Software Testing, Quality Assurance and Maintenance Winter 2026

Lecture 11 — February 9, 2026

Patrick Lam *version 1*

I write scientific content in LaTeX, which is somewhat user-unfriendly. If one posts in the relevant StackExchange with a problem, a Minimal Working Example (MWE) is pretty much required[1]. We need to reproduce a bug before we can fix it—hence a *working example*. To the point of this lecture, a *minimal* working example saves a lot of time for the person who is charged with fixing the bug.

The relevant *Fuzzing Book* content is at `https://www.fuzzingbook.org/html/Reducer.html`.

Specifically in our context of fuzzing: fuzzers produce potentially large inputs. Often, the created input contains more than is needed to reproduce a bug, and that makes it hard, as a human oracle, to understand what is going on.

In this lecture, we show how to *reduce* a failing input—that is, "to identify those circumstances of a failure that are relevant for the failure to occur, and to *omit* (if possible) those parts that are not". The *Fuzzing Book* quotes Kernighan and Pike in *The Practice of Programming*:

> For every circumstance of the problem, check whether it is relevant for the problem to occur. If it is not, remove it from the problem report or the test case in question.

The *Fuzzing Book* provides an example, where they have obscured the problem, adding some mystery to our lives.

```python
class MysteryRunner(Runner):
    def run(self, inp: str) -> Tuple[str, Outcome]:
        x = inp.find(chr(0o17 + 0o31))
        y = inp.find(chr(0o27 + 0o22))
        if x >= 0 and y >= 0 and x < y:
            return (inp, Runner.FAIL)
        else:
            return (inp, Runner.PASS)
```

This `Runner` fails on some inputs. At this point, we have a number of fuzzing techniques at our disposal, but we can use plain old `RandomFuzzer` to find a failure.

```python
def fuzz_mystery_runner():
    mystery = MysteryRunner()
    random_fuzzer = RandomFuzzer()
    while True:
        inp = random_fuzzer.fuzz()
        result, outcome = mystery.run(inp)
        if outcome == mystery.FAIL:
            break
    print (result)
```

OK, so we do that. It works—or fails, actually. (I tried it manually and it took 6 tries to get a failing input.)

[1] Discussion: `https://tex.meta.stackexchange.com/questions/6255/why-does-tex-require-such-elaborate-mwes`

```
$ python3 mystery_runner.py
(%*50  1)-&7,;49:4?%:43*(-.
```

But the cause of the failure is not exactly clear from this input.

Manual Input Reduction

Before we write some code to do it, let's see how we can reduce an input manually. Kernighan and Pike continue by suggesting a divide and conquer process:

> Proceed by binary search. Throw away half the input and see if the output is still wrong; if not, go back to the previous state and discard the other half of the input.

Does this work?

```
>>> from mystery_runner import *
>>> failing_input = "(%*50  1)-&7,;49:4?%:43*(-."
>>> mystery = MysteryRunner()
>>> mystery.run(failing_input)
('(%*50  1)-&7,;49:4?%:43*(-.', 'FAIL')
>>> half_length = len(failing_input) // 2 # integer division
>>> first_half = failing_input[:half_length]
>>> mystery.run(first_half)
('(%*50  1)-&7,', 'FAIL')
```

That's progress. We now have a string that's half as long as the original and still triggers the failure. Let's try the same trick again, on the first half.

```
>>> quarter_length = len(first_half) // 2
>>> first_quarter = first_half[quarter_length:]
>>> mystery.run(first_quarter)
(' 1)-&7,', 'PASS')
>>> second_quarter = first_half[:quarter_length]
>>> mystery.run(second_quarter)
('(%*50 ', 'PASS')
```

Halving doesn't quite work this time. We need both the first quarter and the second quarter to trigger the failure—looking at the code, it's looking for two characters, but the characters aren't in the same quarter in our test case.

Delta Debugging

There are other ways to do binary searches. What we tried above was directly searching for the offending part of the input, but that didn't work. *Delta debugging* is another way. The insight here is to try to *remove* smaller and smaller parts of the input, and see whether the input still triggers the failure. Contrast that to trying to run on smaller and smaller parts of the input. Intuitively, it's more likely that removing parts keeps the input still-broken.

