TEA TEMPLATE LANGUAGE

AUGUST 27, 2001

PLEASE SEND ALL COMMENTS TO: OPENSOURCE@DIG.COM

PREFACE

This document introduces the Tea template language to Java developers. It covers all aspects of Tea, but it is not quite a complete language specification. Because of Tea's open source status, a reference implementation is available which is also the standard implementation. The source code, being freely available, is the most concise specification. A separate architecture document guides those who wish to peruse the source code.

Tutorials on template writing and writing implementations that use Tea can be based on information contained within this document. This document will evolve both to clarify features and to reflect Tea's evolution. Use of specific products that integrate Tea is outside the scope of this document.

There are a number of other documents complementary to this reference that cover topics associated with Tea and Tea-related tools. For additional Tea-related documentation, please visit http://opensource.go.com.

DOCUMENT REVISION

Primary Author(s)	Description of Version	Date Completed
Brian S O'Neill	Initial public release.	07-30-2000
Michael Rathjen	Minor update.	08-12-2000
Michael Rathjen	Reorganized content.	09-28-2000
Michael Rathjen	Minor errors corrected.	12-12-2000
Michael Rathjen	Minor errors corrected.	03-01-2001
Michael Rathjen	Tea 3.1.5 functions added.	05-18-2001
Michael Rathjen	Break statement added.	06-26-2001
Michael Rathjen	Looping information updated for Tea 3.2.0.	08-27-2001

CONTENTS

1. INTRODUCTION

Tea is a strongly typed, compiled programming language designed to work within a Java-based hosting environment. Tea's main goal is to provide an enforced separation between data processing and presentation, without sacrificing basic programming constructs. Tea modules are called templates, and they are used for data presentation. Templates are generally used for textual output and HTML, and thus text printing operations are designed to be as simple as possible. A template cannot directly acquire or modify information. It relies on its host to provide data. Data access operations are as important as printing output and are also very simple.

Tea's basic constraints:

- Data or data types cannot be directly created. They are acquired.
- Acquired data cannot be directly modified in any way.
- A template cannot directly cause harm to its hosting environment.
- Only the minimum amount of programming constructs is provided.

Because of the above constraints, Tea cannot do anything useful without a hosting environment. Tea's host must be Java-based, and all data are treated as objects. Properties are accessed from objects by treating them as JavaBeans. The restricted programming model forces complex programming tasks to be performed in Java, and this is how presentation is kept separate from processing.

Here's a very basic "Hello World" template that reads a message property from an object:

```
<% template HelloWorld(ContentObject obj) %>
Hello <% obj.message %>!
```

The "<%" and "%>" symbols delimit code regions, and text not in any code regions is printed as is. The above template may also be written in a single code region as:

```
<% template HelloWorld(ContentObject obj)
"Hello " obj.message
%>
```

The way in which an instance of ContentObject is passed into the above template is left up to the host. Often, data is acquired through custom functions, which must be written in Java.

The set of programming constructs provided in Tea are designed to resemble Java in syntax, except semi-colons are not required to terminate statements. Like Java, the Tea character set is Unicode. Tea has an if statement, just like Java's. Tea does not, however, have a switch statement, as it is not necessary. Looping in Tea is provided via a `foreach` statement. There is not any `while` or `for` statement.

With the exception of passed-in parameters, local variables are not (and cannot) be declared against a type. Variable declarations are implicit. Variables can use primitive types or object references, but all data appears as if they were objects. Yet Tea is still strongly and statically typed.

Tea templates are compiled directly to Java byte code, without the use of a Java compiler. Tea does not use any special kind of interpreter. Tea templates perform as well as hand-written Java code. Any runtime exception caused while executing a template contains a stack trace that identifies which template was executing and what line number was executing.

Because Tea must be integrated into a hosting environment, the implementation is very modular at both the compiler and runtime levels. Tea templates can be used to generate dynamic or static HTML pages. There is no restriction on how data is output. Most Tea environments that support running of Tea templates also support safe, dynamic template re-compilation. Any compilation errors are reported in a fashion that best suits that environment. A web-based compilation page, for example, will show detailed compilation errors on the web page.

1.1 Why Use Tea

In general, neither developers nor page designers author Tea templates. The goal is that they be written and maintained by technical producers who are liaisons between developers and designers.

Tea resulted from several years of experience with other web page building mechanisms. Most web-based applications start out with HTML tags embedded in code, whether it be C, Perl, or Java. This approach is adequate for small or first-time projects because it doesn't take very long to develop.

Because changes to page formatting can occur frequently, and developers don't wish to make these changes, they inevitably evolve into using some kind of token replacement templating mechanism. Each token is just a placeholder for a string, which contains application-created data. These template systems further evolve into supporting special constructs for formatting tables, forms, and simple conditional logic.

When introducing programming constructs into a template, the challenge is to come up with something that is powerful enough, yet at the same time be simple and safe. If its too powerful, then complete applications could be developed in templates. If it is too weak, then HTML formatting ends up in the application. If it isn't simple or safe, then application developers end up writing and maintaining templates.

Rather than embedding an existing language into something like an ASP or JSP, Tea is a language specially designed to meet the requirements of a templating system. It is safe, simple, efficient, and powerful.

In one instance, Tea is integrated with a special servlet. This servlet gives Tea templates control over page building, while retaining strong ties to a back-end application written by a Java developer. While this servlet provides functionality similar to that of JSPs, Tea enforces correct model-view separation because of the intentional language limitations. Although this is also the suggested separation model in JSPs, it cannot be enforced. In addition, Tea templates don't support programming features that can be used irresponsibly. Modifications need not go through a strict review and testing phase, which would still be required for JSPs.

Everyone working on a project should be empowered to do their job the most effectively, and Tea does its part by letting you do exactly what you need, as easily as possible, and no more. Even on projects run by just developers, using Tea is still beneficial. It encourages good development practices and it makes applications easier to maintain.

2. JAVA INTEGRATION

Tea is designed to work within a Java environment, and the Tea language itself is patterned after the Java language. Tea accesses properties from objects that follow the JavaBean conventions, and because Java is case-sensitive with respect to names, so too is Tea. Standard Java objects are used in Tea for strings, numbers, collections, etc.

Tea differs from Java in many respects, mostly due to the reduced set of features. Variables in Tea are implicitly types, but passed-in parameters are specified just like in Java. Tea supports both primitive types and objects, but from the template they all appear as objects. Tea converts between primitive types and their peers (int ↔ Integer) automatically. Other special type conversion rules (see section 6 Type Conversion) are also applied automatically.

Tea templates compile into Java class files and execute just the same within a Java Virtual Machine. The Tea compiler does not generate intermediate Java files and hand them off to a Java compiler, but it instead generates the class files directly. Still, the resulting class files are as efficient as any handwritten Java code.

