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

February 22, 2002

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.

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

```
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`

```
    { 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.

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 }
```

Having introduced Green, let us show the problems found in the compiler and the solutions employed, which is made in the next section.

2 Problems and Solutions

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 subsections 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.

2.1 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`".

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 do semantic analysis. So there are two lexical and two syntactic analyses.

But that 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 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`".

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.

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

2.2 Code Generation Tools

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 experiented 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 criptic, 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.

2.3 Code Generation

Some basic Green classes like `char` or `byte` and the IRL cannot be implemented in Green alone. They need to be implemented, at least partially, in Java, which is the target language of the compiler. Then we had to add a statement to Green: `java$begin` and `java$end`. They delimit a piece of Java code inside a Green method or class.

2.4 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)` $\longrightarrow$ `B(Point, char)` $\longrightarrow$ `A(Point, char)` $\longrightarrow$ `B(char, Point)` $\longrightarrow$ `A(char, Point)`

Then to create a parameterized class is not 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:

```
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.

2.5 Abstract Syntax Tree

The Green Abstract Syntax Tree (AST) has more than 100 classes. Since Java requires a file for each class, we have more than 100 files for the AST, which imposes tremendous difficulties in managing the AST. 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.

3 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. simulation of the Green type system using Java, since Java code is generated from Green code. This is interesting because the Green type system is more general than that of Java and even though it is possible to simulate it in this language;

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

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

5. name server. There is a class that supplies Java names used in the generated code. For example, method set of class `Store` of Figure 1 is translated to `_set$1$int`.

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].

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. Moder 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`.

[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`