NextGen Developer's Guide

Eric Allen
eallen@cs.rice.edu
Rice University
6100 S. Main St.
Houston TX 77005

February 16, 2003

1 Introduction

This document describes the high-level architecture and code layout of the NextGen prototype compiler, developed at Rice University. The NextGen compiler was developed as an extension to the GJ compiler under special license from Sun Microsystems. This same compiler was extended independently by Sun Microsystems to form the JSR-14 prototype compiler, scheduled for inclusion in J2SE 1.5. In the process of developing NextGen, we have refactored the original GJ compiler substantially, and no attempt has been made to maintain compatibility with the JSR-14 source code. Nevertheless, the reader may find that some of the architectural features of NextGen described here are helpful when deciphering the JSR-14 code base (modulo class, package, and variable name changes).

The GJ compiler, as well as the NextGen and JSR-14 compilers derived from it, are written in Generic Java.[1] As a result, we enjoy much more precise type checking in the source code of these compilers than we would have with ordinary Java. But we are also unable to use many of the powerful development tools available for standard Java.[2]

[1]"Generic Java" is a specification for a family of languages that add generic types to Java. This family includes both GJ and NextGen.

[2]Two very useful tools that are compatible with Generic Java are the CodeGuide and DrJava IDEs. All NextGen developers are encouraged to leverage both tools. DrJava is particularly useful for writing new unit tests; CodeGuide provides support for automated refactoring and incremental compilation.

Throughout this document, it is assumed that the reader is familiar with the GJ, NextGen, and MixGen language designs, as well as the published descriptions of how these languages can be compiled to the JVM so as to maintain compatibility with existing compiled binaries. Readers not already familiar with this material are referred to [10, 9, 3]. The References section also includes optional background material discussing various extensions of the Java type system, including first-class genericity and non-generic mixins.

It is also assumed that the reader has set up a JavaPLT development environment, is familiar with basic CVS commands, with JUnit, with Ant, and with DrJava. Instructions on setting up a JavaPLT environment are available at

```
http://www.cs.rice.edu/~javaplt/doc/developer.pdf
```

A short tutorial on CVS is available at `http://www.cvshome.org/docs/manual`. A tutorial on JUnit is available at `http://www.junit.org`. A tutorial on Ant is available at `http://www.jakarta.org`. DrJava documentation is available at `http://drjava.sf.net`.

2 The NextGen CVS Repository

The NextGen source code is maintained under the `javaplt` CVS repository. To checkout the NextGen source code, go to your `javaplt` directory and checkout the module `nextgen`. Because NextGen is written in JSR-14, you will need to prepend JSR-14 to the bootclasspath when starting Ant. The best way to do that is to set your `ANT_OPTS` environment variable to the location of the JSR-14 jar. For example, on Unix systems, set it to:

```
-Xbootclasspath/p:`javaplt-home`/packages/jsr14_adding_generics-1_3-ea/javac.jar
```

Of course, on Windows/Cygwin, your path will look different.[3]

Once you have checked out a copy of the code, the first thing you should do is `cd` to the new `nextgen` directory and type

```
$ ant all
```

Doing so will run all Ant targets in the project. An *essential* invariant of the NextGen project is that `ant all` should succeed when invoked on a clean checkout, on all supported platforms (Linux, Solaris, OS X, and Windows XP/Cygwin) with a proper JavaPLT development environment. If it doesn't, be sure to correct any problems with your environment before continuing.

[3]Because we must set this variable before Ant starts, this is one place where Ant's ability to automatically convert path names doesn't save us.

3 The NextGen Project Directory Structure

The `nextgen` project directory contains the following files and subdirectories:

- `build.xml`. This file contains the XML source code for all Ant targets associated with the NextGen project.

- `anttools`. This directory contains the Java source code for custom Ant tasks associated with NextGen. In particular, the `NextGenCompilerTask` and `NextGenJUnitRunner` source code is contained here. Because all Java source code in the NextGen project is placed in various subpackages of `edu.rice.cs.nextgen`, the files in this directory (and all other directories containing Java source code) are all placed in subdirectories of `edu/rice/cs/nextgen`.

  `NextGenCompilerTask` is used in the `ant compile` target in `build.xml`. Although the NextGen compiler could be invoked within Ant simply as a Java program, the `NextGenCompilerTask` is much more convenient because it allows the compiler to be invoked on Ant filesets and other compound structures instead of just an explicit list of command-line arguments.

  `NextGenJUnitRunner` is used for NextGen acceptance testing. It is called by the `simpletests` target. This task uses the NextGen classloader to invoke JUnit on `TestCases` compiled with the `NextGenCompiler`. Note that such `TestCases` can't be invoked by JUnit with the default class loader because some of them may be template classfiles.

- `benchmarks`. Contains the source code for the NextGen benchmark suite. There are three versions of this suite: `edu.rice.cs.nextgen.javabenchmarks`, `edu.rice.cs.nextgen.gjbenchmarks`, `edu.rice.cs.nextgen.nextgenbenchmarks`, corresponding to the three compilers tested in NextGen benchmarking. All of these suites are compiled by the Ant target `compile-benchmarks`. A bash script is provided for running the benchmarks in the `bin` subdirectory, discussed below. Note that the benchmarks can't be run (easily) from within Ant because an entire run of the benchmarks involves invoking several different JVMs with different classloaders. Also, to prevent skewing performance results, we don't want to use up resources by running an Ant process during benchmarking.