Because no Java compiler is used, Tea can be distributed much more easily. All that's required is a Java2 runtime environment. Sun's JRE can be used, but the SDK isn't necessary.

When compiled to class files, the Tea source file information, including line numbers, is preserved. If an exception is thrown from a template or from a method called by a template, an entry appears in the stack trace referring directly to the template. The source file information contains the ".tea" extension and the line number matches up directly with the templates source file. Tea templates should be compatible with most Java debugging environments.

Tea isn't designed as a standalone system. Whereas a Java program can be launched from a main method in a class, this concept doesn't really make sense in Tea. Tea requires a hosting system to be compiled to and run in.

The Tea implementation basically supports two components: the compiler and the runtime. A host environment may integrate with either of these components or with pieces of them. Integrating with both components results in a complete Tea integration.

Creating a new host environment requires some knowledge of the Tea implementation. The following is an example of a very simple complete integration, but it is enough to get one started in creating a custom host environment. Most developers, however, will use an existing environment like the TeaServlet, and won't need to know how to create a host environment.

```
import java.io.File;
import java.io.IOException;
import com.go.tea.runtime.TemplateLoader;
import com.go.tea.runtime.DefaultContext;
import com.go.tea.util.FileCompiler;
import com.go.tea.util.ConsoleErrorReporter;
import com.go.trove.util.ClassInjector;


public class SimpleTea {
    /**
     * Test program executes a template named Test that accepts a
     * single String argument.  For example,
     * <pre>
     * <% template Test(String arg) %>
     * Message is: <% arg %>
     * </pre>
     */
    public static void main(String[] args) throws Exception {
        SimpleTea tea = new SimpleTea(new File("."), "simple");
        String result =
            tea.executeTemplate("Test", new Object[] {"Hello World"});
        System.out.println(result);
    }

    private String[] mNames;
    private int mErrorCount;
    private TemplateLoader mLoader;

    public SimpleTea(File rootSourceDir, String rootPackage)
        throws IOException
    {
        ClassInjector injector = new ClassInjector((File)null, rootPackage);
        FileCompiler compiler = new FileCompiler
            (rootSourceDir, rootPackage, null, injector);
        compiler.setRuntimeContext(SimpleContext.class);
        compiler.addErrorListener(new ConsoleErrorReporter(System.err));
        compiler.setForceCompile(true);
        mNames = compiler.compileAll(true);
        mErrorCount = compiler.getErrorCount();
        mLoader = new TemplateLoader(injector, rootPackage);
    }

    public int getErrorCount() {
        return mErrorCount;
    }

    public String[] getTemplateNames() {
        return (String[])mNames.clone();
    }

    public TemplateLoader.Template getTemplate(String name)
        throws ClassNotFoundException, NoSuchMethodException
    {
        return mLoader.getTemplate(name);
    }

    public String executeTemplate(String name, Object[] parameters)
        throws Exception
    {
        StringBuffer buf = new StringBuffer(256);
        SimpleContext context = new SimpleContext(buf);
        getTemplate(name).execute(context, parameters);
        return buf.toString();
    }

    public static class SimpleContext extends DefaultContext {
        private StringBuffer mBuffer;

        public SimpleContext(StringBuffer buffer) {
            mBuffer = buffer;
        }

        public void print(Object obj) {
            mBuffer.append(toString(obj));
        }
    }
}
```

Since Tea templates have no direct way of obtaining data, a host environment is responsible for providing it. There are two ways in which this can be done. The host may pass in objects to the template parameters and it can provide a custom context that supplies functions.

A context is a special class that Tea templates require for execution. It defines functions that are callable by templates and it also provides the methods for templates to output data. The print functions can be called directly, but templates usually don't do this. See the section on functions (5.1 Calling Functions and Templates) and expression statements (4.1.7 Expression Statement) for more information.

The SimpleTea example doesn't provide any special context functions other than the required print method and defaults, but it does pass in objects to the template via its parameters. Once a template has obtained an object, it can access bean properties and objects from it.

Although Tea is designed primarily for textual output, you may notice that there are no restrictions on how a context can be defined and what objects the print method may contain. Tea-based systems can also be created for producing image data or even for building GUIs. The ability to pass in a block of code to a function also helps in this capacity (see section 4.1.5 Substitution Statement).

3. Lexical Structure

The lexical structure of Tea is based on Java's. The chapter titled "Lexical Structure" in *The Java Language Specification*, defines most of Tea's lexical structure.

Tea templates are defined in the unicode character set and the language is case-sensitive, just like Java. Tea uses Java's style and early processing of unicode escapes. Comments and white space are the same. Tea identifiers do not allow '$' characters, but are otherwise the same. String and number literals are very similar.

A template is composed of code and text regions. Code regions are delimited by "`<%`" and "`%>`" symbols. Any text not in the code region is output by the template as-is with only one minor conversion. All line break separator codes in text regions are converted to linefeeds `'\n'` (ASCII 10). No string escapes (including unicode escapes) are processed in text regions. Here is a template that mixes code and text regions:

```
<% template MyTemplate(PageInfo page) %>
<html><head><title><% page.title %></title></head>
<body>
<h1><% page.heading %></h1>
<table>
       <% foreach (item in page.list) { %>
              <tr><td><% item.name %></td><td><% item.desc %></td></tr>
       <% } %>
</table>
</body>
</html>
```

The code delimiter symbols are not actually Tea lexemes, but are instead processed in an initial preprocessing step that identifies text as being in a code region or not. The Tea scanner converts text regions into string literals. Put another way, text regions are simply multi-line string literals delimited by "`%>`" and "`<%`" symbols (see section 3.2.3 String Literals).

3.1 Keywords

The following are the keywords in the Tea language. The reserved words "`null`", "`true`", "`false`", and "`isa`", discussed in the next sections, are not considered to be keywords.

Keyword:

```
and
break
call
else
foreach
if
in
not
or
reverse
```

`template`

3.2 Literals

Literal:

NullLiteral

BooleanLiteral

StringLiteral

IntegerLiteral

FloatingPointLiteral

3.2.1 Null Literal

NullLiteral:

`null`

3.2.2 Boolean Literals

BooleanLiteral:

`true`

`false`

3.2.3 String Literals

StringLiteral:

`"` *StringCharacters*$_{opt}$ `"`

`'` *StringCharacters*$_{opt}$ `'`

Tea string literals are for the most part, just like Java's. The first difference is that they may be delimited by apostrophes instead of quotes. Apostrophes are used by Java to delimit character literals, but Tea does not have character literals.

The other difference is that octal escapes are not supported in strings, although `\0` (the null character) is.

3.2.4 Integer Literals

IntegerLiteral:

DecimalIntegerLiteral

HexIntegerLiteral

Tea integer literals are just like Java integer literals except Tea does not support octal integer literals. As a result, decimal integer literals may start with any number of '`0`' digits.

3.2.5 Floating Point Literals

The only difference between Tea's floating-point literals and Java's is that the decimal point must always have a digit before and after it.

