0. Instructions and Notes

- One group per project, call dibs[1] by email. Taken projects will be updated with their title in ~~strikeout~~ in the file, but please be aware of eventual consistency issues - a project may be taken before it's marked as such.
- Projects on this list are **not** ordered by difficulty.
- Even if you pick from this list, you still have to submit a project description document. These are ideas, not specifications.
- If you are an undergraduate, your team's project must be from this list. If everyone on your team is not an undergraduate, please start by browsing this list for inspiration and then trying to come up with your own project.
- Projects can be **implemented** in the language of your choice.
- The language **the tool is for** should be the one in the examples/project description unless you have a very good reason and it's been approved.
- If you are using code from an existing tool/research project:
    - Use it as a module if possible
    - Always credit it
    - Be aware of infectious licenses
- Some of the paper links may require the Technion network or library VPN to access

[1] https://www.merriam-webster.com/words-at-play/word-origin-dibs

1. Grammar-based grep (up to 2 students)

Problem: when using grep on code, it's difficult to find or replace syntactic patterns. For instance, let us assume test code has many assertions of the old JUnit format:

```
assertEquals(6,var.length);
```

The programmer now wishes to switch them over to `assert`, which has the format

```
assert(<boolean expression>);
```

In addition, where the format for `assertEquals` is

```
assertEquals(<expected value>,<actual value>);
```

the testing tools expect `assert` on equality to have the format

```
assert(<actual value>==<expected value>);
```

Using regular grep, in this simple example, the user can capture the two elements expected value and actual value with the following regular expression:

```
"assertEquals\(([^,]+),([^)]+)\);"
```

then use the following substitution

```
"assert($2==$1);"
```

to get the desired result,

```
assert(var.length==6);
```

However, if either value involves a function call with multiple arguments, this will fail. For example, for

```
assertEquals(7,obj.foo());
```

the capturing regex does not match and will need to be made more complex. All the while, syntactically speaking it's quite easy for the programmer to describe:

```
assertEquals(<expression>,<expression>);
```

If both expressions are captured, the same substitution expression will produce the correct result.

PL technology: Language lexing and parsing. Grep operates on text files as a stream of characters. However, code files are better represented as a stream of tokens, and they (or parts of them) can be parsed with the language parser.

Interaction suggestion: the user can represent a regular expression as a combination of tokens and language nonterminals (the specifications of the regex language are up to you), then use it as grep is normally used (or as regex replace in the IDE is normally used).

2. Syntactic diff tool (up to 2 students)

Problem: diff is a line-based tool that compares two text files line-by-line. When used for code, it still maintains this line by line behavior. This means that, for instance, this change:

```
- index[0] = mat + 5*i;
+ index[n-2] = mat + 5*i;
```

will mark the full line has changed, when only the dereferencing has changed. Likewise, this diff is actually empty except for the comments:

```
- foo(1,2,3,4);
+ foo(1, /*first arg*/
       2, /*second arg*/
       3, /*third arg */
       4 /*done*/);
```

Finally, a diff may include dependent changes. For instance, in the diff

```
- for(int c = 1; c <= num; ++c)
+ for(int count = 1; count <= num; ++count)
```

there are three changes of `c` to `count`, but the later ones are *a result* of the first one. Likewise, a file can include two diff chunks, earlier in the file an initialization change:

```
- int c = 0;
+ int count = 0;
```

and later in the file a change in use:

```
- for(c = 1; c <= num; ++c)
+ for(count = 1; count <= num; ++count)
```

where the change in the second diff chunk is *a result of* the first change.

Other possible causes of changes: renaming a function, changing the number of arguments to a function. Notice that in a realistic setting multiple files may be on either side of the diff, and the cause of changes may be cross file boundaries. It's best to initially ignore such diffs.

PL technology: the two versions of the file being "diffed" can be parsed (and possibly even fully compiled using the compiler frontend). AST differencing can then be applied (see useful papers), and finally refactoring identification can find changes that should be tracked throughout the diff.

Interaction suggestion: the user runs a diff command line tool on two files. It should comply with the difftool requirements for git. The command opens a UI (html page is sufficient) that displays the two files side by side, highlighting only the changes. Additions, deletions, and renames should have visual signifiers, and if there is a cause for a change, it should be clearly marked with what the cause is, and should be able to take the user to the source of the change.

Useful papers:

- Fine-grained and Accurate Source Code Differencing: https://hal.archives-ouvertes.fr/hal-01054552/document

3. Unit test shrinking (up to 3 students)

Problem: Programmers often enter real-world bugs into unit tests to help them debug. However, these might not be the most minimal cases for the bug because they stem from real-world values, which makes them harder to debug - they may include longer loops or more complex computations.