- `bin`. Contains various bash scripts useful for performing tasks associated with the NextGen project. Ultimately, we'd like to turn all of these scripts into platform-independent Ant targets, but in same cases, we haven't yet found a good way to do that.

- `doc`. Contains all documentation associated with the NextGen project, including the sources for this document. The NextGen documentation available online is a checked out copy of this directory. To make changes to the live webpage, you must have write access to `/home/javaplt`. After committing your changes, go to `/home/javaplt/public_html` and execute `cvs update`. That command will update both the `public_html` module and the copy of `nextgen/doc`. To update only the NextGen documentation, simply perform a `cvs update` in the `/home/javaplt/public_html/nextgen` directory.

- `lib`. Contains all library classes needed by the Ant targets, including JSR-14, JUnit, and all `bootclasses` for Linux, Solaris, OS X, and Windows XP. Note that the bootclasses are included inside this project subdirectory to maintain the invariant that `ant all` always succeeds on a clean check-out. All bootclasses must be accessible not just to Ant, but also to the NextGen compiler, to prevent it from signaling errors when compiling sources with references to standard library classes. There is no standardization of the placement of bootclassses across JDKs[4] and we want to decouple any quirks of the bootclasspath on a platform from the ability of the Ant targets to work. By putting all bootclasses in a well-defined location, we can ensure that we can always find them.

- `manifests`. Contains the manifest files used when bundling the NextGen compiler and classloader into jar files.

- `jars`. Contains the jar files for the compiler and classloader, constructed with target `ant jar`.

- `simpletests`. Contains all acceptance tests for the NextGen compiler, in the form of JUnit `TestCases` written in NextGen. Running an acceptance test consists of compiling it under the NextGen compiler and then invoking JUnit on it with the NextGen classloader. By running JUnit on the compiled code, we check that the semantics of that code is as expected, providing a powerful check on NextGen's correctness. The acceptance tests can be run by invoking the Ant target `simpletests`.

[4]Non-Sun JDKs such as the Mac OS X JDK do not always follow the file layout of the Sun JDKs, and Sun hasn't made JDK file layout part of any standard specification.

- `src`. Contains all source code for the NextGen compiler and classloader, in packages `edu.rice.cs.nextgen.compiler` and `edu.rice.cs.nextgen.classloader`, respectively. In the section on NextGen package design, we will discuss the various subpackages in this directory.

- `built`. This directory is empty on a clean checkout. It's used by Ant to store the class files for all Generic Java sources associated with NextGen.

4 Ant Targets in the NextGen Project

For a complete list of targets, always refer to the `build.xml` file. Below are some of the most common targets you will invoke.

- `clean`. Deletes all compiled classfiles.
- `compile`. Compiles the compiler and classloader.
- `test`. Runs all unit tests on the compiler and classloader.
- `simpletests`. Compiles the compiler and classloader and runs all acceptance tests.
- `testall`. Compiles the compiler and classloader and runs all unit tests and acceptance tests.
- `compile-tests`. Compiles all acceptance tests with the most recently compiled version of NextGen.
- `update`. Updates this checked out copy of NextGen with the CVS repository.
- `commit`. Synchronizes with the CVS repository, compiles, ensures that all unit tests and acceptance tests still pass, and, if so, creates new jar files and commits the newly synchronized version to the repository. Run this often! At least once a day, if not once an hour when actively working with the code. The more often you run it, the easier it will be to diagnose new bugs.
- `jar`. Constructs jar files for the most recently compiled versions of the compiler and classloader.
- `all`. Runs all targets in the project. Useful for making sure that everything still works.

- `alloffline`. Runs all targets in the project except commit. Useful for making sure that everything testable works when you're not connected to a network.

5 Releasing a New Version of NextGen

The NextGen documentation and jar files are available online at `http://www.cs.rice.edu/ javaplt/nextgen`. To release the latest committed version to the live website, simply go to `/home/javaplt/nextgen` on csnet and do a `cvs update`. Concerning the documentation: be sure that the latest committed LaTeX file corresponds to the latest committed pdf file.[5]

WARNING: If you edit anything in the live directories, be sure to perform a `cvs commit` so your changes are included in the CVS repository. Better yet, never edit the live directories directly. Instead, always make changes to your local copy and update the live directory.

6 NextGen Package Design

6.1 Classloader Package Design

All classes in the NextGen classloader are contained in a single package: `edu.rice.cs.nextgen.classloader`. There are no subpackages. The main entry point is class `Runner`.

6.2 Compiler Package Design

The source code for the NextGen compiler is divided into the following packages, found in `src/edu/rice/cs/nextgen/compiler`:

- `main`. Contains the main entry point for the compiler, in class `Main`, as well as supporting code. In particular, class `JavaCompiler` contains the code for invoking the various phases of the compiler.
- `parser`. Contains the code for scanning and parsing NextGen source files.
- `tree`. Contains the code defining NextGen abstract syntax trees.

[5]In the long run, the pdf file should be re-generated during `ant commit`. Doing so would require finding a portable LaTeX to pdf generator, putting it in the NextGen lib directory, and calling it in the `commit` target.

- `comp`. Contains the code for computing most phases of compilation, including symbol table entry, type checking, data flow checking, and bytecode generation.

- `code`. Contains code for many of the datastructures utilized by the various phases of compilation. Unfortunately, there is no sharp conceptual distinction between the classes contained in this package and the classes contained in the `code` package. A useful refactoring would be to move the classes in these two packages so as to define the roles of each package more precisely.

