Prof. Jingke Li (FAB120-06, lij@pdx.edu), Mon 8:15-10:05 @FAB 171, Lab: Wed 8:15-10:05 @FAB 88-10

Assignment 4: Static Analysis

(Due 3/11/15 @ 11:59pm – Firm Deadline!)

This assignment explores static analysis. Specifically, it asks you to implement a type-checker for the miniJava language and two other checks, one for detecting missing return statement and one for detecting uninitialized variables. You will implement these tasks by writing a single program, to be called `Checker.java`. The program reads in an miniJava AST tree, and traverses the tree to perform the type-checking and the analysis tasks. The assignment carries a total of 125 points.

The assignment assumes familiarity with the OO tree-traversal techniques discussed in Labs 7 and 8, and is intended to be attempted after you have completed those labs.

Preparation

Download the zip file "`hw4.zip`" from the D2L website. After unzipping, you should see an `assignment3` directory with the following items:

- `hw4.pdf` — this document
- `mjManual.pdf` — the miniJava language manual
- `Checker0.java` — a starting version of the checking and analysis program
- `ast` — a directory containing the AST definition program file, `Ast.java` (the same as in hw3), and a set of Java programs for an AST parser
- `tst` — a directory containing sample miniJava programs programs in both `.java` and `.ast` forms (the same as in hw3)
- `tst2` — a directory containing miniJava programs with type errors and programs for testing the two analyses
- `Makefile` — for building the parser
- `run` — a script for running tests

Copy `Checker0.java` to `Checker.java` and add code to it to complete this assignment.

Program Structure

Conceptually, the type-checker and the return-statement analysis are to be implemented as computations over miniJava AST trees. But instead of inserting local computation routines into individual AST node classes as we have seen in Labs 7 and 8, we take a slightly different approach in this assignment. You'll still write code for individual AST nodes, but instead of modifying the `Ast.java` program file, all your code will be placed in a separate file, `Checker.java`. With an AST tree read in from input via the provided `astParser`, a couple of dynamic dispatch routines will send the execution to the proper individual routines to perform type-checking and return-statement analysis. Take a look inside the program `Checker0.java` to see more details of the program structure.

Environment Management

As we have seen in Lab 8, environments are a key part of many computations over ASTs. They are used for mapping variables to their types or values, functions to their definitions, and so forth. They provide symbol-related contexts for computations at individual AST nodes. For the tasks of this assignment, we also need environment support. For instance, to type-check expression `a+b`, we need to know variables `a` and `b`'s types; to type-check a `NewObj` node, we need to verify the corresponding class exists; and to type-check a method call, we need to find the method's definition, so that we can verify that the number and types of actual arguments match those of the formal parameters.

In miniJava, variable declarations may only appear at class level or at method level. In other words, there are no nested scopes in the form of statement blocks. This means that there are only three scope levels in a miniJava program. The *global scope*, in which classes are defined; a *class scope* for each class, in which fields and methods are defined; and a *method scope* for each method, in which local variables and parameters are declared.

Accordingly, we use the following environments in the checker program:

- A name-definition environment for classes. While a mapping from `String`s to `ClassDecl` AST nodes would work, it would not be very convenient for accessing a class's parent's information, which is frequently needed in type-checking. So we define a wrapper data structure to have a direct pointer from a class declaration to its parent's declaration:

  ```
  static class ClassInfo {
    Ast.ClassDecl cdecl;
    ClassInfo parent;
  }
  ```

  The class environment hence is defined to be a mapping from `String`s to `ClassInfo` nodes.

- A name-type environment for a method's local variables and parameters, *e.g.* a mapping from `String`s to `Type` AST nodes. Note that local variables and parameters of a method are in the same name space (*e.g.* a local variable and a parameter can't share the same name), hence they are represented in a single environment.

  Since methods' scopes are non-overlapping, we use a single static copy, `typeEnv`, for representing this environment. Your checker program should reset this environment to empty at the beginning of the check routine for the `MethodDecl` node, so each method will have a fresh new copy of the enrironment.

- For field variables and method declarations, we choose not to create separate mapping environments. Instead, we rely on the `ClassDecl` AST node's own `flds` (a `VarDecl[]`) and `mthds` (a `MethodDecl[]`) components. For instance, to find a field `x`'s type, we would (linearly) search the host `ClassDecl` node's `flds` component for a matching `VarDecl` and fetch the type information from it.

  A pair of utility routines, `findFieldDecl(String fname)` and `findMethodDecl(String mname)`, serve as the equivalence to environments' `get()` routine for fields and methods. Both are to be invoked from a `ClassInfo` object.

  The reason for this choice is because the name space for fields and the name space for methods are both complicated by the language's inheritance features (*e.g.* we have to include ancestor's fields, but not those that are overshadowed, etc.). So we chose simplicity over performance on this one.

Overall, the following are the static variables defined for environment support. The whole environment management portion of the type-checking program is provided to you in `Checker0.java`.

from Checker0.java

```
//------------------------------------------------------------------------
// Global Variables
// ----------------
// classEnv - an environment (a className-classInfo mapping) for class declarations
// typeEnv - an environment (a var-type mapping) for a method's params and local vars
// thisCInfo - points to the current class's ClassInfo
// thisMDecl - points to the current method's MethodDecl
//
private static HashMap<String, ClassInfo> classEnv = new HashMap<String, ClassInfo>();
private static HashMap<String, Ast.Type> typeEnv = new HashMap<String, Ast.Type>();
private static ClassInfo thisCInfo = null;
private static Ast.MethodDecl thisMDecl = null;
```

Task 1: Type-Checking (75 points)

The main task of this assignment is to implement a type-checker for the miniJava language. The type-checker traverses the input AST tree, and for each node in the tree, check its correctness with respect to miniJava's typing rules. Once a type error is detected, the checker program raises a `TypeException` and quits.

