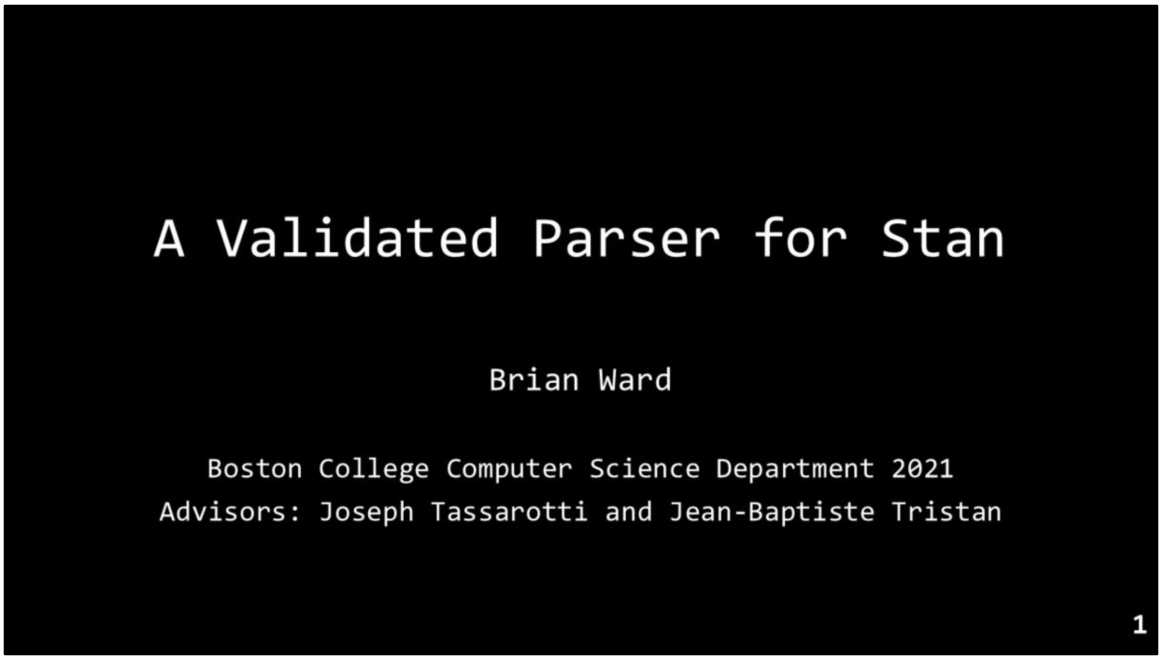
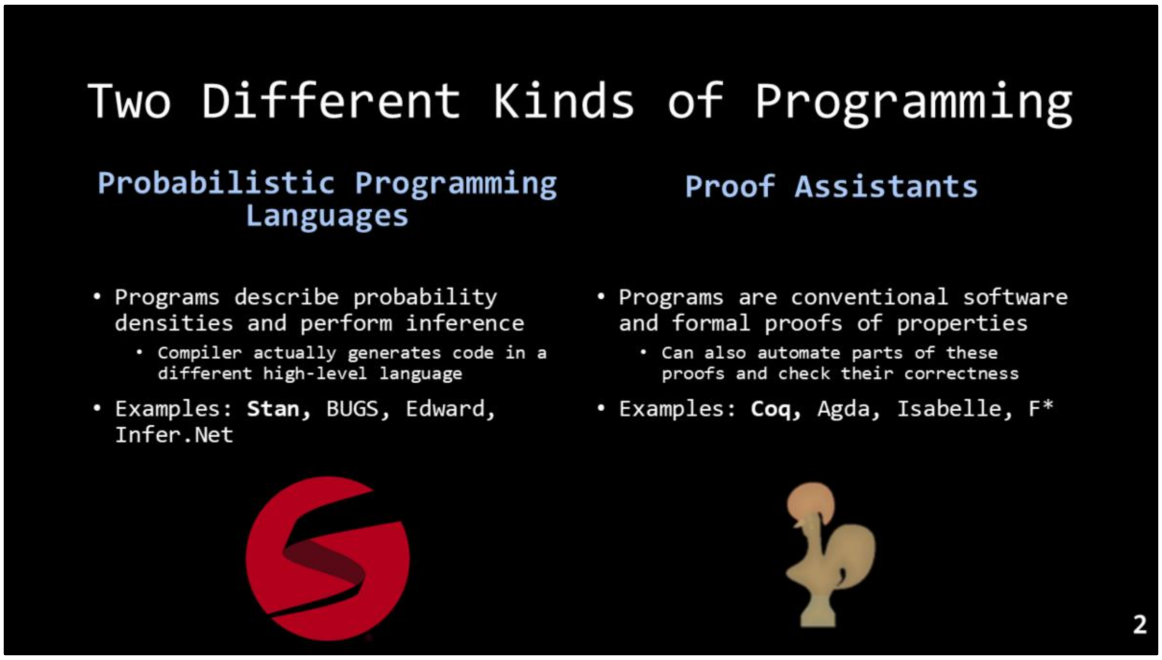


