COMP4500/7500
Advanced Algorithms and Data Structures

School of Information Technology and Electrical Engineering
The University of Queensland, Semester 2, 2014

Assignment 1

Due at 1:00pm, Monday 15 September 2014.
This assignment is worth 20% (COMP4500) or 15% (COMP7500) of your final grade.

Please read this entire handout before attempting any of the questions.

This assignment is to be attempted **individually**.

- Your solutions to Part A may be submitted in one of two ways:
  1. As a paper copy your solutions must be submitted via the EAIT submission service and must include a coversheet. More information on the cover sheet and submission service is available from: `https://student.eait.uq.edu.au/coversheets/`
  2. As an A4 **PDF** called "partA.pdf" via the course Blackboard site.
- Your solutions to the coding parts of Part B must be submitted via the course BlackBoard site as the files
  - Dependencies.java,
  - FlowGraph.java,
  - Primitive.java, and
  - Statement.java.

  No other files will be accepted. The files should be submitted individually, not as a zipped archive. You can submit multiple times before the assignment deadline and only the last copy submitted will be retained for marking.

Submitted work should be neat and legible – you may be penalised for untidy or illegible work. For the programming part, you will be penalised for submitting files that are not compatible with the assignment requirements, in particular, your solution should be compatible with the supplied testing framework.

Late submission. A penalty of 5% of the maximum mark for an assignment will be deducted for each day or part thereof late. As we plan to hand back assignments about 1 week after submission, requests for extension will not be accepted more than one week late. Requests for extensions should be directed to the course co-ordinator and should be accompanied by suitable documentation (see Section 6.1 of the course profile for details). Personal hardware or computer failures are not grounds for extension. Assignments must be backed up on the university system.

School Policy on Student Misconduct. You are required to read and understand the School Statement on Misconduct, available at the School's website at:

`http://www.itee.uq.edu.au/itee-student-misconduct-including-plagiarism`

Part A (35 marks total)

This assignment aims to test your understanding of graphs and graph algorithms.

Question 1: Constructing SNI and directed graph [5 marks total]

(a) (1 mark) **Creating your SNI.** In this assignment you are required to use your student number to generate input.

Take your student number and prefix it by "98" and postfix it by "52". This will give you a twelve digit initial input number. Call the digits of that number $d[1], d[2], \ldots, d[12]$ (so that $d[1] = 9, d[2] = 8, \ldots, d[12] = 2$).

Apply the following algorithm to these twelve digits:

```
1  for i = 2 to 12
2      if d[i] == d[i − 1]
3          d[i] = (d[i] + 3)  mod 10
```

After applying this algorithm, the resulting value of d forms your 12-digit SNI. Write down your initial number and your resulting SNI.

(b) (4 marks) Construct a graph S with nodes **all** the digits $0, 1, \ldots, 9$. If 2 digits are adjacent in your SNI then connect the left digit to the right digit by a directed edge. For example, if "15" appears in your SNI, then draw a directed edge from 1 to 5. Ignore any duplicate edges. Draw a diagram of the resulting graph. (You may wish to place the nodes so that the diagram is nice, e.g., no or few crossing edges.)

Question 2: Strongly connected components [30 marks total]

Given a directed graph $G = (V, E)$, a subset of vertices U (i.e., $U \subseteq V$) is a *strongly connected component* of G if, for all $u, v \in U$ such that $v \neq u$,

a) u and v are mutually reachable, and

b) there does not exist a set $W \subseteq V$ such that $U \subset W$ and all distinct elements of W are mutually reachable.

For any vertices $v, u \in V$, v and u are mutually reachable if there is both a path from u to v in G and a path from v to u in G.

The problem of finding the strongly connected components of a directed graph can be solved by utilising the depth-first-search algorithm. The following algorithm SCC(G) makes use of the basic depth-first-search algorithm given in lecture notes and the textbook, and the transpose of a graph; recall that the transpose of a graph $G = (V, E)$ is the graph $G^T = (V, E^T)$, where $E^T = \{(u, v) \mid (v, u) \in E\}$ (see Revision Exercises 3: Question 6). (For those who are interested, the text provides a rigorous explanation of why this algorithm works.)

SCC(G)

1 call DFS(G) to compute finishing times $u.f$ for each vertex u
2 compute G^T, the transpose of G
3 call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing $u.f$
4 output the vertices of each tree in the depth-first forest of step 3 as a separate strongly connected component

(a) (15 marks) Perform step 1 of the SCC algorithm using S as input. Do a depth first search of S (from Question 1b), showing colour and immediate parent of each node at each stage of the search as in Fig. 22.4 of the textbook and the week 3 lecture notes. Also show the start and finish times for each vertex.

For this question you should visit vertices in numerical order in all relevant loops:

for each vertex $u \in G.V$ and

for each vertex $v \in G.Adj[u]$.

