Learning the Language of Failure

CASA Distinguished Lecture, Bochum, 2020-06-18

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

CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

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.

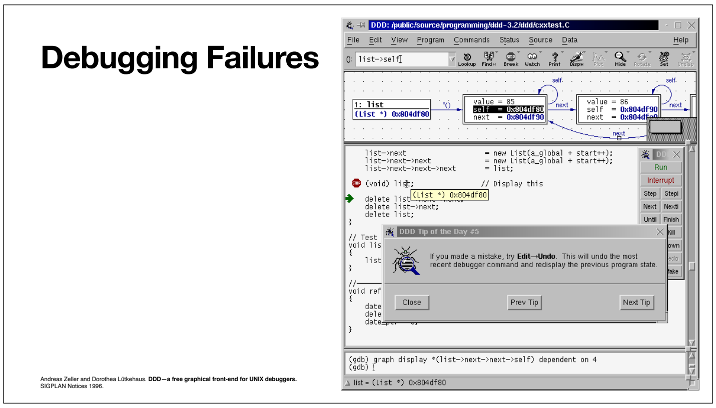
In this talk, I present and introduce tools and techniques that automatically learn circumstances of a given failure, expressed over features of input elements. I show how to automatically infer input languages as readable grammars, how to use these grammars for massive fuzzing, and how to systematically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

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

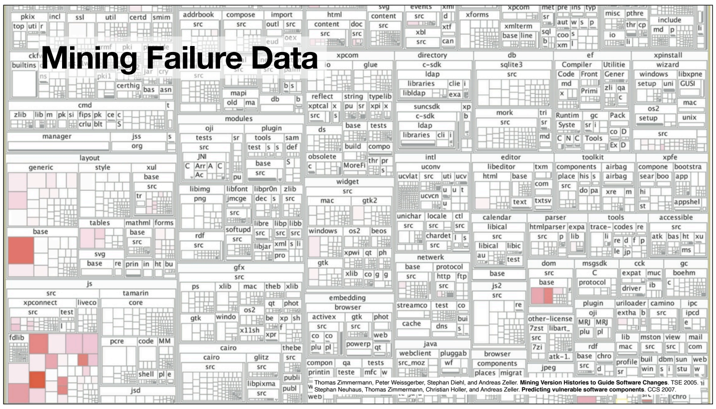
Failure

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

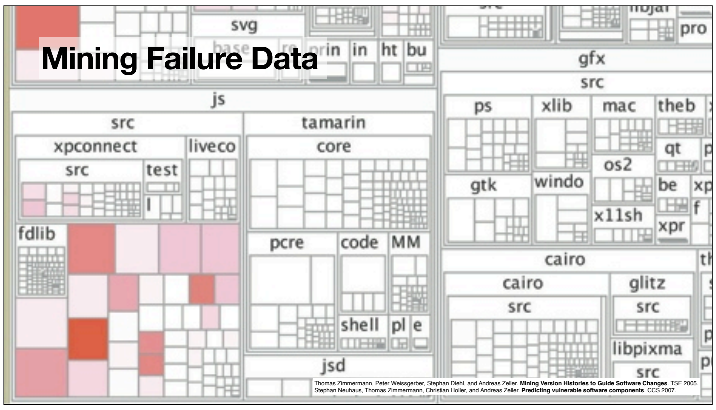
Actually, my work always has been about failures. (The work itself has been less of a failure.)



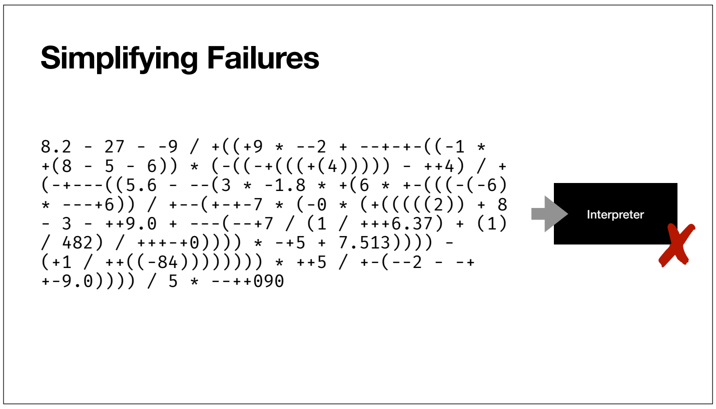
As a PhD student still, Dorothea Lütkehaus and myself built GNU DDD, a GUI front-end for command line debuggers. Great for debugging failures.



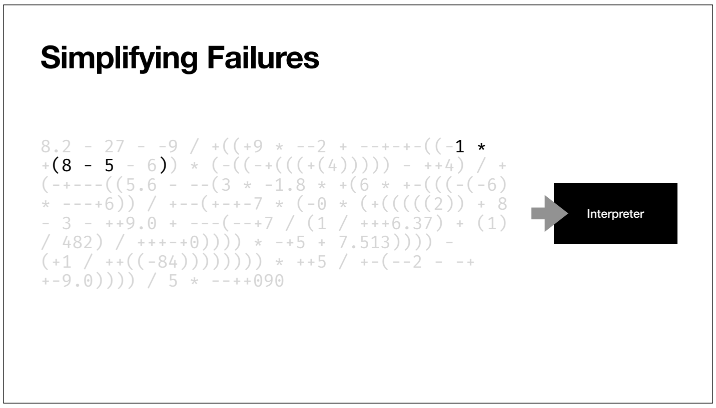
Later, my co-workers and I would mine version and bug repositories to see where in a program the most bugs would be fixed. This is a map of Firefox components (boxes) and vulnerabilities (shades of red).



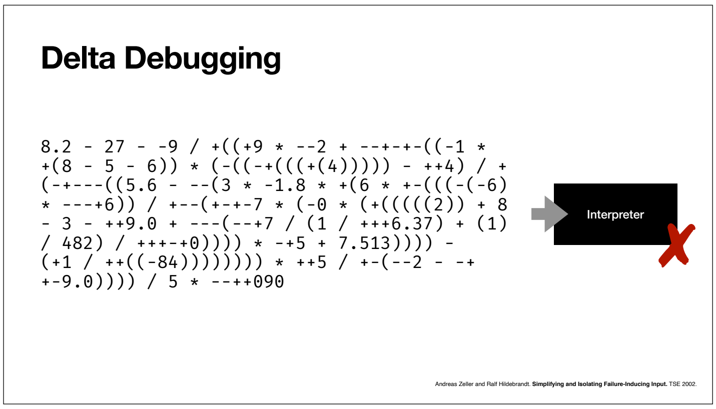
Almost all vulnerabilities are in JavaScript.



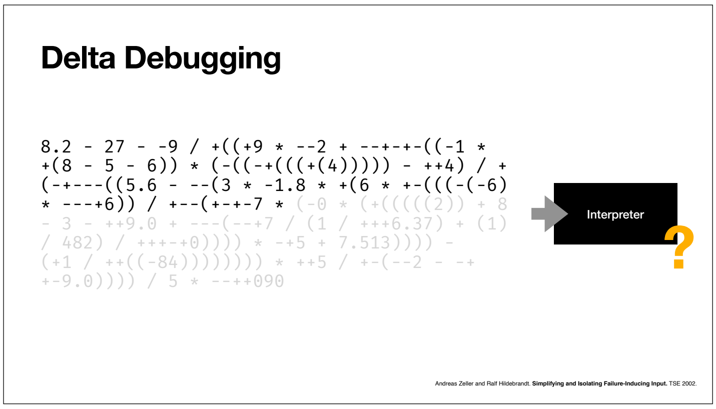
Another contribution my name is associated with is simplifying failure-inducing inputs. Here's a long input that causes a program to fail.



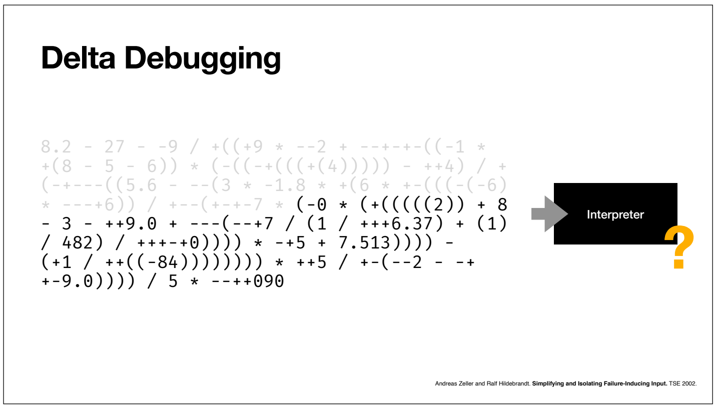
Yet, only a part of this input actually is relevant for the failure.



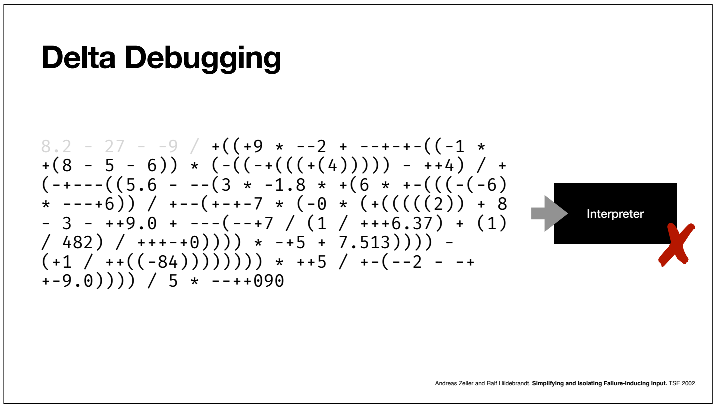
Delta debugging automatically determines this failure-inducing subset.



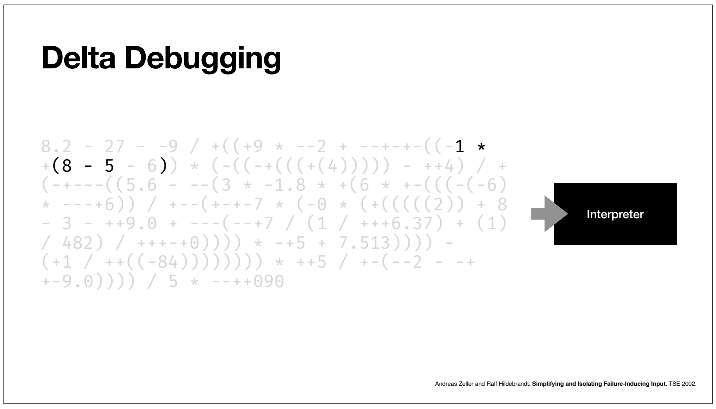
Delta Debugging takes away parts of the input and checks whether the failure still occurs.



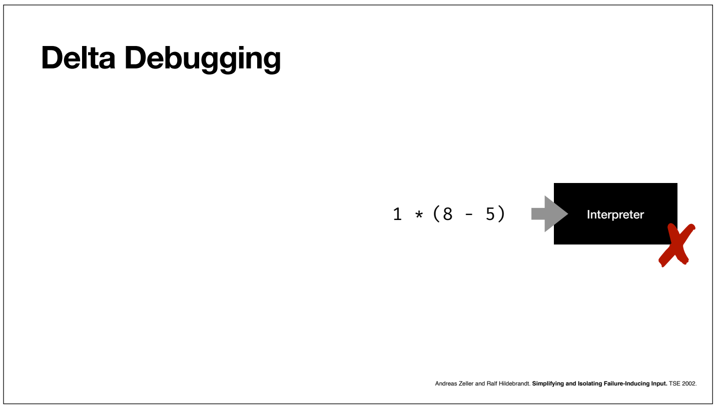
Such reduced inputs can be invalid, though.



Then, delta debugging takes out smaller parts and repeats.



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



Such as these ones, for instance.

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

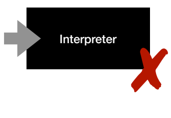
Which tells you that you can make a career out of failures.

