Software Testing, Quality Assurance & Maintenance—Lecture 9

Patrick Lam
University of Waterloo

February 2, 2026

Part I

Intro: Grammar-Based Fuzzing

Types of Fuzzing

Goal: generate many test cases automatically.

Mutation-based fuzzing: generate new inputs automatically, by modifying known inputs.

Grammar-based fuzzing: generate new inputs automatically, using a grammar.

Example: Regular expression

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

Uses:

- check if a number is valid
- generate numbers of the right shape

But: also must satisfy checksum rules!

Checksum rules for credit card numbers

Luhn algorithm:

calculate a checksum from all-but-last digits;
the last digit must match the checksum.

Can’t specify checksum rules as regexp or context-free grammars.

Example (standard): expressions

Context-free grammars (CFGs):

```
<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
```

Can also recognize and generate strings for CFGs.

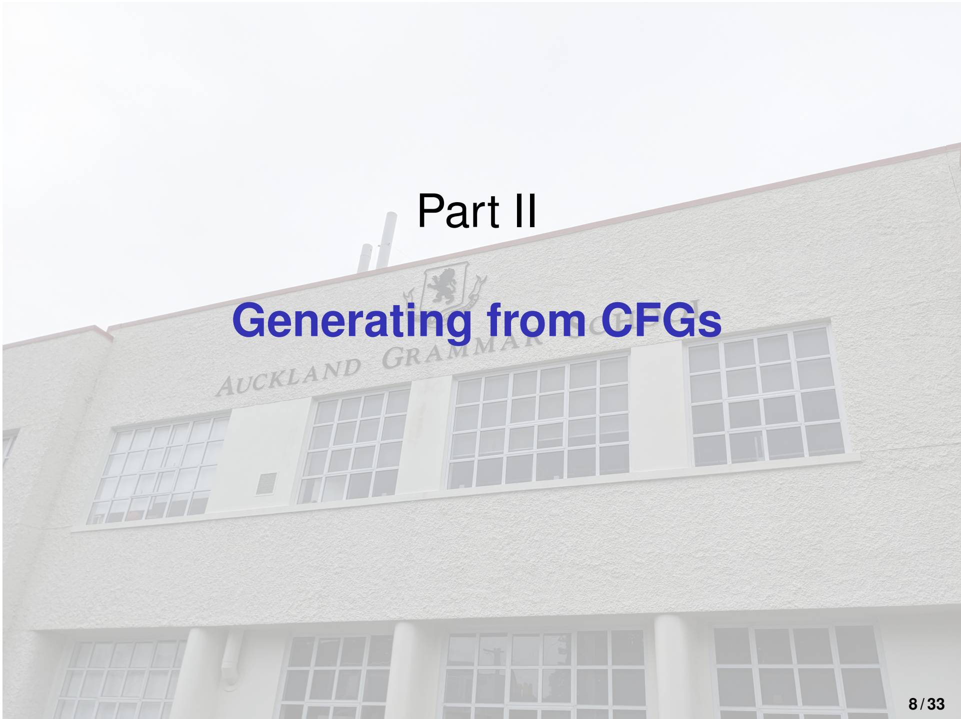
Generating from grammars

What we're doing in this lecture:
generating (trees and) strings
from grammars.

These are more interesting than random sequences of characters;
they test behaviour beyond input validation.

Aim: create grammars that specify all legal inputs.

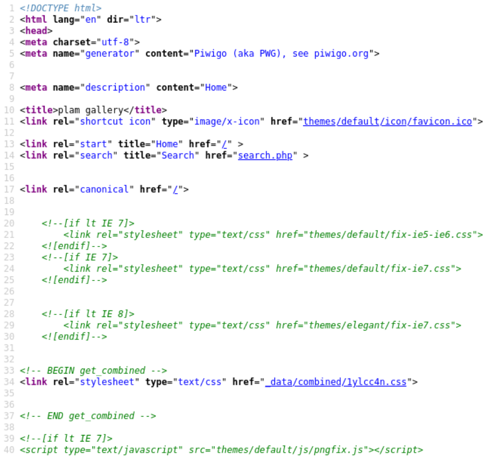
Other uses: configs, APIs, GUIs, etc.