A Validated Parser for Stan

Brian Ward

Boston College Computer Science Department 2021
Advisors: Joseph Tassarotti and Jean-Baptiste Tristan

Hi all
For my senior thesis I've been working with Joseph Tassarotti and Jean-Baptiste Tristan to create a verified parser for the language Stan



Two Different Kinds of Programming

Probabilistic Programming Languages

- Programs describe probability densities and perform inference
  - Compiler actually generates code in a different high-level language
- Examples: **Stan**, BUGS, Edward, Infer.Net

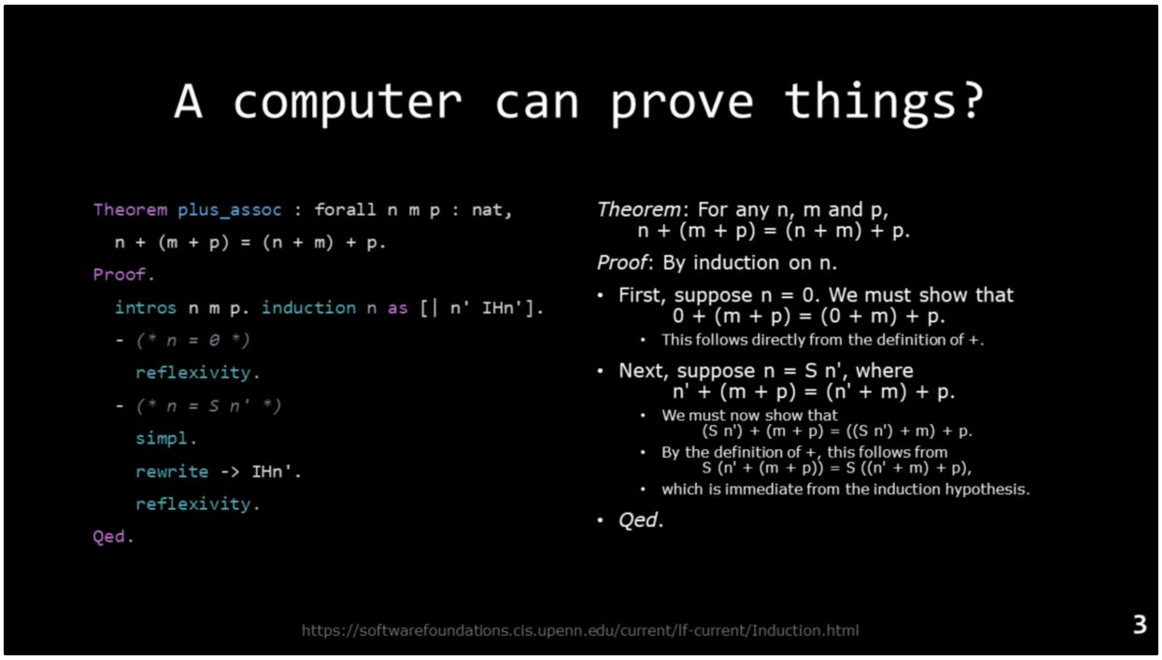
Proof Assistants

- Programs are conventional software and formal proofs of properties
  - Can also automate parts of these proofs and check their correctness
- Examples: **Coq**, Agda, Isabelle, F*

2

Now, I think the most interesting thing about this project was that it required understanding two kinds of programming language that you may not encounter in undergraduate studies.
The first is probabilistic programming languages, of which our target language Stan is one. These are very different from your standard language – the program is not a list of instructions, but a description of a probability density function. When you compile and run these programs, they can perform inference and provide samples from that distribution.

The other kind is known as a proof assistant. This is where the 'validated' part of my title comes in. These are languages, usually functional ones, that can also contain mathematical proofs about the code. These proofs can be checked by the computer, and the computer can aid in their proving alongside the programmer. The proper use of these languages means you actually don't need to do any testing – you have proved, in a truly strict sense, that the program does what you said


A computer can prove things?

```
Theorem plus_assoc : forall n m p : nat,
  n + (m + p) = (n + m) + p.
Proof.
  intros n m p. induction n as [| n' IHn'].
  - (* n = 0 *)
    reflexivity.
  - (* n = S n' *)
    simpl.
    rewrite -> IHn'.
    reflexivity.
Qed.
```

Theorem: For any n, m and p,
n + (m + p) = (n + m) + p.

Proof: By induction on n.

- First, suppose n = 0. We must show that
  0 + (m + p) = (0 + m) + p.
  - This follows directly from the definition of +.
- Next, suppose n = S n', where
  n' + (m + p) = (n' + m) + p.
  - We must now show that
    (S n') + (m + p) = ((S n') + m) + p.
  - By the definition of +, this follows from
    S (n' + (m + p)) = S ((n' + m) + p),
  - which is immediate from the induction hypothesis.
- *Qed*.

