5.1 Java language processing tools framework

The architecture that we plan to use for supporting language levels for Java is source-to-source translation mapping "language level Java" source to standard Java code. This architecture is designed to minimize the number of components we have to write — namely, to keep from writing a complete source-to-bytecode compiler for each level.

The core of such an implementation of the language levels themselves would be the series of checks and elaborations that map the input to standard Java. However, there are a number of ancillary components that are required to allow the development of this core. There must be a consistent intermediate representation of Java code that supports language levels, a framework for developing a parser for each language level, and a mechanism for translating the intermediate representation back to source. These substantial components had to be developed before language level development could proceed.

This section outlines the development of the tools I developed to support Java language processing.

5.1.1 Java AST

The heart of programming language implementations is the intermediate representation of the code. These data structures form the interface used between all different components of language processing systems. That said, we decided that we wanted to invest some effort to create a nice, reusable abstract syntax for Java (including

support for GJ-style parametric types), one that we'd be happy to use for language levels and other future projects. (We would have been happy to use a pre-existing AST framework for Java, but we found none that fit our criteria and were available under acceptable licensing terms.)

Our previous experience working on Java language projects (NextGen [37], which is based on the Generic Java [38] code base) suggested three criteria of an easy-to-use Java AST framework: (i) immutable objects, (ii) no usage of `null` in ASTs, and (iii) strong static typing of nodes (having each concrete class only represent one form of syntax). Also, since we expect there to be many different functions to be applied to the AST, we wanted to use the Visitor pattern to allow these functions to be developed independently of the AST classes themselves. (In fact, we decided against putting *any* behavior in the AST classes at all, other than equality testing, hash code creation, and String generation for debugging.)

To create such a framework would require building scores of classes, all of which were basically just data structures organized in a complicated inheritance hierarchy, and all needing common utility methods (`.equals`, `.hashCode` and `.toString`). These classes would have substantial redundancy (just to rename a class would affect all classes that contained any references to it), making maintenance a nightmare.

Because each class was to have so little behavior, and because the behavior it was to have could be easily algorithmically determined, we decided the best way to proceed was to develop a specification language for AST-like structures and then to write a tool to generate the Java source from such a specification. This resulted in

the tool ASTGen (documented in the next section). With this written, creating our Java AST classes was simply a matter of creating the input specification to feed to the tool. This specification (which also serves as good overview documentation for the class hierarchy) is presented in Appendix A.

5.1.2 ASTGen

ASTGen* is a tool I wrote to generate code for a Composite pattern hierarchy of immutable† data objects. The need to generate such a hierarchy comes up in many circumstances, particularly when creating compilers and interpreters. The purpose of this tool is to allow the maintenance of the tree structure in a simple, high level description, and then to automatically generate the code to implement it. Note that its purpose is that the output of the tool is never edited; any changes are instead made in the input file. Thus, this input becomes the single point of control for the entire hierarchy, vastly simplifying maintenance. Because the data classes have no explicit behavior, they can be acted upon by visitors.

ASTGen is distributed under the General Public License [6] and is available from `http://astgen.sourceforge.net`.

Input file syntax

The input syntax for ASTGen is terse and simple. It is designed so that the input syntax can be easily read by a human as well as the tool, so that it can serve as simple

*__A__bstract __S__yntax __T__ree __Gen__erator, although it can be used to generate other hierarchies of immutable data objects as well.

†The tool could fairly trivially be extended to support mutation, but thus far we have had no need for this, as we prefer to keep things immutable when possible.

documentation for the class structure. An example of an input file to define a simple expression structure is shown in Figure 5.1.

```
// Example ASTGen input, for a simple expression structure
visitmethod apply;
visitormethodprefix for;
tabsize 2;
outputdir expr;
allownulls no;

package expr;

begin ast;
interface Expression(int lineNumber);
  // line number is the line the expression was on in when it was parsed
  // It is ignored for equality because it doesn't change the value of
  // the expression
  abstract AbstractExpression(ignoreForEquals int lineNumber);
    abstract BinaryOpExpression(Expression left, Expression right);
      AddExpression();
      MultiplyExpression();

    IntegerConstant(int value);
    FloatConstant(float value);
end;
```

Figure 5.1 An example ASTGen input, defining a simple expression abstract syntax.