- `flatten`. Contains the visitors and support code that convert type dependent operations into snippet calls. No comparable package exists in the JSR-14 compiler. Parametric types are flattened according to the rules for NextGen name mangling. Snippet methods are added to classes as necessary. Template classes are generated. Finally, the jump targets for `break` and `continue` statements are fixed after flattening.[6]

- `instrument`. Contains the code for pretty-printing all datastructures used by the compiler. No comparable package exists in the JSR-14 compiler. We originally added this code to facilitate diagnosis of errors when modifying the datastructures during NextGen-specific phases of compilation. It has been an invaluable tool in retrofitting unit tests over `NextGen`.

- `util`. Contains many general-purpose datastructures, including parametric versions of many Java collections classes. These collections classes were written as part of the GJ compiler before the JSR-14 compiler was available. We've modified and rewritten many of them, particularly the `List` classes. Although JSR-14 provides parametric versions of the collections classes, our versions (particularly of `Lists`) have many advantages over those in JSR-14, so in most cases there is little (or negative) incentive to refactor them out of the codebase. One useful feature that is currently missing from our hash tables is an iterator (however, a `map` facility is provided).

- `hist`. Contains some useful DrJava interactions histories. `ImportAll.hist` imports all `NextGen` source packages. `NextGenTestCase` sets up all variables initialized in class `edu.rice.cs.nextgen.compiler.main.NextGenTestCase`.

[6]These targets are broken by flattening because the positions of the targets are changed when snippet methods are added.

`JavaCompilerTest.hist` performs the various tasks done in the `imports` and `setUp` method of `JavaCompilerTest`, i.e., it initializes a new `JavaCompiler` on a `NextGen` sourcefile:

```
> package edu.rice.cs.nextgen.compiler.main;
> import edu.rice.cs.nextgen.compiler.code.*;
> import edu.rice.cs.nextgen.compiler.comp.*;
> import edu.rice.cs.nextgen.compiler.instrument.*;
> import edu.rice.cs.nextgen.compiler.util.*;
> import edu.rice.cs.nextgen.compiler.tree.*;
> import java.io.IOException;
> import junit.framework.TestCase;
> import junit.framework.TestSuite;
>
> String[] args = new String[] {
    "-classpath",System.getProperty("sun.boot.class.path"),
    "simpletests/edu/rice/cs/nextgen/simpletests/InstanceofParameter.java"
  };
> CompilerOptions options = new CompilerOptions();
> _fileNames = options.processArgs(args);
> _compiler = new JavaCompiler(options);
```

Be sure to add other useful interactions histories as you build them.

7 Unit Tests in NextGen

In both the compiler code and the classloader code, unit tests are contained in files ending in `Test`. Each such file contains a public class that extends `junit.framework.TestCase`. Although there is a significant set of working unit tests over the source code, not all code is covered directly (yet). Nevertheless, skeleton test cases are provided for all code, making it easier to add new tests. Also, the `PrintableObject` class in package `instrument`, and the `NextGenTestCase` class in package `main` make writing new unit tests easy over any part of the code base.

7.1 Class `PrintableObject`

`PrintableObject` provides a mechanism for pretty-printing complex datastructures. By extending `PrintableObject`, a class inherits all the functionality needed to pretty-print itself. All that is needed is for the extending class to override method `print()` and use the inherited methods to specify how it should be printed. A `PrintableObject` decides which fields of itself to print. It can also tell its constituent fields that are `PrintableObjects` to print themselves selectively, passing a `Filter` object to handle cyclic references.[7] Virtually all complex datastructures in NextGen extend `PrintableObject`. See their `print` methods for examples of how to call the `PrintableObject` functionality for pretty-printing.

In addition to method `print`, which sends a pretty-printed representation of an object to `System.out`, every `PrintableObject` has a `longString()` method that takes no arguments and returns a (multiline) `String` representation of the object. `longString()` is extremely useful when writing unit tests. Reasonable `toString` methods are also written for most classes, but they contain much less detailed information. The `toString` methods can be useful for quickly checking the identity of an object.

To familiarize yourself with the various datastructures used in NextGen, it is recommended that you experiment with printing out instances of them in the DrJava interactions window while reading through their descriptions in this document. Another great way to learn how these datastructures work is by writing new unit tests over them.

7.2 Class `NextGenTestCase`

`NextGenTestCase` initializes default values for the most commonly used NextGen datastructures, allowing you to quickly build complex datastructures in a test case. It also defines the following convenience methods:

- `String removeWhiteSpace(String)`. This method takes a `String` and returns a new `String` with all whitespace removed. It's useful for checking that a multiline `String` representation of an `Object` is as expected (without breaking everytime that the whitespace in the representation is changed).
- `String removeNewlines(String)`. Like `removeWhiteSpace`, with similar motivation, but only `newlines` are removed.
- `String wrapInQuotes(String)`. Often in a test method, you will want to check that the `String` representation of an object is as expected. To specify

[7] See classes `edu.rice.cs.nextgen.compiler.code.Scope` and `edu.rice.cs.nextgen.compiler.code.Symbol.ClassSymbol` for two examples of cyclic structures printed with `Filters`.

what's expected in source code, you have to wrap the `String` representation inside quotes. For multiline `String` representations, this task can be extremely tedious. `wrapInQuotes` is a static method that takes a `String` and returns a new `String` with each line wrapped in quotes. This method can be conveniently called in the DrJava interactions window while constructing new test methods. In the long run, `wrapInQuotes` should become functionality provided by DrJava on selected blocks of text in the editor. Until that functionality is added, this method will save you time.

