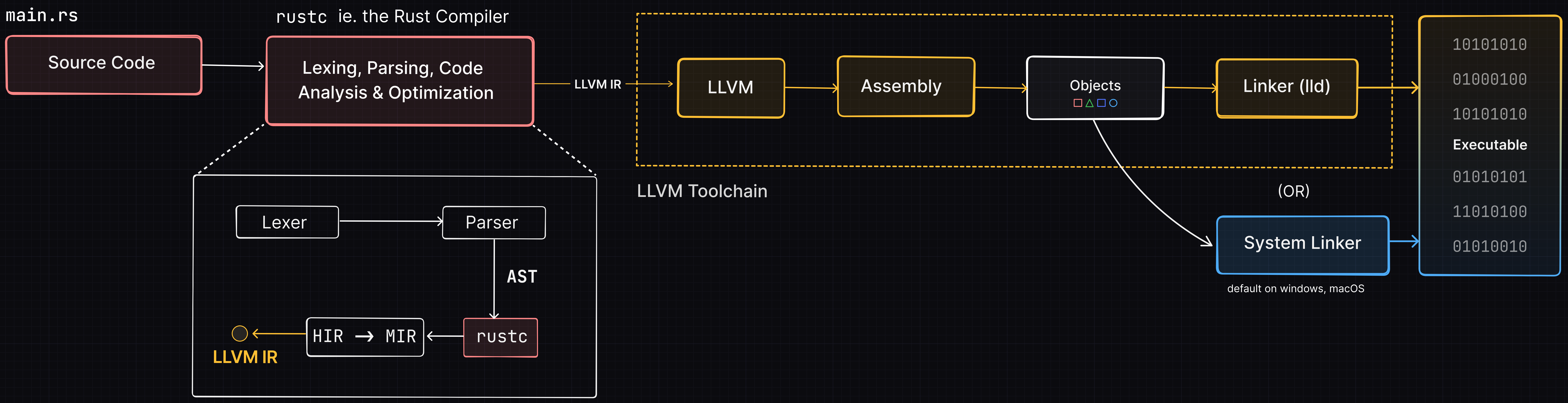
Rust Unlinked

Compiler, Symbols, Linkers & Static Libraries



main.rs

Source Code

rustc ie. the Rust Compiler

Lexing, Parsing, Code Analysis & Optimization

Lexer

Parser

AST

rustc

HIR → MIR

LLVM IR

LLVM IR

LLVM

Assembly

Objects

□△□○

Linker (lld)

LLVM Toolchain

(OR)

System Linker

default on windows, macOS

10101010

01000100

10101010

Executable

01010101

11010100

01010010

Shriram Balaji, Microsoft

Rust.Tokyo 2024

ABOUT ME

- Working on **M365 Core** at Microsoft
- Tinkering with Rust since 2020, newbie `rustc` contributor
- Lately, I've been into reading about databases, systems programming languages, and distributed systems
- **@shrirambalaji** everywhere

AGENDA

- Understanding Linking
- Rust Compilation - A High Level Overview
- Object Files and What’s inside them? (ELF)
- Symbols, Symbol Tables and how to visualize them?
- Simple Program to try manually linking object files
- Link Time Optimization
- Stable ABI & Static Linking
- Experiments with Dynamic Linking

Understanding Linking

Linking involves combining object files into an executable or shared library. It’s like putting together puzzle pieces to create a working program.

→ .exe

Linking involves combining object files into an executable or shared library. It’s like putting together puzzle pieces to create a working program.

→ `./program`

Linking does the magic of **Symbol Resolution**, where the linker matches variable and function names (ie. symbols) to their specific memory addresses, making sure everything fits together.

→ `./program`

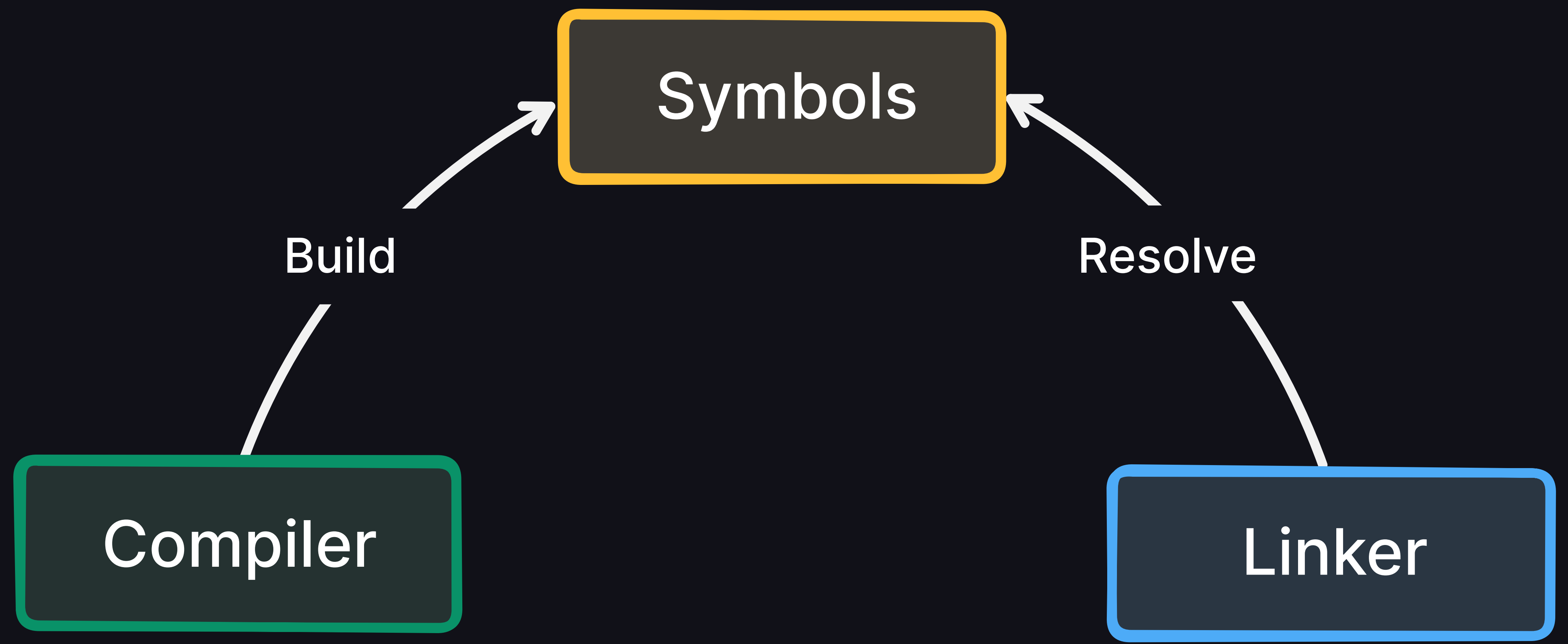
Why is understanding Linking necessary?

Linking time is often a big part of compilation time.

In large Rust projects, roughly *half* of the time could be spent in the linker.

Phases of Compilation

- a compiler compiles source files into object files (`.o files`)
- then, a linker takes all object files and combines them into a single executable or shared library file.



It is **crucial** to understand a little about the stages of rust compilation, *before* we get to linking.

Disclaimer: I’m a newbie `rustc` compiler dev, so there might be certain things that are oversimplified based on my understanding 😄

