelloName
  module com.example.HelloName with role print-name
}
```

Listing 8: The slice implicitly defined by the module in Listing 1.

Remapping. Imagine that you have two modules with different syntaxes but with the same meaning. For instance, consider Perl’s `if` control flow statement, and the `if` statement modifier:

```
if ($i > 10) { print "Hello" ;} # if statement
print "Hello" if $i > 10 ; # statement modifier
```

The first statement could be described by the reference syntax of the `com.example.IfStatement` in Listing 9, and the second can be described by the reference syntax of `com.example.IfStatementModifier`. The semantics of the first module is given in the module `com.example.IfStatementSemantics`, and syntax and semantics are tied together in the slice `com.example.IfStatementSlice`. However, we do not need to write a new module to describe the semantics of the `if` statement modifier, because we can remap nonterminal 2 onto

⁵that is, such that you can there exist an isomorphism between parse trees

1 and 1 onto 2 using the `remap` construct. This way we can define the `com.example.IfStatementModifierSlice` without writing a single new line of semantics.

```

module com.example.IfStatement {
  reference syntax {
    IfStatement ← "if" "(" IfCondition ")" Statement ;
  }
}
module com.example.IfStatementModifier {
  reference syntax {
    IfStatementModifier ← Statement "if" "(" IfCondition ")" ;
  }
}
module com.example.IfStatementSemantics {
  reference syntax from com.example.IfStatement
  role(evaluation) {
    0.{
      eval $1;
      Boolean condition = $1.value;
      if (condition) { eval $2; }
    }.
  }
}
slice com.example.IfStatementSlice {
  concrete syntax from com.example.IfStatement
  module com.example.IfStatementSemantics with role evaluation
}
slice com.example.IfStatementModifierSlice {
  concrete syntax from com.example.IfStatementModifierSemantics
  module com.example.IfStatementSemantics with role evaluation
  mapping { /* 0 => 0, (implicit) */ 1 => 2, 2 => 1 }
}

```

Listing 9: Remapping the semantics of a module onto the syntax of another.

2.2.1 Endemic Slices

Endemic slices are a particular kind of slice that define an object that should be available in every semantic action.

They can be referred from within any semantic action using the special syntax `$$TypeName`. For instance, the endemic slice in Listing 10 defines the object instances `$$SomeType` and `$$AnotherType` of type `SomeType` and `AnotherType` respectively.

A fresh instance of these objects is created each time a new string is

```

endemic slice com.example.FooEndemic {
  declare {
    SomeType : my.pkg.SomeTypeImplementation
    AnotherType .{ some.pkg.Factory.getInstance() }.
  }
}

```

Listing 10: Endemic slices.

parsed (memory is lost between each parsing operation). The constructor for these instance can be either specified *implicitly* using the first form in Listing 10, or explicitly using the extended form (second line).

The **implicit form** assumes that the type name on the right is a concrete type that can be instantiated using an implicit constructor, and causes the declaration to be translated to (pseudo-code):

```
$$SomeType = new my.pkg.SomeTypeImplementation();
```

The **explicit form** assumes expects a valid Java *expression* (i.e., no semicolons, and it must return a value) within the code delimiters ‘{’ and ‘}.’. In this case the expression is took verbatim:

```
$$AnotherType = some.pkg.Factory.getInstance() ;
```

Because of their simple nature, endemic slices have no imports section⁶, and therefore every type name, beside those on the right shall be unqualified.

Endemic slices are currently implemented as attributes of the root node of the parse tree. If \$\$*T* is the name of the object instance, then it can be retrieved from attribute of the root node called *T* (notice the single dollar sign).



Caveats

The syntax mandates `TypeName` to be a valid identifier (see Listing 2); since identifiers do not include dots, it follows that this type name is always unqualified. Therefore, remember to always import this type name in your modules using the **imports** section!

static instances. Object instances are thrown away for each input that is given to the generated compiler. Occasionally, you may want *not to discard* these instances, and rather retain them for the whole duration of the execution of the compiler. Typical examples would be a compatibility table between types, or a table for type coercions, etc. In this case you can just premit the keyword **static** before the name of the object, as in :

⁶at least, in current implementation

