Software Testing, Quality Assurance and Maintenance Winter 2026

Lecture 9 — February 2, 2026

Patrick Lam *version 1*

We said that there was mutation-based fuzzing and grammar-based fuzzing. Let's look at grammar-based fuzzing now.

Grammar-based fuzzing

Regular expressions and context-free grammars are core CS 241 material, for those of you who are Software Engineering students. In CS 241, you used regexps and CFGs to parse input.

Examples. Here is a Perl-syntax regular expression for (some) Visa credit card numbers:

```
^4[0-9]{12}(?:[0-9]{3})?$
```

It says that a Visa number starts (^) with a 4, then contains 12 digits between 0 and 9, optionally another 3 more digits, and ends. (This regexp is old; today, Visa numbers can be 19 digits).

You can use this regular expression to check that a given number is a valid Visa number. Or—especially useful for the purpose of this course—you can use it to generate numbers of the right shape. You can imagine how to do that, and we'll write out implementations to do so today.

It turns out, though, that 90% of numbers that you generate from this regular expression are definitely *not* valid Visa numbers, because credit card numbers have a checksum as the last digit, computed using the Luhn algorithm. This algorithm can't be specified using either regular expressions or context-free grammars. We'll get back to this point.

Moving along, we also have context-free grammars. The standard example is expressions.

```
<start>   ::= <expr>
<expr>    ::= <term> + <expr> | <term> - <expr> | <term>
<term>    ::= <term> * <factor> | <term> / <factor> | <factor>
<factor>  ::= +<factor> | -<factor> | (<expr>) | <integer> | <integer>.<integer>
<integer> ::= <digit><integer> | <digit>
<digit>   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

For our purposes, CFGs are the same as regular expressions, in the sense that we can recognize strings in the language, and we can generate strings from the language. The difference in expressiveness is not that important to us.

Thus: when programs' input formats are determined by grammars, then it is possible to use the grammar backwards to generate inputs. These test inputs are going to be more interesting than totally random sequences of characters—if properly designed, they can test system behaviour

beyond just superficial input validation (as we saw in Lecture 7). In particular, we aim to create grammars that specify (a superset of) the legal set of inputs to our system under test.

Although character-based input formats are easiest, it's also possible to use grammars for configurations, APIs, GUIs, etc.

Generating from Context-Free Grammars (CFGs)

We'll continue by talking about grammars, as seen in `https://www.fuzzingbook.org/html/Grammars.html`.

Recall that regular expressions can specify regular languages, as can be recognized by a finite state machine (FSM). FSMs can't count—for instance, specifically, they can't match parentheses.

Context-free grammars are more powerful than regular expressions, and they can count. Even so, as I wrote above, some restrictions can't be expressed in context-free languages/recognized by context-free grammars, like the Luhn algorithm checksum requirement.

Here's another grammar, expressing the set of two-digit numbers.

```
<start> ::= <digit><digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

(This could also be expressed as a regular expression.)

We are aiming to generate inputs from a grammar. Let's do that for this example.

1. We start with the <start> production and see that we have two instances of the <digit> nonterminal.
2. For each <digit>, we visit its production, which says that there are 10 valid alternatives for a digit, each of them a distinct terminal.
3. To generate an input, we randomly choose one alternative for each digit, say "1" and "7". Concatenating our choices, we get a result like "17".

This is grammar fuzzing. We can also carry out grammar fuzzing for the expressions example.

1. Once again, start at <start>.
2. There is one alternative, which is <expr>. Visit <expr>.
3. <expr> can be three things. Randomly choose <term> + <expr>. Visit <term>, with + and <expr> still to generate.
4. There are also 3 alternatives for <term>. Randomly choose <factor>.
5. Of the 5 alternatives for <factor>, choose <integer>.
6. There are 2 alternatives at <integer>. Choose <digit>.
7. Randomly choose the terminal 4 and generate it. We've bottomed out and continue with the remaining bits of the expr, which are + and expr.
8. ...

One possible thing we might generate, if we finish this example, is `4 + 22 * 5.3`. We can randomly generate lots of expressions from this grammar, and they are all valid expressions.

