Assignment 0: Groups & Projects

CS3213 Foundations of Software Engineering (AY21/22 Sem2)

Submission Deadline: **Fri 14/01/2022, 11:59 pm**

- Note that this is not a typical assignment sheet, as it is meant for your preparation for the course.
- Any questions can be posted in the **LumiNUS** forum.
- If the LumiNUS forum should not (yet) be available, or you should have any personal question which you do not want to share with others, feel free to drop an email to **yannic@comp.nus.edu.sg**. But note that any general(izable) question submitted via email will just be copied to the forum and answered there. Therefore, we recommend to use the LumiNUS forum whenever possible.
- There will be no *marks* for this sheet, but finding a group **and** a project will be **necessary** to participate in the lab.

Overview

All lab assignment in CS3213 (except for Assignment 1) need to be submitted in *groups*. Therefore, it will be crucial to have your group ready for action as soon as possible. We are not going to assign random groups, so it will be your task to form groups. Furthermore, CS3213 is accompanied with a software engineering project, for which you need to select a topic for your group.

In this assignment sheet we take the opportunity to present the overall system idea and the available project topics, which you can pick as a group project. You can already make yourself familiar with the topics, conduct some readings and collect background knowledge. This way you save time during the semester and you can make an informed decision when selecting a topic.

Task 1: Find a Group!

For the future assignments and project works, you will need to form groups of 3-4 students. Note that the members of your group need to be in the same tutoring group. Please coordinate with your fellow students, e.g., by using the forum in LumiNUS. For all future assignments (except for Assignment 1) it will be mandatory to submit solutions as a group.

As long as the LumiNUS module is not available yet, you can enter your temporary group and project selection in this Google Sheet: `https://docs.google.com/spreadsheets/d/15sk6WnvQHTjClhMi_TUuyDkylow6lOmMHVhD5n3Qu-I/edit?usp=sharing`. We will later take care of porting the groups and project selections to LumiNUS.

Task 2: Pick a Project!

Within your group, you need to select one particular project to work on during the lab. The project specific tasks will start with Assignment 4, in which you will design your project's implementation. It is important that you as a *group* make this selection so that everyone in the group is committed to the project. The projects are different and concern different aspects of the overall system. So read the descriptions carefully and do some background check of the topics.

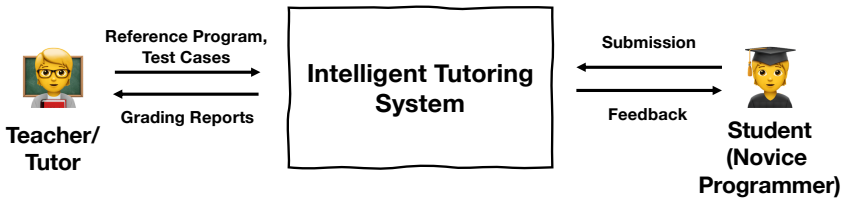

Figure 1: High-Level Overview

System Overview and Available Projects

The high-level idea of the *Intelligent Tutoring System* is to introduce an *automated* technique to provide **feedback** and **grading suggestions** for programming assignments. As shown in Figure 1, for a given programming assignment, the tutor would provide a reference solution and some test cases, while the student would submit a solution and would receive feedback for that. The feedback should go beyond the simple execution of test cases, but should tell the student where and how to fix the submission. More sophisticated and gradual ways of feedback could be also introduced.

Developing such a system includes many conceptual and technical challenges. We summarized some of them into projects, from which you will need to choose one for this course. Please find below a collection of all projects, and more detailed descriptions on the following pages. We group projects by their provided functionalities. Each project has assigned estimated difficulty levels (*Low*, *Medium*, *High*) in the categories: *coding* (i.e., programming intensity), *theoretical* complexity (i.e., need for background study), amount of *research* involved (i.e., be innovative and create something new), and the involvement of *HCI* (Human-Computer Interaction) aspects.

Topic 1 – Parsing

Project 1.1 *C Parser*: Develop a parser to transform C programs into a (provided) common data structure based on the control-flow graph (CFG). Additionally, provide a concretizer, which back-transforms the program in the internal common data structure to a C source file.
[Coding: *High*, Theory: *Low*, Research: -, HCI: -]

Project 1.2 *Python Parser*: Develop a parser to transform Python programs into a (provided) common data structure based on the control-flow graph (CFG). Additionally, provide a concretizer, which back-transforms the program in the internal common data structure to a Python source file.
[Coding: *High*, Theory: *Low*, Research: -, HCI: -]