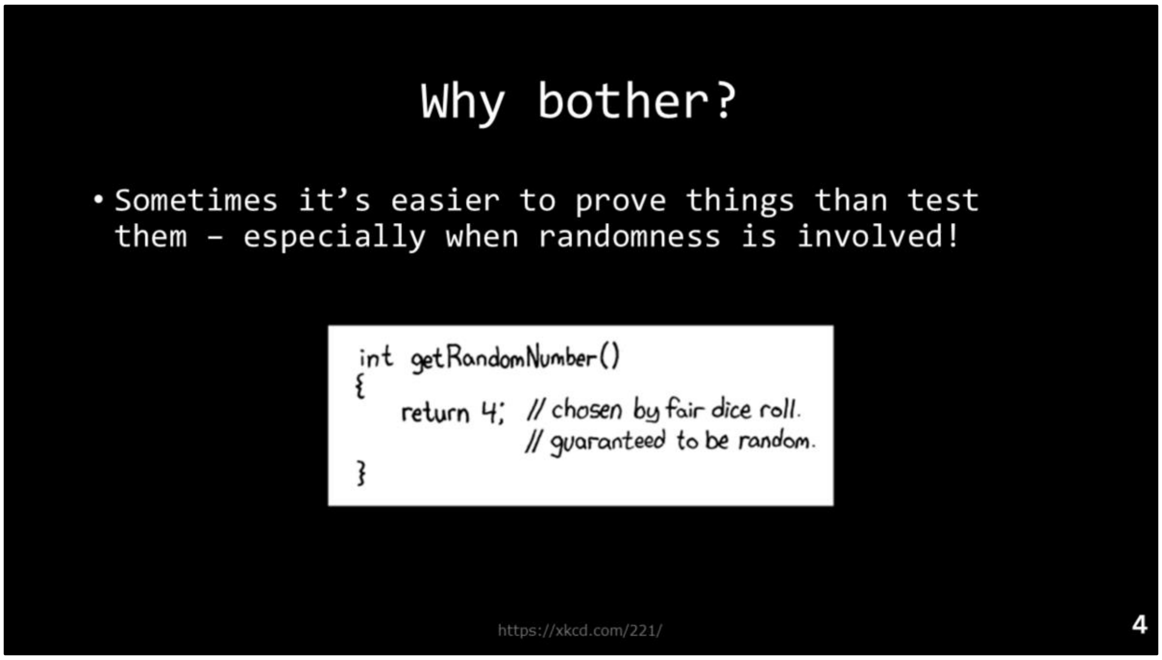
https://softwarefoundations.cis.upenn.edu/current/lf-current/Induction.html

3

Now, you may have the same instinct I did when I first heard about these proof assistants. How does that work? Is it really possible?

The short answer is yes. Here is an example of a proof of the same thing in Coq, the language my thesis used, and in a more standard English proof.

You don’t need to understand everything happening on the screen here, but you should take away this: computers can do proofs, but they specialize in simple proofs. And as it turns out, that is good enough for a lot of things. Proofs about a program tend to be intellectually pretty simple, but take a ton of tiny little steps. These can be automated away using the more advanced features of the proof assistants.



Why bother?

- Sometimes it's easier to prove things than test them – especially when randomness is involved!

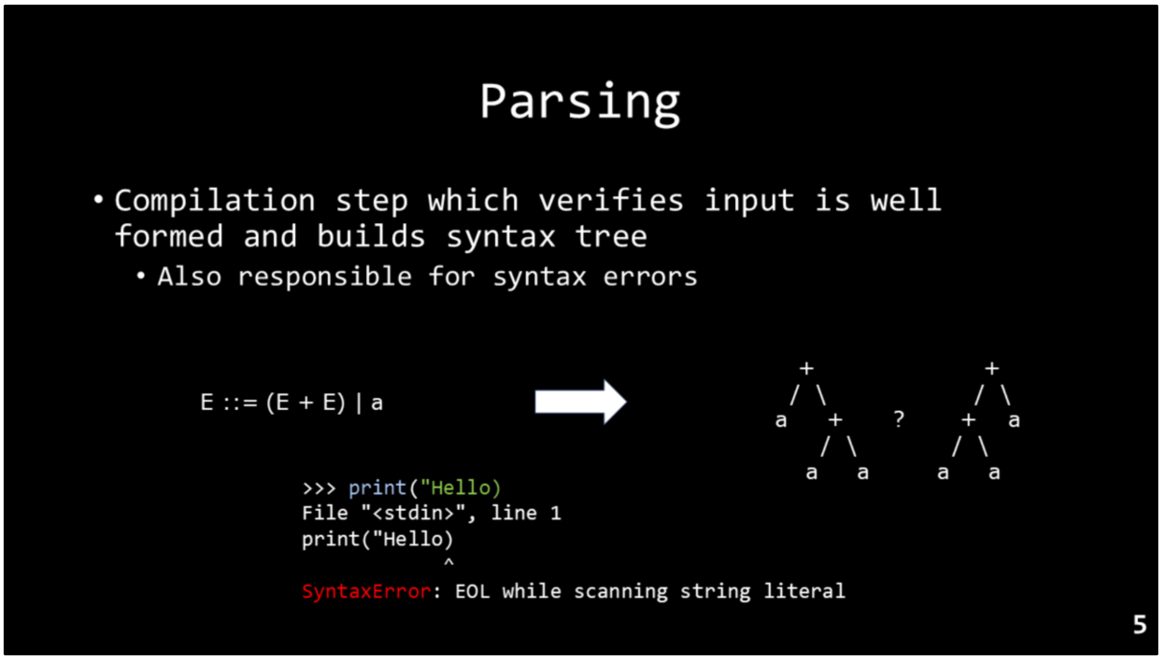
```
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
               // guaranteed to be random.
}
```

