Learning the Language of Failure

NUS Computer Science Research Week, January 8, 2021

Andreas Zeller
with Rahul Gopinath and Zeller's team at CISPA

CISPA

Learning the Language of Failure

Andreas Zeller, CISPA Helmholtz Center for Information Security
Joint work with Rahul Gopinath and Zeller's team at CISPA

Watch: https://www.youtube.com/watch?v=3ZW1DI2PxvI

When diagnosing why a program fails, one of the first steps is to precisely understand the *circumstances* of the failure – that is, when the failure occurs and when it does not. Such circumstances are necessary for three reasons. First, one needs them to precisely *predict when the failure takes place*; this is important to devise the severity of the failure. Second, one needs them to design a *precise fix*: A fix that addresses only a subset of circumstances is incomplete, while a fix that addresses a superset may alter behavior in non-failing scenarios. Third, one can use them to *create test cases* that reproduce the failure and eventually validate the fix.

In this talk, I present and introduce tools and techniques that automatically learn circumstances of a given failure, expressed over features of input elements. I show how to automatically infer input languages as readable grammars, how to use these grammars for massive fuzzing, and how to systematically and precisely characterize the set of inputs that causes a given failure – the "language of failure".

https://andreas-zeller.info/
https://www.cispa.saarland/

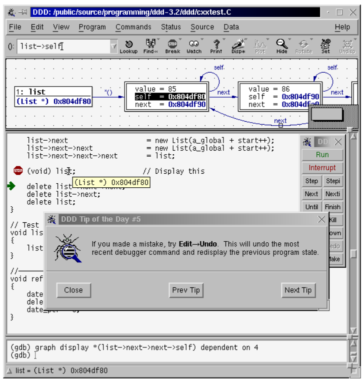
Failure

Welcome everyone to "Learning the Language of Failure". These five words will follow us throughout the talk. To begin, lets talk about failures.

Failure

Actually, my work always has been about failures. (The work itself has been less of a failure.)

Debugging Failures



DDD: /public/source/programming/ddd-3.2/ddd/cxxtest.C

```
(): list->self
```

```
1: list
(List *) 0x804df80
```

```
value = 85
self  = 0x804df80
next  = 0x804df90
```

```
value = 86
self  = 0x804df90
next  = 0x804df
```

```
    list->next               = new List(a_global + start++);
    list->next->next         = new List(a_global + start++);
    list->next->next->next   = list;

    (void) list;             // Display this

    delete list
    delete list->next;
    delete list;
}

// Test
void lis
{
    list
}

//
void ref
{
    date
    dele
    date
}
```

(List *) 0x804df80

DDD Tip of the Day #5

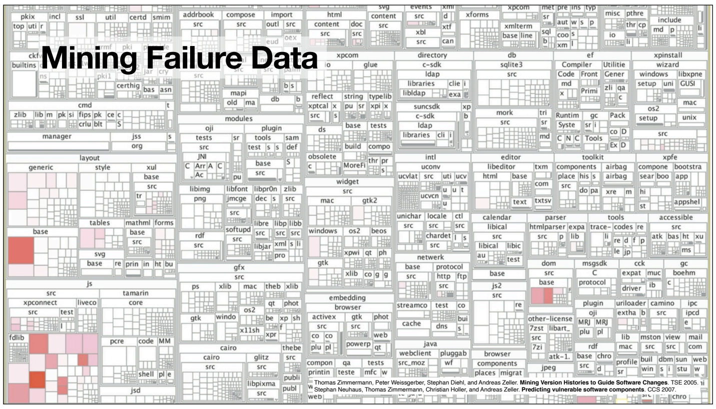
If you made a mistake, try **Edit→Undo**. This will undo the most recent debugger command and redisplay the previous program state.

```
(gdb) graph display *(list->next->next->self) dependent on 4
(gdb)
```

list = (List *) 0x804df80

Andreas Zeller and Dorothea Lütkehaus. **DDD—a free graphical front-end for UNIX debuggers.** SIGPLAN Notices 1996.

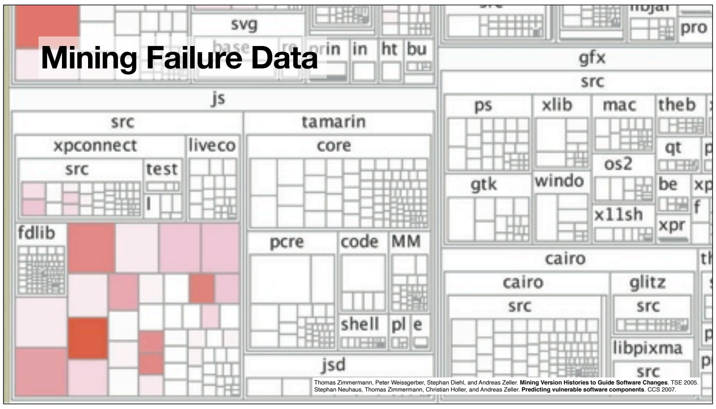
As a PhD student still, Dorothea Lütkehaus and myself built GNU DDD, a GUI front-end for command line debuggers. Great for debugging failures.



Mining Failure Data

Thomas Zimmermann, Peter Weissgerber, Stephan Diehl, and Andreas Zeller. **Mining Version Histories to Guide Software Changes**. TSE 2005.
Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. **Predicting vulnerable software components**. CCS 2007.

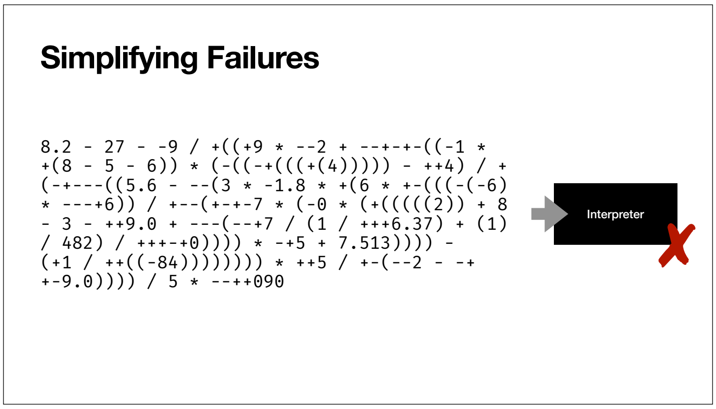
Later, my co-workers and I would mine version and bug repositories to see where in a program the most bugs would be fixed. This is a map of Firefox components (boxes) and vulnerabilities (shades of red).



Mining Failure Data

Thomas Zimmermann, Peter Weissgerber, Stephan Diehl, and Andreas Zeller. **Mining Version Histories to Guide Software Changes**. TSE 2005.
Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. **Predicting vulnerable software components**. CCS 2007.

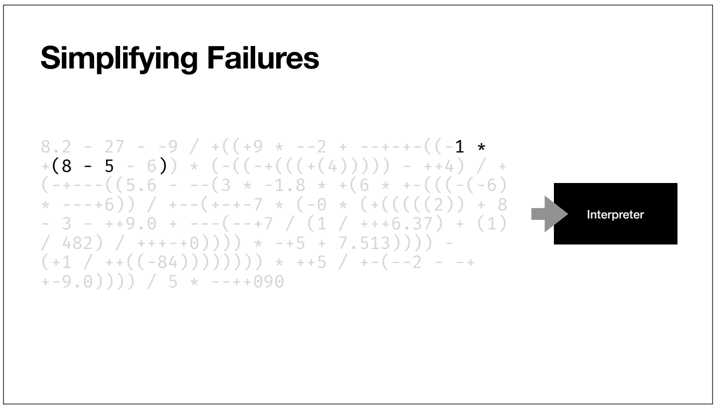
Almost all vulnerabilities are in JavaScript.



Simplifying Failures

```
8.2 - 27 - -9 / +((+9 * --2 + --+-+-((-1 *
+(8 - 5 - 6)) * (-((-+(((+(4)))))) - ++4) / +
(-+---((5.6 - --(3 * -1.8 * +(6 * +-(((-(-6)
* ---+6)) / +--(+-+-7 * (-0 * (+(((((2)) + 8
- 3 - ++9.0 + ---(--+7 / (1 / +++6.37) + (1)
/ 482) / +++-+0)))) * -+5 + 7.513)))) -
(+1 / ++((-84)))))))) * ++5 / +-(--2 - -+
+-9.0)))) / 5 * --++090
```

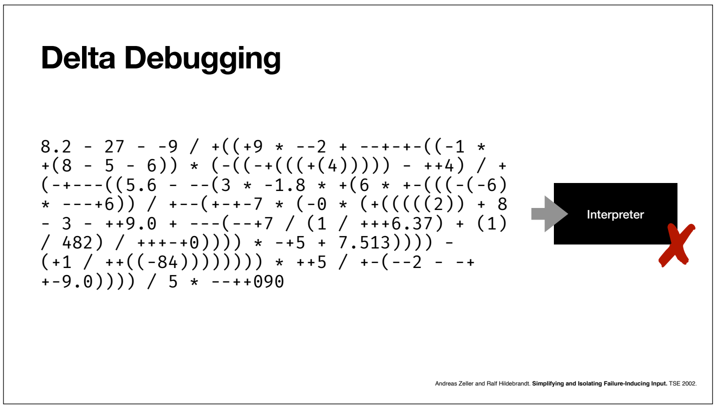
Another contribution my name is associated with is simplifying failure-inducing inputs. Here's a long input that causes a program to fail.



Simplifying Failures

```
8.2 - 27 - -9 / +((+9 * --2 + --+-+-((-1 *
+(8 - 5 - 6)) * (-((-+(((+(4)))))) - ++4) / +
(-+---((5.6 - --(3 * -1.8 * +(6 * +-(((-(-6)
* ---+6)) / +--(+-+-7 * (-0 * (+(((((2)) + 8
- 3 - ++9.0 + ---(--+7 / (1 / +++6.37) + (1)
/ 482) / +++-+0)))) * -+5 + 7.513)))) -
(+1 / ++((-84)))))))) * ++5 / +-(--2 - -+
+-9.0)))) / 5 * --++090
```

Yet, only a part of this input actually is relevant for the failure.

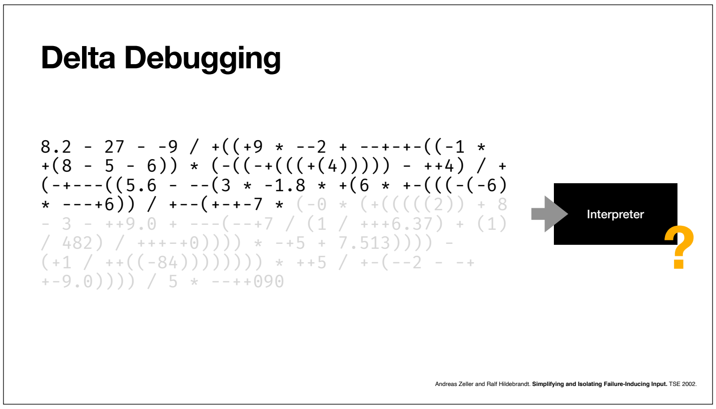


Delta Debugging

```
8.2 - 27 - -9 / +((+9 * --2 + --+-+-((-1 *
+(8 - 5 - 6)) * (-((-+(((+(4))))) - ++4) / +
(-+---((5.6 - --(3 * -1.8 * +(6 * +-(((-(-6)
* ---+6)) / +--(+-+-7 * (-0 * (+(((((2)) + 8
- 3 - ++9.0 + ---(--+7 / (1 / +++6.37) + (1)
/ 482) / +++-+0)))) * -+5 + 7.513)))) -
(+1 / ++((-84))))))) * ++5 / +-(--2 - -+
+-9.0)))) / 5 * --++090
```

Andreas Zeller and Ralf Hildebrandt. **Simplifying and Isolating Failure-Inducing Input.** TSE 2002.

Delta debugging automatically determines this failure-inducing subset.

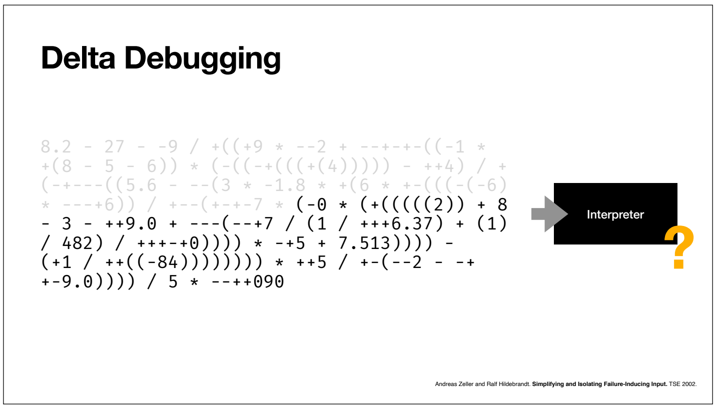


Delta Debugging

```
8.2 - 27 - -9 / +((+9 * --2 + --+-+-((-1 *
+(8 - 5 - 6)) * (-((-+(((+(4))))) - ++4) / +
(-+---((5.6 - --(3 * -1.8 * +(6 * +-(((-(-6)
* ---+6)) / +--(+-+-7 * (-0 * (+(((((2)) + 8
- 3 - ++9.0 + ---(--+7 / (1 / +++6.37) + (1)
/ 482) / +++-+0)))) * -+5 + 7.513)))) -
(+1 / ++((-84))))))) * ++5 / +-(--2 - -+
+-9.0)))) / 5 * --++090
```

Andreas Zeller and Ralf Hildebrandt. **Simplifying and Isolating Failure-Inducing Input.** TSE 2002.

Delta Debugging takes away parts of the input and checks whether the failure still occurs.

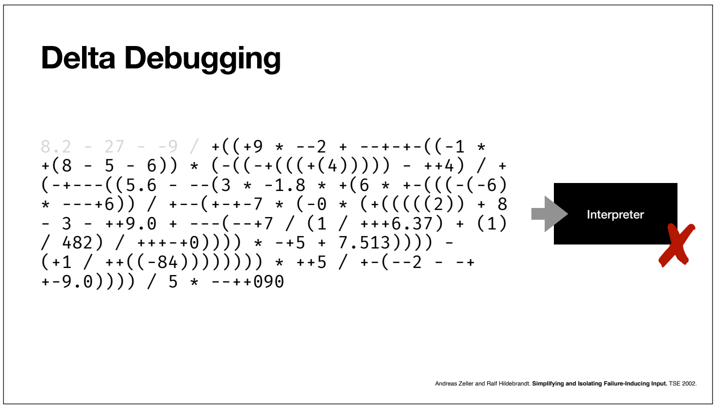


Delta Debugging

```
8.2 - 27 - -9 / +((+9 * --2 + --+-+-((-1 *
+(8 - 5 - 6)) * (-((-+(((+(4))))) - ++4) / +
(-+---((5.6 - --(3 * -1.8 * +(6 * +-(((-(-6)
* ---+6)) / +--(+-+-7 * (-0 * (+(((((2)) + 8
- 3 - ++9.0 + ---(--+7 / (1 / +++6.37) + (1)
/ 482) / +++-+0)))) * -+5 + 7.513)))) -
(+1 / ++((-84))))))) * ++5 / +-(--2 - -+
+-9.0)))) / 5 * --++090
```

Andreas Zeller and Ralf Hildebrandt. **Simplifying and Isolating Failure-Inducing Input.** TSE 2002.

Such reduced inputs can be invalid, though.

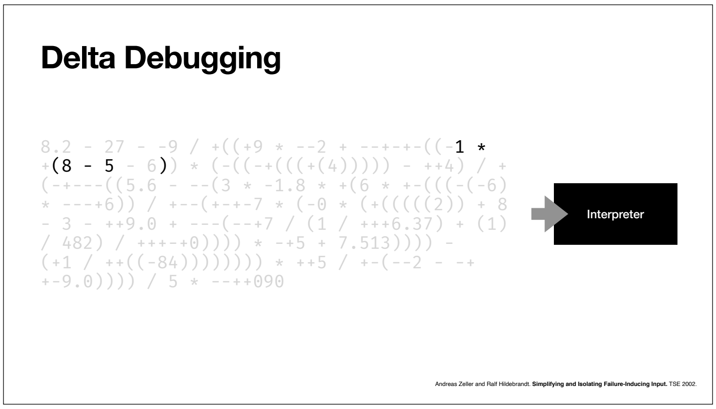


Delta Debugging

```
8.2 - 27 - -9 / +((+9 * --2 + --+-+-((-1 *
+(8 - 5 - 6)) * (-((-+(((+(4))))) - ++4) / +
(-+---((5.6 - --(3 * -1.8 * +(6 * +-(((-(-6)
* ---+6)) / +--(+-+-7 * (-0 * (+(((((2)) + 8
- 3 - ++9.0 + ---(--+7 / (1 / +++6.37) + (1)
/ 482) / +++-+0)))) * -+5 + 7.513)))) -
(+1 / ++((-84))))))) * ++5 / +-(--2 - -+
+-9.0)))) / 5 * --++090
```

Andreas Zeller and Ralf Hildebrandt. **Simplifying and Isolating Failure-Inducing Input.** TSE 2002.

Then, delta debugging takes out smaller parts and repeats.

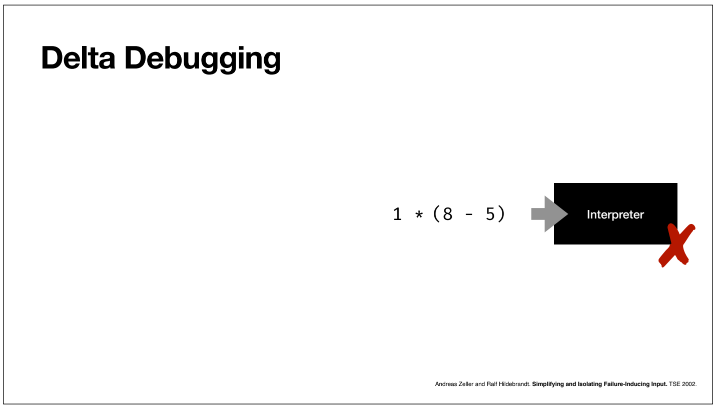


Delta Debugging

```
8.2 - 27 - -9 / +((+9 * --2 + --+-+-((-1 *
+(8 - 5 - 6)) * (-((-+(((+(4))))) - ++4) / +
(-+---((5.6 - --(3 * -1.8 * +(6 * +-(((-(-6)
* ---+6)) / +--(+-+-7 * (-0 * (+(((((2)) + 8
- 3 - ++9.0 + ---(--+7 / (1 / +++6.37) + (1)
/ 482) / +++-+0)))) * -+5 + 7.513)))) -
(+1 / ++((-84))))))) * ++5 / +-(--2 - -+
+-9.0)))) / 5 * --++090
```

Andreas Zeller and Ralf Hildebrandt. **Simplifying and Isolating Failure-Inducing Input.** TSE 2002.

