

# 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

```
1 <!DOCTYPE html>
2 <html lang="en" dir="ltr">
3 <head>
4 <meta charset="utf-8">
5 <meta name="generator" content="Piwigo (aka PWG), see piwigo.org">
6
7
8 <meta name="description" content="Home">
9
10 <title>plam gallery</title>
11 <link rel="shortcut icon" type="image/x-icon" href="themes/default/icon/favicon.ico">
12
13 <link rel="start" title="Home" href="/" >
14 <link rel="search" title="Search" href="search.php" >
15
16
17 <link rel="canonical" href="/">
18
19
20   <!--[if lt IE 7]>
21     <link rel="stylesheet" type="text/css" href="themes/default/fix-ie5-ie6.css">
22   <![endif]-->
23   <!--[if IE 7]>
24     <link rel="stylesheet" type="text/css" href="themes/default/fix-ie7.css">
25   <![endif]-->
26
27
28   <!--[if lt IE 8]>
29     <link rel="stylesheet" type="text/css" href="themes/elegant/fix-ie7.css">
30   <![endif]-->
31
32
33 <!-- BEGIN get_combined -->
34 <link rel="stylesheet" type="text/css" href="_data/combined/ylcc4n.css">
35
36
37 <!-- END get_combined -->
38
39 <!--[if lt IE 7]>
40 <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).

Note:  $\langle \text{xxx} \rangle$  represents nonterminal  $\text{xxx}$ .

## Generating Inputs: double-digits

- $\langle \text{start} \rangle \rightarrow \langle \text{digit} \rangle \langle \text{digit} \rangle$
- visit first  $\langle \text{digit} \rangle$ , have 10 choices;  
choose randomly, say 1;  
replace  $\langle \text{digit} \rangle$  by 1.
- visit second  $\langle \text{digit} \rangle$ ,  
choose say 7 randomly;  
replace  $\langle \text{digit} \rangle$  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

- $\langle \text{start} \rangle \rightarrow \langle \text{expr} \rangle$
- visit  $\langle \text{expr} \rangle$ ; three  $\langle \text{expr} \rangle$  alternatives,  
randomly choose  $\langle \text{term} \rangle + \langle \text{expr} \rangle$ ,  
replace  $\langle \text{expr} \rangle$  by our choice.
- visit  $\langle \text{term} \rangle$ ; also three  $\langle \text{term} \rangle$  alternatives,  
randomly choose  $\langle \text{factor} \rangle$ .
- visit  $\langle \text{factor} \rangle$ ;  
randomly choose  $\langle \text{integer} \rangle$  of the 5 alternatives.
- visit  $\langle \text{integer} \rangle$ ; randomly choose  $\langle \text{digit} \rangle$ ;
- visit  $\langle \text{digit} \rangle$ ;  
randomly choose terminal 4 and generate it.
- continue with next nonterminal  $\text{expr} \dots$

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>.

Rest of the string interpreted 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>"
```

Because 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.

# Termination for Simple Grammar Fuzzer

Let's look at this more closely:

```
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
```

If too many nonterminals after substitution:  
give up on generated string, re-choose.

# 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:

- $\langle \text{symbol} \rangle ?$ :  $\langle \text{symbol} \rangle$  can occur 0 or 1 times;
- $\langle \text{symbol} \rangle ^+$ :  $\langle \text{symbol} \rangle$  can occur 1 or more times;
- $\langle \text{symbol} \rangle ^*$ :  $\langle \text{symbol} \rangle$  can occur 0 or more times;
- parentheses can be used with these shortcuts, e.g.  
 $(\langle s1 \rangle \langle s2 \rangle) ^+$

