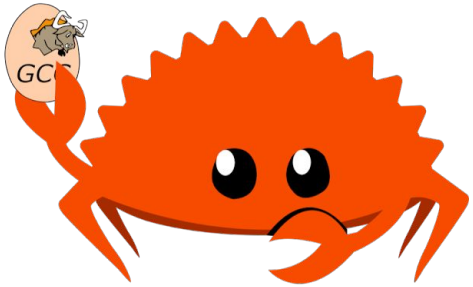
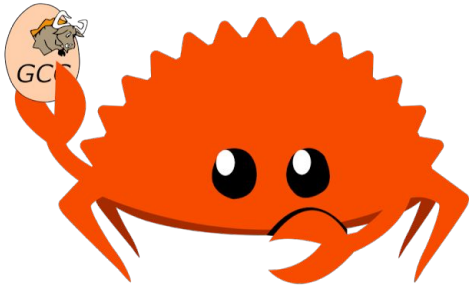


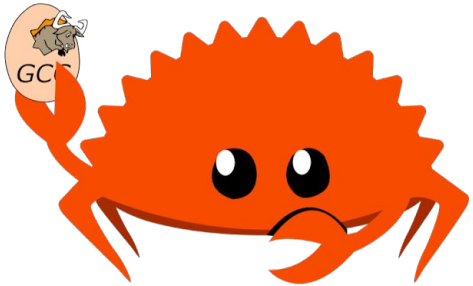
Multi-file support in gcc-rust

LSE

What is gcc-rust?



What is gcc ~~-rust~~?

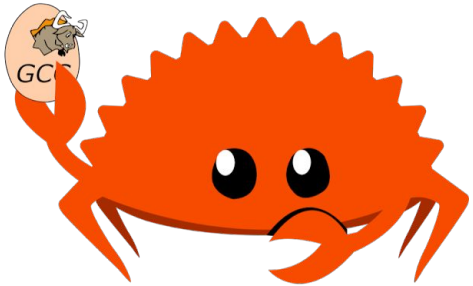


- GNU Compiler Collection
- Multiple frontends tied to a single intermediate representation
- gcc Backend which targets a large set of architectures

- Written in C++
  - … Sometimes

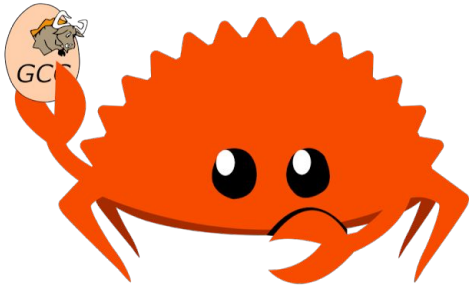
- Large
- Lots of state

What is ~~gee~~ rust?



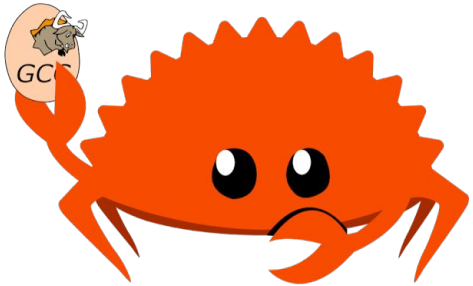
- Safety
- Speed
- Concurrency
- Expressivity
- No garbage collection and no manual memory management! (lifetime analysis)
- Zero-cost abstractions
- Zero setup cross compilation
- Strong ecosystem and tons of libraries at your fingertips!

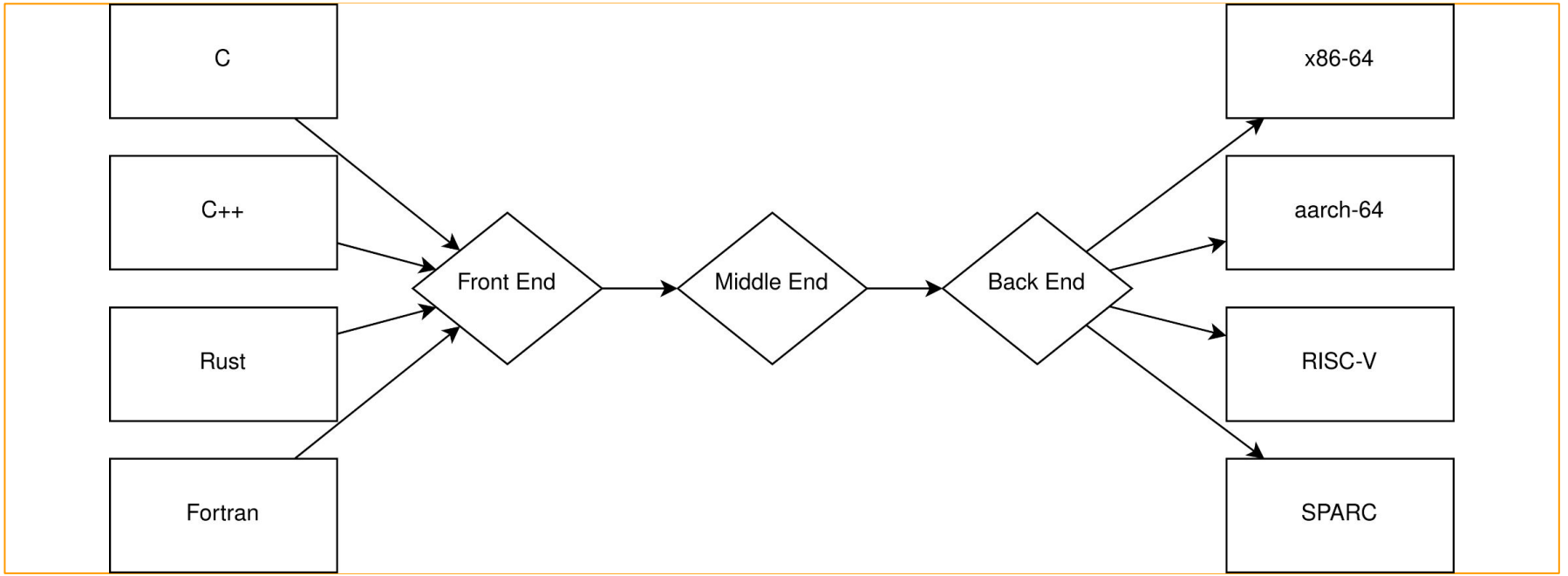
Why?