At the end, it easily determines which characters are necessary for the failure to occur.

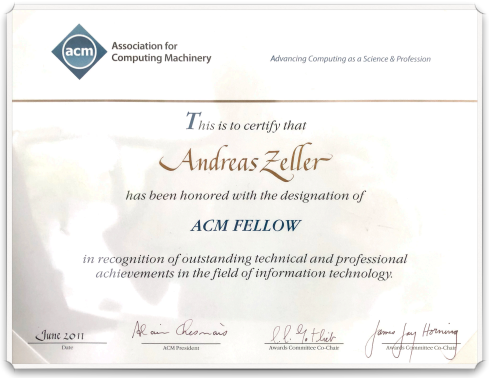


Delta Debugging

```
1 * (8 - 5)
```

Andreas Zeller and Ralf Hildebrandt. **Simplifying and Isolating Failure-Inducing Input.** TSE 2002.

Such as these ones, for instance.



Association for Computing Machinery

Advancing Computing as a Science & Profession

This is to certify that

Andreas Zeller

has been honored with the designation of

ACM FELLOW

in recognition of outstanding technical and professional achievements in the field of information technology.

June 2011 — Date

Alain Chesnais — ACM President

Awards Committee Co-Chair

James Jay Horning — Awards Committee Co-Chair

These things made me an ACM Fellow "For contributions to automated debugging and mining software archives".

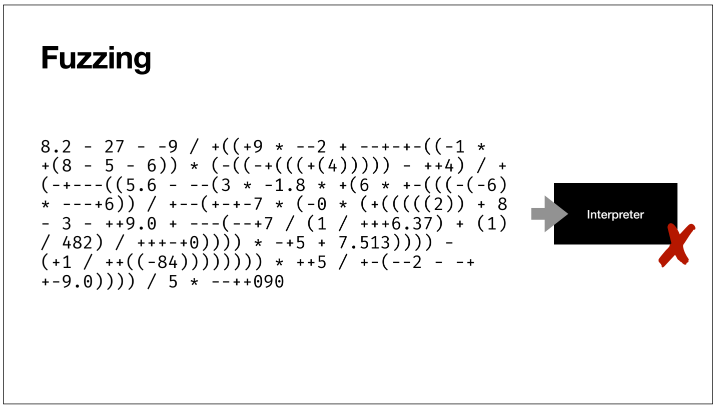
Failure

- You can *mine version and bug histories* to find out where the failures are
- You can *simplify* inputs to find out what causes the failure
- You can make a *career* out of failure

Which tells you that you can make a career out of failures.

The Language of Failure

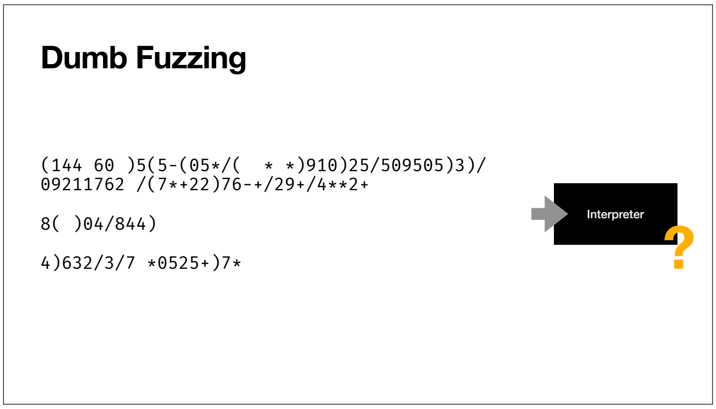
Okay, that was failures. Now, let's move to languages.



Fuzzing

```
8.2 - 27 - -9 / +((+9 * --2 + --+-+-((-1 *
+(8 - 5 - 6)) * (-((-+(((+(4)))))) - ++4) / +
(-+---((5.6 - --(3 * -1.8 * +(6 * +-(((-(-6)
* ---+6)) / +--(+-+-7 * (-0 * (+(((((2)) + 8
- 3 - ++9.0 + ---(--+7 / (1 / +++6.37) + (1)
/ 482) / +++-+0)))) * -+5 + 7.513)))) -
(+1 / ++((-84)))))))) * ++5 / +-(--2 - -+
+-9.0)))) / 5 * --++090
```

Fuzzing means to throw random inputs at a program to see if it crashes.



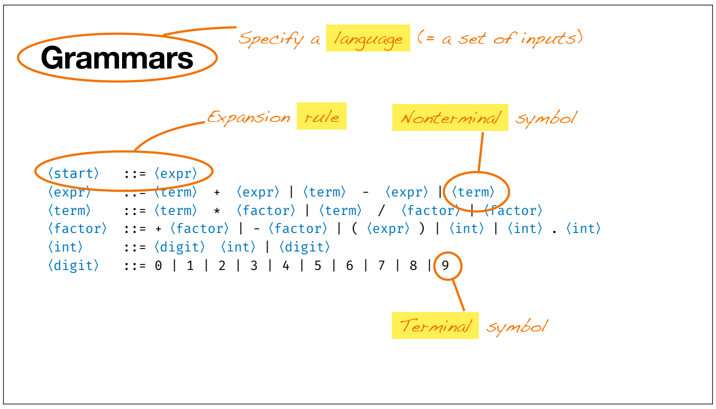
Dumb Fuzzing

```
(144 60 )5(5-(05*/(   * *)910)25/509505)3)/
09211762 /(7*+22)76-+/29+/4**2+

8(  )04/844)

4)632/3/7 *0525+)7*
```

But if you just take sequences of random characters and throw them at an interpreter, all you're going to get is syntax errors. (It's okay to test syntax error handling, but this should not be all.)



Grammars

Specify a language (= a set of inputs)

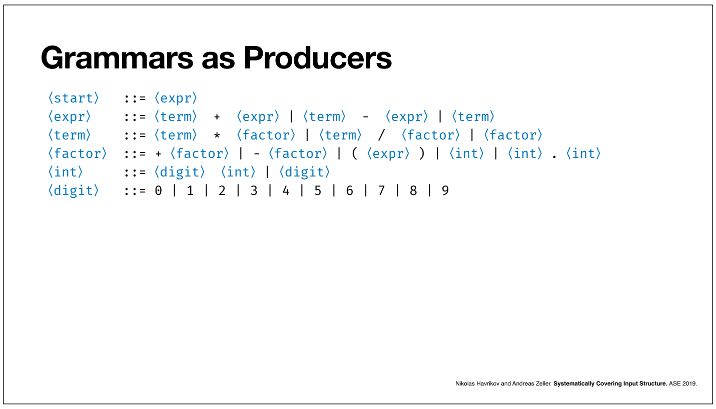
Expansion rule

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

In order to get syntactically valid inputs, you need a specification. A **grammar** specifies the set of inputs as a **language**.

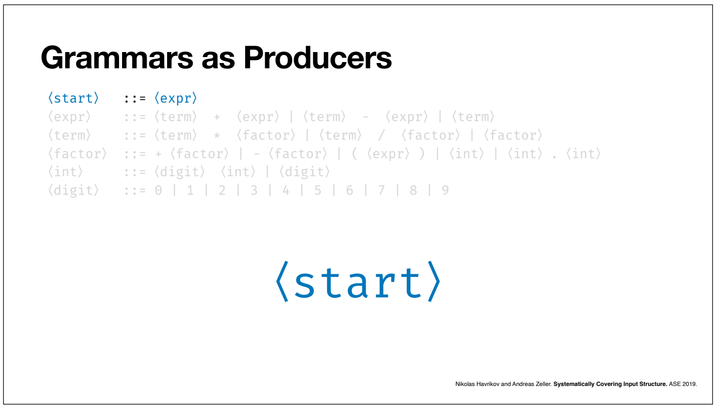


Grammars as Producers

```
⟨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.

You may have seen grammars as **parsers**, but they can also be used as **producers** of inputs.



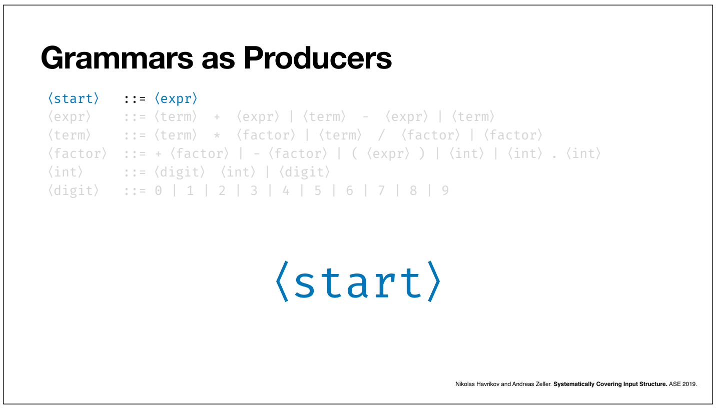
Grammars as Producers

```
⟨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.

You start with a start symbol

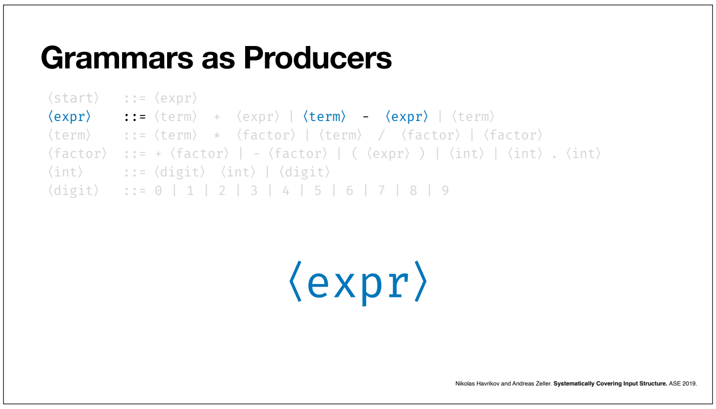


Grammars as Producers

```
⟨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
```

⟨start⟩

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



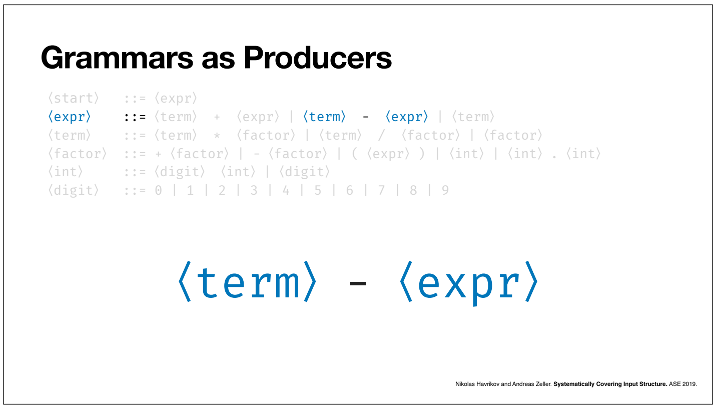
Grammars as Producers

```
⟨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.

which then subsequently gets replaced according to the production rules in the grammar.



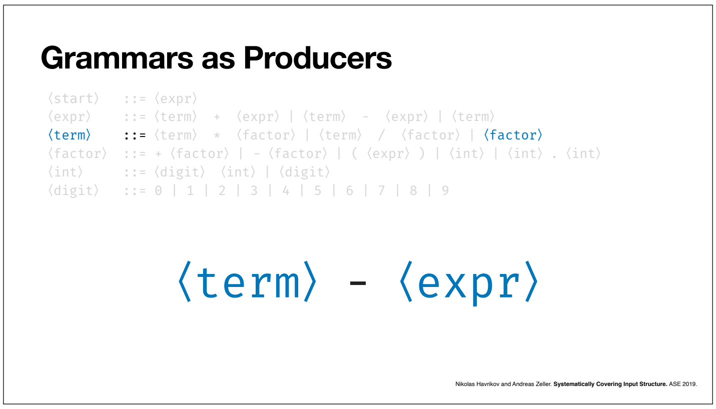
Grammars as Producers

```
⟨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.

If there are multiple alternatives, you randomly choose one.

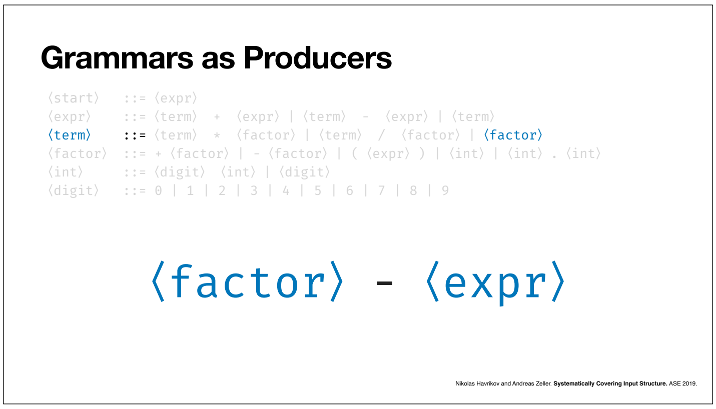


Grammars as Producers

```
⟨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.

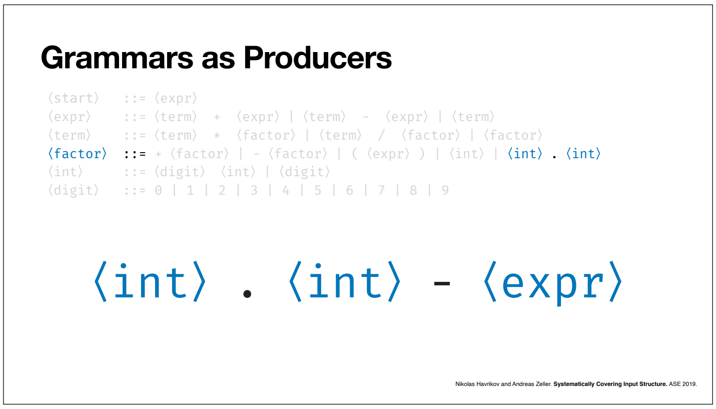


Grammars as Producers

```
⟨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.

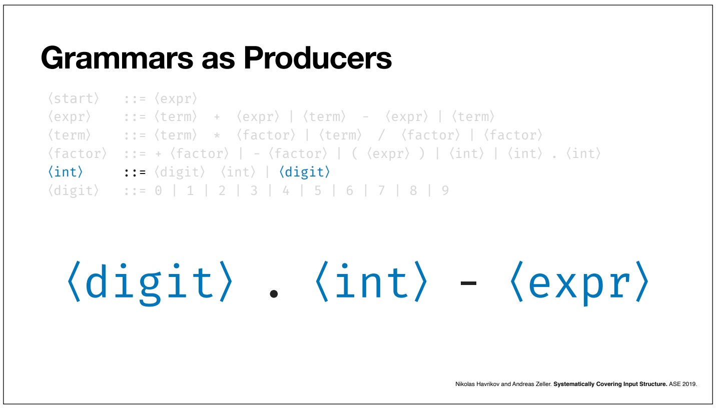


Grammars as Producers

```
⟨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.

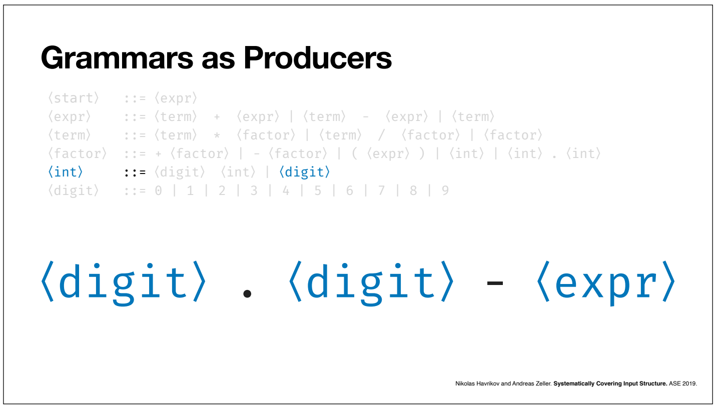


Grammars as Producers

```
⟨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.

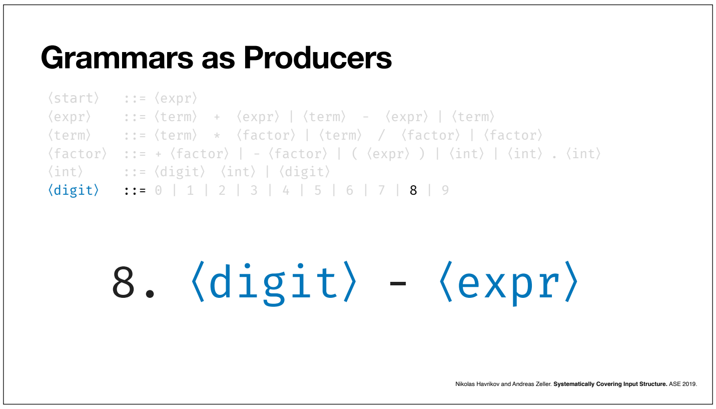


Grammars as Producers

```
⟨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.

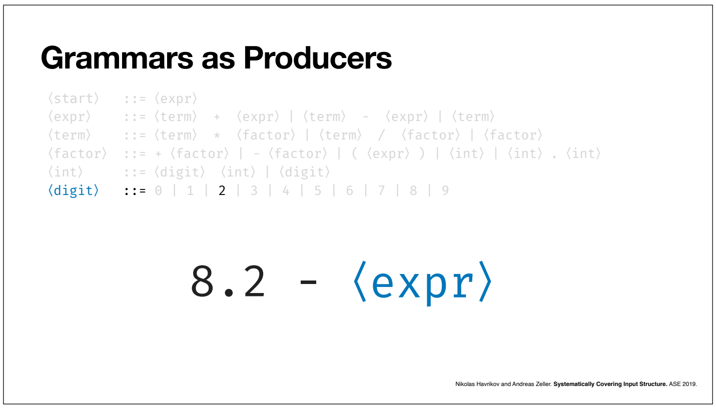


Grammars as Producers

```
⟨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.