The miniJava language's typing rules follows that of Java's. Here are some highlights. For cases that are not covered here (shouldn't be many), consult miniJava's language manual and Java documents.

- Every class, method, or variable used in a program must be declared. For a variable, the declaration must appear before its uses.
- Variable initialization, assignment statement, and actual argument to formal parameter mapping must all follow type compatibility requirements. (See Type Compatibility section below.)
- The number of actual arguments in a method call must match that of the method's formal parameters. Every return object's type from a method must match the method's declared return type.
- The test of `If` and `While` statements must be boolean.
- The argument of `Print` statement must be integer, boolean, or String.
- The object with which a method or a field is accessed must be a class object.
- The object in an `ArrayElm` node must be an array object; and the index must be integer.
- Arithmetic operations must have integer operands; logical operations must have boolean operands; equality comparisons ("==" and "`!=`") must have operands with comparable types; other relational operations must have integer operands.

In the provided program, `Checker0.java`, type-checking hints are provided for all AST nodes.

Note that every check routine for an AST expression node needs to return the expression's type in the form of an `Ast.Type` object. For some nodes, this may involve analysis and searching. Also, it goes without saying the every check routine should recursively check its components by invoking their corresponding check routines.

Type Compatibility (10 points)

Java's type compatibility rules are expressed by the following two routines:

Type Compatibility Routines

```
// Returns true if tsrc is assignable to tdst.
//
boolean assignable(Ast.Type tdst, Ast.Type tsrc) {
```

```
  // if tdst==tsrc or both are the same basic type
  //   return true
  // else if both are ArrayType // structure equivalence
  //   return assignable result on their element types
  // else if both are ObjType   // name equivalence
  //   if (their class names match, or
  //       tdst's class name matches one of tsrc's ancestor's class name)
  //     return true
  // else
  //   return false
}

// Returns true if t1 and t2 can be compared with "==" or "!=".
//
private static boolean comparable(Ast.Type t1, Ast.Type t2) throws Exception {
  return assignable(t1,t2) || assignable(t2,t1);
}
```

As part of your type-checking task, you are asked to convert the pseudo code in the first routine to actual Java code. This part carries a separate 10 points of its own.

Task 2: Detecting Missing Return Statement (20 points)

For a method that has a non-void return type, a return statement is required within *every* possible execution path of it. The Java compiler, `javac`, performs a static analysis to detect and flag method that is missing a return statement. For instance, compiling the following program,

retn01.java

```
class Test {
  public int m() {
    // missing return statement
  }
  public static void main(String[] a) { }
}
```

`javac` would issue an error:

```
linux> javac retn01.java
retn01.java:4: error: missing return statement
  }
  ^
1 error
```

Like many other static analyses, this one calculates an approximation of the program's runtime behavior; that is, its prediction of the runtime behavior may be wrong in some cases. But the analysis only errs in one direction: it might flag some methods that really will always execute a return statement, but it never fail to flag a method that might actually not execute a return statement. As an example, for the following program,

retn02.java

```
class Test {
  public int m() {
    if (true)
      return 1;
    // missing return statement
  }
```

```
  public static void main(String[] a) { }
}
```

we can be certain that the return statement will always be executed, yet, `javac` will still flag a "`missing return statement`" error.

In this part, you will design and implement a return-statement analysis by adding code to the type-checking program you wrote for the first part. In your analysis, you may assume that all expressions' values are unknown. You may also assume that the input AST has already been type-checked, *i.e.* that it represents a type-correct program.

A set of test programs for this analysis are included in the `tst2` directory: `retn0[1-8].java`. You may want to use them to help you derive a strategy. You may also run `javac` on more programs to see its behavior on this analysis. (Note that in some cases, `javac` uses expressions' values to refine the analysis's result, which you don't need to do.) You are free to introduce new global or local variables for your need. (*Hint:* Focus on `If` and `While` nodes.)

Task 3: Detecting Uninitialized Variables (20 points)

Being a strongly-typed language, Java requires that every variable be initialized before being used. A static analysis is included in javac to catch uninitialized variables. For instance, the following program

init01.java

```
class Test {
  public void m() {
    int i = 1;
    int j;
    System.out.println(i + j);  // j not initialized
  }
  public static void main(String[] a) { }
}
```

will trigger an error:

```
linux> javac init01.java
init01.java:5: error: variable j might not have been initialized
   System.out.println(i + j);  // j not initialized
                      ^
1 error
```

Your task is to add this analysis to your `Checker.java` program. You may use the same assumptions as in the return-statement analysis. In fact, the implementation strategies for these two analyses are similar, although this analysis is a little more challenging. A set of test programs for this analysis are included in the `tst2` directory: `init0[1-6].java`.

Running and Testing

You should test your checker program with provided tests in both `tst` and `tst2` directories. The run-script `run` will invoke your program on an input AST program, and save the error message (if any) in the corresponding `.err` file.

All programs in the `tst` directory have been type-checked. You should expect to see the message "`passed static check`" when running your checker:

```
linux> ./run tst/test01.ast
tst/test01: passed static check
```

The programs, `typerr*.ast`, in the `tst2` directory each contains a type error. The runscript will compare your checker's error messages with reference messages in the `.err.ref` files.

Your error messages should match the corresponding reference ones in nature. However, they don't need to match character-by-character.

Requirements and Grading

This assignment will be graded mostly on your checker's correctness. We may use additional miniJava programs to test. The minimum requirement is that your program runs and detects at least one type error.

What to Turn in

Submit a single file, `Checker.java`, through the "Dropbox" on the D2L class website.