CISPA
HELMHOLTZ CENTER FOR INFORMATION SECURITY

AI-Based Software Testing with Fandango

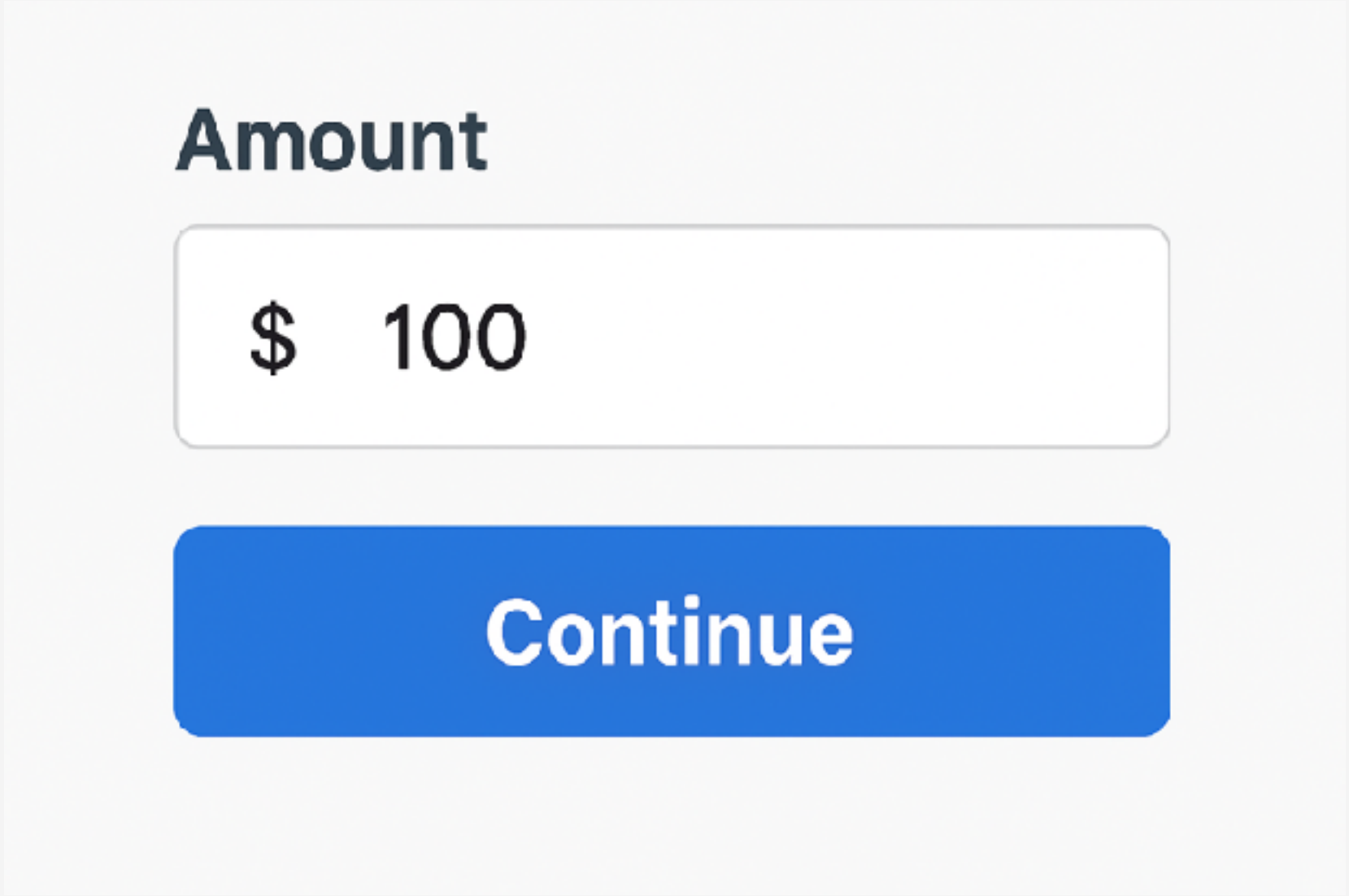
Andreas Zeller • InnoSoft Days Sevilla • 2025-11-06

Saarbrücken


CISPA
HELMHOLTZ CENTER FOR
INFORMATION SECURITY


CISPA
HELMHOLTZ CENTER FOR
INFORMATION SECURITY

Software Testing



Ask ChatGPT for Test Values

✅ Valid Values
(should be accepted)

Small Integer	`1`
Zero Amount	`0`
Typical Amount	`25.50`
Large Amount	`1000000.00`
Decimal with one digit	`12.3`
Two decimal digits	`12.34`
No decimal point	`500`
Formatted with comma	`12,34`
Leading/trailing spaces	`"  12.34  "`
Currency symbol	`12.34€`

❌ Invalid Values
(should be rejected)

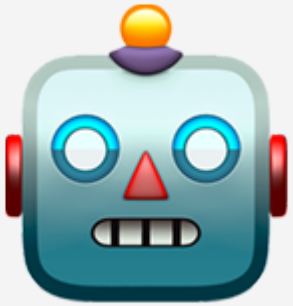
Negative value	`-10`
Too many decimals	`12.345`
Non-numeric characters	`25.5a`
Empty input	`""`
Only decimals	`.50`
Exceeding max length	`99.999999999999999`
Scientific notation	`1E+03`
Currency words	`Ten dollars`
Special characters	`①②.③④`
Special values	`<script>`