https://xkcd.com/221/

4

If I've been able to convince you that you really can prove these things, the next question may be "why, then"?

This is where the nature of Stan can be more important. My thesis is the first step in a larger project to verify an entire compiler for Stan. And while this compiler itself is deterministic, the program it outputs does simulate randomness, and proving that it is correct can be difficult. There have been documented examples of the existing Stan compiler outputting subtly biased numbers, and this can take a long time to notice. Proving that the compiler's transformations were correct, we believe, is an worthwhile endeavor to combat this potential with complete confidence.



Parsing

- Compilation step which verifies input is well formed and builds syntax tree
  - Also responsible for syntax errors

E ::= (E + E) | a

```
    +              +
   / \     ?      / \
  a   +          +   a
     / \        / \
    a   a      a   a
```

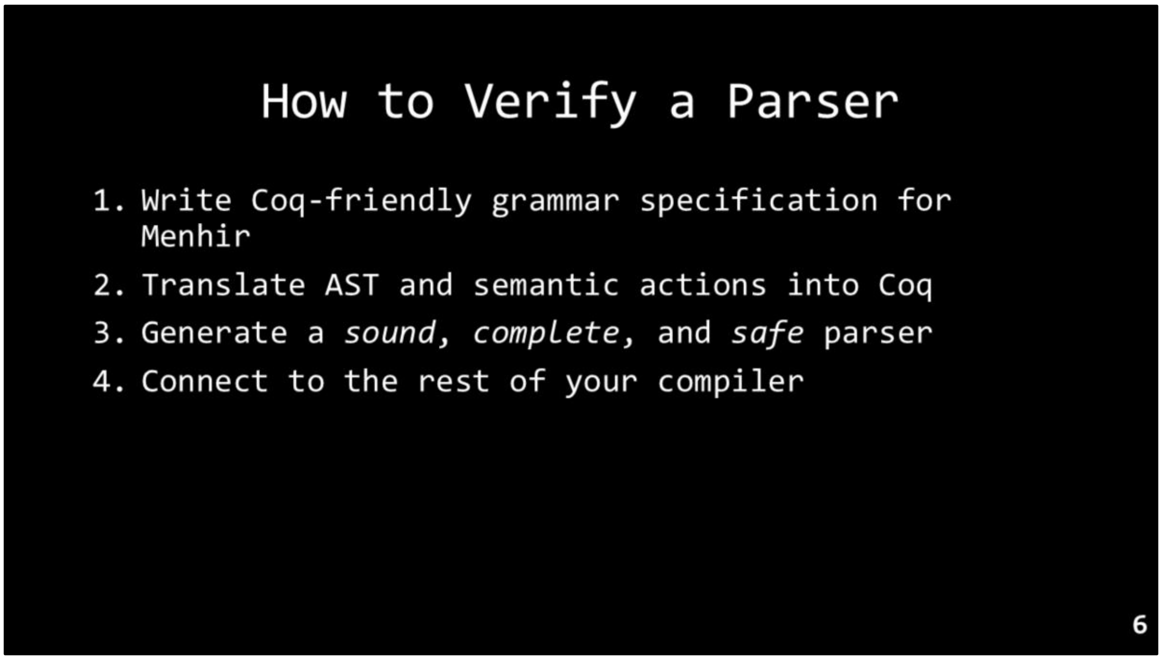
```
>>> print("Hello)
File "<stdin>", line 1
print("Hello)
             ^
SyntaxError: EOL while scanning string literal
```

5

All of that (hopefully interesting) background on programming languages aside, we can start to discuss what my thesis actually did.

Parsing is the first step a compiler does. This takes in the input as a sequence of characters and verifies its structure. If it accepts the code, meaning it was well-formed, it will build a tree called an Abstract Syntax Tree that is used later in the compiler. If it rejects the input, a good parser should tell you why – this is where the oft-maligned syntax error comes from.