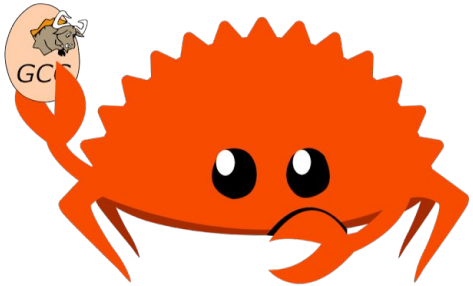
- Alternative implementation of a rust compiler
- Resolve the bootstrapping problem
- Ability to reach more targets

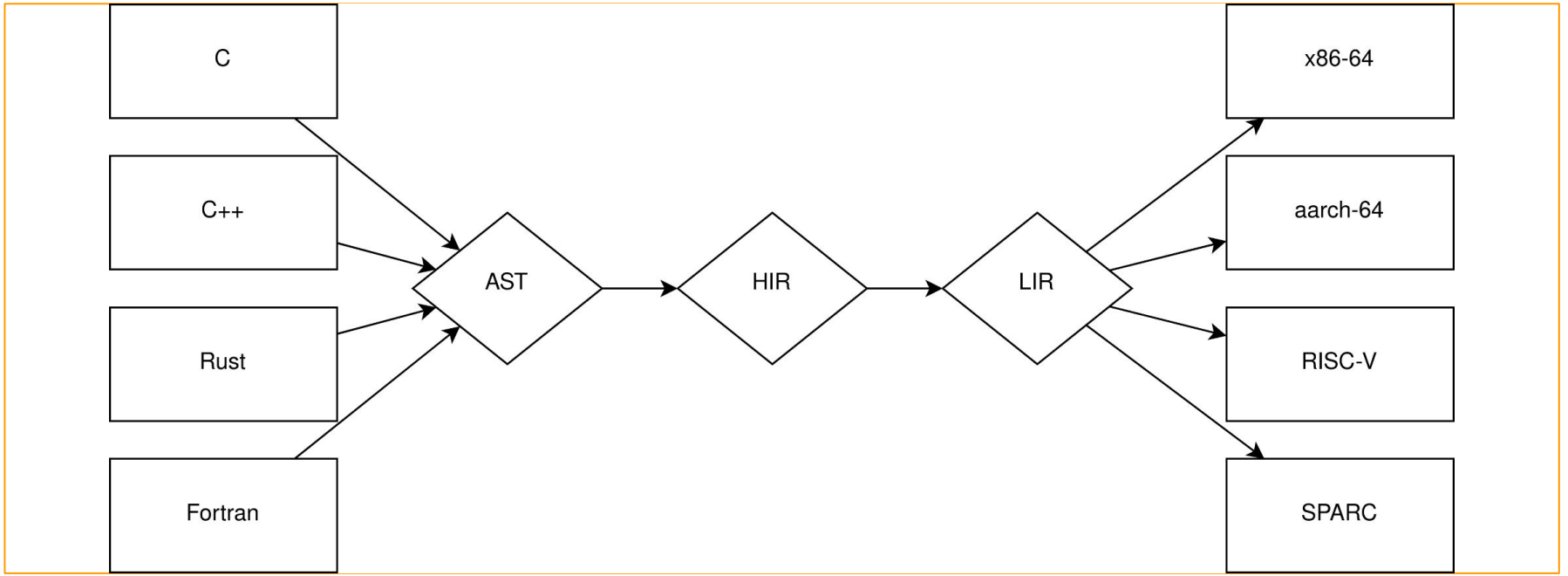
Compilers 101



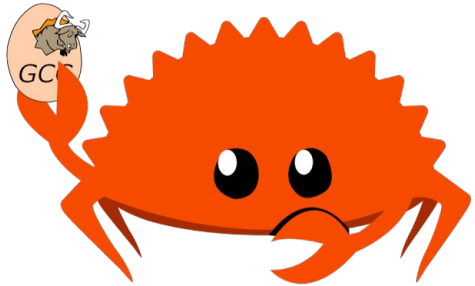


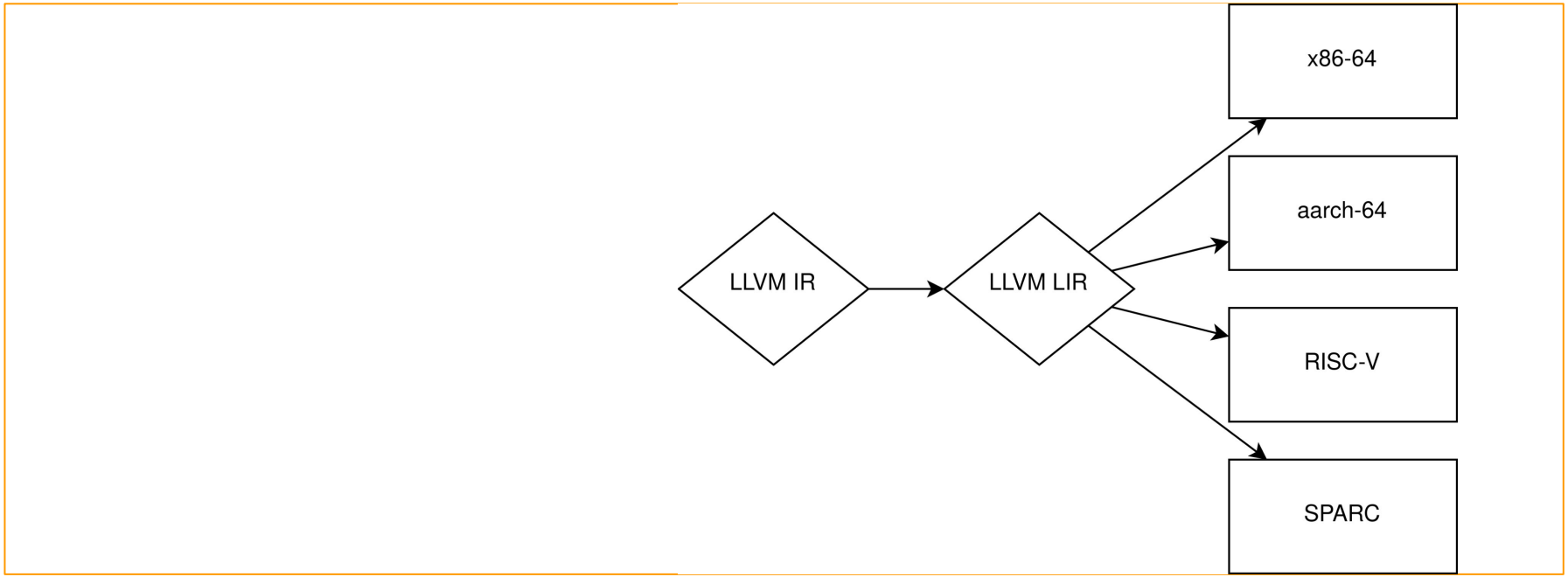
Compilers 101



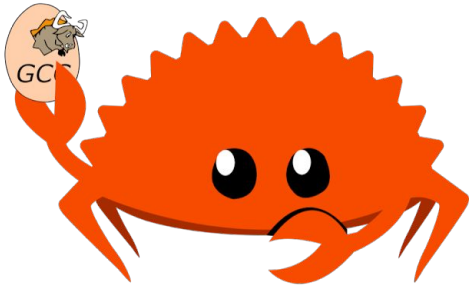


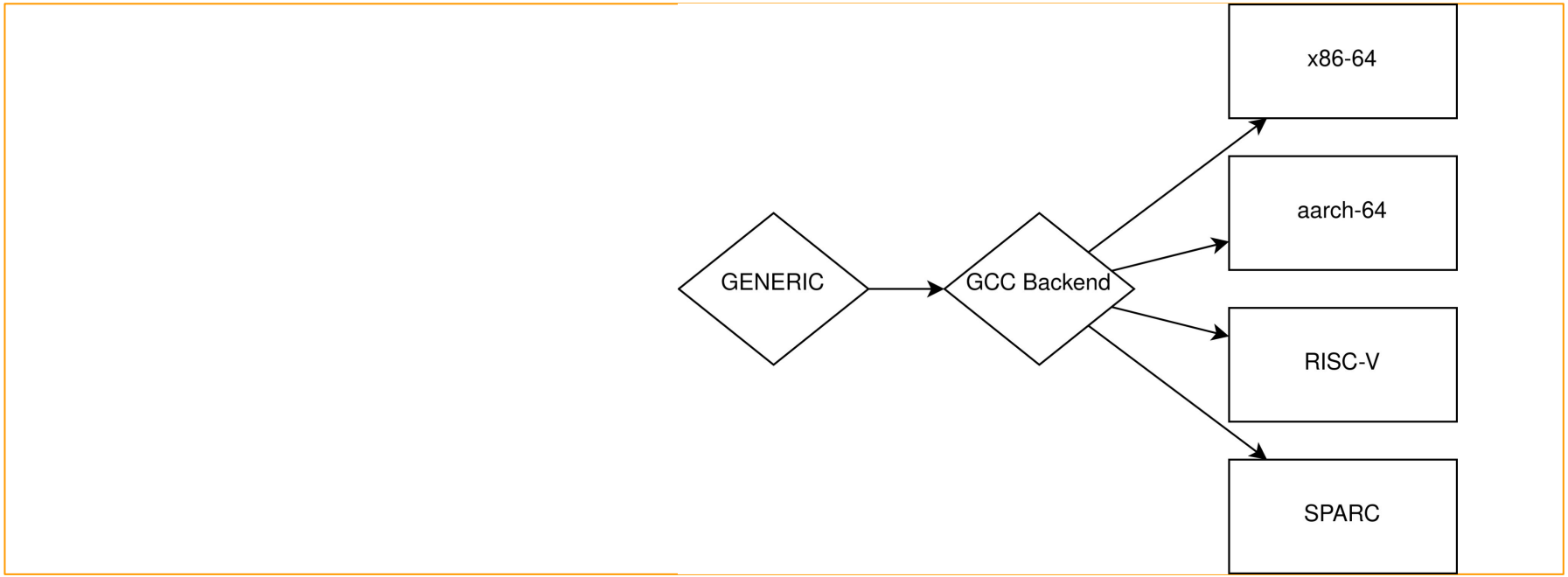
Compilers 101



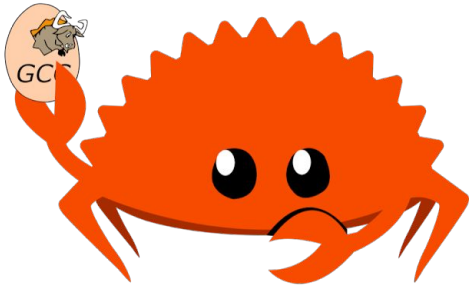


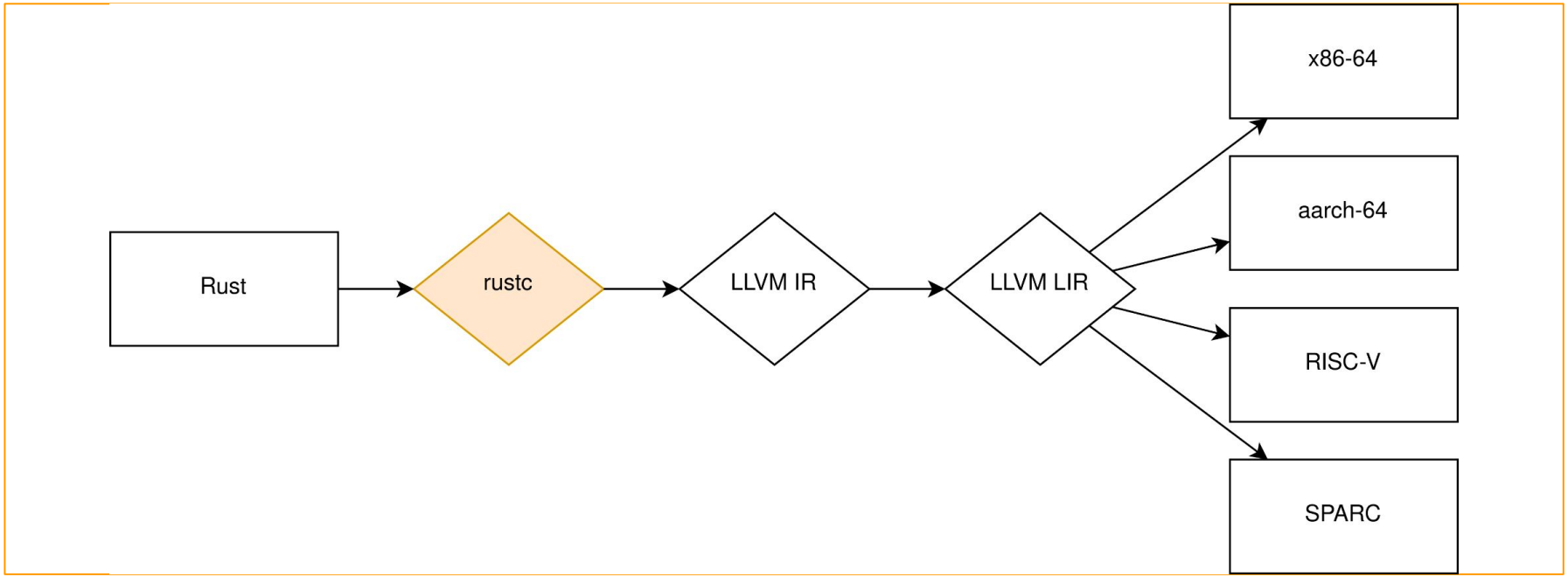
Compilers 101



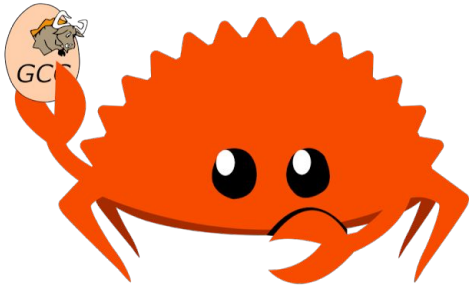


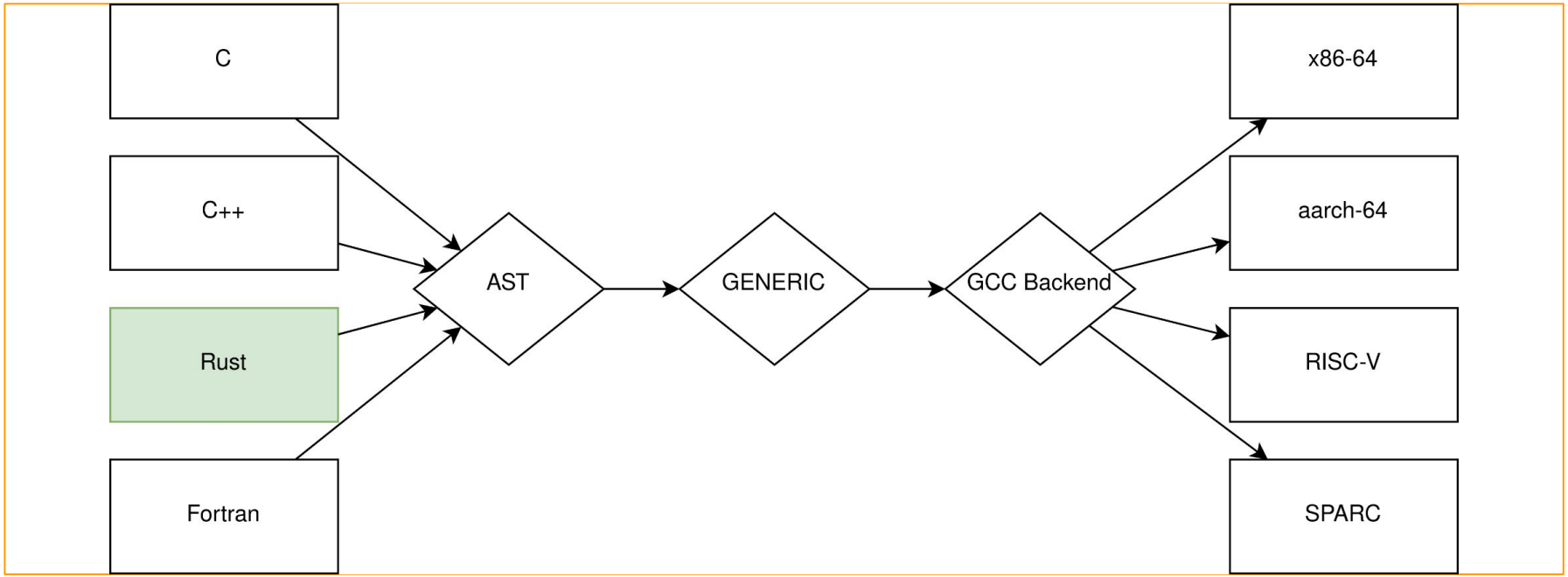
Compilers 101



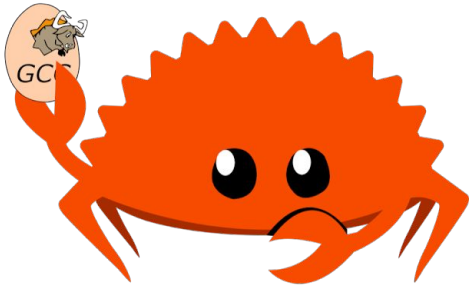