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

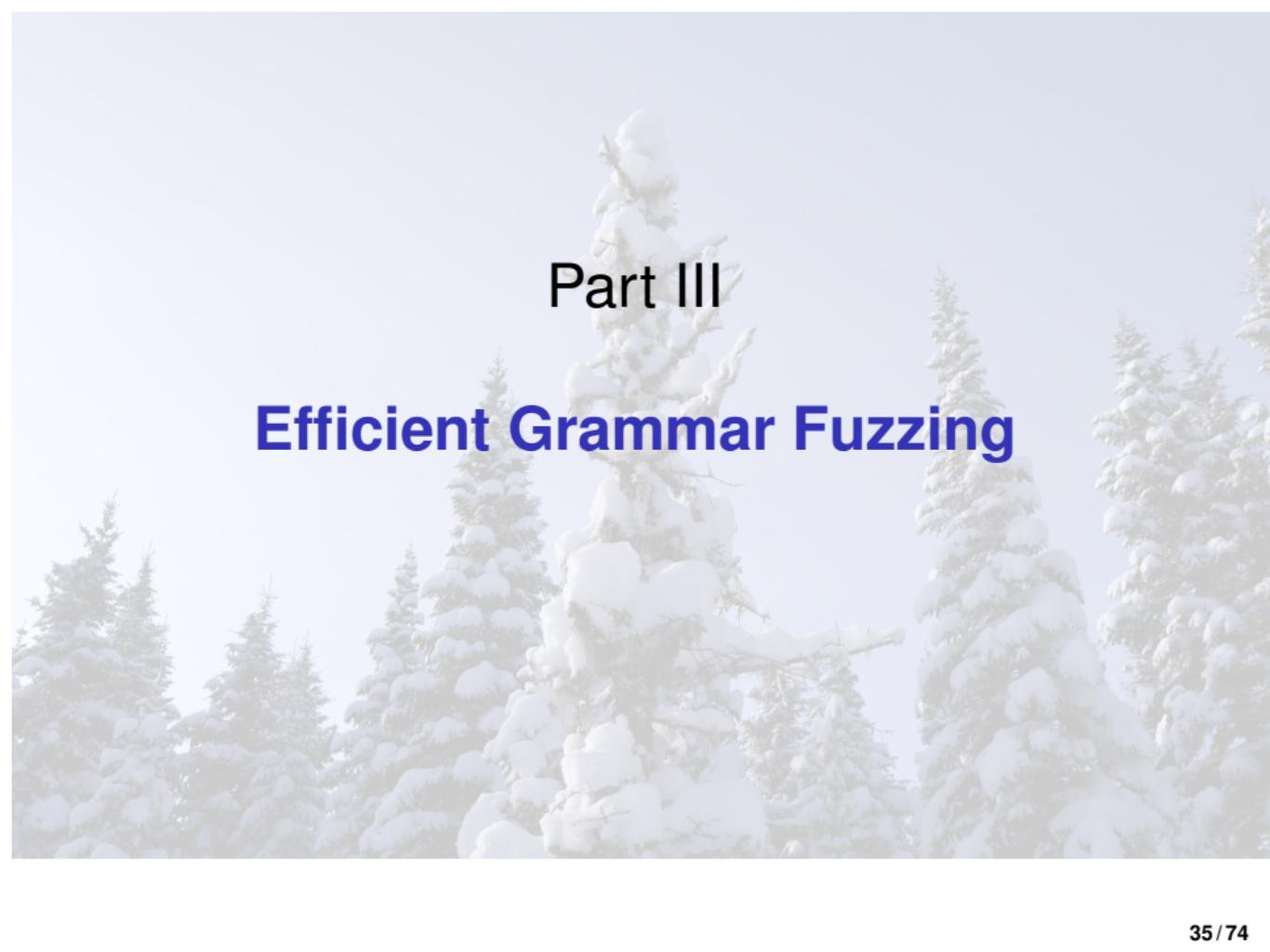
found in `code/L09/grammars.py`:

`trim_grammar()`: remove unneeded expansions;