(b) (3 marks) Perform step 2 of the SCC algorithm and draw S^T.

(c) (12 marks) Perform steps 3, 4 of the SCC algorithm. List the trees in the depth-first forest in the order in which they were constructed. (You do not need to show working.)

Part B (65 marks total)

Question 3: Taint analysis

[Be sure to read through to the practicalities at the end before starting.]

The input to a program is often not under the control of the programmer. For security-critical programs it is important that any variables whose values depend on the inputs to a program are treated with care. Any variables that depend on the inputs are referred to as *tainted*.

The goal of this question is to analyse a program to determine which variables are tainted. In order to do this, the program is represented by a *control-flow graph*, a directed graph with unique entry and exit vertices. Each edge of the graph is labeled with a primitive statement that is either

- the **null** statement, that has no effect, or
- an assignment statement, that updates a variable using an expression which depends on a set of variables.

For example, the following program with input b taints both y and z.

inputs b;
{ y = b; z = y; }

It is represented by a control-flow graph with three vertices, *entry*, *mid* and *exit*, in which there is an edge from *entry* to *mid* labeled with the assignment $y = b$ and an edge from *mid* to *exit* labeled with the assignment $z = y$. At *entry* the input b is tainted; at *mid* both b and y are tainted; and at *exit* b, y and z are all tainted.

To indicate that an assignment to a variable x may depend on multiple variables, the right side of an assignment is represented by a set of variables. For example, the assignment $x = a, b, c$ states that x depends on the three variables a, b and c.[1]

The **select** statement allows alternative paths to be represented abstractly in a program; it abstracts if-then-else and switch statements. The following program has a three-way choice between either assigning b to t, assigning 1 to t or doing nothing.

[1] Normally the expression would include operators, etc., but here we just need the set of variables used in the expression.

inputs b;
select { $t = b$; | $t = 1$; | **null**; }

One path taints t while the others don't, but the overall effect of the select command is to taint t because one path does.

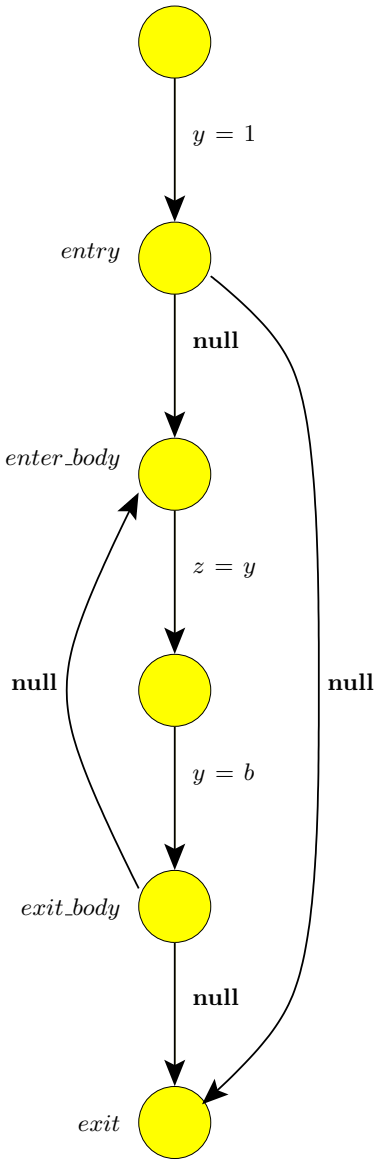
The above program can be represented by a control-flow graph with two vertices *entry* and *exit*. Unusually, we allow it to have three edges from *entry* to *exit* with the first edge labeled with $t = b$, the second labeled with $t = 1$, and the third labeled with **null**.[2]

The **repeat** statement allows a loop to be represented abstractly in a program; it abstracts while-do and do-while loops, etc. The following program assigns 1 to y and then repeats $z = y; z = b;$ zero or more times before terminating.

inputs b;
{
    $y = 1$;
    repeat { $z = y$; $y = b$; }
}

Before the repeat statement starts, b is a tainted input but y is untainted. After the first iteration of the repeat, y is tainted from b but z is not yet tainted. After the second iteration z also becomes tainted because y was tainted on the first iteration. No further changes to the tainted status of variables happens after the second iteration. The overall effect of the repeat statement is to taint both y and z because they both become tainted after a finite number of iterations.

The figure on the right represents the graph for the above example program. A **repeat** statement is represented by a control-flow graph by four vertices *entry*, *enter_body*, *exit_body* and *exit*. The overall entry and exit to the **repeat** statement are the vertices *enter* and *exit*. The subgraph with entry node *enter_body* and *exit_body* represents the statement in the body of the repeat statement. There are edges from *entry* to *enter_body* and from *exit_body* to *exit* both labeled with a **null** statement. To allow the body to be repeated there is an edge back from *exit_body* to *enter_body* labeled with a **null** statement. Finally there is an edge from *entry* to *exit* bypassing the body representing zero iterations of the **repeat** statement.