3.3 Separators

Separator: one of

```
(       )       {       }       [       ]
;       ,       .
```

3.4 Operators

Operator: one of

```
=       #       ##      ..      ...     isa
==      !=      <       >       <=      >=
+       -       *       /       %       &
```

3.4.1 Ellipsis Operator

The ellipsis operator "`...`" is used when declaring that templates require a substitution parameter, and it is used where block substitutions should be placed. In Tea, the ellipsis means, "more statements may follow."

4. Templates

A Tea compilation unit consists of a single template. If templates are stored on a file system, then each template file must contain exactly one template definition. The name of the file must consist of the template name followed by the extension ".tea".

Every template must begin with a template declaration, which is defined in a code region (see section 3 Lexical Structure). Therefore, every template must begin with a code region. The template declaration defines the name of the template and any parameters that must be passed to the template.

Template:

TemplateDeclaration StatementList$_{opt}$

TemplateDeclaration:

`template` *Identifier* (*FormalParameterList*$_{opt}$) *SubstitutionParameter*$_{opt}$

The template's formal parameter list follows the template name in the declaration and must be delimited by parentheses. A comma separates each parameter in the list. Formal parameters require a class name followed by an identifier, which names it. Tea does not allow primitive types to be declared, only object types. Object type names must be fully qualified except for those defined in the java.lang and java.util packages. The type declaration may define an array (or an array of arrays) just like in Java, using '`[`' and '`]`' symbols.

FormalParameterList

FormalParameter

FormalParameterList `,` *FormalParameter*

FormalParameter

Type Variable

Type:

Name

Type `[ ]`

After the template declaration, a template can have text regions or code regions with any statements allowed by Tea. Statements can also appear inside the same code region that the template declaration was in.

Templates can invoke other templates, just like calling a function. Any template can be invoked by another template. Templates may also return a value to the caller. No special syntax is required in the template declaration to support callable templates or return types.

A template may require that it receive one substitution block parameter. This parameter is denoted by the ellipsis symbol '. . .'. With this parameter, any template invoking this template must pass in a block of code. A block of code begins with a left curly brace and ends with a right curly brace. The template that accepts the substitution parameter uses the substitution statement to insert the substitution.

Examples:

From SmallTemplate.tea:

```
<% template SmallTemplate(pkg.ContentObject obj) %>
<html><body>
This page was created on <% obj.date %>.
<br>
<%
user = obj.user
if (user != null and user.name != null) {
    'Hello ' user.name '!'
}
%>
</body></html>
```

From NewsPage.tea:

```
<% template NewsPage(org.news.NewsStory story)
// Pass a title and a substitution block to SimplePage
call SimplePage(story.title, "#ffffff") {
    '<h1>' story.headline '</h1>'
    foreach (paragraph in story.body) {
        '<p>' paragraph.body
    }
}
%>
```

From SimplePage.tea, called from NewsPage:

```
<% template SimplePage(String title, String color) { ...  }
'<html><head><title>' title '</title></head>'
if (color == null) {
    '<body>'
}
else {
    '<body bgcolor="' & color & '">'
}
...
'</body></html>'
%>
```

4.1 Statements

Tea statements are used for variable assignments, controlling the flow of execution, performing looping, calling functions, and calling other templates. Statements are also used to output from a template and perform block substitutions. Statements need not terminate with a semi-colon, as they do in Java. Statements are not separated by new lines, but by a carefully designed language grammar. In very rare circumstances, a semi-colon is required to disambiguate statement separation.

Statement:

EmptyStatement

IfStatement

ForeachStatement

SubstitutionStatement

AssignmentStatement

CallStatement

ExpressionStatement

The set of operations that can be performed by Tea statements is less than that provided by most application programming languages. This aids in enforcing a policy that keeps templates from performing complex operations that should be left to a hosting system.

4.1.1 Semi-colons

Although Tea does not require the use of the semi-colon as a statement terminator, they can still be used. Java programmers are likely to use them out of habit or for stylistic reasons. Under certain special conditions, a semi-colon may be required to enforce statement separation. The following two statements

```
a
-b
```

would not be interpreted as printing the value of variable "`a`" followed by the negative value of variable "`b`". Instead it would be

```
a - b
```

which means print the value of "`a`" minus "`b`". This would not be a problem if the "`-`" token was not overloaded to mean both subtraction and negation. Here is the separated version:

```
a;
-b
```

Alternatively, parentheses can be used, but it is a clumsier notation:

```
(a)
(-b)
```

In this example, if the parentheses were omitted from the first expression, the compiler would decide that "`a`" is a function receiving parameter "`-b`".

4.1.2 Break Statement

The `break` statement can be used to exit from a loop. See 4.1.4 Looping.

Example:

```
foreach (count in 1..10) {
    count
    if (count > 4) {
        break
    }
}
```

Placing unreachable code after the break statement will result in an error at compile time.

4.1.3 If Statement

Tea's `if` statement is just like Java's except in one way: braces are always required for its enclosed statements.

IfStatement:

`if` `(` *Expression* `)` *Block* *ElseStatement*$_{opt}$

ElseStatement:

`else` *Block*

`else` *ifStatement*

Block:

{ *StatementList*$_{opt}$ }

Examples:

```
if (fileName == null or fileName.length == 0) {
    // Assign a default filename if empty
    fileName = "default"
}


if (name == "Bob") {
    // Bob is welcome...
    "Hello Bob!"
}
else {
    // ...others are not
    "Go away, " & name & "."
}


if (count == -1) { count = 0 } else if (count >= 15) { count = count - 15 }
```

4.1.4 Looping

The `foreach` statement is the only loop flow control statement in Tea. There is no way to modify the loop count variable from within the loop, but the `break` statement can be used to exit the loop. `Foreach` statements iterate over the values of an array, a collection or a range of values. When the optional reverse keyword is specified, the values are iterated in reverse order.

Like the `if` statement, blocks are required to be enclosed in braces.

ForeachStatement:

`foreach` `(` *Variable* `in` *Expression* `)` *Block*

`foreach` `(` *Variable* `in` *Expression* `reverse` `)` *Block*

`foreach` `(` *Variable* `in` *Expression* `..` *Expression* `)` *Block*

`foreach` `(` *Variable* `in` *Expression* `..` *Expression* `reverse` `)` *Block*

The variable used in the `foreach` statement is re-assigned during the loop's execution, and its type is redeclared to be the appropriate element type of the given expression. If the loop variable was already declared outside of the loop, then the new declaration is promoted outside of the `foreach` scope. See also sections 4.1.6 Variable Assignment and 6.4 Variable Promotion.