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

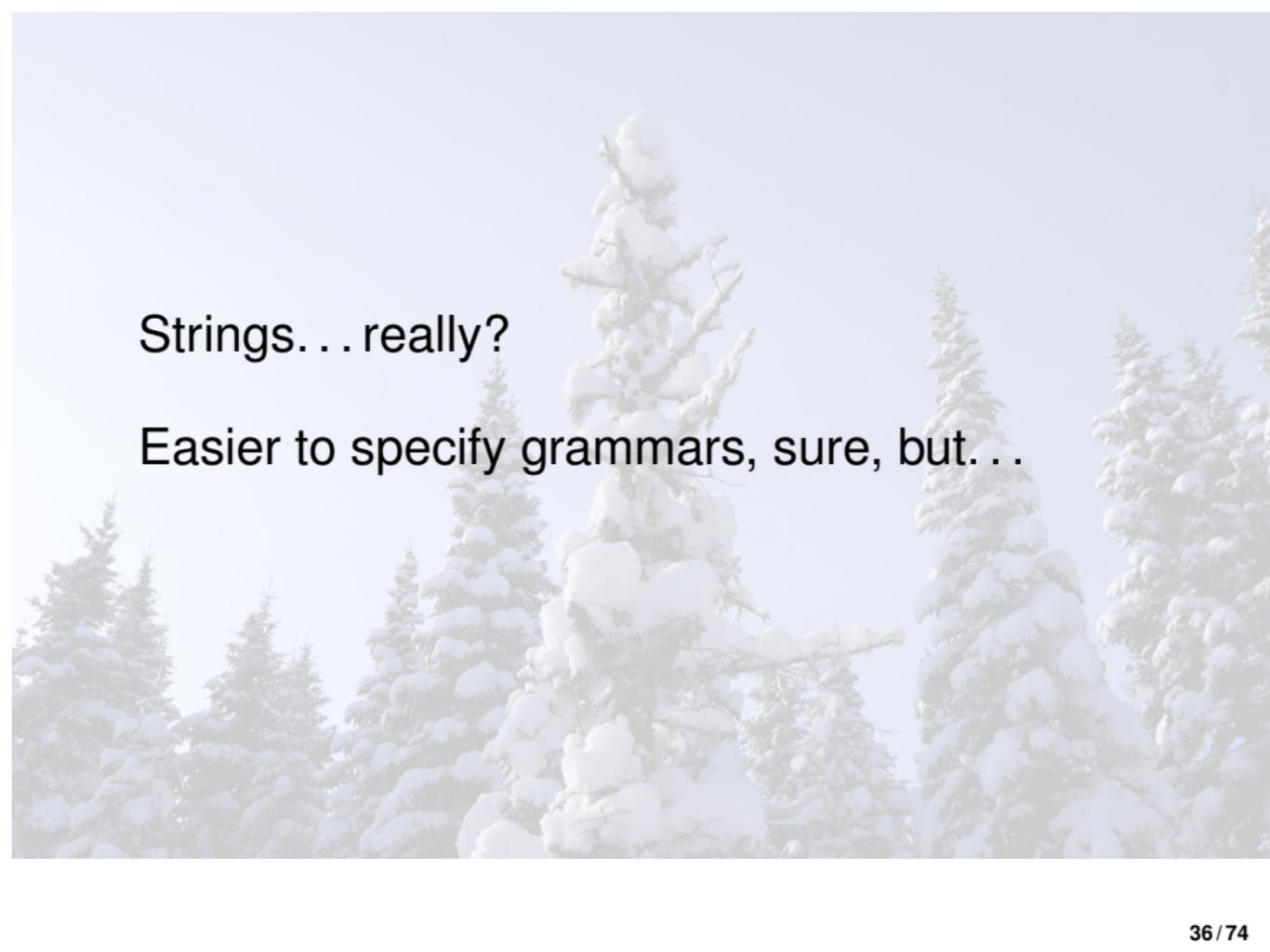
# Applications of Grammars in Testing

- 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.



# Part III

## Efficient Grammar Fuzzing

A photograph of a forest of tall, thin evergreen trees heavily laden with white snow. The trees are silhouetted against a very bright, almost featureless sky, creating a high-contrast scene where the details of the branches and needles are visible through the light.

Strings... really?

Easier to specify grammars, sure, but...

# A Simple Problem

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

does indeed raise a `TimeoutError`.

(see:  
L09/simple\_grammar\_fuzzer\_problem.py)

11

# ))) Why?

Rule for <factor>:

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

# Termination for Simple Grammar Fuzzer

Let's look at this more closely:

```
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
```

If too many nonterminals after substitution:  
give up on generated string, re-choose.

## simple\_grammar\_fuzzer fails

simple\_grammar\_fuzzer() throws away generated strings with > 3 nonterminals.

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

So, only allowed alternative for `<factor>` is  
`'(<expr>)'`...

... other alternatives add a non-terminal, so the fuzzer won't.

