# CS453 Automated Software Testing

Spring 2020

## Coursework #2: Search Based Test Data Generation
## Due by 23:59, 1 June 2020

### 1 Aim

Write an automated test data generation tool for a subset of Python. Your tool should be able to take a Python source code as input, and generate a branch adequate test suite for a function in the source code as output (i.e., the generated test suite should achieve as high branch coverage as possible).

The dynamic typing in Python makes test data generation for arbitrary function highly challenging. Therefore, we are going to target the subset of Python functions that:

- takes only integer arguments (however, there may be an unspecified number of arguments).
- has only integer type local variables; these can be assigned, and be used in predicates too.
- can contain loops
- contains predicates that only involve relational operators (`==`, `!=`, `<`. `>`. `<=`, `>=`), integer variables, and calls to functions with integer return type
- can call external functions, including external libraries (but we only want to achieve coverage for the target function itself)

You would first instrument the target source code, so that whenever you execute the function with an arbitrary input, you can compute the standard fitness function for branch coverage for any branch in the source code. Note that, to do this, you may have to add additional code to the original source code. Then minimise this fitness function using whatever optimisation algorithm. A good starting point would be AVMFramework(`http://avmframework.org`), but you can be creative as well.

The program should print out the following:

```
> python covgen.py target.py
1T: 3, 5, 4
1F: 3, 3, 4
2T: 4, 5, 3
2F: -
```

Each line corresponds to a branch in the target program. You should number predicates using a consistent scheme, and use `T` and `F` to mark true and false branches. The actual test input should follow this branch index. If your tool decides that a specific branch is not reachable (either because the condition is unsatisfiable, or because the tool failed to find a satisfying input), print `-`.

Achieving the branch coverage is the *minimum* functional requirement for this coursework. A non-trivial portion of the mark will be reserved to reward unique and imaginative ideas, as well as how polished your tool is. Remember that you are writing a (small) tool (i.e., something that can be used by a 3rd party user), and not just writing assignment code. Which features would you like to see in such a tool?

## 2 Marking

We will use a set of hidden target functions to evalute the tools you submit. Marking criteria will include:

- The branch coverage achieved by your tool
- The quality of your implementation and report
- (Where applicable) the uniqueness of your idea

## 3 Deliverables

Each person should submit the following deliverables by the submission deadline:

- **Implementation**: your implementation, as much self-contained as possible.
- **Report**: include a written report that contains detailed descriptions of how you approached the problem. There is no page limit.

For ease of marking, follow the following directory structure, and submit a zip file containing the top level directory, through KLMS.

```
[your student number]
├── report.pdf ..... Your report documenting both implementations
└── tool ..... Your implementation
```

## 4 Guidelines

- For analysis of the source code, `astor` (`https://pypi.python.org/pypi/astor`) provides a convenient wrapper for Python Abstract Syntax Trees (ASTs).
- **Do your own work**: I trust you not to commit academic misconduct, for which there will be serious consequences. In the case of this coursework, cheating would include taking materials from existing online material. You are required to provide links to the original repository: copying from that repository will be, by definition, easily detected.
- Report should be in English.