Representing Grammars in Python

In Java we would set up a class hierarchy, and in Python we could, but perhaps it's more idiomatic to just use Python's data structures to represent grammars, with conventions for meaning. We can represent a grammar in Python as a mapping from the alternative's left-hand side (LHS) to its right-hand side (RHS), e.g.

```
DIGIT_GRAMMAR = {
    "<start>":
        ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"]
}
```

Nonterminals are in <brackets> and the rest of a string is taken as terminals. This is a single-production grammar that says that the start symbol can resolve to one of the 10 alternatives, which are terminals representing the digits 0 through 9.

A type hint that works in Python 3.12 and up is:

```
from typing import Dict, List
type Expansion = str
type Grammar = Dict[str, List[Expansion]]
```

(in older Pythons, omit the keyword type).

We can express the expression grammar above in our Python representation also:

```
EXPR_GRAMMAR: Grammar = {
    "<start>":
        ["<expr>"],
    "<expr>":
        ["<term> + <expr>", "<term> - <expr>", "<term>"],
    "<term>":
        ["<factor> * <term>", "<factor> / <term>", "<factor>"],
    "<factor>":
        ["+<factor>", "-<factor>",
         "(<expr>)",
         "<integer>.<integer>", "<integer>"],
    "<integer>":
        ["<digit><integer>", "<digit>"],
    "<digit>":
        ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"]
  }
```

Since it's a mapping, we can use lookups and inclusion tests:

```
>>> EXPR_GRAMMAR["<digit>"]
>>> "<integer>" in EXPR_GRAMMAR
```

Our grammars must always have start symbol <START>:

```
START_SYMBOL = "<start>"
```

Here's some helper functions for nonterminals, using a regular expression.

```
import re
RE_NONTERMINAL = re.compile(r'(<[^<> ]*>)')
```

```
def nonterminals(expansion):
    # In later chapters, we allow expansions to be tuples,
    # with the expansion being the first element
    if isinstance(expansion, tuple):
        expansion = expansion[0]

    return RE_NONTERMINAL.findall(expansion)

def is_nonterminal(s):
    return RE_NONTERMINAL.match(s)
```

There are more examples in the Fuzzing Book, but we can do this:

```
>>> assert nonterminals("<term> * <factor>") == ["<term>", "<factor>"]
>>> assert is_nonterminal("<symbol-1>")
```

A Simple Grammar Fuzzer

We have set up enough infrastructure to write a simple string-replacement-based fuzzer now.

```
import random
class ExpansionError(Exception):
  pass
def simple_grammar_fuzzer(grammar: Grammar,
                          start_symbol: str = START_SYMBOL,
                          max_nonterminals: int = 10,
                          max_expansion_trials: int = 100,
                          log: bool = False) -> str:
    """Produce a string from `grammar`.
       `start_symbol`: use a start symbol other than `<start>` (default).
       `max_nonterminals`: the maximum number of nonterminals
         still left for expansion
       `max_expansion_trials`: maximum # of attempts to produce a string
       `log`: print expansion progress if True"""

    term = start_symbol
    expansion_trials = 0

    while len(nonterminals(term)) > 0:
        symbol_to_expand = random.choice(nonterminals(term))
        expansions = grammar[symbol_to_expand]
        expansion = random.choice(expansions)
        if isinstance(expansion, tuple):
            expansion = expansion[0]

        new_term = term.replace(symbol_to_expand, expansion, 1)

        if len(nonterminals(new_term)) < max_nonterminals:
            term = new_term
            if log:
                print("%-40s" % (symbol_to_expand + " -> " + expansion), term)
            expansion_trials = 0
        else:
            expansion_trials += 1
            if expansion_trials >= max_expansion_trials:
                raise ExpansionError("Cannot expand " + repr(term))
```

```
37
38      return term
```

You can find this in `code/L09/simple_grammar_fuzzer.py` and you can run this fuzzer with the invocation:

```
1  >>> simple_grammar_fuzzer(grammar=EXPR_GRAMMAR, max_nonterminals=3, log=True)
```

This function takes a string, finds a nonterminal (using the brackets), picks an alternative for that nonterminal, and replaces the nonterminal with one of the permissible RHSs. It is all string replacement. One might do this by tree manipulations instead, which would be more effective, but less quick-and-dirty. There are some hard limits on expansions to guarantee fast-enough termination.

Other Grammar Examples

The *Fuzzing Book* talks about railroad diagrams for visualizing grammars, and provides code to use libraries to do that. We don't, but we may include some pictures from there as needed.