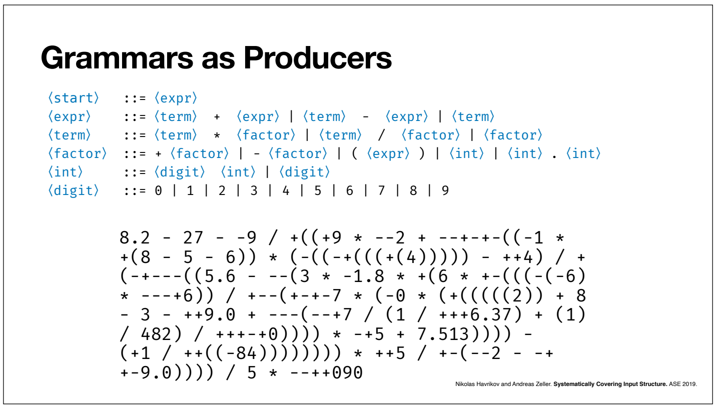
Grammars as Producers

```
⟨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.

Over time, this gives you a syntactically valid input. In our case, a valid arithmetic expression.



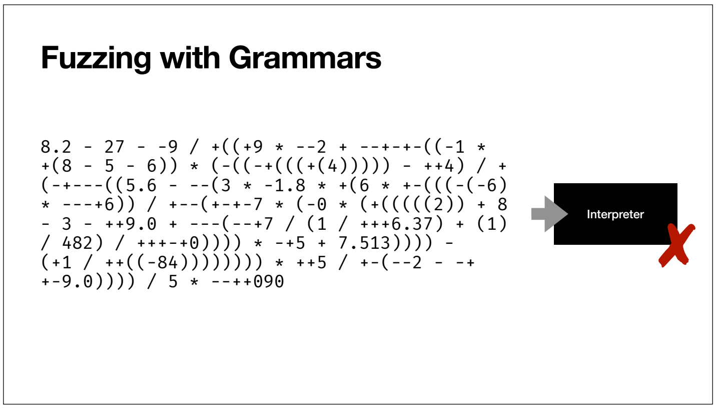
Grammars as Producers

```
⟨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 - 27 - -9 / +((+9 * --2 + --+-+-((-1 *
+(8 - 5 - 6)) * (-((-+(((+(4))))) - ++4) / +
(-+---((5.6 - --(3 * -1.8 * +(6 * +-(((-(-6)
* ---+6)) / +--(+-+-7 * (-0 * (+(((((2)) + 8
- 3 - ++9.0 + ---(--+7 / (1 / +++6.37) + (1)
/ 482) / +++-+0)))) * -+5 + 7.513)))) -
(+1 / ++((-84))))))))) * ++5 / +-(--2 - -+
+-9.0)))) / 5 * --++090
```

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

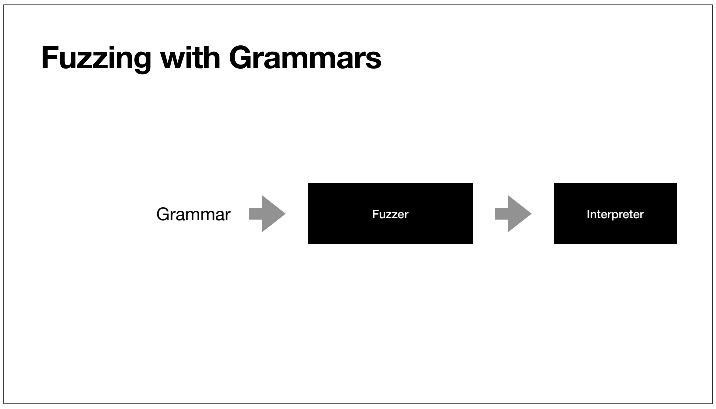
Actually, a pretty **complex** arithmetic expression.



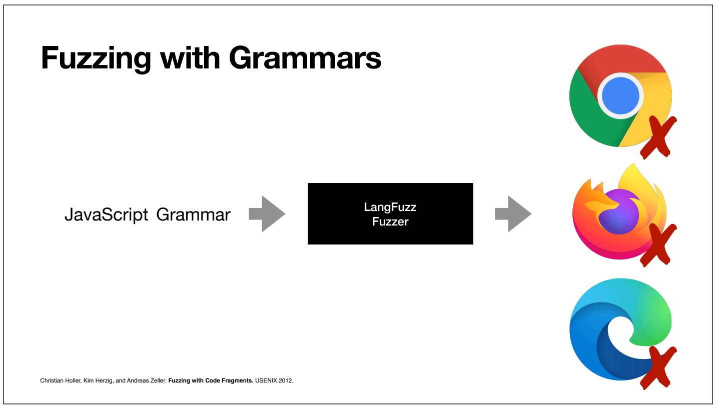
Fuzzing with Grammars

```
8.2 - 27 - -9 / +((+9 * --2 + --+-+-((-1 *
+(8 - 5 - 6)) * (-((-+(((+(4))))) - ++4) / +
(-+---((5.6 - --(3 * -1.8 * +(6 * +-(((-(-6)
* ---+6)) / +--(+-+-7 * (-0 * (+(((((2)) + 8
- 3 - ++9.0 + ---(--+7 / (1 / +++6.37) + (1)
/ 482) / +++-+0)))) * -+5 + 7.513)))) -
(+1 / ++((-84))))))))) * ++5 / +-(--2 - -+
+-9.0)))) / 5 * --++090
```

These can now be used as input to your program.



Fuzzing with Grammars

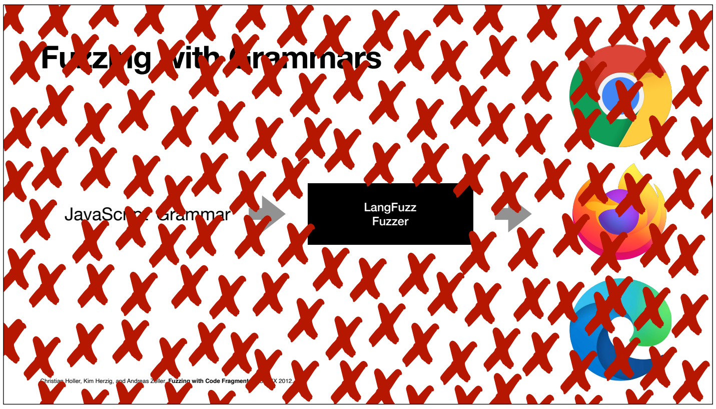


Fuzzing with Grammars

JavaScript Grammar → LangFuzz Fuzzer →

Christian Holler, Kim Herzig, and Andreas Zeller. **Fuzzing with Code Fragments.** USENIX 2012.

A couple of years ago, we used a JavaScript grammar to fuzz the interpreters of Firefox, Chrome and Edge.

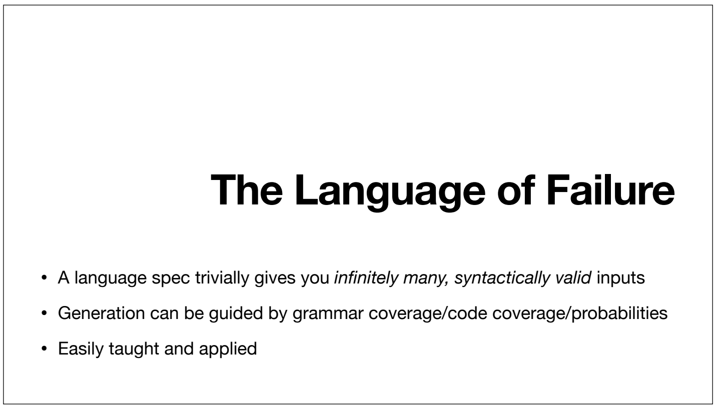


Fuzzing with Grammars

JavaScript Grammar → LangFuzz Fuzzer →

Christian Holler, Kim Herzig, and Andreas Zeller. **Fuzzing with Code Fragments.** USENIX 2012.

My student Christian Holler found more than 2,600 bugs, and in the first four weeks, he netted more than $50,000 in bug bounties. If you use a browser to read this, one of the reasons your browser works as it should is because of grammar-based fuzzing.



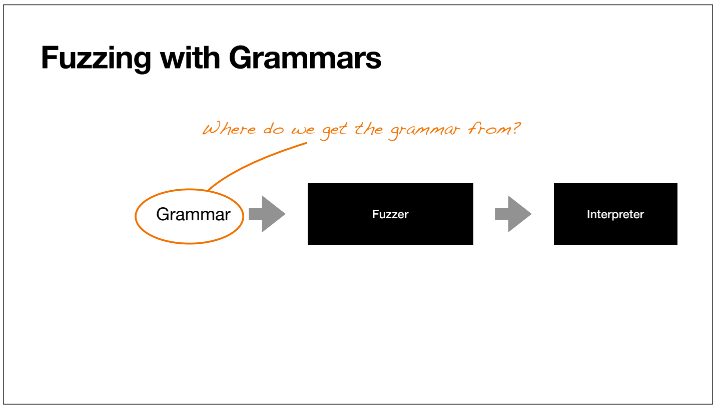
The Language of Failure

- A language spec trivially gives you *infinitely many, syntactically valid* inputs
- Generation can be guided by grammar coverage/code coverage/probabilities
- Easily taught and applied

And if you are interested in how to use grammar for fuzzing, the book will give you lots of inspiration.

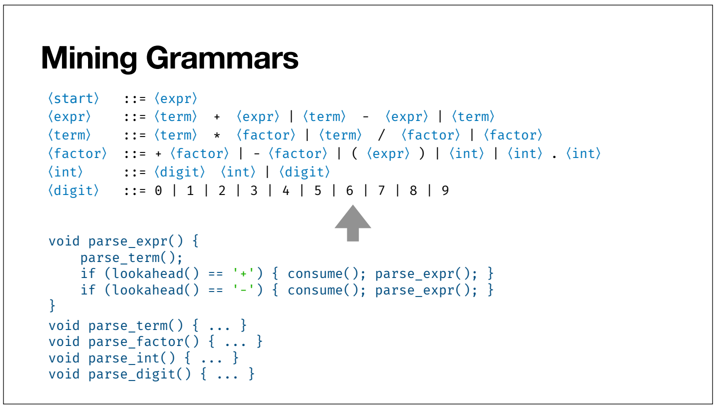
Learning the Language

But all of this still requires a grammar in the first place.



Fuzzing with Grammars

So where did you get this grammar from?

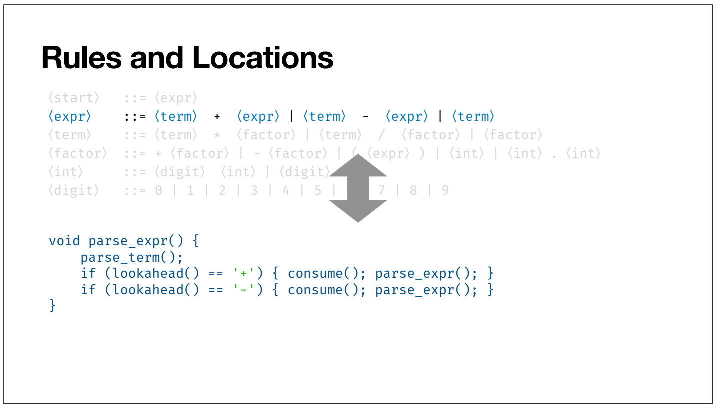


Mining Grammars

```
⟨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
```

```
void parse_expr() {
    parse_term();
    if (lookahead() == '+') { consume(); parse_expr(); }
    if (lookahead() == '-') { consume(); parse_expr(); }
}
void parse_term() { ... }
void parse_factor() { ... }
void parse_int() { ... }
void parse_digit() { ... }
```

So let me tell you a bit about how to mind such grammars. The idea is to take a program that parses such inputs and extract the input grammar from it.

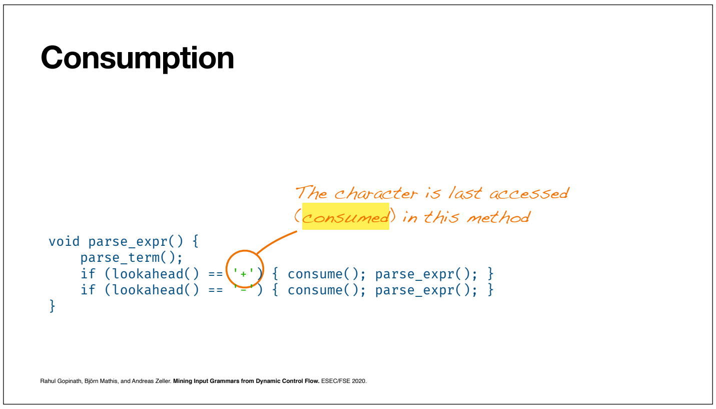


Rules and Locations

```
⟨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
```

```
void parse_expr() {
    parse_term();
    if (lookahead() == '+') { consume(); parse_expr(); }
    if (lookahead() == '-') { consume(); parse_expr(); }
}
```

The interesting thing is that there is a correspondence between individual rules in the input grammar and locations in the parsing code.



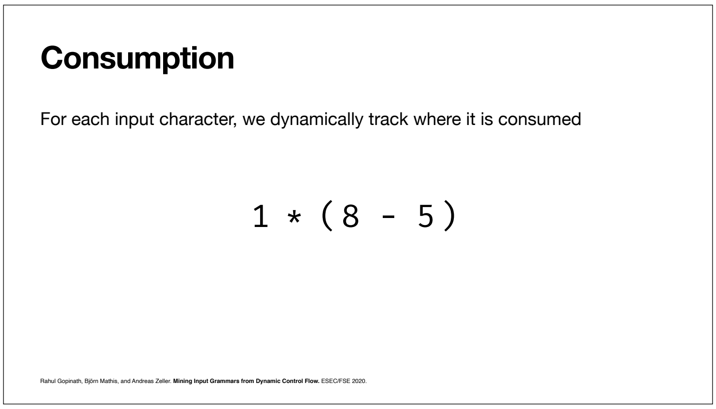
Consumption

The character is last accessed (consumed) in this method

```
void parse_expr() {
    parse_term();
    if (lookahead() == '+') { consume(); parse_expr(); }
    if (lookahead() == '-') { consume(); parse_expr(); }
}
```

Rahul Gopinath, Björn Mathis, and Andreas Zeller. **Mining Input Grammars from Dynamic Control Flow.** ESEC/FSE 2020.

The concept of consumption establishes this correspondence. A character is **consumed** in a method *m* if *m* is the last to access it.

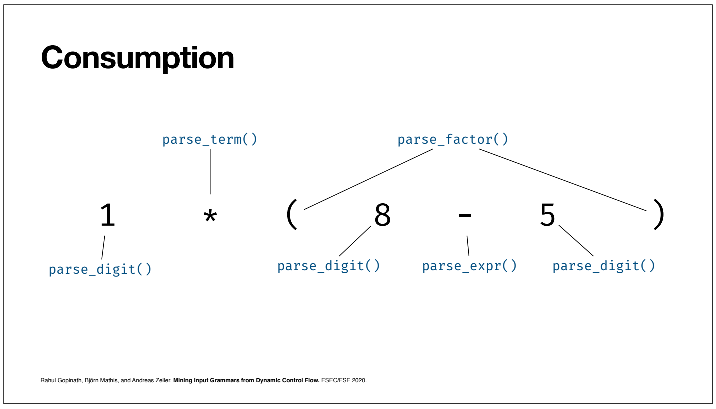


Consumption

For each input character, we dynamically track where it is consumed

