Project 1:
Interprocedural Exception Analysis

17-654/17-754: Analysis of Software Artifacts
Jonathan Aldrich (`jonathan.aldrich@cs.cmu.edu`)

Out: Thursday, February 10, 2005
Due: Thursday, February 24, 2005 (11:59 pm)

100 points total

The goals of this project are to write a real analysis based on traversing the abstract syntax tree of a program, to build a simple call graph, to put the iterative worklist analysis algorithm to work in practice, and to get a taste of interprocedural analysis.

Pairs. You may work on this programming project in pairs. Pair projects will be given a single grade. You are free to choose your own partner, subject to one constraint: the instructors reserve the right to assign pairs in the case that students lacking significant previous Java experience are unable to find a more experienced partner. Please be sensitive to this criterion as you pair up.

Collaboration Policy. It is permitted to discuss the homework problems in general terms, to study together, to help each other with notation, and to communicate clarifications from the instructor and TAs. It is *not* permitted to discuss specific answers to homework, or to look at another student's solution. Partners working in pairs should share the workload equally and ensure they are both familiar with all of the material. *This policy will apply to all future assignments (although not all future assignments will necessarily be done in pairs).*

Hand-in Instructions. Same as Project 0. Hand in all Java files that were added or modified for all parts of this assignment. Also hand in sample output from running your analysis on the input file `Project1Example.java` provided as part of this assignment. If you are working in pairs, include readme.txt file with the names of the two partners. Put all of the above files into a zip file and hand in via Blackboard.

1 Intra-Procedural Exception Analysis (30 points)

Java's type system tracks the exceptions that could be thrown by a method using `throws` clauses on its method declarations. This is important for engineers who are using a library, so they know what errors they might need to handle.

The problem is, Java only tracks so-called *checked exceptions* which are not subclasses of `RuntimeException`. Sometimes programmers get lazy and don't want to specify which exceptions are thrown by a function. Instead, they throw a `RuntimeException` so they don't have to do the specification work. Unfortunately, this tends to get them in trouble later: since there is no documentation about what exceptions a function might throw, the caller may not handle all exceptions appropriately, causing errors when the program is executed.

In this assignment, you'll build an analysis for tracking all exceptions, including `RuntimeException`s that aren't tracked by Java. First, you will build an intra-procedural version of the analysis. Intra-procedural means that the analysis is done one procedure at a time, and only relies on type information of called procedures. Since some procedures may throw `RuntimeException`s that aren't declared in their type, our analysis may miss some exceptions that might actually be thrown as a result of callees throwing runtime exceptions. We'll find a solution inter-procedural analysis, in section 3.

For section 1, write an intra-procedural exceptions analysis that computes, for each method, the set of exceptions that method may throw, including runtime exceptions. When you come to a method call, you should assume it might throw only the exceptions that it declares. Your analysis will be structured as a visitor over the abstract syntax tree, similar to what you did in Programming 0.

To get you started, we've provided an updated file `Project0Analysis.java`, a file `Util.java` with some utilities that you might find useful, and a file `MethodCollectionVisitor.java` that gathers up all the methods in a file and groups them by name. `MethodCollectionVisitor` is useful when you want to find out which methods might be called at a given call site in the program. Your solution should be implemented by editing `Proj0Analysis.java` in the appropriate places and adding other files as needed. The file `sample-output.txt` shows the format that your output should take for each part, on a sample input file `Project1Example.java`.

Hints:

- You should not use the `getMethodBinding` function in interface `IMethodCall` to find out who you are calling. This method returns only the static method called, not the entire set of methods that could be called dynamically. Additionally, it looks like there might be a bug in the current implementation. Use the `GetMethods` function of `MethodCollectionVisitor` instead–it returns a `java.util.Set` of `IMethod`s.
- Don't forget that the expression `catch(ExnType e) {...}` will catch expressions of type `ExnType` or any subclass of `ExnType`.
- You should assume that all exceptions are thrown explicitly; i.e., there is no need to guess when a `NullPointerException` might be thrown by the system when a null pointer is dereferenced.
- You only need to correctly handle the constructs in the example file we gave you. However, your implementation should be general enough to work if these same constructs are used in other ways in another file.
- The order in which results are reported does not matter.

2 Call Graph Analysis (30 points)

Write an analysis to construct a call graph of the program. Your analysis should output a pair [m1, m2] if method m1 may call method m2. In Part 3, you will be using this call graph backwards, so make sure your data structures allow you to query the data structure for the callers of a method as well as the callees of a method.

3 Inter-Procedural Exception Analysis (50 points)

In this part, you will extend the code written above to an inter-procedural data flow analysis for tracking which exceptions may be thrown by a function. Your data flow values will be sets of exceptions. Your data flow analysis will work over the interprocedural control flow graph (ICFG) which you have just constructed in part 2. However, since exceptions are thrown by a callee to the caller, exception data flow information flows in the reverse direction of calls in the ICFG, so you will have to reverse the edges computed in Part 2.

You will need to keep a data structure which stores for each method, what set of exceptions that method might throw. Use your knowledge of may analyses to determine how to initialize this data structure. Take the iterative worklist algorithm discussed in class (and in NNH 2.4) and adapt it to work over the ICFG. To make this work, think of the nodes in the ICFG as methods, and the transfer functions for the nodes are basically the intra-procedural analysis you wrote in Part 1. The only difference is that when you come to a method call you should use the current data flow information for the methods that might be called from that call site. Thus, your algorithm will be able to track `RuntimeException`s that are thrown across functions, even if they are not listed in Java's throws clauses.

Hints:

- Make a subclass of your Part 1 solution that differs only in what it does at method call sites.
- One thing is different about the method "flow functions" in this inter-procedural analysis, versus intra-procedural analyses we have looked at in class. The difference is that different inputs (exceptions thrown by different called methods) are treated differently depending on where the call is. For example, if you have calls to one method inside a try/catch block, and another one outside, you don't want to treat them the same way. Thus, you don't want to join the analysis information from called methods together and store that as the "input" for the method.
- While you can implement the worklist algorithm given in class given the right amount of care, because of the reason above (inputs are not all treated the same) it's cleaner to modify the worklist algorithm to store output values instead of input values. The idea is expressed in the psuedo-code below (which makes another minor modification, storing nodes instead of edges on the worklist):

$worklist = new\ Stack();$

$\forall \ell \in labels(S_*)\ do$
    $Analysis[\ell] = \bot$
    $worklist.push(\ell)$

$\forall \ell \in E\ do$
    $Analysis[\ell] = i$

$while(!worklist.isEmpty())\ do$
    $\ell = worklist.pop();$
    $if(f_\ell(Analysis) \not\sqsubseteq Analysis[\ell])\ then$
        $Analysis[\ell] = Analysis[\ell] \sqcup f_\ell(Analysis)$
        $worklist.pushAll(\{\ell' \mid (\ell, \ell') \in F\})$