There are a lot of extra details in the theory of parsing, especially wrapped up the the phrase “builds a syntax tree”. In particular, there is something known as ambiguity, which can lead to a situation like in the picture above – which of these two trees do we want the parser to output? This basic idea ended up being essential to my work on the parser.



How to Verify a Parser

1. Write Coq-friendly grammar specification for Menhir
2. Translate AST and semantic actions into Coq
3. Generate a *sound*, *complete*, and *safe* parser
4. Connect to the rest of your compiler

6

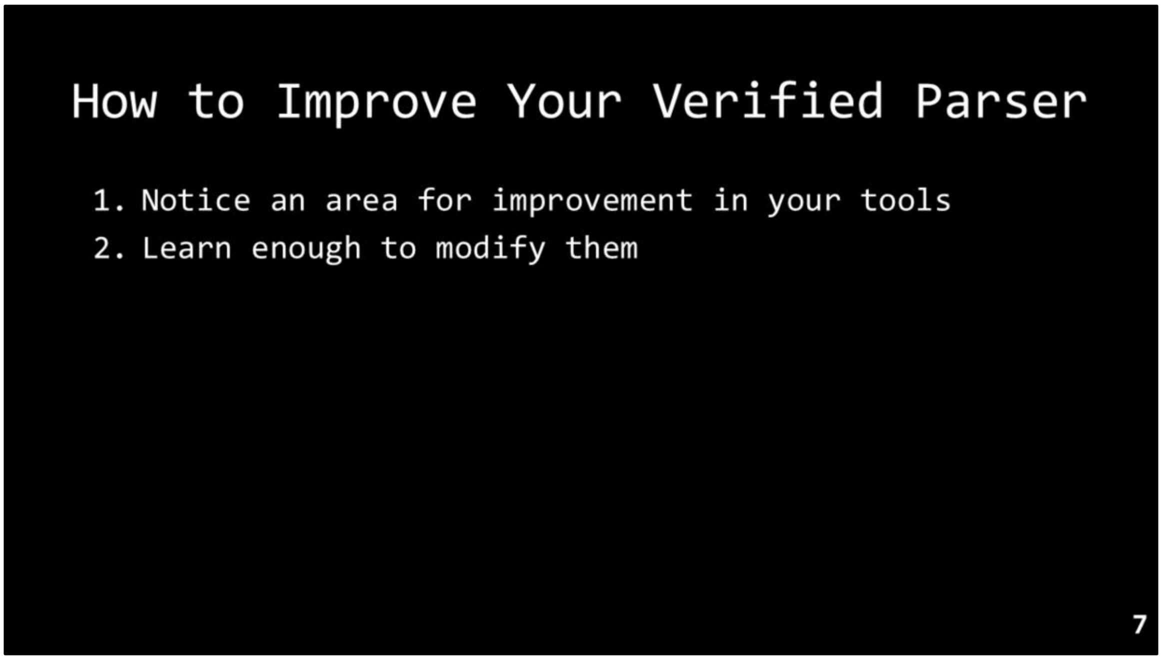
So let's get down to it then. We used a tool known as a parser generator, which takes in a specification of a grammar for the language you want to parse, and outputs code that implements that parser. Specifically we used Menhir, which can output code for Ocaml or Coq

A lot of this was just about reading the Stan specification and using the existing compiler as a template, but there were some important details that needed to be cleared up. The specification was incomplete, and the existing compiler resolved ambiguity (from the last slide) using annotations to tell the parser what to do. The Coq mode of Menhir, so I had to re-write a large section of the grammar myself.

Then, all the code the parser needs to work needs to be written in Coq. This also involves describing the type structure of the language to Menhir.

After that, Menhir handles the rest. You can be sure that the parser it generates only recognizes things in the language, recognizes all of them, and doesn't produce any internal errors while parsing. They have the proofs to show it.

Finally, there is some work to make this actually usable by the rest of the ecosystem.