- `Parser makeParser(String)`. Takes a `String` representing source code and returns an instance of `edu.rice.cs.nextgen.parser.Parser` to read that source code.

- `Tree.TopLevel makeClassTree(String)`. Takes a `String` representing the source code of a complete Java source file, parses it and returns an instance of `Tree.TopLevel`.[8]

- `Tree.Import makeImport(String)`. Takes a `String` representing an import declaration, parses it and returns an instance of `Tree.Import`.

Other `make` methods should be added to this class as they become useful in the unit tests. It is recommended that all new NextGen test case classes extend `NextGenTestCase`.

8 Phases of The NextGen Compiler

When class `Main` is invoked with a sequence of keyword arguments and files, it stores all keyword arguments in a new `CompilerOptions` object named `options`, creates a new instance of `JavaCompiler` named `compiler`, with a field reference to `options`, and calls `compiler.compile()` on a `List` of all source files to compile. The phases of compilation corresponding precisely to the sequence of method calls in method `JavaCompiler.compile()`. Those phases are as follows.

```
ListBox<Tree> trees = parseFiles(filenames);
List<Environment<AnalyzerContext>> envs = enterClasses(trees);
checkTypes(envs);
envs = flatten(envs);
patchSnippets();
eraseTypes(envs);
```

[8]For more information on `Trees`, seee the section"Phases of the NextGen Compiler".

```
    flattenGroundedSuperClasses();
    List<ClassSymbol> classSyms = generateByteCode(envs);
```

We will now provide a high-level overview of the entire compilation process and then delve into the datastructures manipulated during each phase.

During parsing (Stage I), the initial `List<String>` of filenames is converted into a `List<Tree>`. Next, (Stage II), the classes and packages in this tree are entered into a `SymbolTable` object, and `Symbols` for various lexical items are filled into the `Trees`. A `List<Environment<AnalyzerContext>>` (basically the original `List<Tree>` with each `Tree` wrapped in an `Environment`) is returned from this phase. The `List<Environment<AnalyzerContext>>` is named `envs` and is the primary datastructure passed through subsequent phases.

After symbols are entered, `envs` is type checked (Stage III). The type of each `Tree` is recorded in a field in the `Tree` . After type-checking comes NextGen-specific processing. Type names are flattened, snippets are added and template classes and interfaces are generated as static nested classes in the corresponding base class (Stage IV). Afterward, various source-position-specific targets must be patched in the base class (Stage V), since the locations of items will move after adding snippet methods.

After snippets are patched, types are erased, as in GJ (Stage VI).[9] Then the superclasses of grounded types are flattened (Stage VII). [10] Finally, bytecode is generated from `envs` (Stage VIII) and returned as a `List<ClassSymbol>`. These `ClassSymbols` are then written out to disk to form the corresponding classfiles.

Notice that not all stages take all datastructures they depend on as arguments. In particular, `patchSnippets` and `flattenGroundedSuperClasses` do not take `envs` as an argument. They operate on `envs` via field references set to it. Obviously, this situation is undesirable; an important refactoring is to finish decoupling all field references to `envs` and to always pass it as an argument.[11]

Every major phase of compilation after parsing[12] is implemented as a walk over the parsed `Trees`. Consequently, each phase is associated with a subclass of class `Tree.Visitor`. For example, symbol entry is handled by visitor

[9]Notice that flattened types won't be erased, as they are no longer parametric.

[10]This part of flattening must be deferred because Stage IV depends on the unflattened representation. In addition, Stage II as currently written depends on this deferral because it tests whether or not the superclass is parametric and ground to determine whether the supercall for a constructor must be modified.

[11]Originally, there were many other phases that kept field references to `envs`. The remaining phases are the most difficult to refactor.

[12]Snippet patching is not performed by a visitor because the targets to fix are accumulated during type flattening. Class `SnippetPatcher` simply walks over the accumulated list.

SymbolEnterer. Static checking is handled primarily by visitor StaticAnalyzer, which is assisted by visitor TypeChecker. Flattening of parametric type references is handled by visitor TypeFlattener. Type erasure is handled by visitor TypeEraser[13] Code generation is handled by visitor CodeGenerator. We will now examine the various datastructures manipulated by these phases. We will start with Trees because the first internal representation of the source code constructed is a List<Tree>, and all other datastructures built depend on them.

8.1 Trees

```
Tree ::= TopLevel | Import | ClassDef | MethodDef | VarDef | Block
       | DoLoop | WhileLoop | ForLoop | Labelled | Switch | Case
       | Synchronized | Try | Catch | Conditional | Exec | Break
       | Continue | Return | Throw | Apply | NewInstance | NewArray
       | Assign | Assignop | Operation | TypeCast | TypeTest
       | Indexed | Select | Ident | Literal | TypeIdent | TypeArray
       | TypeApply | TypeParameter | Erroneous
```

Figure 1: Abstract Syntax Trees

Trees in NextGen represent abstract syntax trees. Trees form a composite class hierarchy rooted at class edu.rice.cs.nextgen.compiler.tree.Tree. All subclasses in this composite hierarchy are static nested classes in class Tree. Every syntactic construct in NextGen corresponds to a subclass of class Tree. For example, source files are parsed to TopLevels and class definitions are parsed to ClassDefs. The entire composite hierarchy is represented in BNF notation in Figure 8.1.