Rust Compilation - High Level Overview

Simplified

Rust Compilation - High Level Overview

*presented linearly for clarity

Rust Compilation - High Level Overview

*actual implementation is query based

Rust Compilation - High Level Overview

Lexing and Parsing

Source Code

main.rs

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

Lexing and Parsing

Source Code

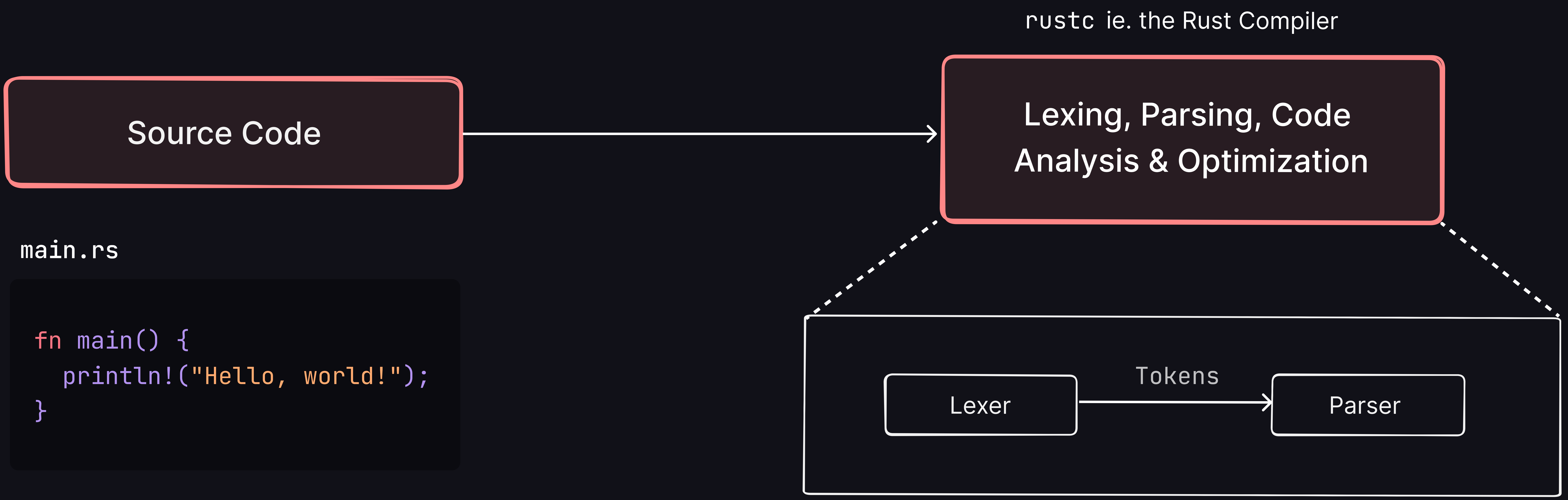
`rustc` ie. the Rust Compiler

Lexing, Parsing, Code Analysis & Optimization

`main.rs`

```rust
fn main() {
  println!("Hello, world!");
}
```

Lexing and Parsing



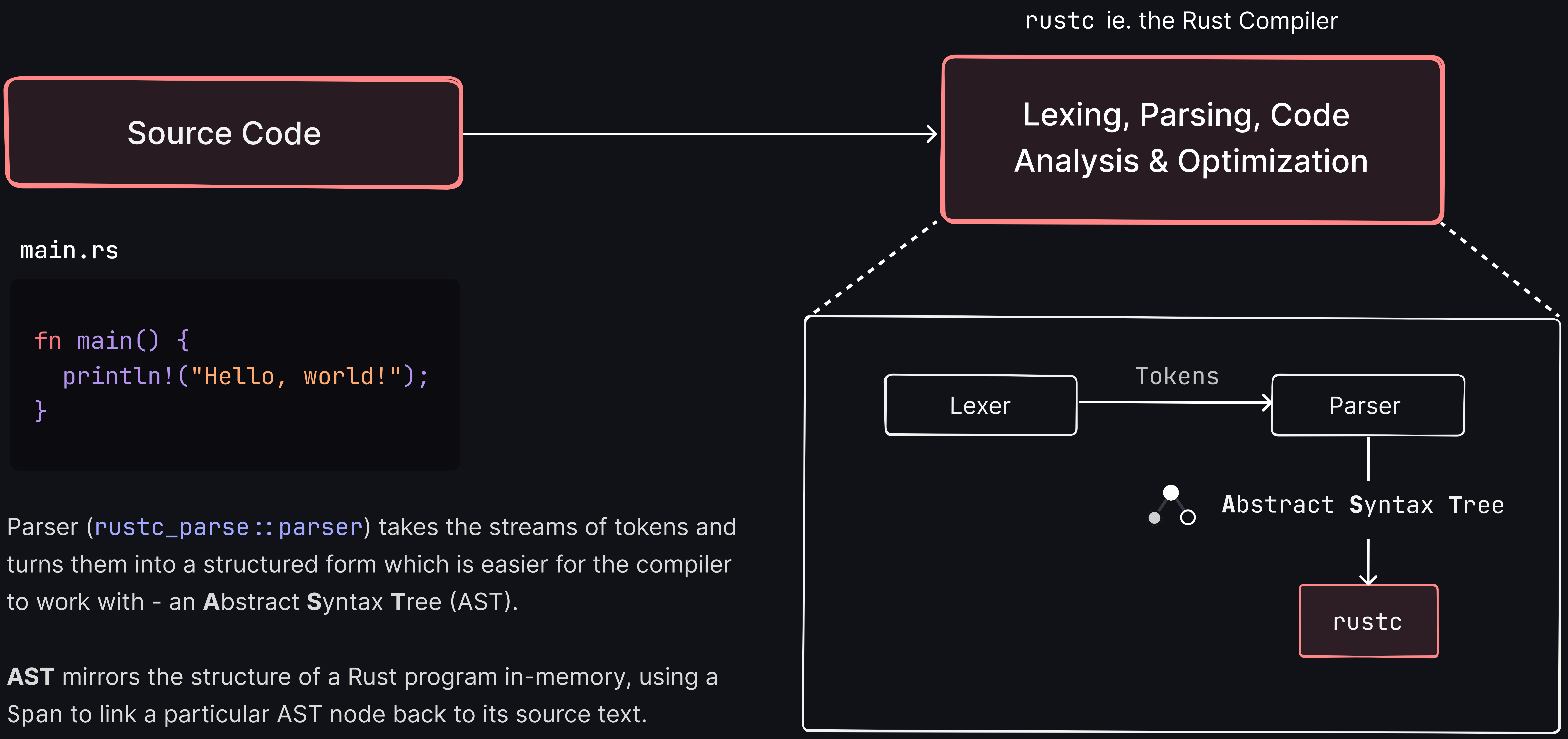
Source Code
rustc ie. the Rust Compiler
Lexing, Parsing, Code Analysis & Optimization
Lexer
Tokens
Parser

main.rs

```rust
fn main() {
  println!("Hello, world!");
}
```

`rustc_lexer + rustc_parse::lexer` converts source code `&str` into parse-able token types for the

Lexing and Parsing


Source Code
rustc ie. the Rust Compiler
Lexing, Parsing, Code Analysis & Optimization
Lexer
Tokens
Parser
Abstract Syntax Tree
rustc

main.rs

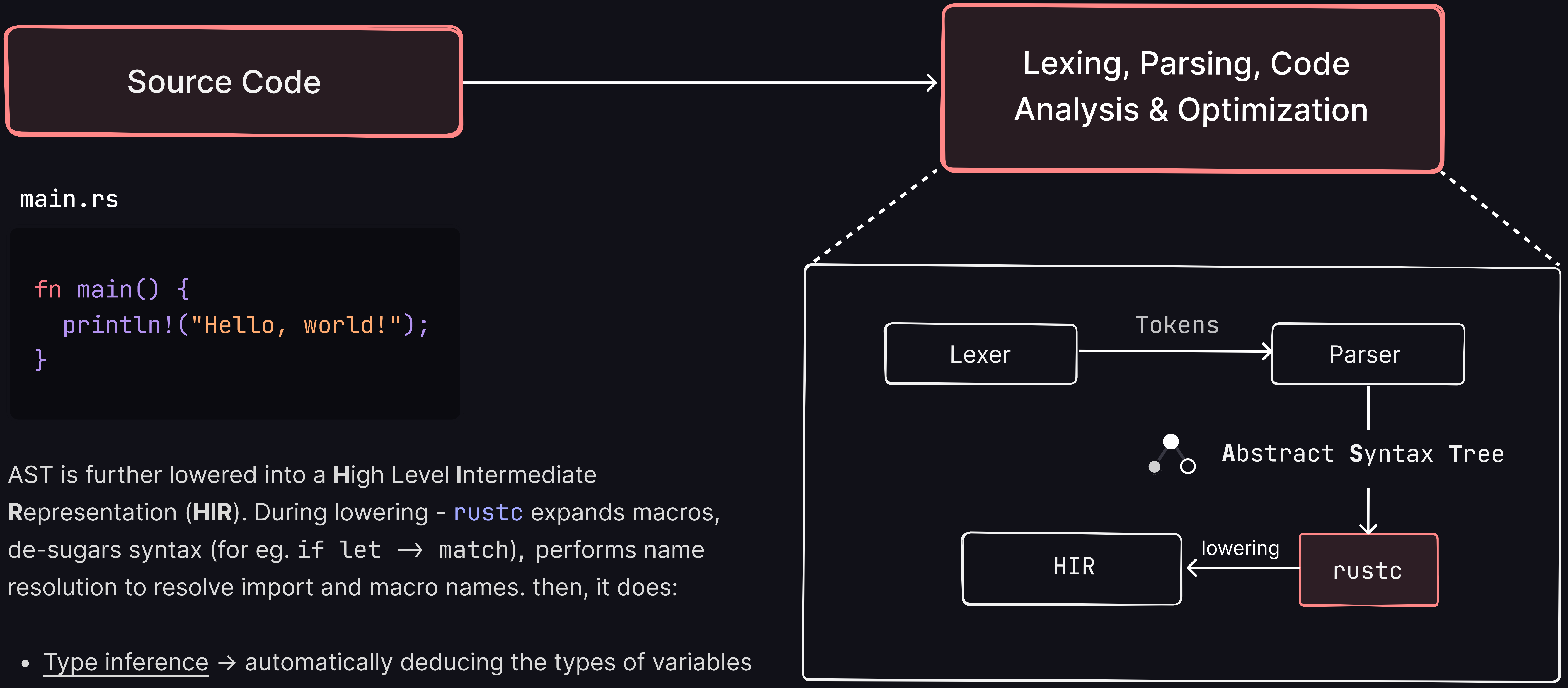
```rust
fn main() {
  println!("Hello, world!");
}
```

Parser (`rustc_parse::parser`) takes the streams of tokens and turns them into a structured form which is easier for the compiler to work with - an **A**bstract **S**yntax **T**ree (AST).

AST mirrors the structure of a Rust program in-memory, using a Span to link a particular AST node back to its source text.

Code Analysis & Optimization

`rustc` ie. the Rust Compiler



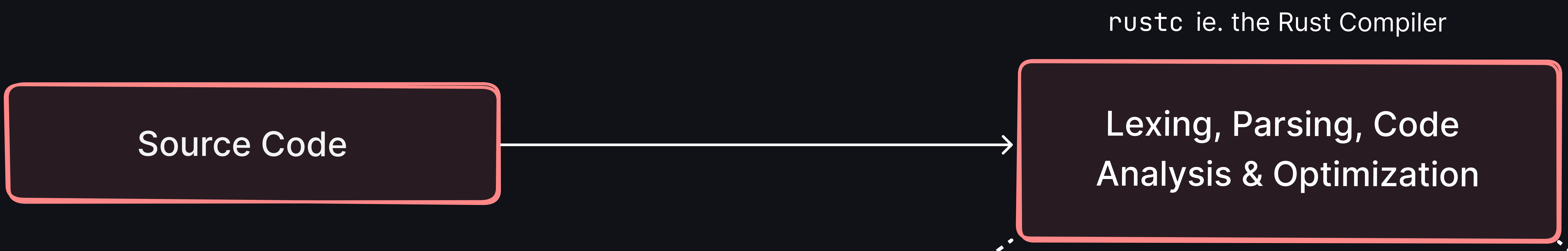
main.rs

```rust
fn main() {
  println!("Hello, world!");
}
```

AST is further lowered into a **H**igh Level **I**ntermediate **R**epresentation (**HIR**). During lowering - `rustc` expands macros, de-sugars syntax (for eg. `if let` → `match`), performs name resolution to resolve import and macro names. then, it does:

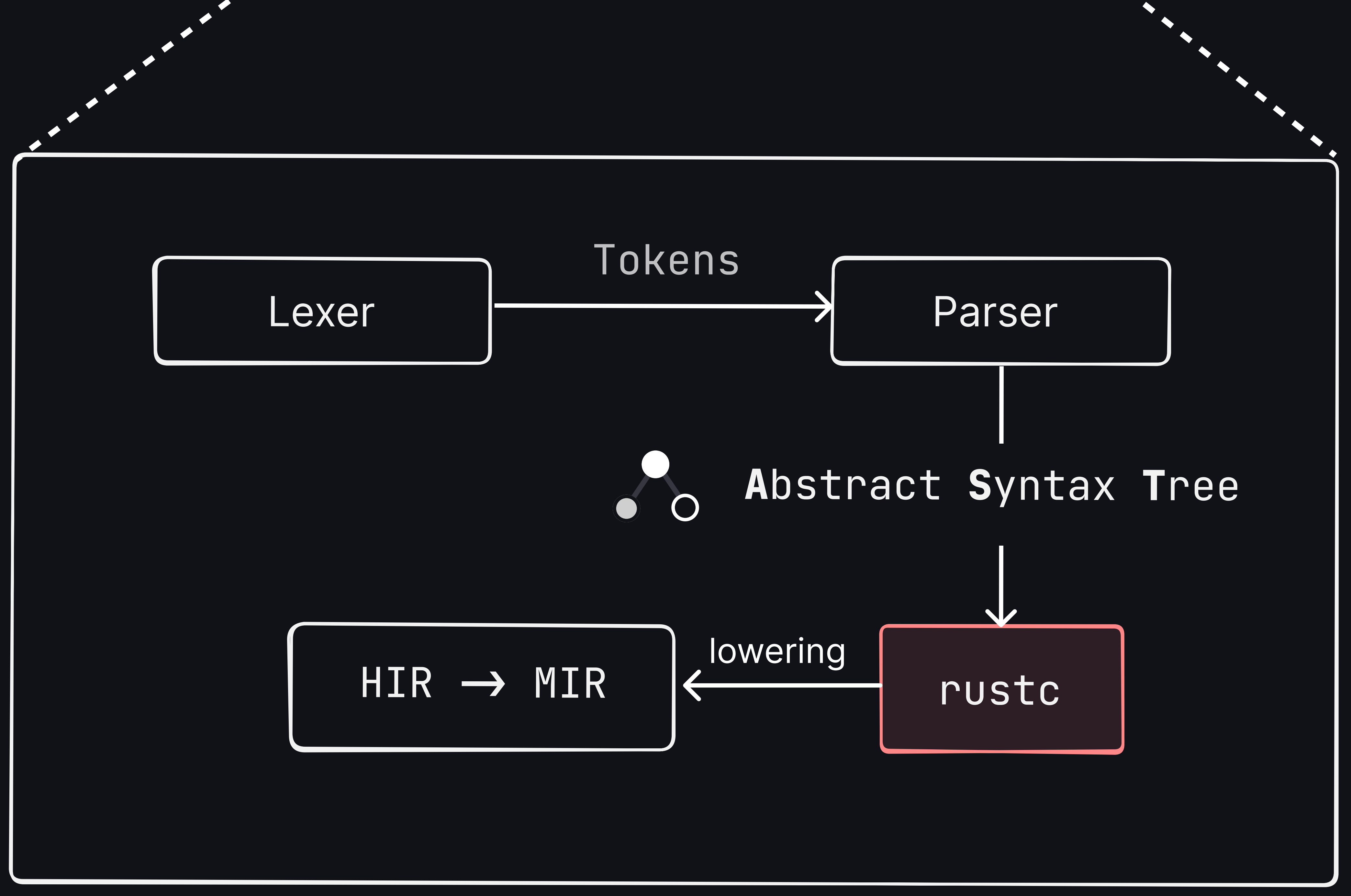
- Type inference → automatically deducing the types of variables and expressions
- Trait Solving → Finding the correct implementation of a trait for a type

Code Analysis & Optimization


rustc ie. the Rust Compiler
Source Code
Lexing, Parsing, Code Analysis & Optimization

main.rs

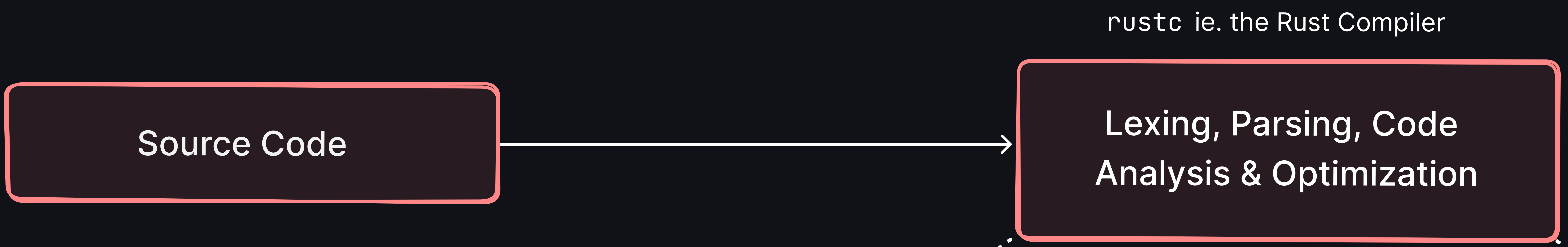
```rust
fn main() {
  println!("Hello, world!");
}
```



The Compiler then runs **Type Checking** on the **HIR,** and is lowered into a **T**yped **HIR** (**THIR**) and then even further into **M**id-Level **IR** (**MIR**). Borrow Checking happens in this phase and along with that `rustc` does operator lowering, monomorphization and many more optimizations *after* borrow checking.

