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`.

I found the design choice in `simple_grammar_fuzzer` to manipulate strings directly to be... interesting. I would have thought that one would use a tree, though there is more faffing with data structures. And indeed, we are now going to use trees, and introduce various controls that we can use to control what trees get generated.

We had talked about a grammar for expressions earlier today. In `code/L09/ebnf.py` you'll find an EBNF grammar for expressions. We can convert it to BNF so that we can work with it.

```
1 expr_grammar = convert_ebnf_grammar(EXPR_EBNF_GRAMMAR)
```

Because we're talking about grammar fuzzing, let's try to fuzz this grammar. I'll anticipate timeouts. You can find this in `code/L09/simple_grammar_fuzzer_problem.py`.

```
1 with ExpectTimeout(1):
2   simple_grammar_fuzzer(grammar=expr_grammar, max_nonterminals=3)
```

This does indeed raise a `TimeoutError`.

You can look at the grammar, and it's a good exercise to do so. But if you call `simple_grammar_fuzzer` with `log=True` you can see the expansions, and the proximate reason why it times out should become abundantly clear: too many parens. But why?

The rule for <factor> has RHS:

```
1 ['<sign-1><factor>', '(<expr>)', '<integer><symbol-1>']
```

and our simple fuzzer throws away generated strings with more than the maximum allowed number of non-terminals (3 in our case). So the only allowed string to be generated for <factor> is '(<expr>)': the other alternatives all add a non-terminal, and when at the limit, the fuzzer won't do that.

Any grammar-based approach can create arbitrarily long strings. The problem with the approach as we have implemented it, though, is that the hard limit on non-terminals prevents it from generating some strings that we want.

In general, the single `max_nonterminals` control that `simple_grammar_fuzzer` provides us is insufficient. And, manipulating strings is less efficient than manipulating trees: the fuzzer has to do a search through the string for non-terminals.

The *Fuzzing Book* continues with a graph showing that the time required to generate an output grows quadratically with the output length, and also that very many outputs are tens of thousands of characters long.

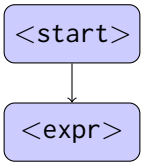
We want an efficient grammar fuzzer, which doesn't get stuck on bad alternatives.

Derivation Trees

The obvious choice for representing derivations is *derivation trees*. The nodes in the trees can be terminals or nonterminals; nonterminals may have children corresponding to a derivation, as we'll see. The *Fuzzing Book* says that they also known as concrete syntax trees or parse trees, though my definition of a concrete syntax tree is one where leaves are terminals.

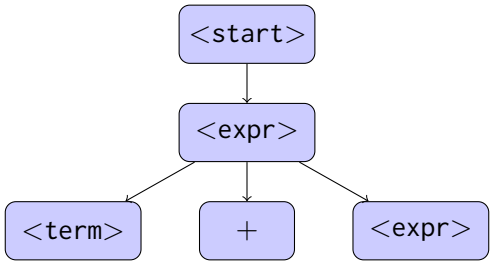
Let's look at an example of deriving a string, again using the arithmetic expression grammar. We start with the <start> node.

In general, we search the tree for a nonterminal S that does not have children and hence is a candidate for expansion. In this case, there is only <start>, and its only alternative is <expr>, so we expand the tree by adding <expr> as a child of <start>.



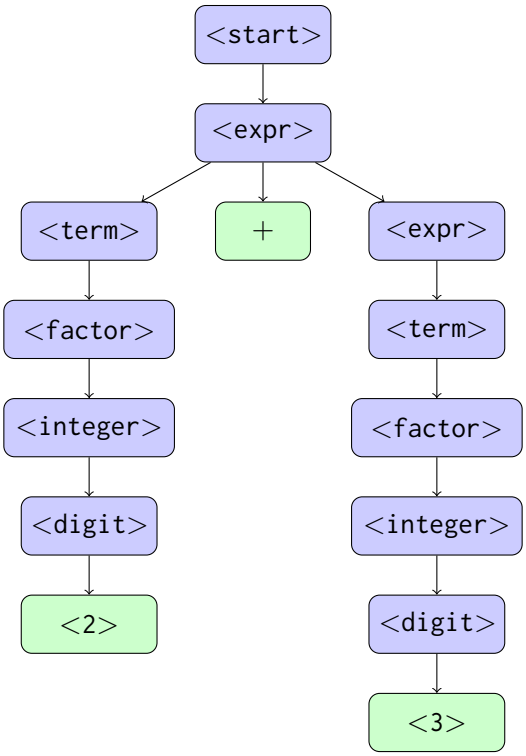
This tree can be unparsed back to the string "<expr>" by collecting the leaves of the tree.