The Language of Failure

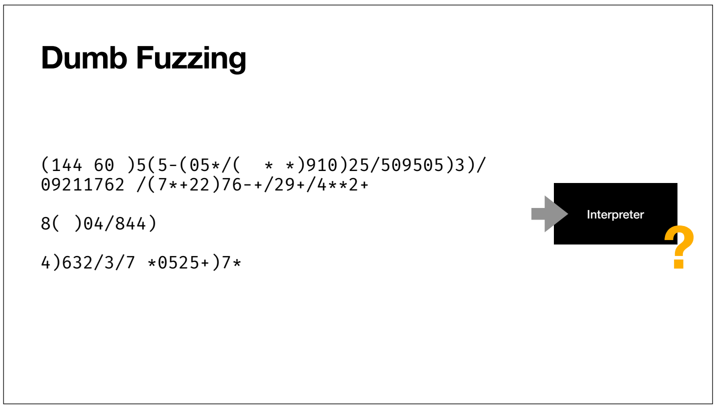
Okay, that was failures. Now, let's move to languages.

Fuzzing

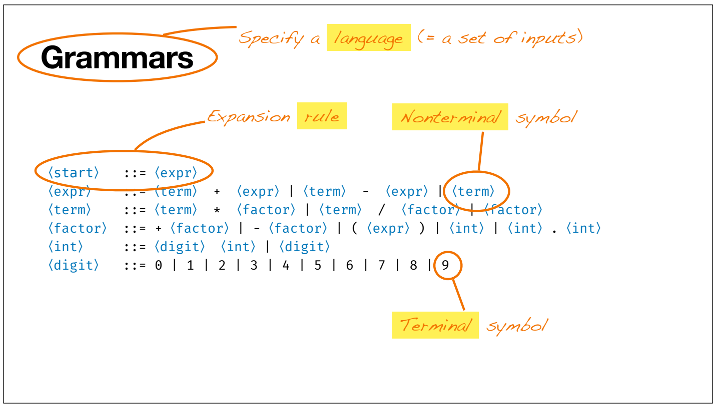
```
8.2 - 27 - -9 / +((+9 * --2 + --+-+-((-1 *
+(8 - 5 - 6)) * (-((-+(((+(4)))))) - ++4) / +
(-+---((5.6 - --(3 * -1.8 * +(6 * +-(((-(-6)
* ---+6)) / +--(+-+-7 * (-0 * (+(((((2)) + 8
- 3 - ++9.0 + ---(--+7 / (1 / +++6.37) + (1)
/ 482) / +++-+0)))) * -+5 + 7.513)))) -
(+1 / ++((-84))))))) * ++5 / +-(--2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



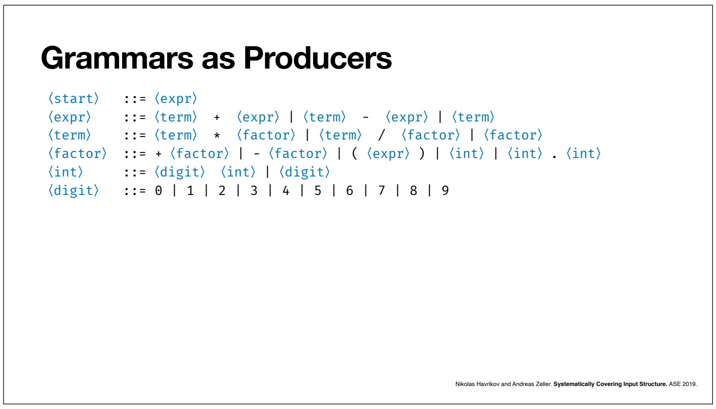
Fuzzing means to throw random inputs at a program to see if it crashes.



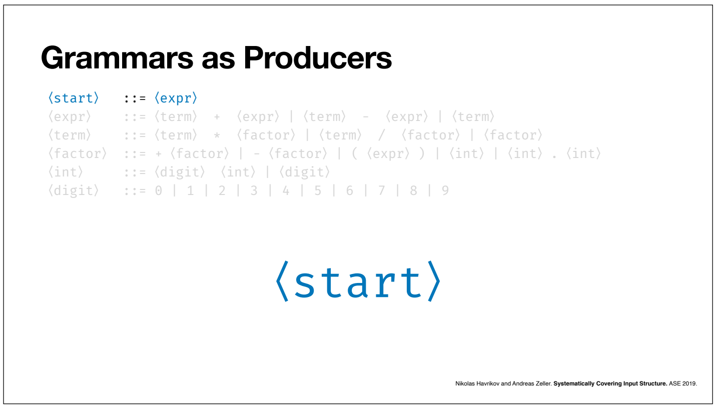
But if you just take sequences of random characters and throw them at an interpreter, all you're going to get is syntax errors. (It's okay to test syntax error handling, but this should not be all.)



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



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



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⟩

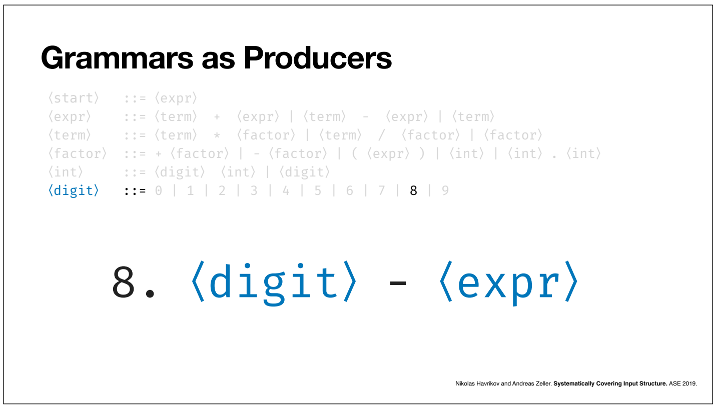
Nikolas Havrikov and Andreas Zeller. **Systematically Covering Input Structure.** ASE 2019.

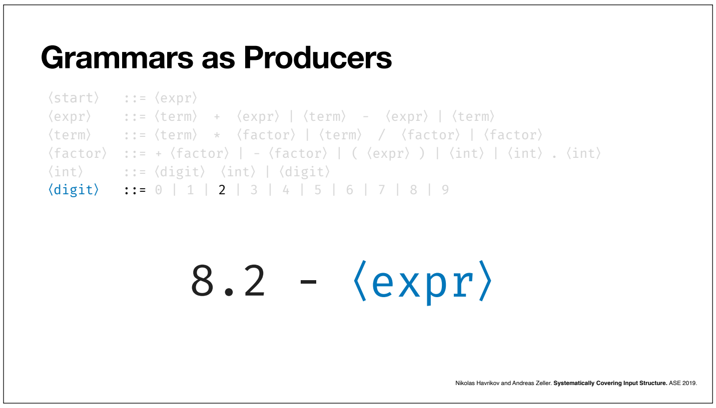
Grammars as Producers