## Not enough control surfaces

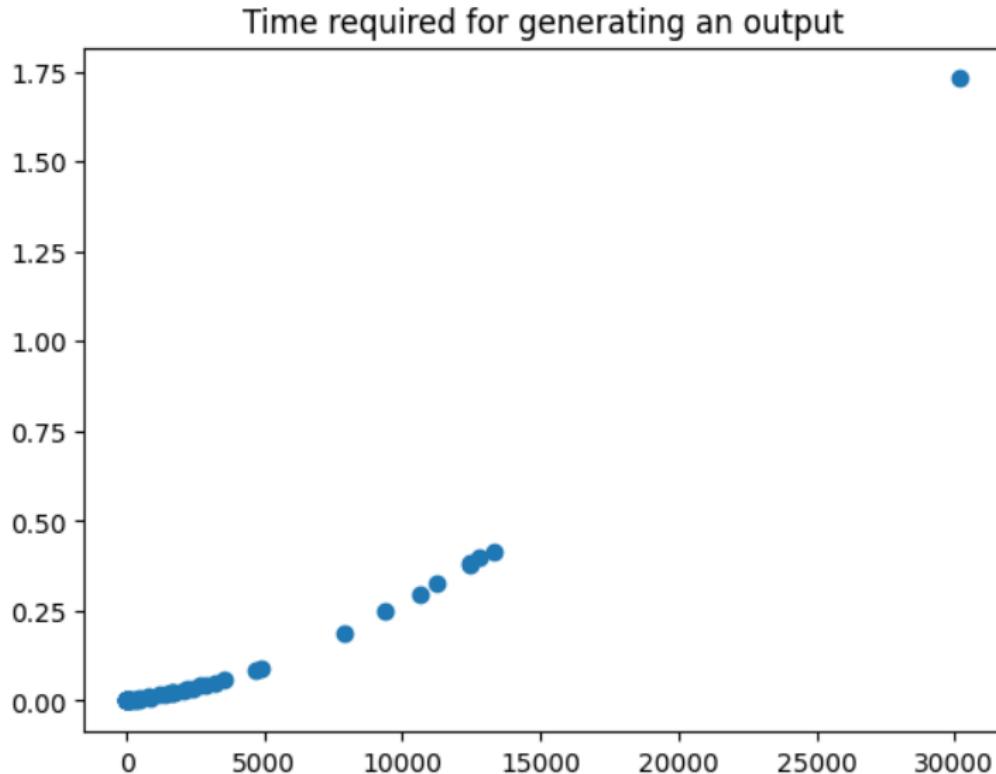
Any grammar-based approach can create arbitrarily long strings.

But `simple_grammar_fuzzer()` is too depth-first; we want more breadth.

The `max_nonterminals` control is too coarse.

## Too slow

Also, we don't want to manipulate strings.

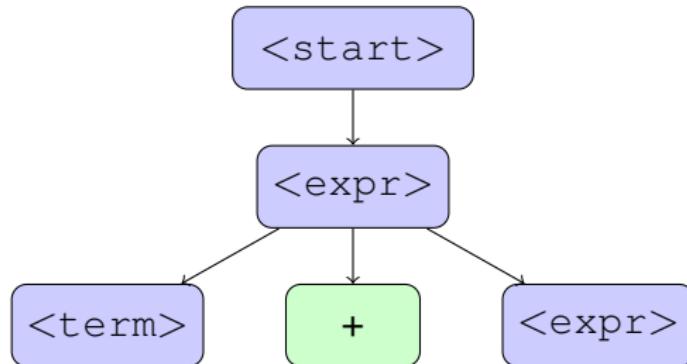


# Better controls?

A photograph of a forest scene. The foreground and middle ground are filled with tall evergreen trees, their branches heavily laden with thick white snow. The sky above is a clear, pale blue. In the upper left quadrant of the image, the word "Trees?" is overlaid in a large, bold, blue sans-serif font.

Trees?

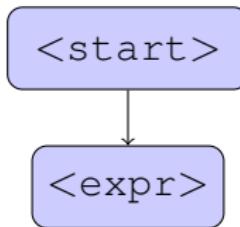
# Derivation Trees!