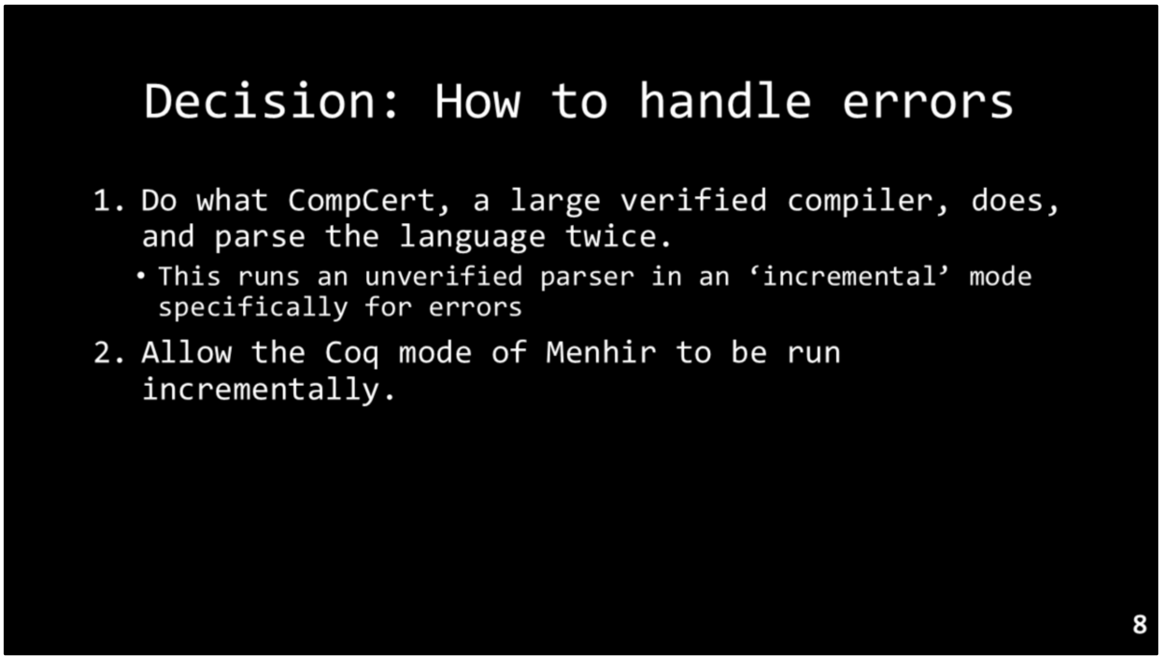
How to Improve Your Verified Parser

1. Notice an area for improvement in your tools
2. Learn enough to modify them

7

The other thing I mentioned that parsers are responsible for is syntax errors. This is an incredibly useful part of parsing, and we decided it was a necessary part of my project.
But there was a problem: Menhir doesn't have any good options for doing this in the Coq mode.

So, we decided I should add one. We thought of a couple ways to do this.



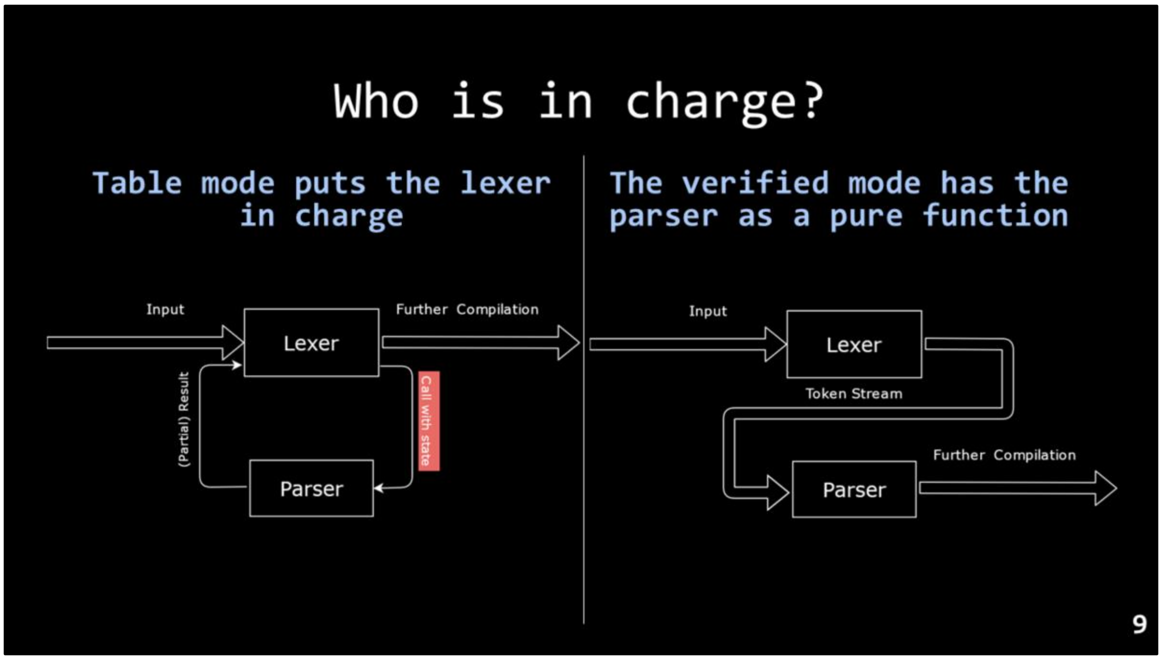
Decision: How to handle errors

1. Do what CompCert, a large verified compiler, does, and parse the language twice.
   - This runs an unverified parser in an ‘incremental’ mode specifically for errors
2. Allow the Coq mode of Menhir to be run incrementally.

8

First, we could do what’s been done before. CompCert is a verified compiler, written in Coq, for the C language. It is actually the reason Menhir can output Coq code to begin with. They run two parsers, one of which is not verified and handles the errors, and the other which actually creates the tree. There are some oddities in C that also make this a good idea for them, but we decided against it. It would have been really easy, but it struck us as inelegant.