```
⟨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
```

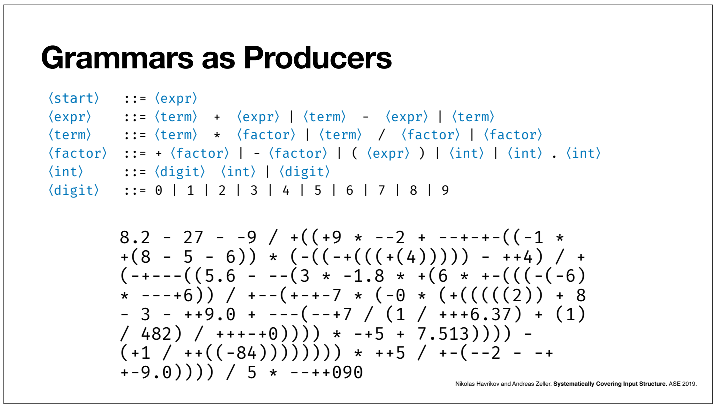
⟨digit⟩ . ⟨digit⟩ - ⟨expr⟩

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

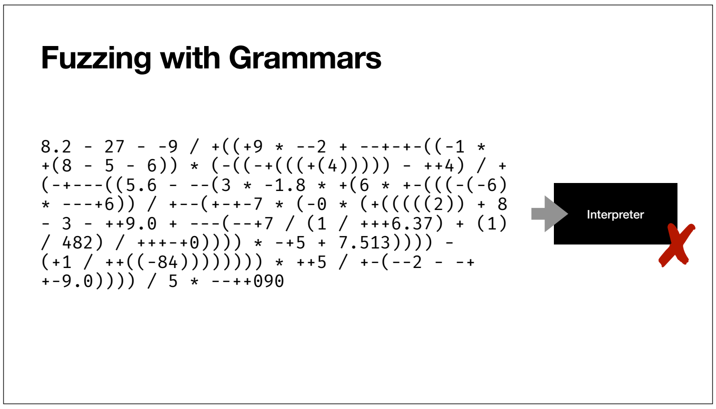




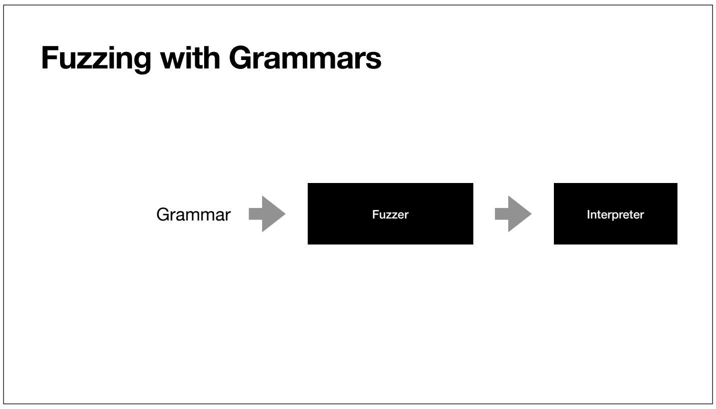
Over time, this gives you a syntactically valid input. In our case, a valid arithmetic expression.

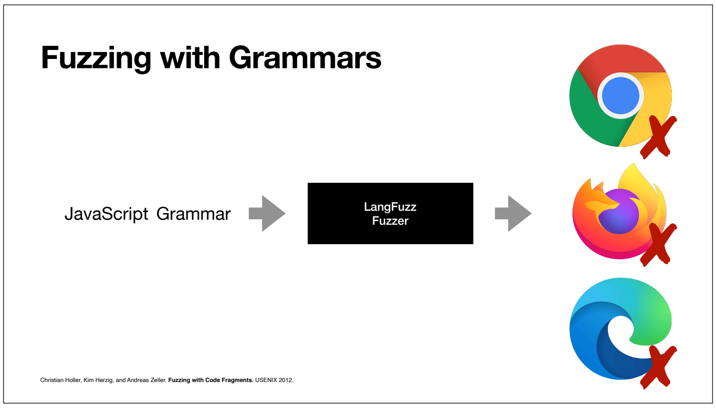


Actually, a pretty **complex** arithmetic expression.

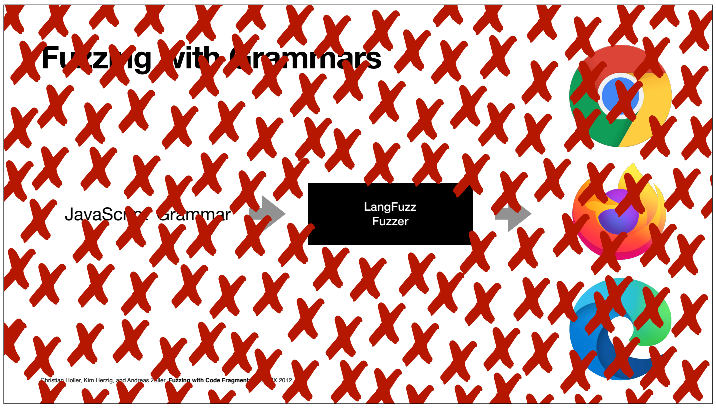


These can now be used as input to your program.

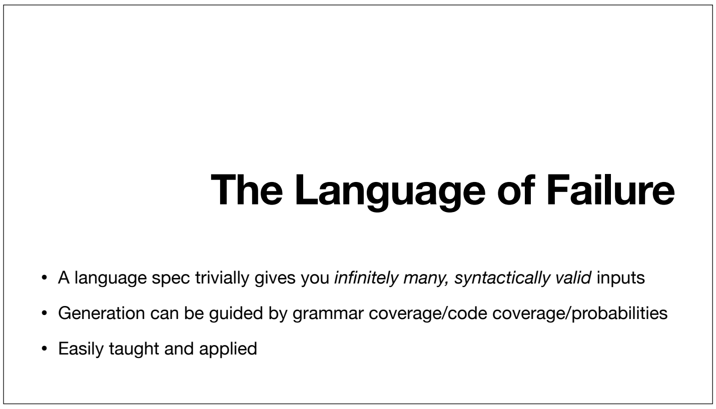




A couple of years ago, we used a JavaScript grammar to fuzz the interpreters of Firefox, Chrome and Edge.

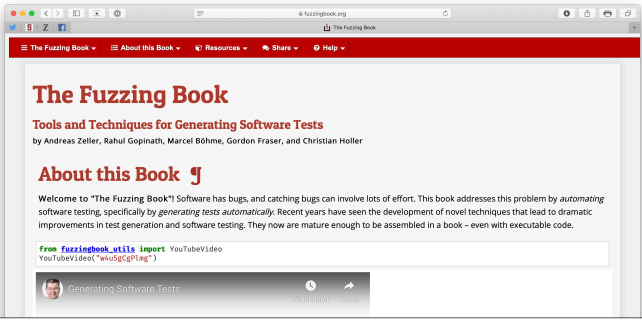


My student Christian Holler found more than 2,600 bugs, and in the first four weeks, he netted more than $50,000 in bug bounties. If you use a browser to read this, one of the reasons your browser works as it should is because of grammar-based fuzzing.



So this was the language of failure.

fuzzingbook.org



We have put all our knowledge on fuzzing and grammars into a book named the fuzzingbook, where you can actually try out all the basic algorithms yourself.

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

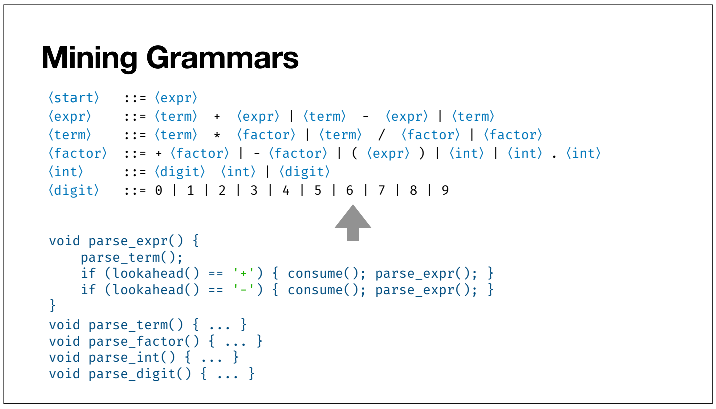
And if you are interested in how to use grammar for fuzzing, the book will give you lots of inspiration.

Learning the Language

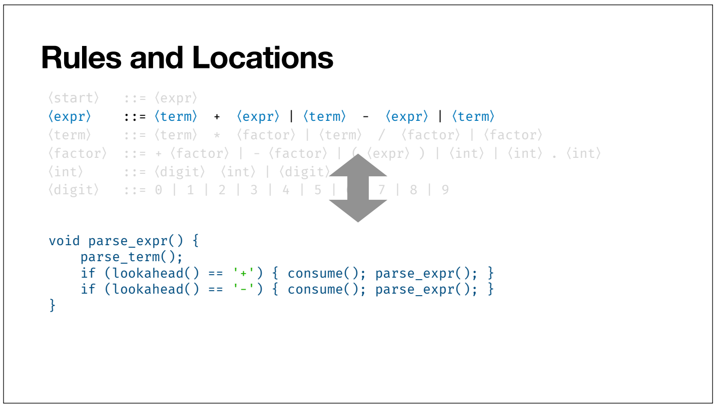
But all of this still requires a grammar in the first place.

Fuzzing with Grammars

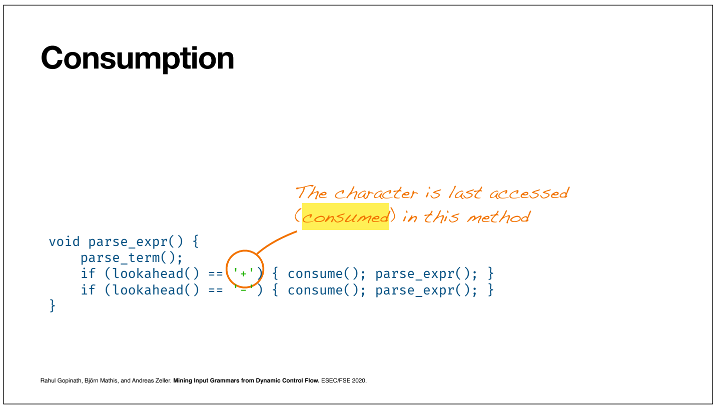
So where did you get this grammar from?



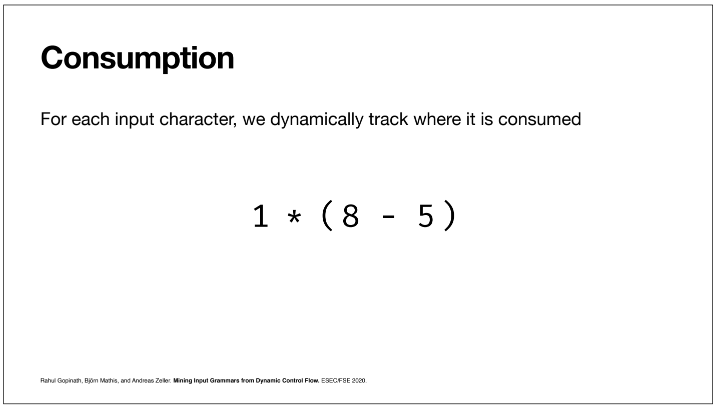
So let me tell you a bit about how to mind such grammars. The idea is to take a program that parses such inputs and extract the input grammar from it.

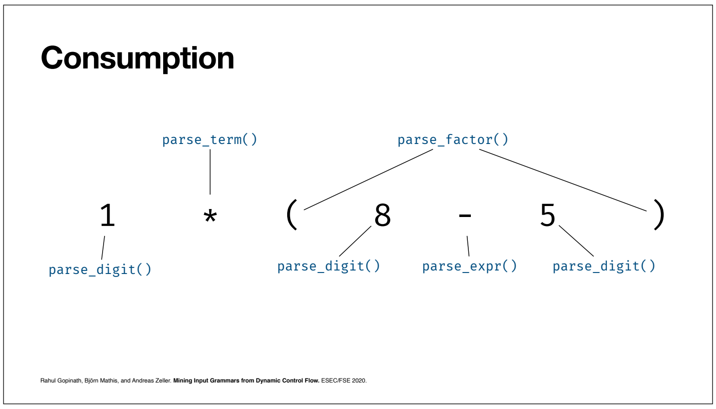


The interesting thing is that there is a correspondence between individual rules in the input grammar and locations in the parsing code.

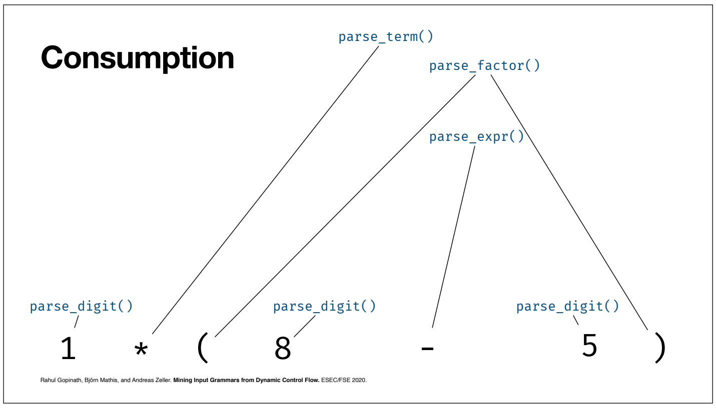


The concept of consumption establishes this correspondence. A character is **consumed** in a method *m* if *m* is the last to access it.

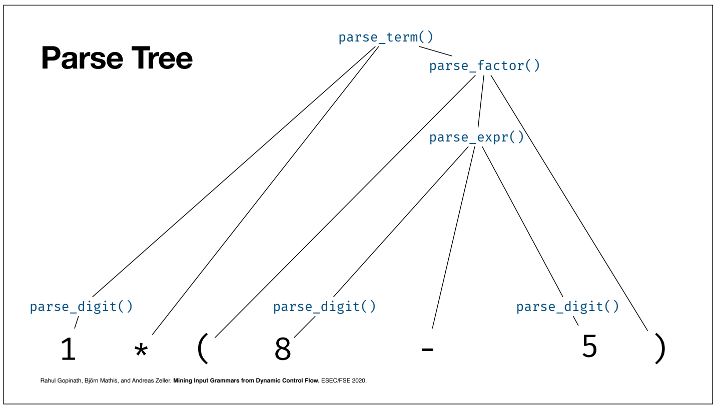




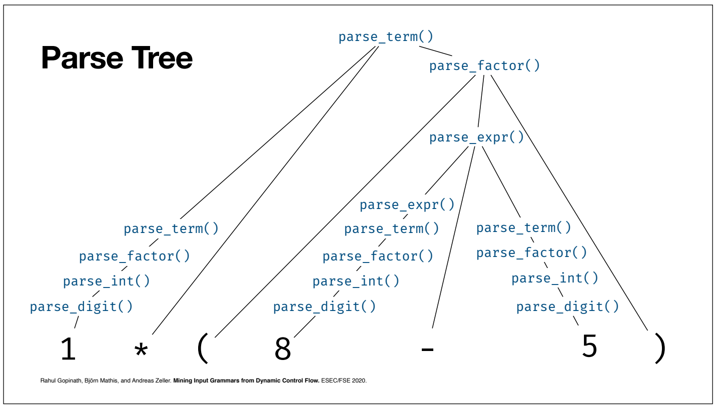
During program execution we can track where characters are consumed using dynamic tainting.