Below is an informal description of the input syntax:

- Java-style line comments (beginning with `//`) and block comments (between `/*` and `*/`) are allowed anywhere, and their contents are ignored. Blank lines are also allowed anywhere.

- The first section of the `.ast` file is a list of unary options. (All options must be at the top of the file, before any class declarations.) Each option is set by

putting the option name, a space, and then the value, and then it is terminated by a semicolon. All option statements may be left unspecified, as each option has a default value. The set of valid options is declared below:

visitmethod This is the name of the "visit" or "accept" method that all concrete classes will implement. This is the hook used to run a visitor on a class. The default value is *visit*.

visitormethodprefix This is the prefix for all of the visitors' "case" methods. Each visitor has one such method to handle each concrete class in the hierarchy. The method name is of the form [prefix][class], where [prefix] is the value set by this option and [class] is the name of the class the method handles. The default prefix is *for*.

tabsize The number of spaces to be used to represent an indentation level. (ASTGen does not support using actual tab characters.) This value is used when generating the Java source files, and it's also used when parsing the ASTGen input file itself, for the part that defines the classes in the hierarchy. The default value is *2*.

outputdir This specifies the directory, relative to the location of the input file, where the generated output will be put. The default is ".". *Note: It is recommended to keep the automatically generated files in a different package, so that it can easily be removed when the tool is going to regenerate them.*

allownulls This boolean-valued parameter (yes/no/true/false) specifies whether fields in the generated classes can be assigned the value of `null`. If this is set to no/false, each class's constructor will throw an `IllegalArgumentException` if it is ever given a value of `null` for a parameter. The default value is *no*.

- After the list of options, optionally a package and list of import statements can be declared. These statements are simply copied verbatim into each of the generated Java source files. So, if the generated files are to be in package *edu.rice.cs.javaast.tree*, there needs to be a line `package edu.rice.cs.javaast.tree;` in the input file. Also, any references to other classes that are not in the same package must be explicitly imported via an import statements. *Note: You must ensure that the package statement is consistent with the outputdir.*

- The rest of the file will declare all of the classes and interfaces in the hierarchy. To begin this section, there must be a line of the form `begin ast;`, and there must be a line of the form `end;` at the end. Each line in this section declares one class or interface in the hierarchy. The first class or interface, designated the *root class*, is declared on the line immediately after the `begin ast` statement. It must have no spaces preceding its definition. All succeeding classes are indented multiples of *tabsize* spaces; indentation is used to signify subtyping relationships. The class/interface declared on a line is a subtype of the class/interface declared on the closest preceding line at one lower level of indentation. It is an error for an interface to attempt to be the subtype of a

class.

- Each class/interface is declared by a single line, of a form that is very similar to the declaration of a constructor. The general form of a class/interface declaration line is: `["abstract" | "interface"]` *name* `"("` *paramlist* `")"` `["implements"` *implementslist*`]`. The `abstract` or `interface` keywords signify that the line declares an abstract class or an interface, respectively. *name* is the name of the class or interface being declared. *paramlist* is a comma-separated list of fields declared in this class, or if it is an interface, the fields that all implementing classes must provide getters for.

  Each parameter is of the form `["ignoreForEquals"]` *type* *paramname*, where *type* is the type of the field and *paramname* is its name. `ignoreForEquals` signifies that this field should be ignored for the purposes of determining equality in the automatically generated `.equals` method. In class declarations, a parameter defines a field of that class. In interface declarations, a parameter simply requires that implementing classes have a field by that name, with a standard-named accessor of the form `get`*Field*, where *Field* is the name of the field with the first letter of its name capitalized.

  implementslist is a comma-separated list of additional interfaces that this class/interface should implement/extend, beyond those implied by this class/interface's position in the indentation hierarchy.