Part II

Generating from CFGs

Expressiveness



```
<!DOCTYPE html>
<html lang="en" dir="ltr">
<head>
<meta charset="utf-8">
<meta name="generator" content="Piwigo (aka PWG), see piwigo.org">


<meta name="description" content="Home">

<title>plam gallery</title>
<link rel="shortcut icon" type="image/x-icon" href="themes/default/icon/favicon.ico">

<link rel="start" title="Home" href="/" >
<link rel="search" title="Search" href="search.php" >


<link rel="canonical" href="/">


    <!--[if lt IE 7]>
        <link rel="stylesheet" type="text/css" href="themes/default/fix-ie5-ie6.css">
    <![endif]-->
    <!--[if IE 7]>
        <link rel="stylesheet" type="text/css" href="themes/default/fix-ie7.css">
    <![endif]-->


    <!--[if lt IE 8]>
        <link rel="stylesheet" type="text/css" href="themes/elegant/fix-ie7.css">
    <![endif]-->


<!-- BEGIN get_combined -->
<link rel="stylesheet" type="text/css" href="_data/combined/1ylcc4n.css">


<!-- END get_combined -->

<!--[if lt IE 7]>
<script type="text/javascript" src="themes/default/js/pngfix.js"></script>
```

Regexps can’t count (so no HTML parsers).
CFGs can’t implement Luhn algorithm.

Generating Inputs

How to generate inputs from this grammar?

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

(Could be expressed as a regexp).

Generating Inputs: double-digits

- `<start>` $\rightarrow$ `<digit><digit>`
- visit first `<digit>`, have 10 choices;
  choose randomly, say `1`;
  replace `<digit>` by `1`.
- visit second `<digit>`,
  choose say `7` randomly;
  replace `<digit>` by `7`.
- generated input: `17`

Generating Inputs: back to expr

```
<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
```

Generating Inputs: expr

- `<start>` → `<expr>`
- visit `<expr>`; three `<expr>` alternatives,
  randomly choose `<term>` + `<expr>`,
  replace `<expr>` by our choice.
- visit `<term>`; also three `<term>` alternatives,
  randomly choose `<factor>`.
- visit `<factor>`;
  randomly choose `<integer>` of the 5 alternatives.
- visit `<integer>`; randomly choose `<digit>`;
- visit `<digit>`;
  randomly choose terminal `4` and generate it.
- continue with next nonterminal `expr` ...

Could generate, for instance, `4 + 22 * 5.3`, or many other expressions.

Coding Things Up: Grammars in Python

Just use Python data structures.

Grammar = mapping from an alternative's LHS to its RHS.
Here is a single-production grammar.

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

Nonterminals in `<brackets>`.
All else taken as terminals.

Python Type Hint for Grammars

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

Expression Grammar in Python

```
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"]
  }
```

Playing with Grammars

```
>>> EXPR_GRAMMAR["<digit>"]
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
>>> "<integer>" in EXPR_GRAMMAR
True
```

Conventions and Helpers

Must always have start symbol `<start>`:

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

Non-terminals always in `<brackets>`:

```
import re
RE_NONTERMINAL = re.compile(r'(<[^<> ]*>)')
def nonterminals(expansion):
    if isinstance(expansion, tuple):
        # can be a tuple, use first element
        expansion = expansion[0]

    return RE_NONTERMINAL.findall(expansion)

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

Using these Utilities

```
>>> nonterminals("<term> * <factor>")
["<term>", "<factor>"]
>>> is_nonterminal("<symbol-1>")
<re.Match object; span=(0, 10), match='<symbol-1>'>
>>> is_nonterminal("<symbol-1")
>>>
```

A Simple Grammar Fuzzer

```
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"""
```

It’s all strings: implementation

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

  if len(nonterminals(new_term)) < max_nonterminals:
      term = new_term
      expansion_trials = 0
  else:
      expansion_trials += 1
      if expansion_trials >= max_expansion_trials:
          raise ExpansionError("Cannot expand " + repr(term))

return term
```

Comments on Simple Grammar Fuzzer

- takes a string, finds a nonterminal, replaces with an alternative.
- it’s all string replacement.
- tree manipulations would be better; stay tuned.
- there are some limits to help w/termination.

Grammar: CGI

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

Generating CGI strings

```
>>> for i in range(10):
...     print(simple_grammar_fuzzer(grammar=CGI_GRAMMAR,
                                   max_nonterminals=10))
...
%e5
+
+3
_1a
e+
%625%ee
%db%df%5d
+44
%4b
+%b8+2
```

Grammar: URL

```
URL_GRAMMAR: Grammar = {
    "<start>": ["<url>"],
    "<url>": ["<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>": ["abc", "def", "x<digit><digit>"],
    "<query>": ["", "?<params>"],
    "<params>": ["<param>", "<param>&<params>"],
    "<param>": ["<id>=<id>", "<id>=<nat>"],
}
```

Generating URLs

```
>>> for i in range(10):
...     print(simple_grammar_fuzzer(grammar=URL_GRAMMAR,
                                    max_nonterminals=10))
...
https://user:password@www.google.com/abc
http://user:password@fuzzingbook.com/
http://user:password@patricklam.ca/def?x97=60
ftp://user:password@fuzzingbook.com/x60?abc=def
https://patricklam.ca/?x84=31&x95=x12
ftp://www.google.com:1/abc
ftp://user:password@fuzzingbook.com:80/x40?def=6&x12=abc
ftp://user:password@www.google.com
http://user:password@fuzzingbook.com/def?x35=1
ftp://user:password@www.google.com/abc
```

This isn't all URLs, but we do get better coverage of our chosen subset.
Other examples possible, e.g. book titles (see *Fuzzing Book*).