Example:

```
// Loop through and print every item in the page's list
foreach (item in page.list) {
    item
}

// Loop through in reverse
foreach (item in page.list reverse) {
    item
}

// Loop from 1 to 10, inclusive
foreach (count in 1..10) {
    "Count is " & count
}

// Print all the characters of a string
// NOTE: if the message length is zero, the range is 0..-1.
// Whenever the second value in the range is less than the
// first value, the foreach will loop zero times.
message = "hello"
foreach (index in 0..message.length - 1) {
    "Letter at index " & index & " is: " & message[index] & "\n"
}

// Loop from 10 to 0, inclusive
foreach (count in 0..10 reverse) { ...  }
```

Unless looping through a range, the actual object looped through can be an array or a Collection (`java.util.Collection`). Objects that implement the Collection interface cannot be iterated in the reverse direction, but those that implement the List interface (`java.util.List`) can be.

If looping over a value of ranges, they can only be integers. Both the beginning and ending of the range is inclusive. If a number at the beginning or ending of a range is not an integer, it is rounded down to the nearest integer.

Because Java has no parameterized types, when a Collection is used in the `foreach` loop, Tea can only infer that its elements are Objects, even if they may be subclasses of Object. If the template only needs to print the values of the elements, then this isn't a problem, but if the template needs a specific property of the object, access is denied. In addition, Tea has no *direct* cast operation.

There are several ways to handle this situation. The easiest thing to do is to pass an array to the template instead of a Collection. An alternative is to extend the Collection class being used and provide a special field that Tea uses to determine the element type. The Java signature of the field is:

```
public static final Class ELEMENT_TYPE
```

The compiled code then trusts that the elements are of the specified type and performs a blind cast. If the cast fails, a ClassCastException is thrown at runtime.

The least preferred technique is to use the "`isa`" operator in an `if` statement in order to perform a cast on the elements (see 4.2.5 "isa" Operator).

You can exit a loop by using the `break` statement (see 4.1.2 Break Statement).

4.1.5 Substitution Statement

The substitution statement is only allowed in templates that declare that they receive a block substitution parameter. Templates can declare at most one block substitution parameter. The substitution statement is used to substitute the code for the block passed in at the statement's location. The substitution statement is simply the ellipsis (`...`).

SubstitutionStatement:

```
...
```

A substitution statement can appear in multiple locations in a template and even in `foreach` and `if` statements. The most common use for a substitution is for creating "shell" templates, from the example earlier:

```
<% template SimplePage(String title, String color) { ...  }
'<html><head><title>' title '</title></head>'
if (color == null) {
    '<body>'
}
else {
    '<body bgcolor="' & color & '">'
}
...
'</body></html>'
%>
```

A substitution can also be passed to a context-defined function, using the special `com.go.tea.runtime.Substitution` class. This example defines a simple looping function:

```
public void loop(int count, Substitution s) throws Exception {
    while (--count >= 0) {
        s.substitute();
    }
}
```

The template might invoke this function as:

```
loop (100) {
    "This message is printed 100 times\n"
}
```

4.1.6 Variable Assignment

With the exception of passed-in parameters, Tea variables are not declared. Rather, their type is inferred based on assignment. A variable's type is dynamic, and can change after another assignment. Passed-in parameters behave like ordinary local variables in that they can be re-assigned, and they can change type. Variables can also be assigned by a `foreach` statement (see 4.1.4 Looping).

AssignmentStatement:

Variable = *Expression*

Assignments can only be made to variables. Array elements cannot be assigned values, like they can in Java. Assignments of this form are not allowed: `a[i] = x;` This restriction makes it possible to pass collections to templates without fear of the template modifying it.

Examples:

```
message = "Hello"          // Assign the string literal "Hello" to message
result = 50                // Assign the integer literal 50 to result
amount = result + 20       // Assign the calculated sum of result and 20
message = amount           // Re-assign message with the integer amount
```

Unlike Java, assignment statements cannot be expressions. This means that assignments cannot be chained together like they can in Java: `a = b = c = 2`.

4.1.7 Expression Statement

All expressions can be statements (see 4.2 Expressions). When an expression takes on the form of a statement, the return value is sent directly to the template's execution context. The execution context will typically take this value, convert it to a string, and write it to an output stream. This is the main way in which templates output data.

ExpressionStatement:

Expression

Any expression, which is not part of another statement, is automatically passed to the built-in print function as if it were done directly. The following two statements are equivalent:

```
"Hello world"
print("Hello world")
```

Calling the print function directly in templates is not preferred, and tools that display execution context functions should hide the print function.

If the last statement in a template is an expression statement, the expression results (complete with type information) are passed back to the calling template. The caller can then use the call statement as a call expression (see 5.1 Calling Functions and Templates). If the caller does not do anything special, then the call expression is treated as an expression statement, and the caller automatically prints out the results.

4.2 Expressions

An expression is any program fragment that produces some kind of value. Expressions are evaluated in a specific order, and parentheses can be used to override that ordering. A sub-expression bounded by parentheses is evaluated first within a sub-expression. Otherwise, the order of operations is the same as that for Java, wherever applicable. See section 7 Grammar Summary for the order of operations.

4.2.1 Accessing Properties

All objects have properties, and these properties are accessible using the JavaBeans Introspector. Essentially, any object passed to or used in a template is a JavaBean. JavaBeans defines two kinds of properties, non-indexed and indexed. Tea can access all non-indexed properties on a JavaBean, but can only access a certain kind of indexed property.

Defining a non-indexed property in a Java class is really simple. The method must return something, must take no parameters, and its name must start with "`get`". The part of the method name that follows "`get`" defines the name of the property. Capitalization rules are applied so that the method `getSizeOfThing` defines the property "`sizeOfThing`" and the method `getURL` defines the property "`URL`".

Arrays, Collections and strings also have a special property named "`length`" which isn't defined by a `get` method. For arrays and Collections, length is the number of elements they contain. For strings, it is the number of characters they contain.

JavaBeans defines several different forms for indexed properties, but Tea only understands one kind: unnamed indexed properties. A method that returns something, takes a single parameter and is named "`get`" follows the design rules for unnamed indexed properties. Java's Map and List interfaces follow these rules, and Tea accesses elements from them via this design pattern.

LookupExpression:

Factor Lookup$_{opt}$

Lookup:

Lookup$_{opt}$ `.` *Identifier*

Lookup$_{opt}$ `[` *Expression* `]`

A property access is called a lookup, and Tea has different syntax for its two supported kinds of lookups. A lookup can be applied to any expression, but it is usually applied to variables or chained to other lookups. Non-indexed properties are looked up from an expression using a dot (`.`) followed by the property name. Indexed properties are looked up using an expression bounded by square brackets.

If the type of an indexed property is Object, (like it is in Map and List) and a element type is defined for that class (by extending Map or List), then Tea will cast the object to that element type. An element type for a class that has an indexed property is defined the same as for

Collections described by the foreach statement (see 4.1.4 Looping). The Java signature of the field is

```
public static final Class ELEMENT_TYPE
```

Like its use in `foreach`, if an element is returned that can't be cast to the specified element type, a `ClassCastException` is thrown at runtime.