We will, however, provide more grammars. Here's the grammar accepted by the `cgi_decode()` function we saw earlier.

```
1  CGI_GRAMMAR: Grammar = {
2      "<start>":
3          ["<string>"],
4      "<string>":
5          ["<letter>", "<letter><string>"],
6      "<letter>":
7          ["<plus>", "<percent>", "<other>"],
8      "<plus>":
9          ["+"],
10     "<percent>":
11         ["%<hexdigit><hexdigit>"],
12     "<hexdigit>":
13         ["0", "1", "2", "3", "4", "5", "6", "7",
14             "8", "9", "a", "b", "c", "d", "e", "f"],
15     "<other>":  # Actually, could be _all_ letters
16         ["0", "1", "2", "3", "4", "5", "a", "b", "c", "d", "e", "-", "_"],
17 }
```

and we can produce CGI strings:

```
1  for i in range(10):
2      print(simple_grammar_fuzzer(grammar=CGI_GRAMMAR, max_nonterminals=10))
```

The hit rate for valid CGI strings is much higher with grammar fuzzing than for basic fuzzing and for mutation-based fuzzing.

Here's another example:

```
1  URL_GRAMMAR: Grammar = {
2      "<start>":
3          ["<url>"],
4      "<url>":
5          ["<scheme>://<authority><path><query>"],
```

```
    "<scheme>":
        ["http", "https", "ftp", "ftps"],
    "<authority>":
        ["<host>", "<host>:<port>", "<userinfo>@<host>", "<userinfo>@<host>:<port>"],
    "<host>":  # Just a few
        ["patricklam.ca", "www.google.com", "fuzzingbook.com"],
    "<port>":
        ["80", "8080", "<nat>"],
    "<nat>":
        ["<digit>", "<digit><digit>"],
    "<digit>":
        ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"],
    "<userinfo>":  # Just one
        ["user:password"],
    "<path>":  # Just a few
        ["", "/", "/<id>"],
    "<id>":  # Just a few
        ["abc", "def", "x<digit><digit>"],
    "<query>":
        ["", "?<params>"],
    "<params>":
        ["<param>", "<param>&<params>"],
    "<param>":  # Just a few
        ["<id>=<id>", "<id>=<nat>"],
}
```

and we can push the button and generate URLs.

```
for i in range(10):
    print(simple_grammar_fuzzer(grammar=URL_GRAMMAR, max_nonterminals=10))
```

It's a narrow subset of the entire space of valid URLs, but we get better coverage of our chosen subset.

The *Fuzzing Book* also provides an example for generating mad-lib style book titles using grammars, but we'll omit that.

Validity of generated inputs. To reiterate what we've said before: if you have a grammar, you can easily generate strings belonging to the grammar. Inputs generated from a grammar will always satisfy the grammar. However, they may fail to be valid inputs for a program if there are some additional constraints for inputs to be valid for that program. A port number needing to be between 1024 and 2048 is hard. At least as hard is a checksum requirement, as with credit card numbers.

It's possible to attach constraints to grammars to make sure one generates more valid inputs. (Other, unrelated, things that one can do are to eliminate redundancy and to weight some alternatives more heavily than others.)

Grammars as Mutation Seeds

So far, we've generated only syntactically valid inputs from grammars, which is helpful. But also, part of the point of fuzzing is to generate *invalid* inputs. Mutation can help with that, so back to

mutation.

Our mutation fuzzer starts with a number of seeds and then mutates them. Grammar-based generation can first create valid seeds, and then the mutation fuzzer can work from those. This can give a mix of valid and (slightly) invalid inputs, which helps exercise the parser in ways that are difficult for humans to generate.

```
number_of_seeds = 10
seeds = [
    simple_grammar_fuzzer(
        grammar=URL_GRAMMAR,
        max_nonterminals=10) for i in range(number_of_seeds)]
seeds
m = MutationFuzzer(seeds)
[m.fuzz() for i in range(20)]
```

If we run this, we can see that we start with 10 inputs generated by the grammars, and then we have a mix of valid and invalid inputs that are fuzzed. One could run a URL validator to see how many of them are actually valid, and also check to see if they would parse according to the grammar (though we do not have infrastructure for that).