```
1 * ( 8 - 5 )
```

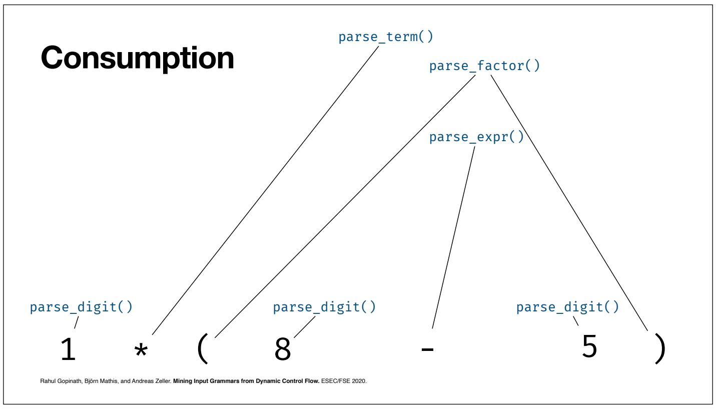
Rahul Gopinath, Björn Mathis, and Andreas Zeller. **Mining Input Grammars from Dynamic Control Flow.** ESEC/FSE 2020.



Consumption

Rahul Gopinath, Björn Mathis, and Andreas Zeller. **Mining Input Grammars from Dynamic Control Flow.** ESEC/FSE 2020.

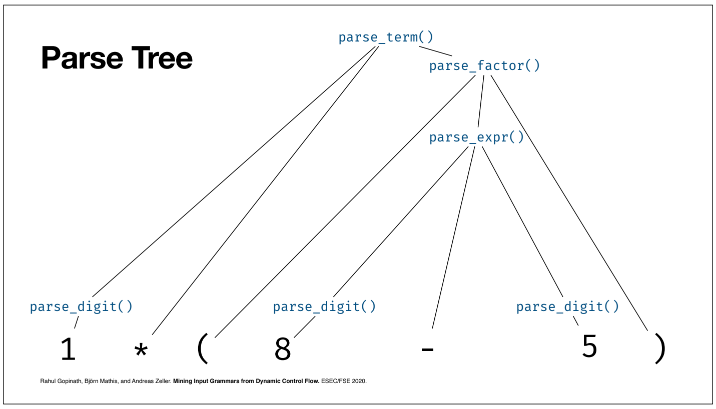
During program execution we can track where characters are consumed using dynamic tainting.



Consumption

Rahul Gopinath, Björn Mathis, and Andreas Zeller. **Mining Input Grammars from Dynamic Control Flow.** ESEC/FSE 2020.

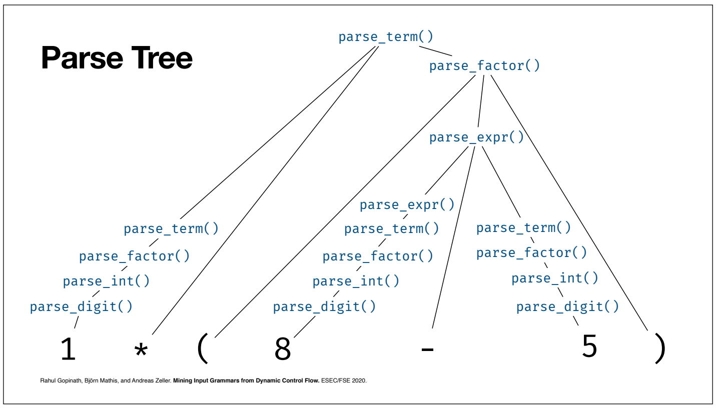
This gives us a tree like structure.



Parse Tree

Rahul Gopinath, Björn Mathis, and Andreas Zeller. **Mining Input Grammars from Dynamic Control Flow.** ESEC/FSE 2020.

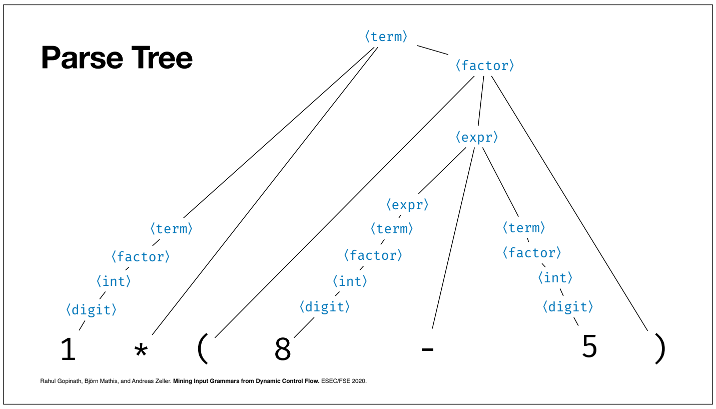
Which we can augment with caller-callee relations.



Parse Tree

Rahul Gopinath, Björn Mathis, and Andreas Zeller. **Mining Input Grammars from Dynamic Control Flow.** ESEC/FSE 2020.

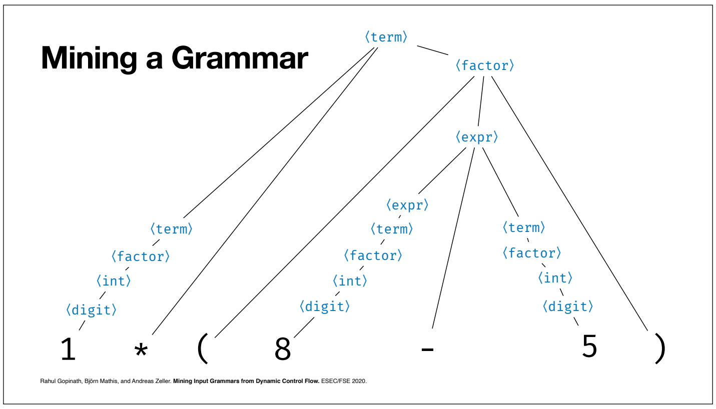
Even for those functions which do not consume anything.



Parse Tree

Rahul Gopinath, Björn Mathis, and Andreas Zeller. **Mining Input Grammars from Dynamic Control Flow.** ESEC/FSE 2020.

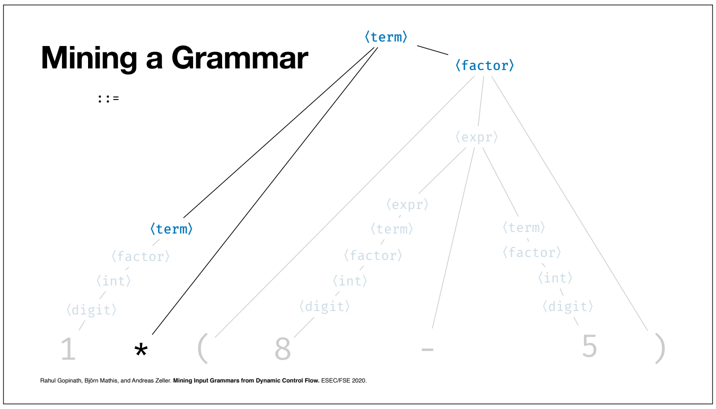
If we take the function names and only use the nouns, we can use those nouns as non-terminal symbols.



Mining a Grammar

Rahul Gopinath, Björn Mathis, and Andreas Zeller. **Mining Input Grammars from Dynamic Control Flow.** ESEC/FSE 2020.

From these parse trees, we can now mine a grammar.

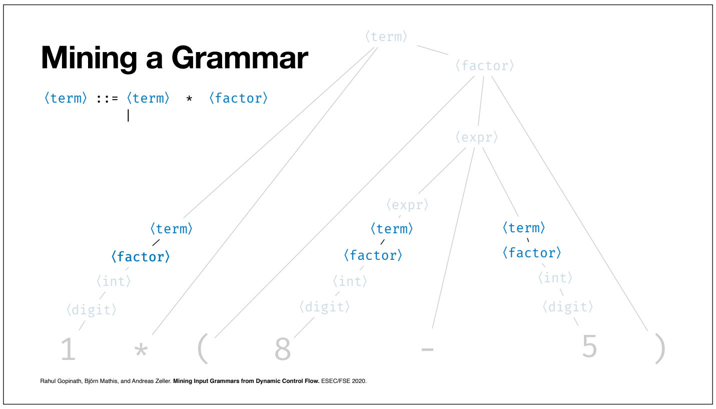


Mining a Grammar

```
      ::=
```

Rahul Gopinath, Björn Mathis, and Andreas Zeller. **Mining Input Grammars from Dynamic Control Flow.** ESEC/FSE 2020.

A term obviously can consist of another term, a multiplication symbol, and a factor.

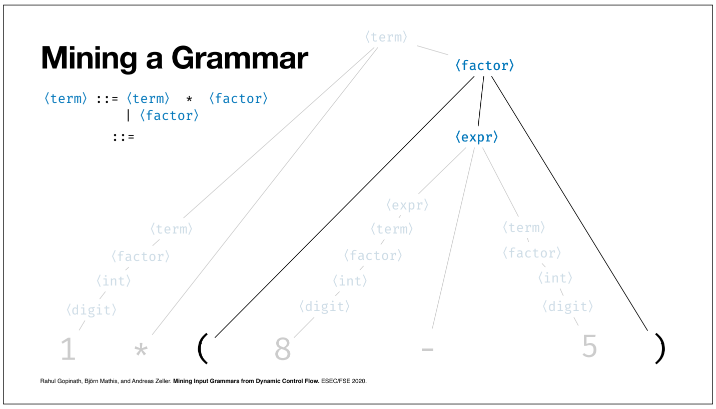


Mining a Grammar

```
⟨term⟩ ::= ⟨term⟩  *  ⟨factor⟩
         |
```

Rahul Gopinath, Björn Mathis, and Andreas Zeller. **Mining Input Grammars from Dynamic Control Flow.** ESEC/FSE 2020.

So we add this as a rule to our grammar.

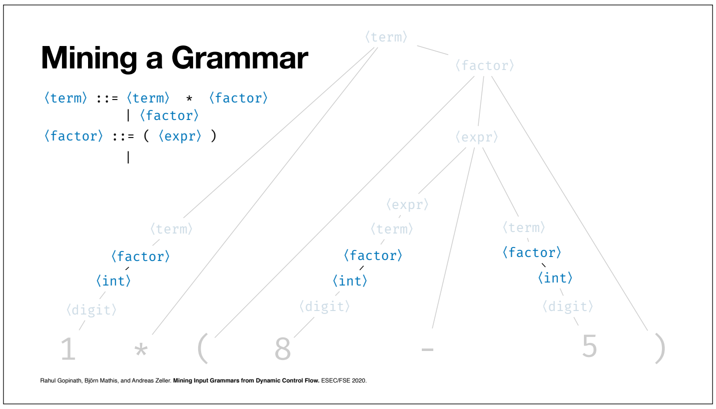


Mining a Grammar

```
⟨term⟩ ::= ⟨term⟩  *  ⟨factor⟩
         | ⟨factor⟩
       ::=
```

Rahul Gopinath, Björn Mathis, and Andreas Zeller. **Mining Input Grammars from Dynamic Control Flow.** ESEC/FSE 2020.

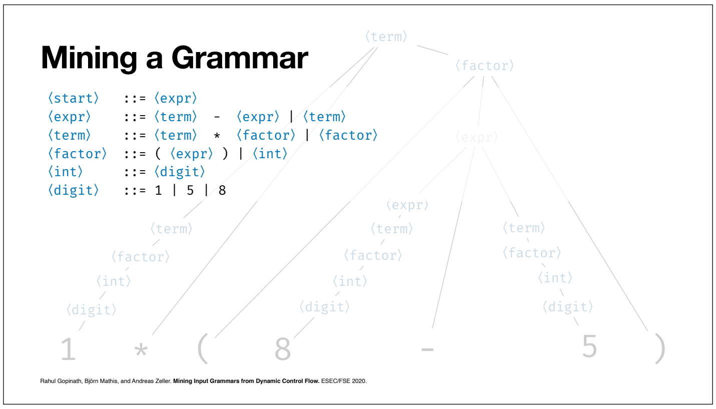
And likewise for other symbols.



Mining a Grammar

```
⟨term⟩ ::= ⟨term⟩  *  ⟨factor⟩
         | ⟨factor⟩
⟨factor⟩ ::= ( ⟨expr⟩ )
         |
```

Rahul Gopinath, Björn Mathis, and Andreas Zeller. **Mining Input Grammars from Dynamic Control Flow.** ESEC/FSE 2020.

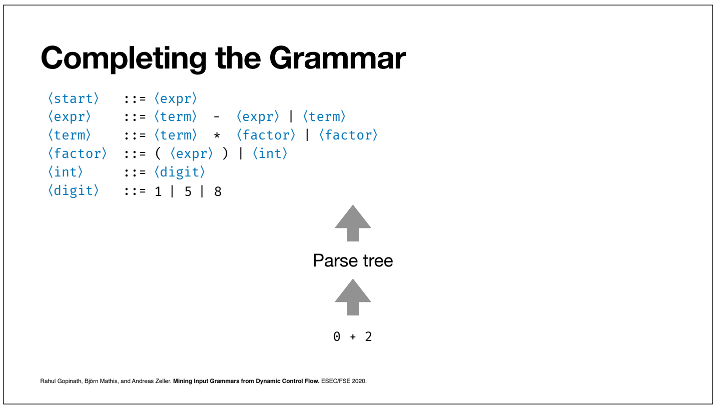


Mining a Grammar

```
⟨start⟩   ::= ⟨expr⟩
⟨expr⟩    ::= ⟨term⟩  -  ⟨expr⟩ | ⟨term⟩
⟨term⟩    ::= ⟨term⟩  *  ⟨factor⟩ | ⟨factor⟩
⟨factor⟩  ::= ( ⟨expr⟩ ) | ⟨int⟩
⟨int⟩     ::= ⟨digit⟩
⟨digit⟩   ::= 1 | 5 | 8
```

Rahul Gopinath, Björn Mathis, and Andreas Zeller. **Mining Input Grammars from Dynamic Control Flow.** ESEC/FSE 2020.

From this single input, we already get the basics of a grammar.

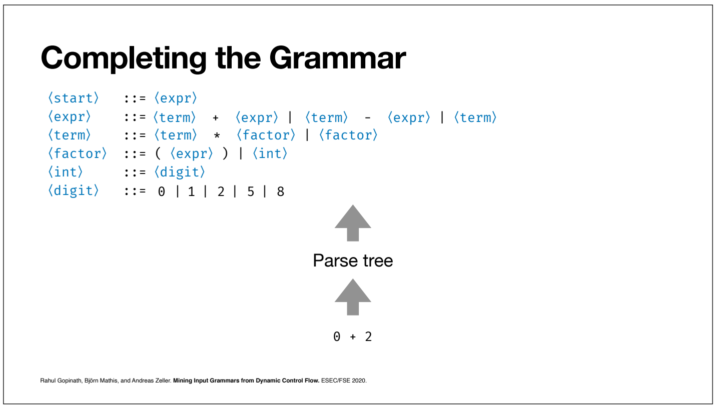


Completing the Grammar

```
⟨start⟩   ::= ⟨expr⟩
⟨expr⟩    ::= ⟨term⟩  -  ⟨expr⟩ | ⟨term⟩
⟨term⟩    ::= ⟨term⟩  *  ⟨factor⟩ | ⟨factor⟩
⟨factor⟩  ::= ( ⟨expr⟩ ) | ⟨int⟩
⟨int⟩     ::= ⟨digit⟩
⟨digit⟩   ::= 1 | 5 | 8
```

Rahul Gopinath, Björn Mathis, and Andreas Zeller. **Mining Input Grammars from Dynamic Control Flow.** ESEC/FSE 2020.

And if we add more inputs, ...



Completing the Grammar

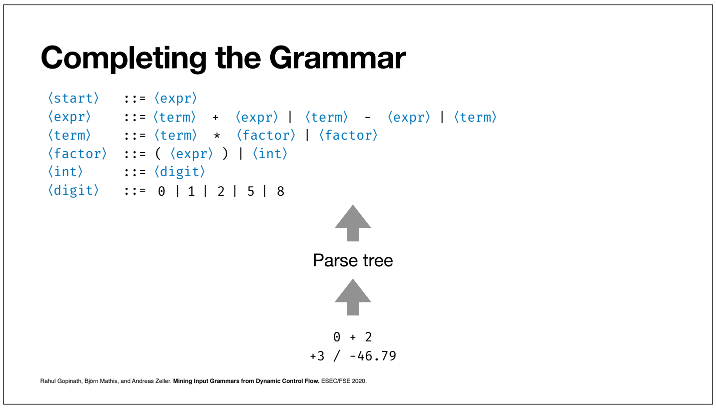
```
⟨start⟩   ::= ⟨expr⟩
⟨expr⟩    ::= ⟨term⟩  +  ⟨expr⟩ | ⟨term⟩  -  ⟨expr⟩ | ⟨term⟩
⟨term⟩    ::= ⟨term⟩  *  ⟨factor⟩ | ⟨factor⟩
⟨factor⟩  ::= ( ⟨expr⟩ ) | ⟨int⟩
⟨int⟩     ::= ⟨digit⟩
⟨digit⟩   ::=  0 | 1 | 2 | 5 | 8
```

Parse tree

```
0 + 2
```

Rahul Gopinath, Björn Mathis, and Andreas Zeller. **Mining Input Grammars from Dynamic Control Flow.** ESEC/FSE 2020.

... the grammar reflects the structure of these additional inputs.



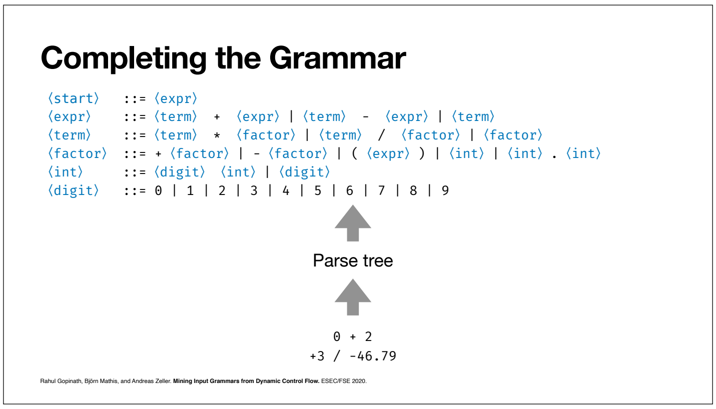
Completing the Grammar

```
⟨start⟩   ::= ⟨expr⟩
⟨expr⟩    ::= ⟨term⟩  +  ⟨expr⟩ | ⟨term⟩  -  ⟨expr⟩ | ⟨term⟩
⟨term⟩    ::= ⟨term⟩  *  ⟨factor⟩ | ⟨factor⟩
⟨factor⟩  ::= ( ⟨expr⟩ ) | ⟨int⟩
⟨int⟩     ::= ⟨digit⟩
⟨digit⟩   ::=  0 | 1 | 2 | 5 | 8
```

Parse tree

```
0 + 2
+3 / -46.79
```

Rahul Gopinath, Björn Mathis, and Andreas Zeller. **Mining Input Grammars from Dynamic Control Flow.** ESEC/FSE 2020.



Completing the Grammar

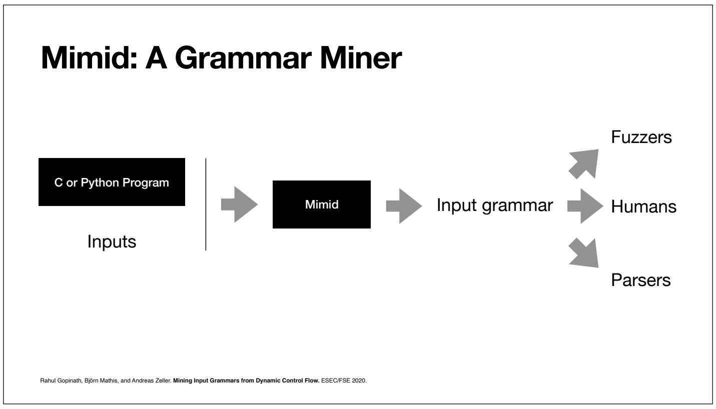
```
⟨start⟩   ::= ⟨expr⟩
⟨expr⟩    ::= ⟨term⟩  +  ⟨expr⟩ | ⟨term⟩  -  ⟨expr⟩ | ⟨term⟩
⟨term⟩    ::= ⟨term⟩  *  ⟨factor⟩ | ⟨term⟩  /  ⟨factor⟩ | ⟨factor⟩
⟨factor⟩  ::= + ⟨factor⟩ | - ⟨factor⟩ | ( ⟨expr⟩ ) | ⟨int⟩ | ⟨int⟩ . ⟨int⟩
⟨int⟩     ::= ⟨digit⟩  ⟨int⟩ | ⟨digit⟩
⟨digit⟩   ::= 0 | 1 | 2 | 3 | 4 |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

Parse tree

```
0 + 2
+3 / -46.79
```

Rahul Gopinath, Björn Mathis, and Andreas Zeller. **Mining Input Grammars from Dynamic Control Flow.** ESEC/FSE 2020.

