EECS 3311
Software Design
Lab 2 (100 points), Version 1
Building an Analyzer for a Simple Programming Language

Instructor: Song Wang
Release Date: Feb 8, 2020

Due: 11:59 PM, Friday, March 5, 2020

All your lab submissions must be compilable on the department machines. It is then crucial that should you choose to work on your own machine, you are responsible for testing your project before submitting it for grading. This lab is intended to help you get familiar with the basic OOP design principles.

Check the Amendments section of this document regularly for changes, fixes, and clarifications.

Ask questions on the course forum on the eClass site.

1 Policies

- Your (submitted or un-submitted) solution to this assignment (which is not revealed to the public) remains the property of the EECS department. Do not distribute or share your code in any public media (e.g., a non-private Github repository) in any way, shape, or form before you get the permission from your instructors.

- You are required to **work on your own for this lab**. No group partners are allowed.
- When you submit your solution, you claim that it is solely your work. Therefore, it is considered as an violation of academic integrity if you copy or share any parts of your code or documentation.
- When assessing your submission, the instructor and TA may examine your doc/code, and suspicious submissions will be reported to the department/faculty if necessary. We do not tolerate academic dishonesty, so please obey this policy strictly.

- You are entirely responsible for making your submission in time.

- You may submit multiple times prior to the deadline: only the last submission before the deadline will be graded.
- Practice submitting your project early even before it is in its final form.
- No excuses will be accepted for failing to submit shortly before the deadline.
- Back up your work periodically, so as to minimize the damage should any sort of computer failures occur. You can use a **private** Github repository for your labs/projects.
- The deadline is strict with no excuses.
- Emailing your solutions to the instruction or TAs will not be acceptable.

Amendments

- so far so good

2 Grading Criterion of this Project

- When grading your submission (separate from the report), your Eclipse project will be compiled and then executed on a number of JUnit test cases for evaluating your implementation of Directed Graph.
- For testing your implementation of the CFG, we will executed a number of acceptance tests similar to at01.txt, at02.txt, at03.txt, at04.txt, etc. Each acceptance test is considered as passing only if the output generated by your program is identical to **expected output**.

3 Problem and background

In this project, you are asked to complete the design and implementation of analyzing **Python--** program (i.e., a simplified version of Python program language), which contains two associated functionalities, i.e., control-flow-graph (**CFG**) generation and analysis.

Python-- is a strict indentation-based program language, which uses indentation to separate code blocks and provide a visual feedback on the level of structure your program is in.

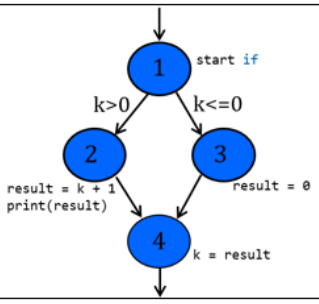
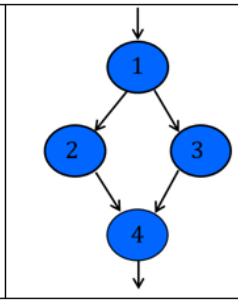
Python-- shares the same syntax with Python while only contains two types of control flow statements (i.e., **if** and **for**). To build CFG of **Python--** code, we categorize program code blocks into five statements showed in Table 1. We have also show the corresponding CFGs and their directed-graph based representations.

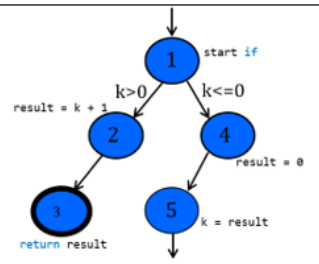
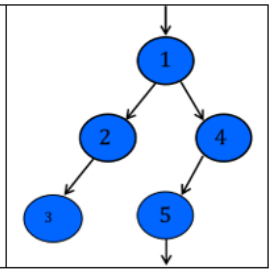
Python-- only has three control flow keywords, i.e., **if**, **else**, and **for**, with which we can break a program into different blocks. **Note that, we allow embedded control flow statements (at most two level of nesting).** We use the code blocks' sequential order in a program to number them in a CFG.

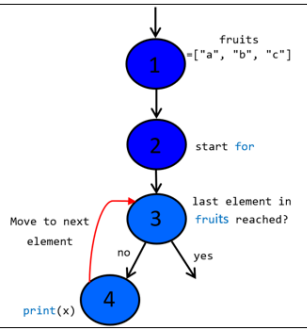
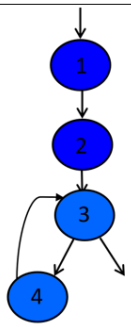
Python-- has one entrance point while can have at least one exit point. There are two types of **exit points**, i.e., the **return** statement (i.e., explicit exit point) or the end of a method (i.e., implicit exit point).