8.2 Trees and PrintableObjects

Trees are a subclass of class edu.rice.cs.nextgen.compiler.instrument.PrintableObject, allowing them to be pretty-printed easily in the DrJava interactions pane. For example, suppose we want to pretty-print the Tree for this simple NextGen source file:

```
package example;
public class C<T> {}
```

[13]TypeEraser is a subtype of TreeTranslator, which is a subtype of Tree.Visitor.

We can print it in the DrJava interactions window as follows:

```
> package edu.rice.cs.nextgen.compiler.main;

> NextGenTestCase.makeClassTree(
     "package example; \n" +
     "public class C<T> { } \n"
   ).longString()
```

Resulting in:

```
"{TopLevel for null
  packageId: {Ident example} packageSymbol: null
  starImportScope (except for java.lang): null
  namedImportScope: null
  ==========
  {ClassDef C<{TypeParameter T extends null implements null}> extends null implements ()
    flags: 1
    null}}
"
```

This result is a pretty-printed representation of a `Tree.TopLevel`. The first line indicates the source file corresponding to the `TopLevel`. In this case it is `null` because this `TopLevel` was parsed directly from a `String` instead of a file. If the `TopLevel` were constructed from a file `f`, the first line would read `TopLevel for f`. The fields printed represent the package, the imports, and the list of `ClassDefs`. Notice the use of `null` values as defaults for many fields.

Underneath the horizontal bar are listed the constituent class definitions in the source file. The pretty-printed `ClassDef` follows the structure of the original source code. The `flags` field of a `ClassDef` is an `int` that stores various attributes of the class, such as the included visibility modifiers. Underneath the header of the `ClassDef` is its contained `ClassSymbol`, initialized to `null` and filled in during symbol entry.

If class `C` contained any members, the `Trees` for those members would be printed out underneath the `ClassSymbol`. For example, below is the source for acceptance test `ReturnIntegerTest`:

```
package edu.rice.cs.nextgen.simpletests;

class C<T> {
  Object m(Integer i) {
    return i;
  }
}
```

This source will parse to

```
{TopLevel for simpletests/edu/rice/cs/nextgen/simpletests/ReturnIntegerTest.java
  packageId: {Select {Select {Select {Select {Ident edu}.rice}.cs}.nextgen}.simpletests}
  packageSymbol: null
  starImportScope (except for java.lang): null
  namedImportScope: null
  ==========
  {ClassDef C<{TypeParameter T extends null implements null}> extends null implements ()
    flags: 0
    null
    {MethodDef <> {Ident Object} m({VarDef {Ident Integer} i}) throws  flags: 0
      null
      {Block {Return {Ident i}}}}}}
```

The headers of method definitions include the type parameters (between angle brackets), the return type, name, parameter types, `throws` clause, and the visibility flags. Beneath the header is a `MethodSymbol` (filled in during symbol entry). Finally, beneath the `MethodSymbol` is the parsed body of the method.

The generated bytecodes of a method are stored in a `MethodSymbol`. After bytecode generation, the printed `MethodSymbol` will display these bytecodes. For example, here is the same `TopLevel` as above after all phases including bytecode generation:

```
 {TopLevel for simpletests/edu/rice/cs/nextgen/simpletests/ReturnIntegerTest.java
   packageId: {Select {Select {Select {Select {Ident edu}.rice}.cs}.nextgen}.simpletests}
   packageSymbol: [PackageSymbol edu.rice.cs.nextgen.simpletests]
   starImportScope (except for java.lang): null
   namedImportScope:
     [Scope: [ClassSymbol  fullname: edu.rice.cs.nextgen.simpletests.C
        flatname: edu.rice.cs.nextgen.simpletests.C flags: 1024
        members (cyclic references have been filtered):
              [Scope: [MethodSymbol m
                                  [Code: aload_1; -80; ]]
                             [MethodSymbol <init>
```

```
                              [Code: aload_0; -73; nop; aconst_null; -79; ]]]]]
==========
{ClassDef C<{TypeParameter T extends null implements null}> extends null implements ()
  flags: 0
  [ClassSymbol  fullname: edu.rice.cs.nextgen.simpletests.C
    flatname: edu.rice.cs.nextgen.simpletests.C flags: 1024
    members (cyclic references have been filtered):
    [Scope: [MethodSymbol m
                 [Code: aload_1; -80; ]]
               [MethodSymbol <init>
                 [Code: aload_0; -73; nop; aconst_null; -79; ]]]]
   {MethodDef <> null <init>() throws  flags: 0
     [MethodSymbol <init>
       [Code: aload_0; -73; nop; aconst_null; -79; ]]
     {Block {ExpressionStatement {Apply {Ident super}()}}}}
   {MethodDef <> {Ident Object} m({VarDef {Ident Integer} i}) throws  flags: 0
     [MethodSymbol m
       [Code: aload_1; -80; ]]
     {Block {Return {Ident i}}}}}}
```

`MethodSymbols` inside `ClassSymbols` contain bytecode, as well as the the `MethodSymbols` attached to `MethodDefs` in `Trees` (which are actually the same as (`==` to) the `Symbols` contained in the `ClassSymbol`. Incidentally, notice that a default constructor has been added to class `C`.