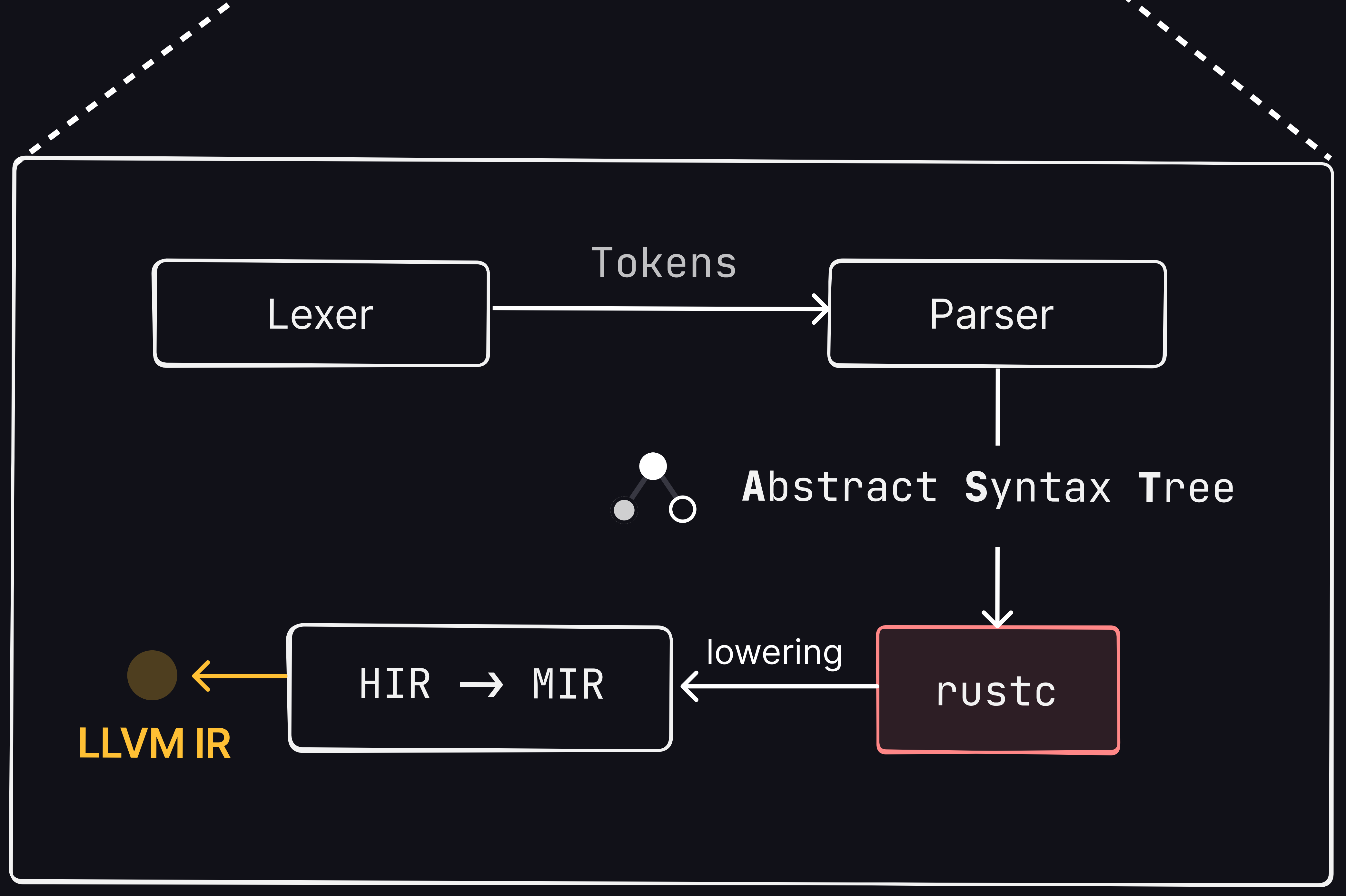
Monomorphization is the fancy term for generating specialized code for each type that a generic function is called with.

Preparing for Code Generation


rustc ie. the Rust Compiler
Source Code
Lexing, Parsing, Code Analysis & Optimization

main.rs

```rust
fn main() {
  println!("Hello, world!");
}
```



After all the optimizations, the MIR needs to get ready for code generation. By default rustc uses LLVM for codegen, and hence the MIR is converted to **LLVM I**ntermediate **R**epresentation (LLVM IR), which is what the LLVM Toolchain works with.

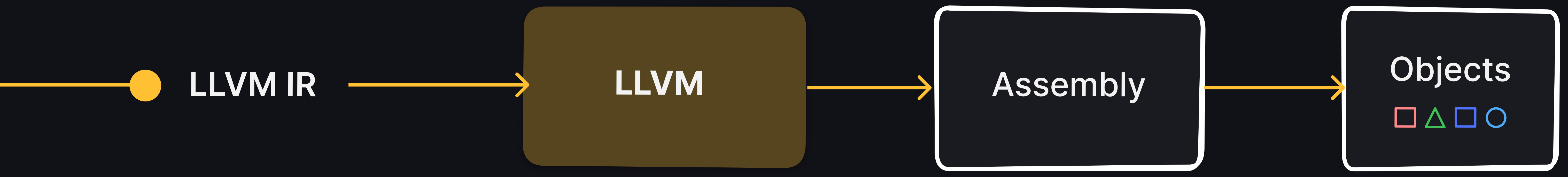
LLVM project contains a modular, reusable & pluggable compiler backend used by many compiler projects, including the clang C compiler and rustc.

Code Generation & Building the executable

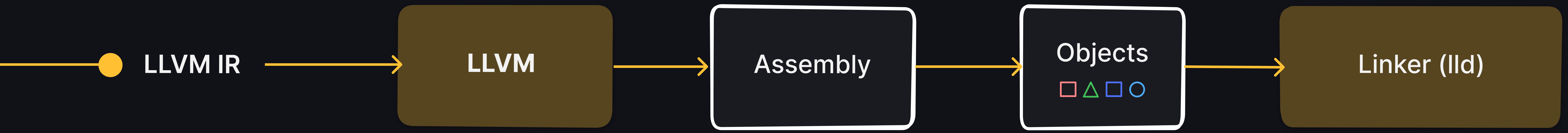
LLVM IR
LLVM

Quick detour: What does the LLVM IR look like?

Code Generation & Building the executable



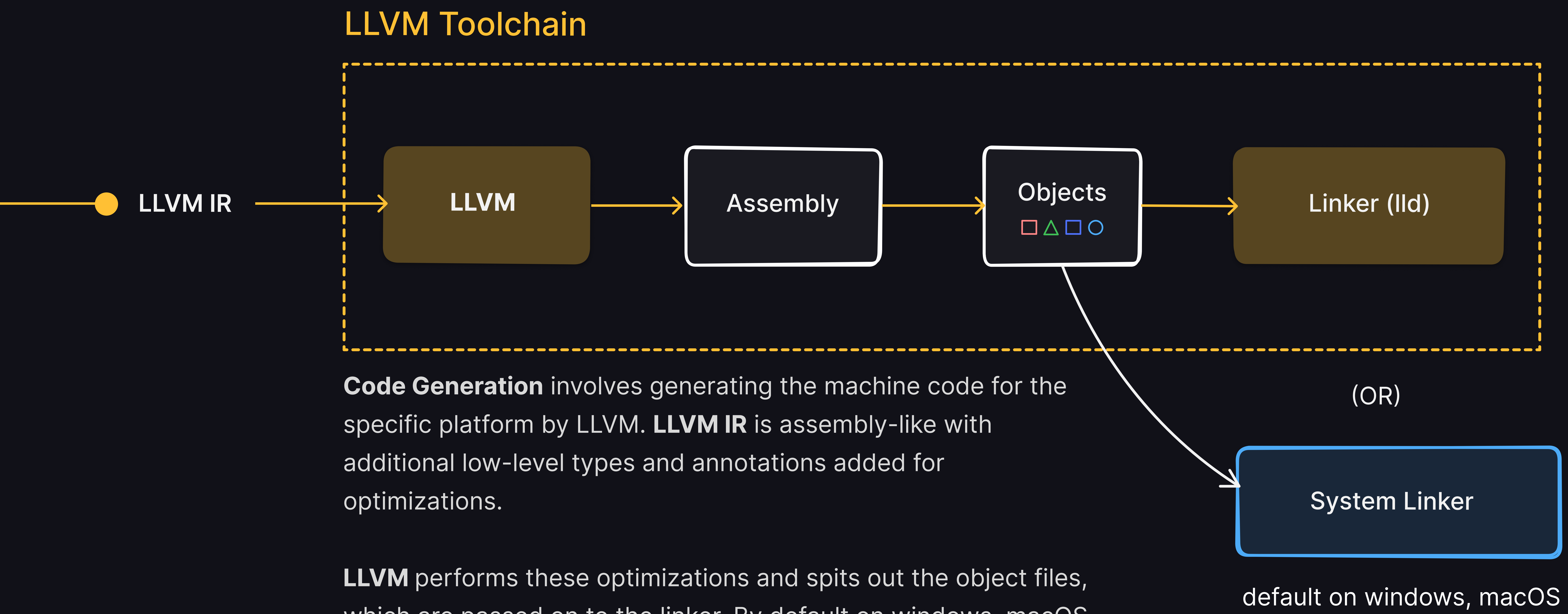
Code Generation & Building the executable



Code Generation involves generating the machine code for the specific platform by LLVM. **LLVM IR** is assembly-like with additional low-level types and annotations added for optimizations.

LLVM performs these optimizations and spits out the object files, which are passed on to the linker.

