Experiences in Building a Compiler for an Object-Oriented Language

José de Oliveira Guimarães
Departamento de Computação
UFSCar, São Carlos - SP, Brazil
jose@dc.ufscar.br

Abstract

Traditionally books on compiler construction focus on procedural languages leaving some questions on compilation of object-oriented languages unanswered. This article presents some problems found in the building of a compiler for the object-oriented language Green using Java. The solutions used for these problems are exposed and whenever possible alternatives are discussed. We hope to provide programmers with some hints and caveats on compilation of object-oriented languages.

Keywords: Compiler construction, object-oriented languages, Java, Green.

1 Introduction

This article describes some problems encountered during the construction of a compiler, in Java, for the object-oriented language Green [5]. These problems are, in general, related to the programming of the compiler itself and not to topics well explored in the literature such as parsing and code generation. Compiler construction books [1] [3] [4] focus on techniques for lexical analysis, parsing, building of abstract syntax tree, semantic analysis, and code generation and optimization. They explain in high level how to implement a compiler, which is sufficient for the implementation of a small Pascal compiler but insufficient for the implementation of a much more complex object-oriented language such as Green.

This article exposes some problems found in the building of the Green compiler and shows how the problems were solved. To begin with, let us show a little bit of Green. This is a statically-typed object-oriented language which supports garbage collection, parameterized or generic classes, method overloading, classes as classless objects, arrays as classes, separation of subtyping and subclassing, metaobjects, and an introspective reflection library.

Figure 1 shows a class `Store` in Green which has methods `get` and `set` and instance variable `n`. The type of a class is the set of public method signatures. The signature of a method is its name, parameter types, and return value type, if any. Then, the type of `Store` is

```
{ get() : integer, set(integer) }
```

Whenever an object of type `Store` is expected, as in an assignment or parameter passing, we can use an object of a class that has at least the same method signatures as `Store` (methods `get` and `set`). This is checked at compile time.

Figure 2 shows the declaration of a class object. That is, an object that represents the class `Store`. This object is similar to objects of prototype or delegation-based languages — it does not have a class. The compiler adds a "`new() : Store`" method to this object. This method returns a newly created `Store` object.

```
class Store
   public:
      proc get() : integer
         begin
         return n;
         end
      proc set( pn : integer )
         begin
         n = pn;
         end
   private:
      var n : integer;
end
```

Figure 1: Class `Store` in Green

```
object Store
   public:
      proc getMax() : integer
         begin
         return 2002;
         end
end
```

Figure 2: Class object `Store`

The declaration of a class object and the corresponding class should be put in the same file which must have the name of the class.

Class `Store` of Figure 1 declares the methods and variables objects of `Store` have at run time. Class object `Store` of Figure 2 defines a classless object representing the class itself. This object may receive messages as any other:

```
m = Store.getMax();
```

Although class objects have no class, they do have a type and the subtype rules also apply to them. The type of class object `Store` is

```
{ getMax() : integer, new() : Store }
```

In building the Green compiler we have met several problems whose solutions are not commented in the literature. These problems are related to forward references, cross references, parameterized classes, code generation tools, management of the Abstract Syntax Tree (AST), etc. In the next sections we describe each of these problems and the solutions we used.

Our compiler builds an AST of the Green code and uses it to generate code. Each node of the AST is a Java object which has a method `gen` that generates code, also in Java. The compiler takes as input a text file which contains a list of all classes that compose the program. Then, when it finds a declaration "`var a : A;`" when compiling the code of a class, it knows whether `A` is a valid class or not.

2 Forward and Cross references

In Figure 1, variable `n` is referenced in methods `get` and `set` although it is declared at the end of the class. When the compiler finds `n` in method `set`, it does not know what "`n`" is: a variable, a constant, a method name, or something else. Then it cannot build the AST node corresponding to "`n = pn`". Assume the AST should have all the information available in the program. This will be justified below.

A similar problem occurs when a class `A` declares a variable of class `B` and vice-versa. To do semantic analysis in class `A` we need to know the methods declared in `B` and vice-versa. So we cannot do two things in a one-pass compiler: to build the AST and do semantic analysis.

We opted for a compiler with two passes: the first collects interface information of all classes that compose the program and the second builds the AST and does semantic analysis. So there are two lexical and two syntactic analyses.

The first step creates one object for each program class. The object describes the class interface. The interface contains not only the method signatures[1] but also the instance variables.

The types of parameters, instance variables, and return of methods are represented as strings in the interface object. That cannot be different, for a type may be a class not yet analyzed. But this is unfortunate, since later on the compiler will need detailed information on these types, such as the methods each one defines.