The dependencies of variables on other variables are determined by the following rules. With each vertex dependencies on inputs are recorded for every variable that depends on an input. At the entry node for the whole program, the fact that b is an input is represented by b depending on itself. For an edge, its target dependencies are calculated from the dependencies at its source vertex based on the form of the statement labelling the edge:

[2] One can avoid multiple edges between two vertices by adding additional vertices and edges labeled with **null** but it is simpler to allow multiple edges.

null the target dependencies equal the source dependencies, and

$x = y, z$ the target dependencies equal the source dependencies but with the dependencies for x replaced with the union of the dependencies of y and z in the source dependencies, for example, if y depends on b and c, and z depends on a and b, then x depends on a, b and c.

Finally, if there is more than one edge entering a node the target dependencies of all the edges entering the node are merged. For example, if the target dependencies of one edge are $\{x \mapsto [b, c], z \mapsto [b]\}$ and the target dependencies of a second edge are $\{x \mapsto [a, b], y \mapsto [c]\}$, the merged dependencies at their common target vertex are $\{x \mapsto [a, b, c], y \mapsto [c], z \mapsto [b]\}$.

The following gives the syntax for the Simple programming language used for this assignment. Bold identifiers are keywords. Alternatives in the grammar are separated by "|" and constructs enclosed in curly braces are repeated zero or more times – be careful to distinguish the braces used for repetition in the grammar description from the strings "{" and "}" used in the Simple language, and similarly between alternatives in the grammar description represented by a vertical bar and the string "|" used in the Simple language.

$$
\begin{array}{rcl}
\mathit{Program} & \rightarrow & \mathit{InputVariables}\ \mathit{Statement} \\
\mathit{InputVariables} & \rightarrow & \textbf{inputs}\ \mathit{Variables}\ \text{";"} \\
\mathit{Variables} & \rightarrow & \mathit{IDENTIFIER}\ \{\ \text{","}\ \mathit{IDENTIFIER}\ \} \\
\mathit{Statement} & \rightarrow & \mathit{NullStatement} \mid \mathit{Assignment} \mid \mathit{CompoundStatement} \mid \mathit{Select} \mid \mathit{Repeat} \\
\mathit{NullStatement} & \rightarrow & \textbf{null}\ \text{";"} \\
\mathit{Assignment} & \rightarrow & \mathit{IDENTIFIER}\ \text{"="}\ \mathit{Expression}\ \text{";"} \\
\mathit{Expression} & \rightarrow & \mathit{NUMBER} \mid \mathit{Variables} \\
\mathit{CompoundStatement} & \rightarrow & \text{"\{"}\ \mathit{Statement}\ \{\ \mathit{Statement}\ \}\ \text{"\}"} \\
\mathit{Select} & \rightarrow & \textbf{select}\ \text{"\{"}\ \mathit{Statement}\ \{\ \text{"|"}\ \mathit{Statement}\ \}\ \text{"\}"} \\
\mathit{Repeat} & \rightarrow & \textbf{repeat}\ \mathit{Statement}
\end{array}
$$

An Eclipse project is provided with partially completed code for the assignment. The code for parsing the program has already been implemented (in packages parseDependencies and Source). An adjacency list representation of a graph is provided in package graphs. The parts of the code you need to modify are in package dependencies.

Your code must be implemented in Java (level 1.6) and conform to the testing framework supplied because your programs will be tested in essentially the same framework. The only output that may be produced by your (submitted) program is that specified in the testing framework. Please be sure to remove any debugging output before submitting.

Your programs will be assessed for both correctness as well as efficiency. As usual, you may lose marks for poorly structured, poorly documented, or hard to comprehend code.

(a) (25 marks) Complete the code to build a control-flow graph representing the program. In class Statement there are a few lines commented out. You need to uncomment these and implement the corresponding methods in class FlowGraph.

(b) (40 marks) Implement the dependency calculation on the flow graph representation by implementing method calculateDependencies in class FlowGraph. This can be implemented in the form of a graph traversal. You will need to implement calculateDependencies for the assignment statement in class Primitive and extend class Dependencies with additional methods.

Practicalities. The program must calculate the taint dependencies for all vertices in the graph but the test framework only outputs the taint dependencies for the exit vertex.

You should use Java level 1.6 for the purposes of this assignment.

There is a set of test input programs supplied in directory test-pgm.

To run the taint analysis program in Eclipse set up a run configuration for the project with main program parseDependencies.Main and argument the full path name of the test file used for input. Instead of the path name you can use `"${resource_loc}"` (including the quote marks if your path name includes blanks) to use the currently selected file as the input.