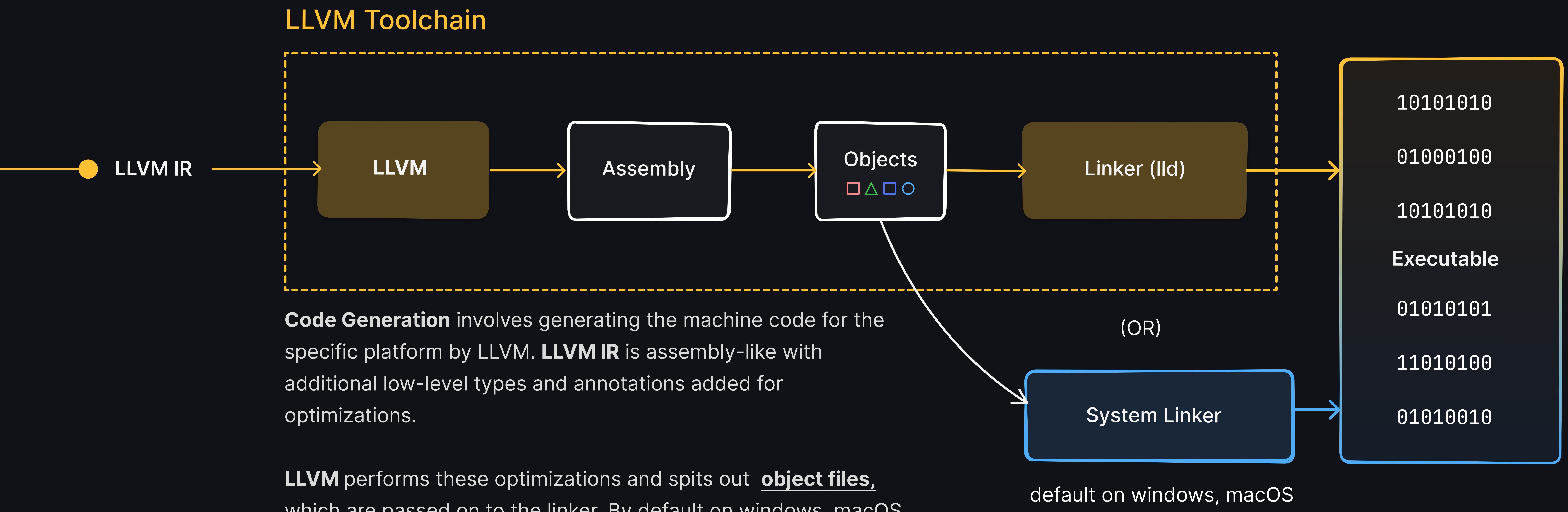
Code Generation & Building the executable



Code Generation involves generating the machine code for the specific platform by LLVM. **LLVM IR** is assembly-like with additional low-level types and annotations added for optimizations.

LLVM performs these optimizations and spits out the object files, which are passed on to the linker. By default on windows, macOS they are passed to system's linker. On linux, as of May 2024 it's passed onto rust-lld in nightly builds.

Code Generation & Building the executable



Code Generation involves generating the machine code for the specific platform by LLVM. **LLVM IR** is assembly-like with additional low-level types and annotations added for optimizations.

LLVM performs these optimizations and spits out **object files,** which are passed on to the linker. By default on windows, macOS they are passed to system's linker. On linux, as of **May 2024** it's passed onto rust-lld in nightly builds. The linker then links together the object files to return an executable.

What is query based compilation?

Demand Driven Compilation with Queries

Query

Demand Driven Compilation with Queries

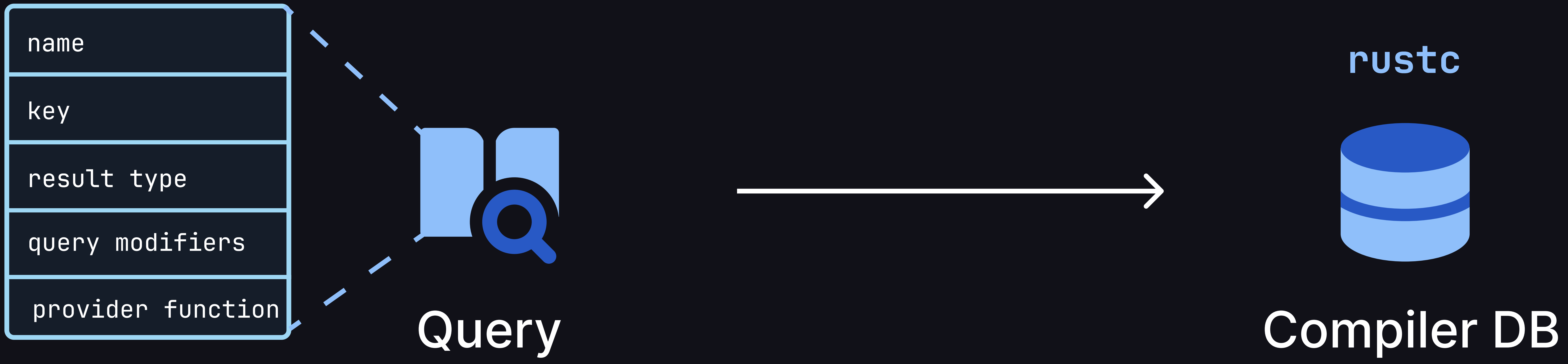
rustc

Query

Compiler DB

compiler's knowledge about a crate → "database"

Demand Driven Compilation with Queries



name

key

result type

query modifiers

provider function

Every Step from earlier is modeled as a “Query”

Let’s look at a query from the “Trait Solving” Step

Demand Driven Compilation with Queries

```
/// Given a crate and a trait, look up all impls of that trait in the crate.
/// Return `(impl_id, self_ty)`.
query implementations_of_trait(key: (CrateNum, DefId)) -> &'tcx [(DefId, Option<SimplifiedType>)] {
    desc { "looking up implementations of a trait in a crate" }
    separate_provide_extern
}
```

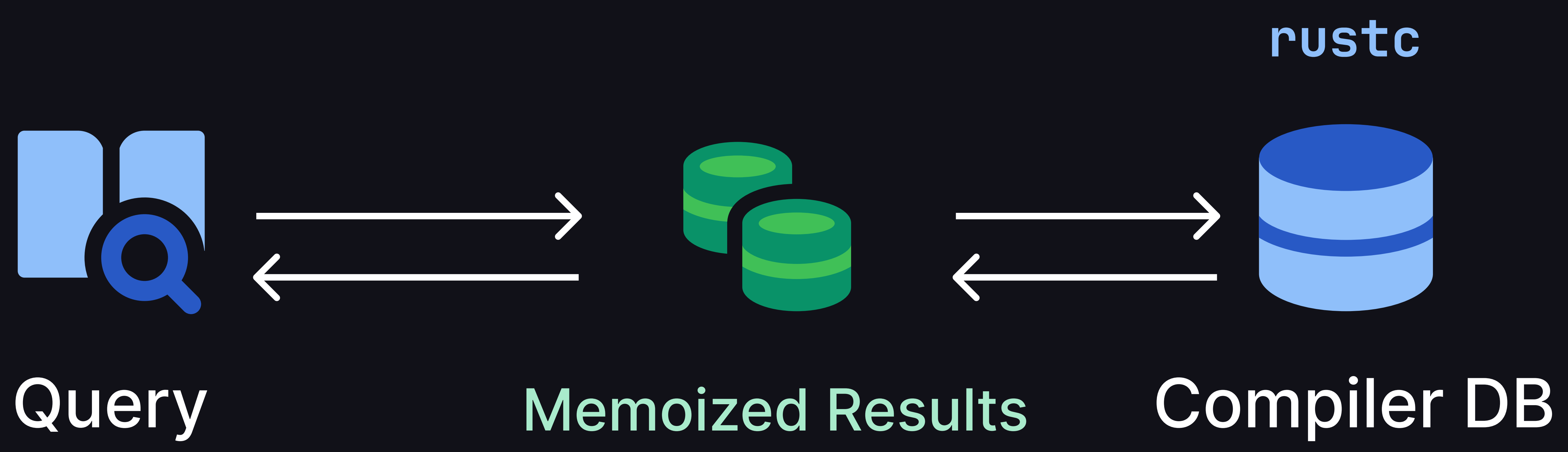
source: compiler/rustc_middle/src/query/mod.rs

Demand Driven Compilation with Queries

```
/// Given a crate and a trait, look up all impls of that trait in the crate.
/// Return `(impl_id, self_ty)`.
query implementations_of_trait(key: (CrateNum, DefId)) -> &'tcx [ ... ]
----- ------------------------      ---------------       ------  ----
▲              ▲                          ▲                 ▲      ▲
|              |                          |                 |      |
|              |                          |                 |  query modifiers
|              |                          |                 |
|              |                          |            result type
|              |                    query key type
|         query name
|
keyword
```

source: compiler/rustc_middle/src/query/mod.rs

Demand Driven Compilation with Queries



*Memoization enables incremental compilation, and faster builds

Enough about Compilation, Back to Linking ⛓️

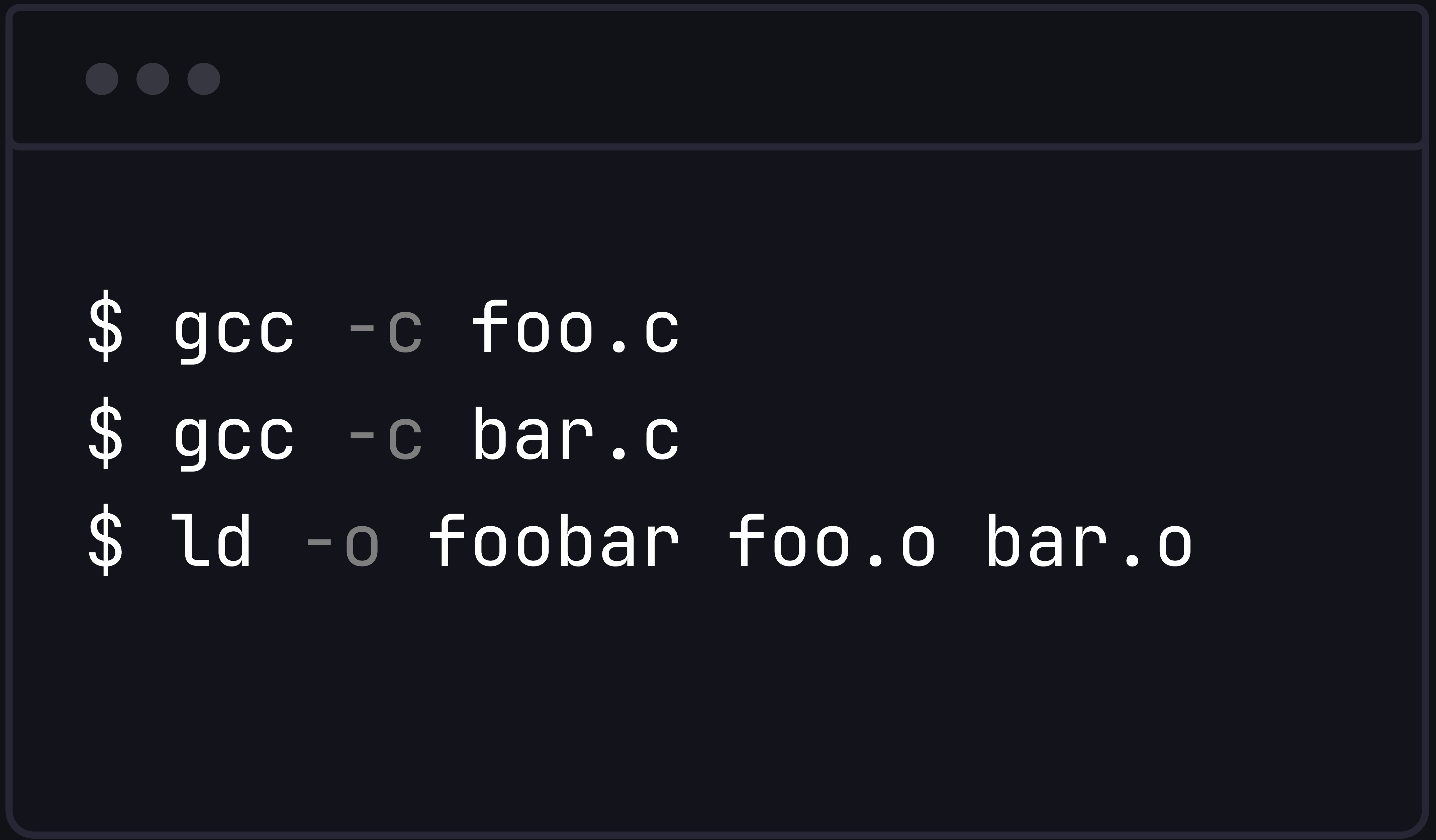
After compiling, there’s a step where object files are generated and later <u>linked</u>

What’s in these `.o` files?