Let's see an example of one step of delta debugging: we next *remove* quarters of our failing input. First, the first quarter.

```
>>> quarter_length=len(failing_input)//4
>>> input_without_first_quarter=failing_input[quarter_length:]
>>> mystery.run(input_without_first_quarter)
(' 1)-&7,;49:4?%:43*(-.', 'PASS')
```

Because we're looking for a failure, we can see that we have to keep the first quarter to get the failure. Similarly, we can try to remove the second quarter.

```
>>> input_without_second_quarter=failing_input[:quarter_length]+failing_input[
                                                quarter_length*2:]
>>> mystery.run(input_without_second_quarter)
('(%*50 ,;49:4?%:43*(-.', 'PASS')
```

Again, removing the second quarter doesn't trigger the failure. From earlier, we would expect that we can remove the third and fourth quarters, so let's do that, in keeping with running an algorithm.

```
>>> input_without_third_quarter=failing_input[:quarter_length*2]+failing_input[
                                                quarter_length*3:]
>>> mystery.run(input_without_third_quarter)
('(%*50  1)-&7?%:43*(-.', 'FAIL')
```

Indeed, we can remove the third quarter. What about the fourth quarter?

```
>>> input_without_fourth_quarter=failing_input[:quarter_length*3]
>>> mystery.run(input_without_fourth_quarter)
('(%*50  1)-&7,;49:4', 'FAIL')
```

At some level, we're no further ahead yet than before. But the approach is different: there is a clear next step, which is to remove eighths from the first failing input we encountered.

The actual algorithm isn't quite like that, but it's close. The *Fuzzing Book* includes a `Reducer` base class.

```
class Reducer:
    """Base class for reducers."""

    def __init__(self, runner: Runner, log_test: bool = False) -> None:
        """Attach reducer to the given `runner`"""
        self.runner = runner
        self.log_test = log_test
        self.reset()
```

and an abstract `reduce` implementation. Also `test`.

There is also a `CachingReducer` which remembers what has been previously tested.

```
class CachingReducer(Reducer):
    def test(self, inp):
        if inp in self.cache:
            return self.cache[inp]

        outcome = super().test(inp)
        self.cache[inp] = outcome
        return outcome
```

The crux is the `DeltaDebuggingReducer`:

```python
class DeltaDebuggingReducer(CachingReducer):
    """Reduce inputs using delta debugging."""

    def reduce(self, inp: str) -> str:
        """Reduce input ‘inp‘ using delta debugging. Return reduced input."""

        self.reset()
        assert self.test(inp) != Runner.PASS

        n = 2     # Initial granularity
        while len(inp) >= 2:
            start = 0.0
            subset_length = len(inp) / n
            some_complement_is_failing = False

            while start < len(inp):
                complement = inp[:int(start)] + \
                    inp[int(start + subset_length):]

                if self.test(complement) == Runner.FAIL:
                    inp = complement
                    n = max(n - 1, 2)
                    some_complement_is_failing = True
                    break

                start += subset_length

            if not some_complement_is_failing:
                if n == len(inp):
                    break
                n = min(n * 2, len(inp))

        return inp
```

It's not actually halving the size every time—it removes a chunk of size $1/n$, doubling n after running through all the chunks, but it decreases n by 1 when there is a test failure.

One can run the delta debugger:

```python
dd_reducer = DeltaDebuggingReducer(mystery, log_test=True)
dd_reducer.reduce(failing_input)
```

and there is an example run in the *Fuzzing Book*, which I'll show excerpts from:

```
Test #1 ' 7:,>((/$$-/->.;.=;(.%!:50#7*8=$&&=$9!%6(4=&69\':\'<3+0-3.24#7=!&60)2/+";+<7+1<2!4$>92+$1<(
Test #2 '\'<3+0-3.24#7=!&60)2/+";+<7+1<2!4$>92+$1<(3%&5\'\'>#' 49 PASS
Test #3 " 7:,>((/$$-/->.;.=;(.%!:50#7*8=$&&=$9!%6(4=&69':" 48 PASS
Test #4 '50#7*8=$&&=$9!%6(4=&69\':\'<3+0-3.24#7=!&60)2/+";+<7+1<2!4$>92+$1<(3%&5\'\'>#' 73 FAIL
Test #5 "50#7*8=$&&=$9!%6(4=&69':<7+1<2!4$>92+$1<(3%&5''>#" 49 PASS
Test #6 '50#7*8=$&&=$9!%6(4=&69\':\'<3+0-3.24#7=!&60)2/+";+' 48 FAIL
...
Test #23 '(460)' 5 FAIL
Test #24 '460)' 4 PASS
```

```
Test #25 '(0)' 3 FAIL
Test #26 '0)' 2 PASS
Test #27 '(' 1 PASS
Test #28 '()' 2 FAIL
Test #29 ')' 1 PASS
'()'
```