Generated output

From a single input file, ASTGen generates many files in order to create a composite hierarchy as defined and to create visitors over it. There is one source file created for each defined class/interface in the input, plus four visitor classes and one standard helper class, `TabPrintWriter`. An example of the top-level interface and some of the visitors generated by ASTGen (based on the input shown in Figure 5.1) is shown in Figure 5.2.

Defined classes For each class line in the input file, a Java class is generated. This class has a `private final` field for each field declared in the class definition, in addition to inheriting all fields from its superclass. A constructor is generated automatically, and the constructor takes as arguments all of the fields, in order, contained in the class and all of its ancestors. (The fields are in declaration order, so those declared at the root of the ancestry tree come first.) The constructor enforces the restriction (if turned on in the options) that no field can have a null value. Also, to allow comparison on `String` instances with `==` and to possibly save memory, all `String`s are automatically interned when they are stored.

To make the data members accessible, a getter method is automatically generated, with the name get*Field*, where *Field* is the name of the field, with the first letter capitalized. (Since the classes are all immutable, there are no setter methods.) Also automatically generated are `.equals` and `.hashCode`, which implement member-wise equality, and a `.toString` method, which prints the values of all the fields so as to

form a nice tree.

Each class also has two "accept" methods to use with visitors, one for visitors that return a value and another for visitors that do not. The value-returning visitor is parametric (using Generic Java-style type parameters), so the accept method is of the signature: `<T> public T accept(`*RootClass*`Visitor<T> v)`. The other accept method is of the signature: `public void accept(`*RootClass*`Visitor_void)`.

Defined interfaces Interfaces defined in the input file have a getter method for each "field" that is declared for it. Also, the interfaces mandate the presence of the accept methods that all classes generated by ASTGen will have, so that visitors can be dispatched on variables with a static type of an interface.

Visitor interfaces/classes Two visitor interfaces are created for each ASTGen input file: *rootClass*`Visitor<T>`, a Generic Java interface supporting a visitor that returns an instance of type parameter `T`, and *rootClass*`Visitor_void`, an interface supporting a visitor that returns no value. Each of these visitor interfaces has one method for each concrete class in the defined composite hierarchy, named [*visitorMethodPrefix*][*className*].

Also automatically generated are two abstract visitor classes that simplify creating visitors that process data in a depth-first fashion: *rootClass*`DepthFirstVisitor<T>` and *rootClass*`DepthFirstVisitor_void`. These visitors, patterned off the depth-first visitors generated by JTB [39], allow subclasses to implement only the part of each

case method that actually processes the subcomputed results and returns a value, without having to explicitly include the recursive calls. They also simplify visitor implementation by having each case's visit method default to calling the superclass's visit methods. (See the example output in Figure 5.2 and the simple evaluation visitor in Figure 5.3 for better illustration.)

TabPrintWriter This helper class is generated, verbatim, in every ASTGen output. It is used in the implementation of the `.toString` methods in the generated classes, to simplify the output of nicely indented trees.

5.1.3 A first attempt at a parser

With an abstract syntax defined, the next step needed to effectively process Java source is a parser that produces these ASTs. Our first inclination was to use a parser generator, particularly one that already had a defined Java grammar. Since a student in our research group had already adapted the JavaCC grammar for Java to support GJ-style generics, we decided to start there, adapting the grammar to produce our new AST. (It previously generated the AST that was produced by JTB [39].)

This effort produced a working parser for Generic Java, written in Java, that produced our abstract syntax. The parser is available under the LGPL [40] from `http://javalangtools.sourceforge.net`.

This parser, while perfectly useful for many language projects, proved difficult to extend towards our goals for language levels. Error handling is an important part of the language levels implementation; at each language level, the error messages must fit