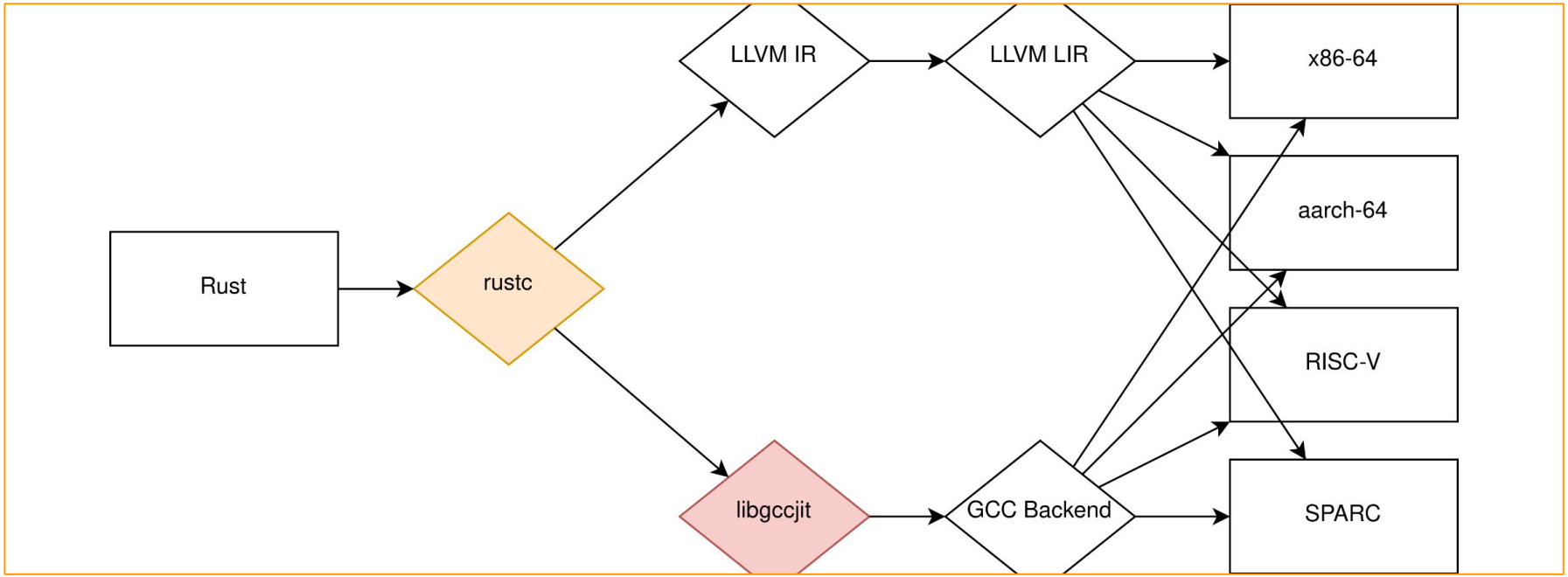
Compilers 101



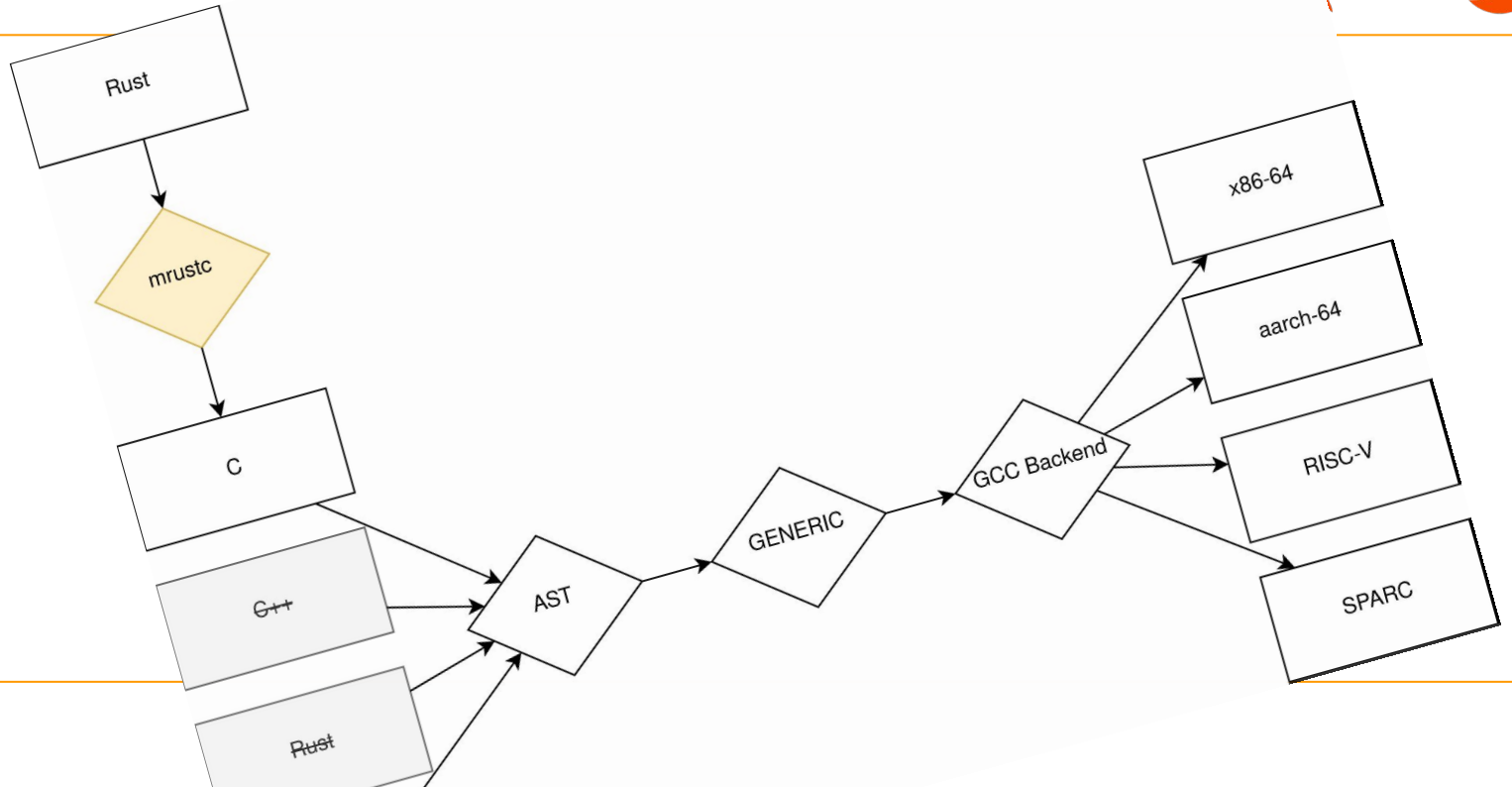


Compilers 101

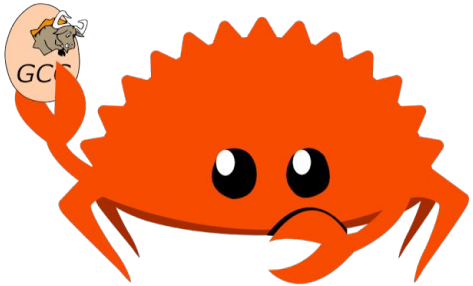




Compilers 101

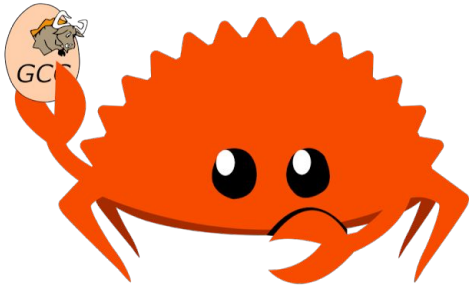


gccrs and Linux



- The Rust-for-Linux effort is getting traction!
- Historically, the Linux kernel is very tied to gcc
- Only recently compilable through clang and LLVM
- Heavy usage of gcc plugins
- We could also backport the rust frontend to older gcc versions to target even more systems and distributions!

Current state



Let’s take a look at some basic Rust examples...

What's working


```
fn main() {
    println!("Hello World!");
}
```