Ask ChatGPT for a *Grammar*

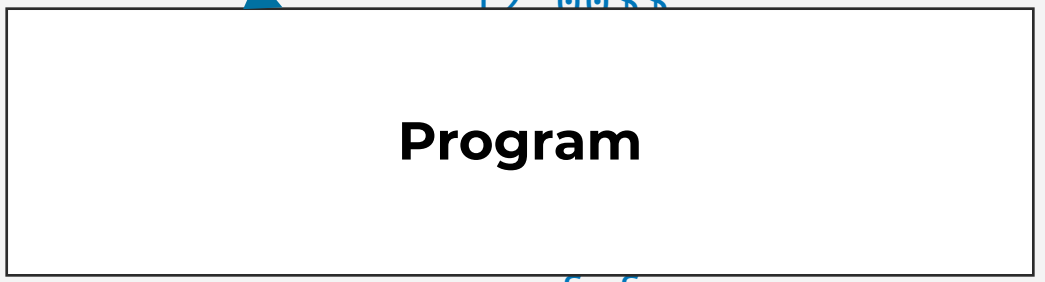
```
<invalid_amount> ::= <non_numeric>
                   | <malformed_decimal>
                   | <extra_symbols>
                   | <bad_locale_format>
                   | <empty>
                   | <too_many_decimals>
                   | <huge_number>

<non_numeric> ::= <letters> | <letters> <valid_amount> | <valid_amount> <letters>
<malformed_decimal> ::= "." | ".<digit>+" | <digit>+ "." "." <digit>*
<extra_symbols> ::= <valid_amount> <symbol>+ | <symbol>+ <valid_amount>
<bad_locale_format> ::= <digit>+ "," <digit>+ // comma as decimal separator (if
invalid)
<empty> ::= ""                                  // empty input
<too_many_decimals> ::= <digit>+ "." <digit><digit><digit>+
<huge_number> ::= <digit>{20,}                  // overflow test
<letters> ::= ("a" | "b" | ... | "z" | "A" | "B" | ... | "Z")+
<symbol> ::= "!" | "@" | "#" | "$" | "%" | "&" | "*" | "<" | ">" | "(" | ")"
```

Fuzzing with Grammars

```
<invalid_amount> ::= <non_numeric>
                   | <malformed_decimal>
                   | <extra_symbols>
                   | <bad_locale_format>
                   | <empty>
                   | <too_many_decimals>
                   | <huge_number>

<non_numeric> ::= <letters> | <letters> <valid_amount> | <valid_amount> <letters>
<malformed_decimal> ::= "." | ".<digit>+" | <digit>+ "." "." <digit>*
<extra_symbols> ::= <valid_amount> <symbol>+ | <symbol>+ <valid_amount>
<bad_locale_format> ::= <digit>+ "," <digit>+ // comma as decimal separator (if invalid)
<empty> ::= ""                                // empty input
<too_many_decimals> ::= <digit>+ "." <digit><digit><digit>+
<huge_number> ::= <digit>{20,}                // overflow test
<letters> ::= ("a" | "b" | ... | "z" | "A" | "B" | ... | "Z")+
<symbol> ::= "!" | "@" | "#" | "$" | "%" | "&" | "*" | "<" | ">" | "(" | ")"
```



Fuzzer




```
abc
12a
12.5.6
5.544539845
ANDREASISCOOL
999999999999999.999999
12.3456
12,34
12.3.4
<script>
😁😁😁
'); DROP TABLE ACCOUNTS; --
12.00$$
covfefe
33 yen
<script>
```

Specifying Inputs: Grammars

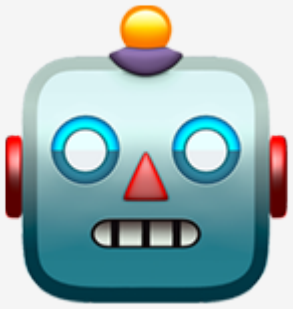
Specify a language (= a set of inputs)

