# Should we have a 2021 Edition?

Steve Klabnik• 07.27.2020

I recently joined Oxide Computer Company!

We are writing all kinds of things in Rust, from the firmware up. And we're hiring! https://oxide.computer/careers/

Embedded Rust is awesome :)


I've started streaming my open source Rust development!
https://www.twitch.tv/steveklabnik

# Overview

**What are Editions?**
Let's get on the same page

**A look back at Rust 2018**
Half case study, half retrospective

**What should we do?**
Should we have Rust 2021?

# What are editions?

# What are editions? (socially)

## A way reflect on longer-term progress

Every six weeks is great, but too fast in some ways

## A way to entice new users

Not everyone can pay such close attention to Rust

## A nice “rallying cry”

Get people excited about things

# What are editions? (technically)

## Editions are a way to make breaking changes...

… without actually making breaking changes.

## New editions are opt-in

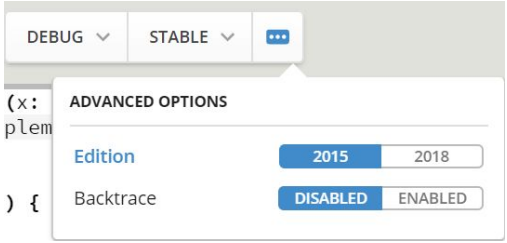
So your old code continues to work just fine.

## Editions can’t change everything

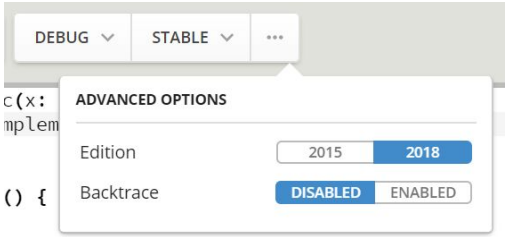
This is useful for both humans and compilers.

# Editions can make breaking changes

```
1 fn async(x: i32) -> i32 {
2     x
3 }
4
5 fn main() {
6     let _s = async(5);
7 }
```




```
   Compiling playground v0.0.1 (/playground)
    Finished dev [unoptimized + debuginfo] target(s) in 0.69s
     Running `target/debug/playground`
```




```
   Compiling playground v0.0.1 (/playground)
error: expected identifier, found keyword `async`
 --> src/main.rs:1:4
  |
1 | fn async(x: i32) -> i32 {
  |    ^^^^^ expected identifier, found keyword
  |
```

# This is usually set in Cargo.toml

```
1   [package]
2   authors = ["Steve Klabnik <steve@steveklabnik.com>"]
3   edition = "2018"
4   readme = "README.md"
```

# Editions can interoperate

```
help: you can escape reserved keywords to use them as identifiers
  |
1 | fn r#async(x: i32) -> i32 {
  |    ^^^^^^^
```

```
1  fn r#async(x: i32) -> i32 {
2      x
3  }
4
5  fn main() {
6      let _s = r#async(5);
7  }
8
```

# Editions can't change everything

## Can change:

- New keywords
- Repurpose syntax
  - Deprecate `Trait`
  - Introduce `dyn Trait`

## Cannot change:

- Coherence rules
- Breaking changes to the standard library
  - One stdlib for your whole program

# “Multi-pass” compiler

Instead of one pass, `rustc` does many passes over your code in order to create an executable

Input: Source code

- AST (“Abstract Syntax Tree”)
- HIR
- MIR
- LLVM-IR

Output: final binary

# The three phases of compilers

## Lexical/Syntax analysis

Is your code well-formed?

## Semantic analysis

Does your code make sense?

## Code generation

Once everything is verified, produce output

# Two kinds of steps

## Lowering

- These go from one form to another. “The MIR is then lowered to LLVM-IR.”
- At each form, things get simpler. We throw away stuff we’ve already validated, making future steps easier.

## Pass

- A compiler **check** validates some aspect of your program, e.g. type checking
- A compiler **pass** does some form of transformation, e.g. an optimization

# Building an Abstract Syntax Tree

## Take some source code, produce AST

For example,

```
pub fn plus_one(x: i32) -> i32 { x + 1 }
```

This gives us:

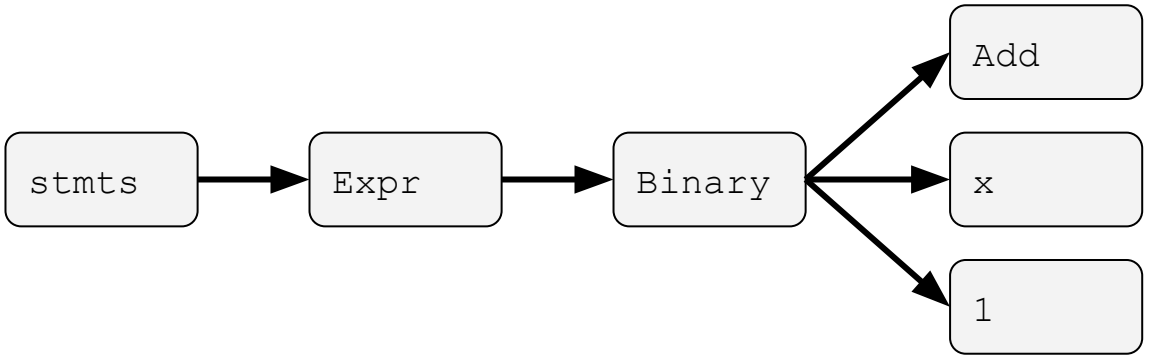
```
"stmts": [{"node": {"variant": "Expr","fields": [{"node": { "variant": "Binary",

    "fields": [{"node": "Add"},

        {"node": {"variant": "Path", "fields": ["segments": [{"ident": "x",
```

