Lund University
Computer Science
Jesper Öqvist

Compiler Construction
EDA180
2014-09-17

Programming Assignment 3
Imperative Computations

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.*

You will use visitors to implement a simple analysis called *Maximal Statement Nesting* for the SimpliC language. In addition, you will use static AOP to implement *pretty printing* and *name analysis* for SimpliC.

There is a new example project, CalcComp, that demonstrates the visitor pattern and aspect programming for the Calc language from the previous assignment.

▶ Tasks that you need to do in advance are marked with a black triangle, like this.

▷ Additional tasks that need to be completed are marked with a hollow 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 `CheckInteractiveVisitor.` 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 to the top of the stack.

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

```
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 for MSN=$\{1, 2, 3\}$.

2.2 Pretty printing for SimpliC

▷ Implement pretty printing of SimpliC programs using Aspect Oriented Programming. 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.