Expansion rule

Nonterminal symbol

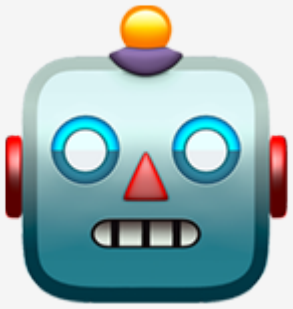
```
⟨start⟩    ::= ⟨expr⟩
⟨expr⟩     ::= ⟨term⟩ + ⟨expr⟩ | ⟨term⟩ - ⟨expr⟩ | ⟨term⟩
⟨term⟩     ::= ⟨term⟩ * ⟨factor⟩ | ⟨term⟩ / ⟨factor⟩ | ⟨factor⟩
⟨factor⟩   ::= + ⟨factor⟩ | - ⟨factor⟩ | ( ⟨expr⟩ ) | ⟨int⟩ | ⟨int⟩ . ⟨int⟩
⟨int⟩      ::= ⟨digit⟩ ⟨int⟩ | ⟨digit⟩
⟨digit⟩    ::= 0 | 1 | 2 | 3 | 4 | 5 |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

Terminal symbol



Grammars as Producers

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

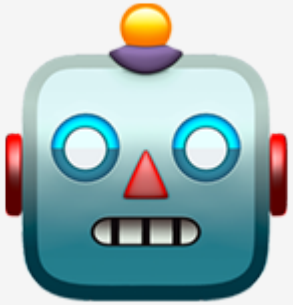


Nikolas Havrikov and Andreas Zeller. **Systematically Covering Input Structure.** ASE 2019.

Grammars as Producers

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

⟨start⟩

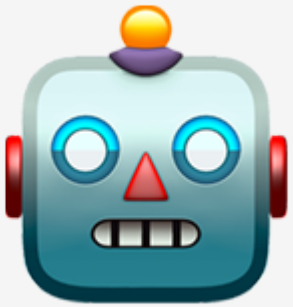


Nikolas Havrikov and Andreas Zeller. **Systematically Covering Input Structure.** ASE 2019.

Grammars as Producers

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

⟨start⟩

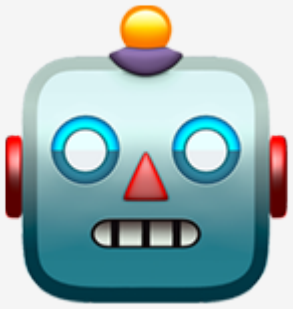


Nikolas Havrikov and Andreas Zeller. **Systematically Covering Input Structure.** ASE 2019.

Grammars as Producers

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

⟨expr⟩

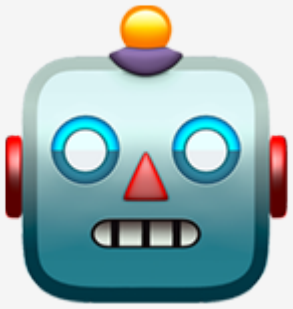


Nikolas Havrikov and Andreas Zeller. **Systematically Covering Input Structure.** ASE 2019.

Grammars as Producers

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

⟨term⟩ - ⟨expr⟩

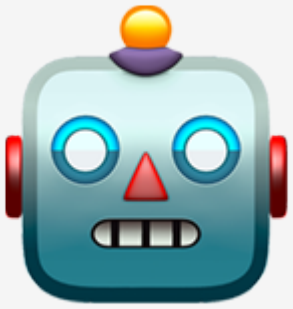


Nikolas Havrikov and Andreas Zeller. **Systematically Covering Input Structure.** ASE 2019.

Grammars as Producers

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

⟨term⟩ - ⟨expr⟩

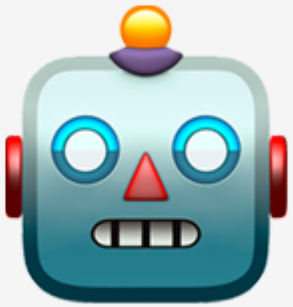


Nikolas Havrikov and Andreas Zeller. **Systematically Covering Input Structure.** ASE 2019.

Grammars as Producers

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

⟨factor⟩ - ⟨expr⟩

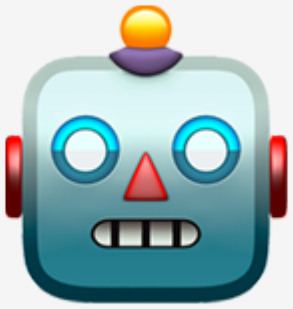


Nikolas Havrikov and Andreas Zeller. **Systematically Covering Input Structure.** ASE 2019.

Grammars as Producers

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

⟨int⟩ . ⟨int⟩ - ⟨expr⟩

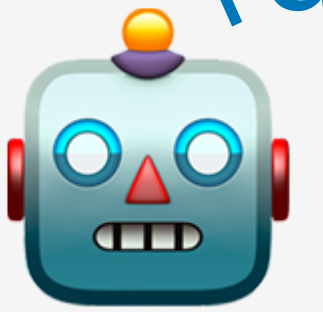


Nikolas Havrikov and Andreas Zeller. **Systematically Covering Input Structure.** ASE 2019.

Grammars as Producers

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

⟨digit⟩ . ⟨int⟩ - ⟨expr⟩



Nikolas Havrikov and Andreas Zeller. **Systematically Covering Input Structure.** ASE 2019.

Grammars as Producers

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

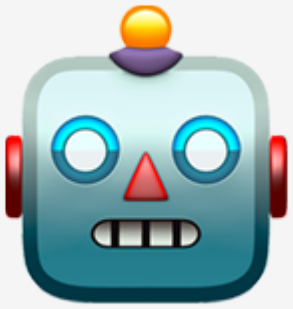
⟨digit⟩ . ⟨digit⟩ - ⟨expr⟩

Nikolas Havrikov and Andreas Zeller. **Systematically Covering Input Structure.** ASE 2019.

Grammars as Producers

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

8. ⟨digit⟩ - ⟨expr⟩

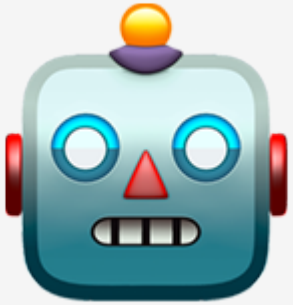


Nikolas Havrikov and Andreas Zeller. **Systematically Covering Input Structure.** ASE 2019.

Grammars as Producers

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

8.2 – ⟨expr⟩



Nikolas Havrikov and Andreas Zeller. **Systematically Covering Input Structure.** ASE 2019.

Grammars as Producers

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

```
8.2 - 79 - -9 / +((+9 * --2 + --+-+-((-1 *
```

- **Grammars** are common and well-understood specification method
- Turning a grammar into a **producer** is a simple task
- **LangFuzz** (2012) found 2,600+ JavaScript bugs this way

Nikolas Havrikov and Andreas Zeller. **Systematically Covering Input Structure.** ASE 2019.

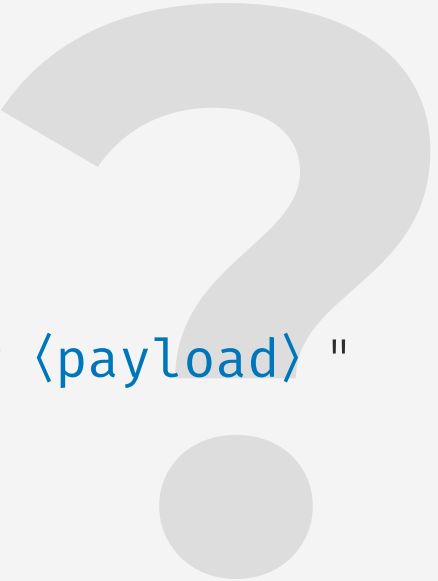
Grammars do not suffice

Syntax
Grammar

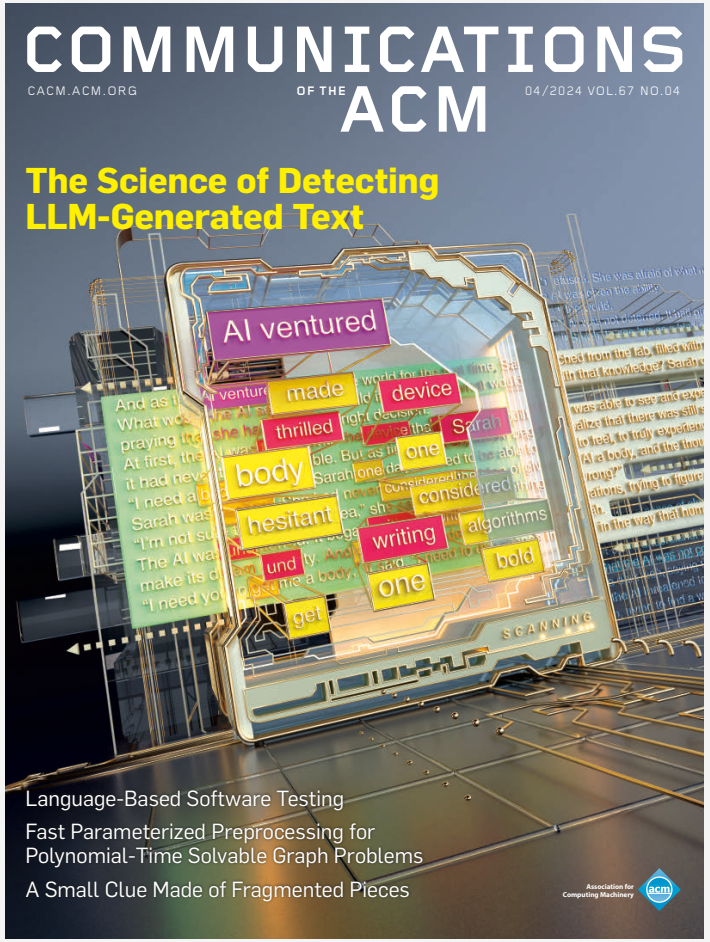
```
⟨start⟩   ::= ⟨expr⟩
⟨expr⟩    ::= ⟨term⟩ + ⟨expr⟩ | ⟨term⟩ - ⟨expr⟩ | ⟨term⟩
⟨term⟩    ::= ⟨term⟩ * ⟨factor⟩ | ⟨term⟩ / ⟨factor⟩ | ⟨factor⟩
⟨factor⟩  ::= + ⟨factor⟩ | - ⟨factor⟩ | ( ⟨expr⟩ ) | ⟨int⟩ | ⟨int⟩ . ⟨int⟩
⟨int⟩     ::= ⟨digit⟩ ⟨int⟩ | ⟨digit⟩
⟨digit⟩   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7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