> “An **object file** contains machine code or bytecode, as well as other data and metadata, generated by a compiler or assembler from source code during the compilation or assembly process. The machine code that is generated is known as object code.”

source: Wikipedia

In C, we can typically link object files together by passing them to the linker



```
$ gcc -c foo.c
$ gcc -c bar.c
$ ld -o foobar foo.o bar.o
```

Let’s try something similar with Rust

FOO.RS

```rust
#![no_main]

#[no_mangle]
pub static mut Global: i32 = 5;

#[no_mangle]
pub fn foo() {
    unsafe {
        Global = 10;
    }
}
```

BAR.RS

```rust
#![no_main]

extern "C" {
    static mut Global: i32;
}

#[no_mangle]
pub extern "C" fn bar() {
    unsafe {
        Global = 20;
    }
}
```

importing a `Global` variable from foo.rs in bar.rs and update it’s value to 20.

FOO.RS

```rust
#![no_main]

#[no_mangle]
pub static mut Global: i32 = 5;

#[no_mangle]
pub fn foo() {
    unsafe {
        Global = 10;
    }
}
```

The `#![no_main]` attribute tells the compiler that there is no main function, and effectively not to throw a compiler error when it doesn’t find one.

BAR.RS

```rust
#![no_main]

extern "C" {
    static mut Global: i32;
}

#[no_mangle]
pub extern "C" fn bar() {
    unsafe {
        Global = 20;
    }
}
```

FOO.RS

```rust
#![no_main]

#[no_mangle]
pub static mut Global: i32 = 5;

#[no_mangle]
pub fn foo() {
    unsafe {
        Global = 10;
    }
}
```

The #[no_mangle] attribute disables **mangling**. When Rust code is compiled, identifiers are “mangled” ie. transformed into a different name.

BAR.RS

```rust
#![no_main]

extern "C" {
    static mut Global: i32;
}

#[no_mangle]
pub extern "C" fn bar() {
    unsafe {
        Global = 20;
    }
}
```

FOO.RS

```rust
#![no_main]

#[no_mangle]
pub static mut Global: i32 = 5;

#[no_mangle]
pub fn foo() {
    unsafe {
        Global = 10;
    }
}
```

for eg. `Global` variable gets mangled to `__ZN11foo6Global17ha2a12041c4e557c5E`. This is done to avoid naming conflicts when linking with other libraries.

BAR.RS

```rust
#![no_main]

extern "C" {
    static mut Global: i32;
}

#[no_mangle]
pub extern "C" fn bar() {
    unsafe {
        Global = 20;
    }
}
```

FOO.RS

```rust
#![no_main]

#[no_mangle]
pub static mut Global: i32 = 5;

#[no_mangle]
pub fn foo() {
    unsafe {
        Global = 10;
    }
}
```

however, we disable it with `#[no_mangle]` so that the symbol name is preserved, and can be easily linked by name.

BAR.RS

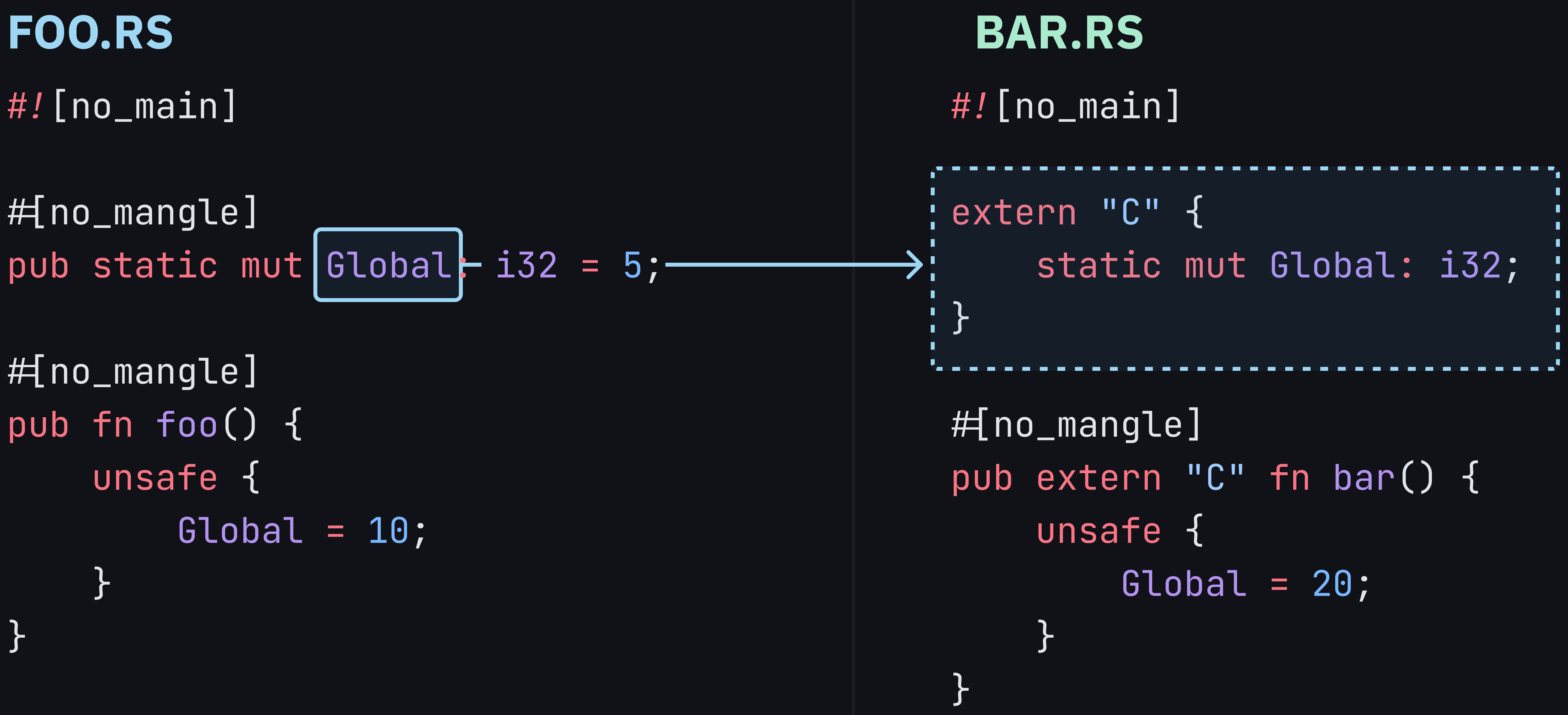
```rust
#![no_main]

extern "C" {
    static mut Global: i32;
}

#[no_mangle]
pub extern "C" fn bar() {
    unsafe {
        Global = 20;
    }
}
```

FOO.RS

```
#![no_main]

#[no_mangle]
pub static mut Global: i32 = 5;

#[no_mangle]
pub fn foo() {
    unsafe {
        Global = 10;
    }
}
```

BAR.RS

```
#![no_main]

extern "C" {
    static mut Global: i32;
}

#[no_mangle]
pub extern "C" fn bar() {
    unsafe {
        Global = 20;
    }
}
```



FOO.RS

```rust
#![no_main]

#[no_mangle]
pub static mut Global: i32 = 5;

#[no_mangle]
pub fn foo() {
    unsafe {
        Global = 10;
    }
}
```

BAR.RS

```rust
#![no_main]

extern "C" {
    static mut Global: i32;
}

#[no_mangle]
pub extern "C" fn bar() {
    unsafe {
        Global = 20;
    }
}
```

FOO.RS

```rust
#![no_main]

#[no_mangle]
pub static mut Global: i32 = 5;

#[no_mangle]
pub fn foo() {
    unsafe {
        Global = 10;
    }
}
```

BAR.RS

```rust
#![no_main]

extern "C" {
    static mut Global: i32;
}

#[no_mangle]
pub extern "C" fn bar() {
    unsafe {
        Global = 20;
    }
}
```

The `extern "C"` block tells the compiler that `Global` is defined elsewhere in a foreign library.

FOO.RS

```rust
#![no_main]

#[no_mangle]
pub static mut Global: i32 = 5;

#[no_mangle]
pub fn foo() {
    unsafe {
        Global = 10;
    }
}
```

BAR.RS

```rust
#![no_main]

extern "C" {
    static mut Global: i32;
}

#[no_mangle]
pub extern "C" fn bar() {
    unsafe {
        Global = 20;
    }
}
```

`extern "C"` doesn't mean we are inter-operating with C, but rather using the platform's C ABI (**A**pplication **B**inary **I**nterface).

FOO.RS

```rust
#![no_main]

#[no_mangle]
pub static mut Global: i32 = 5;

#[no_mangle]
pub fn foo() {
    unsafe {
        Global = 10;
    }
}
```

BAR.RS

```rust
#![no_main]

extern "C" {
    static mut Global: i32;
}

#[no_mangle]
pub extern "C" fn bar() {
    unsafe {
        Global = 20;
    }
}
```

`bar.rs` assumes that a variable declaration for `Global`, is present in a foreign library.

FOO.RS

```rust
#![no_main]

#[no_mangle]
pub static mut Global: i32 = 5;

#[no_mangle]
pub fn foo() {
    unsafe {
        Global = 10;
    }
}
```

This block is unsafe because we are updating a global static mutable.

BAR.RS

```rust
#![no_main]

extern "C" {
    static mut Global: i32;
}

#[no_mangle]
pub extern "C" fn bar() {
    unsafe {
        Global = 20;
    }
}
```

FOO.RS

```rust
#![no_main]

#[no_mangle]
pub static mut Global: i32 = 5;

#[no_mangle]
pub fn foo() {
    unsafe {
        Global = 10;
    }
}
```

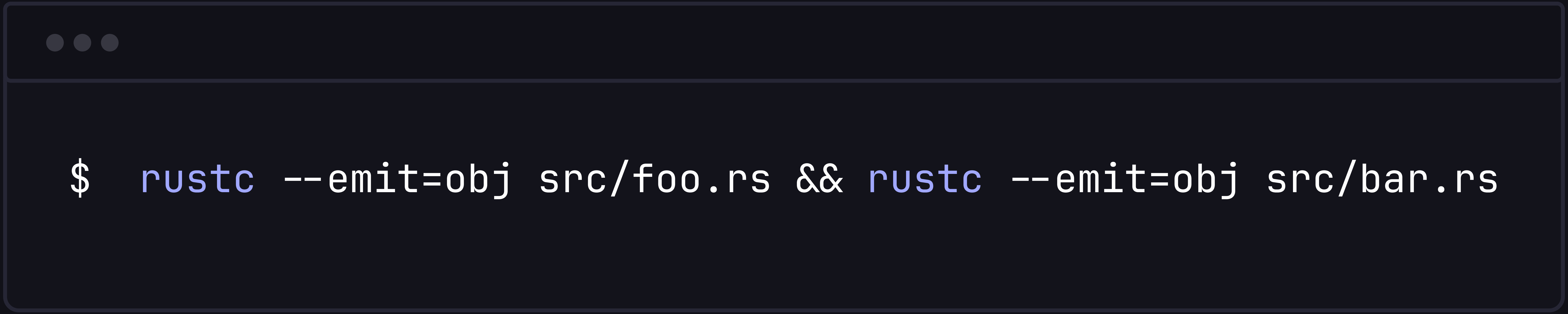
BAR.RS

```rust
#![no_main]

extern "C" {
    static mut Global: i32;
}

#[no_mangle]
pub extern "C" fn bar() {
    unsafe {
        Global = 20;
    }
}
```

This block is `unsafe` because Rust cannot guarantee safety in FFI calls. We are trying to mutate a global static variable imported from a library, which cannot be memory-safe.

