# Generating random Rust programs

David Pikas

April 7, 2021

## Background

Rust is a low level programming language with a focus on safety and performance. Rust utilizes a concept called ownership which allows the compiler to statically determine when to allocate and free memory and additionally guarantees that no data races occurs.[1]

Like all software, the Rust compiler is susceptible to bugs. Compiler bugs can easily cause bugs downstream that are hard to debug, something that can be extra dangerous if the compiler in question is for a language that places a lot of emphasis on security and correctness like Rust.

Finding bugs is commonly done through testing. When testing a compiler, one approach is to generate random programs and then either check that the compiler behaves as expected (e.g. it successfully compiles semantically valid programs and rejects semantically invalid programs) or check that different compiler implementations produce semantically equivalent code.

## Project description

The goal of this project is to create a program capable of randomly generating semantically valid Rust code that makes use of Rust's ownership semantics in order to test the Rust compiler *rustc*.

There exist other similar projects such as Csmith[2] which generates random C programs for the purpose of testing C compilers. Generating Rust code differs from generating C code in a couple of key ways, however. It is much easier to avoid undefined behaviour in Rust but on the other hand the programs must conform to Rust's ownership semantics in order for them to be correct. There are also several different C compilers that can be compared against each other, but there is just one Rust compiler meaning that bugs can't be found by differential testing.

## Approach

The strategy for generating Rust code is to first create an abstract syntax tree (AST) representing the program and then translate it into a textual represen-

tation. While generating the AST a record containing information about the already generated code will be kept so that the program can avoid e.g. referencing variables that haven't been declared or other semantic errors.

There are various projects related to generating code in other languages such as the aforementioned Csmith. Part of the project would involve exploring these different projects as well as searching for relevant literature.

## Relevant courses

Of the courses I've taken, the following are especially relevant to this project:

- Compiler Design 1
- Semantics of Computer Languages
- Programming Theory

## Delimitations

While the intended use of the randomly generated code is to test the compiler, the project doesn't necessarily have to find any bugs in the compiler in order to be successful. The project doesn't attempt to generate code from all parts the Rust language either, particularly not "unsafe" Rust.

There won't be any attempt to keep track of what the code is expected to evaluate to, that it terminates or any other similar qualities beyond whether the code is semantically valid.

## Time plan

| | |
|---|---|
| Week 1–2 | Research similar projects, relevant literature and the Rust language itself. |
| Week 2–3 | Generate basic syntactically valid Rust code. |
| Week 4–5 | Generate basic semantically valid Rust code. |
| Week 6 | Create more advanced infrastructure around testing the compiler. |
| Week 6–7 | Expand the amount of code that can be generated. |
| Week 8–9 | Create strategies for generating more "interesting" code that is more likely to cause errors. |
| Week 9–10 | Write the report. |

Expanding the amount of code that is generated and creating strategies for generating interesting code are both optional parts of the project that will only be done if the rest of the work goes as planned.

# References

[1] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. 2019. ISBN: 9781718500440.

[2] Xuejun Yang et al. "Finding and understanding bugs in C compilers". In: *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 2011, pp. 283–294.