What's working


```
fn main() {

}
```

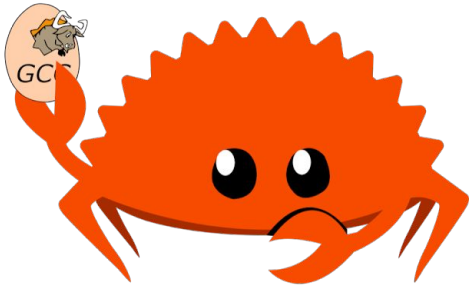
What's working


```
extern "C" {
    fn printf(s: *const i8, ...);
}

fn main() {
    unsafe {
        printf("Hello World!");
    }
}
```

What's working


```
extern "C" {
    fn printf(s: *const i8, ...);
}

fn main() {
    unsafe {
        let s = "Hello World!";
        let s_ptr = s as *const str;
        let s_i8_ptr = s_ptr as *const i8;

        printf(s_i8_ptr);
    }
}
```

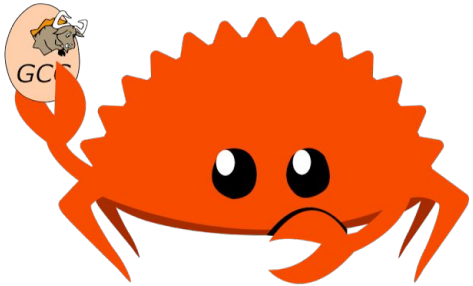
Current state



Let’s take a look at some basic Rust examples…

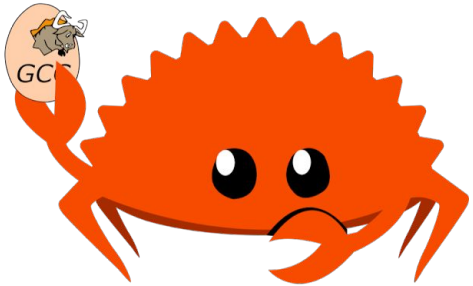
https://docs.google.com/spreadsheets/d/1B_JFzHgGclpdtPcQvnThkNJnP7Hh8fCIAU1rYFu_23M/edit#gid=0