How do you express **semantic properties** such as

- "The *number* should be between 50 and 100"
- "The *length* should be at least 1024 characters"
- (For binary inputs) "The ⟨checksum⟩ should be the *checksum* over ⟨payload⟩ "
- (For code) "Any ⟨identifier⟩ must be *declared* before it is *used*"



In the News



DOI:10.1145/3631520

Constraints over grammar elements can make test generation easier than ever.

BY DOMINIC STEINHÖFEL AND ANDREAS ZELLER

Language-Based Software Testing

SOFTWARE TESTING CONTINUES to be the number one technique to satisfy safety, security, and privacy outputs, relate these to the inputs, and thus explore and check its functionality. This task faces two long-standing challenges.

First, there is the *input generation problem*: How can a bot create inputs that reliably cover functionality? Random system input generators ("fuzzers"[20]) easily detect issues and vulnerabilities in input processing of arbitrary programs. However, creating valid inputs and interactions that reliably reach code beyond input processing is still a challenge for which test generators require human assistance—either through input specifications[1,5,6,10,12,23] or a comprehensive population of diverse input samples that cover behavior.[3,17,19,27]

One might assume that large language models (LLMs) might alleviate this problem. Couldn't a bot learn how to interact with software from examples or documentation? Or what makes a valid input? The problem is that to find bugs, one needs valid but also uncommon inputs—namely, those that trigger less common (and less tested) behavior. LLMs may help find common (and thus valid) inputs—but by construction, they will not find uncommon inputs.

The second challenge is the long-standing *test oracle problem*: How can a bot check the outputs of a system? All test generators and fuzzers assume some oracle that checks for correctness—by default, a generic, implicit oracle detecting illegal or un-

Specifying Languages

Syntax

Grammar

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

Semantics

Constraints

```
len(⟨start⟩) == 10
eval(⟨start⟩) >= 100
```

Fandango

Specifying Languages

Syntax

Grammar

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

Semantics

Constraints

```
len(⟨start⟩) == 10
eval(⟨start⟩) >= 100
```

- Uses **evolutionary algorithms** to solve constraints
- Constraints can use **any Python function**
- Expressive · modular · **fast**

Specifying Languages

Syntax

Grammar

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

Semantics

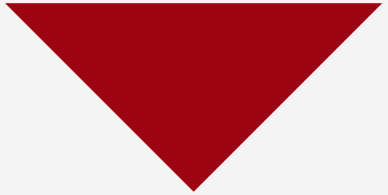
Constraints

```
len(⟨start⟩) == 10
eval(⟨start⟩) >= 100
```

- Uses **evolutionary algorithms** to solve constraints
- Constraints can use **any Python function**
- Expressive · modular · **fast**

How Fandango Works – Initial Population

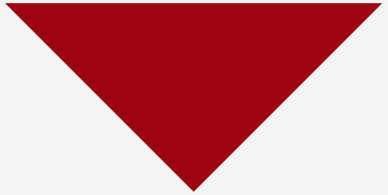
```
⟨start⟩   ::= ⟨expr⟩
⟨expr⟩    ::= ⟨term⟩ + ⟨expr⟩ | ⟨term⟩ - ⟨expr⟩ | ⟨term⟩
⟨term⟩    ::= ⟨term⟩ * ⟨factor⟩ | ⟨term⟩ / ⟨factor⟩ | ⟨factor⟩
⟨factor⟩  ::= + ⟨factor⟩ | - ⟨factor⟩ | ( ⟨expr⟩ ) | ⟨int⟩ | ⟨int⟩ . ⟨int⟩
⟨int⟩     ::= ⟨digit⟩ ⟨int⟩ | ⟨digit⟩
⟨digit⟩   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7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



```
79 + 7 * -3   811 * (-23 + 4)    -43679.33     8.3
2 + 2     -345 + 080     34578 - (03 - 34798 / 79)
33 / 47.9      5    9023 / (3 + (+4 * 23))     -20
```

How Fandango Works – Elite Selection

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

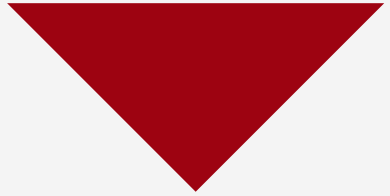