We can continue the expansion process. Nonterminal <expr> can expand to <term> + <expr>, yielding the tree:



Here we have one terminal, +, as well as two more nonterminals.

We can further continue the expansion process until we get to terminals.



A derivation tree can yield the strings that we've seen before. But it also records the nonterminals that were expanded to yield a given final string. And, being a tree, it doesn't require string searches and replacements; it's far more efficient to manipulate the tree.

Derivation Trees in Python

Here's the way we represent derivation trees in Python. A node is a pair

```
1  (SYMBOL_NAME, CHILDREN)
```

where `SYMBOL_NAME` is a string representing the node (e.g. "`<start>`" or "+") and `CHILDREN` generally represents the children. So the node also contains the subtree starting at that node.

We also allow `CHILDREN` to be `None`, which means that this node is a nonterminal still to be expanded; or, `CHIDLREN` can be `[]`, meaning that this node is a terminal. There is an invariant that `SYMBOL_NAME` is `<bracketed>` iff the node is a nonterminal.

We can thus state a Python type:

```
1  DerivationTree = Tuple[str, Optional[List[Any]]]
```

which could actually be a recursive type using `DerivationTree` instead of `Any`, but the *Fuzzing Book* says the Python type checker can't handle that, which I've confirmed.

Here is an example derivation tree for `<expr> + <term>`.

```
derivation_tree: DerivationTree = ("<start>",
                   [("<expr>",
                     [("<expr>", None),
                      (" + ", []),
                         ("<term>", None)]
                     )])
```

There is a lot of implementation talk next in the *Fuzzing Book*. I'll skip most of it, but include it in `code/L09/derivation_tree.py` and `code/L09/grammar_fuzzer.py`.

I will say that there is a `GrammarFuzzer` class, which will take the following parameters:

```
class GrammarFuzzer(Fuzzer):
    """Produce strings from grammars efficiently, using derivation trees."""

    def __init__(self,
                 grammar: Grammar,
                 start_symbol: str = START_SYMBOL,
                 min_nonterminals: int = 0,
                 max_nonterminals: int = 10,
                 disp: bool = False,
                 log: Union[bool, int] = False) -> None:
        # ...
```

Right now, we are only going to use the grammar and start symbol. There are also diagnostics in the form of `disp` and `log`. The constructor calls `check_grammar()`, which ensures some validity properties:

```
      def check_grammar(self) -> None:
        """Check the grammar passed"""
        assert self.start_symbol in self.grammar
        assert is_valid_grammar(
            self.grammar,
            start_symbol=self.start_symbol,
            supported_opts=self.supported_opts())
```

We didn't talk about `is_valid_grammar`, but it checks that all nonterminals are reachable from either <`start`> or the start symbol that it is given.

Tree manipulation: expanding a node. One of the central operations that we need to be able to perform on a derivation tree is to randomly expand a nonterminal into one of its alternatives; for instance, an <`expr`> can be an addition, subtraction, or just a term, and we need to replace the <`expr`> with, for instance, <`term`> + <`expr`>.

Which alternative? We'll start by randomly choosing an alternative. But we'll want to use different strategies in the future.

```
    def choose_node_expansion(self, node: DerivationTree,
                              children_alternatives: List[List[DerivationTree]]) ->
                                                                        int:
        """Return index of expansion in 'children_alternatives' to be selected.
           'children_alternatives': a list of possible children for 'node'.
           Defaults to random. To be overloaded in subclasses."""
        return random.randrange(0, len(children_alternatives))
```