For example, let us assume the programmer is exploring a crash in the logs, and locates the crash as happening after calling `myWorkObject.process()` with a message string `"Hello, my name is Elder White\n\tAnd I would like to share with you the most amazing book"`. The programmer then creates a JUnit unit test:

```
@Test public void testCrash() {
      Settings settings = ...//initialize settings object
      WorkObj myWorkObject = new WorkObj(settings);
      int result = myWorkObj.process("Hello, my name is Elder
White\n\tAnd I would like to share with you the most amazing
book");
      assertEquals(Codes.OK,result);
}
```

and runs it to see that it crashes in the same way as the main process: by throwing an `ArrayIndexOutOfBoundsException` - so they are ready to start debugging. Since the function `process` performs many passes on the string before failing, it would take the user a lot of work to find out that it, eventually, fails by not properly handling the tab in the middle of the string in one of the later passes.

In fact, it would have been sufficient for the third line of the test to be

```
int result = myWorkObj.process("\t");
```

in order to reproduce the exact same crash. It would be very helpful to the user to get a unit test minimized to the smallest possible examples reproducing a behavior.

It is interesting to note that:

1. Some constants in the unit test are “true constant”, or should not be minimized. For instance, the settings object is likely initialized with some constants and those need to remain the same.
2. Some lines in the unit test may not be necessary. Minimizing the unit test can include removing them as well.

PL technology: Property-based testing (PBT) such as QuickCheck, ScalaCheck, or FsCheck includes a feature called “shrinking”, which in the case of failure tries to simplify constant values and re-run the property under test until the property no longer fails.

Mutation-based program repair uses mutation operators (i.e., program rewrites that are not equivalence-preserving) to modify a program P into a program P’. These can be used in a genetic algorithm or in a brute-force search with backtracking.

A note about programming language: while you can implement the project in any language, it may be useful to look at the list of languages for which PBT has been implemented, and code for generating a stream of values for shrinking starting for a specific value already exists. A list can be found in: https://en.wikipedia.org/wiki/QuickCheck

Interaction suggestion: Once a failing unit test is created by the user, the user can run the tool (optimally with IDE integration, command line that takes the project and test name is fine to start), which will try to simplify the constants in the test using shrinking, as long as the test failure stays the same. If the test fails on an exception, the exception should also be the same. The tool will then suggest the new unit test with new constants.

Assumptions: You can assume that none of the code under test performs functionality that has side-effects on the environment (i.e., write to files, send/recieve packets, etc). This assumption can be partially overcome by sandboxing, but is irrelevant initially.

Useful papers:

- Shrinking and Showing Functions: https://dl.acm.org/doi/abs/10.1145/2364506.2364516
- Using Mutation to Automatically Suggest Fixes for Faulty Programs: https://ieeexplore.ieee.org/document/5477098

4. Alternative code schemas (up to 3 students)

Problem: programmers new to a language are often unfamiliar with the idiomatic ways to do things in that language. For example, a programmer coming from C might find themselves writing this Java code:

```
for(int i = 0; i < arr.length; ++i) {
    String s = arr[i];
    // more code
}
```

or this python code:

```
for i in range(len(arr)):
    s = arr[i]
    # more code
```

instead of the more idiomatic

```
for (String s: arr)
```

or

```
for s in arr:
```

Suggesting an equivalent but more idiomatic way to perform such operations would help improve novice code quality and expose programmers to “the right way” to do things earlier on.

Other such suggestions can be for variable declarations, emptiness-checking with size/length of a collection instead of truthy/falsy values in python or `Collection.isEmpty` in Java, appending to a list instead of using a comprehension, etc. Your project should be able to suggest at least three kinds of schema changes.

PL technology: refactoring (equivalence preserving) transformations on the AST. If a refactoring transformation from a less idiomatic to a more idiomatic schema/template is possible, it can be suggested to the user.

An experimental feature (i.e., it may not be useful! If you choose to take this on, you may want to evaluate it separately!) can be to still suggest transformations that are not entirely equivalence preserving but are still likely.

Interaction suggestion: in the IDE (Eclipse and VSCode are easiest to develop for), periodically scan the code and suggest likely transformations as a recommendation. If you are also suggesting non-equivalent transformations, this must be identified!

5. Additional test generation (up to 2 students)

Problem: unit test coverage might be poor simply because the programmers have not tried more values. For example if the code:

```
public static void foo(int a, String b) {
    if (a > 0)
        return b.substring(a);
    else
        return b.substring(b.length + a);
}
```

is tested with the JUnit test:

```
@Test public void testFoo() {
    foo(12,"test me");
    //assert something
}
```

it only tests 50% of the paths in `foo`. Moreover, if we also consider the code inside `String.substring`, the coverage is even lower.

It may be possible to transform `testFoo` into additional tests that increase coverage, then bring them to the attention of the user.