# Starting at the start

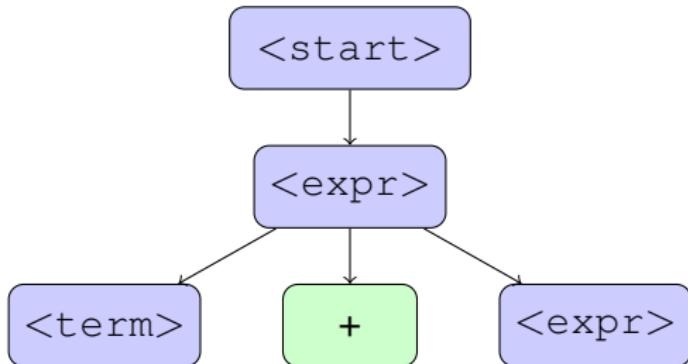
<start>

## Derivation: $\langle \text{start} \rangle \rightarrow \langle \text{expr} \rangle$

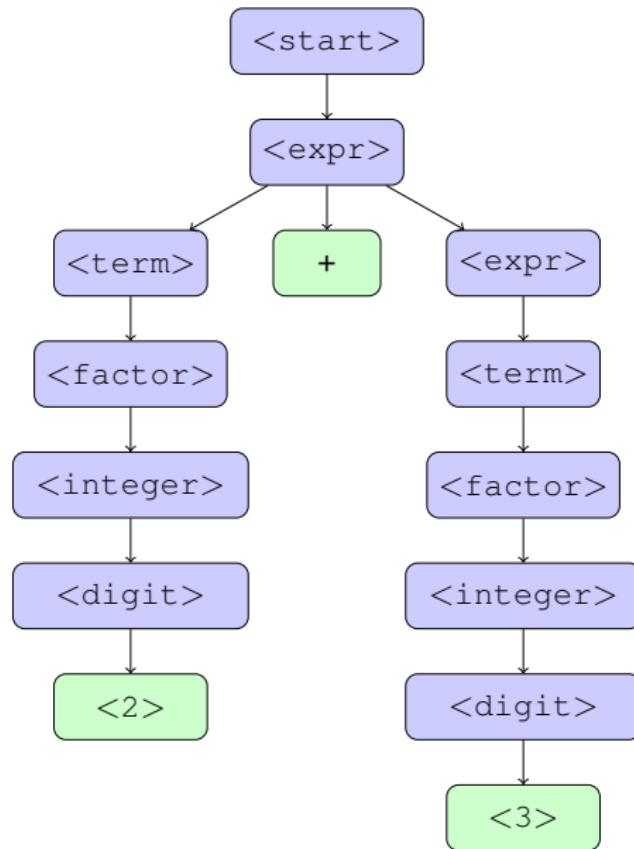


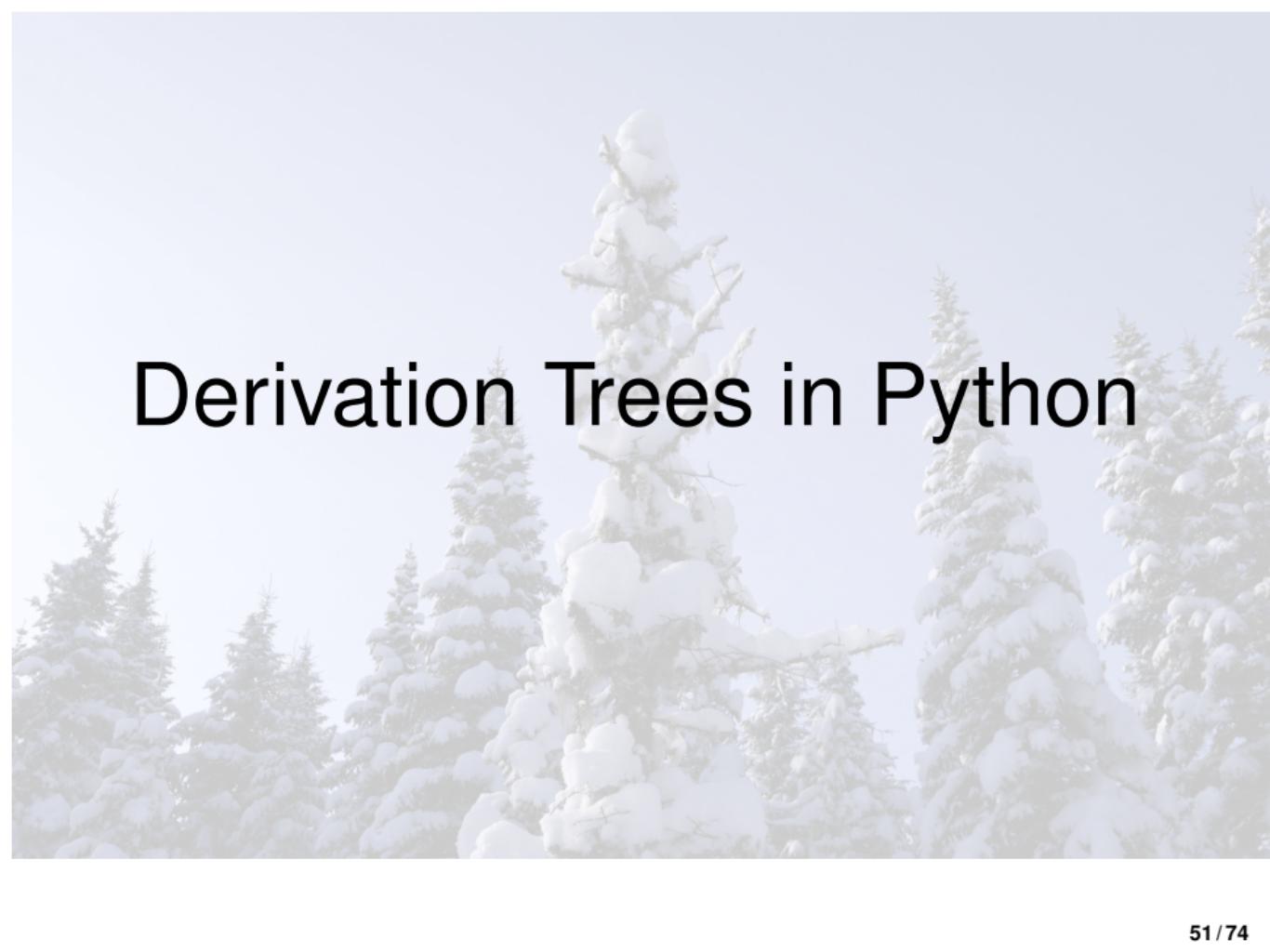
Search for nonterminal  $S$  without children;  
here,  $\langle \text{start} \rangle$ . Replace with its child  $\langle \text{expr} \rangle$ .