Despite moving to derivation trees, we still express our alternatives as strings. (Indeed, that's more user-friendly than the alternative). We need to convert alternatives into something derivation-tree-friendly. As an example, our grammar contains the alternative <term> + <expr> for <expr>. We want to convert it to the list of nodes

```
[('<term>', None), (' + ', []), ('<expr>', None)]
```

and we do so using function `expansion_to_children`, which can be overridden:

```
def expansion_to_children(expansion: Expansion) -> List[DerivationTree]:
    # print("Converting " + repr(expansion))
    # strings contains all substrings -- both terminals and nonterminals such
    # that ''.join(strings) == expansion

    expansion = exp_string(expansion)
    assert isinstance(expansion, str)

    if expansion == "":  # Special case: epsilon expansion
        return [("", [])]

    strings = re.split(RE_NONTERMINAL, expansion)
    return [(s, None) if is_nonterminal(s) else (s, [])
            for s in strings if len(s) > 0]
```

So, given a subtree with a non-expanded non-terminal at its root, we can expand this non-terminal into one of its alternatives, per the grammar, and return the new tree. Recall that being non-expanded is represented by having `children` be `None`.

```
    def expand_node_randomly(self, node: DerivationTree) -> DerivationTree:
        """Choose a random expansion for `node` and return it"""
        (symbol, children) = node
        assert children is None

        if self.log:
            print("Expanding", all_terminals(node), "randomly")

        # Fetch & parse the possible expansions from grammar...
        expansions = self.grammar[symbol]
        children_alternatives: List[List[DerivationTree]] = [
            self.expansion_to_children(expansion) for expansion in expansions
        ]

        # ... and select a random expansion
        index = self.choose_node_expansion(node, children_alternatives)
        chosen_children = children_alternatives[index]

        # Process children (for subclasses) [do nothing for now]
        chosen_children = self.process_chosen_children(chosen_children,
                                                       expansions[index])

        # Return with new children
        return (symbol, chosen_children)
```

Randomly is one strategy, and for now our only strategy, though we will redefine `expand_node` as needed to use other strategies.

```
    def expand_node(self, node: DerivationTree) -> DerivationTree:
        return self.expand_node_randomly(node)
```

Note that expand_node does not modify the input node but indeed returns a fresh DerivationTree, which is consistent with functional programming style.

We can try out expand_node:

```
    f = GrammarFuzzer(EXPR_GRAMMAR, log=True)

    print("Before expand_node_randomly():")
    expr_tree = ("<integer>", None)
    print (display_tree(expr_tree))

    expr_tree = f.expand_node_randomly(expr_tree)
    print("After expand_node_randomly():")
```

which starts with the tree with unexpanded single node (<integer>, None), and can finish with either the tree

```
  ('<integer>', [('<digit>', None), ('<integer>', None)])
```

or the tree

```
  ('<integer>', [('<digit>', None)])
```

as randomly chosen.

Tree manipulation: expanding the tree. We don't just want to be able to expand the root of the tree. We need to be able to find some unexpanded nonterminal in the tree and expand that. Method expand_tree_once() expands the root if it is unexpanded. Otherwise, it looks for any (transitive) children which are unexpanded. We do not need to look at this implementation, but expand_tree_once and any_possible_expansions are in the repo.

Note that unlike expand_node_randomly(), this method expand_tree_once() modifies the tree in-place, which is more efficient.

This method also hardcodes calls to the method expand_node(), but it's Python, so you can, and we do, redefine that method several times to use different strategies; in the example below, it is set to expand_node_randomly().

For instance, we can apply expand_tree_once() twice:

```
derivation_tree = ("<start>",
                   [("<expr>",
                     [("<expr>", None), (" + ", []), ("<term>", None)])])
print (derivation_tree)
f.expand_tree_once(derivation_tree)
f.expand_tree_once(derivation_tree)
print (derivation_tree)
```

Tree manipulation: closing the expansion. If we continue randomly choosing nonterminals, we will probably get derivation trees that are bigger than we'd like. I think that random expansion has finite expected termination time if there is a path to termination, but we would like smaller trees.

The insight that we'll apply, from [Luk00], is that, after inflating the tree up to a size bound, we then choose expansions that increase the tree size least. The <factor> production, for instance, has alternative <integer>(.<integer>)?, which could be <integer>, and <integer> could be a single <digit>. Other alternatives are higher-cost.