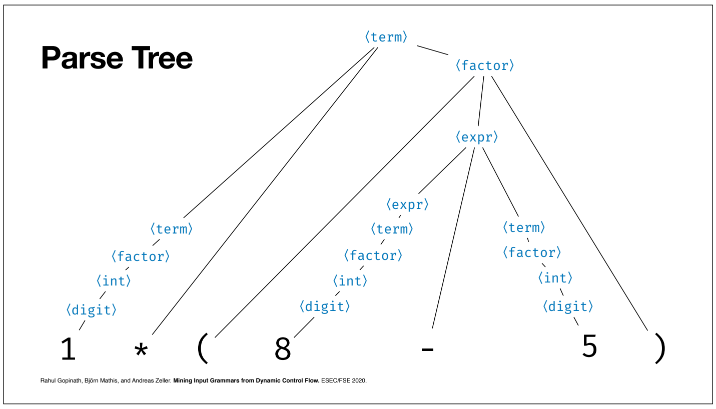
This gives us a tree like structure.



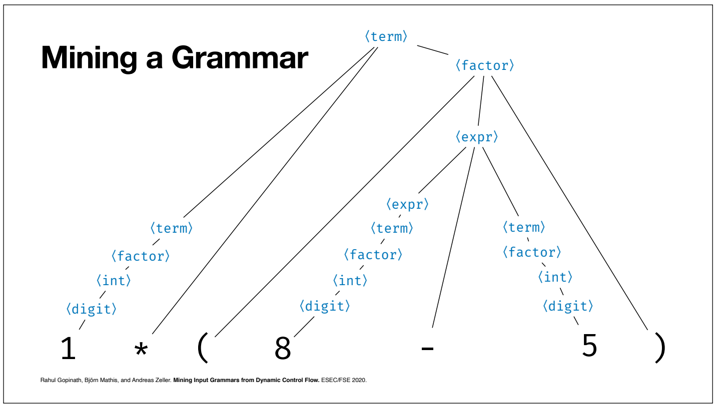
Which we can augment with caller-callee relations.



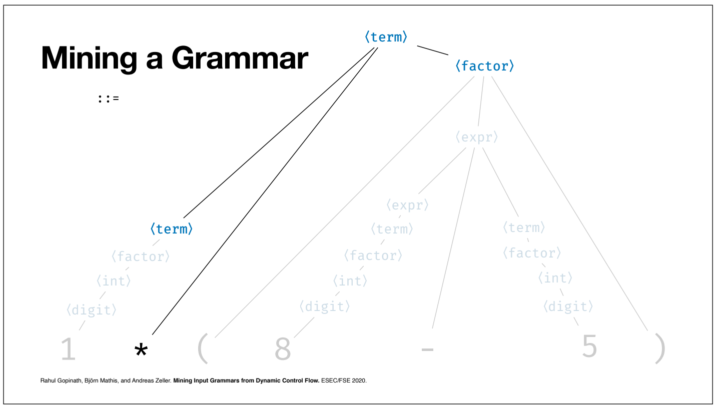
Even for those functions which do not consume anything.



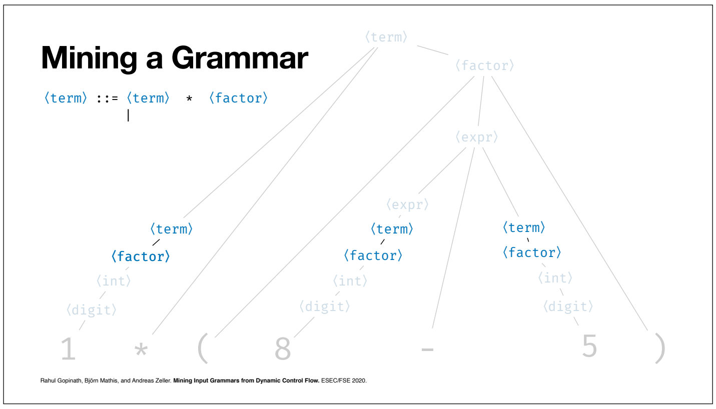
If we take the function names and only use the nouns, we can use those nouns as non-terminal symbols.



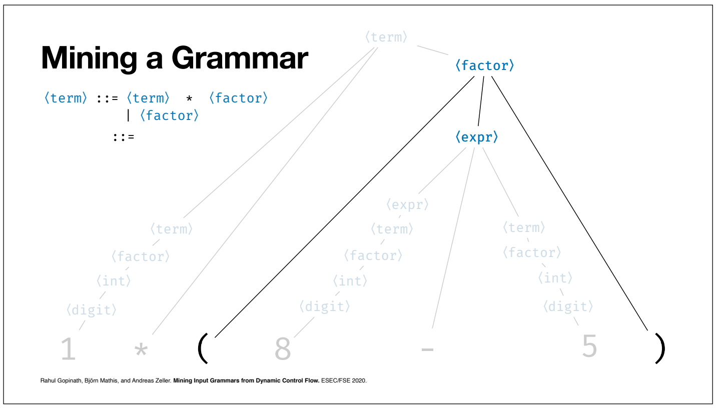
From these parse trees, we can now mine a grammar.



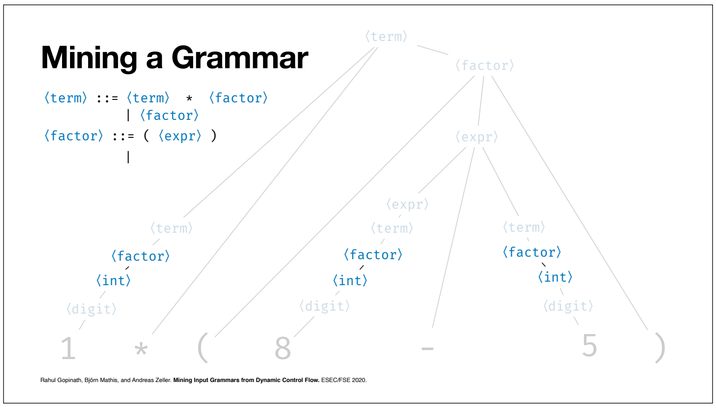
A term obviously can consist of another term, a multiplication symbol, and a factor.

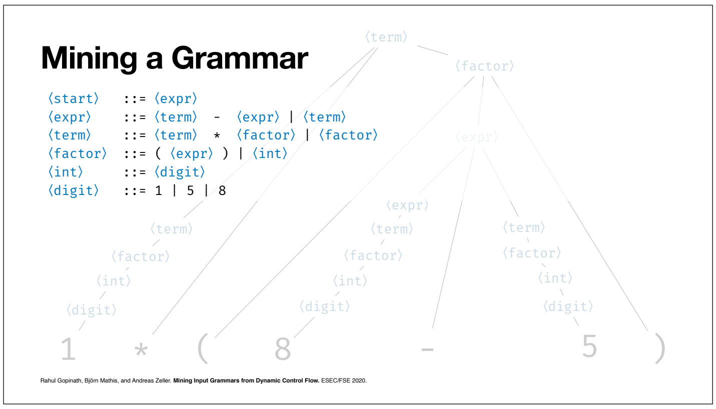


So we add this as a rule to our grammar.

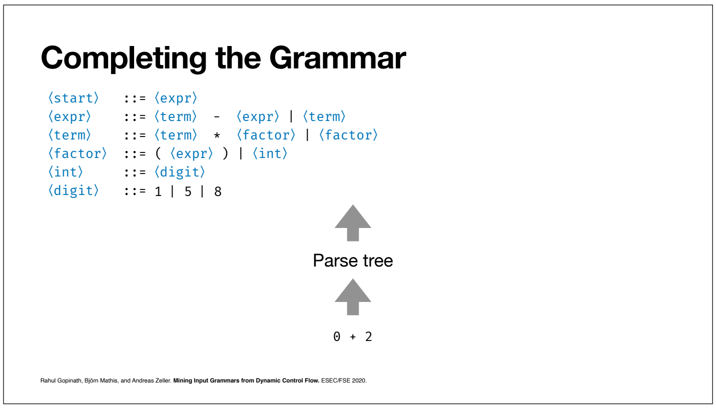


And likewise for other symbols.

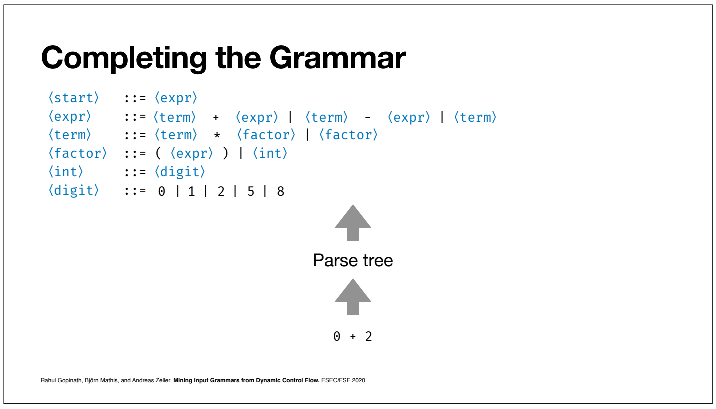




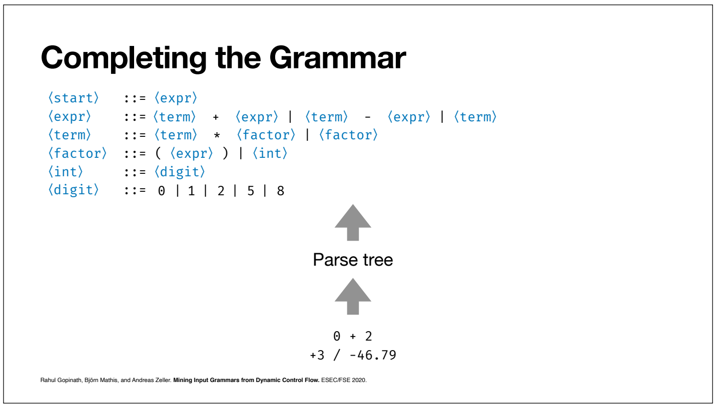
From this single input, we already get the basics of a grammar.

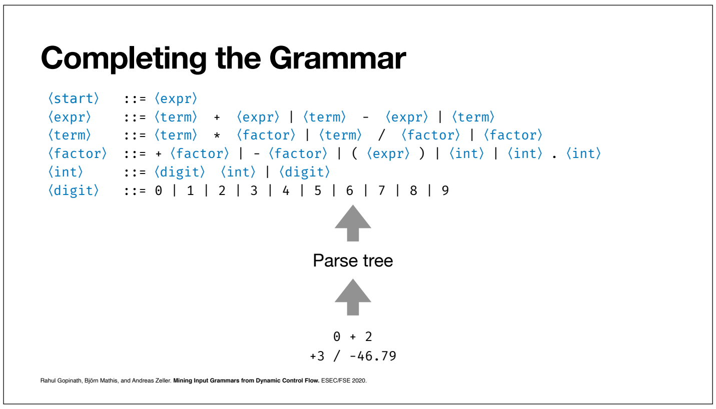


And if we add more inputs, ...

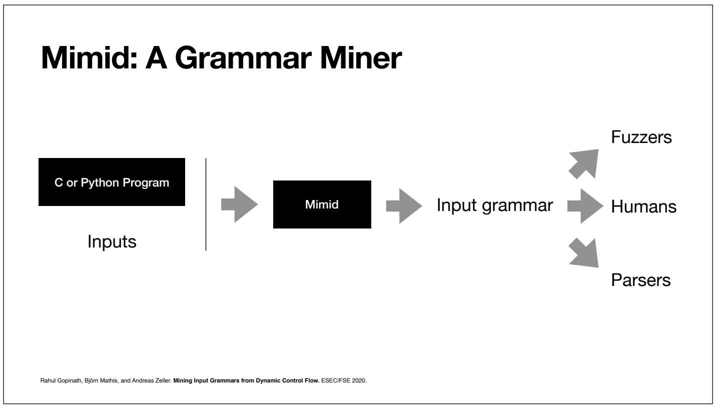


... the grammar reflects the structure of these additional inputs.





We now have successfully mined our example grammar.



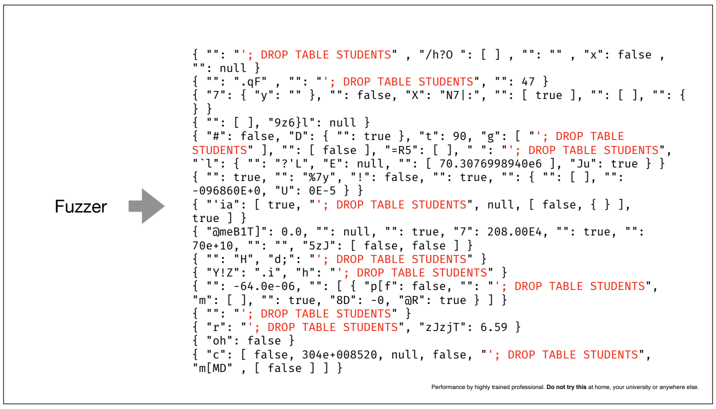
Our Mimid grammar miner takes a program and its inputs and extracts a grammar out of it. This grammar can directly be used by fuzzers, parsers, and humans.

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

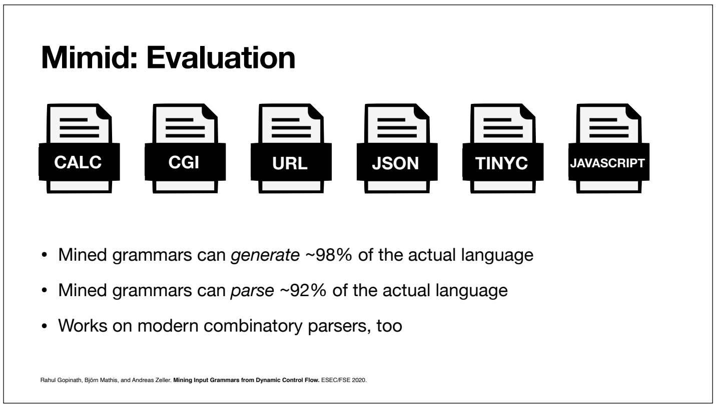
Humans can **edit** these grammars.

For instance, by assigning probabilities to individual productions.

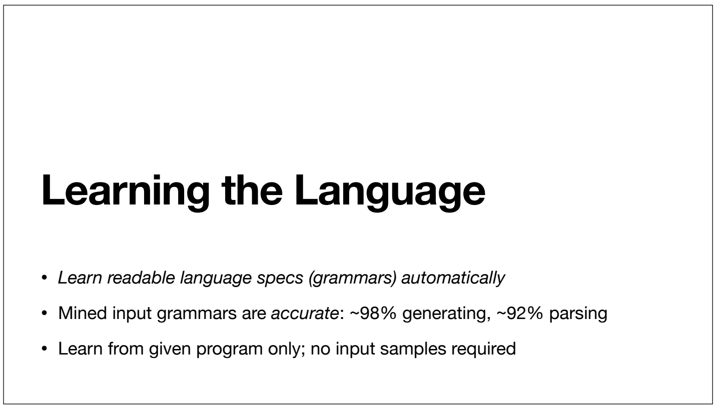
Or by inserting magic strings that program analysis would have a hard time finding out.



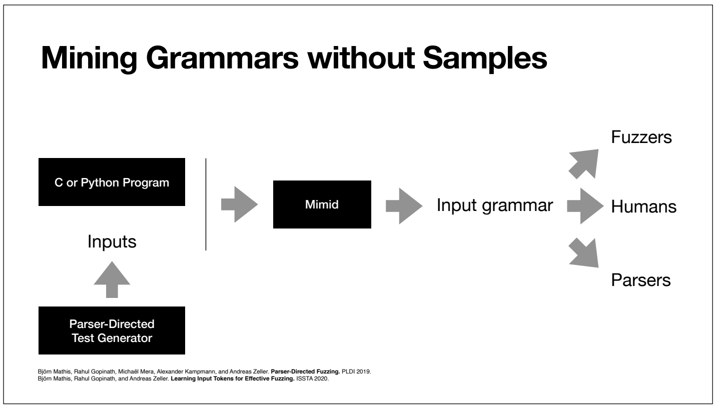
This change to the grammar injects SQL statements everywhere. Do not do this at home, folks – thank you.



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



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



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.

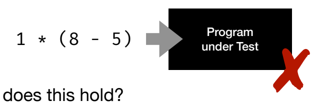
Learning the Language

- *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.

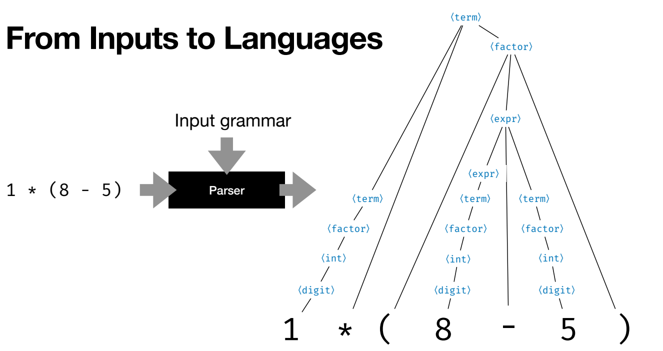
Circumstances of Failure



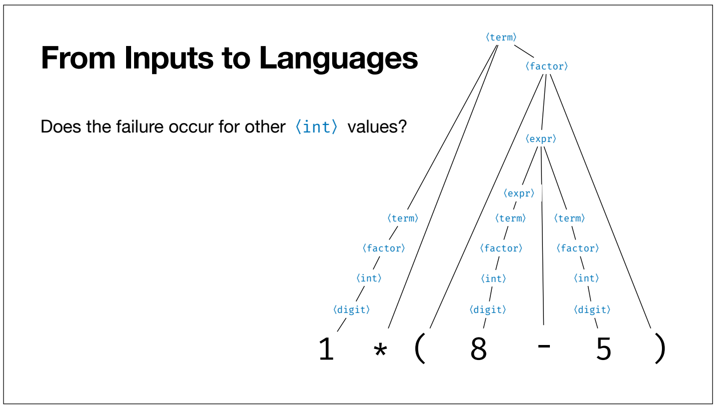
For *which other inputs* does this hold?