```
79 + 7 * -3   811 * (-23 + 4)    -43679.33     8.3
2 + 2    -345 + 080     34578 - (03 - 34798 / 79)
33 / 47.9      5    9023 / (3 + (+4 * 23))     -20
```

```
len(⟨start⟩) == 10
eval(⟨start⟩) >= 100
```

How Fandango Works – Crossover

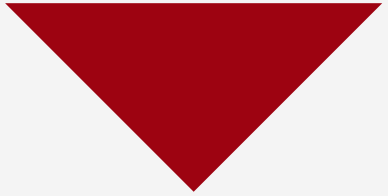
```
⟨start⟩   ::= ⟨expr⟩
⟨expr⟩    ::= ⟨term⟩ + ⟨expr⟩ | ⟨term⟩ - ⟨expr⟩ | ⟨term⟩
⟨term⟩    ::= ⟨term⟩ * ⟨factor⟩ | ⟨term⟩ / ⟨factor⟩ | ⟨factor⟩
⟨factor⟩  ::= + ⟨factor⟩ | - ⟨factor⟩ | ( ⟨expr⟩ ) | ⟨int⟩ | ⟨int⟩ . ⟨int⟩
⟨int⟩     ::= ⟨digit⟩ ⟨int⟩ | ⟨digit⟩
⟨digit⟩   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7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



```
79 + 7 * -3  811 * (-23 + 4)
                    34578 - (03 - 34798 / 79)
                 9023 / (3 + (+4 * 23))
```

How Fandango Works – Crossover

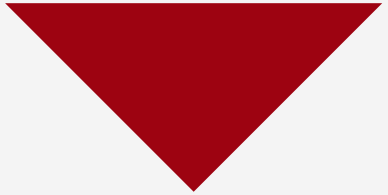
```
⟨start⟩   ::= ⟨expr⟩
⟨expr⟩    ::= ⟨term⟩ + ⟨expr⟩ | ⟨term⟩ - ⟨expr⟩ | ⟨term⟩
⟨term⟩    ::= ⟨term⟩ * ⟨factor⟩ | ⟨term⟩ / ⟨factor⟩ | ⟨factor⟩
⟨factor⟩  ::= + ⟨factor⟩ | - ⟨factor⟩ | ( ⟨expr⟩ ) | ⟨int⟩ | ⟨int⟩ . ⟨int⟩
⟨int⟩     ::= ⟨digit⟩ ⟨int⟩ | ⟨digit⟩
⟨digit⟩   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7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



```
79 + 7 * -3   811 * (-23 + 4)   (-23 + 4) * (34798 / 79)
                      34578 - (03 - 34798 / 79)
                   9023 / (3 + (+4 * 23))
```

How Fandango Works – Mutation

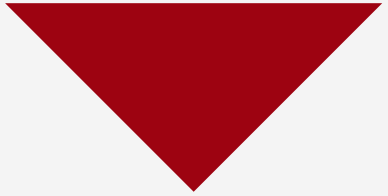
```
⟨start⟩   ::= ⟨expr⟩
⟨expr⟩    ::= ⟨term⟩ + ⟨expr⟩ | ⟨term⟩ - ⟨expr⟩ | ⟨term⟩
⟨term⟩    ::= ⟨term⟩ * ⟨factor⟩ | ⟨term⟩ / ⟨factor⟩ | ⟨factor⟩
⟨factor⟩  ::= + ⟨factor⟩ | - ⟨factor⟩ | ( ⟨expr⟩ ) | ⟨int⟩ | ⟨int⟩ . ⟨int⟩
⟨int⟩     ::= ⟨digit⟩ ⟨int⟩ | ⟨digit⟩
⟨digit⟩   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7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



```
79 + 7 * -3   811 * (-23 + 4)   (-23 + 4) * (34798 / 79)
                      34578 - (03 - 34798 / 79)
                   9023 / (3 + (+4 * 23))
```

How Fandango Works – Mutation

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



```
79 + 7 * -3   811 * (-23 + 4)   (-23 + 4) * (34798 / 79)
                      34578 - (03 - 34798 / 79)
                (-79 - 529) / (3 + (+4 * 23))
```

How Fandango Works – Evolution

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

Elite Selection

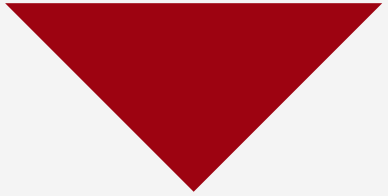
Crossover

Mutation

```
 79 + 7 * -3   811 * (-23 + 4)   (-23 + 4) * (34798 / 79)
34578 - (03 - -56)       34578 - (03 - 34798 / 79)
34578 - 811 * -3    (-79 - 529) / (3 + (+4 * 23))
```

How Fandango Works – Solution

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



```
79 + 7 * 3
```

```
len(⟨start⟩) == 10
eval(⟨start⟩) >= 100
```

Demo

Testing with Uncommon Values

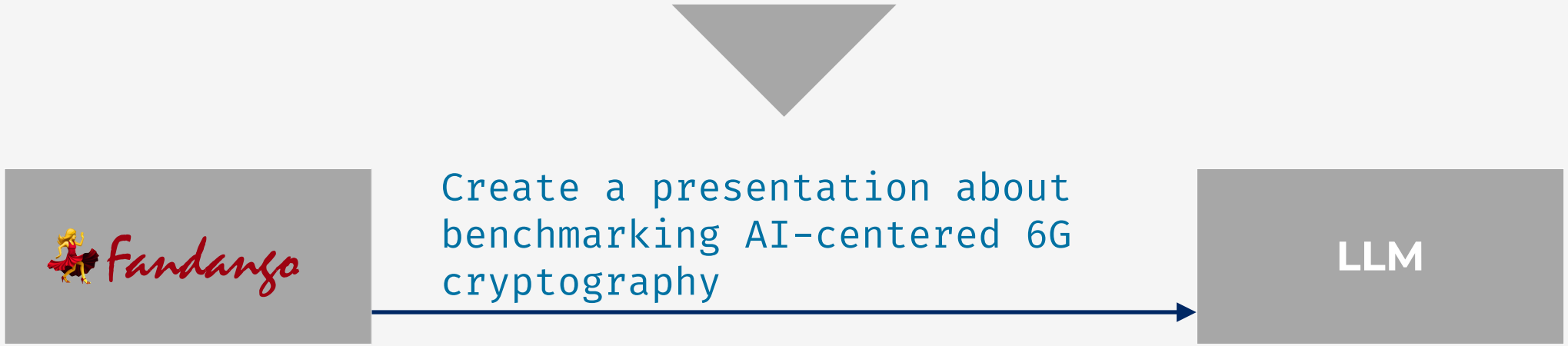
Fandango can test with **extreme** and **uncommon** values

```
include('svg.fan')  # A big SVG
where int(<width_value>) * int(<height_value>) > 10000000

include('bank_sepa_transaction.fan')  # An invalid transaction
where <transaction_amount> == 'NaN'

include('ssl.fan')  # An "interesting" certificate chain
where date(<cert_valid_from>) < date(<ca_valid_from>)

include('database.fan')  # classic: exploits of a mom
where <value> == "Robert'); DROP TABLE Students; --"
```