We can compute the minimum cost by doing a tree traversal and summing the minima for each production.

```
    def symbol_cost(self, symbol: str, seen: Set[str] = set()) \
            -> Union[int, float]:
        expansions = self.grammar[symbol]
        return min(self.expansion_cost(e, seen | {symbol}) for e in expansions)

    def expansion_cost(self, expansion: Expansion,
                       seen: Set[str] = set()) -> Union[int, float]:
        symbols = nonterminals(expansion)
        if len(symbols) == 0:
            return 1  # no symbol

        if any(s in seen for s in symbols):
            return float('inf')

        # the value of a expansion is the sum of all expandable variables
        # inside + 1
        return sum(self.symbol_cost(s, seen) for s in symbols) + 1
```

At a symbol, the symbol cost is the minimum of its children (1 for terminals and recursively calculated for nonterminals). Given an alternative (aka expansion), its minimum expansion cost is 1 if there are no nonterminals in that alternative, or 1 plus the sum of the nonterminals otherwise. But, this loops infinitely if there is a cycle involving a nonterminal, and so we use `float('inf')` in that case, which works properly with the Python `min` function.

We can check that the minimum cost of expanding <digit> is 1:

```
f = GrammarFuzzer(EXPR_GRAMMAR)
assert f.symbol_cost("<digit>") == 1
```

while the minimum cost of expanding <expr> is 5, through <term>, <factor>, <integer>, <digit>, and 1.

```
assert f.symbol_cost("<expr>") == 5
```

Previously, we chose a node to expand randomly (and limited the number of nonterminals). We now parametrize by choice algorithm, `choose`:

```
    def expand_node_by_cost(self, node: DerivationTree,
                            choose: Callable = min) -> DerivationTree:
```

I won't show the implementation, but I'll describe it. This method collects (in a list) the minimum cost of all of the alternatives, asks function `choose` to choose one of the costs in the list, then goes

back and collects all alternatives with that cost, and uses `choose_node_expansion()` to choose one of those alternatives (default: random). It calls a hook to post-process the chosen alternative, and returns it. This method is functional and does not modify the node it is given. Again, you can find the code in `code/L09/grammar_fuzzer.py`.

We can use `min` as the choice algorithm:

```
    def expand_node_min_cost(self, node: DerivationTree) -> DerivationTree:
        if self.log:
            print("Expanding", all_terminals(node), "at minimum cost")

        return self.expand_node_by_cost(node, min)
```

and we can use that to get a sequence of expansions:

```
    f.expand_node = f.expand_node_min_cost
    while f.any_possible_expansions(derivation_tree):
        derivation_tree = f.expand_tree_once(derivation_tree)
    print (derivation_tree)
    print (all_terminals(derivation_tree))
```

which might print out, for instance,

```
('<start>', [('<expr>', [('<expr>', [('<term>', [('<factor>', [('<integer>', [('<digit
                                                >', [('4', [])])])])])]), (' + ', []), ('<
                                                term>', [('<factor>', [('<integer>', [('<
                                                digit>', [('0', [])])])]), (' * ', []), ('<
                                                term>', [('<factor>', [('<integer>', [('<
                                                digit>', [('9', [])])])])])])])])])
4 + 0 * 9
```

The *Fuzzing Book* points out that `expand_node_min_cost()` does not increase the number of symbols, so that all open expansions are closed. They showed more steps than I did. You can't see it from these notes, but you can work through it and convince yourself that this is true, though being able to close all expansions is also a property of the grammar, in addition to the choice of node.

