1 Exercises

- For the following program: Draw the symbol table at the marked point.
- Write the function lookup in class Scope

```
import java.lang.*;

int x;

int f(int x) {
        int res;
        res = 1;
        while (x > 1) {
                res = res * x;
                /*** here ***/
                x = x - 1;
        }
        return res;
}
```



```
class Scope {
        Symbol first;
        Scope outer;
/** lookup a name in the current or an enclosing scope;
  * returns null, if no matching symbol is found
  */
        public Symbol lookup(String name) { ... }
        public void enter(Symbol sym) { ... }
}

class Symbol {
        int pos;
        String name;
        Symbol next;
        ...
}
```


2 Symbols in Jex

- In Jex unqualified names (names that do not follow a dot), can refer to different kind of things:
  - local variables
  - global variables
  - functions
  - java classes
- It is the purpose of the name analysis to find for every unqualified name the point of definition and set the sym - field in the tree correspondingly
- At the definition point we construct a JexSymbol object and put it into the symbol table.
- At a usage point we look it up in the symbol table.

3 Symbols in Jex(2)

We implement this by a class JexSymbol and 4 subclasses.

- LocalSym GlobalSym FunSym ClassSym

We use the class JexSymbol to implement functionality that is common to some symbols, but is specific for Jex and not in class Symbol.

- Example: **boolean** isVariable(), which asks whether the symbol is a LocalSym or GlobalSym.

The information we need to store in the symbols are

- The types of the defined names
- Run-time information (e.g.The number of local variables in a function)

4 Types and class Class

- Types in Jex are the classes or primitive types of Java.
  - **int**, **boolean**, String, ...
- So we can use the class Class from Java to store the type of a defined entity.
- Objects of class Class are run-time representations of Java classes.
- We can get the class of a java object o by o.getClass().
- We can get a class also by calling e.g. Class.classForName("java.lang.String").

5 JexSymbol

- LocalSym and GlobalSym have a field Class cls, which denotes the type of the variable.
- ClassSym has a field Class cls, which denotes the class itself.
- FunSym has a field Class resClass and a field Class[ ] argClasses, which denote return type and argument types respectively.
- The remaining fields are run-time information.
- The class JexSymbol also has some functions.

```
public boolean isVariable() { ... }
public boolean isFunction() { ... }
public boolean isClass() { ... }
```

- Later for interpretation we will add more functions.
- These functions where the reason of adding a class JexSymbol.


```
class JexSymbol extends Symbol {
    static public class FunSym extends JexSymbol {
        Class resClass;
        Class[ ] argClasses;
        Tree.FunDef tree;
        int localCount;
    }
    static public class LocalSym extends JexSymbol {
        Class cls;
        int adr;
    }
    static public class GlobalSym extends JexSymbol {
        Class cls;
        JexValue val;
    }
    static public class ClassSym extends JexSymbol {
        Class cls;
    }
}
```


6 Imports

- If a name does not refer to a defined identifier, it might refer to a Java class.
- Java has a static method Class Class.classForName(String) which yields an object representing the class.
- But, when looking for a class by name we also have to consider the imports.
- Imports are represented by a class Import

```
class Import {
    Import next;
    public Import(String[ ] names, boolean isStar) { ... }
    public Class classForName(String s) { ... }
}
```

- The next field makes a linear list, like for Symbol.

7 ImportScope

Similar to scope we keep all import directives in an import scope.

- For this we have a class ImportScope:

```
class ImportScope {
    Import first;
    public ImportScope() { ... }
    public void enter(Import imp) { ... }
    public Class lookup(String s) { ... }
}
```

- enter puts a new import directive into the import scope.
- lookup tries to find a class with name s, using all the import directives in the import scope.

8 Optimisation

- The current scheme uses a linear search for identifiers in symbol tables.
- In a production compiler this is far too slow.
- Better schemes:
  - additionally link entries as a binary tree and use that for searching
  - Use a hash table for each block
  - Use a global hash table (fastest)

9 A first Visitor for Name Analysis

- It has to construct the symbols.
- It has to attach the computed symbols to the tree.
- It has to enter the symbols into the scope.
- It has to lookup the identifiers in the scope.
- It has to attach the looked upd symbols to the tree.


```
public class Analyzer implements Tree.Visitor {
    Scope scope;              // current scope
    ImportScope imports;      // imports (global)
    boolean toplevel;         // are we on toplevel?

    // the main name analysis method
    public static void analyzeTree(Tree tree, Scope scope,
        ImportScope imports, boolean topLevel) {
        tree.apply(new Analyzer(scope, imports, toplevel));
    }

    // recursive analysis method
    protected void analyze(Tree tree, Scope scope, boolean topLevel) {
        ...
    }
```

```
public void caseFunDef(Tree.FunDef tree) {
    // the scope starting with the parameters
    Scope paramScope = new Scope(scope);

    // analyse formal arguments
    for (int i = 0; i < tree.formals.length; i ++)
        analyze(tree.formals[i], paramScope, false);

    // analyze result type
    analyze(tree.tp, scope, toplevel);

    // symbol for function
    tree.sym = new JexSymbol.FunSym(tree.pos, tree.name, ...);
    scope.enter(tree.sym);

    // analyze body
    analyze(tree.stat, paramScope, false);
}
```



```
public void caseFunCall(Tree.FunCall tree) {
    // analyze arguments
    for (int i = 0; i < tree.args.length; i ++)
        analyze(tree.args[i], scope, toplevel);

    // get function symbol
    tree.sym = (JexSymbol) scope.lookup(tree.name);
    if (sym == null)
        Report.error(tree.pos, "function " + tree.name + " undefined");
}

public void caseDefList(Tree.DefList tree) {
    for (int i = 0; i < tree.defs.length; i ++)
        analyze(tree.defs[i], scope, true);
}
```


10 Making it a bit more convenient

- We often call analyze with similar arguments.
- We define additional recursive analyze methods with fewer arguments.

```
    // recursive analysis method
    protected void analyze(Tree tree) {
        analyze(tree, scope, toplevel);
    }

    // recursive analysis method
    protected void analyze(Tree tree, Scope scope) {
        analyze(tree, scope, false);
    }
```