Since we want to invoke the linker directly,
let’s **not** use `cargo` for now

Compiling & Emitting Object Files



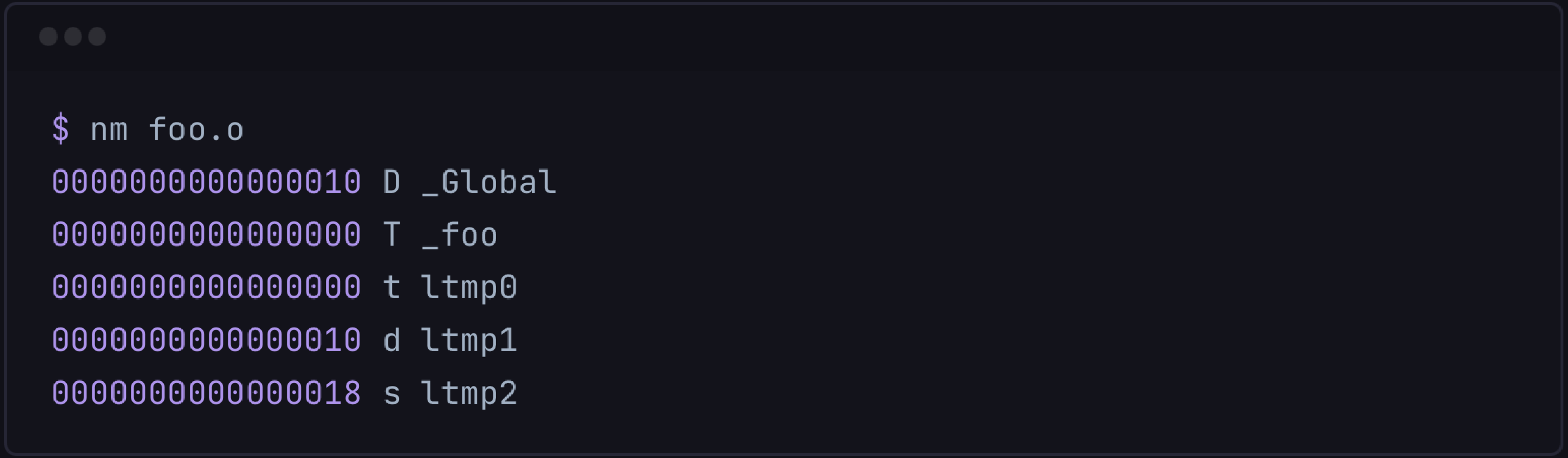
```
$  rustc --emit=obj src/foo.rs && rustc --emit=obj src/bar.rs
```

A **symbol** in a symbol table refers to an identifier, such as a variable name or function name, that is stored in a data structure called a **symbol table**.

Symbols are stored in sections of the object file in a specific format - **ELF** (**E**xecutable and **L**inkable **F**ormat) on Unix-like systems.

In macOS, it’s Mach-O (Mach Object) but similar to ELF.
In Windows, it’s PE / COFF (**P**ortal **E**xecutable / **C**ommon **O**bject **F**ile **F**ormat)

Visualizing Symbols - nm



```
$ nm foo.o
0000000000000010 D _Global
0000000000000000 T _foo
0000000000000000 t ltmp0
0000000000000010 d ltmp1
0000000000000018 s ltmp2
```

The output of `nm` is in the following format:

- `D` - Global Data section symbol
- `T` - Global Text symbol
- `d` - Local symbol in the data section
- `s` - Unitialized Local symbol for small objects

If you haven't noticed, lowercase denotes local symbols, and uppercase denotes global symbols.

The `ltmp` symbols are temporary symbols generated by the compiler during compilation.

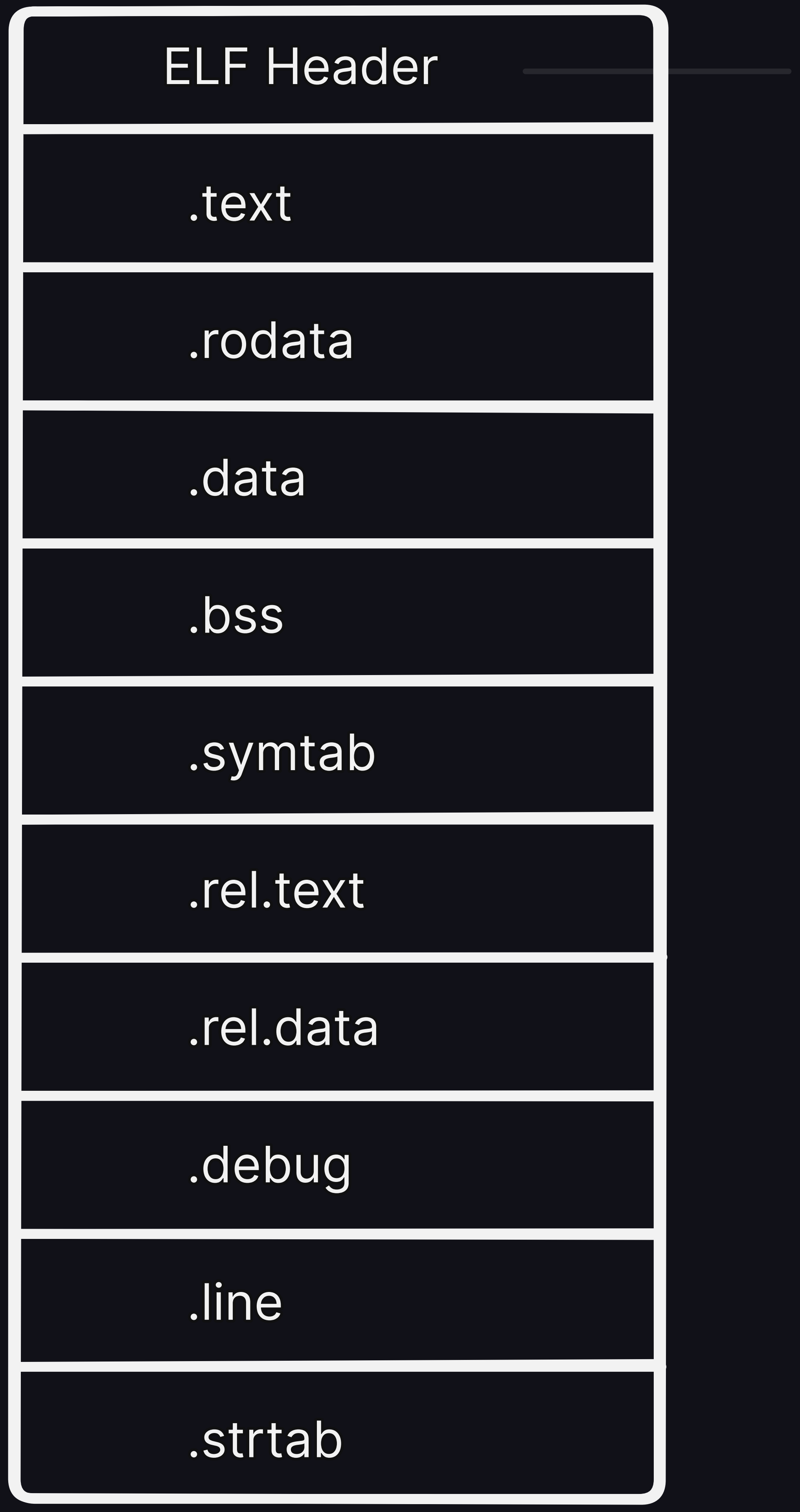
Visualizing Symbols - `nm`

Let's take a look at the symbol table for `bar.o` as well:

```
$ nm bar.o
                 U _Global
0000000000000000 T _bar
0000000000000000 t ltmp0
0000000000000018 N ltmp1
```

wherein `U` denotes an Undefined symbol. Remember, the Undefined pseudo section I was mentioning, that's where the `Global` symbol exists. This is because there's an *undefined* symbol reference to the `Global` variable, which will be resolved only during the linking phase.