Preliminary work



- Tiny refactors (*rust_debug()* function, Session manager singleton) #388, #466, #612...
- Tiny bugfixes (infinite loops, ICEs...) #319, #458...
- Work on *cargo-gccrs*
  - Raising issues on the compiler
  - Figuring out the expected rustc behavior
  - Allow gccrs to behave like rustc despite offering more possibilities

Multiple file support

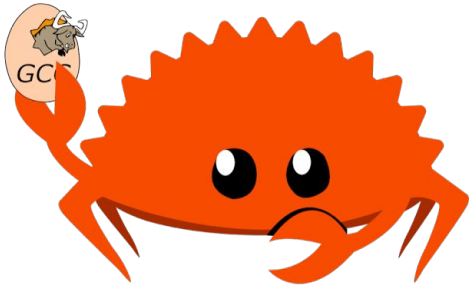


- The *mod* directive in Rust
  - *inner* modules
  - *extern* modules
- But none of them were supported!
- Marc Poulhies, an Adacore GCC Developer had some work done on inner modules
- Strong difference with how C/C++ handles multiple files

Rust modules


```
mod extern;    // Defined in another file, needs fetching

mod intern {   // Defined in this file, nothing to do
    struct SomeStruct(i32);
}
```

Multiple file support



- Multiple steps to get external modules working:
  - Rework some AST Classes
  - Split up some parsing functions to avoid parsing an entire *crate* each time
  - Rework the *SessionManager* in order to use a different location
  - Resolve an external module's filename (more on that later…)
  - Rework some HIR classes
  - Expand the items contained in an external file
  - Allow the user to specify an external mod's path via #[path]
  - Fix some bugs...