I wouldn't want to do this manually on this test input. Since it's a random input it's harder to understand than a human-generated one, but, assuming that the system is deterministic, we can run the algorithm and get the answer. We also assume that test cases can run quickly enough that we can afford dozens of iterations. These are the same conditions as for fuzzing to work well.

In this case, the answer is that the system fails on an input with a `(` and then a `)`.

Delta debugging yields a 1-minimal test case: removing any character is guaranteed to not fail. In the example, we see that the single-paren cases pass. This is a local minimum: in principle, there might be some other smaller test case that one would reach with different choices, though there isn't in this case.

The *Fuzzing Book* points out the following advantages of reduced test cases:

- reduces cognitive load for the programmer: no irrelevant details, easier to understand what's happening.
- easier to communicate: we can say "`MysteryRunner` fails on "`()`"" rather than "`MysteryRunner` fails on 4100-character input (attached)" (or worse, not attached).
- helps identifying duplicates (to some extent—assuming that a failure has a single cause).

In terms of efficiency, delta debugging is best-case $O(\log n)$ and worst-case $O(n^2)$.

Note also that the `DeltaDebugging` implementation checks that the initial test case does fail.

Grammar-Based Input Reduction

Since we've talked about grammars and fuzzing, we should talk about another application of grammars—for smarter test case reduction. The algorithm is Grammar-Based Reduction (GRABR).

As a preview: if we have a grammar for an input language, we can also use it to reduce test cases. Conversely, a language that is generated by a grammar may not work well with naive delta debugging.

Recall that we've said that expressions are a key use case for grammars. Let's try reducing an expression with our delta debugging reducer.

```
>>> expr_input = "1 + (2 * 3)"
>>> dd_reducer = DeltaDebuggingReducer(mystery, log_test=True)
>>> dd_reducer.reduce(expr_input)
```

This does work, but almost all of the substrings aren't valid expressions.

```
Test #2 '2 * 3)' 6 PASS
Test #3 '1 + (' 5 PASS
...
Test #13 '()' 2 FAIL
Test #14 ')' 1 PASS
Test #15 '(' 1 PASS
```

If we were working with some program which could only process valid expressions, we wouldn't get much from delta debugging. The example we'll show is contrived, but the actual situation is not—imagine that the error comes up beyond the parsing phase.

We simulate such a program by parsing the expression and returning `UNRESOLVED` for cases which don't parse. (The *Fuzzing Book* talks more about parsing, but that is definitely out of context for this course.)

```python
class EvalMysteryRunner(MysteryRunner):
  def __init__(self) -> None:
      self.parser = EarleyParser(EXPR_GRAMMAR)

  def run(self, inp: str) -> Tuple[str, Outcome]:
      try:
          tree, *_ = self.parser.parse(inp)
      except SyntaxError:
          return (inp, Runner.UNRESOLVED)
      return super().run(inp)
```

Using this runner, we see that delta debugging "utterly fails", as the *Fuzzing Book* puts it. Zero of the 20 attempted reductions work, because they all don't parse successfully.

```
>>> expr_input = "1 + (2 * 3)"
>>> dd_reducer = DeltaDebuggingReducer(eval_mystery, log_test=True)
... dd_reducer.reduce(expr_input)
Test #1 '1 + (2 * 3)' 11 FAIL
Test #2 '2 * 3)' 6 UNRESOLVED
Test #3 '1 + (' 5 UNRESOLVED
...
Test #20 '1 + (2 * )' 10 UNRESOLVED
Test #21 '1 + (2 * 3' 10 UNRESOLVED
'1 + (2 * 3)'
```

So we are left with the original input, unreduced.