Examples:

```
// Access the name object from user
name = user.name
// Access the first name
f = name.first
// Access the last name by chaining
last = user.name.last

// Access an element from the users array and get the age
"Age is " & users[10].age

// Lookup a team object by code
team = allTeams["SEA"]

// Get the last character from a string
str[str.length - 1]
```

4.2.2 Arithmetic

Tea can perform arithmetic on both integers and floating-point numbers. Supported operations are addition, subtraction, multiplication, division, division remainder and negation. The operators are `+`, `-`, `*`, `/`, `%`, and `-` respectively. In complex arithmetic expressions, negate is performed first, left-to-right. Multiplication, division, and remainder are performed next, followed by addition and subtraction. This order of operations matches Java's.

AdditiveExpression:

MultiplicativeExpression

AdditiveExpression `+` *MultiplicativeExpression*

AdditiveExpression `–` *MultiplicativeExpression*

MultiplicativeExpression:

UnaryExpression

MultiplicativeExpression `*` *UnaryExpression*

MultiplicativeExpression `/` *UnaryExpression*

MultiplicativeExpression `%` *UnaryExpression*

Examples:

```
// Simple addition
x = a + b

// Increment a count
count = count + 1

// More complex example
result = -((value - 10.0/y) * 50) % 2
```

In order to perform a division on integer values, but produce a fractional result, adding `0.0` to at least one of the values forces a floating point division. The addition by zero is optimized away and is not actually performed.

```
x = 3
y = 2
// Integer division prints 1
x / y
"<br>"
// Floating point division prints 1.5
(x + 0.0) / y
"<br>"
```

4.2.3 String Concatenation

String concatenation in Java is performed with the plus operator. Tea uses the ampersand (&) operator instead because of the way the template language determines type information. Overloading the plus operator would create typing ambiguities.

ConcatenateExpression:

AdditiveExpression

ConcatenateExpression `&` *AdditiveExpression*

Examples:

```
// Equivalent to text = "Hello World"
text = "Hello" & " " & 'World!'

y = 5
'498234 divided by "y" is ' & 498234 / y
```

Output:

```
498234 divided by "y" is 99646
```

Both the left and right values of a concatenation are converted to strings (if they are not already strings) using the hidden `toString` function. Any formatting settings are applied when performing the string conversion.

String concatenation can also be used to force an object to be converted to a string. Concatenating the empty string forces a string conversion, but the actual concatenation operation is optimized away.

```
x = 43554
x = x & ""
// x is now a string, and 5 is printed out
x.length
```

4.2.4 Relational Tests

Tea supports expressions for performing relational tests. They are used most often in the context of an `if` statement condition, although their result can be assigned to a variable. The resulting type of a relational test is always `boolean`. Six operators are supported, and they fall into two categories: equality tests and comparison tests.

Equality tests are equals (==) and not-equals (`!=`), and they work for any kind of data type. When an equality test is performed against a string, the other operand is also converted to a string. The following test is legal and performs a string equality test: `5 == "5"`.

Whenever an equality test is performed on objects, the "`equals`" method is invoked is possible. If either operand is null, the Tea compiler ensures that no NullPointerException is ever generated, and it may not call the equals method at all if a test is being made against null. The following test will never invoke the equals method: x == null.

Comparison tests are less than (<), greater than (>), less than or equal (<=), and greater than or equal (>=). Comparison tests can be performed on numbers, strings, or any object that implements the `Comparable` interface. Both values in the comparison must be of the same type. Unlike equality tests against strings, comparisons against strings never force a string conversion.

A comparison between two numbers will perform a numerical comparison, and comparisons between all other objects is performed using the "`compareTo`" method defined in the Comparable interface.

EqualityExpression:

RelationalExpression

EqualityExpression `==` *RelationalExpression*

EqualityExpression `!=` *RelationalExpression*

RelationalExpression:

ConcatenateExpression

RelationalExpression `<` *ConcatenateExpression*

RelationalExpression `>` *ConcatenateExpression*

RelationalExpression `<=` *ConcatenateExpression*

RelationalExpression `>=` *ConcatenateExpression*

RelationalExpression `isa` *Type*

Examples:

```
// Check that the items are not null before looping through them
if (items != null) {
    foreach(item in items) {
        "Item is " & item
    }
}

// Do names A-M followed by names N-Z
"Everyone from A-M<br>"
foreach (name in allNames) {
    if (name < "N") {
        name & "<br>"
    }
}
"<p>Everyone from N-Z<br>"
foreach (name in allNames) {
    if (name >= "N") {
        name & "<br>"
    }
}
```

4.2.5 "isa" Operator

The "`isa`" operator is a special operator that works both like Java's `instanceof` operator and Java's cast expressions. The left side of `isa` must be a simple variable expression. When used with an `if` statement, the tested variable is automatically cast to the given type, within the scope of the `if` statement's `then` or `else` blocks.

Examples:

```
if (content isa app.pkg.News) {
    "Story is: " content.story
}

if (item isa app.pkg.Feature and content isa app.pkg.News) {
    "Info is: " item.info
    "Story is: " content.story
}
else if (item isa Object[]) {
    foreach (element in item) {
        element "<br>"
    }
}
```

4.2.6 Logical Operators

Logical operations are often used in conjunction with relational tests because they only operate on booleans. Like Java, logical operations will short circuit. Tea's logical operators are "`or`", "`and`" and "`not`". This differs from Java in that the symbols `||`, `&&`, and `!` are used to denote these operations.

Examples:

```
notitle = (title == null or title.sub == null)

if (section == "Sports" and sport == "MLB") {
    "Major League Baseball"
}
```

```
value = not (x and y)
```

4.2.7 Array Creation

Tea templates can define read-only, pre-initialized arrays. Arrays created by this expression can be indexed (zero-based) or associative (a Map). The array values can have any data type, and they can be composed of either constants or any expression. Associative arrays can have any type used for their keys, but usually strings or numbers will be used.

Based on the type of elements defined in the array, a common type is inferred. For any object retrieved from the array, the only properties that are accessible are those defined for the common type.

Arrays defined in templates can be used in the same way as an array passed to a template. This means that elements are accessed using array property syntax (i.e. x[i]). In the case of indexed arrays, they have a length property and can be used in a `foreach` statement.

The array creation expression is prefixed with a single hash (#) to denote an indexed array, and a double hash (##) to denote an associative array. The list of elements is comma-delimited and is contained inside parentheses. Associative arrays must have an even number of elements in the list, and list pattern is key, value, key, value, etc.