AST Module classes


```
mod extern;     // Resolves to ModuleNoBody

mod intern {    // Resolves to ModuleBodied
    struct SomeStruct(i32);
}
```

AST Module classes


```
enum Module {
    Loaded(Vec<Ptr<Item>>),
    Unloaded,
}
```

AST Module classes


```
class Module {
    enum ModuleKind {
        LOADED,
        UNLOADED,
    };

    ModuleKind kind;
    std::vector<std::unique_ptr<Item>> items;
};
```

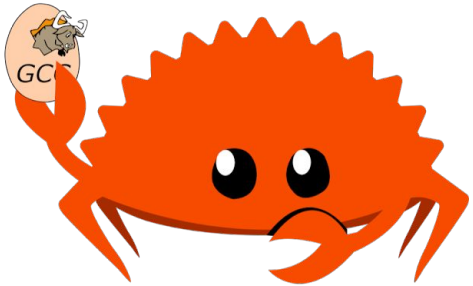
AST Module classes


```
let items = Parser::parse_items(lexer);

*module = Module::Loaded(parsed_items)
```

AST Module classes

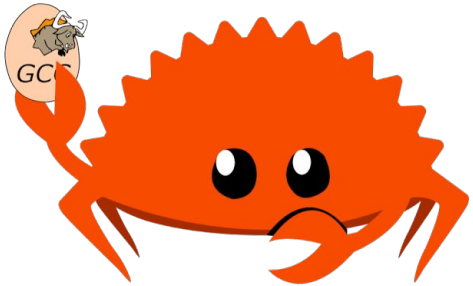

```
auto items = Parser::parse_items(lexer);

module.kind = ModuleKind::LOADED;
module.items = std::move(parsed_items);
```

Resolving the filename



- We're in C++, so surely there is some nifty abstraction to work with the file system
  - The entire *filesystem* header is only available in C++17
    - We are stuck in C++11 for compatibility and bootstrapping reasons…
  - *Boost.FileSystem* is not a possibility in gcc
  - The only remaining option is to search for files the C way… using *access*
  - Which Windows doesn't really like but oh well
  - We also have to accomodate for weird file separators on Windows

Expanding the items

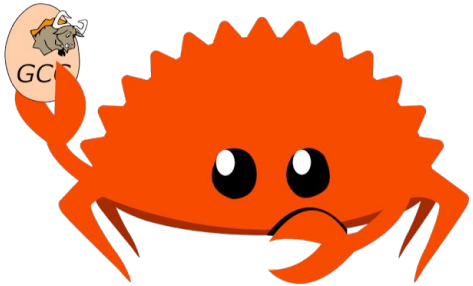


- Module expansion is done during the macro expansion phase
- This way, if a module is hidden behind an unmet cfg-predicate, the items do not get loaded

Expanding the items


```
#[cfg(test)]
mod tests {
    #[test]
    fn validate_bidule() {
        assert_true!(true)
    }
}
```

Expanding the items



- In order to do so, we need to reimplement a subset of the parsing logic, namely:
  - Open a file
  - Lex/Tokenize it
  - Parse the TokenStream as a collection of *Item*s
  - Keep the items as the module's expanded items

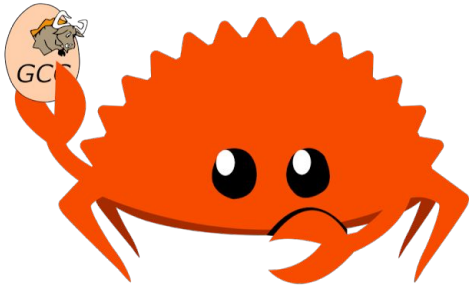
Expanding the items


```
void Module::load_items () {
  process_file_path ();

  RAIIFile file_wrap (module_file.c_str ());
  Linemap *linemap = Session::get_instance ().linemap;

  if (file_wrap.get_raw () == nullptr) {
      rust_error_at (get_locus (), "cannot open module file");
      return;
  }

  rust_debug ("Attempting to parse file");
```

Expanding the items


```
  Lexer lex (module_file.c_str (), std::move (file_wrap), linemap);
  Parser<Lexer> parser (std::move (lex));

  auto parsed_items = parser.parse_items ();

  for (const auto &error : parser.get_errors ())
    error.emit_error ();

  this.items = std::move (parsed_items);
  this.kind = ModuleKind::LOADED;
}
```

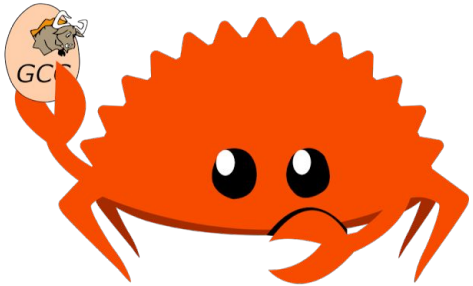
Expanding the items



- To achieve nice looking parsing errors, we need locations…
- In order to get locations, we need to fetch a sh*tload of global variables
- Namely the *Linemap*
- Which is very integrated with the gcc startup logic and hard to use
- In order to figure out how to use the linemap, we need to understand
  - The *Linemap* C++ abstract class
  - The *GccLinemap* class implementing it
  - The C++ low-level *linemap* structure
  - … Which wraps a C *linemap* struct and location

- But no other frontend has to deal with multiple files from one compiler invocation!
- Except… ?

