An Automatic "Visitor" Generator for Ada

Martin C. Carlisle
Department of Computer Science
2354 Fairchild Dr., Suite 1J133
US Air Force Academy, CO 80840
carlislem@acm.org

Ricky E. Sward
Department of Computer Science
2354 Fairchild Dr., Suite 1J134
US Air Force Academy, CO 80840

1 Introduction

This paper describes work that has been done to automatically generate code that implements the Visitor design pattern for a specified grammar. This work is an extension of the work done on the Ada Generator of Object-Oriented Parsers (AdaGOOP) [Car00]. AdaGOOP creates lexer and parser specifications automatically given a specific grammar. The lexer and parser specifications include all methods needed to generate the parse tree. Our work extends AdaGOOP by automatically including the code for visitor objects that can perform operations on the parse tree generated by AdaGOOP.

This paper first describes the Visitor design pattern and the benefits of implementing this pattern, then describes the AdaGOOP tool in more detail and how the tool was extended to automatically generate visitor objects. Finally, we present examples of the visitor objects generated by AdaGOOP.

2 Visitor Design Pattern

Implementing the Visitor design pattern, as presented in [GHJV95], is intended to simplify the task of traversing a parse tree to perform a specific task on each node. Common tasks performed on parse trees include pretty-printing, code optimization, code generation, and type checking. A compiler that does not use the Visitor design pattern will include code for each of these tasks, but the code will be distributed throughout the compiler. An object-oriented compiler will most likely include the code for these tasks within each class that implements the objects in the grammar.

For example, as shown in Figure 1, the code for tree traversal operations involving a statement is included as a method in each of the sub-classes that define the statements, e.g. IF, WHILE, OPEN, and CLOSE. Each sub-class includes a method for pretty printing and for type checking. You can see that distributing the code in this way leads to compiler code that is hard to understand, hard to maintain, and hard to change [GHJV95]. Adding even one new task to be performed requires each affected class to be recompiled.

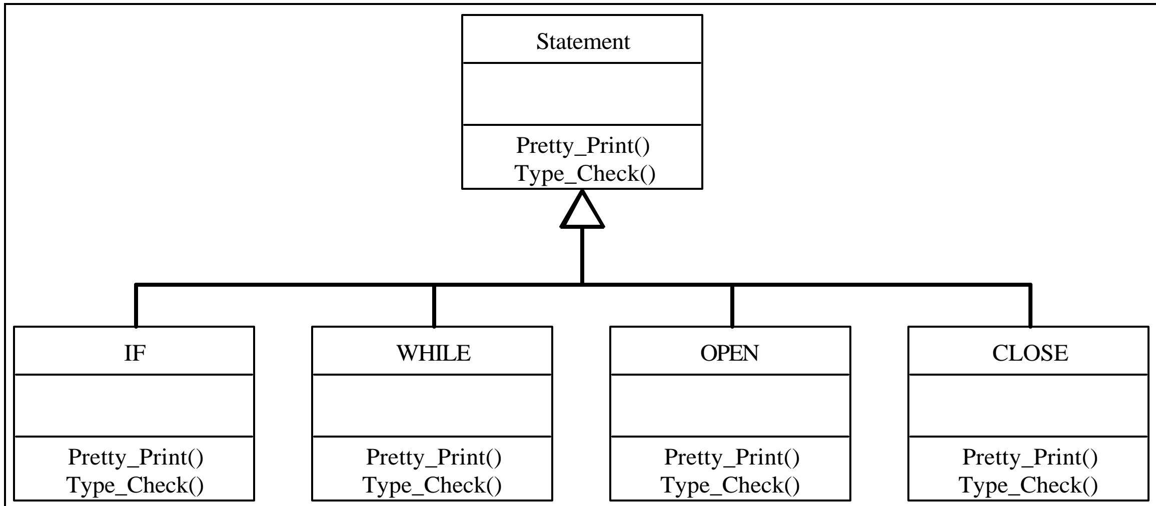

Figure 1 – Tree Traversal Operations without Visitor

Instead of distributing the code for these tasks throughout the classes of the compiler, the Visitor design pattern collects the code for each tree traversal operation into a single class that includes operations for each object in the parse tree. In the purest form of the design pattern, an abstract class is built over the visitor classes.