Inside **ELF** - **E**xecutable & **L**inkable **F**ormat



ELF Header

.text

.rodata

.data

.bss

.symtab

.rel.text

.rel.data

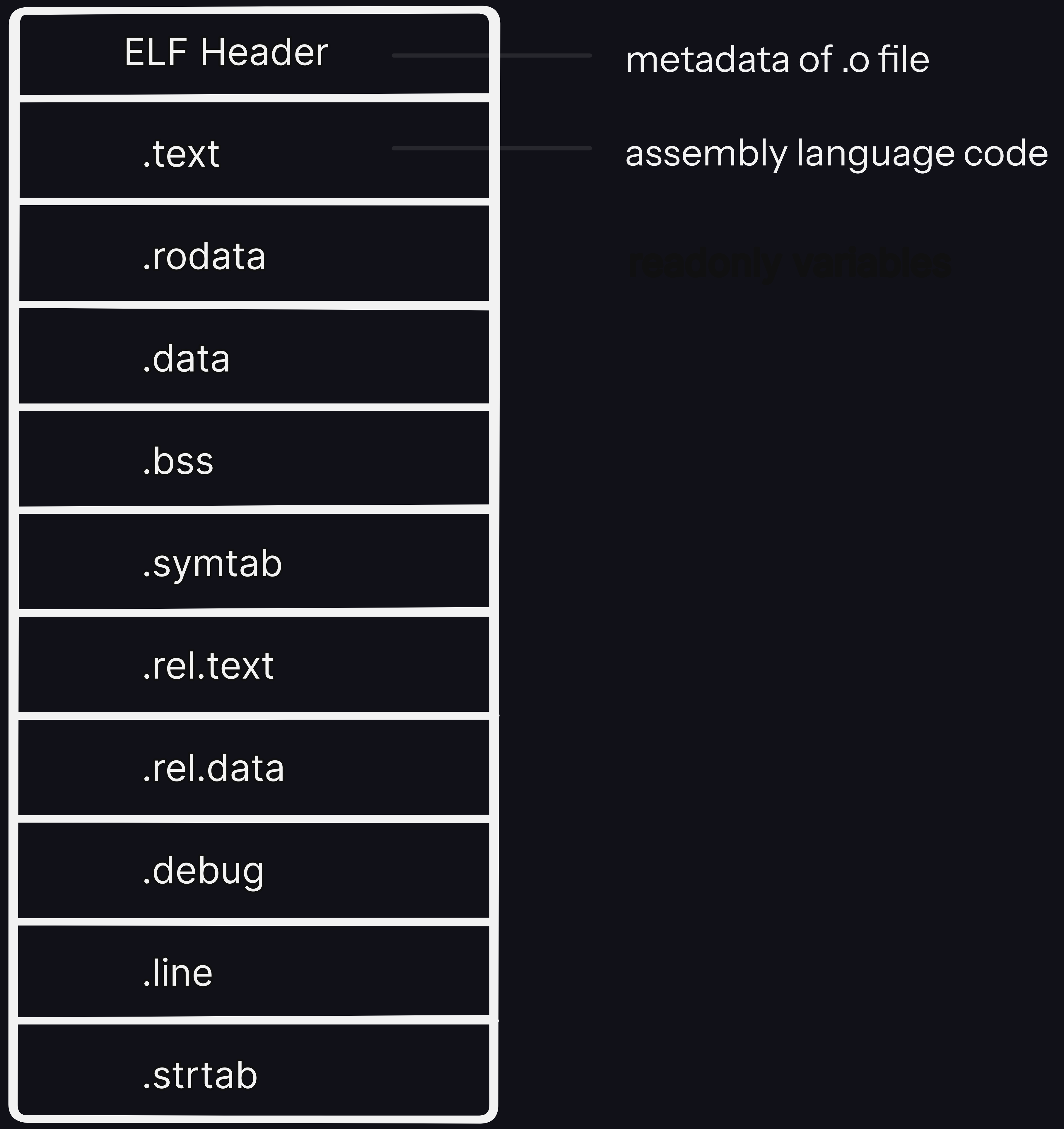
.debug

.line

.strtab

metadata of .o file

Inside **ELF** - **E**xecutable & **L**inkable **F**ormat



ELF Header — metadata of .o file

.text — assembly language code

.rodata — readonly variables

.data

.bss

.symtab

.rel.text

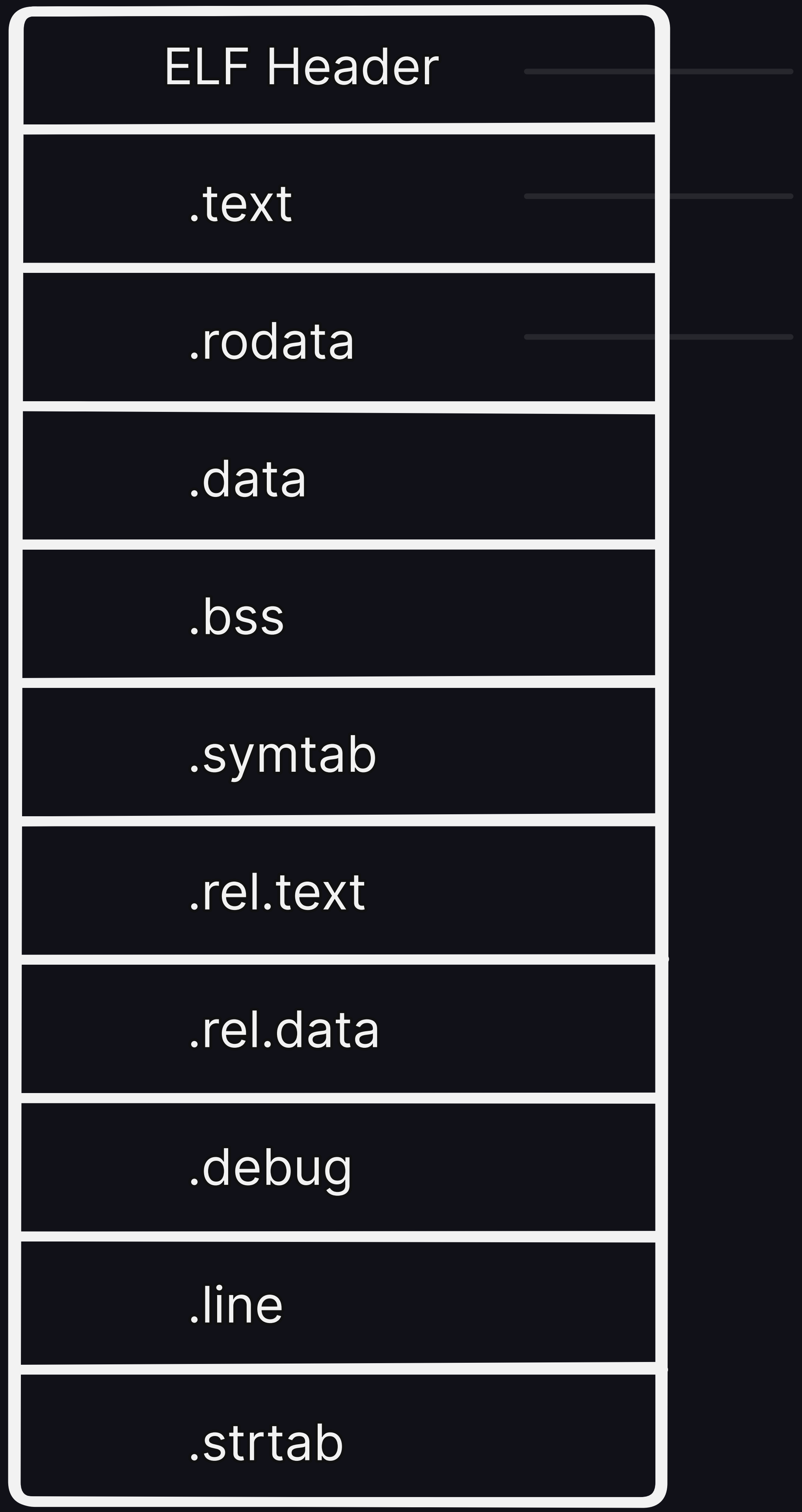
.rel.data

.debug

.line

.strtab

Inside ELF - Executable & Linkable Format



ELF Header

.text

.rodata

.data

.bss

.symtab

.rel.text

.rel.data

.debug

.line

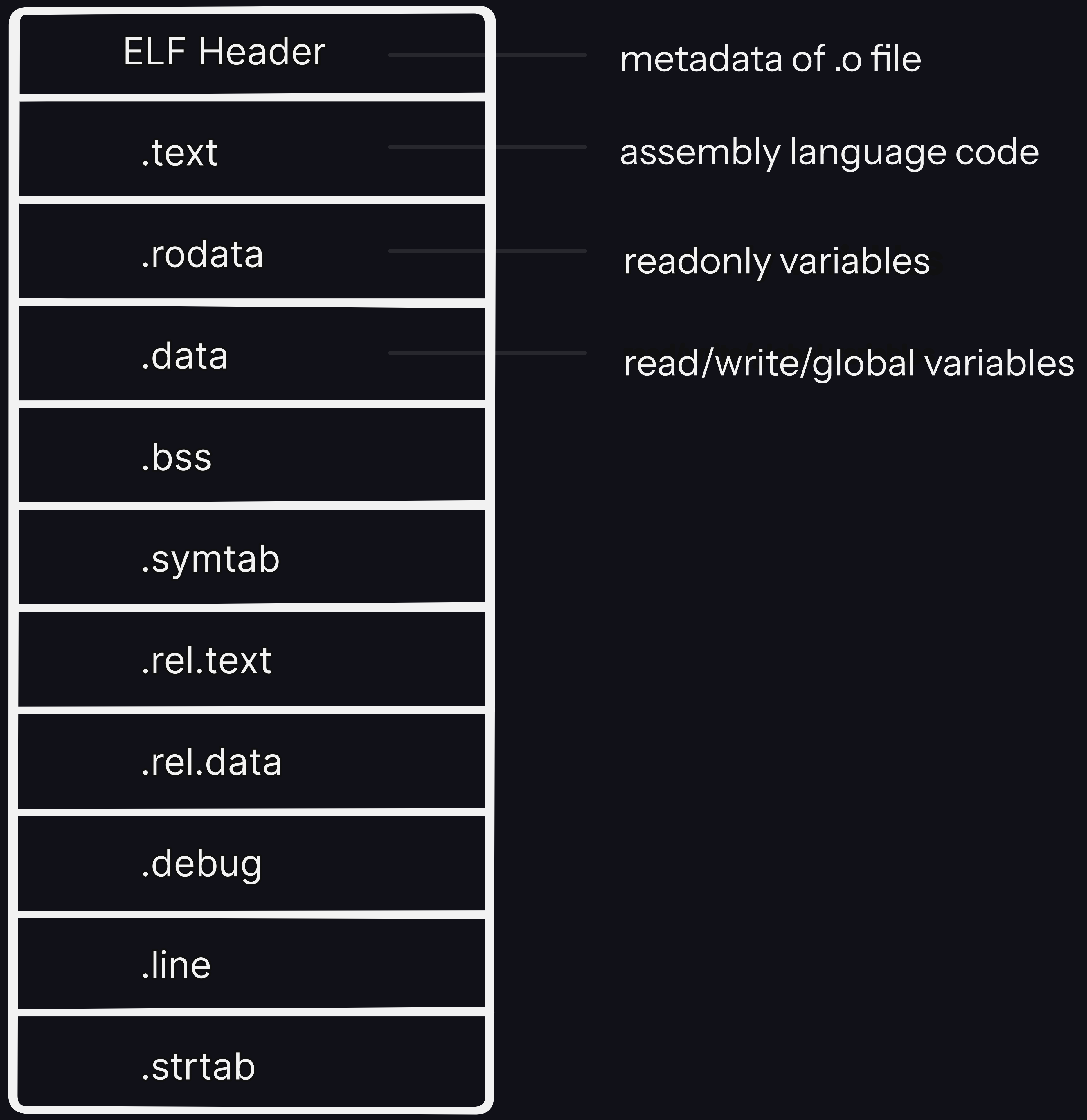
.strtab

metadata of .o file

assembly language code

readonly variables

Inside **ELF** - **E**xecutable & **L**inkable **F**ormat



ELF Header

metadata of .o file

.text

assembly language code

.rodata

readonly variables

.data

read/write/global variables

.bss

.symtab

.rel.text

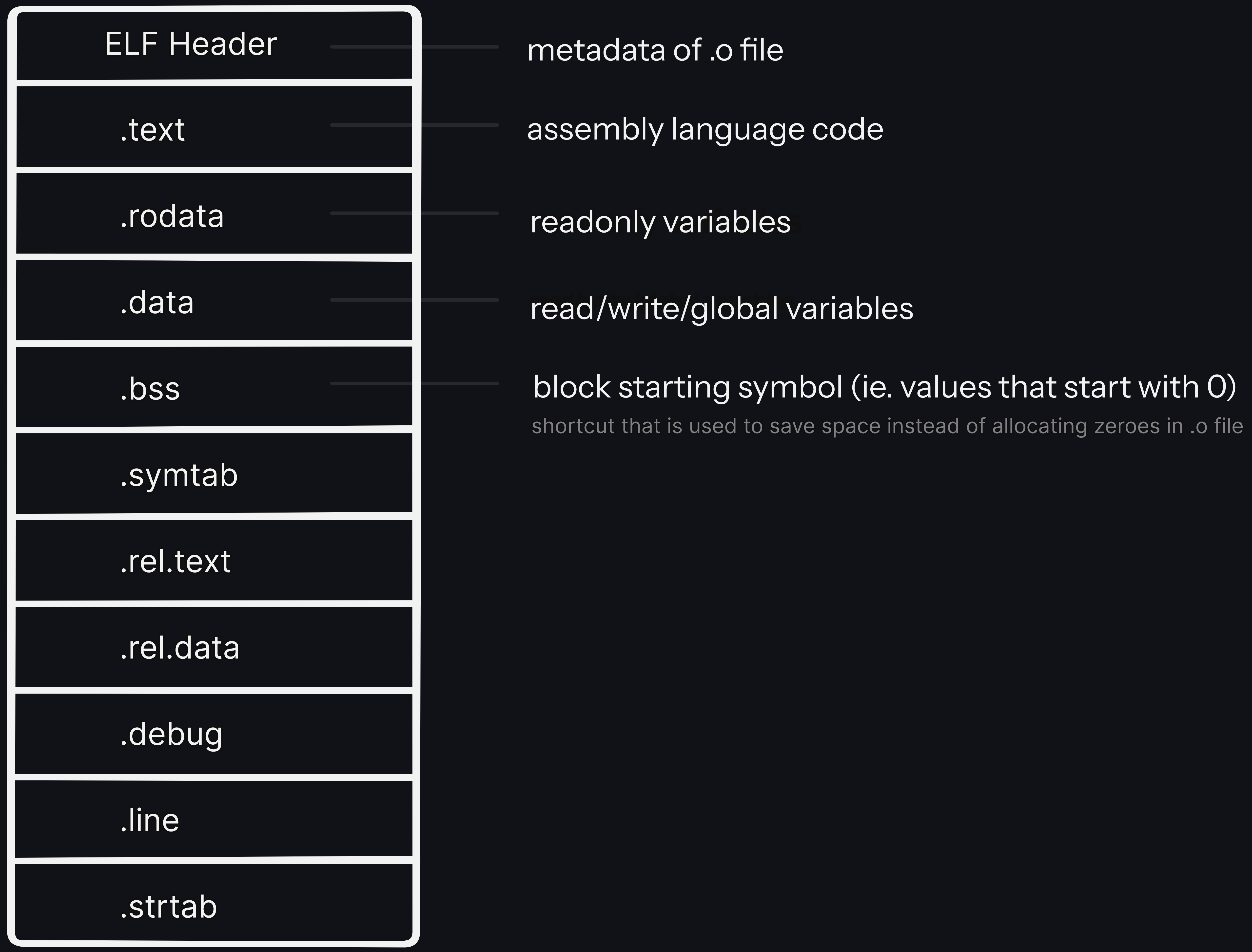
.rel.data

.debug

.line

.strtab

Inside **ELF** - **E**xecutable & **L**inkable **F**ormat



ELF Header — metadata of .o file

.text — assembly language code

.rodata — readonly variables

.data — read/write/global variables

.bss — block starting symbol (ie. values that start with 0)