Topic 2 – Aligning / Matching of Programs

Project 2.1 *CFG-Based Alignment*: Develop an automated alignment of the reference program and the submitted program based on the basic blocks of the programs' control-flow graph (CFG) representation. This also includes the development of an automated mapping for the variables between the reference program and the submitted program.
[Coding: *Medium*, Theory: *Medium*, Research: *Low*, HCI: -]

Topic 3 – Error Localization / Program Interpretation

Project 3.1 *C Interpreter*: Develop an interpreter that allows to execute a C program with regard to the basic blocks in its CFG. Further, use the provided test cases to identify the root cause of the problem with regard to the basic blocks in the CFG. Implement an error localization that compares the execution traces of a reference program and the submitted program.
[Coding: *High*, Theory: *Medium*, Research: -, HCI: -]

Project 3.2 *Python Interpreter*: Develop an interpreter that allows to execute a Python program with regard to the basic blocks in its CFG. Further, use the provided test cases to

identify the root cause of the problem with regard to the basic blocks in the CFG. Implement an error localization that compares the execution traces of a reference program and the submitted program.
[Coding: *High*, Theory: *Medium*, Research: -, HCI: -]

Project 3.3 *Error Localizer*: Conduct a literature study on error localization. Develop at least two error localization algorithms from different domains (e.g., statistical fault localization and analysis-based fault localization) for the provided framework and evaluate their efficacy.
[Coding: *Medium*, Theory: *High*, Research: *Low*, HCI: -]

Topic 4 – Transforming / Repairing Programs

Project 4.1 *Refactoring-based Repair*: Develop a repair workflow that first generates semantic-preserving refactorings of a reference program so that it increases the chances of a structural alignment with a submitted program (see Project 2.1). Afterwards, it uses a matching refactoring to repair the submitted program by mutating program expressions. Strive for a minimal repair which satisfies the failing test case(s).
[Coding: *Medium*, Theory: *Medium*, Research: *Medium*, HCI: -]

Project 4.2 *Optimization-based Repair*: Develop a repair algorithm that (1) generates local repairs at each basic block by matching the submission and the reference solution, and (2) determines the complete repair (i.e., a subset of local repairs) by using some optimization strategy, which minimizes the overall repair cost.
[Coding: *Medium-High*, Theory: *High*, Research: *Low*, HCI: -]

Project 4.3 *Synthesis-based Repair*: Develop a repair algorithm that searches for a repair by synthesizing program expressions. The synthesis will be driven by the available components at the specific source location. It requires a specification inference, which results in a repair constraint.
[Coding: *Medium*, Theory: *High*, Research: *Medium*, HCI: -]

Topic 5 – Feedback Generation

Project 5.1 *Automated Feedback*: Develop a feedback mechanism to summarize all obtained results in an appropriate and comprehensible manner for the user. For example, show root causes of the problems and provide explanation by annotating the code.
[Coding: *Low*, Theory: *Medium*, Research: *Medium*, HCI: *High*]

Project 5.2 *Automated Grading*: Develop a automated grading mechanism, which is beyond simple output of passing and failing test cases, e.g., it should take into account the necessary effort for fixing the submitted program.
[Coding: *Low*, Theory: *High*, Research: *High*, HCI: *Low*]

1 Parsing

1.1 C Parser

Overview: The goal of the *parser* (C or Python) is to generate the intermediate program model from the raw source code. In general, the entire parser workflow covers three different aspects:

1. Generation of the Abstract Syntax Tree (AST).
2. Generation of the Control Flow Graph (CFG).
3. Generation of the Single Static Assignment (SSA).

Additionally, this project should provide a *concretization* functionality, which takes the intermediate program model and produces an actual source file (C or Python). We recommend to first work on the parser functionality because the learned concepts can then easily be applied on developing the concretization part. Furthermore, both components (a parser and a concretizer) can be nicely integrated and tested.

Complexity: [Coding: *High*, Theory: *Low*, Research: -, HCI: -]

Prerequisite and References: To work on the parser (C or Python) project, the students need to understand the principles of a *Lexer*, a *Parser*, and the *Abstract Syntax Tree* (AST). Furthermore, they need to develop an understanding of the Single Static Assignment (SSA) principle and the concept of a Control-Flow Graph (CFG). The following references can help to prepare for this project:

- `https://www.sciencedirect.com/topics/computer-science/abstract-syntax-tree`
- `https://www.sciencedirect.com/topics/computer-science/control-flow-graph`
- `https://www.cs.cmu.edu/~fp/courses/15411-f08/lectures/09-ssa.pdf`
- `https://www.antlr.org`
- `https://github.com/antlr/antlr4`

Assumptions and Dependencies:

- Inputs: Source code of a program.
- Outputs: Intermediate model of the given program, or in case of the *Concretizer*, again the semantically equivalent program in source code.
- Dependencies: This project depends on the definition of the intermediate model for a `Program` in our system. This data structure cannot be changed by the students as all other projects and system components depend on this definition. Additionally, it will have access to auto-generated ANTLR files that can be used for the project, however, the students are allowed to decide for a different implementation direction.