## Derivation: $\langle \text{expr} \rangle \rightarrow \langle \text{term} \rangle + \langle \text{expr} \rangle$



# Down to terminals



A photograph of a forest of tall, thin evergreen trees completely covered in a thick layer of white snow. The branches are heavily laden, and the overall scene is one of a heavy winter snowfall.

# Derivation Trees in Python

# Representing derivation trees

(SYMBOL\_NAME, CHILDREN)

**SYMBOL\_NAME:** e.g. <start>, +  
(again, nonterminals in <brackets>)

**CHILDREN:** the subtree at that node

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

## Example Derivation Tree

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

## Allowed CHILDREN

- the subtree
- None: this is a nonterminal still to be expanded
- []: this node is a terminal

# GrammarFuzzer Implementation

```
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:
        # ...
```

For now, only use `grammar` and `start_symbol`.

# Partial check for grammar validity

```
def check_grammar(self) -> None:  
    assert self.start_symbol in self.grammar  
    # recall: e.g. every nonterminal is reachable etc  
    assert is_valid_grammar(  
        self.grammar,  
        start_symbol=self.start_symbol,  
        supported_opts=self.supported_opts())
```

# Tree manipulations

Better than string manipulations.

We want to expand nonterminals (selectively).

Alternatives in grammars still expressed as strings, e.g.

```
"<expr>":  
    [ ("<term> + <expr>") ,  
      ("<term> - <expr>") ,  
      "<term>" ]
```

# Strings to trees

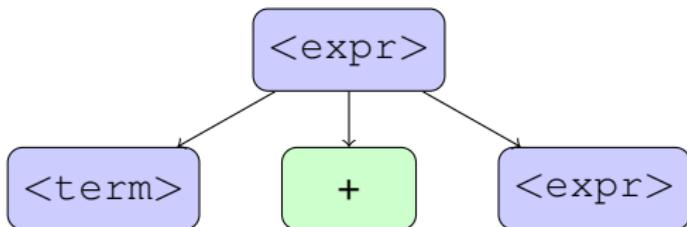
```
def expansion_to_children(expansion: Expansion) ->
    List[DerivationTree]:
    expansion = exp_string(expansion)
    assert isinstance(expansion, str)

    if 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]

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

## Expanding a node



Key operation: expand nonterminal into one of its alternatives; here we have chosen +, but could also be - or a term.

We replaced  $\langle \text{expr} \rangle$  with e.g.  $\langle \text{term} \rangle + \langle \text{expr} \rangle$ .

## Randomly choosing an alternative

Same choice strategy as `simple_grammar_fuzzer`.