On Validity

If you have a grammar,
    you can generate strings belonging to the grammar
    (as we've seen).

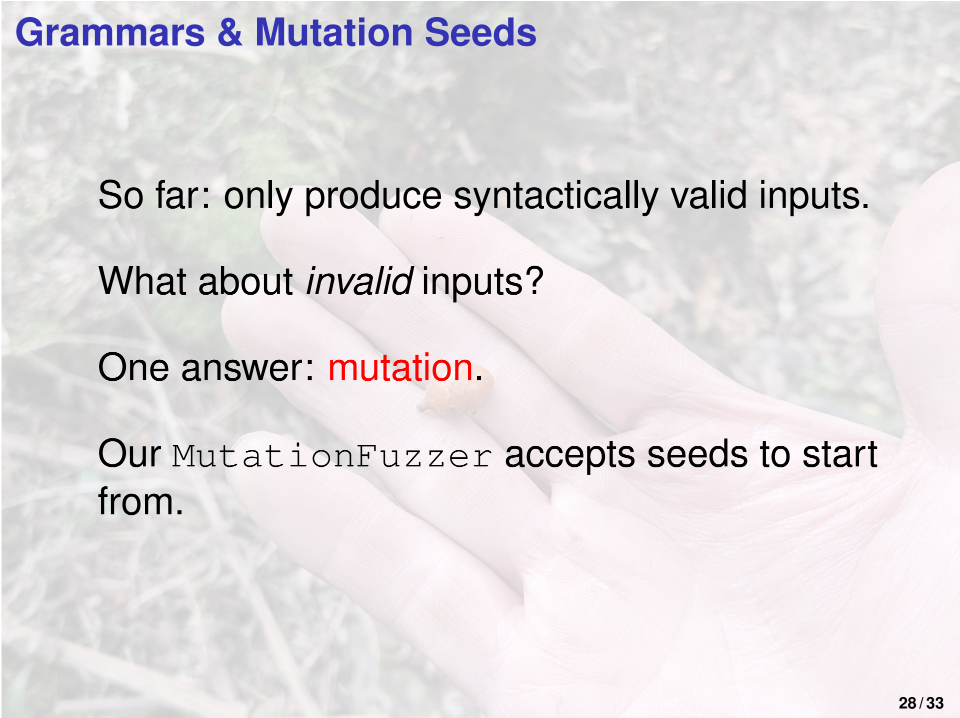
These strings, by definition, always satisfy the grammar.

They may not be valid inputs to a program.

- e.g. port number between 1024 and 2048;
- e.g. checksum digit satisfies Luhn's algorithm.

Can attach constraints to grammars to generate more-valid inputs, or weight some alternatives more heavily (not discussed today).

Grammars & Mutation Seeds



So far: only produce syntactically valid inputs.

What about *invalid* inputs?

One answer: mutation.

Our `MutationFuzzer` accepts seeds to start from.

Generating CGI strings

```
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)]
```

Use the grammar fuzzer to produce seeds, then the mutation fuzzer to mutate them.

EBNF

Extended Backus-Naur form; is syntactic sugar:

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

Instead of

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

just write

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

Function `convert_ebnf_grammar()` (in `code/L09/ebnf.py`) translates EBNF to BNF.

Opts (for future expansion)

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

and there are functions like `opts()`, `exp_string()`, `exp_opt()`, `exp_opts()`, `set_opts()` in `code/L09/opts.py`.

Grammar Utilities

found in `code/L09/grammars.py`:

`trim_grammar()`: remove unneeded expansions;

`is_valid_grammar()`: validity checks, e.g. no unreachable symbols, etc

Applications of Grammars in Testing

- Earliest known: Burkhardt, 1967: “Generating test programs from syntax.”
- CSmith: grammar-based fuzzing; they work hard to only generate valid C programs; pseudo-oracle comparing GCC and LLVM.
- EMI: fuzz dead code, observe mis-compilations.
- LangFuzz: grammar-based fuzzing on test suites.
- Grammarinator: open-source grammar fuzzer in Python.
- Domato: fuzzes Document Object Model inputs.