Additional stuff

The *Fuzzing Book* has a bunch of grammar utilities which I won't define; I'll mention them when they come up. I'm not sure about if you've seen EBNF, but it's a bunch of shortcuts for grammars:

- <symbol>? means that <symbol> can occur 0 or 1 times;
- <symbol>+ means that <symbol> can occur 1 or more times;
- <symbol>* means that <symbol> can occur 0 or more times;
- parentheses can be used with these shortcuts, e.g. (<s1><s2>)+

So, instead of

```
  "<identifier>": ["<idchar>", "<identifier><idchar>"],
```

we can just write

```
  "<identifier>": ["<idchar>+"],
```

The *Fuzzing Book* has a function `convert_ebnf_grammar()` that translates EBNF to BNF. You can find these in `code/L09/ebnf.py`.

The code so far has also allowed expansions to be a pair, like

```
"<expr>":
        [("<term> + <expr>", opts(min_depth=10)),
         ("<term> - <expr>", opts(max_depth=2)),
          "<term>"]
```

and there are some helper functions `opts()`, `exp_string()`, `exp_opt()`, `exp_opts()`, `set_opts()`. These live in `code/L09/opts.py`.

Finally, there is a utility function `trim_grammar()` and a validation function `is_valid_grammar()`. `trim_grammar()` returns a semantically-equivalent grammar to what you passed in, without unneeded expansions, while `is_valid_grammar()` does semantic checks on a grammar.

0.1 Applications of Grammars in Testing

The *Fuzzing Book* lists some projects using grammar-based fuzzing, principally for compiler testing and also web browsers. The earliest citation they have is for work by Burkhardt in 1967 [Bur67] which claims to automatically generate programs from grammars.

CSmith. At some level, Csmith [YCER11] uses grammar-based fuzzing to generate C programs, finding over 400 previously unknown compiler bugs. But there is much more to it: they go to a lot of effort to make sure that Csmith doesn't generate programs with undefined behaviour, which is well beyond what a grammar can guarantee. There is also the notion of a pseudo-oracle here: one doesn't know what the right output is, so they compare GCC and LLVM outputs.

EMI. Still in the C compiler domain, but using a different technique, is Equivalence Modulo Inputs [LAS14]. Here, the idea is to fuzz dead code (as observed by measuring coverage) in valid C programs and observe compilers mis-compiling the fuzzed programs. I wouldn't really put this in the grammars section, but the *Fuzzing Book* did.

LangFuzz. By some of the authors of the *Fuzzing Book*, the LangFuzz tool [HHZ12] performs grammar-based fuzzing on test suites that contained past bugs. There are still new bugs to be found in the vicinity of old bugs. They collected a bunch of bug bounties; almost all of the bugs they found were memory safety bugs (so, implicit oracles).

(Tangentially, I also noticed that Andreas Zeller, one of the *Fuzzing Book* authors, recently published a paper [BZ25] where they inferred the input grammar from recursive descent parser code.)

Grammarinator. Seems to be similar to LangFuzz, but it's an open-source grammar fuzzer[1] written in Python [HKG18].

Domato. Not just compilers: Domato [2] fuzzes Document Object Model input. Specifically, it generates fuzzed versions of HTML, CSS, and JavaScript files, using grammars but starting from scratch. More about this project: `https://projectzero.google/2017/09/the-great-dom-fuzz-off-of-2017.html`.

[1] `https://github.com/renatahodovan/grammarinator`

[2] `https://github.com/googleprojectzero/domato`

Efficient Grammar Fuzzing

Next up: some tricks to generate from grammars more efficiently, from `https://www.fuzzingbook.org/html/GrammarFuzzer.html`.

References

[Bur67] W. H. Burkhardt. Generating test programs from syntax. *Computing*, 2(1):53–73, March 1967.

[BZ25] Leon Bettscheider and Andreas Zeller. Inferring input grammars from code with symbolic parsing. *ACM Trans. Softw. Eng. Methodol.*, November 2025. Just Accepted.

[HHZ12] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, page 38, USA, 2012. USENIX Association.

[HKG18] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. Grammarinator: a grammar-based open source fuzzer. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, A-TEST 2018, pages 45–48, New York, NY, USA, 2018. Association for Computing Machinery.

[LAS14] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 216–226, New York, NY, USA, 2014. Association for Computing Machinery.

[YCER11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 283–294, New York, NY, USA, 2011. Association for Computing Machinery.