```
def choose_node_expansion(self, node: DerivationTree,  
    children_alternatives: List[List[DerivationTree]]) -> int:  
    return random.randrange(0, len(children_alternatives))
```

## Random expansion implementation

For a subtree with non-expanded non-terminal at root (`children == None`):

- expand non-terminal into one of its alternatives (randomly chosen),
- return the tree.

```
def expand_node_randomly(self, node: DerivationTree) ->
                           DerivationTree:
    (symbol, children) = node
    assert children is None

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

## Random expansion implementation 2

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

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

expand\_node\_\* return a fresh DerivationTree,  
consistent with functional programming style.

Randomly is for now our only strategy; we will later  
redefine expand\_node to use other strategies.

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

## expand\_node in action

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

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

>>> expr_tree = f.expand_node_randomly(expr_tree)
>>> print("After expand_node_randomly():")
>>> print(expr_tree)
('<integer>', [('<digit>', None), ('<integer>', None)])
```

or also possible:

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

expand\_node\_\* return a fresh DerivationTree,  
consistent with functional programming style.

Randomly is for now our only strategy; we will later  
redefine expand\_node to use other strategies.

## Picking a node to expand

`expand_node()`: return expanded current node.

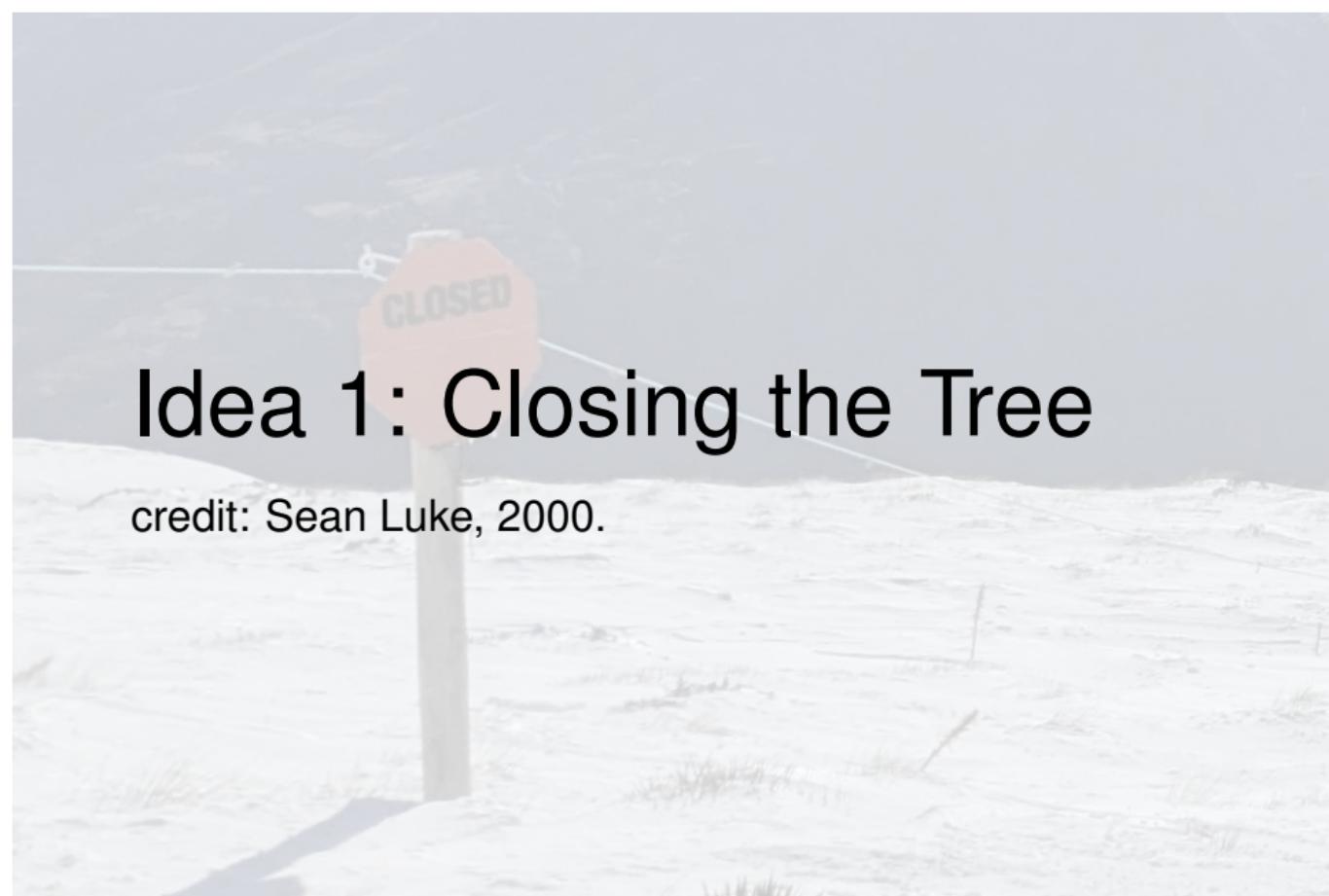
`expand_tree_once()`: modify the tree; pick some node and call `expand_node()` on that.

Specifically: if root is unexpanded, expand that. Otherwise, find a random descendant and expand that.

NB: uses the current definition of `self.expand_node`, which can change.

## expand\_tree\_once() example