# Building an Abstract Syntax Tree

Code: `x + 1`



# Building an Abstract Syntax Tree

**Fundamentally, an AST is a data-structure representation of our code, as written.**

# From AST to HIR

## High level intermediate representation

Is your code well-formed?

## Some things are simplified

Does your code make sense?

# From AST to HIR

```
let values = vec![1, 2, 3, 4, 5];

for x in values {
    println!("{}", x);
}
```

---

```
let values = vec![1, 2, 3, 4, 5];
{
    let result = match IntoIterator::into_iter(values) {
        mut iter => loop {
            let next;
            match iter.next() {
                Some(val) => next = val,
                None => break,
            };
            let x = next;
            let () = { println!("{}", x); };
        },
    };
    result
}
```

# From AST to HIR

## Most checks are done on HIR

HIR was the original Rust IR

## Type checking

Do all the types work?

## Method lookup

Done at compile time

# From HIR to MIR

## MIR is about control flow

How does execution work in your program?

## The core of Rust

Everything superfluous has been removed

## Borrow checking is done here

Are all of your references okay?

# From HIR to MIR

```
fn add_one(_1: i32) -> i32{
    let mut _0: i32;
    let mut _2: i32;

    bb0: {
        StorageLive(_2);
        _2 = _1;
        _0 = Add(move _2, const 1i32);
        StorageDead(_2);
        return;
    }
}
```

# From MIR to LLVM-IR

## LLVM is an awesome compiler toolkit

It's not really a VM

## LLVM does optimizations and code generation

We are deeply indebted to their work

## This is the lowest level

At least until the binary code is generated

# From MIR to LLVM-IR

```
; add_one
; Function Attrs: norecurse nounwind readnone uwtable

define i32 @_ZN10add_one17h3c(i32 %x) unnamed_addr #0 {
start:
  %0 = add i32 %x, 1
  ret i32 %0
}
```

# “Query-based” compiler

Instead of passes, `rustc` calls “queries” in order to create an executable

In this model, the compiler asks questions (“what type is this function?”), and figures out the answers.

Answers are memoized.

This fits incremental compilation far better.

Eventually the whole compiler will be like this, but for now, only parts of it are.

# What does this mean, though?

## For compilers

- Basically, editions are not allowed to differ in MIR
- MIR becomes a common language for all editions
- This helps make interop between editions and overall maintenance easier

## For humans

- Things can change per-edition, but not *that* much
- The ecosystem doesn't split like it would with a major version change

# A sort of “supra-release cycle”

## Rust has three release cycles

Stable, beta, nightly

## There are different cadences

Nightly: every night, Stable + beta: every six weeks

## Editions don’t have a cadence...

… yet. That’s why this talk exists. And an RFC.

# When are editions used?

## No policy, strictly speaking

The RFC did not address this question

## We did it once before, in 2018

And that retroactively made 2015 the first one

## How did that go?

Let’s go over what happened in 2018

# A look back at Rust 2018

# Rust 2018 was a success

## Achieved our goals

- An edition did get shipped
- People understood that this was different than a Rust 2.0
- No major issues at this point

## It was a lot of work

- A really big project for the various teams
- We have never undertaken something so big, and we shipped it!

# … but not a complete success

## The schedule

- We didn't ship everything
- Some things were only halfway shipped
- We barely shipped what we did ship in 2018

## The team

- Lots of people working very hard for far too long
- Extremely high stakes
- A lot of burnout amongst contributors
    - I was a total mess

# Announcing Rust 1.31 and Rust 2018

Dec. 6, 2018 · The Rust Core Team

# We shipped a bunch of stuff!

- Rust 2018
  - Non-lexical lifetimes
  - Module system changes
- More lifetime elision rules
- `const fn`
- New tools
- Tool Lints
- Documentation
- Domain working groups
- New website
- Library stabilizations
- Cargo features
- Contributors

# 2018 Edition was behind schedule

| Rust version | Latest available edition |
|---|---|
| 1.15 | 2015 |
| 1.21 | 2015 |
| 1.23 | 2019 (preview period) |
| 1.25 | 2019 (preview period) |
| 1.27 | 2019 (final) |