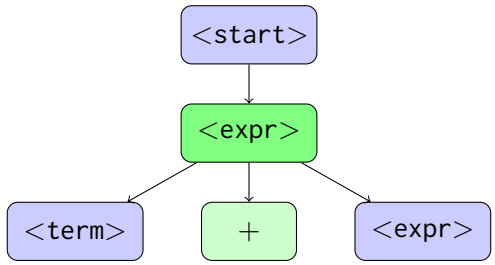
The implementation of delta debugging that we see here doesn't work when there are validity constraints on the inputs—in this case, that the inputs have to parse properly. While it was useful to have invalid inputs for fuzzing, it is not useful in the context of reducing inputs: we want to give the runner something that it can work with, at least the vast majority of the time.

Implementing Grammar-Based Reduction

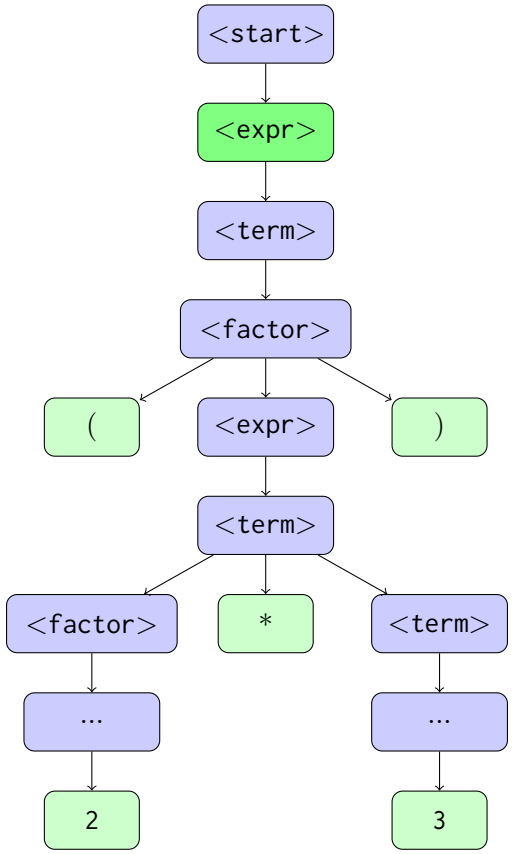
When we talked about generating inputs from grammars, we moved from string manipulations to tree manipulations. Now for reduction, we parse the input and then we manipulate the derivation tree. Specifically, we carry out the reduction by replacing subtrees with smaller subtrees. This means that the tree remains syntactically valid throughout.

Approach 1: Substitution by Subtrees. Specifically, we "substitute subtrees by smaller subtrees of the same type." These subtrees can either come from the initial tree, or by applying different alternatives. Let's see an example.

The derivation trees are taller than I'd like to show here, so I'll show excerpts. Once again, we are working with `1 + (2 * 3)`, and here is the top of the relevant derivation tree.



This tree has the `<expr> (2 * 3)` as a subtree, so we can swap the darker green `<expr>` we see in the above figure with the subtree for `(2 * 3)`. Here is a tree with that substitution carried out.



The relevant substituting code is:

```
import copy
expr_input = "1 + (2 * 3)"
derivation_tree, *_ = EarleyParser(EXPR_GRAMMAR).parse(expr_input)
new_derivation_tree = copy.deepcopy(derivation_tree)
sub_expr_tree = new_derivation_tree[1][0][1][2]
new_derivation_tree[1][0] = sub_expr_tree
all_terminals(new_derivation_tree)
```

If we have the constraint that we only replace <expr> nodes by <expr> nodes that are already in the tree, there is only one more substitution we can do for the top <expr>—the one that strips the parentheses to yield unparenthesized 2*3. Replacing <expr> nodes by non-<expr> nodes will usually violate the grammar.

Approach 2: Simplifying by Replacing Subtrees. Instead of changing like-for-like, we can change a subtree with another alternative that is allowed by the grammar. For instance, in the tree above, we have

```
<term> ::= <factor> * <term>
```

but, per the grammar, a <term> can also be

```
<term> ::= <factor>
```

and we can modify the tree as follows:

```
# term_tree gets "2 * 3"
term_tree = new_derivation_tree[1][0][1][0][1][0][1][1][1][0]
# shorter_term_tree gets "3"
shorter_term_tree = term_tree[1][2]
new_derivation_tree[1][0][1][0][1][0][1][1][1][0] = shorter_term_tree
all_terminals(new_derivation_tree)
# new_derivation_tree is "(3)"
```

to yield a smaller tree, one with just a parenthesized integer (3).