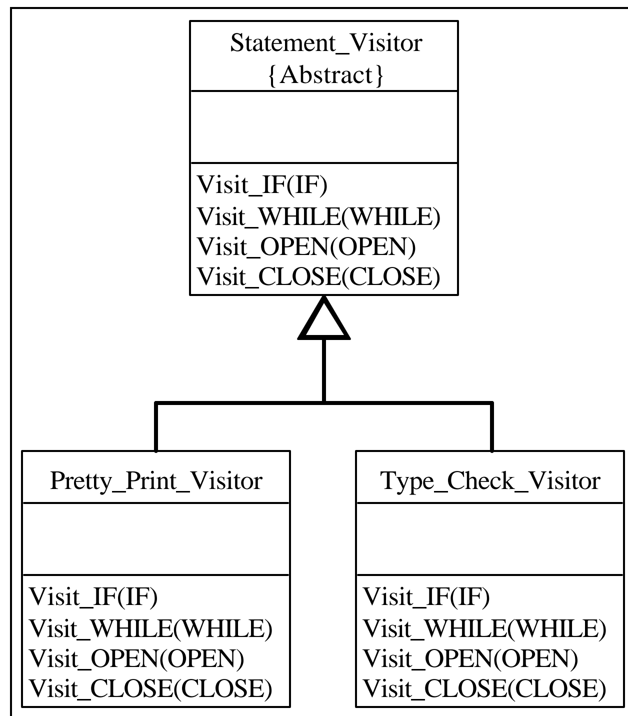

Figure 2 – Tree Traversal with using the Visitor Design Pattern

For example, Figure 2 shows the visitor classes that are needed to implement the pretty printing and type checking tree traversal operations for the statement classes. One class has been built for each tree traversal operation. Each of these classes includes an

operation for each node in the parse tree. An abstract class "Statement_Visitor" is shown in Figure 2 and is a parent to the tree traversal sub-classes.

One challenge for implementing the Visitor design pattern for a specific tree traversal operation is specifying and implementing the methods required for each of the objects in the parse tree. Each object (node) in the parse tree requires a method that implements that specific part of the operation specific to that node. This is a tedious and laborious process that can easily be automated. Our work has extended the AdaGOOP tool to automatically generate the classes and methods needed for the tree traversal operations as shown in Figure 2.

3 AdaGOOP

The AdaGOOP tool [Car00] automatically creates lexer and parser specifications that include all the operations necessary to create the parse tree. The user specifies the grammar for the specific language and AdaGOOP automatically generates the package specification for the parse tree.

```
stmt : IF | WHILE | OPEN | CLOSE;
```

Figure 3 – Production for Statements

For example, Figure 3 shows one production from a small grammar. This production includes the **stmt** non-terminal and shows that **stmt** reduces to either the **IF**, **WHILE**, **OPEN**, or **CLOSE** tokens. The parse tree code generated by AdaGOOP for this production is shown in Figure 4.

```
type stmt_nonterminal is abstract new Parseable with null record;
type stmt_nonterminal1 is new stmt_nonterminal with record
   IF_part : Parseable_Token_Ptr;
end record;

type stmt_nonterminal2 is new stmt_nonterminal with record
   WHILE_part : Parseable_Token_Ptr;
end record;

type stmt_nonterminal3 is new stmt_nonterminal with record
   OPEN_part : Parseable_Token_Ptr;
end record;

type stmt_nonterminal4 is new stmt_nonterminal with record
   CLOSE_part : Parseable_Token_Ptr;
end record;
```

Figure 4 – Parse Tree Code Generated by AdaGOOP

An abstract type has been automatically generated for the **stmt** non-terminal as shown on the first line. Each of the tokens is represented by automatically building a type which inherits from the **stmt** non-terminal with a single attribute that represents the specific token. For example lines three through five of Figure 4 show the code generated for the

IF token. The only attribute of this type is **IF_part**, which holds a parsable token corresponding to the **IF** token.

4 Extending AdaGOOP

In order to automatically generate the visitor code, the AdaGOOP tool was extended to produce the classes and methods that implement tree traversal operations. Within the grammar that specifies the language, the user can now specify names of tree traversal operations. When AdaGOOP processes this input grammar, it generates the code to implement the operations based on the visitor design pattern using a hierarchy similar to the one shown in Figure 2. The only difference is that AdaGOOP does not implement the abstract superclass over the tree traversal operation classes.

```
visitors
PrettyPrint
TypeCheck
```

Figure 5 – Visitor Tree Traversal Operation Names

For example, Figure 5 shows the section of the grammar that defines the names of the visitors that will be automatically generated by AdaGOOP. In Figure 5, PrettyPrint and TypeCheck are names of the tree traversal operations that the user would like AdaGOOP to generate. A class is built for each of these operations and methods are included for each of the objects in the parse tree.

More specifically, AdaGOOP generates an Ada package for each of the tree traversal operations (PrettyPrint and TypeCheck). Within each package, AdaGOOP generates the methods that implement the operation by using the name of the operation supplied by the user. This name is overloaded and used for all the objects in the parse tree.

