Automated Debugging
WS 2022/2023

Prof. Dr. Andreas Zeller
Paul Zhu
Marius Smytzek

Exercise 9 (10 Points)

Due: 17. February 2023

Submit your solutions as a Zip file on your status page in the CMS.

We will provide you a structure to submit your solutions where each task has a dedicated file. You can add new files and scripts if you want, but you must not delete any provided ones. You can verify whether your submission is valid by calling python3.

```
python3 verify.py
```

The output provides an overview if a required file, variable, or function is missing and if a function pattern was altered. If you do not follow this structure or change it, we cannot evaluate your submission. A non evaluable exercise will result in 0 points, so make sure to verify your work before submitting it. Note that the script does not reveal if your solutions are correct.

Exercise 9-1: A Good Test Suite is Half Done (4 Points)

Automated repair helps not only developers, but also novices to programming. Let's consider a programming assignment from a Python beginner's course called *sequential search*: implement a Python function `def search(x: int, seq: List[int]) -> int` that outputs how many numbers in a sorted number sequence `seq` are smaller than `x`.

The following are two buggy implementations submitted by the students (included in `exercise_1.py`):

In [1]:
```python
from typing import List

def search_buggy_1(x: int, seq: List[int]) -> int:
    for i in range(len(seq)):
        if x <= seq[i]:
            return i
        else:
            return len(seq)

def search_buggy_2(x: int, seq: List[int]) -> int:
    if x < seq[0]:
        return 0
    elif x > seq[-1]:
        return len(seq)
    for i, elem in enumerate(seq):
        if x <= elem:
            return i
```

To help them repair their buggy submissions using the `Repairer` introduced in the lecture, one must, first of all, specify a "good" test suite -- that's exact what you are going to do for this exercise. Your testcases must meet the following requirements:

1. They follow the expected semantics of `search`: we provide a reference implementation `search_correct` in `exercise_1.py` and you should make sure that `search_correct` passes all your testcases.
2. They are suitable for statistical debugging: the test suite should contain both passing and failing testcases for the two buggy functions, otherwise statistical debugging does not work on error localization.
3. They allow the `Repairer` to successfully repair the two buggy functions: the fixed program with fitness 1.0 (if found) will be tested against more hidden testcases (think of them as the testcases used in the course online

judge system). NOTE: to mitigate the effect of randomness, `Repairer` will attempt three times for finding a fix with fitness 1.0.
4. The total number of testcases do **not exceed** 10.

Put your testcases in the `TESTCASES` list in `exercise_1.py`. Each testcase is a tuple consisting of an input (which itself is a tuple of `x` and `seq`) and the corresponding expected output -- i.e., `Tuple[Tuple[int, List[int]], int]`. Run `python3 exercise_1.py` to check if the `TESTCASES` meet most of the above requirements, except testing the fixed program against the hidden (that's what *hidden* means!) testcases. All you need to do is to specify the `TESTCASES`; don't modify the other code in `exercise_1.py`.

Exercise 9-2: Conditions, Conditions (6 Points)

Many logical errors of software development comes from control conditions. In the lecture we've already learned the `ConditionMutator`. This is still insufficient for many complicated repairing tasks. In this exercise, we are going to build a more expressive mutator step-by-step.

a. Comparator Mutation (1 Point)

There are times when developers use a wrong comparison operator in a condition, like using `n > 0` (error-prone) instead of `n >= 0` (correct) to test the nonnegativity of `n`. Note that such errors cannot be fixed by simply inverting the condition: the inverse of `n > 0` is identical to `n <= 0`; but meanwhile, `n >= 0` and `n < 0` are also candidates that should be taken into account in mutating.

In this part, your task is to implement a function that computes the set of mutation candidates for a `ast.Compare` condition according to the following table (where `x` and `y` are meta-variables):

```
Condition       Candidates
-----------------------------------
x <= y          x < y, x >= y
x < y           x <= y, x > y
x >= y          x > y, x <= y
x > y           x >= y, x < y
```

For all other conditions, return empty set.

Note:

- Inverses are ignored because they can be produced by the `not` mutation in `ConditionMutator`.
- Assume there is **only one comparator** for each input `ast.Compare`; we don't consider nested comparators like `x < y < z` in the entire exercise.

In [6]:
```
from typing import Set

def comparator_mutation_candidates(node: ast.Compare) -> Set[ast.Compare]:
    pass
```

Implement this function in `exercise_2.py`. Sample tests are provided (run them by `python3 exercise_2.py`).

b. Condition Generation (3 Points)

The `ConditionMutator` supports constructing a new condition via Boolean composition -- but the other operand must also directly come from the source, which makes it restrictive. A more general and expressive approach could be, the other operand is generated from Boolean compositions of any **atomic** conditions (i.e., does not contain any logical connectives `and`, `or`, `not`) extracted from the source. In this way, we are allowed to transform the condition `c == '<'` of `remove_html_markup` into `c == '<' and (not quote and tag)` where its right-hand side `not quote and tag` does not exist in the source, but built from the atomic conditions `quote` and `tag` extracted from the source.

To realize this idea, first you need to implement a function in `exercise_2.py` that collects all the atomic conditions -- limited to control conditions, i.e., the ones returned by `all_conditions` -- of the input AST.