Expanding the items

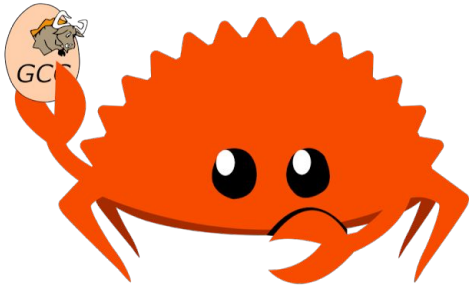


- *gccgo* is a go frontend to gcc
- The Go language allows the import of packages defined in external files, similarly to rust modules
- So we can look at how they use the linemap!

gccgo linemap

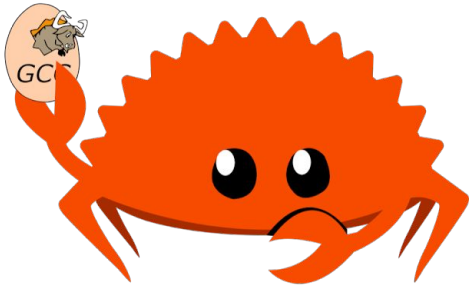

```
#include "gogo.h"

static Gogo* gogo;

GO_EXTERN_C void go_create_gogo(const struct go_create_gogo_args* args)
{
  go_assert(::gogo == NULL);
  ::gogo = new Gogo(...);

  ...
```

Expanding the items



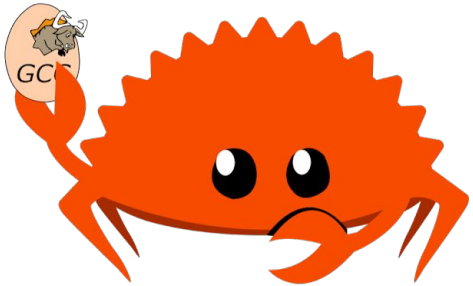
- This is the start of a week-long debugging session
- For some reason we get garbage as the filename of the new file
- Despite looking at *gccgo*'s usage of the linemap, I am not closer to a solution
- We actually have to dive suuuuuuper deep in the C code to find one line of documentation
- Regarding the "lifetime" of the C string used as the filename
- Which makes complete sense
- Would be nice if we had a language that took care of lifetimes for us 👀

Rework the HIR

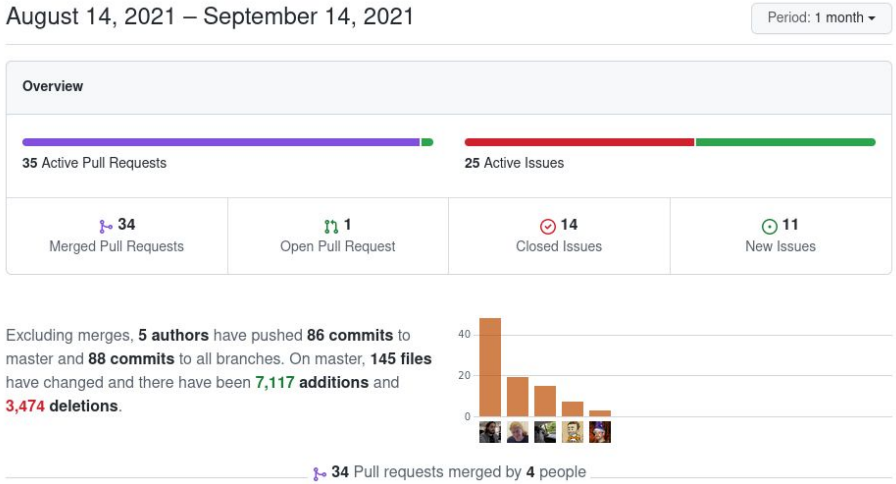


- Similarly to the AST, the HIR classes were also split in two (*ModuleBodied* and *ModuleNoBody*)
- But at this stage of the pipeline, all module items will have been fetched!
- So we can refactor them in a singular *HIR::Module* class which always contains items

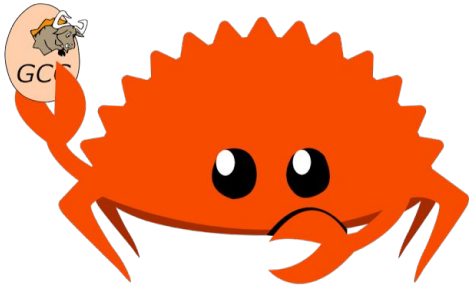
Multiple file support



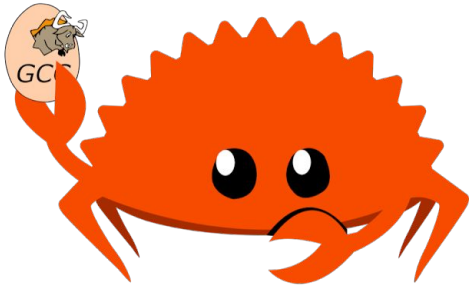
- It was merged!
- Bringing gccrs closer to completing the *Imports and Visibility* milestone
- While Philip is still hacking away on Traits, the compiler is progressing slowly but surely



Reflections



- gccrs is getting closer and closer to being a full-fledged rust compiler
  - Full support is expected in less than a year, minus bug fixes
- There are lots of discussions and considerations around its place in the rust ecosystem
  - … And the “risks” associated with it
- There are way too many globals
- Another thing to tackle in gccrs are macros…
- The compiler gets weekly reports from Philip if you’re interested!
- https://github.com/rust-gcc/reporting



Questions?

Thanks!

cohenarthur.dev@gmail.com

Arthur Cohen