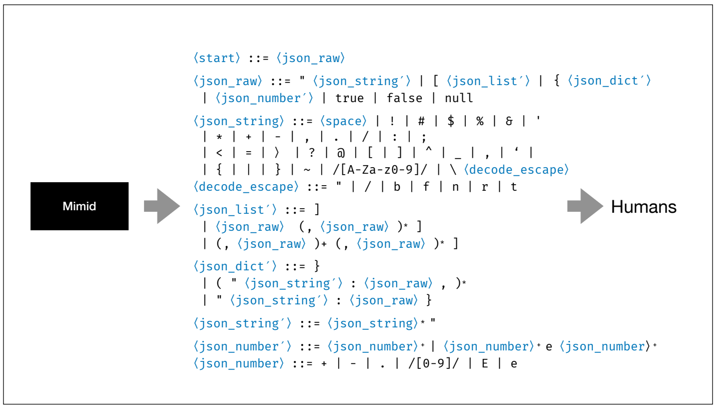
We now have successfully mined our example grammar.



Mimid: A Grammar Miner

Rahul Gopinath, Björn Mathis, and Andreas Zeller. **Mining Input Grammars from Dynamic Control Flow.** ESEC/FSE 2020.

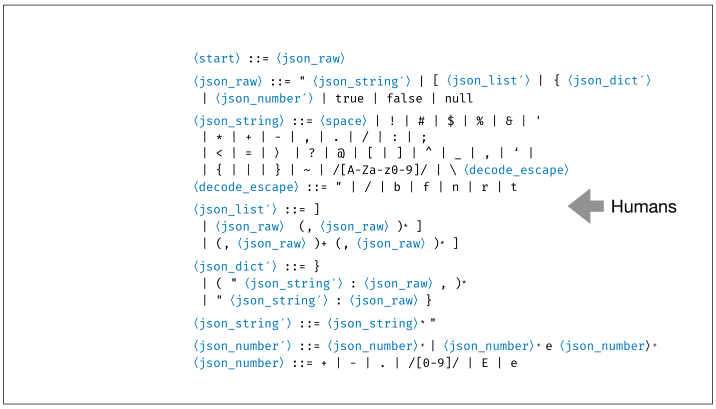
Our Mimid grammar miner takes a program and its inputs and extracts a grammar out of it. This grammar can directly be used by fuzzers, parsers, and humans.



Mimid
Humans

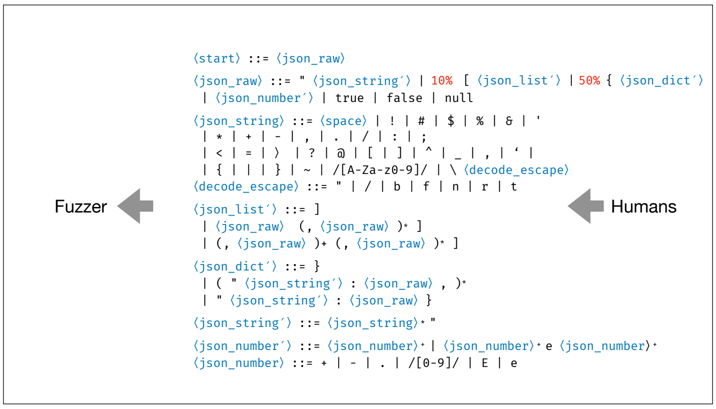
```
⟨start⟩ ::= ⟨json_raw⟩
⟨json_raw⟩ ::= " ⟨json_string′⟩ | [ ⟨json_list′⟩ | { ⟨json_dict′⟩
 | ⟨json_number′⟩ | true | false | null
⟨json_string⟩ ::= ⟨space⟩ | ! | # | $ | % | & | '
 | * | + | - | , | . | / | : | ;
 | < | = | ⟩  | ? | @ | [ | ] | ^ | _ | , | ‘ |
 | { | | | } | ~ | /[A-Za-z0-9]/ | \ ⟨decode_escape⟩
⟨decode_escape⟩ ::= " | / | b | f | n | r | t
⟨json_list′⟩ ::= ]
 | ⟨json_raw⟩  (, ⟨json_raw⟩ )* ]
 | (, ⟨json_raw⟩ )+ (, ⟨json_raw⟩ )* ]
⟨json_dict′⟩ ::= }
 | ( " ⟨json_string′⟩ : ⟨json_raw⟩ , )*
 | " ⟨json_string′⟩ : ⟨json_raw⟩ }
⟨json_string′⟩ ::= ⟨json_string⟩* "
⟨json_number′⟩ ::= ⟨json_number⟩+ | ⟨json_number⟩+ e ⟨json_number⟩+
⟨json_number⟩ ::= + | - | . | /[0-9]/ | E | e
```

The extracted grammars are well structured and human readable as you can see in this grammar extracted from a JSON parser.



```
⟨start⟩ ::= ⟨json_raw⟩
⟨json_raw⟩ ::= " ⟨json_string′⟩ | [ ⟨json_list′⟩ | { ⟨json_dict′⟩
 | ⟨json_number′⟩ | true | false | null
⟨json_string⟩ ::= ⟨space⟩ | ! | # | $ | % | & | '
 | * | + | - | , | . | / | : | ;
 | < | = | ⟩  | ? | @ | [ | ] | ^ | _ | , | ‘ |
 | { | | | } | ~ | /[A-Za-z0-9]/ | \ ⟨decode_escape⟩
⟨decode_escape⟩ ::= " | / | b | f | n | r | t
⟨json_list′⟩ ::= ]
 | ⟨json_raw⟩  (, ⟨json_raw⟩ )* ]
 | (, ⟨json_raw⟩ )+ (, ⟨json_raw⟩ )* ]
⟨json_dict′⟩ ::= }
 | ( " ⟨json_string′⟩ : ⟨json_raw⟩ , )*
 | " ⟨json_string′⟩ : ⟨json_raw⟩ }
⟨json_string′⟩ ::= ⟨json_string⟩* "
⟨json_number′⟩ ::= ⟨json_number⟩+ | ⟨json_number⟩+ e ⟨json_number⟩+
⟨json_number⟩ ::= + | - | . | /[0-9]/ | E | e
```

Humans can **edit** these grammars.

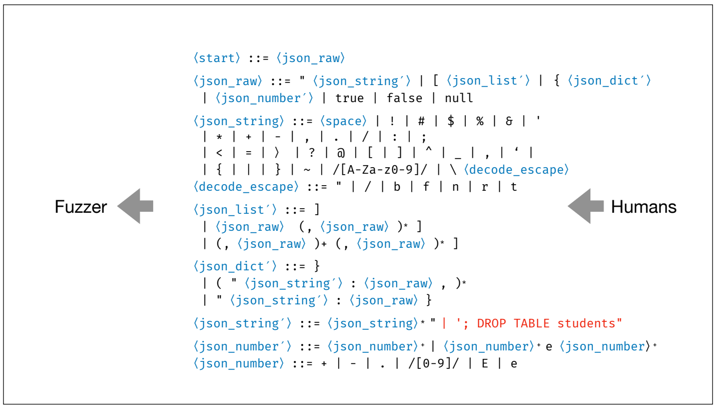


Fuzzer

```
⟨start⟩ ::= ⟨json_raw⟩
⟨json_raw⟩ ::= " ⟨json_string′⟩ | 10% [ ⟨json_list′⟩ | 50% { ⟨json_dict′⟩
 | ⟨json_number′⟩ | true | false | null
⟨json_string⟩ ::= ⟨space⟩ | ! | # | $ | % | & | '
 | * | + | - | , | . | / | : | ;
 | < | = | 〉 | ? | @ | [ | ] | ^ | _ | , | ‘ |
 | { | | | } | ~ | /[A-Za-z0-9]/ | \ ⟨decode_escape⟩
⟨decode_escape⟩ ::= " | / | b | f | n | r | t
⟨json_list′⟩ ::= ]
 | ⟨json_raw⟩  (, ⟨json_raw⟩ )* ]
 | (, ⟨json_raw⟩ )+ (, ⟨json_raw⟩ )* ]
⟨json_dict′⟩ ::= }
 | ( " ⟨json_string′⟩ : ⟨json_raw⟩ , )*
 | " ⟨json_string′⟩ : ⟨json_raw⟩ }
⟨json_string′⟩ ::= ⟨json_string⟩* "
⟨json_number′⟩ ::= ⟨json_number⟩+ | ⟨json_number⟩+ e ⟨json_number⟩+
⟨json_number⟩ ::= + | - | . | /[0-9]/ | E | e
```

Humans

For instance, by assigning probabilities to individual productions.

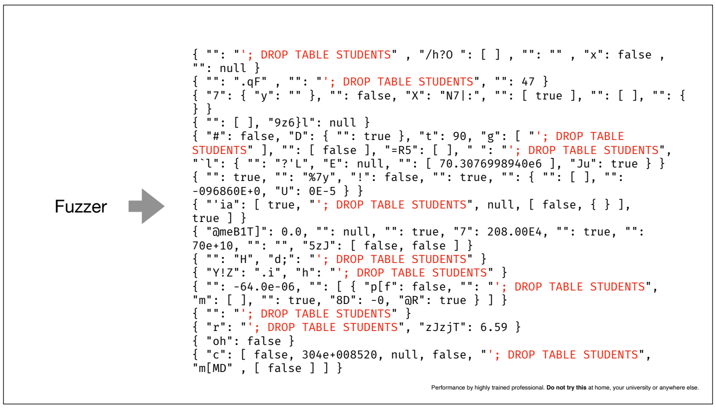


Fuzzer

```
⟨start⟩ ::= ⟨json_raw⟩
⟨json_raw⟩ ::= " ⟨json_string′⟩ | [ ⟨json_list′⟩ | { ⟨json_dict′⟩
 | ⟨json_number′⟩ | true | false | null
⟨json_string⟩ ::= ⟨space⟩ | ! | # | $ | % | & | '
 | * | + | - | , | . | / | : | ;
 | < | = | 〉 | ? | @ | [ | ] | ^ | _ | , | ‘ |
 | { | | | } | ~ | /[A-Za-z0-9]/ | \ ⟨decode_escape⟩
⟨decode_escape⟩ ::= " | / | b | f | n | r | t
⟨json_list′⟩ ::= ]
 | ⟨json_raw⟩  (, ⟨json_raw⟩ )* ]
 | (, ⟨json_raw⟩ )+ (, ⟨json_raw⟩ )* ]
⟨json_dict′⟩ ::= }
 | ( " ⟨json_string′⟩ : ⟨json_raw⟩ , )*
 | " ⟨json_string′⟩ : ⟨json_raw⟩ }
⟨json_string′⟩ ::= ⟨json_string⟩* " | '; DROP TABLE students"
⟨json_number′⟩ ::= ⟨json_number⟩+ | ⟨json_number⟩+ e ⟨json_number⟩+
⟨json_number⟩ ::= + | - | . | /[0-9]/ | E | e
```

Humans

Or by inserting magic strings that program analysis would have a hard time finding out.

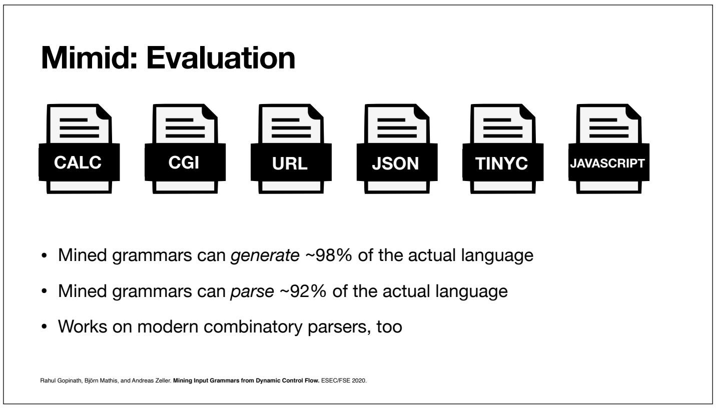


Fuzzer

```
{ "": "'; DROP TABLE STUDENTS" , "/h?O ": [ ] , "": "" , "x": false ,
"": null }
{ "": ".qF" , "": "'; DROP TABLE STUDENTS", "": 47 }
{ "7": { "y": "" }, "": false, "X": "N7|:", "": [ true ], "": [ ], "": {
} }
{ "": [ ], "9z6}l": null }
{ "#": false, "D": { "": true }, "t": 90, "g": [ "'; DROP TABLE
STUDENTS" ], "": [ false ], "=R5": [ ], " ": "'; DROP TABLE STUDENTS",
"`l": { "": "?'L", "E": null, "": [ 70.3076998940e6 ], "Ju": true } }
{ "": true, "": "%7y", "!": false, "": true, "": { "": [ ], "":
-096860E+0, "U": 0E-5 } }
{ "'ia": [ true, "'; DROP TABLE STUDENTS", null, [ false, { } ],
true ] }
{ "@meB1T]": 0.0, "": null, "": true, "7": 208.00E4, "": true, "":
70e+10, "": "", "5zJ": [ false, false ] }
{ "": "H", "d;": "'; DROP TABLE STUDENTS" }
{ "Y!Z": ".i", "h": "'; DROP TABLE STUDENTS" }
{ "": -64.0e-06, "": [ { "p[f": false, "": "'; DROP TABLE STUDENTS",
"m": [ ], "": true, "8D": -0, "@R": true } ] }
{ "": "'; DROP TABLE STUDENTS" }
{ "r": "'; DROP TABLE STUDENTS", "zJzjT": 6.59 }
{ "oh": false }
{ "c": [ false, 304e+008520, null, false, "'; DROP TABLE STUDENTS",
"m[MD" , [ false ] ] }
```

Performance by highly trained professional. **Do not try this** at home, your university or anywhere else.

This change to the grammar injects SQL statements everywhere. Do not do this at home, folks – thank you.



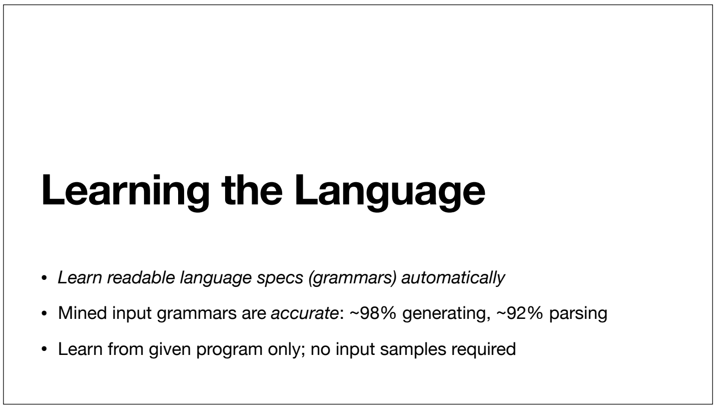
Mimid: Evaluation

CALC
CGI
URL
JSON
TINYC
JAVASCRIPT

- Mined grammars can *generate* ~98% of the actual language
- Mined grammars can *parse* ~92% of the actual language
- Works on modern combinatory parsers, too

Rahul Gopinath, Björn Mathis, and Andreas Zeller. **Mining Input Grammars from Dynamic Control Flow.** ESEC/FSE 2020.

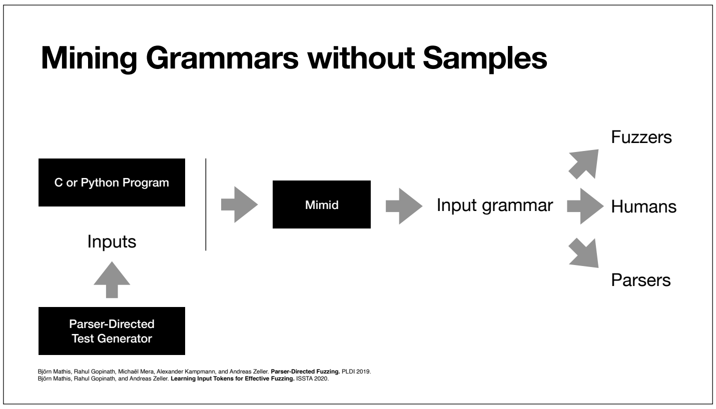
The grammars extracted by Mimid are accurate as producers as well as as parsers.



Learning the Language

- *Learn readable language specs (grammars) automatically*
- Mined input grammars are *accurate*: ~98% generating, ~92% parsing
- Learn from given program only; no input samples required

So this was about **learning** (input) languages.



Mining Grammars without Samples

Björn Mathis, Rahul Gopinath, Michaël Mera, Alexander Kampmann, and Andreas Zeller. **Parser-Directed Fuzzing.** PLDI 2019.
Björn Mathis, Rahul Gopinath, and Andreas Zeller. **Learning Input Tokens for Effective Fuzzing.** ISSTA 2020.

Our grammar miner needs inputs in the first place. But we also have specific **test generators** that systematically cover all alternatives in a parser. So technically, all you need is the program to test.

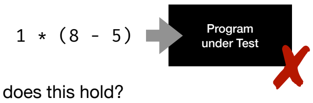
Learning the Language

- *Learn readable language specs (grammars) automatically*
- Mined input grammars are *accurate*: ~98% generating, ~92% parsing
- Learn from given program only; no input samples required

Learning the Language of Failure

And now for the main point.

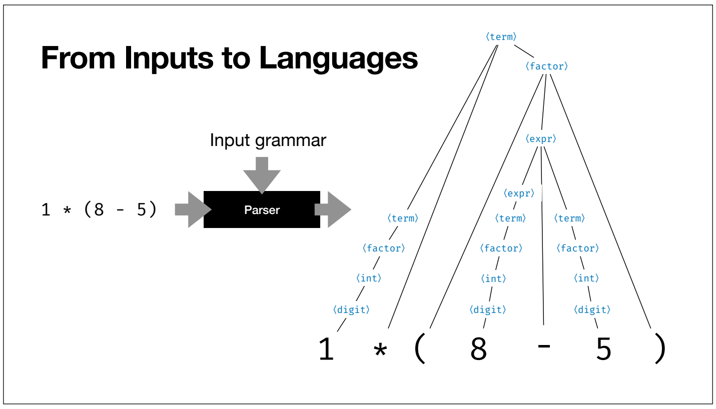
Circumstances of Failure



`1 * (8 - 5)`

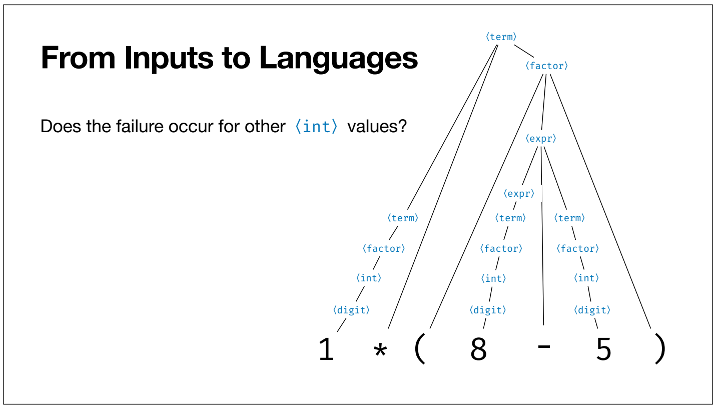
For *which other inputs* does this hold?

