Lund University
Computer Science
Jesper Öqvist, Niklas Fors, Görel Hedin

Compilers
EDAN65
2017-09-14

Programming Assignment 3
Visitors, Aspects, and Attribute Grammars

This assignment will give you experience in using two techniques for imperative computations in a compiler: *static Aspect Oriented Programming* (static AOP) and the *Visitor Pattern.* This assignment will also introduce *Attribute grammars.*

To understand visitors and static AOP, there is a new demonstration example, CalcComp. You will then extend your SimpliC implementation from assignment 2 with a visitor-based computation of *Maximal Statement Nesting*, and static AOP-based computations of *pretty printing* and *name analysis.* As an introduction to attribute grammars, you will work on a demonstration example called MinTree.

Try to solve all parts of this assignment before going to the lab session. If you get stuck, use the Piazza forum to ask for help. If this does not help, you will have to ask at the lab session. Make notes about answers to questions in the tasks, and about things you would like to discuss at the lab session.

▶ Major tasks are marked with a black triangle, like this.

1 The CalcComp Demo

The CalcComp demo project shows how to implement a simple code metric using visitors, and simple error checking and pretty-printing using static AOP.

1.1 Visitors

In CalcComp a visitor is used to check if there are any interactive statements in a Calc program. The visitor traverses the AST looking for `ask` statements.

Visitor framework code for the Calc language includes

- an interface `Visitor`, see `Visitor.jrag` in `src/jastadd`
- an `accept` method for each concrete AST class, also in `Visitor.jrag`
- a class `TraversingVisitor` implementing the `Visitor` interface, see `TraversingVisitor.java` in `src/java/lang`.

This boilerplate code could in principle have been generated automatically by JastAdd from the abstract grammar. However, for this assignment, we think it is a good idea to write it manually, so you get a better understanding of how visitors work.

▶ Download the CalcComp example and study the visitor framework code. Make sure you understand how it works. Try to answer the following questions:

- In what situation might the `accept` method in `ASTNode` be useful?
- Why is there not an `accept` method for the `Expr` class?
- What problem would you run into if you could not use aspects to add the `accept` methods?

The actual visitor that implements the interactive statement checking is located in the file `CheckInteractive-Visitor.java` in the directory `src/java/lang`.

▶ Study the code for `CheckInteractiveVisitor`. Try to answer the following questions.

- Suppose you have a main program with a `Program` AST. How would you use the visitor to find out if the program contains any `Ask` statement?

- What other ways could you solve the problem (without using a visitor)?
- Will the traversal go into subtrees of an `Ask` statement?
- Suppose you wanted to count the number of `Ask` statements instead. How would you change the visitor?

1.2 Pretty printing using aspects

Instead of using visitors, it is usually easier to use JastAdd's static aspects. We will now look at how *pretty printing* (also known as *unparsing*) is implemented in CalcComp using aspects.

Pretty printing is the programmatic printing of an AST back to its corresponding textual form. This can be useful in several situations, for example when printing error messages, or when debugging a compiler.

The file `src/jastadd/PrettyPrint.jrag` contains the `PrettyPrint` aspect. This aspect adds methods for pretty printing Calc programs. A few of the declarations are listed below:

```
aspect PrettyPrint {
  public void ASTNode.prettyPrint(PrintStream out) {
    prettyPrint(out, "");
    out.println();
  }

  public void ASTNode.prettyPrint(PrintStream out, String ind) {
    for (int i = 0; i < getNumChild(); ++i) {
      getChild(i).prettyPrint(out, ind);
    }
  }

  public void Mul.prettyPrint(PrintStream out, String ind) {
    getLeft().prettyPrint(out, ind);
    out.print(" * ");
    getRight().prettyPrint(out, ind);
  }
```

The `ASTNode.prettyPrint(PrintStream)` method provides the method that the user calls to pretty print a tree, this in turn calls the two-argument version where the second argument gives the current indentation prefix string. Each node type will require its own two-argument pretty printing method.

The `PrintStream` argument to `ASTNode.prettyPrint` works just like the `System.out` object, in fact `System.out` has the same type. By printing to the given `PrintStream` we have some flexibility for where the output should be printed, and unlike a `StringBuilder`, the `PrintStream` does not have to buffer the entire output.