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

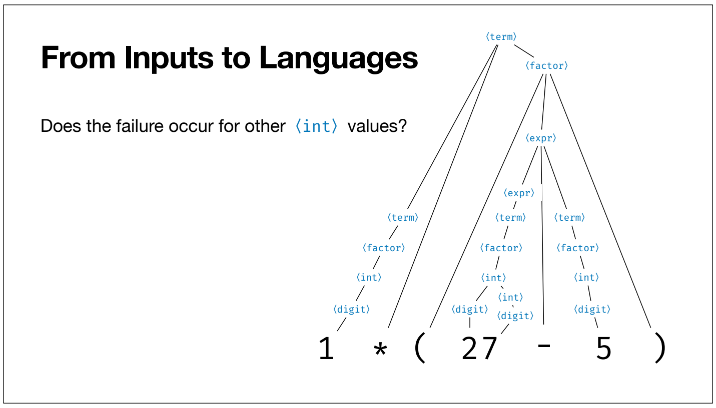
From Inputs to Languages



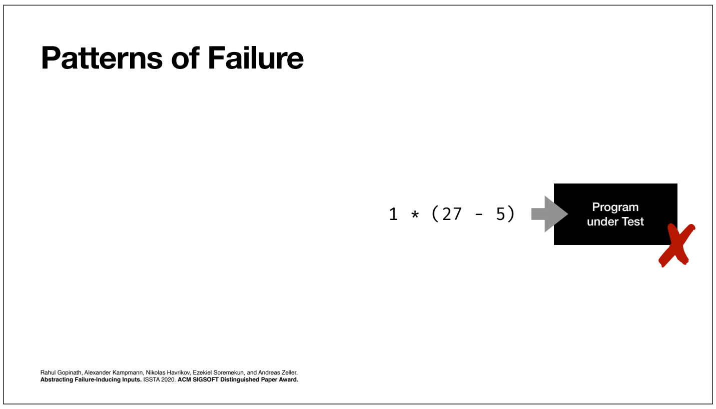
We want to know the **set of inputs** that causes the failure – in other words, the language. To this end, we parse the input into a tree.



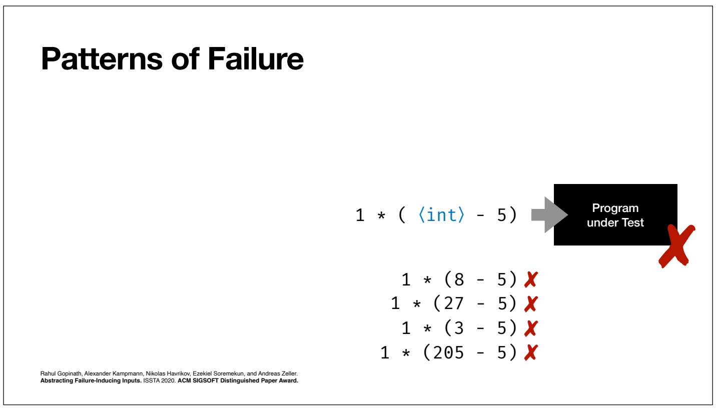
To find out whether the failure occurs for other integer values too, …



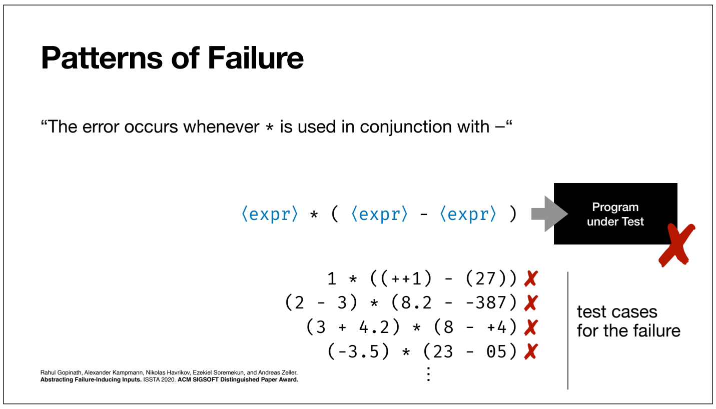
… we replace parts of the parse tree (8) by newly generated alternatives (27).



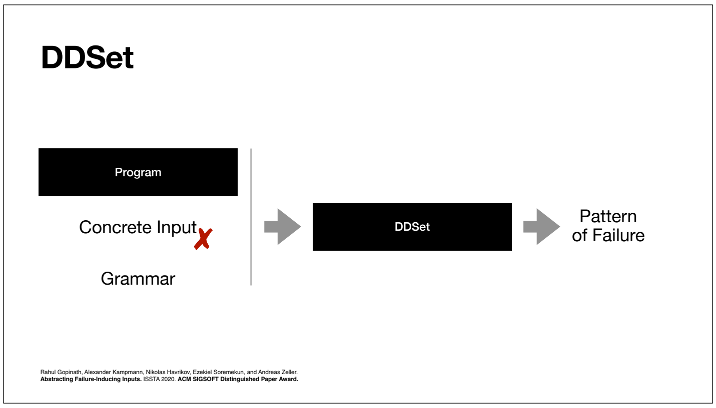
and find that this one fails as well.



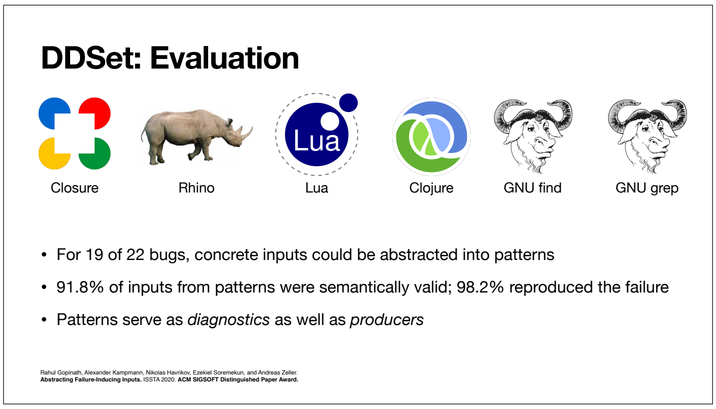
Actually, the program fails for any integer in this position. So we can come up with an abstract pattern that represents the set of failing inputs.



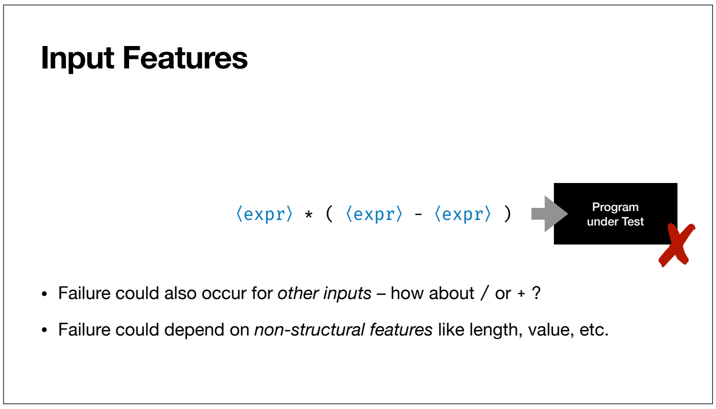
By repeating this, we can come up with a general pattern of which **all** instantiations cause the failure. These instantiations also serve as test cases for validating a fix.



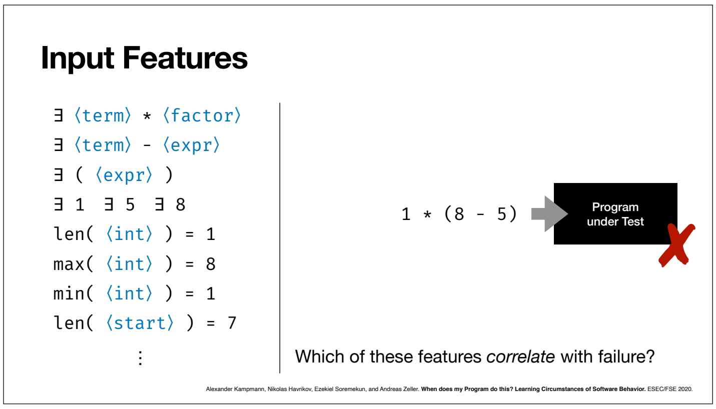
Our tool DDSet takes a program, a failing input, and a grammar, and produces such a pattern of failure.



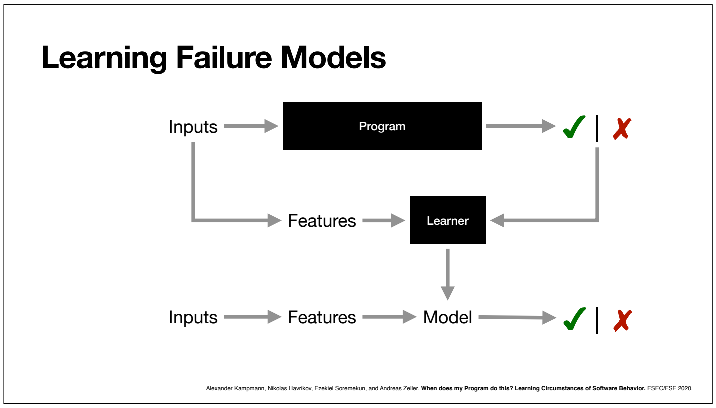
In our evaluation, this works really well.



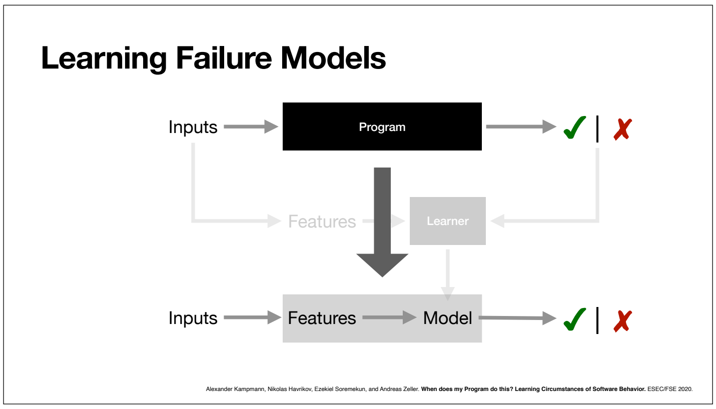
But we can go even further. What other features in the input cause a failure?



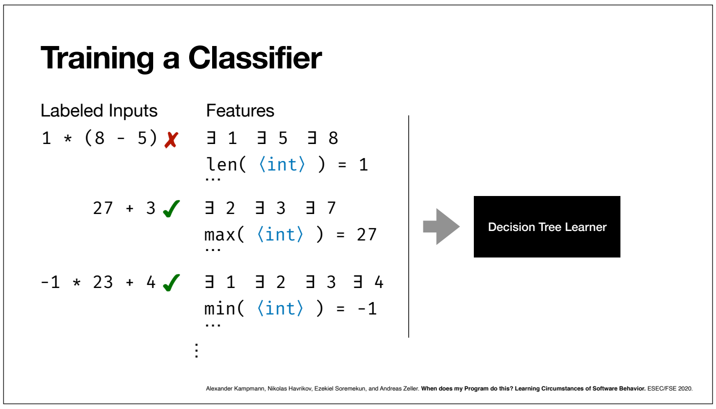
We introduce a number of input features, including existence, length, and maximum and minimum values of specific input elements.



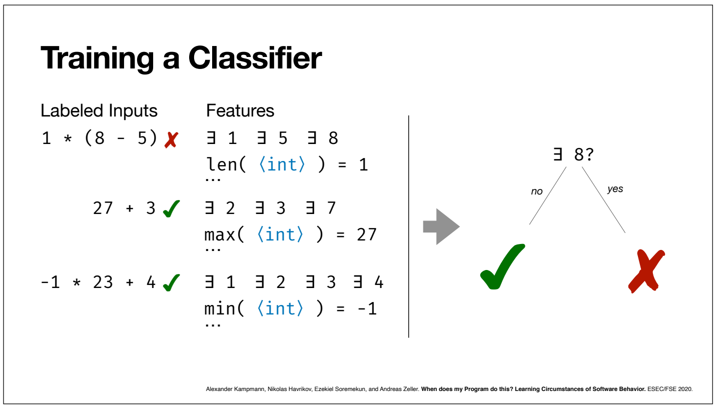
These features together with a pass and fail label then go into a machine learner which produces a predictive model.



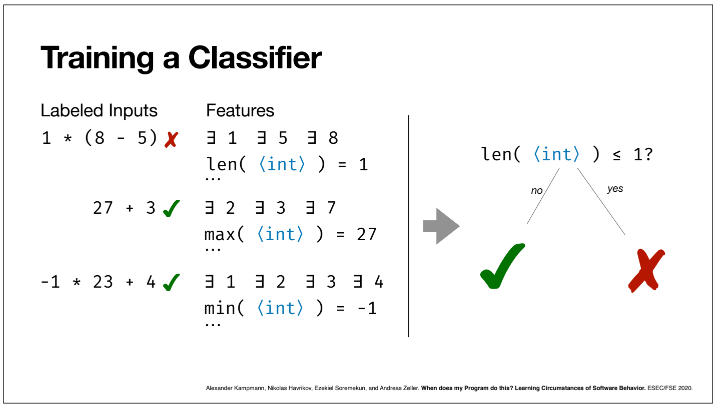
Actually, the produced model serves as a **model of the program** as it comes to failures or non-failures.



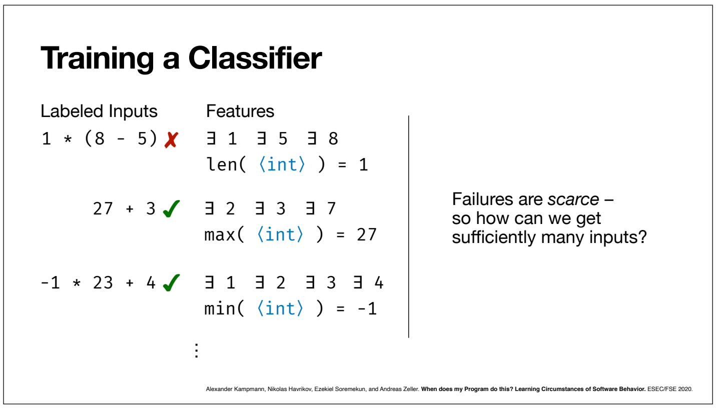
In our experiments, we use decision tree learners as their results are easy to understand.



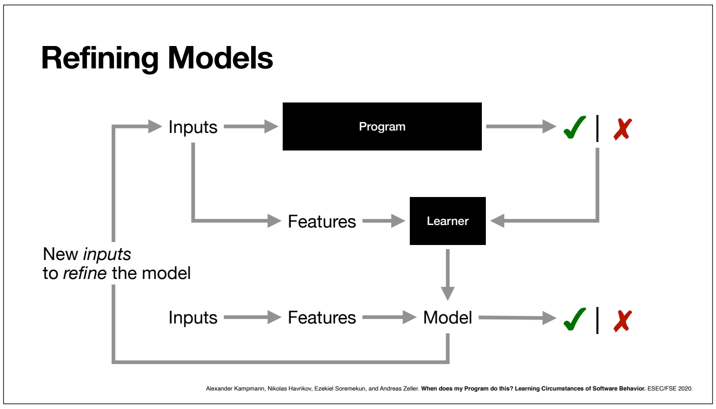
Here is a decision tree that classifies the three inputs on the left. We see that the existence of the digit 8 serves as classifying feature. The model is consistent with all the observations made so far.



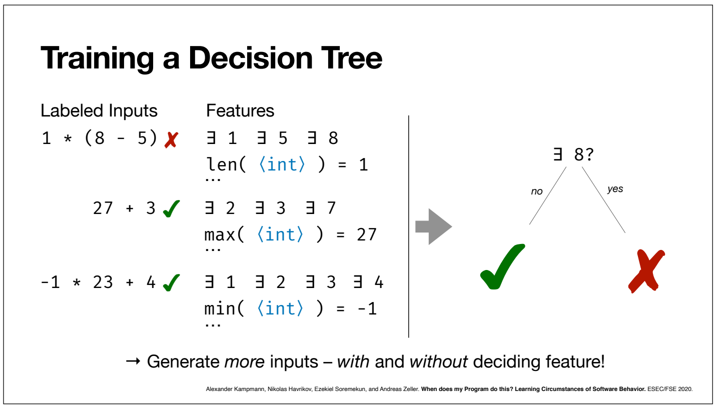
The learner also could come up with another model over the presence or non-presence of multi digit integers. Is any of these correct?