Abstract Syntax Tree (AST): The first step for the parser module would be to generate the abstract syntax tree. This requires a lexer to separate the individual characters in the raw source code into meaningful tokens. These tokens will then be the leaves of the AST. The intermediate nodes in the AST are representative names that identify the type of children it has. Coding a lexer and a parser from scratch is an excruciatingly difficult task due to all the possible edge cases that one may need to consider. We instead decided to use ANTLR to generate a large extent of the parser code for the following reasons:

1. ANTLR is open-source. There exists a high level of documentation and support for students to refer to.
2. There is exhaustive grammar for most popular programming languages, and this allows for scalability for this implementation.

3. A good level of support for most popular IDEs.
4. The syntax for grammar is not too difficult to comprehend, students should be able to change them if there is a need to.

Apart from generating the lexer and parser, ANTLR also generates visitor and listener classes, which intuitively follows the visitor pattern and observer patterns respectively. In particular, the visitor class provides a good level of flexibility to choose which children nodes to visit directly. Therefore, this class will be responsible for traversing through the generated AST from ANTLR, and for extracting the relevant locations and expressions needed for our intermediary program model. While this is in favour of a more fine-grained control of traversing through the AST, it is notable that the generated AST contains a lot of nodes, even for a trivial program due to the exhaustive grammar provided. During implementation, students must be careful to invoke the `visit()` method to all the node's children, else some sub-trees will not be visited.

Control Flow Graph (CFG): The intermediary program model follows the standard control-flow of the program. Each node of the CFG is denoted as a location. Below is an explanation of how the common types of expressions are handled.

If-else statements – Note that for our model, branching of the if-else statements occurs at the same location. This is represented by the special variable 'ite'. The format for the ite operator is as such:

```
ite( <conditional expression>,
     <true branch expression>,
     <false branch expression> )
```

The table below illustrates a simple example:

Program	Parsed expression	Location Number

`int isEven(int a) {`
`  if (a % 2 == 0) {`
`  return 0;`
`  } else {`
`  return 1;`
`  }`
`}`
	`$ret = ite((a%2==0), 0,1)`	1

Note: this optimization of the if-else branch only occurs when there is no looping structure within the if-else statements.

For-loop – Since the CFG should represent the loop structure of the program, this program is split into different locations (i.e., different nodes in the CFG). The following listing shows a simple for-loop, and the following table shows how this specific code should be parsed by the parser project:

```
 1:    #include <stdio.h>
 2:    int main(int c) {
 3:       int counter = 0;
 4:       for (int i=0; i<counter; i++) {
 5:          if (checkPrime(i)) {
 6:              counter = counter + 1;
 7:          }
 8:       }
 9:       printf("%d", counter);
10:       return 0;
11:    }
```

Location	Location Description	True Branch	False Branch	Variable Assignments
1	at the beginning of the function `main`	2	NULL	`counter = 0` `i = 0`
2	the condition of the `for` loop at line 4	4	5	`$cond = i < counter`
3	update of the `for` loop at line 4	2	NULL	`i = i + 0`

4	inside the body of the `for` loop beginning at line 4	3	NULL	`counter = ite(`
				`FuncCall(`
				`checkPrime, i),`
				`counter+1,`
				`counter)`

5	*after* the `for` loop starting at line 4	NULL	NULL	`$out = StrAppend(`
				`$out, StrFormat`
				`("%d",counter))`
				`$ret = 0`

Single Static Assignment (SSA): The final step of the parser is to ensure that in each location (i.e., basic block in the CFG), every variable is in the SSA form. This is crucial for the working functionality of the other modules using this parser's output. SSA form can be achieved by identifying the last appearance of a variable in a particular location and replacing all non-last appearances of that variable by a new (fresh) variable (e.g., `a&1`). Now, all the variables are in SSA form, but the newly created fresh variables are redundant. Some post-processing of the SSA form is done such that the expressions assigned to the fresh variables are propagated, effectively removing all the new variables created. During this post-processing stage, the algorithm will keep track of variables that have been disruptively modified and mark them as primed (e.g., `c'`). The following table demonstrates this approach:

Program	Pre-Process	SSA Form	Post-Processing

`int main(int c) {`
`int a = 5;`
`int b = a + c;`
`c = c + 1;`
`a = c;`
`}`
	`a = 5`
	`b = a + c`
	`c = c + 1`
	`a = c`
		`a&1 = 5`
		`b = a&1 + c`
		`c = c + 1`
		`a = c`
			`a = c'`
			`b = 5 + c`
			`c = c + 1`

1.2 Python Parser

(see description for project 1.1)

2 Aligning / Matching of Programs

2.1 CFG-Based Alignment

Overview: This project represents the pre-processing step before any error localization or repair can be applied on the submitted program. After the programs have been parsed to the intermediate program models, this project would *align* the two programs. Therefore, it records a mapping between matching basic blocks in both programs. Furthermore, it is necessary to create a mapping between the variables. The variable mapping can be explored along various heuristics, e.g.,

- the similarities of variable names, or
- the relation between variable appearances on the left hand side (LHS) and right hand side (RHS) in each basic block.

You can also develop other (more advanced) heuristics. The goal is to identify a *bijective* mapping between the variable sets in both programs. The information about the structural and variable alignment of the two programs, will be the input for the following repairing stages.

Complexity: [Coding: *Medium*, Theory: *Medium*, Research: *Low*, HCI: -]

Prerequisite and References: Since the alignment is based on the basic blocks in the CFG, the students need to understand how a CFG works. It further would be helpful to study some background on how to detect similar variables in two program, e.g., with the provided references below:

- `https://www.sciencedirect.com/topics/computer-science/control-flow-graph`
- *A Comparison of String Distance Metrics for Name-Matching Tasks*, `https://www.cs.cmu.edu/~wcohen/postscript/ijcai-ws-2003.pdf`
- *DECKARD: Scalable and Accurate Tree-based Detection of Code Clones*, `https://www.cs.ucdavis.edu/~su/publications/icse07.pdf`

(Note: the referenced papers are related but not the core of this project)

Assumptions and Dependencies:

- Inputs: This project requires as input two intermediate program models, representing the reference solution and the submitted program.
- Outputs: This project should produce a mapping between two programs in terms of the basic blocks in their CFGs and the included variables.
- Dependencies: This project does not require further dependencies on other modules of the system.

Algorithms: To get the students started with the problem domain, we provide a simple but incomplete algorithm for the structural alignment. Line 10 in the following algorithm contains a function `match_function`,for which no implementation is provided. It will include the details about the matching algorithm that the students will need to figure out themselves. After the students have implemented the structural mapping, they subsequently can work on the variable mapping, which is more challenging.

```
Input: Let P and Q be the given two program objects
Output: Map<String,<int, int>> that maps the basic block locations of P
    with basic-block locations of Q, for each function

 1: def match_program(P, Q):
 2:   mappings = {}
 3:   if len(P.fncs) != len(Q.fncs):
 4:      return false
 5:   for fncName_P, fncP in P.fncs.items():
 6:      if not fncName_P in Q.fncs.items():
 7:         return false
 8:      fncQ = Q.fncs[fncName_P]
 9:      mappings[fncName_P] = {}
10:      match_function(fncP, fncQ, mappings)
```

3 Error Localization / Program Interpretation

3.1 C Interpreter

Overview: As a prerequisite of error localization and program repair, an interpreter is used to concretely execute a program to get its execution trace with specific input. At this stage, the project assumes that the program is presented in some CFG-based representation. In this project, the students are asked to implement an interpreter for the CFG-based representation. Given a program P in CFG-based representation and an input i for P, the interpreter should generate the execution trace which records all variables' values during interpretation. In general, the interpreter mainly requires the following two steps:

- Implement highly reusable and flexible code structure that traverse the CFG-representation of program P correctly.
- Implement functions to execute all components in P's CFG representation and record all variables' values in Memory with input i step by step during the traversing.

Furthermore, the students are asked to use their interpreter to implement a simple variant of a trace-based error localization. It should use the resulting execution traces from the interpreter to detect mismatches between the reference and submitted program with regard to the values along the execution traces.

Complexity: [Coding: *High*, Theory: *Medium*, Research: -, HCI: -]

Prerequisite and References: Students need to understand what the concept of an AST (Abstract Syntax Tree) and the general concept of program interpretation. The following references can help to prepare for this project:

- `https://en.wikipedia.org/wiki/Control-flow_graph`
- `https://en.wikipedia.org/wiki/Abstract_syntax_tree`
- `https://en.wikipedia.org/wiki/Interpreter_(computing)`
- `https://betterprogramming.pub/compiler-vs-interpreter-d0a12ca1c1b6`

Assumptions and Dependencies:

- Inputs: This project requires as input an intermediate program model and the input, for which the program should be executed/interpreted.
- Outputs: The interpreter shall produce a `Trace` object that includes the sequence of executed basic blocks and the corresponding memory values.
- Dependencies: This project does not require further dependencies on other modules of the system.