We have seen how single inputs cause failures. But are these the only inputs?



From Inputs to Languages

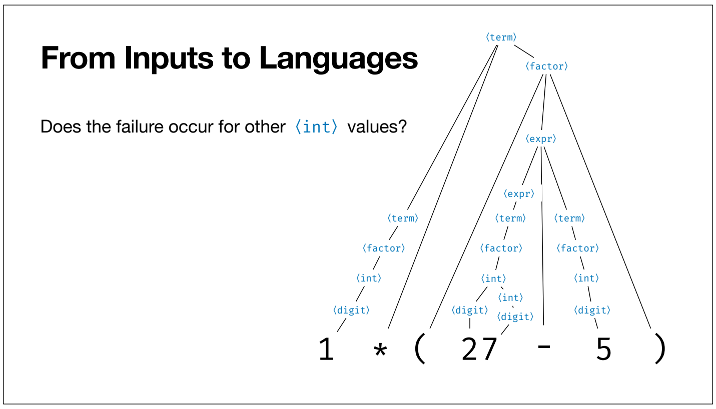
We want to know the **set of inputs** that causes the failure – in other words, the language. To this end, we parse the input into a tree.



From Inputs to Languages

Does the failure occur for other ⟨int⟩ values?

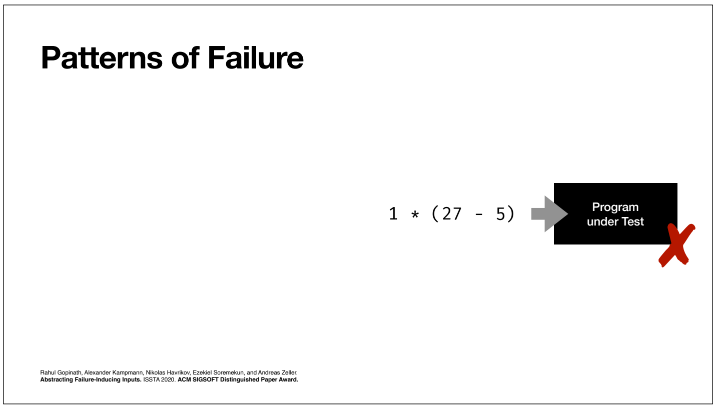
To find out whether the failure occurs for other integer values too, …



From Inputs to Languages

Does the failure occur for other ⟨int⟩ values?

… we replace parts of the parse tree (8) by newly generated alternatives (27).



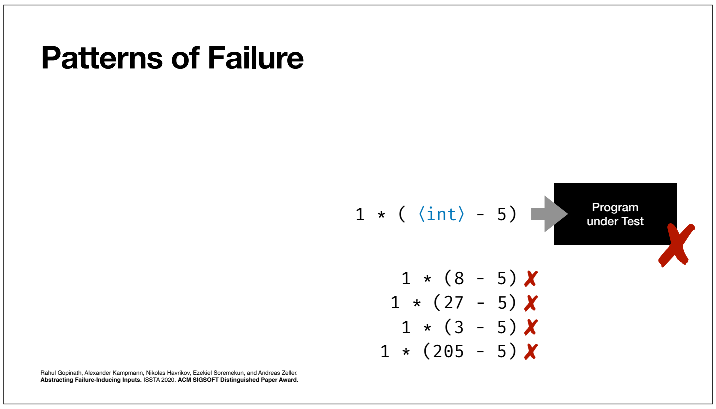
Patterns of Failure

```
1 * (27 - 5)
```

Program under Test ✗

Rahul Gopinath, Alexander Kampmann, Nikolas Havrikov, Ezekiel Soremekun, and Andreas Zeller. **Abstracting Failure-Inducing Inputs.** ISSTA 2020. **ACM SIGSOFT Distinguished Paper Award.**

and find that this one fails as well.



Patterns of Failure

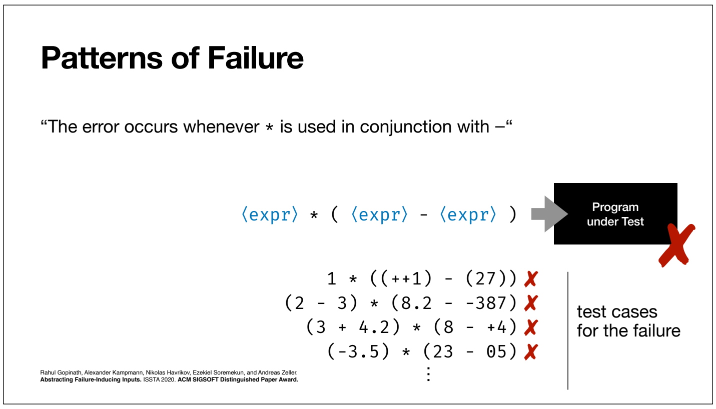
```
1 * ( ⟨int⟩ - 5)
```

Program under Test ✗

```
  1 * (8 - 5) ✗
 1 * (27 - 5) ✗
  1 * (3 - 5) ✗
1 * (205 - 5) ✗
```

Rahul Gopinath, Alexander Kampmann, Nikolas Havrikov, Ezekiel Soremekun, and Andreas Zeller. **Abstracting Failure-Inducing Inputs.** ISSTA 2020. **ACM SIGSOFT Distinguished Paper Award.**

Actually, the program fails for any integer in this position. So we can come up with an abstract pattern that represents the set of failing inputs.



Patterns of Failure

“The error occurs whenever * is used in conjunction with –“

```
⟨expr⟩ * ( ⟨expr⟩ - ⟨expr⟩ )
```

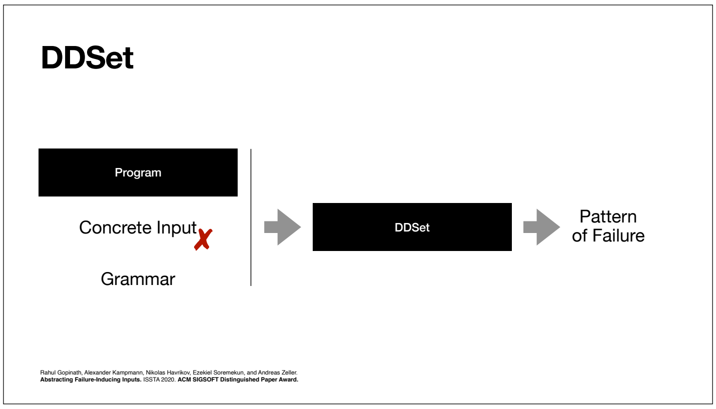
Program under Test ✗

```
    1 * ((++1) - (27)) ✗
(2 - 3) * (8.2 - -387) ✗
  (3 + 4.2) * (8 - +4) ✗
    (-3.5) * (23 - 05) ✗
            ⋮
```

test cases for the failure

Rahul Gopinath, Alexander Kampmann, Nikolas Havrikov, Ezekiel Soremekun, and Andreas Zeller. **Abstracting Failure-Inducing Inputs.** ISSTA 2020. **ACM SIGSOFT Distinguished Paper Award.**

By repeating this, we can come up with a general pattern of which **all** instantiations cause the failure. These instantiations also serve as test cases for validating a fix.



DDSet

Program

Concrete Input ✗

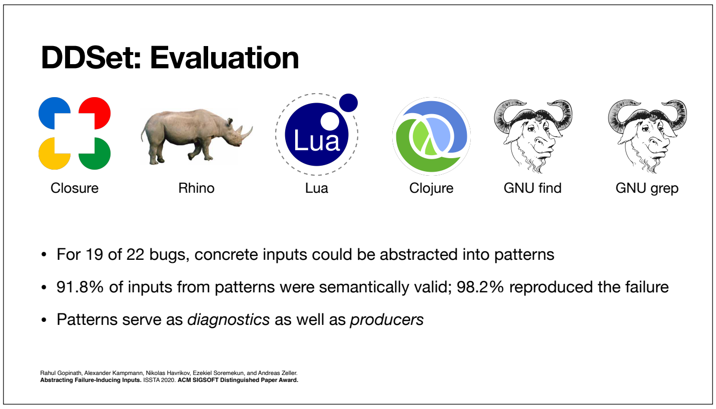
Grammar

DDSet

Pattern
of Failure

Rahul Gopinath, Alexander Kampmann, Nikolas Havrikov, Ezekiel Soremekun, and Andreas Zeller.
Abstracting Failure-Inducing Inputs. ISSTA 2020. **ACM SIGSOFT Distinguished Paper Award.**

Our tool DDSet takes a program, a failing input, and a grammar, and produces such a pattern of failure.



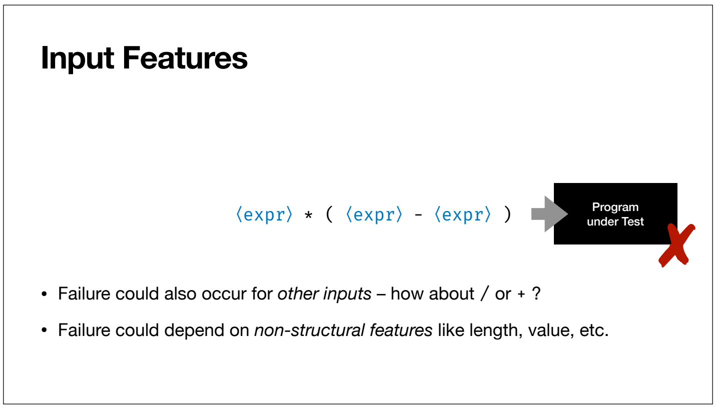
DDSet: Evaluation

Closure
Rhino
Lua
Clojure
GNU find
GNU grep

- For 19 of 22 bugs, concrete inputs could be abstracted into patterns
- 91.8% of inputs from patterns were semantically valid; 98.2% reproduced the failure
- Patterns serve as *diagnostics* as well as *producers*

Rahul Gopinath, Alexander Kampmann, Nikolas Havrikov, Ezekiel Soremekun, and Andreas Zeller.
Abstracting Failure-Inducing Inputs. ISSTA 2020. **ACM SIGSOFT Distinguished Paper Award.**

In our evaluation, this works really well.



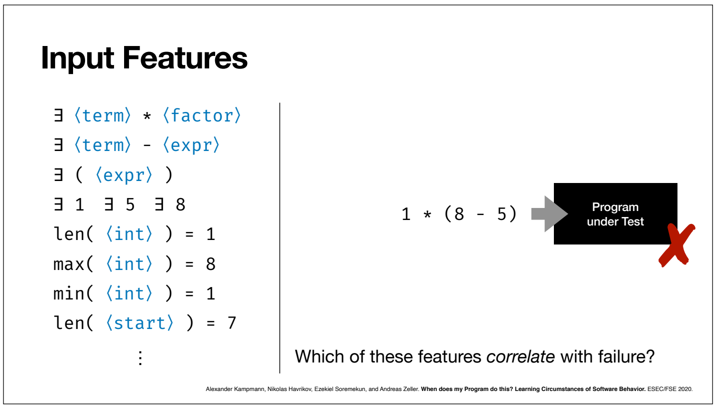
Input Features

`⟨expr⟩ * ( ⟨expr⟩ - ⟨expr⟩ )`

Program
under Test ✗

- Failure could also occur for *other inputs* – how about / or + ?
- Failure could depend on *non-structural features* like length, value, etc.

But we can go even further. What other features in the input cause a failure?



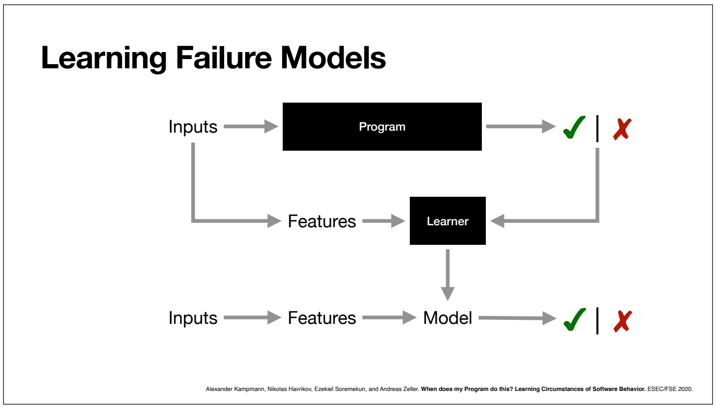
Input Features

```
∃ ⟨term⟩ * ⟨factor⟩
∃ ⟨term⟩ - ⟨expr⟩
∃ ( ⟨expr⟩ )
∃ 1  ∃ 5  ∃ 8
len( ⟨int⟩ ) = 1
max( ⟨int⟩ ) = 8
min( ⟨int⟩ ) = 1
len( ⟨start⟩ ) = 7
        ⋮
```

Which of these features *correlate* with failure?

Alexander Kampmann, Nikolas Havrikov, Ezekiel Soremekun, and Andreas Zeller. **When does my Program do this? Learning Circumstances of Software Behavior.** ESEC/FSE 2020.

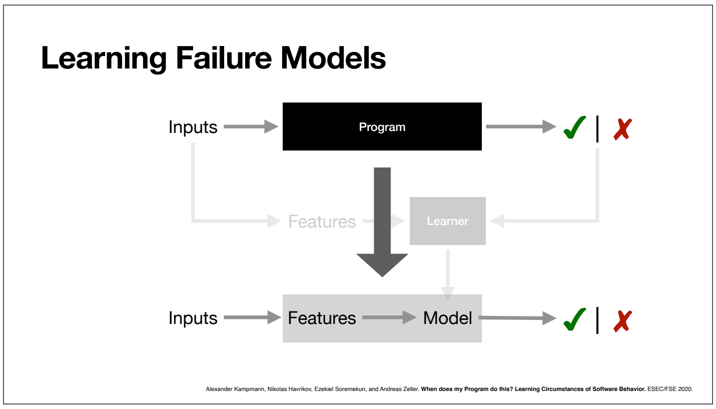
We introduce a number of input features, including existence, length, and maximum and minimum values of specific input elements.



Learning Failure Models

Alexander Kampmann, Nikolas Havrikov, Ezekiel Soremekun, and Andreas Zeller. **When does my Program do this? Learning Circumstances of Software Behavior.** ESEC/FSE 2020.

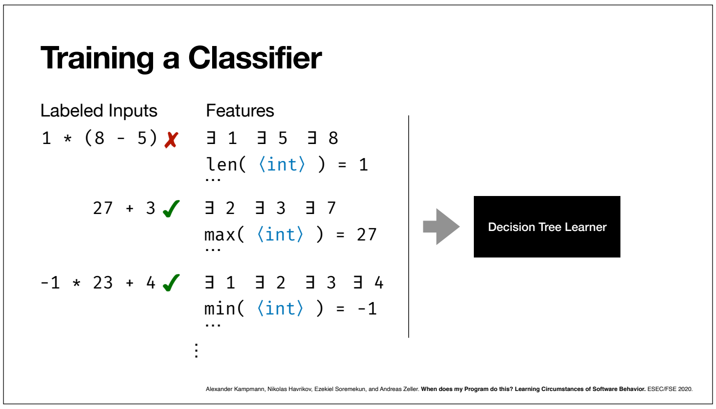
These features together with a pass and fail label then go into a machine learner which produces a predictive model.



Learning Failure Models

Alexander Kampmann, Nikolas Havrikov, Ezekiel Soremekun, and Andreas Zeller. **When does my Program do this? Learning Circumstances of Software Behavior.** ESEC/FSE 2020.

Actually, the produced model serves as a **model of the program** as it comes to failures or non-failures.

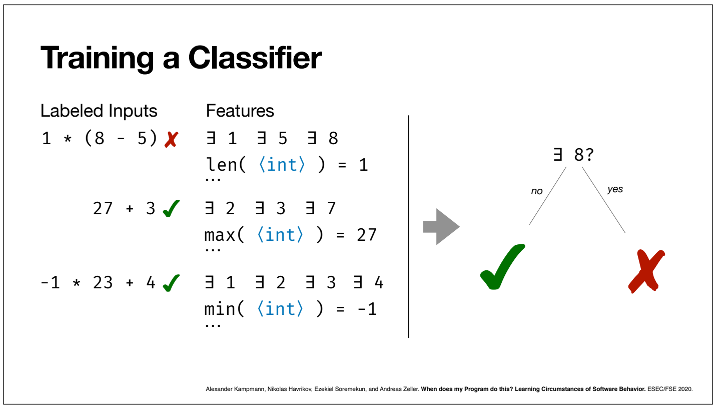


Training a Classifier

Labeled Inputs	Features
1 * (8 - 5) ✗	∃ 1 ∃ 5 ∃ 8 len(⟨int⟩) = 1 …
27 + 3 ✓	∃ 2 ∃ 3 ∃ 7 max(⟨int⟩) = 27 …
-1 * 23 + 4 ✓	∃ 1 ∃ 2 ∃ 3 ∃ 4 min(⟨int⟩) = -1 …
⋮	

Alexander Kampmann, Nikolas Havrikov, Ezekiel Soremekun, and Andreas Zeller. **When does my Program do this? Learning Circumstances of Software Behavior.** ESEC/FSE 2020.

In our experiments, we use decision tree learners as their results are easy to understand.