▶ Study the pretty printing code, and make sure you understand how it works. Try to answer the following questions:

- Suppose you have a main program with a `Program` AST. What would a call look like that prettyprints it to standard output? (Hint: take a look in the main program `Compiler.java` in CalcComp.)

1.3 Name analysis and error checking

Name analysis is added through the `NameAnalysis` aspect. The name analysis in the CalcComp demo is only used for checking multiple declarations or undeclared variables. In the Calc language it is not allowed to use a variable before it has been declared, and one variable may not be re-declared in the same `let` statement:

```
let
    a = b
    b = 3.0
in
    a
end
```

The above example is not valid because the name `b` is used before it has been declared.

It is possible to "shadow" a previous declaration if the new declaration is inside another `let` statement. For example, the following is valid because the second declaration of `b` is inside another `let` statement:

```
let
    a = 1.0
    b = a
in
    let
        b = 2.0
    in
        b
    end
end
```

The name analysis implementation in the CalcComp demo uses a symbol table. This is a simple stack-based data structure that helps in tracking symbols within nested scopes. Each time the name analysis enters a new semantic scope, such as a let statement, a new `SymbolTable` instance is pushed on top of the stack.

The symbol table class is declared in the `NameAnalysis` aspect:

```java
public class SymbolTable {
  private static final SymbolTable BOTTOM = [...];// null-object
  private final SymbolTable tail;
  private final Set<String> names = new HashSet<String>();

  public SymbolTable() {
    tail = BOTTOM;
  }

  public SymbolTable(SymbolTable tail) {
    this.tail = tail;
  }

  /**
   * Attempt to add a new name to the symbol table.
   * @return true if name was not already declared
   */
  public boolean declare(String name) {
    return names.add(name);
  }

  /**
   * @return true if name has been declared
   */
  public boolean lookup(String name) {
    return names.contains(name) || tail.lookup(name);
  }

  /**
   * Push a new table on the stack.
   * @return the new top of the stack
   */
  public SymbolTable push() {
    return new SymbolTable(this);
  }
}
```

The name analysis is implemented in the methods `checkNames` that administrate the symbol tables and check for multiple and missing declarations, printing possible errors to a `PrintStream`.

▶ Study the code for the symbol table and the name analysis in the `NameAnalysis.jrag` aspect. Try to answer the following questions.

- The symbol table is simpler than a traditional symbol table in that it only keeps track of names and not the binding (declaration) of a name. This would be needed to support, for example, type checking. How would you modify the `SymbolTable` class to support name binding?
- Suppose you have a main program with a `Program` AST. What would a call look like that performs name analysis and prints the errors to standard error? (Hint: google *standard streams java* if you don't know what standard error is.)
- How does the push operation work in `SymbolTable`? Why is there no pop operation?

2 SimpliC

You will now extend the compiler for SimpliC that you implemented in Assignment 2 with some analyses using visitors and aspects.

▶ Make sure that your code from Assignment 2 is in a consistent state that builds and tests correctly.

2.1 Maximal Statement Nesting for SimpliC

Maximal Statement Nesting (MSN) is a simple metric which computes the maximum nesting depth of statements in a program. If there are no nested statements then MSN=1, if there is one statement nested inside another then MSN=2 etc. The MSN of the example below is 3:

```
int main() {
  // depth = 1
  int i = 100;
  while (i > 0) {
    // depth = 2
    if (i == 5) {
      // depth = 3
      print(i);
    }
    i = i - 1;
  }
  if (i == 0) {
    // depth = 2
    print(i);
  }
  // depth = 1
  return 0;
}
```

▶ Implement the visitor framework for SimpliC, including a `TraversingVisitor` class.

▶ Think about how you could implement MSN analysis for SimpliC. Think first how you would solve it if you added methods to the AST classes. Then think about how you would solve it using a visitor. Try to answer the following questions.

- What are possible strategies for implementing the MSN analysis? Should you use state variables inside the visitor or the data parameter? What are the pros and cons of these approaches?
- Suppose the visitor framework had type parameters for the return value and the data parameter. What would the advantage be?