Example: To further illustrate the problem and its solution, please find below a simple example. Imagine the source code of program P is defined as follows:

```
def assign_to_one(a):
    b = 1 + a
```

and $P's$ CFG-representation looks like follows:

```
fun assign_to_one () : *
----------------------------------------
initloc : 1
Loc 1 (around the beginning of function 'assign_to_one')
----------------------------------------
b := Add(1, a)
----------------------------------------
True -> null   False -> null
```

To execute the program P, we would expect an execution trace output like:

```
Trace [(fnc=assign_to_one, loc=1, mem={a=10, b=<undef>, $out'=,
   b'=11, a'=10, $out=, $ret'=<undef>, $ret=<undef}>)]
```

In the printed execution trace.

- `fnc=assign_to_one` represents the execution trace is for function `assign_to_one`.
- `loc=1` represents the execution trace is for code in location 1 of function `assign_to_one`.
- `mem={a=10, b=<undef>, $out'=, b'=11, a'=10, $out=, $ret'=<undef>, $ret=<undef>}` represents variables' concrete values in the location 1 of function `assign_to_one`. The variables *without/with* `'` notation store the values *before/after* the interpreter executes the statement in location 1 of function `assign_to_one`.

3.2 Python Interpreter

(see description for project 3.1)

3.3 Error Localizer

Overview: The error localization is the basis for a well-working repair strategy as it identifies one of its main ingredients, the potential *fix locations*. This project will give students the chance for an in-depth study of existing error localization techniques and to apply these approaches in the context of our intelligent tutoring system. This project requires to:

1. Perform a literature study on error/fault localization, and choose (at least) two different techniques.
2. Implement the chosen techniques within the context of our system.
3. Evaluate the techniques and determine strengths and weaknesses of the chosen techniques.

For example, you can choose some *statistical* fault localization and some *analysis-based* fault localization technique.

Complexity: [Coding: *Medium*, Theory: *High*, Research: *Low*, HCI: -]

Prerequisite and References: Since the alignment is based on the basic blocks in the CFG, the students need to understand the concept of a CFG. Furthermore, this project will require the students to conduct their own literature study. The following references can help to prepare for this project:

- `https://www.sciencedirect.com/topics/computer-science/control-flow-graph`
- *A Survey on Software Fault Localization*, `https://doi.org/10.1109/TSE.2016.2521368`
- *Evaluating and improving fault localization*, `https://doi.org/10.1109/ICSE.2017.62`
- *On the Accuracy of Spectrum-based Fault Localization*, `https://doi.org/10.1109/TAIC.PART.2007.13`
- *Fault localization using execution slices and dataflow tests*, `https://doi.org/10.1109/ISSRE.1995.497652`
- *Localizing Vulnerabilities Statistically From One Exploit*, `https://doi.org/10.1145/3433210.3437528`
- *Beyond Tests: Program Vulnerability Repair via Crash Constraint Extraction*, `https://doi.org/10.1145/3418461` (i.e., constraint/dependency-based fault localization)

Assumptions and Dependencies:

- Inputs: This project requires as input two intermediate program models, representing the reference solution and the submitted program. Furthermore, it takes a set of inputs that can be used for the error localization.
- Outputs: This project should produce at least two different error localizer implementations, which both produce a list of error locations (i.e., list of matching pairs of basic blocks from two programs).
- Dependencies: This project has access to the interpreter to execute inputs on the program models.

4 Transforming / Repairing Programs

4.1 Refactoring-based Repair

Overview: As repair we denote the fixing of the submitted program by reusing the existing reference implementation. At this stage, the project assumes that the two programs are presented in some CFG-based representation. However, the general assumption (e.g., as also used in project 4.2) that reference program and submitted program can be fully aligned, does not always hold because the CFG representation of various submitted programs may differ a lot from the reference solution. Therefore, in this project, the students are asked to implement a repair approach based on refactoring rules which increase the possibility of aligning the two programs and fix the submitted program by expression mutation. In general, this specific approach requires four steps:

1. Design and implement a set of refactoring rules that change the syntactic structure but preserve the semantic meaning of the reference program.
2. Design and implement a set of expression mutation operators for the program CFG-representation.
3. Apply the refactoring rules on reference programs to generate a set of refactored programs and align the refactored programs and the submitted program using alignment techniques in project 2.1.
4. Apply the expression mutation operators on the submitted program to find the best (minimal) repair.