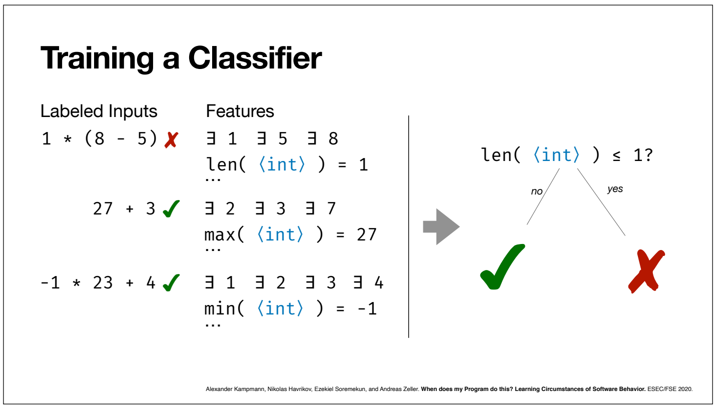


Training a Classifier

Labeled Inputs	Features
1 * (8 - 5) ✗	∃ 1 ∃ 5 ∃ 8 len(⟨int⟩) = 1 …
27 + 3 ✓	∃ 2 ∃ 3 ∃ 7 max(⟨int⟩) = 27 …
-1 * 23 + 4 ✓	∃ 1 ∃ 2 ∃ 3 ∃ 4 min(⟨int⟩) = -1 …

Alexander Kampmann, Nikolas Havrikov, Ezekiel Soremekun, and Andreas Zeller. **When does my Program do this? Learning Circumstances of Software Behavior.** ESEC/FSE 2020.

Here is a decision tree that classifies the three inputs on the left. We see that the existence of the digit 8 serves as classifying feature. The model is consistent with all the observations made so far.

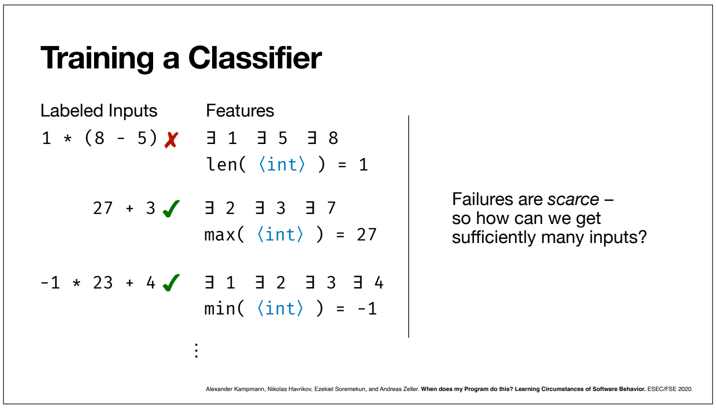


Training a Classifier

Labeled Inputs	Features
1 * (8 - 5) ✗	∃ 1 ∃ 5 ∃ 8 len(⟨int⟩) = 1 …
27 + 3 ✓	∃ 2 ∃ 3 ∃ 7 max(⟨int⟩) = 27 …
-1 * 23 + 4 ✓	∃ 1 ∃ 2 ∃ 3 ∃ 4 min(⟨int⟩) = -1 …

Alexander Kampmann, Nikolas Havrikov, Ezekiel Soremekun, and Andreas Zeller. **When does my Program do this? Learning Circumstances of Software Behavior.** ESEC/FSE 2020.

The learner also could come up with another model over the presence or non-presence of multi digit integers. Is any of these correct?



Training a Classifier

Labeled Inputs	Features

`1 * (8 - 5)` ✗	`∃ 1  ∃ 5  ∃ 8`
	`len( ⟨int⟩ ) = 1`

`27 + 3` ✓	`∃ 2  ∃ 3  ∃ 7`
	`max( ⟨int⟩ ) = 27`

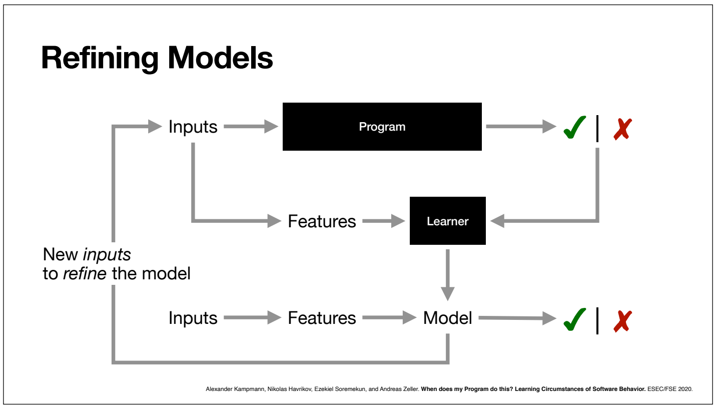
`-1 * 23 + 4` ✓	`∃ 1  ∃ 2  ∃ 3  ∃ 4`
	`min( ⟨int⟩ ) = -1`

⋮	

Failures are *scarce* –
so how can we get
sufficiently many inputs?

Alexander Kampmann, Nikolas Havrikov, Ezekiel Soremekun, and Andreas Zeller. **When does my Program do this? Learning Circumstances of Software Behavior.** ESEC/FSE 2020.

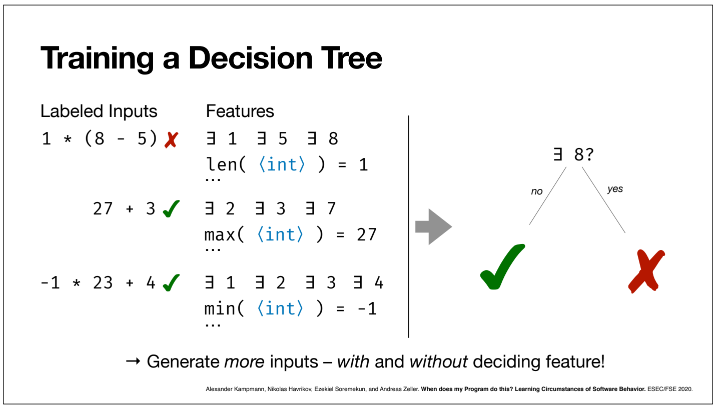
What we need is more inputs and more observations to come up with a more precise model.



Refining Models

Alexander Kampmann, Nikolas Havrikov, Ezekiel Soremekun, and Andreas Zeller. **When does my Program do this? Learning Circumstances of Software Behavior.** ESEC/FSE 2020.

We create **new inputs** right from the model learned so far.



Training a Decision Tree

Labeled Inputs	Features

`1 * (8 - 5)` ✗	`∃ 1  ∃ 5  ∃ 8`
	`len( ⟨int⟩ ) = 1`
	…

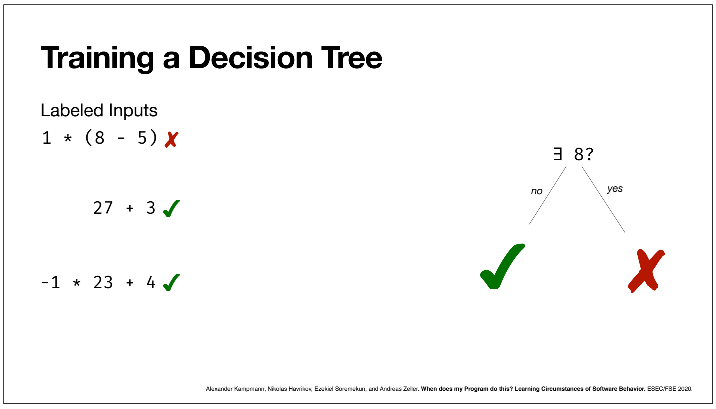
`27 + 3` ✓	`∃ 2  ∃ 3  ∃ 7`
	`max( ⟨int⟩ ) = 27`
	…

`-1 * 23 + 4` ✓	`∃ 1  ∃ 2  ∃ 3  ∃ 4`
	`min( ⟨int⟩ ) = -1`
	…

→ Generate *more* inputs – *with* and *without* deciding feature!

Alexander Kampmann, Nikolas Havrikov, Ezekiel Soremekun, and Andreas Zeller. **When does my Program do this? Learning Circumstances of Software Behavior.** ESEC/FSE 2020.

Specifically, for every path in the tree, we generate more inputs.



Training a Decision Tree

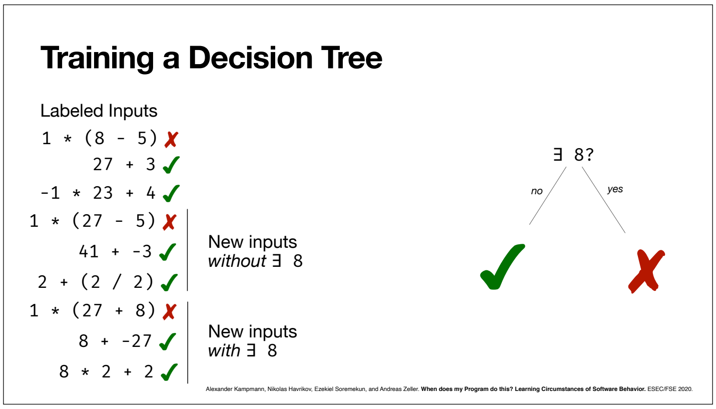
Labeled Inputs

```
1 * (8 - 5) ✗

     27 + 3 ✓

-1 * 23 + 4 ✓
```

Alexander Kampmann, Nikolas Havrikov, Ezekiel Soremekun, and Andreas Zeller. **When does my Program do this? Learning Circumstances of Software Behavior.** ESEC/FSE 2020.



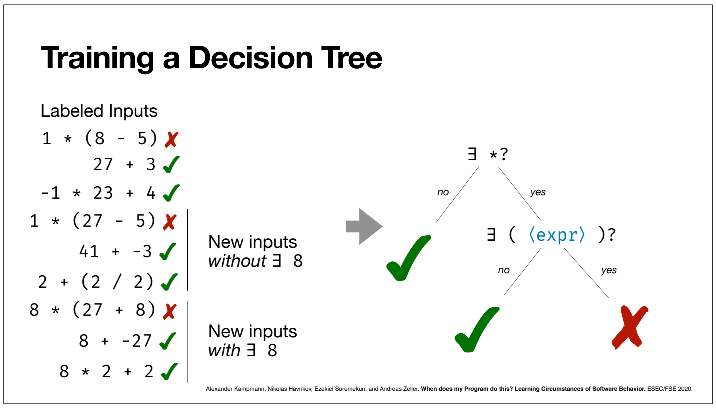
Training a Decision Tree

Labeled Inputs

```
 1 * (8 - 5) ✗
      27 + 3 ✓
 -1 * 23 + 4 ✓
1 * (27 - 5) ✗
     41 + -3 ✓   New inputs without ∃ 8
 2 + (2 / 2) ✓
1 * (27 + 8) ✗
     8 + -27 ✓   New inputs with ∃ 8
   8 * 2 + 2 ✓
```

Alexander Kampmann, Nikolas Havrikov, Ezekiel Soremekun, and Andreas Zeller. **When does my Program do this? Learning Circumstances of Software Behavior.** ESEC/FSE 2020.

So, here are more inputs with and without the digit 8. For every input, we test whether the failure occurs.



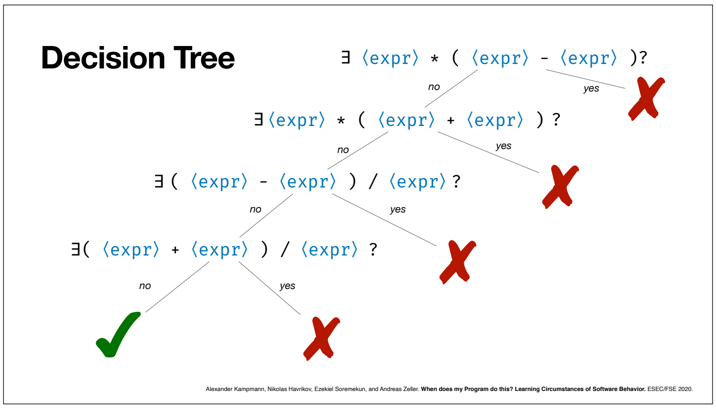
Training a Decision Tree

Labeled Inputs

```
 1 * (8 - 5) ✗
      27 + 3 ✓
 -1 * 23 + 4 ✓
1 * (27 - 5) ✗
     41 + -3 ✓   New inputs without ∃ 8
 2 + (2 / 2) ✓
8 * (27 + 8) ✗
     8 + -27 ✓   New inputs with ∃ 8
   8 * 2 + 2 ✓
```

Alexander Kampmann, Nikolas Havrikov, Ezekiel Soremekun, and Andreas Zeller. **When does my Program do this? Learning Circumstances of Software Behavior.** ESEC/FSE 2020.

For these inputs, the old hypothesis no longer holds. The decision tree now comes up with a more detailed model.

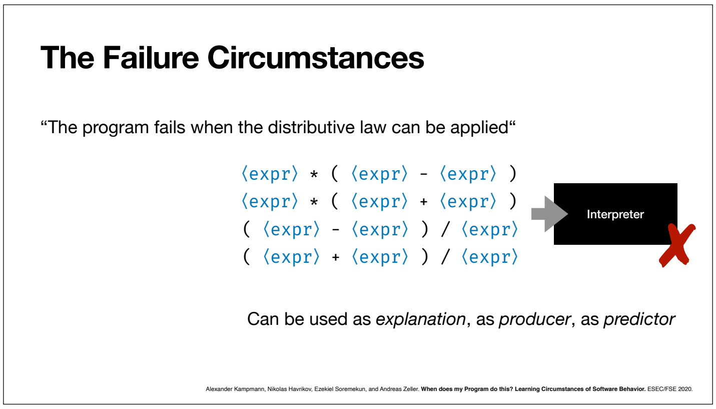


Decision Tree

∃ ⟨expr⟩ * (⟨expr⟩ - ⟨expr⟩)?
no
yes
✗
∃ ⟨expr⟩ * (⟨expr⟩ + ⟨expr⟩) ?
no
yes
✗
∃ (⟨expr⟩ - ⟨expr⟩) / ⟨expr⟩ ?
no
yes
✗
∃(⟨expr⟩ + ⟨expr⟩) / ⟨expr⟩ ?
no
yes
✓
✗

Alexander Kampmann, Nikolas Havrikov, Ezekiel Soremekun, and Andreas Zeller. **When does my Program do this? Learning Circumstances of Software Behavior.** ESEC/FSE 2020.

If we repeat this a number of times, we end up with this decision tree which now accurately characterizes the circumstances of failure.



The Failure Circumstances

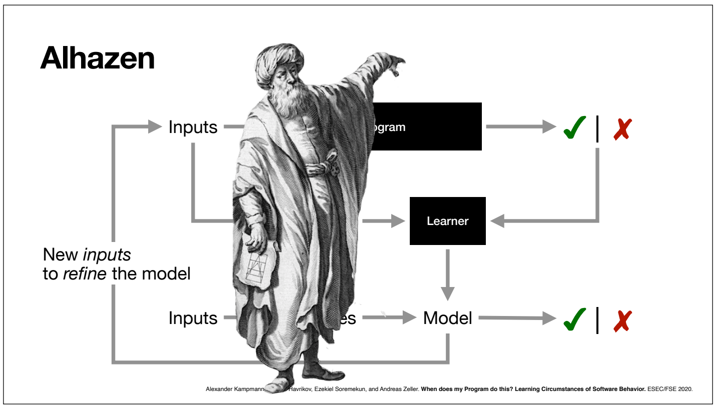
“The program fails when the distributive law can be applied“

```
⟨expr⟩ * ( ⟨expr⟩ - ⟨expr⟩ )
⟨expr⟩ * ( ⟨expr⟩ + ⟨expr⟩ )
( ⟨expr⟩ - ⟨expr⟩ ) / ⟨expr⟩
( ⟨expr⟩ + ⟨expr⟩ ) / ⟨expr⟩
```

Can be used as *explanation*, as *producer*, as *predictor*

Alexander Kampmann, Nikolas Havrikov, Ezekiel Soremekun, and Andreas Zeller. **When does my Program do this? Learning Circumstances of Software Behavior.** ESEC/FSE 2020.

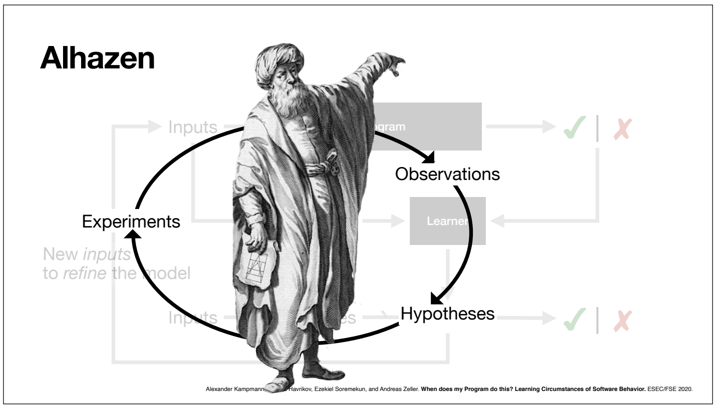
And this now tells us under which circumstance the failure occurs – namely, whenever the distributive law can be applied.



Alhazen

Alexander Kampmann, Havrikov, Ezekiel Soremekun, and Andreas Zeller. **When does my Program do this? Learning Circumstances of Software Behavior.** ESEC/FSE 2020.

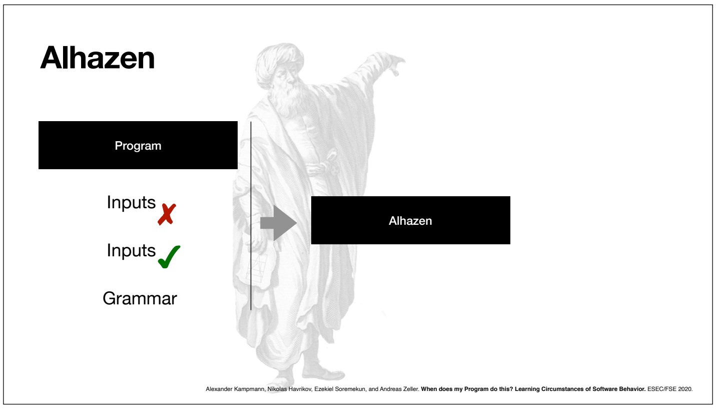
We named our approach **Alhazen**, after Ḥasan Ibn al-Haytham (Latinized as Alhazen /æl'hæzən/; full name Abū ʿAlī al-Ḥasan ibn al-Ḥasan ibn al-Haytham أبو علي، الحسن بن الحسن بن الهيثم; c.965 – c.1040) – an Arab mathematician, astronomer, and physicist of the Islamic Golden Age.



Alhazen

Alexander Kampmann, Havrikov, Ezekiel Soremekun, and Andreas Zeller. **When does my Program do this? Learning Circumstances of Software Behavior.** ESEC/FSE 2020.