8.3 Environments

A `List<Environment<AnalyzerContext>>` is returned from Stage II. An `Environment` is a structure associated with a `Tree`; each `Environment` contains its associated `Tree` as a field. The `List` returned from Stage II contains one environment for each `Tree` in the `List<Tree>` passed to it.

In addition to referencing its associated `Tree`, each `Environment` contains a reference to its enclosing `Environment`. For example, if an `Environment` E is associated with an immediate subtree T of a enclosing `Tree` T', and `Environment` E' is associated with T', then E contains a reference to E'. Each element of the `List<Environment<AnalyzerContext>>` resulting from the parsing phase of compilation is associated with a `TopLevel` representing a file in the original list of filenames. Contained `Environments` can be constructed from enclosing `Environments` by method `spawn(Tree)`.

Environments store information relevant to their associated `Tree`. They are used during two phases of compilation: static analysis and code generation. The information stored during these two phases is significantly different. In order to decouple the general-purpose functionality of `Environments` from the code for storing the information relevant to a particular usage, `Environments` are parameterized by a type parameter `A`. A field `context` of type `A` contains the information stored at each level of an `Environment`. So, during static analysis, we make use of a `List<Environment<AnalyzerContext>>` where each `Environment<AnalyzerContext>` in the `List` contains a `context` field of type `AnalyzerContext`.

8.4 AnalyzerContexts

`AnalyzerContexts` contain many fields relevant to static analysis, but most importantly they contain the following three fields:

1. An `expectedType` field of type `Type`. `Types` represent the results of static checking. This field is initialized to `NoType.ONLY` and filled in during static checking.

2. An `expectedKind` field of type `int`. `Kinds` are very high-level descriptions of the expected syntactic structure of a `Tree` that are filled in during static checking. The possible `Kinds` of a `Tree` are as follows: `NO_KIND`, `PACKAGE_KIND`, `TYPE_KIND`, `VAR_KIND`, `VAL_OR_VAR_KIND`, `METHOD_KIND`, `ALL_KINDS`. Each `Kind` is represented as an `int` in interface `edu.rice.cs.nextgen.compiler.code.Kinds`.[14] The `expectedKind` of an `AnalyzerContext` is initialized to `NO_KIND` and filled in by subsequent analysis.

3. A `Scope`. Conceptually, `Scopes` are lists of identifiers introduced in a lexical context. We now turn our attention to their internal structure.

8.5 Scopes and Entries

Each `Scope` is owned by a `Symbol` corresponding to the lexical environment the `Scope` represents. For example, a `Scope` corresponding to fields in a class would

[14]Representing a dataset of atomic elements as `int` fields in an interface is an common idiom in NextGen, inherited from the `GJ` compiler. This idiom is a workaround for the lack of enumeration types in Java. By "implementing" an interface corresponding to a dataset, a class can refer to all of the elements of the dataset directly. Also, a class can iterate over the elements of the dataset with a simple `int` counter in a `for` loop. One serious disadvantage of this idiom is that values of the dataset must be deciphered when they're printed out in an error message. Also, proper data abstraction is not enforced; there is nothing to prevent us from using an element of a dataset in a nonsensical arithmetic operation.

be owned by the associated `ClassSymbol`. The first identifier appearing in a `Scope` is stored in an `Entry`. Each `Entry` contains a `Symbol` and a reference to a `sibling` in the `Scope`. So, we can iterate over the `Entries` in a `Scope` by starting with the contained `Entry` and following the `sibling` links. The final `Entry` contains `null` as its `sibling`.[15]

Each `Entry` also refers to the `Entry` it shadows in an enclosing `Scope`. The shadowed `Entry` is contained in field `shadowed`.

New `Symbols` may be destructively added to a `Scope` with method `addSymbol`. `Entries` in a `Scope` may be accessed with method `lookup` that takes a `Name` and returns the `Entry` in that `Scope` corresponding to the given `Name`. `Names` are identifiers; they can be constructed from `Strings` with the static method `Name.fromString(String)`. For more information on `Names`, see their unit tests in package `edu.rice.cs.nextgen.util`. For convenience, `lookup` is overridden with a method that takes a `String` and converts it to a `Name`.

Each `Scope` refes to its enclosing `Scope`, in field `nextScope`. Consequently, `Scopes` mirror the structure of the `Environments` they're associated with.[16]

8.5.1 Environments and Scopes as PrintableObjects

Like `Trees`, `Environments` and their constituents can be printed to examine their contents. For example, here is a call on `longString()` of an empty `Scope` (constructed from a default `ClassSymbol` available in `NextGenTestCase`):

```
> new Scope(NextGenTestCase.CLASS_SYMBOL).longString()
[Scope: ]
```

Using `addSymbol` and `lookup`, we can build and examine `Scopes` in the DrJava interactions window:

```
> s = new Scope(NextGenTestCase.CLASS_SYMBOL)
edu.rice.cs.nextgen.compiler.code.Scope@7dfcf1
> s.addSymbol(NextGenTestCase.METHOD_SYMBOL);
> s.lookup("METHOD_SYMBOL")
edu.rice.cs.nextgen.compiler.code.Scope$Entry@7abaab
```

[15]Obviously, `Scopes` and `Entries` are rich with opportunities for refactoring. Because they represent list-like structures, it would be better to make them either subtypes of type `List`, or containers, each with a field of type `List`.