The second thing we considered was changing the Coq parser that Menhir creates to use that same incremental mode which allows the unverified parsers to do error messaging so well. This had two large problems: first, the size of the change would be considerable, taking a lot of effort and breaking backwards compatibility considerably. Secondly, there is a more subtle problem with this feature.

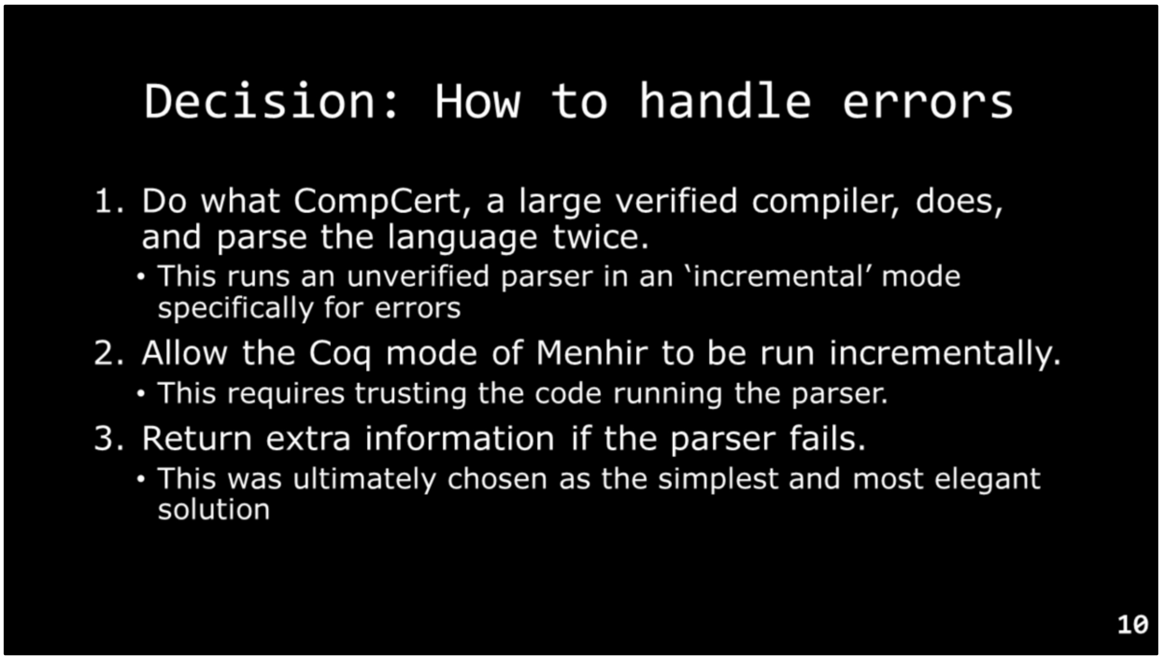


Who is in charge?

Table mode puts the lexer in charge

The verified mode has the parser as a pure function

This change would have the parser be run step by step, and some other piece of code would be in charge of actually running each of those steps. This is counter to the idea of verification, because you'd have to trust the code running the parser to do the right things. The proofs about the parser would be mostly meaningless.