Complexity: [Coding: *Medium*, Theory: *Medium*, Research: *Medium*, HCI: -]

Prerequisite and References: Students need to understand what is program refactoring and program mutation. The following references can help to prepare for this project:

- *Re-factoring based Program Repair applied to Programming Assignments*, `https://www.comp.nus.edu.sg/~abhik/pdf/ASE19.pdf`
- *TBar: Revisiting Template-Based Automated Program Repair*, `https://arxiv.org/pdf/1903.08409.pdf`
- `https://en.wikipedia.org/wiki/Mutation_testing`
- `https://pitest.org/quickstart/mutators/`

Assumptions and Dependencies:

- Inputs: a set of error locations
- Outputs: a set of repair candidates that fix the provided error locations
- Dependencies: This project will have access to an interpreter that can be used to exercise traces through the programs and to observe variable values. Furthermore, it will have access to an implementation of the program alignment (i.e., structural and variable alignment), which can be used to implement the overall repair workflow.

Example: To further illustrate the problem and its solution, please find below a simple example. Imagine the correct reference program is defined as follows:

```
def search(x, seq):
  for i in range(len(seq)):
    if x <= seq[i]:
      return i
return len(seq)
```

and the submitted (incorrect) program looks like follows:

```
def search(e, lst):
  for j in range(len(lst)):
    if e < lst[j]:
      return j
    else:
      return len(lst)
```

Because the reference program and the incorrect program have different control flow structures in their if-statement, their CFG-based representation fail to align. To fix the submitted program, we would first expect a refactored reference program like below to increase the alignment possibility:

```
def search(x, seq):
  for i in range(len(seq)):
    if x <= seq[i]:
      return i
    else:
      pass
return len(seq)
```

We then expect the mutation operator to borrow expressions from refactored program to fix the submitted program, generating a final repair output like:

- *Change* `e < lst[j]` *to* `e <= lst[j]`.
- *Change* `return len(lst)` *to* `pass`.
- *Add* `return len(lst)` *at end of the program.*

4.2 Optimization-based Repair

Overview: As repair we denote the fixing of the submitted program by reusing the existing reference implementation. At this stage, the project assumes that the two programs are present in some CFG-based representation, and that the programs are aligned, i.e., we know which basic block in the reference implementation corresponds to which basic block in the submitted program. Furthermore, we know the mapping(s) between the variables in the basic blocks. Note that there could be multiple variable mappings. In this project, the students are asked to implement a repair approach based an *optimizing* strategy that selects a repair with minimal cost. In general, this approach requires two steps:

1. Generate a set of local repairs for each implementation variable (i.e., its assigned expression).
2. Select a *consistent* subset of local repairs with the smallest cost.

A *local repair* refers to a replacement of an expression in the submitted program with an expression taken form the reference program. Therefore, the goal of overall repair strategy is apply a subset of local repairs so that the submitted program becomes semantically equivalent with the reference program.

A *consistent* repair combination refers to the problem that one cannot combine any local repair possibility, but has to take into account the valid and available variable mappings.

It will be necessary to assign a *cost* value for each local repair. A straightforward approach would be to use the *Levenshtein Distance*[1] to calculate the distance between expressions. More advanced cost metrics can be developed.

The exact optimization strategy can be decided by the students. However, we propose the usage of 0-1 Integer Linear Programming (ILP). Therefore, a concrete sequence of steps could be:

1. Construct a repair search space which enumerates all possible repairs for each error location by matching the submission and reference solution.
2. Develop an algorithm to encode a possible repair into an 0-1 ILP constraint.
3. Apply an ILP solver to find best repair from the ILP constraint.

Complexity: [Coding: *Medium-High*, Theory: *High*, Research: *Low*, HCI: -]

Prerequisite and References: If the students want to follow the ILP-based repair strategy, they need to understand what is ILP program solving and how to apply an ILP solver. The following references can help to prepare for this project:

- `https://en.wikipedia.org/wiki/Integer_programming`
- *Integer Programming*, `https://web.mit.edu/15.053/www/AMP-Chapter-09.pdf`
- `https://developers.google.com/optimization/lp/glop`
- `https://realpython.com/linear-programming-python/#why-is-linear-programming-important`

Assumptions and Dependencies:

- Inputs: a set of error locations, the syntactical alignment of the reference program and the submitted program, variable mapping between the two programs.
- Outputs: a set of repair candidates that fix the provided error locations
- Dependencies: this project will have access to an interpreter that can be used to exercise traces through the programs and to observe variable values. Furthermore it will have access to an ILP solver engine.

