# Introduction

- **Ultimately, the project score should match the appropriate score on the rubric listed on the course policies page of the course. See http://web.cse.ohio-state.edu/software/web/policies.html**
- For this project, `Statement2` will be worth 6 points and `Program2` will be worth 4 points. `Statement2.java` and `Program2.java` should be the only 2 files considered during grading (You are not directly grading the students' JUnit).
- Testing this project will be slightly different than the others: the Eclipse project in the test directory has the set-up that I would recommend using for grading. In the src folder, you'll find `StatementTester.java` and `ProgramTester.java` that were given to students and you can use to do some high-level testing (using the `test/statement.bl` and `test/program.bl` input files or any other files you may come up with); in the test folder, I have included JUnit test fixtures that you should use to test the kernels (they make use of the bl files provided in the test folder). After importing the projects, you should copy and paste each group's `Statement2.java` and `Program2.java` files into the grading project (src folder) and then run the tests. Then remove `Statement2.java` and `Program2.java` and repeat.
- Note: Some suggested point deductions for specific issues are in parentheses. Also, if the project contains any bugs (even just one), the project should not receive a 10.
- Bugs should be penalized and so should also any major violations of recommended good practices and any solution that is way too complicated.

# Implementation - Program

The following is a list of what is the 'expected' solution and some of the common bugs. Suggested penalty for each bug is -1/2, unless otherwise noted.

## Private Methods

- createNewRep
  - Set `this.name` to `"Unnamed"`.
  - Set `this.context` to a new `Map<String, Statement>`.
  - Set `this.body` to a new `Statement` object.

## Kernel Methods

- setName
  - Set `this.name` to `n`.
- name
  - Return `this.name`.
- newContext
  - Return `this.context.newInstance()`.
- swapContext

1. Declare a temp context to hold a copy of the current (soon to be former) reference to `this.context`.
2. Create a new empty Map using any of `c.newInstance`, `this.context.newInstance`, or a call to `newInstance` on the temp context (do not use the `newContext()` method [violation of kernel purity rule]) and store the reference to the new empty Map in `this.context`.
3. Transfer `c` to `this.context`.
4. Transfer the temp context to `c`.

- Typically the solution could use three transferFroms, or only two with another assignment as described above. If the solution only uses assignment, check that either `this.context` or `c` is wrong.
- The solution should not use the `combineWith()` method. (-1/2)

- newBody
  - Return `this.body.newInstance()`.
- swapBody
  1. Declare a temp body to hold a copy of the current (soon to be former) reference to `this.body`.
  2. Create a new empty block Statement using any of `b.newInstance`, `this.body.newInstance`, or a call to `newInstance` on the temp body (do not use the `newBody()` method [violation of kernel purity rule]) and store the reference to the new empty block Statement in `this.body`.
  3. Transfer `b` to `this.body`.
  4. Transfer the temp body to `b`.

  - Typically the solution could use three transferFroms, or only two with another assignment as described above. If the solution only uses assignment, check that either `this.body` or `b` is wrong.

# Implementation - Statement

The following is a list of what is the 'expected' solution and some of the common bugs. Suggested penalty for each bug is -1/2, unless otherwise noted.

## Private Methods

- createNewRep
  - `this.rep` needs to be assigned to a tree with its root as a `StatementLabel` as type `BLOCK` and no children. (-1)
  - The solution must use `this.rep.newSequenceOfTree()` and CAN NOT call the `Sequence()` constructor.

## Kernel Methods

The following is a list of steps that each method will generally follow (the solution does not have to follow them exactly) as well as a list of common issues. If anywhere in the solution disassemble is called more than once, this may be a sign that the students are not taking advantage of the direct access they have to the representation. If this is done only once, apply a 1/2 point penalty, else apply a 1 point penalty. In the three `BLOCK` methods (`addToBlock`, `removeFromBlock`, and `lengthOfBlock`,) described below, two alternatives, both of which are acceptable, are shown. However, if any of the methods use the alternative

involving disassembling and assembling the tree, a comment should be made that the `Tree` API allows more convenient access.

- Kind
  - Return `this.rep.root().kind`.
  - If the student's solution disassembles and then assembles the tree, a comment should be made that there is a method which provides more convenient access to the root.
- addToBlock

  Either

  1. Need to cast the incoming `Statement` to a `Statement2` variable, say `localS`.
  2. Add the representation of `localS` as a subtree at position `pos` to the representation of `this` using a call to `addSubtree`.
  3. Call createNewRep on `localS`.

  or

  1. Need to cast the incoming `Statement` to a `Statement2` variable, say `localS`.
  2. Disassemble the tree.
  3. Add the representation of `localS` to the sequence at position `pos`.
  4. Assemble the tree.
  5. Call createNewRep on `localS`.
- removeFromBlock

  Either

  1. Create a new `Statement2` object, say `s`.
  2. Remove the subtree of the representation of `this` at position `pos` using a call to `removeSubtree`.
  3. Set the rep of `s` to the removed subtree (this and the previous step can be done in one line).
  4. Return `s`.

  or

  1. Create a new `Statement2` object, say `s`.
  2. Disassemble the tree.
  3. Remove the element in the sequence at position `pos`.
  4. Set the rep of `s` to the removed element (this and the previous step can be done in one line).
  5. Assemble the tree.
  6. Return `s`.
- lengthOfBlock
  - Return `this.rep.numberOfSubtrees()`.
  - This can also be accomplished by disassembling the tree, storing the length of the `Sequence`, assembling and then returning the length. (However. in this case, a comment should be made that the `Tree` API allows more convenient access to the number of subtrees.)

- assembleIf
  1. Cast the incoming `Statement` object to a `Statement2` object.
  2. Initialize a `StatementLabel` object of type `IF`, with the condition provided.
  3. Initialize a new `Sequence` of `Tree`. (using `newSequenceOfTree()`)
  4. Add the representation of the casted `Statement2` object to the sequence of trees.
  5. Assemble `this.rep` using the `StatementLabel` created as the root and the `Sequence` of trees as the children.
  6. Clear s (using `createNewRep()`).

  – Code already provided (students should not have changed the code).
- disassembleIf
  1. Cast the incoming `Statement` object as a `Statement2` object.
  2. Create a new `Sequence` of trees (using `newSequenceOfTree()`).
  3. Disassemble `this.rep` and store the returned root as a `StatementLabel`.
  4. Remove the element at position 0 in the `Sequence` and set the casted `Statement2` object's `rep` to the return value.
  5. Clear this (using `createNewRep`).
  6. Return condition from the stored `StatementLabel`.

  – Code already provided (students should not have changed the code).
- assembleIfElse
  – Code will be the similar to `assembleIf`, but will have a second element in the `Sequence`.
  – Need to clear both parameters (using `createNewRep()`, not `clear()`).
- disassembleIfElse
  – Code will be the similar to `disassembleIf()`, but will have a second element in the `Sequence`.
  – Need to clear `this` (using `createNewRep()`, not `clear()`).
- assembleWhile
  – Code will be the same as `assembleIf()`, but the `StatementLabel` will be of type `Kind.WHILE`.
  – Need to clear parameter (using `createNewRep()`, not `clear()`).
- disassembleWhile
  – Code will be the same as `disassembleWhile()`.
  – Need to clear `this` (using `createNewRep()`, not `clear()`)
- assembleCall
  – Assemble `this.rep` with a root as `StatementLabel` of type call with the corresponding instruction, and no children.
- disassembleCall
  – Can disassemble the tree, or access the root directly
  – Need to clear `this` (using `createNewRep()`, not `clear()`)
  – Return as a `String` the `Instruction` held by the `Statement`.