▶ Implement a visitor to compute the MSN of a SimpliC program, including automated tests with examples where MSN is 1, 2, and 3, respectively.

2.2 Pretty printing for SimpliC

▶ Implement pretty printing of SimpliC programs using static aspects. Your code should be structured similarly to the pretty printing in the CalcComp demo (see section 1.2), and include automated tests. For syntactically correct programs, the test input should be the same as the expected output.

2.3 Name analysis and error checking for SimpliC

You should implement name analysis and error checking of two basic types of errors:

- Check and report errors for multiply declared functions and variables
- Check and report errors for uses of undeclared functions and variables

The declaration order is important. Functions and variables must be declared before they are used. For example:

```
int a() {
  return b(); // error: b is not yet declared
}

int b() {
  return 1;
}
```

Declare-before-use for variables:

```
int main() {
  int a;
  a = 2 + b; // error: b is not yet declared
  int b = 3;
  return a;
}
```

The variable names are allowed to be shadowed, if declared inside another statement:

```
int main() {
  int a = 3;
  if (a != 0) {
    int a = 4; // okay: not in same block
    return a;
  }
  int a = 5; // error: redeclaration
  return a;
}
```

Function parameters should of course also be considered in the name analysis:

```
int main(int a) {
  return a; // okay: a refers to the parameter
}
```

The name errors are checked by implementing a simple form of name analysis. You will not need to bind variable uses to their declarations so you can use a very simple symbol table as in the CalcComp demo.

When implementing the variable and function name check it is useful to have `IdDecl` and `IdUse` AST nodes used for *both* variable and function declarations/uses. The declaration checking is then symmetric for variables and functions. `IdDecl` and `IdUse` are used in the CalcComp demo, although in the CalcComp case there are no functions.

▶ Think through how you are going to implement name analysis and error checking. Try to answer the following questions.

- In what circumstances should you be able to re-declare, or shadow, a variable name?
- In your opinion, should it be possible to shadow function parameters? Why, or why not?
- The examples provided above are good test cases. What other important test cases can you think of? Consider multiple parameters, if-then-else, etc.

▶ Implement name analysis and the basic error checking described above, including good automated tests.

A note on line and column numbers

The scanner can set the line and column number of each scanned symbol, by using the metavariables `yyline` and `yycolumn`. See `scanner.jflex` in one of the example projects. During parsing, the line and column information is carried over to the AST nodes, and can be accessed by the calls `getLine(getStart())` and `getColumn(getStart())`. However, Beaver only sets these values for the returned production of a rule. So if one of your rules creates more than one node, the line and column numbers will work correctly only for the returned node. Therefore, if you get the value `0` for rows and columns, try refactoring your Beaver rules so that each new node is returned by a separate beaver rule.

3 MinTree - Introduction to Attribute Grammars

You will now get an introduction to Attribute Grammars, using an example called MinTree. Next week's assignment will go deeper into attribute grammars, and apply them for semantic analysis of SimpliC.

MinTree is a small language that describes trees of numbers, and your task is to compute the minimum number in the tree, using attribute grammars. The MinTree project is provided on the course web page. The project contains an abstract grammar, pretty printing, a main class, and test cases.

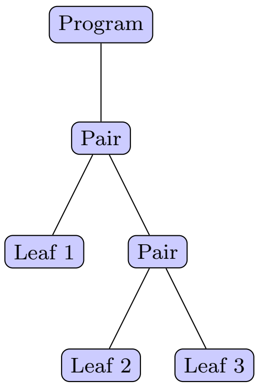
The abstract grammar of MinTree is defined as follows.

```
Program ::= Node;
abstract Node;
Pair : Node ::= Left:Node Right:Node;
Leaf : Node ::= <Number:int>;
```

The class `Program` represents the root node of the AST and the abstract class `Node` with two concrete subclasses, `Pair` and `Leaf`, models the recursive tree structure. Using this grammar, we can create an AST, without any parser, as follows.

```
new Program(new Pair(new Leaf(1), new Pair(new Leaf(2), new Leaf(3))))
```

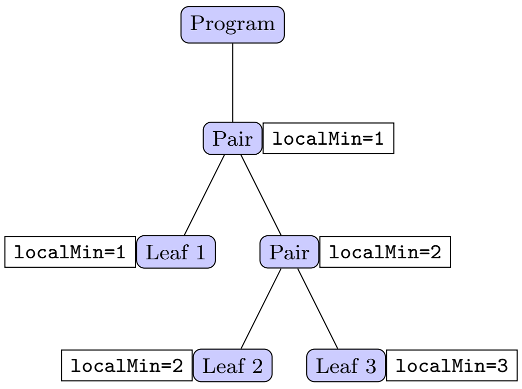
This expression can be visualized as follows.