To solve this problem, each interface object has a `reify` method that sets each type that appears in the interface to the interface object that represents that type. Of course, each type is represented by two variables, a `String` and an interface. The last variable is set by `reify`.

The compiler collects all program interfaces and puts them in a hash table. Then it calls the `reify` method of each object interface.

To do two lexical and two syntactic analyses is not the only solution. Two others were abandoned: the first is to do only one lexical and syntactic analyses. The AST built in the

[1] The signature of a method is composed by its name, parameter types, and return type (if any).

syntactic analysis would not be correct because some information would be missing, like what is "`n`" in the statement "`n = pn`" of method `set` of `Store`. In these cases the compiler would use fake AST objects that would be latter replaced by real ones. That is, "`n = pn`" would result in the creation of an `AssignmentStatement` object of the AST which references an object of `FakeVariable` which keeps the string "`n`" — we don't know at this point what "`n`" is. The `AssignmentStatement` object would also reference a `Parameter` object that represents "`pn`". We know what `pn` is since it is a parameter. And parameters are declared before they are used.

After compiling the whole class `Store`, we know "`n`" is an instance variable and then the compiler can replace the `FakeVariable` object by an `InstanceVariable` object.

This solution is complex because we would have to assure the trick described above works in all situations and with all AST classes. Another problem is that sometimes the fact an AST object belongs to a class forces another AST object (that references the first) to belong to another class. It is as if "`n`" in the example above belonged to a class `StaticVariable` we would have to change the class of the object `AssignmentStatement` to `StaticAssignmentStatement`. Complex, at least.

One could argue that the AST is an *abstract* tree and therefore should not contain detailed information such as whether "`n`" in "`n = pn`" is an instance or static variable. That is not true. In our compiler, the AST should not be too abstract because it is used for code generation,[2] an operation that demands some detailed information on `n` and `pn`. For example, suppose the compiler generates machine code. The AST needs to know whether `n` is static or instance variable in order to generate code for "`n = pn`". The generated code for these two options will be completely different from each other.

Our compiler uses a symbol table whose entries are of the kind (`name`, `obj`). `name` is the name of the class, variable, and so forth. `obj` is the name of the AST object describing `name`. Therefore the AST should have all the existing information on identifier `name`. This will be used by the semantic analyzer, which accesses `obj` after looking for it in the symbol table. Then the symbol table points to AST objects which are expected to have the necessary data for semantic analysis.

If the AST is too abstract, it will not supply the data needed by the semantic analyzer. Then this data must be found in some other place. One solution is to attach to the symbol table objects containing the data. Then we would need to create new classes for these objects, making the design complex because information on identifiers will be in two different places, the AST and the symbol table.

Our AST has all the information on the program but it can only be built by a two-pass compiler. A really abstract syntax tree can be built in a single pass compiler. However, semantic analysis will be more difficult to do than in a two pass compiler. It can only be done after the AST is built and the syntactical analysis ends. Then the semantic analyzer will have to build object trees containing the data it needs, which will not be found in the AST. And it will have to keep information on the program text, which will be used in printing error messages. If some semantic error occurs, the compiler should print an error message, the offending line, and its number. A little bit complex.

The other alternative solution for doing two lexical and two syntactic analyses is to do just one lexical analysis with two syntactic analyses. The tokens found in the lexical analysis would be kept in an array and used in both syntactic analyses.

That is not a good solution. The error messages need the number of line and column of each token. Then the compiler would need to keep these data for all tokens — but these information

[2]Each AST class has a method `gen` that generates code for that node and calls the `gen` methods of the AST objects down in the tree.

would only be used in case of an error. A waste of computer power.

Our compiler makes two double analyses in the code. This may not be the fastest, but it is probably the simplest solution.

3 Parameterized Classes

Suppose we have two parameterized classes `A` and `B` with two parameters each:

```
class A(U, V)
   // methods were omitted
   private:
      var x : B(V, U);
end

class B(G, H)
   private:
      var y : A(G, H);
end
```

The use of `A` in a declaration

```
var w : A(char, Point);
```

causes the creation of a real class "`A(char, Point)`". This class is obtained by replacing `U` by `char` and `V` by `Point` in the body of `A`. Inside `A`, the declaration of `x` becomes "`x : B(Point, char)`" and the compiler must create "`B(Point, char)`". To create this class, it replaces `G` by `Point` and `H` by `char` which demands the creation of "`A(Point, char)`". In its turn, "`A(Point, char)`" demands the creation of "`B(char, Point)`", which demands the creation of "`A(char, Point)`" which has already been created and the cycle ends:

`A(char, Point)` ⟶ `B(Point, char)` ⟶ `A(Point, char)` ⟶ `B(char, Point)` ⟶ `A(char, Point)`

Then to create a parameterized class is not as simple as "replace the formal by the real parameters". And when replacing the parameters, care must be taken to not replace "parameters" inside strings. In fact, there is a worse problem: the parameter may be inside a Java block, which is a piece of Java code inside a Green class. A Java block is delimited by the words `java$begin` and `java$end`. Let us see an example:

```
java$begin
   // this is a Java code inside a Green code
w = new U[max];
...
java$end
```

In this case, `U` must be replaced not by `char` or `Point` but by the name in Java of `char` or `Point` — which is `char` (the same) or `_Point`, whenever the case. Green names are translated to Java, in general, by prefixing them by underscore.

The creation of a real class `A(char, Point)` from a parameterized class `A` is called instantiation of `A`. In our compiler a real class `A(char, Point)` is created from parameterized class `A` by replacing `U` and `V` by `char` and `Point` in a copy of the text of `A`. The compiler could use other mechanism to instantiate `A`. It could first analyze class `A` and build an Abstract Syntax Tree (AST) of it. Then a copy of the AST is made and the parameters `U` and `V`, represented by objects

in the AST, are replaced by objects representing the types `char` and `Point`. This mechanism saves a lexical and syntactic analysis of `A` for each new instantiation. However, to make a copy of the AST of `A` is complex for neither shallow nor deep copy are appropriate. Therefore, to implement this copy we would need to add to almost every AST class a `copy` method. And there are more than one hundred of them.

4 Mapping of the Green Type System to Java

The Green type system is more general than the type system of the target language of the compiler, Java. This is because Green separates subtyping from subclassing. A class `B` is subtype of class `A` if `B` has at least all method signatures of `A`, regardless `B` inherits from `A` or not. If `B` is subtype of `A`, an object of class `B` can be used where an object of `A` is expected, as in an assignment

```
a = b
```

in which the declared types of `a` and `b` are `A` and `B`.

In Java, this assignment is correct if `B` is a descendent from `A`. Here we consider that `B` may inherit from class `A` or `B` may implement interface `A`, directly or indirectly.

Since Green does not demand that `B` be an `A` descendent, Green is more general than Java — there are less requirements in the former than in the latter language. In this example, Green only requires that `B` has all method signatures of `A`.

Then we have a problem, since the compiler translates Green to Java, a more general to a more restrictive language. However, we will soon discover that this incompatibility is not real.

To show how the Green type system is mapped into Java, we will use an example. Suppose a class `B` in Green has a method

```
proc m( x : integer ) : boolean
```

among others.

Every Green class is translated to a Java class with the methods defined in the Green class. `B` is translated to `_B` in Java. If `B` inherits from `A`, `_B` inherits from `_A`. If `B` does not inherit from any class, `_B` inherits from `_Any`. `Any` in Green is the top-level class — every class inherits directly or indirectly from it.

A Java interface is generated for every `B` method. For `m`, the interface is

```
interface m$1$integer$boolean {
   boolean m(int x);
}
```

Class `_B` implements the interfaces corresponding to all its methods, including the inherited ones:

```
class _B extends _Any
         implements
             m$1$integer$boolean,
             ...
{
  ... // methods and instance variables
}
```

Now we show how variable declaration and method call are translated to Java. Let us study the code

```
   // Green
var b : B;
...
ok = b.m(0);
```

which generates

```
   // Java
_Any _b;
...
_ok = ((m$1$integer$boolean ) _b).m(0);
```

Every variable in Green whose type is a class is declared in Java with the type `_Any`. Suppose then `x` and `y` are variables in Green whose types are classes and that "`x = y`" is type correct. This assignment generates "`_x = _y`" in Java which is also type correct since both `_x` and `_y` have the type `_Any`.

Returning to the example, the call "`b.m(0)`" in Green cannot be translated to "`_b.m(0)`" in Java because the class of `_b`, `_Any`, does not define a `m` method. Then it is necessary to cast `_b` into interface `m$1$integer$boolean` and then call `m`. But how can we be sure the object `_b` refers to at runtime belongs to a class that implements `m$1$integer$boolean` ?

Code generation will only take place if there is no error in the Green code. Then assuming "`b.m(0)`" is type correct, at runtime `b` points to an object that has a method

```
proc m( x : integer ) : boolean
```

This is guaranteed by the type system of Green. In the general case we do not know the class of this object — it may be `B` or any of its subtypes. But we do know the class of the object pointed to by `b` has a `m` method. Therefore the corresponding Java class implements interface `m$1$integer$boolean`. Then `_b` can be cast to this interface and the method called.