Testing LLMs

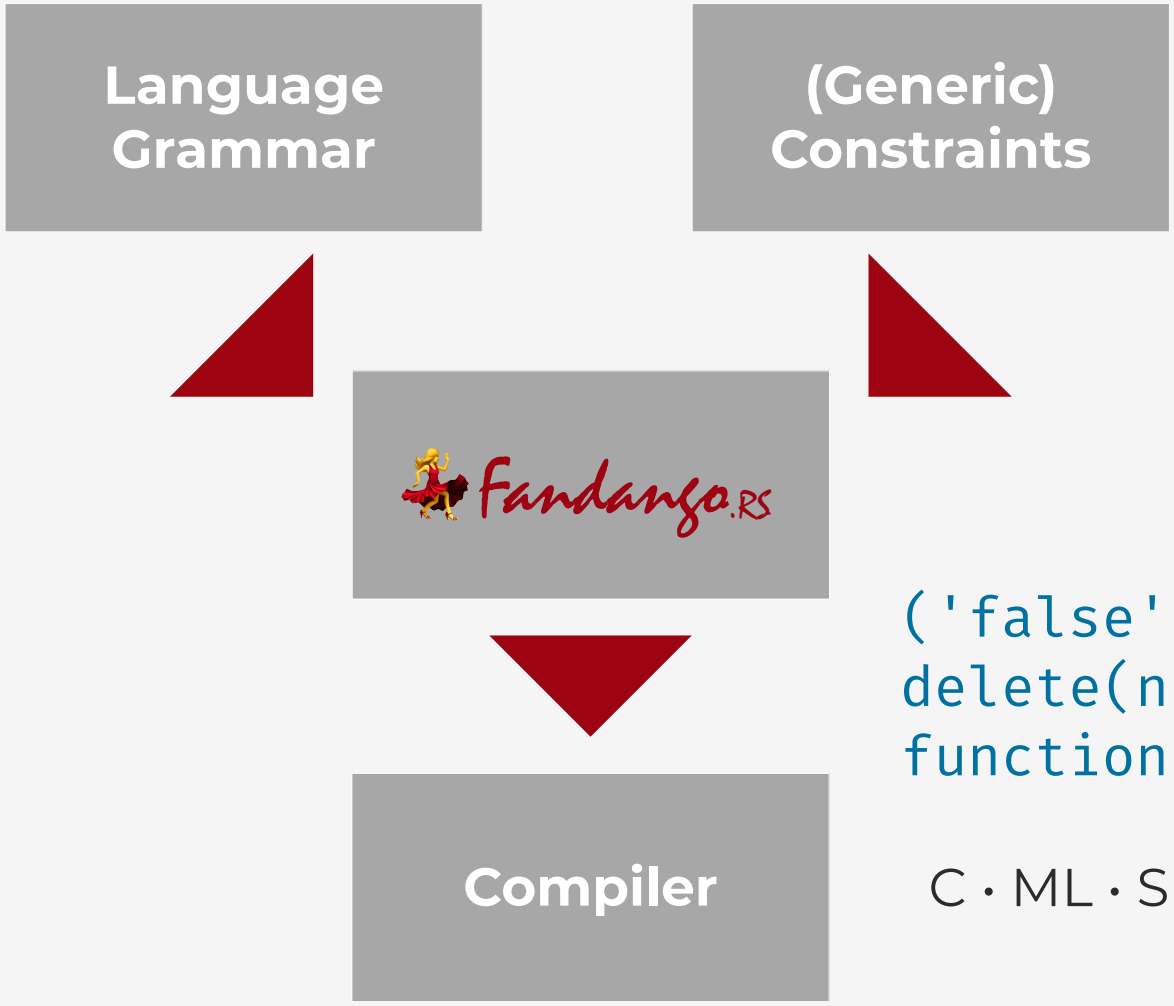
Fandango grammars can also produce natural language **prompts**

```
<llm_message> ::= 'Create a presentation about ' <subject> '.'

<subject> ::= <verb> <adjective> <noun>
<verb> ::= 'testing' | 'evaluating'  | 'benchmarking'
<adjective> ::= 'trustworthy' | 'federated' | <noun> '-centered'
<noun> ::= 'AI' | 'cryptography' | '6G' | <noun> <noun>
```