▶ Download the MinTree project. Build the project and run the test cases (`./gradlew test`). All test cases should fail at this point. Also, build and run the main `lang.Compiler` program that prints out the tree above in a textual format. To build it as a Jar file, use `./gradlew jar`. To run the Jar file, type `java -jar compiler.jar`.

3.1 The localMin attribute

Attributes are *computed properties* of AST nodes that are defined by *equations*. How can we define the minimum value of the AST using attributes? Recall that `Node` is the superclass of `Leaf` and `Pair`. Let's introduce an attribute `localMin` for `Node` that is the minimum value of the `Node`'s subtree. I.e., we would like the following attributed tree:



We can see that the value of `localMin` for `Pair` nodes is the minimum value of its children's `localMin`.

We can declare `localMin` as an attribute on the abstract class `Node`, so both subclasses `Pair` and `Leaf` get the attribute:

```
syn int Node.localMin();
```

This is a **syn**thesized attribute with the type `int` on class `Node` and with the name `localMin`. The attribute has been declared as a synthesized attribute, which means the equation must be located in the same `Node` object (not in an ancestor). We can provide one equation in class `Pair` and another in class `Leaf` to take care of the two kinds of `Node`s. The equations can use tokens and attributes of the node and its children.

An equation for the class `Leaf` can be specified as follows.

```
eq Leaf.localMin() = ...;
```

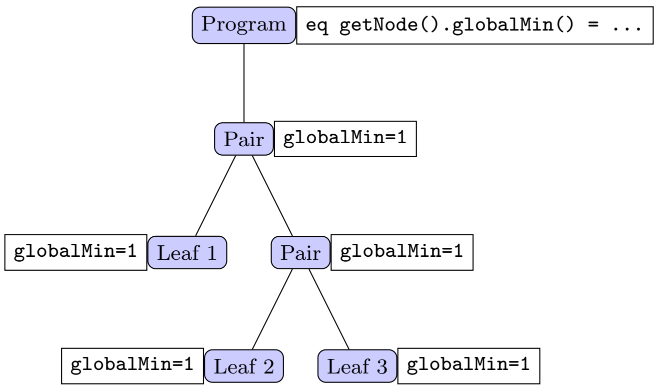
Here, the right-hand side of the equation is an expression. This kind of equation is called *expression-style equation*. For more complicated equations, the equation can be specified as a block. The above equation is equivalent to the following *block-style equation*.

```
eq Leaf.localMin() {
  ...
  return ...;
}
```

▶ Define the attribute `localMin` for the classes `Leaf` and `Pair` by adding two equations in the aspect file `MinValue.jrag`. Use the expression-style equation for the class `Leaf` and block-style equation for the class `Pair`. Verify that the test case for `localMin` passes.

3.2 The globalMin attribute

We have now synthesized the information upwards in the tree. We will now let all nodes know what number is the mininum number. We can do this by using an *inherited* attribute, that is, an attribute whose equation is defined by an ancestor node in the AST. This is illustrated in the following figure.



Here, the class `Node`, and thus `Pair` and `Leaf`, has an inherited attribute `globalMin`, but the *equation* is defined by the `Program` node, that is, an ancestor in the AST. An **inh**erited attribute can be declared as follows.

```
inh int Node.globalMin();
```

And the equation is specified as follows.

```
eq Program.getNode().globalMin() = ...;
```

Note that we specify the child (`getNode`) for which the equation is valid, according to the names in the abstract grammar. The equation above states that the value of the attribute `globalMin` is `...` for the child `Node`. The code `...` executes in the context of `Program`, and can access attributes, tokens, and children in `Program`.