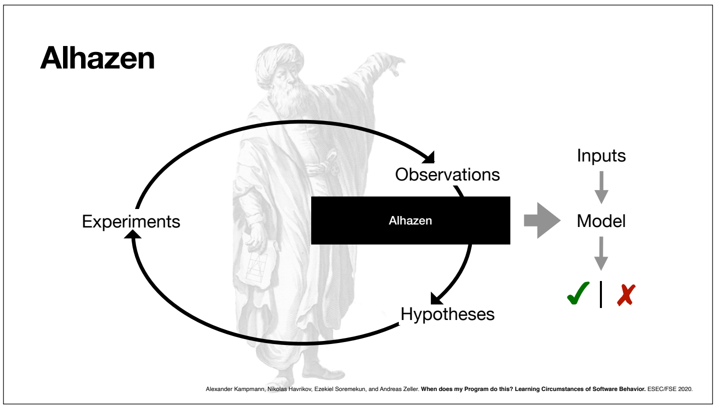
Alhazen was an early proponent of the concept that a hypothesis must be supported by experiments based on confirmable procedures or mathematical evidence—an early pioneer in the scientific method five centuries before Renaissance scientists.



Alhazen

Alexander Kampmann, Nikolas Havrikov, Ezekiel Soremekun, and Andreas Zeller. **When does my Program do this? Learning Circumstances of Software Behavior.** ESEC/FSE 2020.

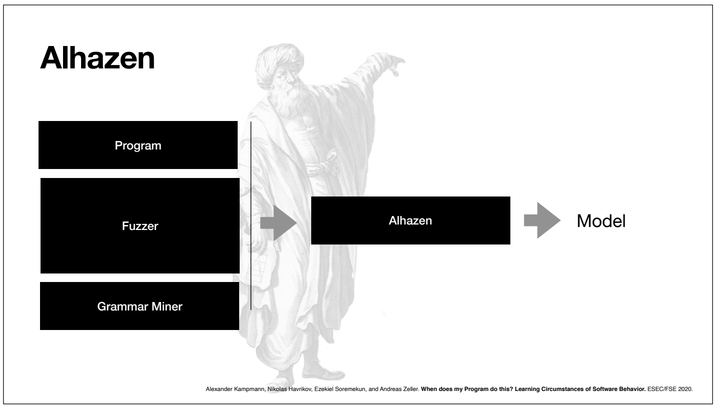
Alhazen takes a program, failing and passing inputs, and a grammar.



Alhazen

Alexander Kampmann, Nikolas Havrikov, Ezekiel Soremekun, and Andreas Zeller. **When does my Program do this? Learning Circumstances of Software Behavior.** ESEC/FSE 2020.

By abstracting over observations, and gradually refining hypothesis through experiments, Alhazen produces a predictive (and generative) model on whether failures occur or not.



Alhazen

Program

Fuzzer

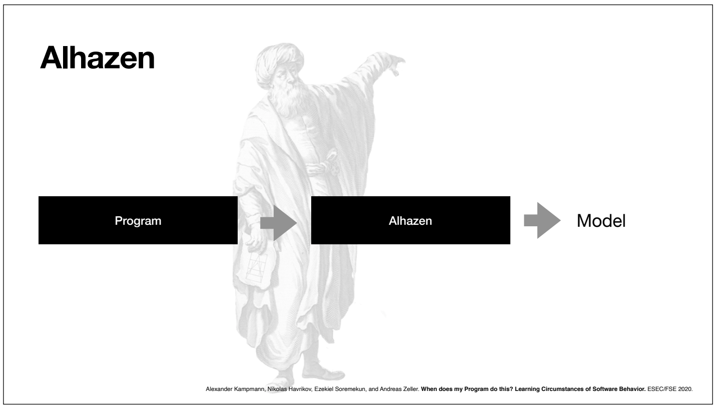
Grammar Miner

Alhazen

Model

Alexander Kampmann, Nikolas Havrikov, Ezekiel Soremekun, and Andreas Zeller. **When does my Program do this? Learning Circumstances of Software Behavior.** ESEC/FSE 2020.

Since the passing and failing inputs can come from a fuzzer, and since the grammar can come from a miner, ...



Alhazen

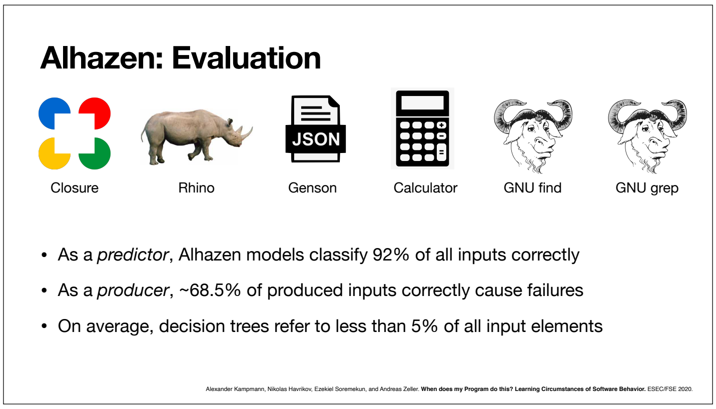
Program

Alhazen

Model

Alexander Kampmann, Nikolas Havrikov, Ezekiel Soremekun, and Andreas Zeller. **When does my Program do this? Learning Circumstances of Software Behavior.** ESEC/FSE 2020.

... Alhazen actually only need the program to be debugged to produce a model.



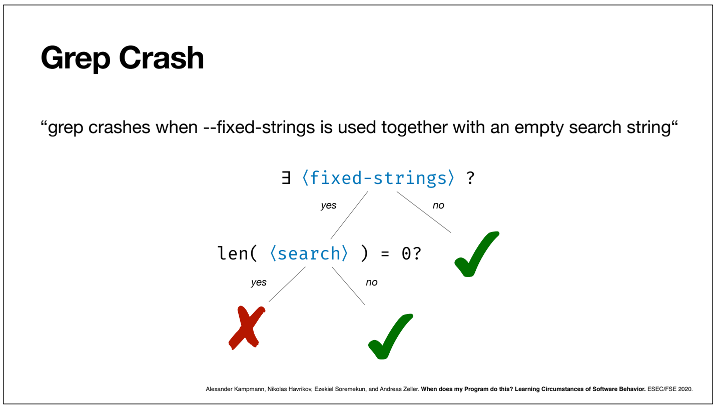
Alhazen: Evaluation

Closure
Rhino
JSON
Genson
Calculator
GNU find
GNU grep

- As a *predictor*, Alhazen models classify 92% of all inputs correctly
- As a *producer*, ~68.5% of produced inputs correctly cause failures
- On average, decision trees refer to less than 5% of all input elements

Alexander Kampmann, Nikolas Havrikov, Ezekiel Soremekun, and Andreas Zeller. **When does my Program do this? Learning Circumstances of Software Behavior.** ESEC/FSE 2020.

Alhazen works great as a predictor and as a producer. Also, the decision trees refer to a small subset of the input grammar, allowing developers to focus on these.



Grep Crash

“grep crashes when --fixed-strings is used together with an empty search string“

∃ ⟨fixed-strings⟩ ?

yes: len(⟨search⟩) = 0?

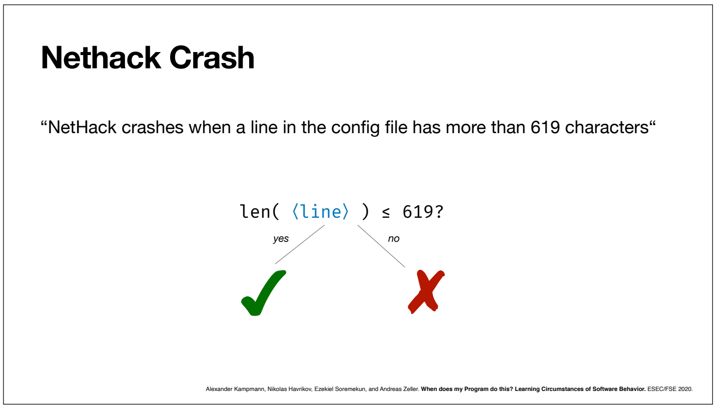
yes: ✗

no: ✓

no: ✓

Alexander Kampmann, Nikolas Havrikov, Ezekiel Soremekun, and Andreas Zeller. **When does my Program do this? Learning Circumstances of Software Behavior.** ESEC/FSE 2020.

Here is an example. Alhazen correctly determines the circumstances of a grep crash.



Nethack Crash

“NetHack crashes when a line in the config file has more than 619 characters“

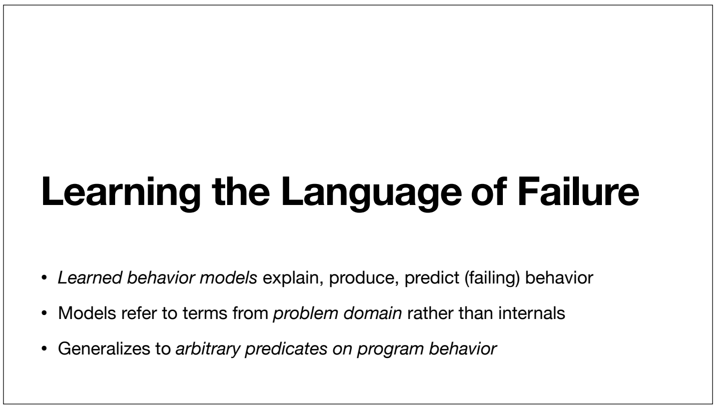
len(⟨line⟩) ≤ 619?

yes: ✓

no: ✗

Alexander Kampmann, Nikolas Havrikov, Ezekiel Soremekun, and Andreas Zeller. **When does my Program do this? Learning Circumstances of Software Behavior.** ESEC/FSE 2020.

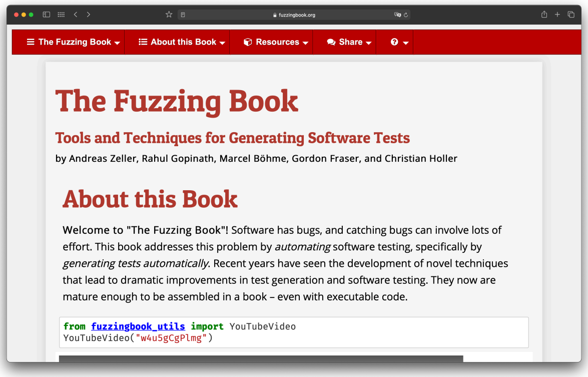
Since my time as a PhD student, I always wanted to have a slide with NetHack on it. This is how Alhazen explains the circumstances of a NetHack crash.



Learning the Language of Failure

- *Learned behavior models* explain, produce, predict (failing) behavior
- Models refer to terms from *problem domain* rather than internals
- Generalizes to *arbitrary predicates on program behavior*

So this is learning the language of failure – the set of inputs that causes a program to fail.



The Fuzzing Book

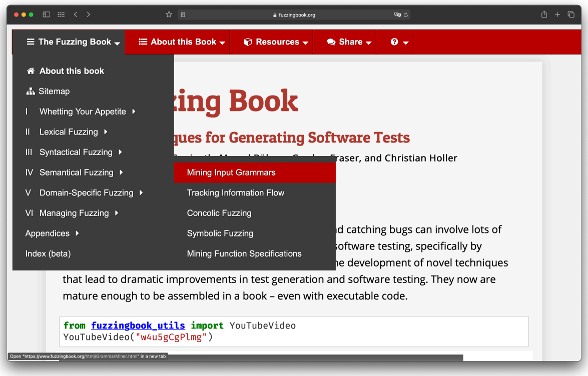
Tools and Techniques for Generating Software Tests

by Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler

About this Book

Welcome to "The Fuzzing Book"! Software has bugs, and catching bugs can involve lots of effort. This book addresses this problem by *automating* software testing, specifically by *generating tests automatically*. Recent years have seen the development of novel techniques that lead to dramatic improvements in test generation and software testing. They now are mature enough to be assembled in a book – even with executable code.

```
from fuzzingbook_utils import YouTubeVideo
YouTubeVideo("w4u5gCgPlmg")
```



ing Book

ques for Generating Software Tests

raser, and Christian Holler

d catching bugs can involve lots of

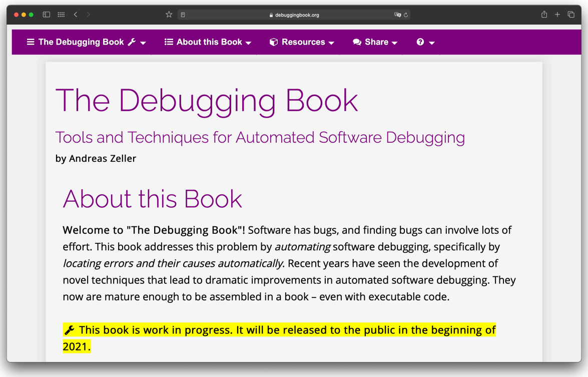
software testing, specifically by

ne development of novel techniques

that lead to dramatic improvements in test generation and software testing. They now are mature enough to be assembled in a book – even with executable code.

```
from fuzzingbook_utils import YouTubeVideo
YouTubeVideo("w4u5gCgPlmg")
```

Open "https://www.fuzzingbook.org/html/GrammarMiner.html" in a new tab



The Debugging Book

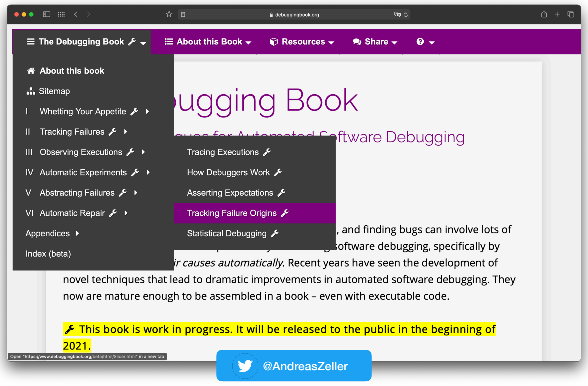
Tools and Techniques for Automated Software Debugging

by Andreas Zeller

About this Book

Welcome to "The Debugging Book"! Software has bugs, and finding bugs can involve lots of effort. This book addresses this problem by *automating* software debugging, specifically by *locating errors and their causes automatically*. Recent years have seen the development of novel techniques that lead to dramatic improvements in automated software debugging. They now are mature enough to be assembled in a book – even with executable code.

This book is work in progress. It will be released to the public in the beginning of 2021.

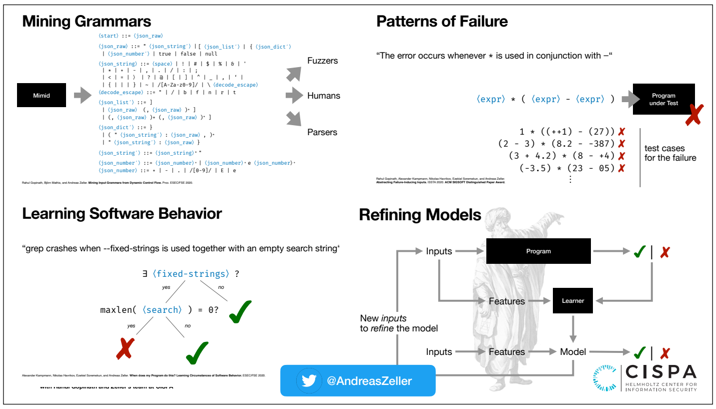


ugging Book

ques for Automated Software Debugging

, and finding bugs can involve lots of software debugging, specifically by ir causes automatically. Recent years have seen the development of novel techniques that lead to dramatic improvements in automated software debugging. They now are mature enough to be assembled in a book – even with executable code.

This book is work in progress. It will be released to the public in the beginning of 2021.



Mining Grammars

```
⟨start⟩ ::= ⟨json_raw⟩
⟨json_raw⟩ ::= " ⟨json_string′⟩ |[ ⟨json_list′⟩ | { ⟨json_dict′⟩
 | ⟨json_number′⟩ | true | false | null
⟨json_string⟩ ::= ⟨space⟩ | ! | # | $ | % | & | '
 | * | + | - | , | . | / | : | ;
 | < | = | ⟩ | ? | @ | [ | ] | ^ | _ | , | ‘ |
 | { | | | } | ~ | /[A-Za-z0-9]/ | \ ⟨decode_escape⟩
⟨decode_escape⟩ ::= " | / | b | f | n | r | t
⟨json_list′⟩ ::= ]
 | ⟨json_raw⟩  (, ⟨json_raw⟩ )* ]
 | (, ⟨json_raw⟩ )+ (, ⟨json_raw⟩ )* ]
⟨json_dict′⟩ ::= }
 | ( " ⟨json_string′⟩ : ⟨json_raw⟩ , )*
 | " ⟨json_string′⟩ : ⟨json_raw⟩ }
⟨json_string′⟩ ::= ⟨json_string⟩* "
⟨json_number′⟩ ::= ⟨json_number⟩* | ⟨json_number⟩* e ⟨json_number⟩*
⟨json_number⟩ ::= + | - | . | /[0-9]/ | E | e
```

Rahul Gopinath, Björn Mathis, and Andreas Zeller. **Mining Input Grammars from Dynamic Control Flow.** Proc. ESEC/FSE 2020.

Patterns of Failure

“The error occurs whenever * is used in conjunction with –“

```
⟨expr⟩ * ( ⟨expr⟩ - ⟨expr⟩ )

   1 * ((++1) - (27)) ✗
(2 - 3) * (8.2 - -387) ✗
 (3 + 4.2) * (8 - +4) ✗
  (-3.5) * (23 - 05) ✗
          ⋮
```

Rahul Gopinath, Alexander Kampmann, Nikolas Havrikov, Ezekiel Soremekun, and Andreas Zeller. **Abstracting Failure-Inducing Inputs.** ISSTA 2020. **ACM SIGSOFT Distinguished Paper Award.**

Learning Software Behavior

“grep crashes when --fixed-strings is used together with an empty search string‘

```
∃ ⟨fixed-strings⟩ ?
```

Alexander Kampmann, Nikolas Havrikov, Ezekiel Soremekun, and Andreas Zeller. **When does my Program do this? Learning Circumstances of Software Behavior.** ESEC/FSE 2020.

with Rahul Gopinath and Zeller's team at CISPA

Refining Models

That's all! If you like this work, and want to know more, follow me on Twitter or visit my homepage at https://andreas-zeller.info/. See you!