[1] `https://en.wikipedia.org/wiki/Levenshtein_distance`

Example: To further illustrate the problem and a potential solution strategy, please find below a simple example. Imagine the correct reference program is defined as follows:

```
 def assign_to_one():
1:    a = 0
2:    b = 1 + a
3:    return b
```

..., and the submitted (incorrect) program looks like follows:

```
 def assign_to_one():
1:    x = 0
2:    y = 5 + x
3:    return y
```

To fix the submitted program, we would *expect* a final repair output like:

Change "y = 5 + x" to "y = 1 + x".

Let's assume that as *input* to our repair strategy, we also receive the set of error location pairs, e.g., $\mathcal{E} \leftarrow \{(1,1)\}$. In this case relatively simple, just location 1. All program lines would be represented in the same basic block in control-flow graph (CFG), hence the single pair of error locations.

Furthermore, we receive the possible variable mappings:
$\mathcal{M} \leftarrow \{m_1 : (a \mapsto x, b \mapsto y),\ m_2 : (a \mapsto y, b \mapsto x)\}$.

For variable mapping $m_1 : (a \mapsto x, b \mapsto y)$, there is local repair:

- $r_1 \leftarrow$ change expression of y to $1 + x$ (with repair cost = 1)

For variable mapping $m_2 : (a \mapsto y, b \mapsto x)$, there are the local repairs:

- $r_2 \leftarrow$ change expression of x to $1 + y$ (with repair cost = 3)
- $r_3 \leftarrow$ change expression of y to 0 (with repair cost = 3)
- $r_4 \leftarrow$ change return expression to x (with repair cost = 1)

Each of the repair is assigned with a repair cost, in this case the *Levenshtein Distance* between the changed code and the original code. Based on the repair search space $\{r_1, r_2, r_3, r_4\}$ we can now encode the 0-1 ILP constraint. For example, as follows:

$$\text{// enforce bijective variable mapping}$$
$$1 * m_1 + 1 * m_2 = 1$$
$$\text{// repair r is chosen iff the bijective mapping is selected}$$
$$-1 * r_1 + 1 * m_1 = 0$$
$$-1 * r_2 + 1 * m_2 = 0$$
$$-1 * r_3 + 1 * m_2 = 0$$
$$-1 * r_4 + 1 * m_2 = 0$$

This ILP constraint can be passed to an ILP solver with the objective to *minimise* $(1 * r_1 + 3 * r_2 + 3 * r_3 + 1 * r_4)$. This would least to the resulting assignment: $r_1 = 1, r_2 = 0, r_3 = 0, r_4 = 0, m_1 = 1, m_2 = 0$, and hence select the repair r_1 as repair with the lowest cost. Therefore, the final output would match the *expected* output from the beginning:

Change "y = 5 + x" to "y = 1 + x".

4.3 Synthesis-based Repair

Overview: As a challenging alternative to the other repair projects, *synthesis-based repair* follows a different approach, and requires two general steps:

1. Specification Inference
2. Program Expression Synthesis

The *Specification Inference* constructs a repair constraint, e.g., by collecting expected values for variables at the identified error location(s). Such a repair constraint can then be used to *synthesize* expression(s) at the identified error location(s) to repair the submitted program. While the repair constraint defines the correctness of the semantics of the expression, the synthesis part also needs the information about how the expression can look like. Therefore, it would be provided with the available syntactical components (i.e., component-based synthesis) or a context-free grammar (i.e., grammar-based synthesis). The task in this project is to develop both, the specification inference and the expression synthesis.

Complexity: [Coding: *Medium*, Theory: *High*, Research: *Medium*, HCI: -]

Prerequisite and References: This project is challenging because it combines several technical aspects and provides several research possibilities, as both, specification inference and program synthesis, are active research areas. The following references can help to prepare for this project:

- *Syntax-guided synthesis*,
  `https://doi.org/10.1109/FMCAD.2013.6679385`
- *Oracle-guided component-based program synthesis*,
  `https://doi.org/10.1145/1806799.1806833`
- *Angelix: scalable multiline program patch synthesis via symbolic analysis*,
  `https://doi.org/10.1145/2884781.2884807`
- *Combinatorial Sketching for Finite Programs Armando*,
  `https://doi.org/10.1145/1168857.1168907`
- *Semantics-Guided Synthesis*,
  `https://doi.org/10.1145/3410258`

Assumptions and Dependencies:

- Inputs: This project requires as input two intermediate program models, representing the reference solution and the submitted program. Furthermore, it takes a set of inputs and the corresponding error locations.
- Outputs: A set of repair candidates that fix the provided error locations.
- Dependencies: This project has access to the interpreter to execute inputs on the program models, as well to an SMT solver engine. More dependencies can be added if necessary.