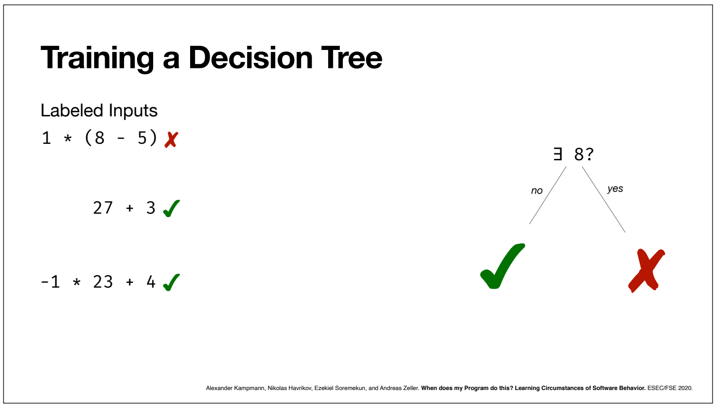
What we need is more inputs and more observations to come up with a more precise model.

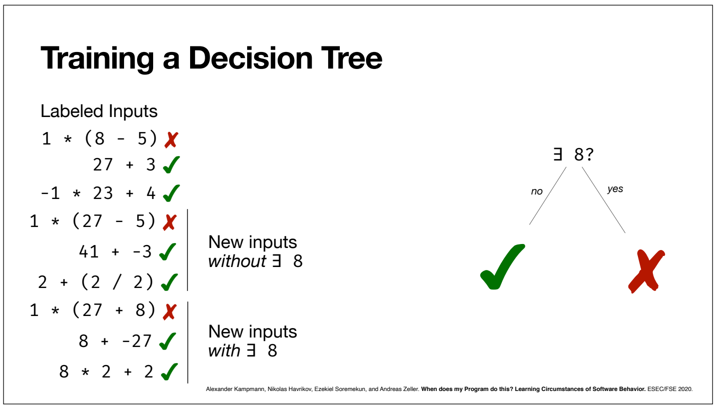


We create **new inputs** right from the model learned so far.

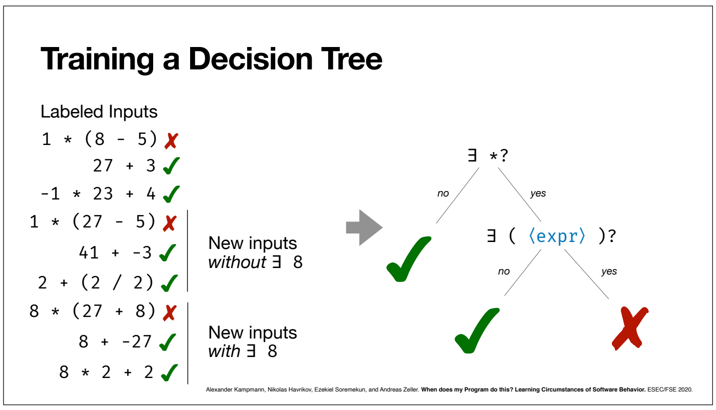


Specifically, for every path in the tree, we generate more inputs.

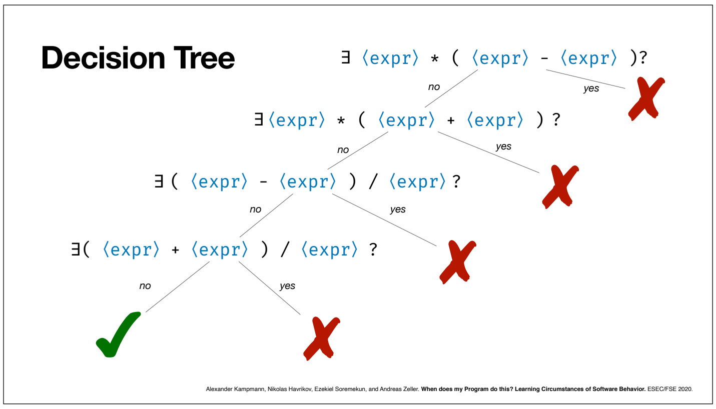




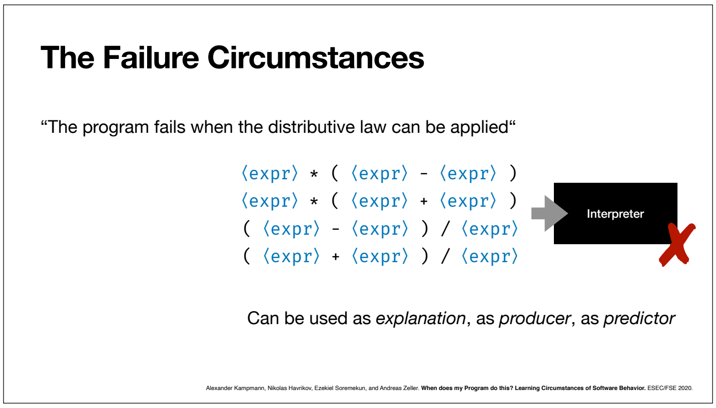
So, here are more inputs with and without the digit 8. For every input, we test whether the failure occurs.



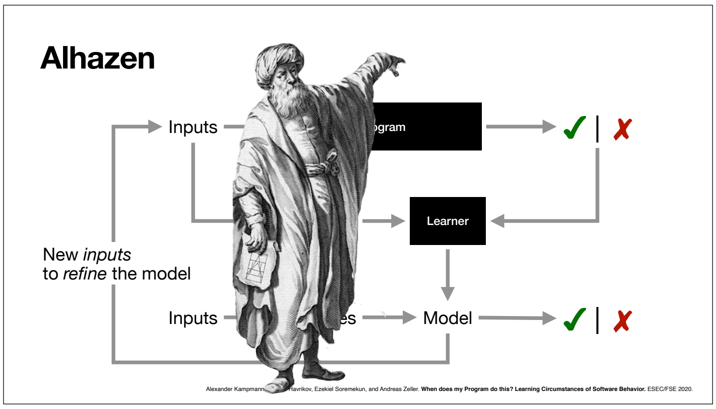
For these inputs, the old hypothesis no longer holds. The decision tree now comes up with a more detailed model.



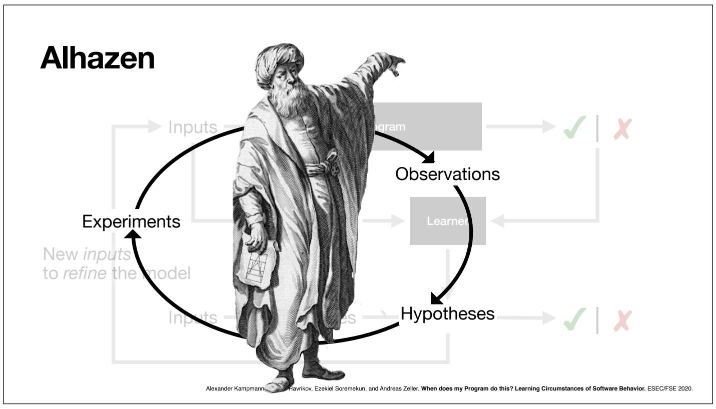
If we repeat this a number of times, we end up with this decision tree which now accurately characterizes the circumstances of failure.



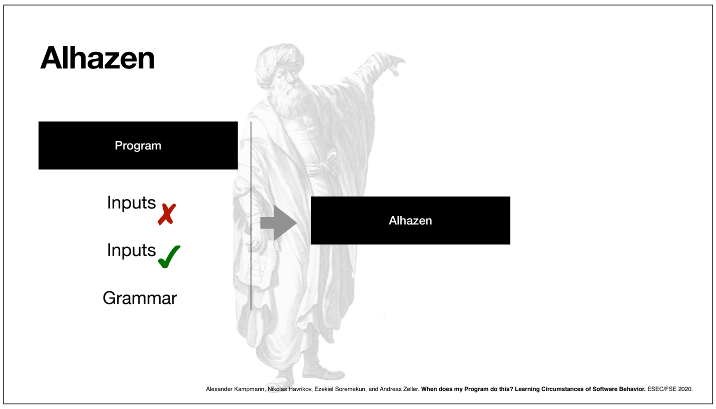
And this now tells us under which circumstance the failure occurs – namely, whenever the distributive law can be applied.



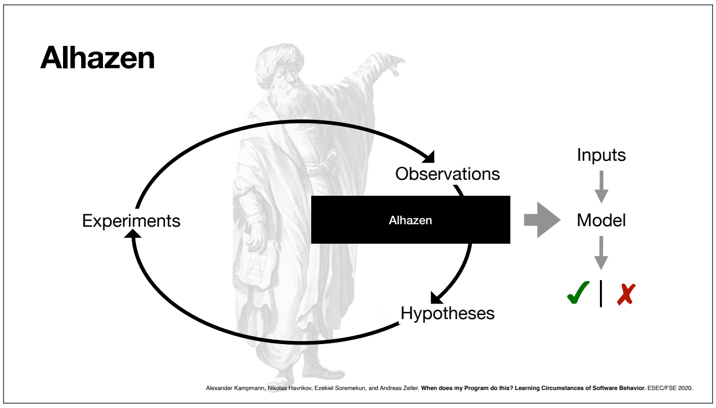
We named our approach **Alhazen**, after Ḥasan Ibn al-Haytham (Latinized as Alhazen /æl'hæzən/; full name Abū ʿAlī al-Ḥasan ibn al-Ḥasan ibn al-Haytham أبو علي، الحسن بن الحسن بن الهيثم; c.965 – c.1040) – an Arab mathematician, astronomer, and physicist of the Islamic Golden Age.



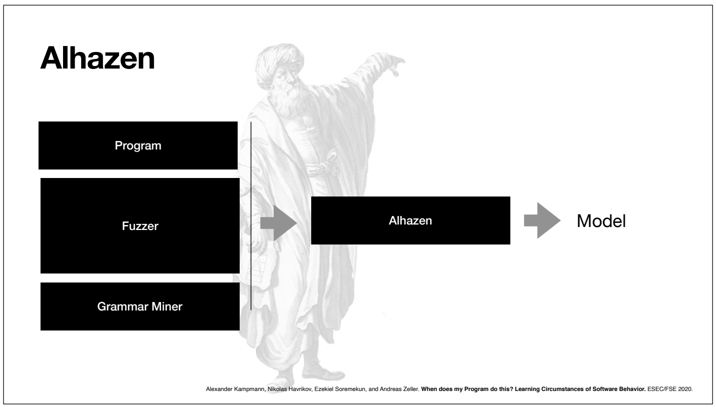
Alhazen was an early proponent of the concept that a hypothesis must be supported by experiments based on confirmable procedures or mathematical evidence—an early pioneer in the scientific method five centuries before Renaissance scientists.



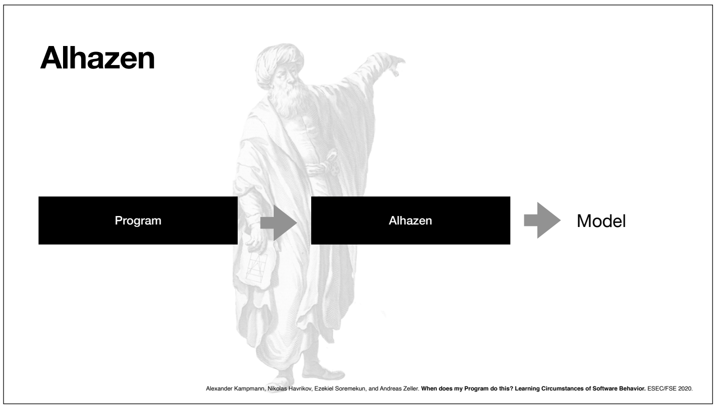
Alhazen takes a program, failing and passing inputs, and a grammar.



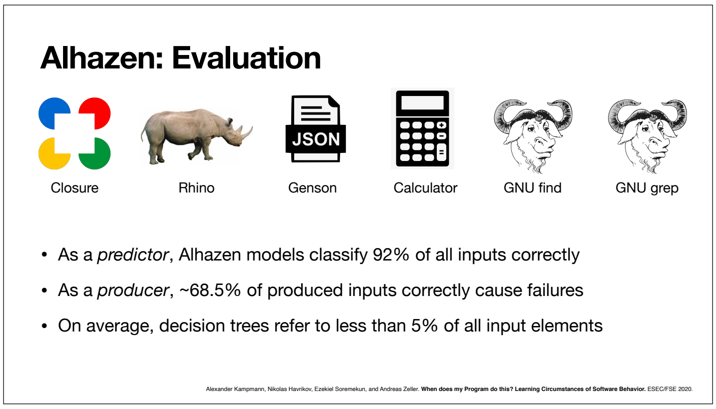
By abstracting over observations, and gradually refining hypothesis through experiments, Alhazen produces a predictive (and generative) model on whether failures occur or not.



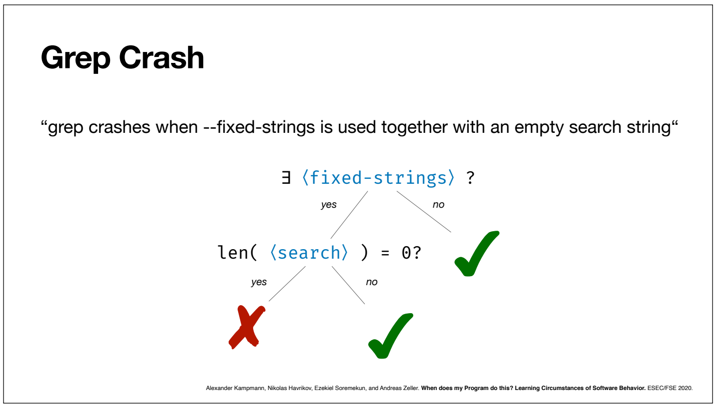
Since the passing and failing inputs can come from a fuzzer, and since the grammar can come from a miner, ...



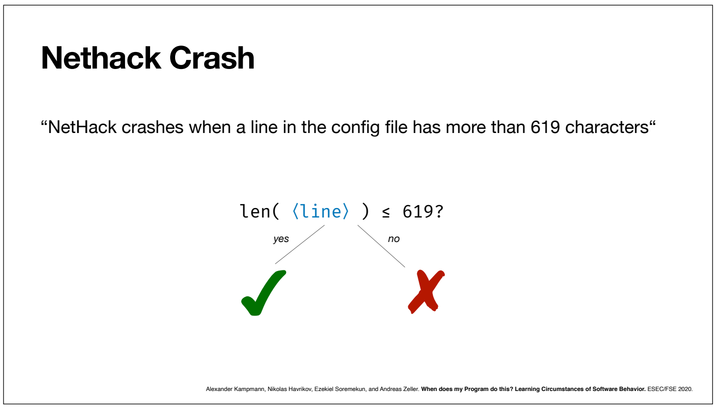
... Alhazen actually only need the program to be debugged to produce a model.



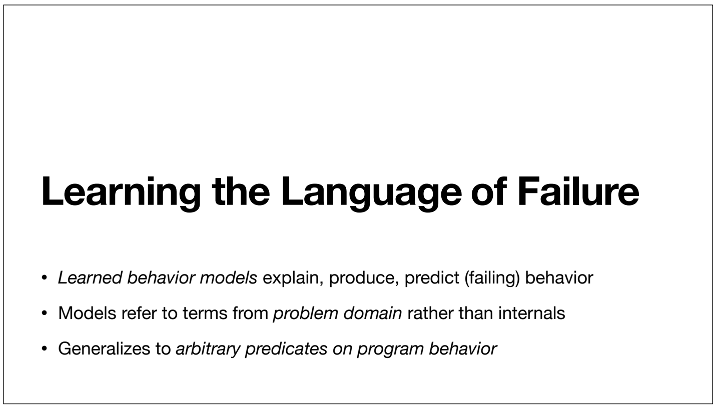