The equation actually applies not only to `getNode`, but to *all* nodes *in the whole subtree* of `getNode` that happen to have a `globalMin` attribute. This is a mechanism called *broadcasting*, that is very useful in order to avoid having to define a lot of so called *copy* attributes that simply copy a value from a parent to a child.

If we want the equation to be valid not just for the subtree of a particular child, but for *all* children, we can specify `getChild` instead of `getNode`, which in this case does not matter since `Program` has only one child.

▶ The `MinValue.jrag` aspect contains an equation for `globalMin` that simply defines it to be 0. Change the equation so that the correct value is computed. Verify that the corresponding test case passes.

▶ There is an attribute `isMinValue` on `Leaf` that is supposed to tell if the node is the minimum number. Change the equation for the attribute to compute the correct value. Verify that the test case for `isMinValue` passes.

▶ The numbers in the tree do not need to be unique. This means that several `Leaf` nodes can have the same number. We will now compute how many `Leaf` nodes that have the minimum number. Define the attribute `nbrOfMinValues` for both the class `Program` (computing the global value) and for `Node` (computing the local value for that subtree). Verify that all test cases pass. Also, add a new test case where the value of `nbrOfMinValues` is larger than 1.

3.3 Side-effect free equations

For attribute grammars to work, the equations must be free from externally observable side effects, i.e., effects that other equations can see. This means that computing the equation several times should yield the same result. For example, the equation code must never change global variables that other equations might use (or call methods that do so). Note, however, that it is fine for an equation block to introduce local variables and assign to them, because these variables are not accessible from outside the block—the side effects on those variables are not externally observable.

Can I add print statements to the equation code? Yes, you can do that, but only for debugging, e.g., to see if an attribute is evaluated or not. You cannot use it for production code. The reason is that you do not control the order in which equations are executed—this is decided by an attribute evaluation engine. If you want to print things as part of the compilation, say, to prettyprint an AST, you will instead use normal methods, and which may access attributes in order to easily print computed information, such as type-checking errors.

3.4 Using attributes from methods

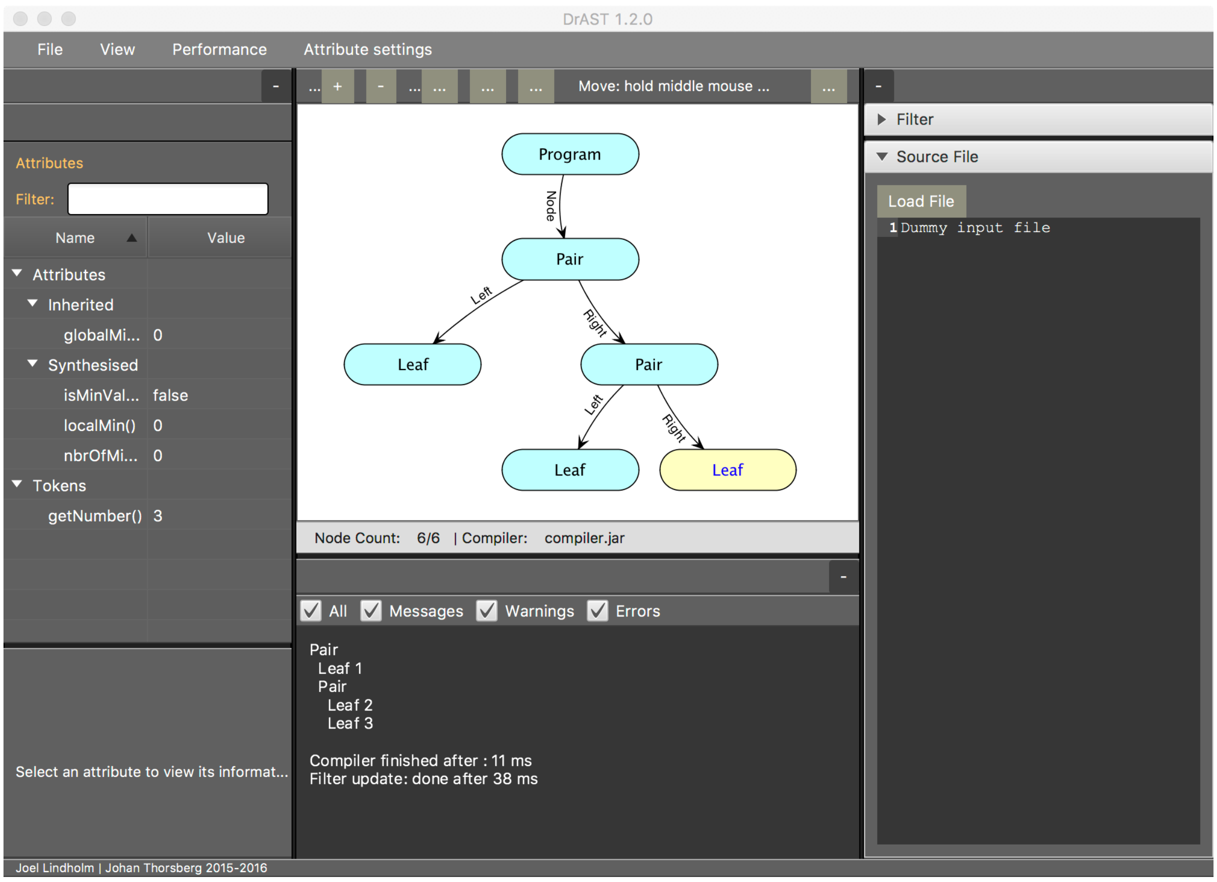
▶ There is an aspect `PrettyPrint.jadd` with methods for prettyprinting a MinTree, and that is called from the `lang.Compiler` class. Change the prettyprinter to print `*** MINIMUM ***` for each leaf node that has the minimum value. Check that it works by running the main `lang.Compiler` program again.