5 General Implementation Problems

This section presents some problems that are independent of the language Green.

5.1 Tools for Compiler Construction

The first version of the Green compiler used CUP [7] and JLex [2], which are the YACC/Lex for Java. These tools were abandoned for several reasons:

1. they generate code for the lexical analyzer and parser that is very difficult to debug. We can understand neither the code they generate nor our own code, which is interwoven in the generated code and modified by the tools;

2. a production of CUP like

   ```
   VarDec ::= VAR ID:name COLON Type:t SEMICOLON {:
       RESULT = new VarDeclaration(name, t);
       :}
   ```

   allows one to manipulate `name` and `t` given by terminal `ID` and rule `Type`. But `VarDec` cannot pass parameters to `ID` and `Type`, which is sometimes necessary and therefore demand the use of global variables. These global variables introduce a lot of errors because we never know when they are correctly set up;

3. `VarDec` returns a value (with `RESULT`). Then it must have been previously declared as returning a `VarDeclaration` object in the beginning of the text defining the CUP grammar. This declaration should be at the `VarDec` declaration to help to keep consistence between declaration and use;

4. some grammar errors are only detected at runtime, as to use the return value `t` of `Type` if `Type` does not return anything. For short, CUP does not do semantic analysis in its input;

5. we spend more time to do a correct parser using CUP than by hand using the recursive descendent method. This may not to apply to everybody but certainly applies to most people, including the author of this article, which has experienced both methods. The same applies to JLex;

6. error recovery is at least difficult. It cannot be fine-tuned to our needs, the mechanism for recovery is fixed by CUP;

7. the error messages CUP and JLex emit are cryptic, very difficult to understand. But this problem can be corrected by perfecting these tools.

We ended doing all the compiler by hand using the recursive descendent method by the reasons exposed above.

The advantage of using tools like CUP is that the language grammar is mirrored in the text given as input to the tool. This guarantees the compiler is faithful to the language. The same cannot be said about recursive descendent parsers built by hand. Their codification usually demand modifications in the language grammar. And mismatches between the grammar and the syntactic analyzer are easily introduced.

5.2 Abstract Syntax Tree

The Green Abstract Syntax Tree (AST) has more than one hundred classes. Since Java requires a file for each class, we have more than one hundred files for the AST, which imposes tremendous difficulties in managing this library. During the compiler development, the AST classes need to be modified at all times. And one modification in one class usually triggers changes in other classes. Then it would be nice to have all the AST in just one file, which is much easier to manipulate then one hundred. And that was just what we did.

We created a tool called classgen [6] that takes as input a text file with all classes of the AST and produces a file for each class. Some commands for the tools can be embedded in the input file such as to produce `get` and `set` methods for instance variables.

6 Conclusion

This article exposed some of the problems found in the building of a compiler for the Green object-oriented language. A lot of more specific problems were left for lack of space. These are related to:

1. method overloading — how to implement it, what data structures and search methods should be used at compile time;

2. code generation for the Green exception system, assertions, arrays, basic types, introspective reflection library, etc;

3. separation of class and type in Green — a type is a set of method signatures, even class objects, which are classless, have types;

4. name server. There is a class that supplies Java names used in the generated code.

This paper intended to fill a gap between books on compiler construction, which teach the subject in terms of high level concepts, and real compilers, which are too difficult to understand by reading their code. Unfortunately we were only able to discuss a small part of our experience with the Green compiler in this article. We hope to put more material on the compiler at [5].

Aknowledgement. This work was partially financed by FAPESP under process number 99/13006-8.

References

[1] Aho, Alfred V. and Ullman, Jeffrey D. Principles of Compiler Design. Addison-Wesley Publishing, 1977.

[2] Ananian, C. Scott. JLex: A Lexical Analyzer Generator for Java. `http://www.cs.princeton.edu/~appel/modern/java/JLex`

[3] Appel, Andre W. Modern Compiler Implementation in Java. Cambridge University Press, 1998.

[4] Grune, D.; Bal, H.; Jacobs, J.H. and Langendoen, K. Modern Compiler Design. John Wiley & Sons, 2000.

[5] Guimarães, José de Oliveira. The Green Language. Available at `http://www.dc.ufscar.br/~jose/green/green.htm`.

[6] Guimarães, José de Oliveira. The Class Generation. Available at `http://www.dc.ufscar.br/~jose/courses/cc/classgen.html`.

[7] Hudson, Scott E. CUP Parser Generator for Java. `http://www.cs.princeton.edu/~appel/modern/java/CUP`