Alhazen works great as a predictor and as a producer. Also, the decision trees refer to a small subset of the input grammar, allowing developers to focus on these.



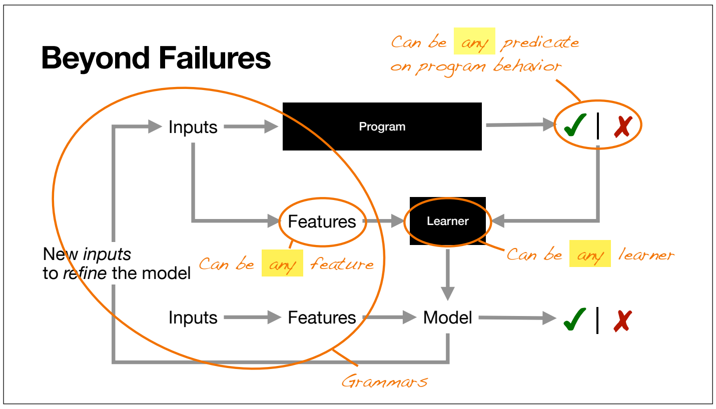
Here is an example. Alhazen correctly determines the circumstances of a grep crash.



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.



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



One exciting thing about our approach is that it can generalize in many ways. For instance, one can use other learners besides decision tree learners.

Learning the Language of Failure

But one can use other predicates too.

Learning the Language of Acceptance

We can learn the set of inputs accepted by a program.

Learning the Language of Coverage

Or the inputs that cover a particular location.

Learning the Language of Data Leaks

The set of inputs that cause a particular unwanted behavior.

Learning the Language of Exploits

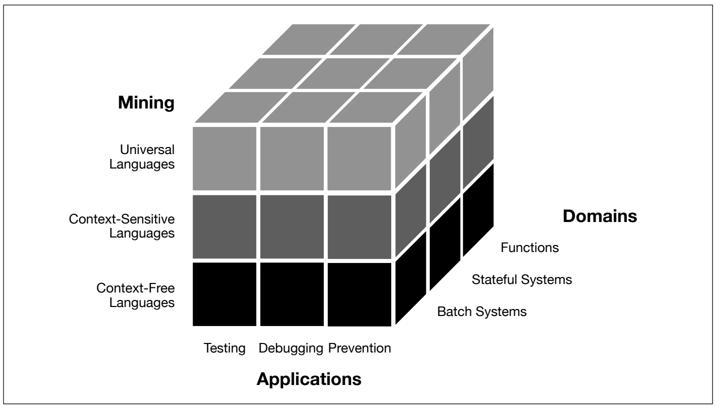
Possibly even the exact language under which an exploit takes place.

Learning the Language of Security

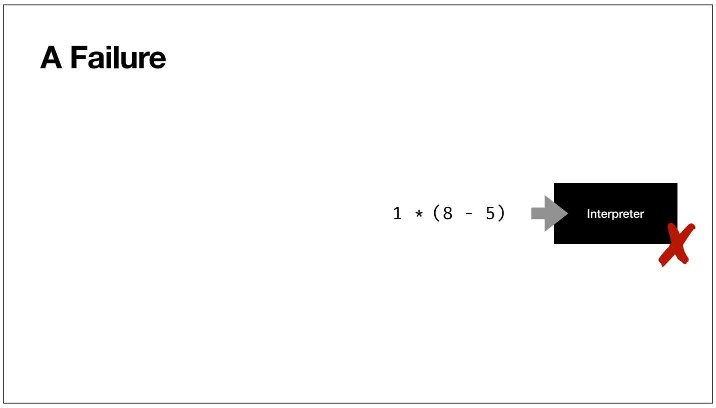
Or, the complement: the language for which nothing bad happens. (There is a lots of future work in that one.)

Perspectives

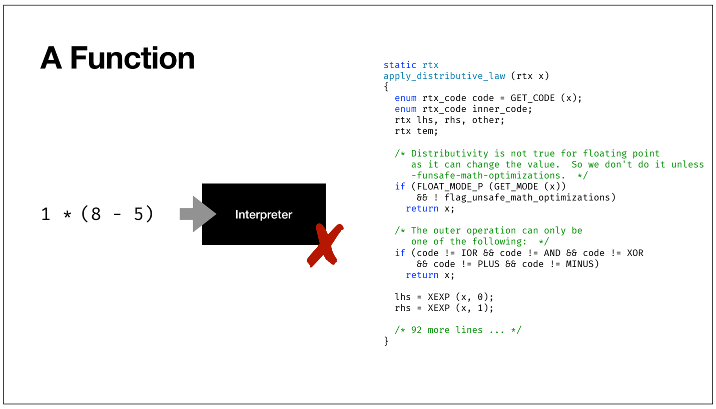
So, where are we going from here?



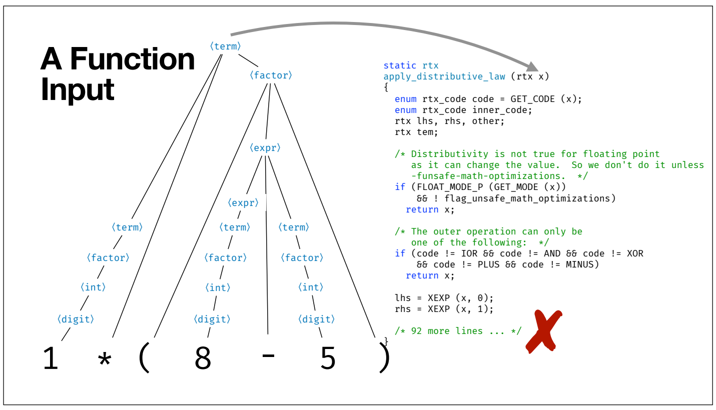
There are many ways languages, applications, and domains, can be combined.



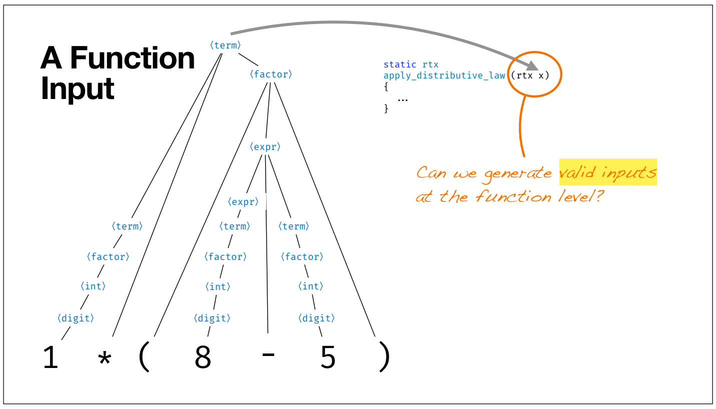
A domain I have not talked about yet is actual **code.**



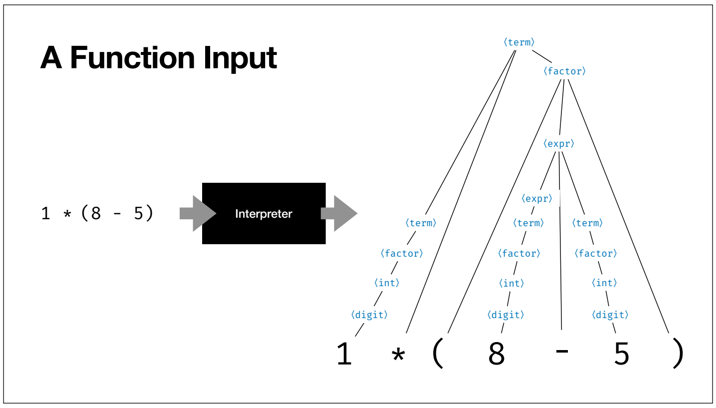
Because somewhere in your program is a piece of code that is faulty and fails. Here we have a function from GCC that once was faulty.



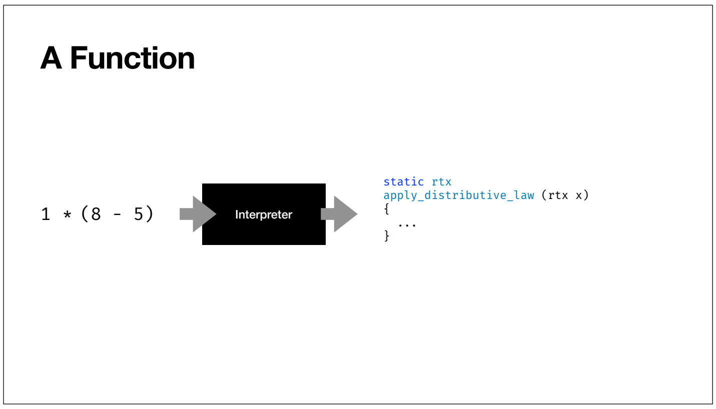
In our case, the input to this function actually is a parse tree.



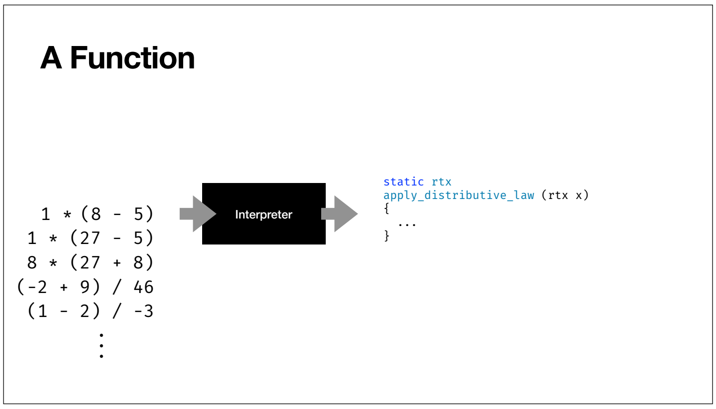
The question is: Can we test apply_distributive_law() by supplying it with valid values for *x*? If you choose a function-level test generator, you will feed plenty of invalid parse trees into this function. This will not test well.



At the system level, however, this is easy, as we can feed valid **system inputs** into our interpreter/compiler.



We can do so for one input...



... and for as many as we like, each one producing a valid value for *x*.

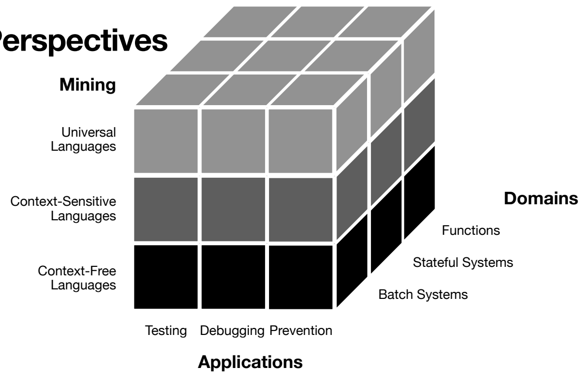
A Function