Our goal, then, will be to replace derivation subtrees by smaller subtrees, and to also replace alternatives with smaller subtrees, thus simplifying the input. Especially for structured inputs this should work much better. But: which strategy when?

Implementation talk. Once again, I'm not going to present all of the implementation in the *Fuzzing Book*. But there is a GrammarReducer.

```
class GrammarReducer(CachingReducer):
    def __init__(self, runner: Runner, parser: Parser, *,
                 log_test: bool = False, log_reduce: bool = False):
```

This class takes a Runner (to test inputs) and a Parser (to create derivation trees from strings) and provides a couple of logging options.

There are also some helper functions; `tree_list_to_string()` creates a string from a list of derivation trees; `possible_combinations()` returns the Cartesian product $\ell_1 \times \ell_2 \times \cdots$ of the lists $\ell_1, \ell_2, \ldots$ it receives; `number_of_nodes()` and `max_height()` count properties of trees.

As for the strategy implementations, `subtrees_with_symbol()` is a straightforward tree traversal, not shown.

```
    def subtrees_with_symbol(self, tree: DerivationTree,
                             symbol: str, depth: int = -1,
                             ignore_root: bool = True) -> List[DerivationTree]:
```

It searches `tree` and returns a list of subtrees that have `symbol` at their root. We can try it out:

```
>>> expr_input = "1 + (2 * 3)"
>>> derivation_tree, *_ = EarleyParser(EXPR_GRAMMAR).parse(expr_input)
>>> grammar_reducer = GrammarReducer(
      mystery,
      EarleyParser(EXPR_GRAMMAR),
      log_reduce=True)
>>> [all_terminals(t) for t in grammar_reducer.subtrees_with_symbol(
      derivation_tree, "<term>")]
['1', '(2 * 3)', '2 * 3', '3']
```

Because `subtrees_with_symbol` tells us all subtrees that have a given symbol as root, we can use it to look for all instances of that symbol, which is most of the work for the substitution strategy and also useful for finding replacements to simplify with.

The other strategy is found in `alternate_reductions()`. If we have a `<x>`, and `<x>` is defined as Y or Z, then we construct Ys and Zs using candidates in the tree, and return the shortest Y and the shortest Z that we can construct.

This function can print all combinations making up `<x>`. Here we print all alternate reductions for `<term>` in our derivation tree.

```
>>> grammar_reducer.try_all_combinations = True
>>> print([all_terminals(t)
       for t in grammar_reducer.alternate_reductions(derivation_tree, "<term>")])
['1', '2', '3', '1 * 1', '1 * 3', '2 * 1', '2 * 3', '3 * 1', '3 * 3', '(2 * 3)', '1 *
                                          2 * 3', '2 * 2 * 3', '3 * 2 * 3', '1 * (2 *
                                           3)', '(2 * 3) * 1', '(2 * 3) * 3', '2 * (2
                                           * 3)', '3 * (2 * 3)']
```

This shows all possible `<digit>`s which are in the tree, as well as the alternative `<factor> * <term>`, where it constructs `<factor>` and `<term>` using components already in the derivation tree.

Without the `try_all_combinations` flag, it just returns the shortest `<factor>` and the shortest `<factor> * <term>`. There is no `<factor> / <term>` in the tree.

```
>>> grammar_reducer.try_all_combinations = False
>>> print([all_terminals(t)
       for t in grammar_reducer.alternate_reductions(derivation_tree, "<term>")])
['1', '1 * 1']
```

Combining Strategies. We just use both strategies and deduplicate. Despite what the comment in the code says, it's not all expansion alternatives, unless we set `try_all_combinations` to `True`.

```
    def symbol_reductions(self, tree: DerivationTree, symbol: str,
                          depth: int = -1):
        reductions = (self.subtrees_with_symbol(tree, symbol, depth=depth)
                      + self.alternate_reductions(tree, symbol, depth=depth))
        # Filter duplicates and put into unique_reductions [omitted]
        return unique_reductions
```

