AST & Symbol Table

Generating the abstract syntax tree was generally a mechanical process. I added decorations to all of the productions which were needed for semantic analysis. I used conditional node descriptors for `#ArgList` with the boolean expression `>1` so that it would only appear in the AST if it had more than one child. The purpose of the AST is to abstract away redundant information so this was a necessary alteration.

I defined a Java symbol table class which was comprised of three Hashtables. The first one maps scope keys to a linked list of all the variables defined in that scope. The second one maps a key, which is the concatenation of id and scope, to a data type. The third one maps the same key to a qualifier (var, const, function, function param). The symbol table is initialised with a global scope mapped to an empty linked list.

The symbol table class provides six methods: put, print, checkforDups, inScope, typeLookup, and qualifierLookup. The put method inserts an entry into the symbol table, the print method outputs the symbol table to standard output, checkForDups returns true if a variable is defined more than once in a scope, inScope returns true if a supplied variable id is in the supplied scope, typeLookup returns the data type associated with the supplied id and scope, and qualifierLookup returns the qualifier associated with the supplied id and scope.

The symbol table gets built at the same time as the AST. Everywhere that `Identifier()` is called in my `.jjt` file, it is followed by a call to `SymbolTable.put(...)`.

```
void VarDecl() #VarDecl : {Token t; String id; String type;}
{
    t = <VAR> id = Identifier() <COLON> type = Type()
    { st.put(id, type, "var", scope); }
}
```

The scope is stored in a static variable in the jjtree file and is initialised to `"global"`. It is updated in a couple of places: in the `Func` production and in the `Main` production.

Semantic Analysis

Semantic analysis is carried out by the `SemanticCheckVisitor` class. It defines a number of static boolean variables which maintain the status of the semantic checks. The entry point to the class is the `Prog` node. The symbol table from the `jjtree` file is passed to the semantic check visitor as a paramater of the `Prog` visit method. The `Prog` node's `childrenAccept(...)` method is invoked which will make the Visitor visit each of the `Prog` node's children. The children node's then have their `childrenAccept(...)` methods invoked and this pattern continues until all the leaf nodes are reached. When the program returns to the `Prog` node it has finished its semantic analysis. Each of the

semantic check variables are then checked and will print out a `"PASS"` message if they have not been altered.

```
public Object visit(Assign node, Object data) {
    SimpleNode child1 = (SimpleNode) node.jjtGetChild(0);
    SimpleNode child2 = (SimpleNode) node.jjtGetChild(1);
    DataType child1DataType = (DataType) child1.jjtAccept(this, data);
    DataType child2DataType = (DataType) child2.jjtAccept(this, data);

    if (child1DataType != child2DataType) {
        correctVarAssign = false;
        System.out.println("Error: Var \"" + child1.value + "\" was assigned a val
ue of the wrong type.");
        System.out.println("\tWas expecting \"" + child1DataType + "\" but instead
 encountered \"" + child2DataType + "/"");
    }

    return Data.Assign;
}
```

The above example shows how the semantic check visitor checks if both sides of an assignment operator are of the same type. The assignment node has two children: the left and right hand side of the operator. The two children have their `jjtAccept(...)` methods invoked which return data types. These two values are compared and the appropriate actions are carried out if they are not the same.

IR Code Generation

IR code is generated by the `ThreeAddrCodeBuilder` class. It works in a similar way to the `SemanticCheckVisitor` by visiting each of the nodes recursively. The class provides two helper functions: one for printing labels and one for printing instructions. These help to ensure that the code is printed in the correct format.

There are also two static variables for keeping track of both the label and temporary variable count. These are incremented as the vaariables are used.

The handling of `while` and `if` statements is shown below:

```
public Object visit(Stm node, Object data) {
    String stm = (String) node.value;
    String beginLabel;
    String condition;
    String endLabel;
    switch (stm) {
        case "while":
            String conditionLabel = "L" + labelCount;
```

```
            labelCount++;
            beginLabel = "L" + labelCount;
            labelCount++;
            endLabel = "L" + labelCount;
            labelCount++;
            printLabel(conditionLabel);
            condition = (String) node.jjtGetChild(0).jjtAccept(this, data);
            printInstruction("if " + condition + " goto " + beginLabel);
            printInstruction("goto " + endLabel);
            printLabel(beginLabel);
            int children = node.jjtGetNumChildren();
            for (int i = 1; i < children; i++) {
                node.jjtGetChild(i).jjtAccept(this, data);
            }
            printInstruction("goto " + conditionLabel);
            printLabel(endLabel);
            return null;
        case "if":
            if (node.jjtGetNumChildren() > 2) {
                beginLabel = "L" + labelCount;
                labelCount++;
                String elseLabel = "L" + labelCount;
                labelCount++;
                endLabel = "L" + labelCount;
                labelCount++;
                condition = (String) node.jjtGetChild(0).jjtAccept(this, data);
                printInstruction("if " + condition + " goto " + beginLabel);
                printInstruction("goto " + elseLabel);
                printLabel(beginLabel);
                node.jjtGetChild(1).jjtAccept(this, data);
                printInstruction("goto " + endLabel);

                printLabel(elseLabel);
                node.jjtGetChild(2).jjtAccept(this, data);
                printLabel(endLabel);

            } else {
                beginLabel = "L" + labelCount;
                labelCount++;
                endLabel = "L" + labelCount;
                labelCount++;
                condition = (String) node.jjtGetChild(0).jjtAccept(this, data);
                printInstruction("if " + condition + " goto " + beginLabel);
                printInstruction("goto " + endLabel);
                printLabel(beginLabel);
                node.jjtGetChild(1).jjtAccept(this, data);

                printLabel(endLabel);
```

```
            }
            return null;
        default:
            return null;
    }
}
```

The `else` in the `if` case accounts for occasions where an `if` statement is followed by an `else` which only contains a `skip` instruction. If this was not included you would observe a NullPointerException as the program attempts to visit the second child.