3.5 Inspecting the AST interactively with DrAST

DrAST is a tool that can be used for interactively inspecting a JastAdd AST with attributes. The tool has been developed as a master's thesis project. While it is still in an experimental stage, it can be very useful for inspecting an attributed AST. The MinTree compiler has been prepared for use with DrAST.

▶ Try out DrAST by using the following steps:

- Download `DrAST-1.2.1.jar` (or later)[1] from `https://bitbucket.org/jastadd/drast/downloads`. Place it, for example, in the MinTree directory.
- Go to the MinTree directory. Start DrAST by the command
  `java -jar DrAST-1.2.1.jar`
  (Or adjust the command, depending on where you placed the jar file.)
- A dialog box will appear where you fill in the path to the compiler jar: `compiler.jar`, and the name of the file to compile as the first compiler argument. The MinTree compiler doesn't read in any files, but fill in the name of an existing text file anyway, e.g., `testfiles/dummy.in`, as DrAST currently assumes the compiler always reads in a file. Click Open.
- You can now click on nodes and see their attribute values to the left.
- The view should be something like the figure below.
- Attribute values are computed on demand, so right-click[2] to see the value of an attribute.
- You may also configure the system to compute all attributes by the menu command `Attribute settings -> Compute all attributes`.
- DrAST will leave a configuration file, `DrAST.cfg` in the MinTree directory.
- If you would like to, you can also try out DrAST on your own SimpliC compiler. Make sure that the main program includes a static variable called `DrAST_root_node`, and which is set to the AST root. See `MinTree/src/java/lang/Compiler.java` for an example. When you run DrAST, you need to enter one of the test programs as the first compiler argument in the initial dialog. Even if the project has no attributes defined, it can still be useful to use DrAST to look at the AST itself.
- For more information on DrAST, see `https://bitbucket.org/jastadd/drast`.



Name	Value
Attributes	
Inherited	
globalMi...	0
Synthesised	
isMinVal...	false
localMin()	0
nbrOfMi...	0
Tokens	
getNumber()	3

Select an attribute to view its informat...

Node Count: 6/6 | Compiler: compiler.jar

```
Pair
 Leaf 1
 Pair
  Leaf 2
  Leaf 3

Compiler finished after : 11 ms
Filter update: done after 38 ms
```

```
Dummy input file
```

Joel Lindholm | Johan Thorsberg 2015-2016

[1]These instructions have been checked with DrAST-1.2.1.
[2]or double tap if you are using a Mac touch pad