We can see this. Recall that a `<expr>` is, in our case, either `<term> + <expr>` or just `term`.

```
>>> print ([all_terminals(t) for t in grammar_reducer.subtrees_with_symbol(
                                        derivation_tree, "<expr>")])
['1 + (2 * 3)', '(2 * 3)', '2 * 3']
>>> print ([all_terminals(t) for t in grammar_reducer.alternate_reductions(
                                        derivation_tree, "<expr>")])
['1', '1 + (2 * 3)']
>>> print ([all_terminals(t) for t in grammar_reducer.symbol_reductions(
                                        derivation_tree, "<expr>")])
['1 + (2 * 3)', '(2 * 3)', '2 * 3', '1']
```

We can also look at potential reductions for `<term>`. The `1 * 1` is an alternate expansion.

```
>>> print ([all_terminals(t) for t in grammar_reducer.symbol_reductions(
                                        derivation_tree, "<term>")])
['1', '(2 * 3)', '2 * 3', '3', '1 * 1']
```

Overall strategy. There's a lot of code in the implementation of `reduce_subtree`, but it boils down to, for each child, (1) collecting the set of `symbol_reductions()`, (2) checking that a reduction is indeed smaller, and then (3) replacing the child with the reduction. If the reduction fails (as desired), we continue by processing the tree with the reduction in place. The method recursively calls itself to reduce its children.

There are a bunch of helper methods, but the overall API of the `GrammarReducer` is, as with all `Reducers`, the `reduce()` method.

```
>>>  grammar_reducer = GrammarReducer(
       eval_mystery,
       EarleyParser(EXPR_GRAMMAR),
       log_test=True)
>>> grammar_reducer.reduce(expr_input)
Test #1 '(2 * 3)' 7 FAIL
Test #2 '2 * 3' 5 PASS
Test #3 '3' 1 PASS
Test #4 '2' 1 PASS
Test #5 '(3)' 3 FAIL
(3)
```

This does work, in 5 steps, with something that is a valid input. Indeed, all of the tests are valid inputs, and we avoid the `UNRESOLVED` parse errors.

Adding depth. So far, we try to replace subtrees with the smallest possible subtrees; we replace `2 * 3` by first `2` and then `3`. Going straight to small trees may remove too much at once, and delta

debugging tries to remove a "sensible" amount at once. In the tree context, we want to replace with larger subtrees instead of the smallest possible subtree.

How do we get larger subtrees? We replace by subtrees that are closer to the root, i.e. that have smaller depth. By definition, they are larger.

The `subtrees_with_symbol()` function can take a `depth` parameter, which makes it only return subtrees at the given depth.

The example in the *Fuzzing Book* doesn't completely make sense to me, but the search starts with depth 0 and increases it. So, we preferentially replace with bigger subtrees.

We can observe the behaviour:

```
>>>  grammar_reducer = GrammarReducer(
       eval_mystery,
       EarleyParser(EXPR_GRAMMAR),
       log_test=True)
>>> grammar_reducer.reduce_tree = grammar_reducer.reduce_tree_with_depth
>>> grammar_reducer.reduce(expr_input)
Test #1 '(2 * 3)' 7 FAIL
Test #2 '(3)' 3 FAIL
Test #3 '3' 1 PASS

'(3)'
```

The key differences are that this doesn't try the subtrees `2 * 3`, or the single-digit subtrees. It replaces the entire <`expr`> with the `(2 * 3)` <`expr`>, as before. Then, it replaces the remaining <`term`> by the <`factor`> that goes to the <`digit`> 3. There is nothing more that fails, so it terminates. This terminates in 3 steps instead of 5.

Grammar-Based versus Delta Debugging

Let's measure effectiveness. Here is a long expression:

```
long_expr_input = GrammarFuzzer(EXPR_GRAMMAR, min_nonterminals=100).fuzz()
```

And we can use the grammar reducer:

```
grammar_reducer = GrammarReducer(eval_mystery, EarleyParser(EXPR_GRAMMAR))
with Timer() as grammar_time:
    print(grammar_reducer.reduce(long_expr_input))
```

I ran it and it needed 9 tests, finishing in 0.033s.

versus the delta debugger:

```
dd_reducer = DeltaDebuggingReducer(eval_mystery)
with Timer() as dd_time:
    print(dd_reducer.reduce(long_expr_input))
```

This took 618 tests and took 0.503s.

We can see that where there is a grammar, the grammar reducer wins handily.