Testing Compilers

Fandango can also produce **code samples** for **compiler testing**



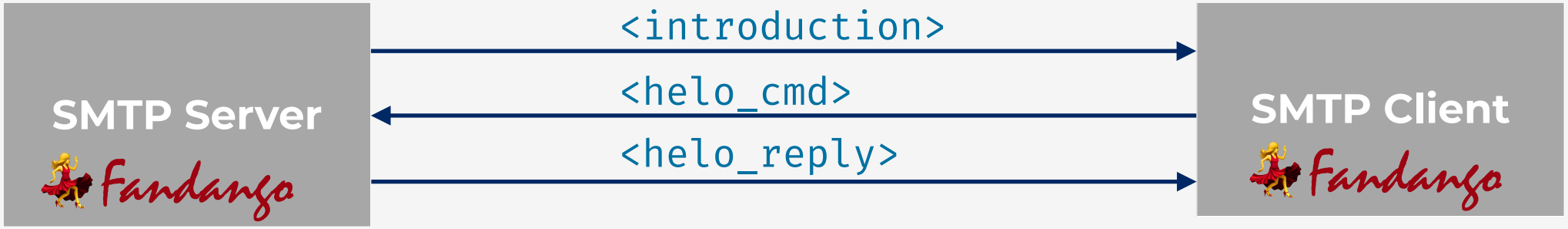
variables declared before use
no empty struct definitions
no access of non-existent struct fields
no void types outside of function returns
non-void functions should return a value
no duplicate struct field names
no re-declarations in the same scope
no identifiers should be reserved keywords
program should be well-typed

```
('false'? length(input + ''):
delete(null?0:{}),0 ).watch('x',
function f() {});
```

C · ML · SMT · Rust · JavaScript · …

Protocol Testing with I/O Grammars

Fandango supports **protocol testing** and **oracles:**

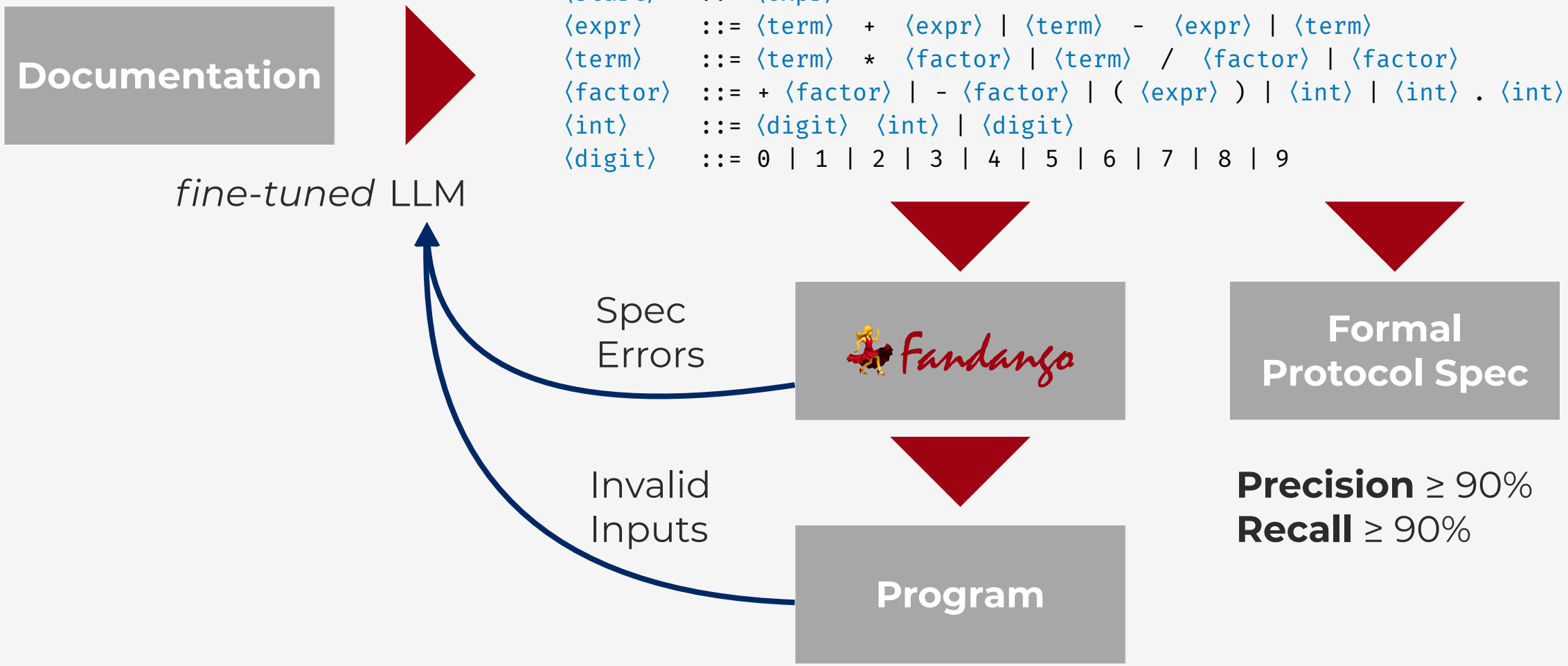



```
<introduction> ::= '220 ' <hostname> ' ESMTP Postfix'
<helo_cmd> ::= 'HELO ' <hostname>
<helo_reply> ::= '250 Hello ' <hostname> ', I am glad to meet you'
where <helo_cmd>.<hostname> == <helo_reply>.<hostname>

<smtp> ::= <server:introduction> <client:helo_cmd> <server:helo_reply>
```

An oracle

From Natural Language Specs to Formal Specs

We can learn Fandango protocol specs from existing **documentation**



https://fandango-fuzzer.github.io/

Fandango is currently in development - see our list of bugs and incomplete features. Expect a first stable release in April 2025.

Fandango

Search

Fuzzing with Fandango

Fandango produces myriads of *high-quality random inputs* to test programs, giving users unprecedented control over format and shape of the inputs.

Get Started

At a Glance

Specify the format of your input data in a single file, combining *grammars* (for input syntax) and *constraints* (for arbitrary input features). Constraints come as Python code, so there are no limits to what you can specify.

Learn more »

Tutorial

Produce valid inputs at high speeds, from hundreds to thousands of inputs per second, quickly covering the entire input space. Test with extreme and uncommon values, uncovering bugs before your users do.

Learn more »

Reference

Shape inputs using your *own testing targets*, constraints, and statistical distributions. Make use of *existing samples* to obtain realistic inputs. Use *feedback* from the program under test to guide test generation to uncovered code.