```
static Foo : a.b.Bar
```

This way, the `$$Foo` object reference will always point to the same object instance throughout every execution of the compiler, until the program will naturally quit.

2.3 Language

```
language com.example.HelloLang {
  slices com.example.HelloName
    ... // other qualified slice names

  // optional
  endemic slices com.example.FooEndemic

  // first role is always syntax
  roles syntax < print-name < ... // other roles
}
```

Listing 11: Language.

The first statement of the **language** construct is **slices**, followed by a list of qualified slice names.

At least one slice should contain concrete syntax that includes the definition of a `Program` nonterminal (e.g., Listing 1). This nonterminal always represent the root of a program written in the language that is being defined.

Optionally, the second statement may be a list of endemic slices introduced by the keywords **endemic slices**.

The last statement lists roles in the order they will be evaluated.

2.3.1 Roles

The **roles** keyword introduces a list of identifiers for role names. The first role is obligatorily **syntax**, the others are custom. Between each role name a '`<`' shall appear, as in:

```
roles syntax < first < second < third
```

Postorder and Preorder. The default evaluation order is **postorder** (see Sect. 1.2.2). However, it is possible to select the preorder strategy with manual descent by premitting x '+' sign before the role name, as in:

```
roles syntax < first < +second < third
```

Or, if you prefer:

```
roles syntax < first <+ second < third
```

Juxtaposition (Layering). Roles are evaluated as subsequent visits of the parse tree. Each role restarts from the root until the visit ends. However, it is possible to “layer” several roles on top of each other, so that the evaluation is interleaved. The syntax is:

```
roles syntax < first:second < third
```

That is, instead of ‘<’ use ‘:’. In this case, for each node, the action for role `first` will be evaluated, and then, immediately, for the same node, action for `second` will execute. Then, as usual, the role `third` will execute from the beginning to the end.



Caveats

Because of the special nature of the **preorder** evaluation strategy, it is *not advisable* to use layering together with preorder, as you would probably obtain unexpected results.

Chapter 3

Neverlang Tools

In this chapter we will describe ...

3.1 The Neverlang Compiler `nlgc`

The new Neverlang compiler `nlgc` has been developed using the Neverlang runtime (Sect. ??) 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 `com.example.HelloName` in Listing 1 is translated into 4 independent but logically related classes:

1. `com.example.HelloName`, which lists each sub-component
2. `com.example.HelloName$role$syntax`, which describe the syntactic part of the module
3. `com.example.HelloName$role$print_name$2`, because a semantic action has been bound to the 2-nd nonterminal in the `print-name` role (notice how the dash has been translated to an underscore)
4. `com.example.HelloName$role$print_name$6`, because a semantic action has been bound to the 6-th nonterminal in the `print-name` role

As we will see in Sect. ?? 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 runtime was to have very few system requirements. Thus, the Neverlang 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 4 shows an example of the syntax.

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

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

Where `Context` is a class that contains a reference to the node that is being currently visited (`$ctx.node()`)

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 ?? 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. ??)¹.

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 `$ctx.node().nchild(int)`). Currently we have implemented support for Java and Scala. Listing 12 shows how this is done for Java. Code for Scala is similar. The plugin itself can be written in any JVM-supported language.