PL technology: concolic testing and white-box fuzzing are both concerned with generating additional inputs to increase coverage. Both use a concrete input and SMT solving in order to find a *different* input that will take a different path in the program.

A note about programming language: while this suggests doing this project in Java, it is also possible to implement this in python using the techniques shown in The Fuzzing Book, given strong type assumptions.

Interaction suggestion: as an initial interaction model, the tool should accept a function, a line number in that function (i.e., a statement), and an existing test for that function, and return a new unit test that reaches the desired line. Given this, the next milestone will be, given a function, its tests and a threshold, to create additional tests until coverage passes that threshold. An open question is what to do with the assertions in the test - both because of how they impact the coverage metrics, and because a decision about their result needs to be made.

Useful papers:

- CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools: https://link.springer.com/content/pdf/10.1007/11817963_38.pdf
- Automated Whitebox Fuzz Testing: https://patricegodefroid.github.io/public_psfiles/ndss2008.pdf

6. Stitching in new parameters (up to 2 students)

Problem: a programmer needs to access a resource in a function, for instance this one:

```
private int foo(String msg, Settings settings) {
                //…
}
```

They then add the resource to the function prototype and use it, like so:

```
private int foo(String msg, Settings settings, Logger logger) {
                //…
                logger.log(//…
}
```

and then begin stitching the codebase around the function call. The programmer might find they've gone through dozens of files upward before they find the availability of a `Logger` and maybe even have to change an interface method, which means modifying its instances larger still.

However, once the programmer submits their commit for review, an expert on the codebase points out that the entire change could have been avoided since it's possible to access the logger through the existing parameters:

```
settings.getIOFactory().getLogger()
```

and reject what must have been hours of unpleasant work.

PL Technology: the synthesizer SyPet is a type-based synthesizer. It creates a network of types available in a system, and searches for a path from the existing input types to output types. Specifically in this case, it accepts the specification: input: `Settings`, output: `Logger` and finds as a result program the line above.

Interaction suggestion: there are two possible interactions for this tool. First, there is the option of creating a lint-time tool where a commit such as the one described above is analyzed and the problem is discovered before the codebase expert must waste their time looking through a huge commit. However, this does not solve the problem of a programmer wasting a large amount of work, so a second option is an IDE plugin that runs this (does not have to be automatically) for a prototype change before any additional work is done. In order to work on code that still compiles, one can ask "should I add" with a new argument for the function rather than simply adding it.

Useful papers:

- Component-Based Synthesis for Complex APIs: https://www.cs.utexas.edu/~isil/sypet-popl17.pdf
- Jungloid mining: helping to navigate the API jungle: https://dl.acm.org/doi/10.1145/1064978.1065018

7. Local synthesis specifications in code (up to 3 students)

Problem: Currently using program synthesis means either copying the specifications to other files to launch a standalone synthesizer, or using IDE-based synthesizers where specifications are entered into a new window and disappear when the synthesis task returns (with or without code as a result). This means there is no documentation for any examples entered for synthesis, of the line of code being the result of synthesis, and also means that if the code is then manually modified to contradict one of the examples, no one will know.

PL technology: PBE synthesizer. Any program synthesis with example specifications will be useful here. Some examples are suggested below.

Interaction suggestion: Instead of entering specifications into a new window, specifications will be written in a specially formatted comment in the code file, for example (though not necessarily like so):

```
/*@@
input: {x=2,z="abc"}
output: {y="abcabc"}
@@*/
```

or:

```
#@@
#input: {x=2,z="abc"}
#output: {y="abcabc"}
#@@
```

depending the language this is implemented for (you can target any language you choose). An IDE like IntelliJ or Eclipse can recognize special comments, at which point it will suggest running a synthesizer. If the synthesizer returns a program, it will insert it into the code immediately after the comment, with a delimiter:

```
y = z*x
#@@ END @@
```

This also allows checking the code between the synthesis specification and the end-of-synthesized-code indicator as a very local form of unit testing: if the code in the block does not meet the requirements of the output, a warning can be indicated in the IDE.

Useful papers:

- transit: Specifying Protocols with Concolic Snippets: https://viterbi-web.usc.edu/~jdeshmuk/Papers/pldi13.pdf
- Accelerating Search-Based Program Synthesis using Learned Probabilistic Models: https://www.cis.upenn.edu/~alur/PLDI18.pdf
- Recursive Program Synthesis: https://www.microsoft.com/en-us/research/publication/recursive-program-synthesis/

- Perfect is the Enemy of Good: Best-Effort Program Synthesis: https://hilap.cswp.cs.technion.ac.il/wp-content/uploads/sites/11/2021/06/ecoop2020.pdf
- FrAngel: component-based synthesis with control structures: https://dl.acm.org/doi/10.1145/3290386
- LooPy: Interactive Program Synthesis with Control Structures: https://weirdmachine.me/papers/2021_loopy.pdf