# 2018 Edition was behind schedule

| Rust version | Latest available edition | |
|---|---|---|
| 1.15 | 2015 | |
| 1.21 | 2015 | |
| 1.23 | 2019 (preview period) | Jan 4, 2018 |
| 1.25 | 2019 (preview period) | March 29, 2018 |
| 1.27 | 2019 (final) | June 21, 2018 |

# 2018 Edition was behind schedule

| | Planned | Actual |
|---|---|---|
| Available in nightly | Jan 4, 2018 | Feb 6, 2018 |
| Release | June 21, 2018 | Dec 6, 2018 |

# We tried to do too much

## That's partially because there was a lot to do!

Rust 1.0 was really small

## We tried to move in front of the community

Inventing patterns, rather than codifying them

## Contributors != employees

Much, much, much harder to plan big initiatives

# We proved the mechanism works

## 2015 and 2018 interoperate just fine

We have our cake and eat it too

## We did not split the ecosystem

There's not two different camps of users who can't communicate

## It's pretty much silent

I don't know that most users think about editions 99% of the time

# We underestimated the cost

## Upgrading is not simple

Even if it's easy

## It takes a lot of time

There's just a lot of moving pieces

## Biggest users bear the biggest burden

… and production users are our biggest users

# Feature-driven vs time-boxed

## Regular Rust releases are time-boxed

There's a schedule. 🚂🚄🚅🚆🚈

## Rust 2018 was designed around features

There were specific things we needed to get in

## It felt like the lead up to Rust 1.0

This was also a huge amount of hard work & therefore burnout

# Feature-driven vs time-boxed

## Async/await is a great example of this!

Even though it may not feel like it....

## Keyword reserved in Rust 2018

We committed to it, but weren't ready in December 2018

## Feature shipped in 2019

IMHO, this is how the process should work

# What should we do?

# We should have a 2021 Edition

We should also commit to a train model for editions, and have one every three years.

It should be much smaller than Rust 2018 was.

A small edition has many positives, and not many negatives.

# Release trains are good

And editions are a form of release.

If six weeks goes by without new big features, we still release a new `rustc`.

If three years goes by without new big breaking changes needed, we should still release a new edition.

# The social part of editions matters

Some folks argue that it's not.

It's nice for people to be able to get a quick summary of what's happened in the past three years.

Folks who don't use Rust don't pay close attention to our (very frequent) releases. Nor should they!

Smaller editions are a nice marketing message; Rust is in fact, stable.

# What features should be in Rust 2021?

I actually don’t personally care.

I don’t think that we really need to consider what should be in the edition to justify doing an edition.

In fact, I think the argument is stronger without it.

Consistency and scheduling is key. If a feature misses an edition train, there’ll be another in three years.

# Why a three-year cadence?

It’s just a good amount of time.

C++ explicitly follows a three-year cadence, after C++0x.

Yearly is far too much.

Five years would be too long.

# C++0x: a cautionary tale

## The first draft was in 2008

C++03 was the previous standard

## They hoped for C++08 or C++09

There were specific things we needed to get in

## It ended up shipping in 2011

There's a joke about C++0a, since `a` in hex is `10` in decimal

## 2021 edition?

internals

**mark-i-m** Apr 13

I was just taking a look at the roadmap RFC again, and I was wondering what features/breaking changes we are planning to do for the proposed edition (or if we have settled that we want an edition). Is there a list of things that we want to complete by October supposing we do an edition?

## Prepare for a Rust 2021 edition

Editions were established as a means to help communicate the progress of the Rust language and provide a rallying point for overarching pieces of work. One of our goals for this year should be plan out any changes that we wish to make as part of the next Rust edition. If we are to continue the three-year cadence established with the release of Rust 2018, then the next edition would be released in 2021.

One thing that we learned quite clearly from the experience of Rust 2018 was the importance of preparation. If we wish to do a Rust 2021 edition, we need to be planning for it now. **The goal should be that any changes we wish to make in Rust 2021 are completed by October of 2020**. Completed here means that the changes are available on Nightly. This leaves 2021 to do tooling and polish work, such as lints that will port code forward.

We have not yet formally decided to do an edition. **One specific scenario where we *would* expect to go forward with an edition is if we have work landed by October 2020 that relies on one.** The final decision will be made in October with an RFC, and it will be based on the work that has been completed until that date.

**What might an edition contain?** We've got a number of "in progress" language design features that may require minor changes tied to an edition, but this list is by no means exhaustive:

- Error handling, which could potentially see the introduction of new syntactic forms;
- Improvements to the trait system;
- Improvements to unsafe code, which might involve introducing new syntax like the `&raw` form proposed in RFC 2582.

One goal for this year, then, is to flesh out those areas in more detail and decide what changes, if any, we would like to do for Rust 2021. It is key to identify and plan out the changes we want to make sufficiently early that the tooling and documentation around these changes has time to mature before shipping.

# We should have specifics soon

## Lang team has had a few discussions

Goal is still to have a plan by October

## Nothing on the scale of 2018 changes

Those were huge! And there were a lot of them!

## There will be an RFC

Niko and I have been working on this. Mostly Niko.

# Thanks!