In [9]:
```python
def collect_atomic_conditions(tree: ast.AST) -> List[ast.expr]:
    pass
```

Next, build a `ConditionGenerator` class in `exercise_2.py`:

In [10]:
```python
class ConditionGenerator:
    """Generate conditions built from the atomic conditions in an AST"""

    def __init__(self, tree: ast.AST) -> None:
        self.atomic_conditions = collect_atomic_conditions(tree)

    def sample(self) -> ast.expr:
        """Return a random condition."""
        pass
```

whose `sample` method yields a random condition belonging to the grammar below:

```
<start>  ::= <cond>
<cond>   ::= <simple> and <simple> | <simple> or <simple> | <simple>
<simple> ::= <term> | not <term>
<term>   ::= <atomic> | comparator mutation candidates of an <atomic>
<atomic> ::= all atomic conditions
```

Note:

- A `<term>` could be either an atomic condition or a comparator mutation candidate of an atomic condition if it is `ast.Compare` and its operator is in `<=, <, >=, >`.
- For simplicity, the operands for `and` and `or` do not recursively contain `and` nor `or`, but only (at most one) `not`.
- Follow this strategy to enumerate a random condition:
    - when constructing a `<cond>`, use the three production alternatives (`and`, `or`, just a `<simple>`) with equal probability;
    - when constructing a `<simple>`, use the two production alternatives (with and without `not`) with equal probability;
    - when constructing a `<term>`, first choose an atomic condition (with uniform distribution) `c`; if `c` has any comparator mutation candidates, choose one of the candidates plus `c` itself with equal probability, or else we can only choose `c`

  Using this strategy, we try to equalize the probabilities of all possible conditions, which makes it easier for the helper function `should_sample` (see `exercise_2.py`) to test if your `ConditionGenerator` can indeed produce a set of expected conditions.

Implement the `sample` method in `exercise_2.py` (you are free to add helper methods, but do not change the signature of `__init__` and `sample` in class `ConditionGenerator`). Sample tests are provided (run them by `python3 exercise_2.py`).

c. Finally, the Mutator (2 Points)

Now it's time to use the `ConditionGenerator` you implemented in (b) to build a more powerful mutator for condition:

In [13]:
```python
import copy
from typing import Any, Callable
from debuggingbook.Repairer import StatementMutator

class ConditionMutator(StatementMutator):
    """Mutate conditions in an AST"""

    def __init__(self, *args: Any, **kwargs: Any) -> None:
        """Constructor. Arguments are as with `StatementMutator` constructor."""
        super().__init__(*args, **kwargs)

    def mutate(self, tree: ast.AST) -> ast.AST:
        self.condition_generator = ConditionGenerator(tree)
```

```
        return super().mutate(tree)

    def choose_op(self) -> Callable:
        return self.swap # always do swapping

    def choose_bool_op(self) -> str:
        return random.choice(['set', 'not', 'and', 'or'])

    def swap(self, node: ast.AST) -> ast.AST:
        """Replace `node` condition by a constructed condition"""
        if not hasattr(node, 'test'):
            return node  # do not mutate nodes other than conditional statements

        node = cast(ast.If, node)
        new_node = copy.deepcopy(node)

        # TODO: YOUR CODE HERE

        return new_node
```

The actual mutation takes place in the `swap()` method. If the node to be replaced has a `test` attribute (i.e. a controlling predicate), then we pick a random condition `cond` generated by `condition_generator` (which is initialized in the `mutate` method) and randomly choose from:

- `set` : We change `test` to `cond`.
- `not` : We invert `test`.
- `and` : We replace `test` by `cond and test`.
- `or` : We replace `test` by `cond or test`.

Remarks:

- For the exercise only, we intentionally override `choose_op` method to always perform the `swap` operation and also prevent the `swap` on nodes other than conditional statements, which makes this `ConditionMutator` mutates control conditions merely.
- In other words, this new `ConditionMutator` behaves similarly to the one we've seen in the lecture, but instead of choosing `cond` from the source, now we generate it by a `ConditionGenerator`.

Implement the `swap` method in `exercise_2.py` (you are free to add helper methods, but do not change the signatures of all the other methods already defined in `ConditionMutator`). You will receive full marks for this part if the `Repairer` that uses your `ConditionMutator` can successfully (i.e., fitness = 1.0) fix two buggy programs `sort_by_second_descending` and `multiple_2_3_5` provided in `exercise_2.py` (run `python3 exercise_2.py` to check them).

The `sort_by_second_descending` function is error-prone because it sorts, by the second element, in an ascending but not descending order. To fix it, we should change the `lst[i][1] > lst[j][1]` condition into `lst[i][1] < lst[j][1]`.

In [14]:
```
def sort_by_second_descending(lst: List[Tuple[str, int]]) -> List[Tuple[str, int]]:
    for i in range(0, len(lst) - 1):
        for j in range(i + 1, len(lst)):
            if lst[i][1] > lst[j][1]:
                tmp = lst[i]
                lst[i] = lst[j]
                lst[j] = tmp

    return lst
```

The `multiple_2_3_5` function expects to test if `n` is a multiple of 2, 3, and 5. But this error-prone version only tests if `n` is a multiple of 5. To fix it, we should strengthen the condition `n % 5 == 0` into `(n % 2 == 0 and n % 3 == 0) and n % 5 == 0`, where the added condition `n % 2 == 0 and n % 3 == 0` needs to be synthesized from the atomic conditions `n % 2 == 0` and `n % 3 == 0`.

In [15]:
```
def multiple_2_3_5(n: int) -> bool:
    if n % 2 == 0:
        pass

    if n % 3 == 0:
        pass
```

```
    if n % 5 == 0:
        return True
    else:
        return False
```