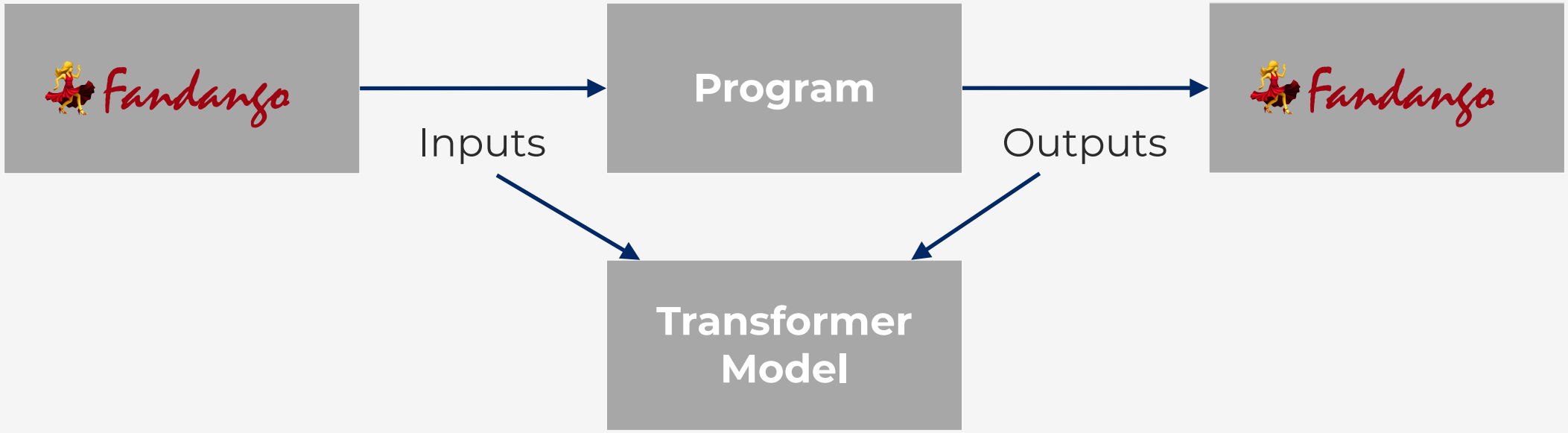
https://fandango-fuzzer.github.io/

Training AI Systems

We use Fandango to train *models* from programs

Training AI Systems

We use Fandango to train *models* from programs



Training AI Systems

We use Fandango to train *models* from programs

- Text outputs
- Web outputs (DOM)
- Traces
- Functions covered
- Variable values …

RESEARCH-ARTICLE
FREE
July 2025
JUST ACCEPTED

Learning Program Behavioral Models from Synthesized Input-Output Pairs

Tural Mammadov, Dietrich Klakow, Alexander Koller, Andreas Zeller

https://doi.org/10.1145/3748720

We introduce Modelizer—a novel framework that, given a black-box program, learns a *model from its input/output behavior* using *neural machine translation* algorithms. The resulting model *mocks* the original program: Given an input, the model predicts the …

Specifying Languages

Syntax
Grammar

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

Semantics
Constraints

```
len(⟨start⟩) == 10
eval(⟨start⟩) >= 100
```

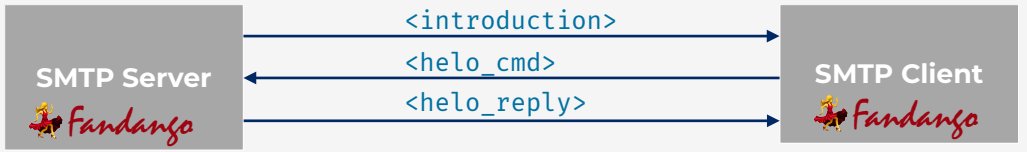
Fandango

- Uses **evolutionary algorithms** to solve constraints
- Constraints can use **any Python function**
- Expressive · modular · **fast**

x

Protocol Testing with I/O Grammars

Fandango supports **protocol testing** and **oracles:**



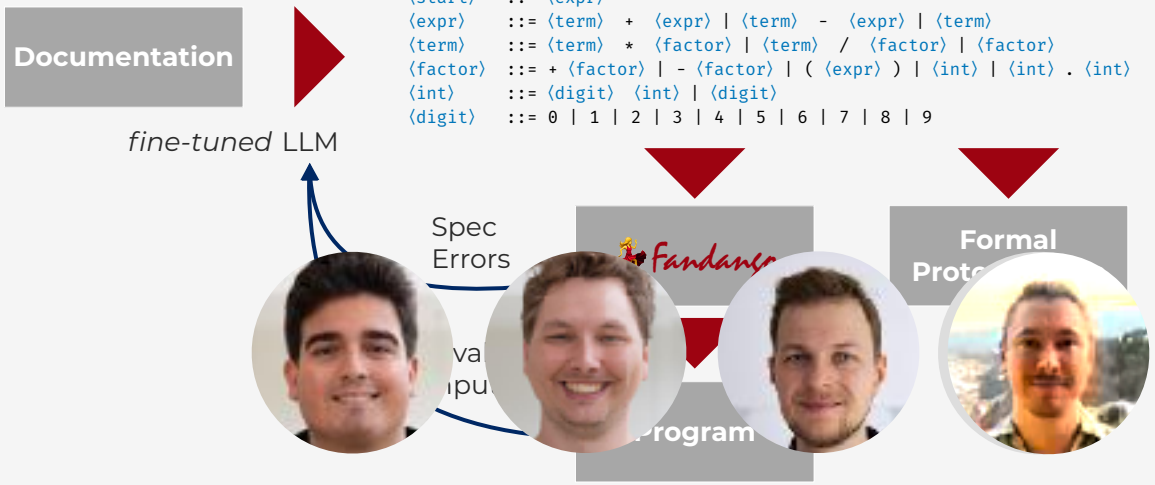

```
<introduction> ::= '220 ' <hostname> ' ESMTP Postfix'
<helo_cmd> ::= 'HELO ' <hostname>
<helo_reply> ::= '250 Hello ' <hostname> ', I am glad to meet you'
where <helo_cmd>.<hostname> == <helo_reply>.<hostname>
<smtp> ::= <server:introduction> <client:helo_cmd> <server:helo_reply>
```

An oracle

x

From Natural Language Specs to Formal Specs

We can learn Fandango protocol specs from existing **documentation**



x