PrettyPrint_package	TypeCheck_package
PrettyPrint(stmt_nonterminal1) PrettyPrint(stmt_nonterminal2) PrettyPrint(stmt_nonterminal3) PrettyPrint(stmt_nonterminal4)	TypeCheck(stmt_nonterminal1) TypeCheck(stmt_nonterminal2) TypeCheck(stmt_nonterminal3) TypeCheck(stmt_nonterminal4)

Figure 6 – AdaGOOP generated classes and methods

For example, Figure 6 shows the classes generated by AdaGOOP given the operation names shown in Figure 5. The package **PrettyPrint_package** defines the class for the pretty printing tree traversal operation and includes methods for each of the objects in the parse tree. Referring back to Figure 4, you can see that **stmt_nonterminal1** is the type used to represent the **IF** token, **stmt_nonterminal2** is the type used to refer to the

WHILE token, and so on. Note that there is no abstract super class defined for the tree traversal operation classes, ie there is no super class over **PrettyPrint_package** or **TypeCheck_package**. This choice of implementation for AdaGOOP was made in order to simplify the code generated for the tree traversal operations.

```
procedure PrettyPrint(This : in out stmt_nonterminal1) is
begin
   PrettyPrint_Classwide(This => This.IF_part.all);
end PrettyPrint;

procedure PrettyPrint(This : in out stmt_nonterminal2) is
begin
   PrettyPrint_Classwide(This => This.WHILE_part.all);
end PrettyPrint;

procedure PrettyPrint(This : in out stmt_nonterminal3) is
begin
   PrettyPrint_Classwide(This => This.OPEN_part.all);
end PrettyPrint;

procedure PrettyPrint(This : in out stmt_nonterminal4) is
begin
   PrettyPrint_Classwide(This => This.CLOSE_part.all);
end PrettyPrint;
```

Figure 7 – Ada Code Generated for PrettyPrint Visitor

Figure 7 shows the Ada code generated by AdaGOOP for the **PrettyPrint** visitor. A **PrettyPrint** procedure has been generated for each of the four tokens under the **stmt** non-terminal.

One of the problems with implementing the tree traversal operations outside the parse tree node class is that the method for pretty printing is now not part of the node's class. Thus, as implemented in Ada, there is no way to automatically dispatch objects to the proper **PrettyPrint** procedure since they are no longer part of the node's class. To solve this problem, classwide **PrettyPrint** operations are generated by AdaGOOP.

```
procedure PrettyPrint_Classwide(This : in out stmt_nonterminal'Class) is
begin
   if This in stmt_nonterminal1'Class then
      PrettyPrint(This => stmt_nonterminal1(This));
   elsif This in stmt_nonterminal2'Class then
      PrettyPrint(This => stmt_nonterminal2(This));
   elsif This in stmt_nonterminal3'Class then
      PrettyPrint(This => stmt_nonterminal3(This));
   elsif This in stmt_nonterminal4'Class then
      PrettyPrint(This => stmt_nonterminal4(This));
   else
      raise Constraint_Error;
   end if;
end PrettyPrint_Classwide;
```

Figure 8 – Classwide Operation to Dispatch

Figure 8 shows the classwide procedure that is used to dispatch the object to the proper operation. For example, in Figure 8, the procedure **PrettyPrint_Classwide** takes a classwide argument representing the **stmt_nonterminal** type and its sub-classes. Any of the statements **IF**, **WHILE**, **OPEN**, or **CLOSE** can be passed to the **PrettyPrint_Classwide** procedure and dispatched properly. If the input argument is

stmt_nonterminal1, an **IF**, then the **PrettyPrint** procedure for **stmt_nonterminal1** is called to process the **IF**. This manual method of dispatching objects is needed since the tree traversal operations are now implemented outside of the parse tree node classes instead of implemented as methods in the node classes.

5 Conclusions

By extending the AdaGOOP tool, code for implementing the Visitor design pattern is now generated for each tree traversal operation specified by the user in the input grammar. A package is generated for each operation and the methods required to implement the operation are generated and included in the package. Classwide procedures are also generated to handle object dispatching.

By generating the code for the tree traversal operations, many hours of tedious monotonous coding is prevented. For example, in creating a tool to parse MSIL for an Ada implementation for .NET [CHS02], a 1,174 non-blank line input file to AdaGOOP automatically generated a 6,781 non-blank line visitor package body, as well as over 8,000 lines of other code for automatically generating the parse tree. Less than 1,000 lines of user-written Ada 95 code were necessary to complete the tool. Additionally, the chance of introducing an error in the tree traversal operations is reduced to zero. By using the visitor pattern, the code produced is easier to maintain, change, and extend. The extension to AdaGOOP improves the code generated for a specific grammar adding automatically generated code for tree traversal operations.

References

[Car00] Carlisle, Martin C. "An Automatic Object-Oriented Parser Generator for Ada", *Ada Letters*, Vol XX, Number 2, June 2000.

[CHS02] Carlisle, Martin C., Ricky E. Sward and Jeffrey W. Humphries. "Weaving Ada95 into the .NET Environment", submitted to SIGAda 2002.

[GHJV95] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, c1995, Addison-Wesley, Reading, MA.