NewArrayExpression:

`#` `(` $List_{opt}$ `)`

`##` `(` $List_{opt}$ `)`

Examples:

```
// An indexed array containing some letters
vowels = #("a", "e", "i", "o", "u")
// Access the 3rd vowel
letter = vowels[2]

// Loop through some words
foreach (word in #("The", "quick", "brown", "fox", "jumped")) {
    "Word: " & word & "<br>"
}

// Iterate over some special values
foreach (value in #(2,3,5,7,11,13,17,19,23,29)) {
    "The value is " & cardinal(value) & "<br>"
}

// Map abbreviations to descriptions
map = ##("MLB", "Major League Baseball",
         "NBA", "National Basketball Association",
         "NHL", "National Hockey League",
         "NFL", "National Football League")
"<title>" & map[sport.abbrev] & "</title>"
```

New indexed arrays are always implemented as ordinary Java arrays, and can be passed to functions and other templates in the usual fashion. Associative arrays will implement the Map interface, and in the current implementation, HashMap is used. When associative arrays are passed to and from other templates, the element type information is preserved.

5. FUNCTIONS

5.1 Calling Functions and Templates

The call statement is used to invoke other templates or to call built-in Java functions. Both templates and Java functions may return a value. Ones that don't, (they return void) are invoked with a call statement. Otherwise, it is invoked with a call expression. The syntax of the two types of calls are identical, and are only distinguished by the presence or absence of a return value. A template returns a value using its last expression statement (see 4.1.7 Expression Statement).

CallStatement / CallExpression:

Name `(` $List_{opt}$ `)` $Block_{opt}$

`call` *Name* `(` $List_{opt}$ `)` $Block_{opt}$

The optional block part of the call statement is used to pass a block to another template or to a built in Java function.

A template can be called by its short name or its full name. A template's short name is the same as defined in its declaration. Its full name is based on the root directory from which it is stored. Essentially, templates can be packaged, but templates don't have package declarations. For example, the full name of a template declared as "header" in the "common" directory is named "common.header". When a template invokes another template within the same package, the full name is not required.

Functions are defined in a template's runtime context class. The template has no control over what context it receives. It is the responsibility of the hosting system to provide one. Any function that contains '$' characters in the name can be invoked in Tea by substituting '.' characters.

A set of standard functions are defined in the Java interface `com.go.tea.runtime.Context` and its sub-interface, `com.go.tea.runtime.UtilityContext`. Functions that apply to locale and formatting are "sticky" and remain active until the top-most template returns or the setting is changed again. In addition, a called template can change these settings, possibly affecting the caller. The string manipulation functions do not alter the first string argument, but instead return (possibly) a new string with changes.

Examples:

```
// Convert to lowercase
message = toLowerCase(message)

// Invoke a template named "header" in the same package as this one
call header("Header text")

// Invoke a template using its full name
call common.header("Header text")

// Invoke a template and pass a substitution block
name = "John"
call bigtext() {
       if (name == "Bob") {
              // Bob is welcome...
              "Hello Bob!"
       }
       else {
              // ...others are not
              "Go away, " & name & "."
       }
}
```

The header and bigtext templates are defined as follows:

```
<% template header(String title) %>
<html>
<head><title><% title %></title></head>
<body>

<% template bigtext() {...} %>
<font size="50"><% ...  %></font>
```

5.2 Function List

setLocale

```
setLocale(Locale locale)
setLocale(String language, String country)
setLocale(String language, String country, String variant)
```

Setting the locale resets date and number formats to the default for that locale. Setting a locale of null resets date and number formats to the system defaults. Since a template cannot create a specific Locale object itself, the first form is most useful if the hosting system provides a Locale object via a function or bean property.

getLocale

```
java.util.Locale getLocale()
```

Returns the current locale setting. Null if none set.

getAvailableLocales

```
Locale[] getAvailableLocales()
```

This function returns, in Locale objects, a list of all the available locales in the system. This is useful for writing a template that displays the capabilities of its hosting system.

nullFormat

```
nullFormat(String format)
```

By default, null references are printed with the string “null”. This function allows that setting to be overridden. Passing null sets the format back to “null”.

getNullFormat

```
String getNullFormat()
```

Returns the current null format specification. Null if none set.

dateFormat

```
dateFormat(String format)
dateFormat(String format, String timeZoneID)
```

Defines a format to use when printing dates from templates. Passing null sets the format back to the default. The format string is of the form “MM d yyyy”. Date formatting is provided by the Java class `java.text.SimpleDateFormat`. Refer to its documentation for more information.

getDateFormat

```
String getDateFormat()
```

Returns the current date format specification.

getAvailableTimeZones

```
TimeZone[] getAvailableTimeZones()
```

Returns, in TimeZone objects, a list of all the available time zones in the system. This is useful for writing a template that displays the capabilities of its hosting system.

getDateFormatTimeZone

```
String getDateFormatTimeZone()
```

Returns the current date format time zone.

numberFormat

```
numberFormat(String)
```

Defines a format to use when printing numbers from templates. Passing null sets the format back to the default. The format string is of the form “#.#”. Number formatting is provided by the Java class `java.text.DecimalFormat`. Refer to its documentation for more information.

getNumberFormat

```
String getNumberFormat()
```

Returns the current number format specification. Null if none set.

getNumberFormatInfinity

```
String getNumberFormatInfinity()
```

Returns the current number format for infinity. Null if none set.

getNumberFormatNaN

```
String getNumberFormatNaN()
```

Returns the current number format for NaN (Not-A-Number). Null if none set.

currentDate

```
Date currentDate()
```

Returns a Date object with the current date and time of when this function is called. Subsequent calls within the same template may return a different date.

startsWith

```
boolean startsWith(String str, String prefix)
```

Tests if the given string starts with the given prefix.

endsWith

```
boolean endsWith(String str, String suffix)
```

Tests if the given string ends with the given suffix.

find

```
int[] find(String str, String search)
int[] find(String str, String search, int fromIndex)
```

Finds the indices (in an array) for each occurrence of the given search string in the source string, optionally starting from the given index.

findFirst

```
int findFirst(String str, String search)
int findFirst(String str, String search, int fromIndex)
```

Finds the index of the first occurrence of the given search string in the source string, optionally starting from the given index, or -1 if not found.

findLast

```
int findLast(String str, String search)
int findLast(String str, String search, int fromIndex)
```

Finds the index of the last occurrence of the given search string in the source string, optionally starting from the given index, or -1 if not found.

substring

```
String substring(String str, int start)
String substring(String str, int start, int end)
```

Returns a substring using the substring method defined for the Java String class. Start and end position is zero-based, and the end index is exclusive.

toLowerCase

```
String toLowerCase(String str)
```

Returns a new string of all lowercase letters from the contents of the string passed in.

toUpperCase

```
String toUpperCase(String str)
```

Returns a new string of all uppercase letters from the contents of the string passed in.

trim

```
String trim(String str)
```

Trims all leading and trailing whitespace characters from the given string.

trimLeading

```
String trimLeading(String str)
```

Trims all leading whitespace characters from the given string.

trimTrailing

```
String trimTrailing(String str)
```

Trims all trailing whitespace characters from the given string.