```
>>> f.expand_node = f.expand_node_randomly
>>> derivation_tree = ("<start>",
                      [("<expr>",
                        [("<expr>, None), (" + ", []),
                         ("<term>, None)])])
>>> print(derivation_tree)
('<start>', [('<expr>', [('<expr>, None), (' + ', []),
                           ('<term>, None)]))]
>>> f.expand_tree_once(derivation_tree)
Expanding <term> randomly
>>> f.expand_tree_once(derivation_tree)
Expanding <expr> randomly
>>> print(derivation_tree)
('<start>', [('<expr>', [('<expr>', [('<term>, None),
                           (' + ', []),
                           ('<expr>, None)]), (' + ', []),
                           ('<term>, [('<factor>, None), (' / ', []),
                           ('<term>, None)]))]))
```



# Idea 1: Closing the Tree

credit: Sean Luke, 2000.

## How to close the tree?

Once our tree is big enough,  
choose expansions that increase tree size  
least.

e.g. **<factor>** could be  
**<integer>(.<integer>)?**,  
**which could be <integer>**,  
**which could be a single <digit>**.

(Other alternatives are higher-cost).

# Choosing the minimum-cost expansion

Traverse the tree & sum the minima.

```
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

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

    return sum(self.symbol_cost(s, seen) for s in symbols) + 1
```

## Minimum-cost expansion commentary

```
def symbol_cost(self, symbol: str, seen: Set[str] = set()) ->
    Union[int, float]:
```

is the minimum cost of the children of symbol.

```
def expansion_cost(self, expansion: Expansion,
    seen: Set[str] = set()) -> Union[int, float]:
```

if expansion has no nonterminals, return 1;  
if any nonterminal in expansion is in seen,  
return inf;  
else, return  $1 + \sum \text{symbol\_cost}(\text{children})$ .

## Minimum-cost expansion examples

```
>>> f = GrammarFuzzer(EXPR_GRAMMAR)
>>> f.symbol_cost("<digit>")
1
>>> f.symbol_cost("<expr>")
5
```

The min cost of `<expr>` is 5, through  
`<term>`, `<factor>`, `<integer>`,  
`<digit>`, and 1.

## Beyond random: parametrizing by algorithm

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

This functional-style method:

- collects min cost of alternatives;
- asks `choose` to choose a desired cost;
- collects all alternatives with chosen cost;
- uses `choose_node_expansion()` to choose an alternative  
(default: `random`);
- returns chosen alternative.

# Using min as choice algorithm

```
def expand_node_min_cost(self, node: DerivationTree) ->
    DerivationTree:
    return self.expand_node_by_cost(node, min)
```

and using it:

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

Expanding <term> at minimum cost
Expanding <factor> at minimum cost
Expanding <term> at minimum cost
...
>>> print(all_terminals(derivation_tree))
5 + 5 + 6 / 1
```

## Why closing works

- `expand_node_min_cost()` does not increase the number of symbols,
- so all open expansions are eventually closed.

I think this is a property of the grammar as well as the implementation.

# Node Inflation

To Be Continued