Tree manipulation: node inflation. We also want to have trees of at least a certain size, so that our programs have something to work with. We can easily look for the max-cost expansion using our machinery as well.

```
    def expand_node_min_cost(self, node: DerivationTree) -> DerivationTree:
        if self.log:
            print("Expanding", all_terminals(node), "at maximum cost")

        return self.expand_node_by_cost(node, max)
```

We can repeat a couple of times:

```
    derivation_tree = ("<start>",
                   [("<expr>",
                     [("<expr>", None),
                      (" + ", []),
                         ("<term>", None)]
                     )])
```

```
    f = GrammarFuzzer(EXPR_GRAMMAR, log=True)
    f.expand_node = f.expand_node_max_cost

    # do this a number of times
    if f.any_possible_expansions(derivation_tree):
        derivation_tree = f.expand_tree_once(derivation_tree)
    print (derivation_tree)
    print (all_terminals(derivation_tree))
```

Max, random, min. What we want is to combine all three types of tree expansion. Start by inflating the tree (max cost expansions), do some random expansions, and then close the tree (min cost expansions). The inflation step continues until there are at least `min_nonterminals` nonterminals (possibly set to 0), continue randomly expanding until there are at least `max_nonterminals` (or maybe the tree just closes itself), and then, if the grammar is properly designed, we get all terminals.

```
    def expand_tree(self, tree: DerivationTree) -> DerivationTree:
        """Expand `tree` in a three-phase strategy until all expansions are complete.
                                                """
        self.log_tree(tree)
        tree = self.expand_tree_with_strategy(
            tree, self.expand_node_max_cost, self.min_nonterminals)
        tree = self.expand_tree_with_strategy(
            tree, self.expand_node_randomly, self.max_nonterminals)
        tree = self.expand_tree_with_strategy(
            tree, self.expand_node_min_cost)

        assert self.possible_expansions(tree) == 0
        return tree
```

We omit `expand_tree_with_strategy`, but it repeatedly calls the given strategy's `expand_node` until the tree has at least the given limit of possible expansions.

We can try it again on the same initial derivation tree as before.

```
# derivation tree as before
f = GrammarFuzzer(
    EXPR_GRAMMAR,
    min_nonterminals=3,
    max_nonterminals=5,
    log=True)
f.expand_tree(derivation_tree)
```

What you'll probably see is one expansion at max cost, a few randoms, and then a sequence of min cost expansions. I end up with the tree of terminals `5 - 9 - 5 + 1 * 9` on one run, though of course you can get others (that's the whole point).

Just a fuzzer. We can abstract everything and provide just the `fuzz()` interface.

```
    def fuzz_tree(self) -> DerivationTree:
        """Produce a derivation tree from the grammar."""
        tree = self.init_tree()
        # print(tree)
```

```
        # Expand all nonterminals
        tree = self.expand_tree(tree)
        if self.log:
            print(repr(all_terminals(tree)))
        if self.disp:
            display(display_tree(tree))
        return tree

    def fuzz(self) -> str:
        """Produce a string from the grammar."""
        self.derivation_tree = self.fuzz_tree()
        return all_terminals(self.derivation_tree)
```

and we can invoke it easily:

```
    f = GrammarFuzzer(EXPR_GRAMMAR)
    print (f.fuzz())
```

There are sensible defaults for `min_nonterminals` and `max_nonterminals`, and we get a relatively large input from the fuzzer. You can also retrieve the derivation tree by reading the `.derivation_tree` property. One output you might get is:

```
+-(-1 * 8 * 7 - 9 * 5 + 4) / -2 + 031.429 * 2 * +--(+(8 + 1 - 3) / +1)
```

but it also might be as simple as

```
-49 / 901 / 4
```

We have our other grammars `URL_GRAMMAR` and `CGI_GRAMMAR` which we can just plug in:

```
https://user:password@www.google.com:33/abc?def=9
%11%55_%6b+
```

The *Fuzzing Book* times how long this fuzzer takes, comparing it to the `simple_grammar_fuzzer` above, and finds that it's much faster and produces smaller inputs. It concludes that we have much better control over grammar production.

And, yes, we work just fine with the EBNF grammar motivating all of this extra development.

```
    expr_grammar = convert_ebnf_grammar(EXPR_EBNF_GRAMMAR)
    f = GrammarFuzzer(expr_grammar, max_nonterminals=10)
    print (f.fuzz())
```

yields, for instance, input

```
97.01 * 5 * 7 - 8 * 9 - (9 + 0)
```

and not an infinite loop.

We will continue with techniques for learning grammars, probably.

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.

[Luk00] Sean Luke. Two fast tree-creation algorithms for genetic programming. *IEEE Transactions on Evolutionary Computation*, 4(3):274–283, 2000.

[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.