replace

```
String replace(String source, String pattern, String replacement)
String replace(String source, String pattern, String replacement,
               int fromIndex)
String replace(String source, Map patternReplacements)
```

Replaces all exact matches of the given pattern in the source string with the provided replacement, optionally starting from the given index. Another form applies string replacements using the pattern-replacement pairs provided by the given map (associative array). The longest matching pattern is used for selecting an appropriate replacement.

replaceFirst

```
String replaceFirst(String source, String pattern, String replacement)
String replaceFirst(String source, String pattern, String replacement,
                    int fromIndex)
```

Replaces the first exact match of the given pattern in the source string with the provided replacement, optionally starting from the given index.

replaceLast

```
String replaceLast(String source, String pattern, String replacement)
String replaceLast(String source, String pattern, String replacement,
                   int fromIndex)
```

Replaces the last exact match of the given pattern in the source string with the provided replacement, optionally starting from the given index.

shortOrdinal

```
String shortOrdinal(Number n)
```

Returns a number’s abbreviated ordinal text i.e. 1st, 2nd, 3rd etc. Tea currently does not support localized ordinal numbers.

ordinal

```
String ordinal(Number n)
```

Returns a number’s ordinal text i.e. first, second, third etc. Tea currently does not support localized ordinal numbers.

cardinal

```
String cardinal(Number n)
```

Returns a number’s cardinal text i.e. one, two, three etc. Tea currently does not support localized cardinal numbers.

6. TYPE CONVERSION

Although variables in Tea are implicitly typed, Tea is still strongly typed. Because no direct cast operation or conversions are supported, type conversion operations are performed automatically.

Conversions that are automatically applied are usually guaranteed to always succeed. For example, an automatic conversion from Integer to String is preferred over String to Integer because Strings can represent the value of any Integer, but the opposite is not always true. Other kinds of conversions are applied to convert primitive data types to objects and vice versa. When performing addition on an int and an Integer, the Integer is converted to an int in order for the addition to be applied. This conversion could fail, however, if the Integer type being converted references null. In this case, a NullPointerException is thrown.

If the compiler cannot perform a conversion for any reason, the type checker generates an error and the template doesn't compile. This is generally caused by a true error on the part of the template writer, for example trying to use a boolean where an int should go.

6.1 Method Binding

Since Java methods can be overloaded to accept parameters of different types, the Tea compiler needs to decide on which method to bind to based on which types fit best. In Java, a cast operation can be applied which forces the binding if there is an ambiguity. Since Tea has no cast operation, the compiler binds to a method by calculating a conversion cost to all the parameters.

A conversion from an int to an Integer is deemed to cost higher than converting an int to a long. If a method is overloaded to accept an Integer or a long as a parameter and the template has an int, the method that takes the long is preferred and is called instead.

6.2 Primitive Types

Tea treats primitive types as though they were objects, except it will use primitive values wherever possible for performance. If a template passes an int to a method that accepts an Integer, a Number or just any Object, the int is converted to an Integer. If the template passes a Number to a method that accepts an int, the intValue method is called to get at the int value.

Conversion can also be applied between primitives of differing precision. If given a choice, the compiler attempts to apply the conversion with the least loss of precision. A conversion of an int to a double is preferred over the reverse, for example. Also, if given a choice, the compiler will choose to apply a conversion that does not create a new object.

Tea only performs conversions between primitive data types that represent numbers. For example, a number cannot be used where a boolean is accepted. Even though Java treats char as an unsigned 16-bit number, Tea treats it as a character, and it often gets converted to a String.

A numerical conversion from an integer to a double or float can always be forced by adding 0.0 or 0.0f. This is useful for displaying the result of a division without losing the fractional part.

6.3 Conversion to String

Any object and primitive type in Java can be converted to a string. If given the choice, however, the Tea compiler avoids doing this as it is considered a last resort. Binding an int to an Object parameter is preferred over a String parameter because it will convert it to an Integer, which is a more precise representation.

String conversion may cause a formatting operation to be applied. If the object being converted is null, the current null format is applied (see 5.2 Function List). If a date or a number, the current date or number format is applied.

The string concatenation operation, '&', always forces its operands to be converted to strings, and the resulting expression is (of course) a string. Concatenating with "" is a convenient way of forcing an expression to be converted to a string.

6.4 Variable Promotion

A variable assignment in the scope of a block can be promoted to appear after that scope. This applies to `if` statements, `foreach` statements and calls that accept blocks. Consider the following code:

```
w = "Monday" // w is a String
if (value == "something") {
    x = "Hello" // x is a String
    y = 76.6    // y is a double
    z = "ZZZ"   // z is a String
}
else {
    w = currentDate() // w is a Date
    x = 56            // x is an int
    y = 89            // y is an int
}
```

After the `if` statement, the following variables will be available for use: w, x and y with the respective types: Object, Object and double. The z variable is unavailable since it was only assigned in the then part of the `if` statement, and therefore there is no guarantee that will contain a legal value.

When promoting variables from an inner scope, the Tea compiler chooses a compatible type that represents the merged type as specifically as possible. For objects, it does this by analyzing the inheritance graph. The common type produced for java.util.Vector and java.util.Set is java.util.Collection.

7. Grammar Summary

Tea's grammar is designed to resemble Java's, but because semi-colons are optional, some elements have unique appearances in order for the grammar to be more unambiguous. Most notably is the format to define new arrays. Without the leading hash symbols, a new array list can appear to be a parameter list or a simple expression. Unlike Java, new array values are specified within a list delimited by parentheses instead of braces. Use of braces seems inconsistent because they should be used to group statements, not expressions.

Call statements can optionally begin with a "`call`" keyword to indicate a template call over a function call. Because the function calling form does not begin with this keyword, Tea parsers may require special logic in order to properly parse it.

Template:

> *TemplateDeclaration StatementList*$_{opt}$

StatementList:

> *Statement*
>
> *StatementList Statement*

TemplateDeclaration:

> `template` *Identifier* `(` *FormalParameterList*$_{opt}$ `)` *SubstitutionParameter*$_{opt}$

FormalParameterList

> *FormalParameter*
>
> *FormalParameterList* `,` *FormalParameter*

FormalParameter

> *Type Variable*

Type:

> *Name*
>
> *Type* `[ ]`

Name:

Identifier

Name `.` *Identifier*

Variable:

Identifier

SubstitutionParameter:

`{ ... }`

Statement:

EmptyStatement

IfStatement

ForeachStatement

SubstitutionStatement

AssignmentStatement

CallStatement

ExpressionStatement

EmptyStatement:

`;`

IfStatement:

`if (` *Expression* `)` *Block* *ElseStatement*$_{opt}$

ElseStatement:

`else` *Block*

`else` *ifStatement*

Block:

`{` *StatementList*$_{opt}$ `}`

ForeachStatement:

> `foreach (` *Variable* `in` *Expression* `)` *Block*
>
> `foreach (` *Variable* `in` *Expression* `reverse )` *Block*
>
> `foreach (` *Variable* `in` *Expression* `..` *Expression* `)` *Block*
>
> `foreach (` *Variable* `in` *Expression* `..` *Expression* `reverse )` *Block*

SubstitutionStatement:

> `...`

AssignmentStatement:

> *Variable* `=` *Expression*

CallStatement:

> *Name* `(` $List_{opt}$ `)` $Block_{opt}$
>
> `call` *Name* `(` $List_{opt}$ `)` $Block_{opt}$

List:

> *Expression*
>
> *List* `,` *Expression*

ExpressionStatement:

> *Expression*

Expression:

> *OrExpression*

OrExpression:

> *AndExpression*
>
> *OrExpression* `or` *AndExpression*

AndExpression:

 EqualityExpression

 AndExpression `and` *EqualityExpression*

EqualityExpression:

 RelationalExpression

 EqualityExpression `==` *RelationalExpression*

 EqualityExpression `!=` *RelationalExpression*

RelationalExpression:

 ConcatenateExpression

 RelationalExpression `<` *ConcatenateExpression*

 RelationalExpression `>` *ConcatenateExpression*

 RelationalExpression `<=` *ConcatenateExpression*

 RelationalExpression `>=` *ConcatenateExpression*

 RelationalExpression `isa` *Type*

ConcatenateExpression:

 AdditiveExpression

 ConcatenateExpression `&` *AdditiveExpression*

AdditiveExpression:

 MultiplicativeExpression

 AdditiveExpression `+` *MultiplicativeExpression*

 AdditiveExpression `-` *MultiplicativeExpression*

MultiplicativeExpression:

 UnaryExpression

 MultiplicativeExpression `*` *UnaryExpression*

 MultiplicativeExpression `/` *UnaryExpression*

 MultiplicativeExpression `%` *UnaryExpression*

UnaryExpression:

`not` *UnaryExpression*

`-` *UnaryExpression*

LookupExpression

LookupExpression:

Factor $\textit{Lookup}_{opt}$

Lookup:

$\textit{Lookup}_{opt}$ `.` *Identifier*

$\textit{Lookup}_{opt}$ `[` *Expression* `]`

Factor:

NewArrayExpression

`(` *Expression* `)`

Literal

CallExpression

Variable

NewArrayExpression:

`#` `(` $\textit{List}_{opt}$ `)`

`##` `(` $\textit{List}_{opt}$ `)`

CallExpression:

Name `(` $\textit{List}_{opt}$ `)` $\textit{Block}_{opt}$

`call` *Name* `(` $\textit{List}_{opt}$ `)` $\textit{Block}_{opt}$

7.1 LL Grammar Reference

The grammar provided in the summary and everywhere else in the document is suitable for LR parsers, but a grammar suitable for LL parsers is useful as well. The current implementation of Tea uses a manually written LL(n) parser.

The notation used for the LL grammar follows an EBNF convention. Brackets (“[” and “]”) denote optional derivations. Braces (“{” and “}”) denote derivations that may expand zero or more times. Key terminals are shown with a fixed width font, and key terminals that are punctuation characters are shown inside double quotes.

Template	::=	*TemplateDeclaration StatementList*
StatementList	::=	{ *Statement* }
TemplateDeclaration	::=	`template` *Identifier* “`(`” [*FormalParameterList*] “`)`”
		[*SubstitutionParameter*]
FormalParameterList	::=	*FormalParameter* { “`,`” *FormalParameter* }
FormalParameter	::=	*Type Variable*
Type	::=	*Name* { “`[`” “`]`”}
Name	::=	*Identifier* { “`.`” *Identifier* }
Variable	::=	*Identifier*
SubstitutionParameter	::=	“`{`” “`...`” “`}`”
Statement	::=	*EmptyStatement*
	|	*IfStatement*
	|	*ForeachStatement*
	|	*SubstitutionStatement*
	|	*AssignmentStatement*
	|	*CallStatement*
	|	*ExpressionStatement*
EmptyStatement	::=	“`;`”
IfStatement	::=	`if` “`(`” *Expression* “`)`” *Block* [*ElseStatement*]
ElseStatement	::=	`else` *Block*
	|	`else` *IfStatement*
Block	::=	“`{`” *StatementList* “`}`”
ForeachStatement	::=	`foreach` “`(`” *Variable* `in` *Expression* [`..` *Expression*]
		[`reverse`] “`)`” *Block*

SubstitutionStatement	::=	“`...`”
AssignmentStatement	::=	*Variable* “`=`” *Expression*
CallStatement	::=	[`call`] *Name* “`(`” [*List*] “`)`” [*Block*]
List	::=	*Expression* { “`,`” *Expression* }
ExpressionStatement	::=	*Expression*
Expression	::=	*OrExpression*
OrExpression	::=	*AndExpression* { `or` *AndExpression* }
AndExpression	::=	*EqualityExpression* { `and` *EqualityExpression* }
EqualityExpression	::=	*RelationalExpression* { “`==`” *RelationalExpression* }
	|	*RelationalExpression* { “`!=`” *RelationalExpression* }
RelationalExpression	::=	*ConcatenateExpression* { “`<`” *ConcatenateExpression* }
	|	*ConcatenateExpression* { “`>`” *ConcatenateExpression* }
	|	*ConcatenateExpression* { “`<=`” *ConcatenateExpression* }
	|	*ConcatenateExpression* { “`>=`” *ConcatenateExpression* }
	|	*ConcatenateExpression* { “`isa`” *Type* }
ConcatenateExpression	::=	*AdditiveExpression* { “`&`” *AdditiveExpression* }
AdditiveExpression	::=	*MultiplicativeExpression* { “`+`” *MultiplicativeExpression* }
	|	*MultiplicativeExpression* { “`-`” *MultiplicativeExpression* }
MultiplicativeExpression	::=	*UnaryExpression* { “`*`” *UnaryExpression* }
	|	*UnaryExpression* { “`/`” *UnaryExpression* }
	|	*UnaryExpression* { “`%`” *UnaryExpression* }
UnaryExpression	::=	`not` *UnaryExpression*
	|	“`-`” *UnaryExpression*
	|	*LookupExpression*
LookupExpression	::=	*Factor* { *Lookup* }
Lookup	::=	“`.`” *Identifier*
	|	“`[`” *Expression* “`]`”
Factor	::=	*NewArrayExpression*
	|	“`(`” *Expression* “`)`”
	|	*Literal*
	|	*CallExpression*
	|	*Variable*

NewArrayExpression	::=	“`#`” “`(`” [*List*] “`)`”
	|	“`##`” “`(`” [*List*] “`)`”
CallExpression	::=	[`call`] *Name* “`(`” [*List*] “`)`” [*Block*]