[16]This mirrored structure is undesirable because it can result in "Rogue Data" where one of the two structures is modified but the other is not. A potential refactoring would be to eliminate the `nextScope` field and replace it with a method that accesses the scope contained in the enclosing `Environment`.

```
> s.lookup("METHOD_SYMBOL").longString()

"[MethodSymbol METHOD_SYMBOL]
"
```

We can also build and print `Environments`[17]:

```
> new Environment(null, null)
edu.rice.cs.nextgen.compiler.comp.Environment@53be7c
> new Environment(null, null).longString()

"[Environment
  null
  null]
"
```

The two arguments to the constructor are the associated `Tree` and the associated context. Likewise, the `longString` representations of `Environments` include the associated context and tree. Here is a more complex example:

```
> Tree.TopLevel simpleTree = NextGenTestCase.makeClassTree("public class C {}");
  Environment env = new Environment(simpleTree, null);
> env.longString()

"[Environment
  null
  {TopLevel for null
    packageId: null packageSymbol: null
    starImportScope (except for java.lang): null
    namedImportScope: null
    ==========
    {ClassDef C<> extends null implements () flags: 1
      null
      }}]
"
```

`Environments` can be spawned from other `Environments` as follows:

[17]Raw types are used in this example because, at the time of this writing, the DrJava interactions window does not yet support generic types. In some unit tests where the context is unimportant, `Environments` are instantiated as `Environment<VoidContext>`. Class `VoidContext` is never used outside the unit tests.

```
> e1 = new Environment(null, null);
> e2 = e1.spawn(null);
```

When printed, spawned `Environments` first print out their own contents, and then print the contents of their enclosing `Environment`:

```
> e2.longString()

"[Environment
  null
  null]
[Environment
  null
  null]
"
```

Of course, we can also spawn `Environments` containing non-null `Trees`:

```
> Tree.TopLevel innerTree = NextGenTestCase.makeClassTree("public class C {}");
  Environment innerEnv = new Environment(innerTree, null);
  Tree.TopLevel outerTree = NextGenTestCase.makeClassTree("public class D {}");
  Environment outerEnv = innerEnv.spawn(outerTree);
> outerEnv.longString()

"[Environment
  null
  {TopLevel for null
    packageId: null packageSymbol: null
    starImportScope (except for java.lang): null
    namedImportScope: null
    ==========
    {ClassDef D<> extends null implements () flags: 1
      null
      }}]
[Environment
  null
  {TopLevel for null
    packageId: null packageSymbol: null
    starImportScope (except for java.lang): null
    namedImportScope: null
    ==========
    {ClassDef C<> extends null implements () flags: 1
      null
      }}]
"
```

Notice that the original `Environment` is unaffected by the spawn.

8.6 Symbols

`Symbols` are containers for identifiers. They also hold relevant information such as the bytecode generated for an identifier. The various types of `Symbols` form a composite class hierarchy rooted at class `Symbol`. All subclasses are static nested classes of class `Symbol`. The composite hierarchy is represented in BNF notation in Figure 8.6.[18] Each `ClassSymbol` contains a `Scope` of its members.

```
Symbol ::=  TypeSymbol | VarSymbol | MethodSymbol | OperatorSymbol
TypeSymbol ::= (type variable) TypeSymbol | ClassSymbol | PackageSymbol
```

Figure 2: Symbols

Because each method's bytecode is stored in its corresponding `MethodSymbol`, and because the `Scope` of a `ClassSymbol` contains all `Symbols` for methods (and other members) appearing in a class, all of the information needed to write out a classfile for a given class can be accessed in the class's `ClassSymbol` after bytecode generation.

8.7 SymbolTables and ClassReaders

During Phase II, a `SymbolTable` is constructed and kept as a field in `JavaCompiler` for use in subsequent phases. A `SymbolTable` contains a field `classReader` (of type `ClassReader`) that stores all constituent symbols. `ClassReaders` store their symbols in two `Hashtables`: one mapping package names to their corresponding `PackageSymbols`, and one mapping class names to their `ClassSymbols`. In addition to all classes explicitly referred to in a program, `SymbolTables` include all classes in `java.lang`. They also contain mappings for all Java infix operators. Because the `ClassSymbols` in `java.lang` and the infix operators are the same for

[18]Notice that concrete class `TypeSymbol` forms the root of a composite subhierarchy, and also represents naked type parameters! An important refactoring would be to make `TypeSymbol` abstract and add a new subclass for naked type parameters.

all `SymbolTables`, they are omitted from the `longString` representation. To see precisely what is entered during `SymbolTable` initialization, refer to the source file `edu.rice.cs.nextgen.compiler.comp.SymbolTable.java`.

In the DrJava interactions session below, we create a new `SymbolTable` corresponding to `NextGen` acceptance test `InstanceofParameter.java`. The text of that test is:

```
package edu.rice.cs.nextgen.simpletests;

public class InstanceOfParameter<T> {
  public boolean is(Object o) {
    boolean ret = o instanceof T;
    return ret;
  }
}
```

Here is the interactions session:

> *load JavaCompilerTestSetup.hist*

```
> trees = _compiler.parseFiles(_fileNames);
> envs = _compiler.enterClasses(trees);
> _compiler.getSymbolTable().longString()

"[SymbolTable
  packages:
      edu.rice.cs.nextgen => [PackageSymbol edu.rice.cs.nextgen]
      edu.rice => [PackageSymbol edu.rice]
      edu.rice.cs.nextgen.simpletests =>
        [PackageSymbol edu.rice.cs.nextgen.simpletests]
      edu.rice.cs => [PackageSymbol edu.rice.cs]
      edu => [PackageSymbol edu]

  classes: edu.rice.cs.nextgen.simpletests.InstanceOfParameter =>
    [ClassSymbol  fullname: edu.rice.cs.nextgen.simpletests.InstanceOfParameter
        flatname: edu.rice.cs.nextgen.simpletests.InstanceOfParameter flags: 4194305
        members (cyclic references have been filtered):
           [Scope: [MethodSymbol is]
                          [MethodSymbol <init>]]]
           ]
```

"

8.8 CodeGenerators, GenContexts, and ClassWriters

Class `JavaCompiler` drives class generation by first using a `CodeGenerator` visitor to write bytecode to a `ClassSymbol`, and then passing that `ClassSymbol` to method `ClassWriter.writeClassFile()`, on an instance of `ClassWriter` stored in the `SymbolTable`.

When generating bytecode for a given `Tree`, `CodeGenerators` keep information for each subtree in an `Environment<GenContext>`. Class `GenContext` is a non-public class, written in source file `CodeGenerator.java`, that stores the following information for each subtree:

1. The `expectedType` of the subtree (a `Type`).

2. All unresolved `exitingJumps` out of the code represented by the subtree, represented as aa `Chain`. `Chains` are described below.

3. All unresolved `continuingJumps` to the code represented by the subtree, also represented as a `Chain`.

Class `Chain` is a static nested class of class `Code` (contained in source file `Code.java`).[19] Conceptually, `Chains` are lists of jumps. Each `Chain` contains three fields:

1. The `next Chain` in the list.

2. `pc` (an `int`). This field represents the position of the jump instruction.

3. `stackSize` (an `int`). The stack size after the jump instruction.[20]

An important invariant of a chain is that all elements of the chain list have the same `stackSize`.

[19]Instances of `Code` represent all information that goes into a code attribute in a classfile.

[20]When the locations of jumps are resolved, the compiler ensures as a sanity check that the `stackSize` non-negative in each block of code. In an old (unreleased) version of the `NextGen` compiler, jump targets were resolved incorrectly because snippet methods moved the locations of all code in a parametric class. As a result, the sanity check on `stackSizes` would fail during compilation of some classes.

8.8.1 Writing to Disk

Once all bytecode has been generated and stored in the `ClassSymbols`, `ClassWriters` are responsible for writing class files to disk. The `ClassSymbols` to be written are not stored in class `ClassWriter`. Instead, each class component is passed as an argument to the appropriate method, and written to a file.

9 Conclusion

We have attempted to provide a high-level roadmap of the `NextGen` code base, capturing enough detail that the reader possesses an accurate understanding of the architecture, while keeping the discussion at a level high enough that there is an advantage to reading this document as opposed to simply perusing the code base. Consequently, there are many details of various stages of compilation that have been omitted. The reader is encouraged to bolster his understanding of the code base through interactive exploration in the DrJava interactions window and through adding unit tests. Additionally, an IDE such as CodeGuide that allows one to quickly navigate from uses of identifiers to definitions of identifiers, and that provides support for automated refactoring of code, can be extremely helpful when working with this code base.

The source for this document is contained in the NextGen code repository, in the `doc` directory. Readers are strongly encouraged to update and improve this document while using it to learn the code base. New readers are in a particularly good position to assess the clarity and usefulness of this document. By improving it, you will help future generations of NextGen programmers to understand NextGen more quickly, just as this document has helped you to do so.

References

[1] O. Agesen, S.. Freund and J. Mitchell. Adding Type Parameterization to the Java Language. In OOPLSA'97.

[2] D. Ancona and E. Zucca. A Theory of Mixin Modules: Basic and Derived Operators. Mathematical Structures in Computer Science, 8(4):401–446, 1998.

[3] E. Allen, J. Bannet, R. Cartwright. First-class Genericity for Java. Submitted to ECOOP 2003.

[4] E. Allen, J. Bannet, R. Cartwright. Mixins in Generic Java are Sound. Technical Report, Computer Science Department, Rice University, December 2002.

[5] E. Allen, R. Cartwright, B. Stoler. Efficient Implementation of Run-time Generic Types for Java. IFIP WG2.1 Working Conference on Generic Programming, July 2002.

[6] E. Allen, R. Cartwright.The Case for Run-time Types in Generic Java. Principles and Practice of Programming in Java, June 2002.

[7] D. Ancona, G.Lagorio, E.Zucca. JAM-A Smooth Extension of Java with Mixins. ECOOP 00, LNCS, Spring Verlag, 2000.

[8] J. Bloch, N. Gafter. Personal communication.

[9] R. Cartwright, G. Steele. Compatible Genericity with Run-time Types for the Java Programming Language. In *OOPSLA '98*, October 1998.

[10] G. Bracha, M. Odersky, D. Stoutamire, P. Wadler. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In *OOPSLA '98*, October 1998.

[11] M. Flatt, S. Krishnamurthi, M. Felleisen. A Programmer's Reduction Semantics for Classes and Mixins. Formal Syntax and Semantics of Java, volume 1523, June 1999.

[12] Martin Odersky and Philip Wadler. Pizza into Java: Translating Theory into Practice. In *POPL 1997*, January 1997, 146–159.

[13] Sun Microsystems, Inc. JSR 14: Add Generic Types To The Java Programming Language. Available at `http://www.jcp.org/jsr/detail/14.jsp`.