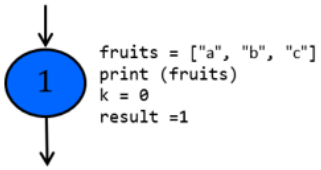
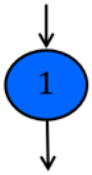
Table 1. Statement types in Python--

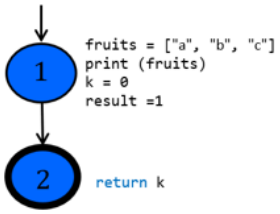
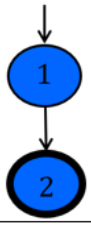
Statement Type (no break/continue)	Example Code	CFG	Directed Graph

If statement	`if (k > 0):`
	`  result = k + 1`
	`  print(result)`
	`else:`
	`  result = 0`
	`k = result`
			

If-Return statement	`if (k > 0):`
	`  result = k + 1`
	`  return result`
	`else:`
	`  result = 0`
	`k = result`
			

For statement	`fruits = ["a", "b", "c"]`
	`for x in fruits:`
	`  print(x)`
			

Basic block
(no control flow
in between)
	`fruits = ["a", "b", "c"]`
	`print (fruits)`
	`k = 0`
	`result =1`
			

Return statement	`fruits = ["a", "b", "c"]`
	`print (fruits)`
	`k = 0`
	`result =1`
	`return k`
			

In this lab, we use LinkedList based directed graph to build and represent CFGs. A directed graph is a set of vertices and a collection of directed edges that each connects an ordered pair of vertices. We say that a directed edge points from the first vertex in the pair and points to the second vertex in the pair. We use the names 1 through V for the vertices in a V-vertex graph. A typical directed graph is shown as follows in Figure 1.

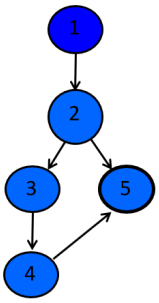


Figure 1: An example directed graph

Branch in the graph: Given a directed graph and the start vertex **s** and a set of end point set **T ={t1, t2, ..., tn}**, each branch starts from s to t ($t \in T$). **A branch can only contain the same vertex at most twice.**

Adjacency matrix representation of graph: The size of the matrix is **VxV** where V is the number of vertices in the graph and the value of an entry A_{ij} is either 1 or 0 depending on whether there is an edge from vertex i to vertex j. **The matrix of example graph in Figure 1 is:**

0 1 0 0 0
0 0 1 0 1
0 0 0 1 0
0 0 0 0 1
0 0 0 0 0

3.0.1 Functionalities of CFG Analyzer

To simplify **Python--** analyzer, we limit the scope of its input to **a single Python-- method**. An input method could contains multiple control flow statements and basic blocks, which always starts from a method declaration. An example input method is as follows:

```
1  def method(k):
2      if (k > 0):
3          result = k + method(k - 1)
4          print(result)
5          return
6      else:
7          result = 0
8      fruits = ["apple", "banana", "cherry"]
9      for x in fruits:
10       print(x)
```

Figure 2: An example input **Python--** method.

Line 1 is the method declaration, which is a fix format for all inputs. **Method declaration is the first vertex in the CFG**. We can use the three control flow keywords (i.e., **if**, **else**, and **for**), **return** statement, and **indentation style** to break a method into sequential code blocks. Line 2 to 7 is a if-return statement, line 8 is a basic block, line 9 to 10 is a for statement, and there also exists an implicit exit point at the end of this method (i.e., **the last vertex in the CFG**).

Python-- analyzer takes 5 options, which is showed as follows:

```
usage: analyzer [-d] [-g] [-h] [-n] -p <input path>
Build and analyze the CFG for a python method.
 -d                 show each branch
 -g                 build CFG and print the adjacency matrix
 -h                 show help information
 -n                 show the number of branch
 -p <input path>    path of the method to be analyzed
```

Figure 3: Commands of analyzer.

Option **p** is required, whose value is the path of the method to be analyzed. Other options are optional. If the input argument has **h**, analyzer will print the help information showed in Figure 3 (**this has already been implemented**). If the input argument has **g**, analyzer will build its CFG and print adjacency matrix of the CFG, if has **n**, analyzer will show the number of branches in the CFG, if has **d**, analyzer will print each branch. You can find more details in the **acceptance tests.**

Steps to run analyzer are as follows:

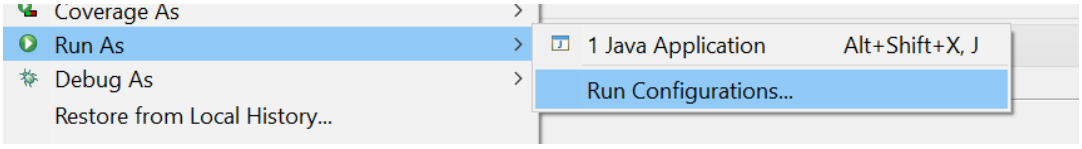


Figure 4: Step 1: Open the run-time configuration of analyzer.