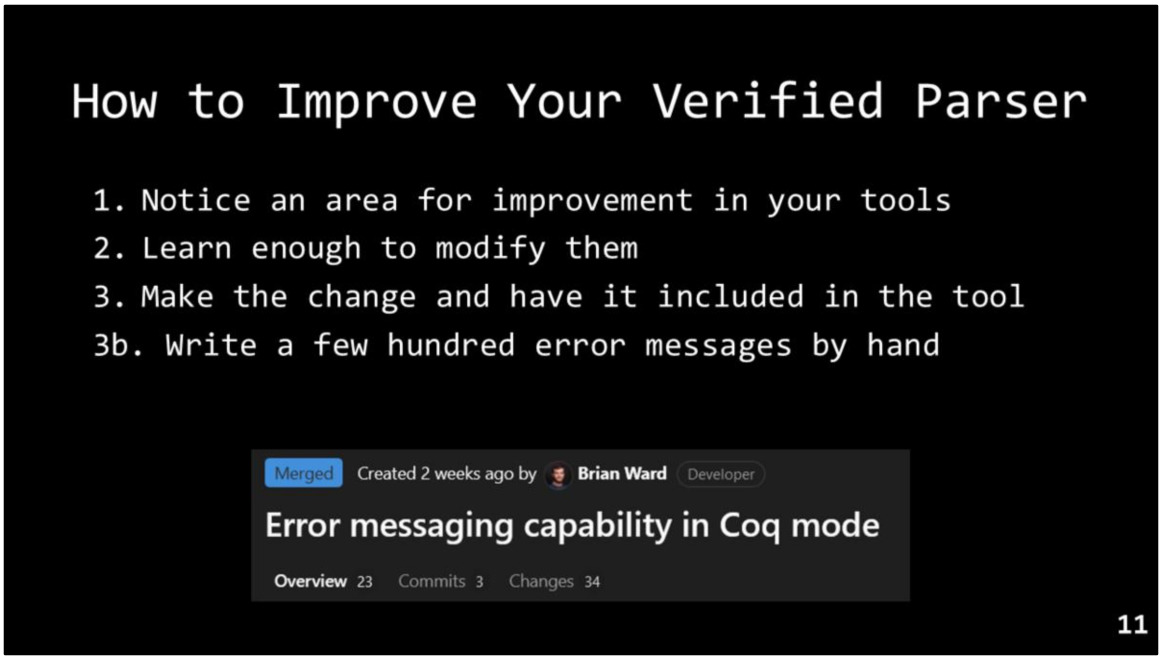


Decision: How to handle errors

1. Do what CompCert, a large verified compiler, does, and parse the language twice.
   - This runs an unverified parser in an 'incremental' mode specifically for errors
2. Allow the Coq mode of Menhir to be run incrementally.
   - This requires trusting the code running the parser.
3. Return extra information if the parser fails.
   - This was ultimately chosen as the simplest and most elegant solution

10

So, we came to the third option. We can modify the return type of the generated parsers to include some crucial information about the state of the parser and the piece of the input that caused this error.
We chose to pursue this.



How to Improve Your Verified Parser

1. Notice an area for improvement in your tools
2. Learn enough to modify them
3. Make the change and have it included in the tool

3b. Write a few hundred error messages by hand

Merged Created 2 weeks ago by Brian Ward Developer

Error messaging capability in Coq mode

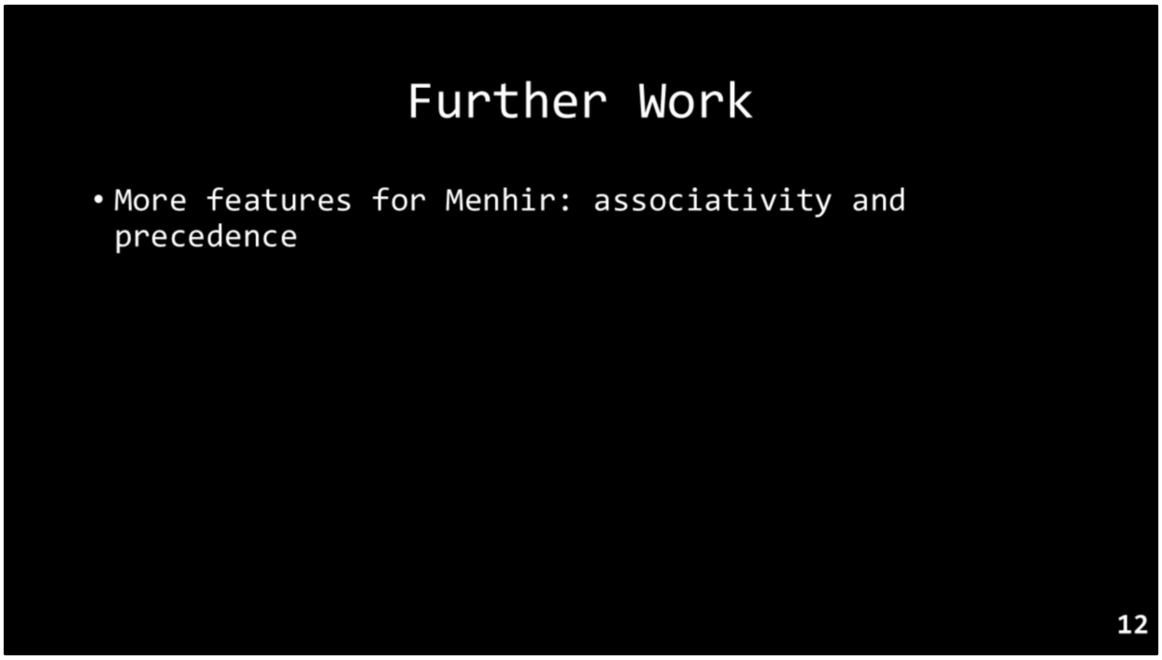
Overview 23 Commits 3 Changes 34

11

So, understanding the options and having made our choice, I did it. And when we reached out to the developers of Menhir, they included it in the tool.

There was just one thing left to do: Now that Menhir’s Coq mode supported error messages, I needed to write them. Using the existing stan compiler as a reference, I wrote about a 160 error messages, for just over two hundred possible syntax problems, with some overlap between them.

Putting all that together, we got what we wanted



Further Work

- More features for Menhir: associativity and precedence

12

A verified parser that also gives meaningful information to the programmer. This parser will be used as the first part of the larger verified Stan compiler project.

Thank you!

13