```

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

        // ( other definitions are omitted here for brevity )
    }
}

```

Listing 12: Translator Plug-in for Java

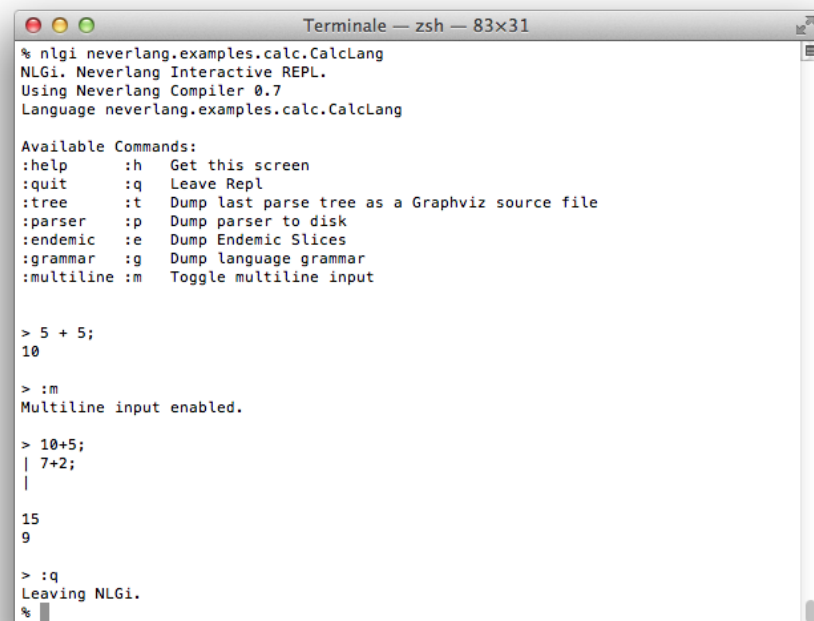
3.2 The Neverlang Interactive REPL *nlgi*

Neverlang comes bundle with the *nlgi* interactive REPL (Read-Eval-Print Loop). The *nlgi* tool can be launched from you shell using the command

```
% nlgi my.nlg.lang.LanguageName
```

The result is an interactive environment where you can test interactively the body of a program using a compiler implemented with Neverlang. In Fig. 3.1 the *calcLang* example (in the *NeverlangExamples.jar* archive)

¹Attributes are implemented as a map *attribute* \rightarrow *value* attached to each node. Setting or getting is implemented as a method call. See Listing 12.



```
% nlgi neverlang.examples.calc.CalcLang
NLGi. Neverlang Interactive REPL.
Using Neverlang Compiler 0.7
Language neverlang.examples.calc.CalcLang

Available Commands:
:help      :h  Get this screen
:quit      :q  Leave Repl
:tree      :t  Dump last parse tree as a Graphviz source file
:parser    :p  Dump parser to disk
:endemic   :e  Dump Endemic Slices
:grammar   :g  Dump language grammar
:multiline :m  Toggle multiline input

> 5 + 5;
10

> :m
Multiline input enabled.

> 10+5;
| 7+2;
|

15
9

> :q
Leaving NLGi.
%
```

Figure 3.1: `nlgi` running the `CalcLang` example language

Appendix A

Installing Neverlang

In this chapter we will go through the steps that are necessary to set up your build environment in order to use Neverlang.

A.1 Setting up the environment

Neverlang comes conveniently bundled in a tarball. Unpack the tarball in your preferred location. A common one is `$HOME`. Set the environment variable `$NEVERLANG_HOME` to point to this directory, and add `$NEVERLANG_HOME/bin` to your `$PATH`¹. Usually you would carry on this by adding to `~/.profile`, `~/.bashrc` or similar:

```
export NEVERLANG_HOME=$HOME/path/to/neverlang
export PATH=$PATH:$NEVERLANG_HOME/bin
```

On Windows, you can open the environment variable configuration panel and add these variables in.

```
NEVERLANG_HOME: %UserProfile%/path/to/neverlang
PATH: ... other entries ... ; %NEVERLANG_HOME%
```

If Neverlang has been set up correctly, every new instance of your favorite terminal emulator (or command prompt) should be able to reach the `nlgc` command. In this case you should see this output:

```
$ nlgc
No input file specified.
```

A.1.1 Configuring Ant

Neverlang comes bundled with an Ant task that you can conveniently add to your Ant build files to automate the generate and compile process. The

¹Alternatively, you can also copy `bin/nlgc` to some directory that may already be in `$PATH`, such as `/usr/local/bin`

easiest way to set this up is to drop (or symlink) at least `Neverlang.jar`, `Dexter.jar`, and `Lexter.jar` in `~/.ant/lib`. Another option is to pass the `-lib` option to the ant processor with the full path to these jars (they are all in `$NEVERLANG_HOME` and `$NEVERLANG_HOME/lib`).

A.2 Examples

Unpack the examples. Compile with `ant`.

Bibliography

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