5 Feedback Generation

5.1 Automated Feedback

Overview: Though the techniques under Topic 4 (*Transforming / Repairing Programs*) are able to generate program patches for an incorrect student submission. It is not considered as a good teaching practice to directly present the fixes as feedback to a student. The student might simply follow the fixes to edit their incorrect submission and submit again, losing the motivation to think thoroughly why his/her submission was wrong. In this project, the students are asked to propose *innovative* feedback mechanisms, which guide the students to understand his/her mistakes comprehensively. Given a repair candidate, this project should automatically generate meaningful, customized feedback for each submission at different levels, including but not limited to (1) analyze the repair candidate and evaluate the difficulty level to fix the incorrect submission; (2) explain the root cause of the incorrect submission and guide the student to relevant material. All reasonable answers will be accepted. In general, the automated feedback require three steps:

- Analyze what information are provided in the repair candidate.
- Propose reasonable feedback mechanisms by using the information in the repair candidates.
- Implement an automated feedback generation system that applies the proposed feedback mechanism.

Complexity: [Coding: *Low*, Theory: *Medium*, Research: *Medium*, HCI: *High*]

Prerequisite and References: Students need to search the literature about what kind of feedback can help students improve their learning efficiency, and combine with the repair candidate that we provide to design unique feedback mechanism to students. The following references can help to prepare for this project:

- *Automated Feedback Generation for Introductory Programming Assignments*, `https://arxiv.org/pdf/1204.1751.pdf`
- *Introductory programming: a systematic literature review*, `https://dl.acm.org/doi/abs/10.1145/3293881.3295779`
- *A feasibility study of using automated program repair for introductory programming assignments*, `https://dl.acm.org/doi/abs/10.1145/3106237.3106262`

Assumptions and Dependencies:

- Inputs: This project requires as input the generated repair candidates.
- Outputs: Textual output that represents the generated feedback.
- Dependencies: This project has no access direct access to other modules of the system, however, they can be granted if necessary.

5.2 Automated Grading

Overview: In recent years, due to the fact that programming skill can help to improve employment outcomes, the demand of computer science education is higher than ever. In university, the phenomenon has led to explosive growth of student enrolment in introductory programming courses. To guarantee the teaching quality, course instructor and human tutors need more effort in assessing students' learning outcome at large scale, one of typical ways is through grading their programming assignments. In this project, students are asked to propose precise automated grading mechanisms for introductory programming assignments and implement such an auto-grading system. Given an incorrect submitted program, the auto-grading system should automatically generate a final grade, the grading metric can include but not limited to (1) analyze the behavior difference between reference program and incorrect program with same input; (2) evaluate the effort required to generate a repair using techniques in Project 4.2 and also evaluate the generated repair's quality. In general, the automated grading require three steps:

- Analyze what information in the intelligent tutoring system (can take from all other projects) can help to assess an incorrect submitted program.
- Propose reasonable automated grading mechanisms by using the information you need.
- Implement an automated grading system with your mechanisms and evaluate it in the dataset.

Complexity: [Coding: *Low*, Theory: *High*, Research: *High*, HCI: *Low*]

Prerequisite and References: Students need to search the literature about how existing automated grading system works and propose their own strategies based on previous work. The following references can help to prepare for this project:

- *Automated Grading of Programming Assignments*, `https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.83.6483&rep=rep1&type=pdf`
- *Automatic Grading of Programming Assignments: An Approach Based on Formal Semantics*, `https://www.cs.tufts.edu/~nr/cs257/archive/autograde-icse-2019.pdf`
- *Automatic Grading of Computer Programs: A Machine Learning Approach*, `https://ieeexplore.ieee.org/document/6784592`
- *Using Latent Semantic Analysis for automated grading programming assignments*, `https://ieeexplore.ieee.org/abstract/document/5995769`
- *Semantic similarity-based grading of student programs*, `https://www.sciencedirect.com/science/article/abs/pii/S0950584906000371`

Assumptions and Dependencies:

- Inputs: Due to its research nature, this exact interface for this project is not defined. However, the students can expect to get as input all available artifacts of the repair system, i.e., the aligned programs in their intermediate model, the identified error locations, and the generated repair candidates.
- Outputs: Textual output that represents the generated grading.
- Dependencies: This project has no access direct access to other modules of the system, however, they can be granted if necessary.