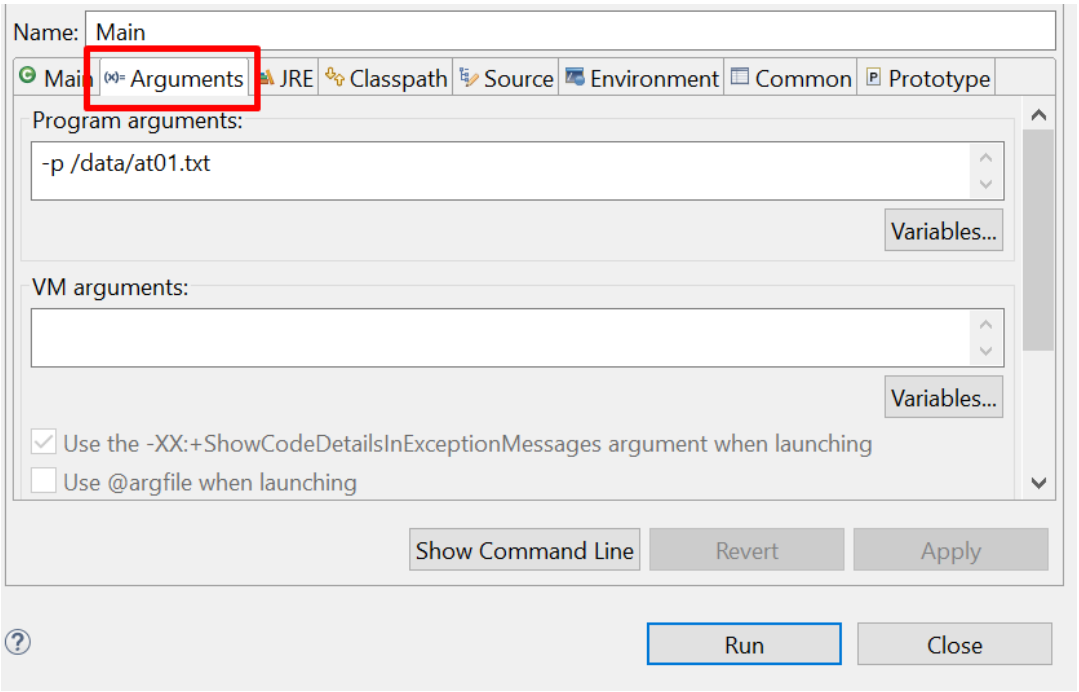


Figure 5: Step 2: Run analyzer with appropriate arguments.

4 Getting Started

- Go to the course eClass page for Section Z. Under Lab 2, download the file **EECS3311_Lab2.zip** which contains the starter project for this lab.
- Unzip the file, and you should see a directory named **analyzer**. It's a Eclipse project.
- You can import this project into your Eclipse as an general Java project.

5 You Tasks

5.1 Task1: Directed Graph (40 points)

- You are expected to write valid implementations in the **Edge**, **ListDGraph**, and **Vertex** classes. Each instance of "TODO:" in these classes indicates which implementation you have to complete.
- Study the **DGraphTest** class carefully: it documents how **Edge**, **ListDGraph**, and **Vertex** expected to work.
- You must not change any of the methods, parameters, or statements in the **DGraphTest** class.
- In the **StudentTest** class, you are required to add as many tests as you judge necessary to test the correctness of directed graph. You must add at least 20 test cases in **StudentTest**, and all of the must pass. (In fact, you should write as many as you think is necessary.)
- You will not be assessed by the quality or completeness of your tests (i.e., we will only check that you have **at least 20 JUnit test cases/methods** and all of them pass). However, write tests for yourself so that your software will pass grading tests that we run to assess your code.

5.2 Task2: CFG Analysis (50 points)

- You are expected to write valid implementations for building CFG for a given **Python--** method based on the directed graph you built in Task 1.

- You are not allowed to use any graph related third-party libraries in your Task 2. You can create new classes/methods in the analyzer package to finish your Task 2.
- Study the **acceptance tests** (i.e., at01.txt, at02.txt, ...) and their expected outputs under different options carefully: it documents how the analyzer expected to work.

5.3 Task3: Design Report (10 points)

- Compile and print off a report including: names and your CSE logins.
- The class diagrams for your design (https://app.diagrams.net/); You must also include the **draw.io XML source file of your class diagram and its exported PDF** in the docs directory when you make your electronic submission.
- Explain in details how your design your software, which design patterns you have used in your implementation.
- At most 2 pages.

6 Submission

To get ready to submit:

- Close Eclipse
- Zip your lab2 project with name 'EECS3311_Lab2.zip'.

By the due date, submit via the following command:

```
submit 3311 lab2 EECS3311_Lab2.zip
```