```
static rtx
apply_distributive_law (rtx x)
{
  ...
}
```

What is the language of x?

But if we want to test the function in isolation, or reason about how it works, we need to know the set of valid values for *x*. This is a type and a language. How can we learn this? This is an open question.

Perspectives



Teamwork

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

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

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.

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

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

Björn Mathis, Rahul Gopinath, Michaël Mera, Alexander Kampmann, and Andreas Zeller. **Parser-Directed Fuzzing.** PLDI 2019.

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

Björn Mathis, Rahul Gopinath, and Andreas Zeller. **Learning Input Tokens for Effective Fuzzing.** ISSTA 2020.

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

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

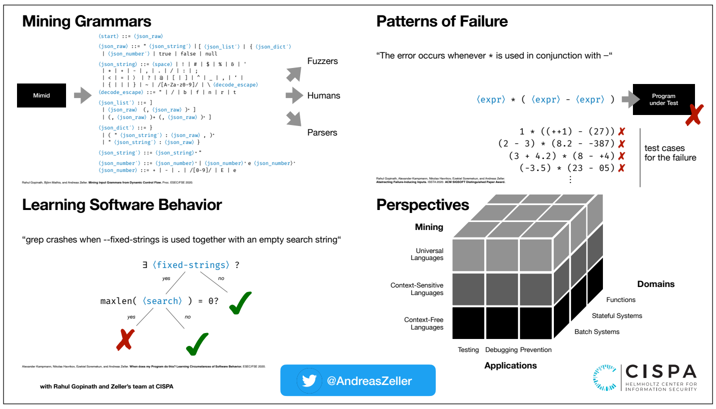
Open science
Open source

Here is a list of all the papers that went into this talk.

Teamwork

Thomas Zimmermann   Dorothea Lütkehaus   Peter Weissgerber   Stephan Diehl   Stephan Neuhaus   Christian Holler   Ralf Hildebrandt
Nikolas Havrikov   Kim Herzig   Rahul Gopinath   Björn Mathis   Michaël Mera   Alexander Kampmann   Ezekiel Soremekun
Nataniel Pereira Borges Junior   Rafael Dutra   Konstantin Kuznetsov   Jenny Rau   Sascha Just   Matthias Höschele   Andreas Rau
Clemens Hammacher   Kevin Streit   Konrad Jamrozik   Alessio Gambi   Vitalii Avdiienko   María Gómez Lacruz   Alessandra Gorla
Sudipta Chattopadhyay   Andrey Tarasevich   Juan Pablo Galeotti   Gordon Fraser   Ilaria Tavecchia   Florian Gross   Eva May   Marcel Böhme
Valentin Dallmeier   Bernd Pohl   Michael Mirold   Christian Lindig   Silvia Breu†   Stephan Neuhaus   Martin Burger   Frank Padberg
Rahul Premraj   Yana Mileva   Matthias Schur   David Schuler   Jeremias Rößler   Andrzej Wasylkowski   Irina Brudaru   Holger Cleve

And these are my all of my students and post-docs over the years. My work became possible only through them. Thank you!



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!