shortcut that is used to save space instead of allocating zeroes in .o file

.symtab

.rel.text

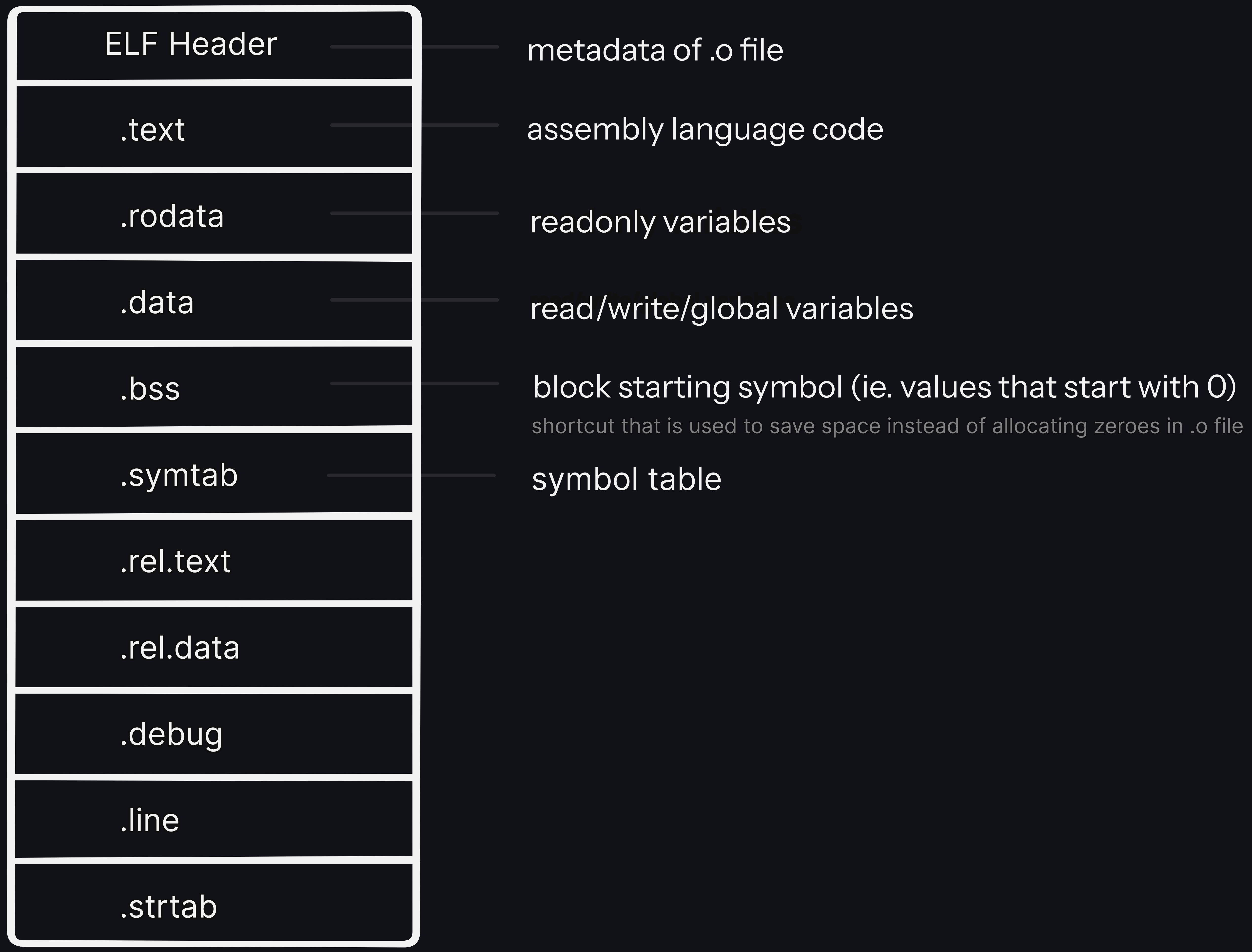
.rel.data

.debug

.line

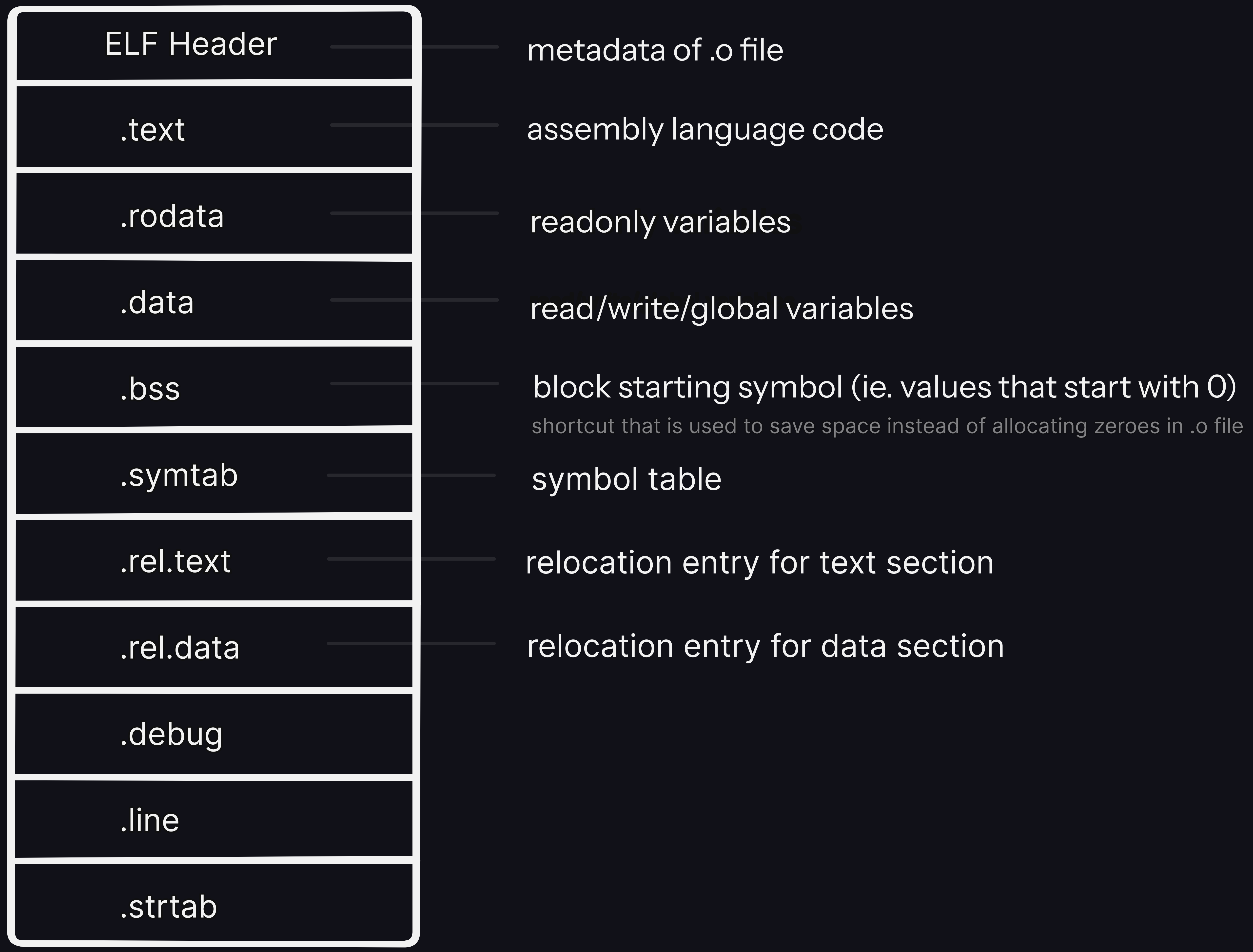
.strtab

Inside **ELF** - **E**xecutable & **L**inkable **F**ormat



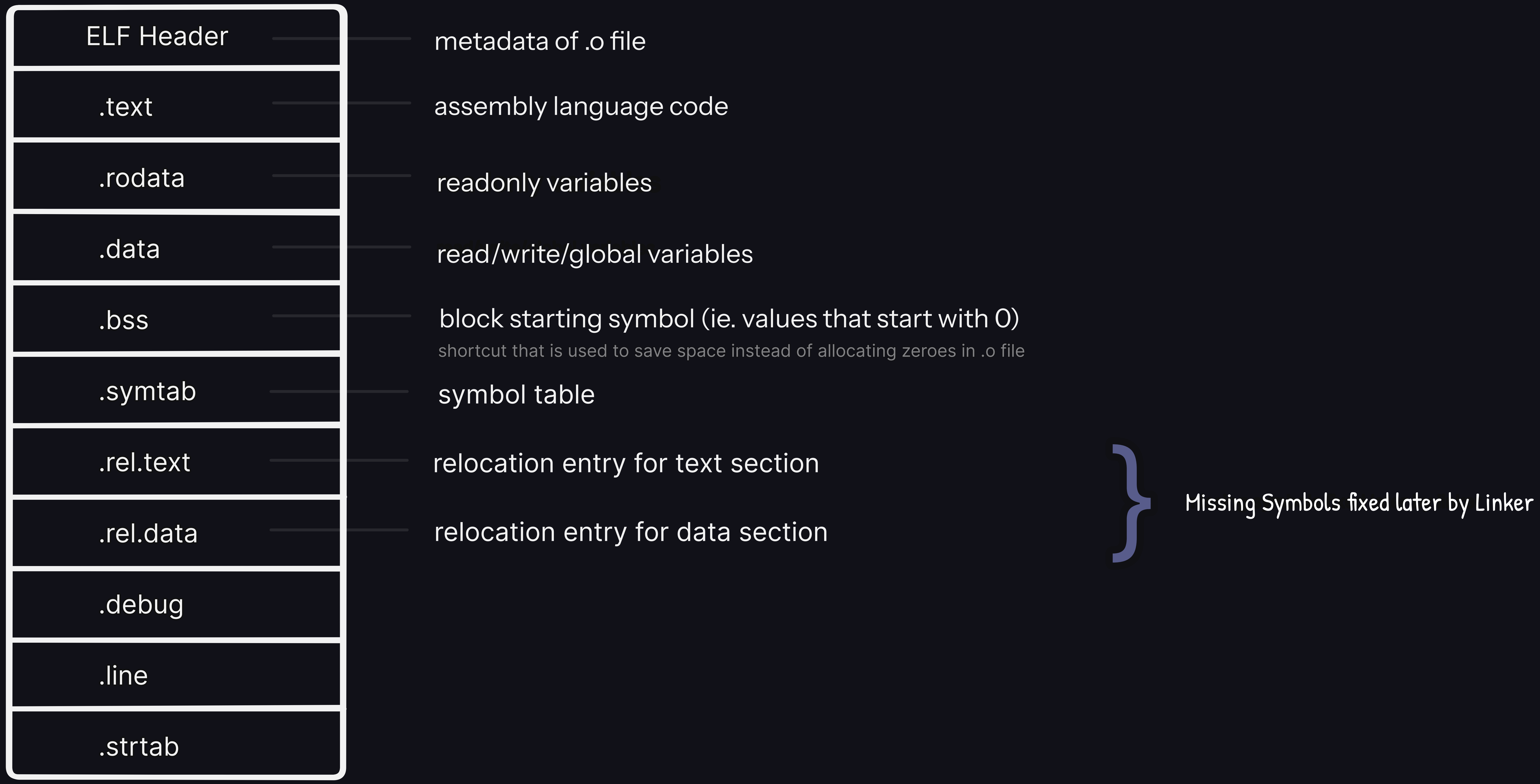
ELF Header	metadata of .o file
.text	assembly language code
.rodata	readonly variables
.data	read/write/global variables
.bss	block starting symbol (ie. values that start with 0) shortcut that is used to save space instead of allocating zeroes in .o file
.symtab	symbol table
.rel.text	
.rel.data	
.debug	
.line	
.strtab	

Inside **ELF - E**xecutable **& L**inkable **F**ormat



ELF Header	metadata of .o file
.text	assembly language code
.rodata	readonly variables
.data	read/write/global variables
.bss	block starting symbol (ie. values that start with 0) shortcut that is used to save space instead of allocating zeroes in .o file
.symtab	symbol table
.rel.text	relocation entry for text section
.rel.data	relocation entry for data section
.debug	
.line	
.strtab	

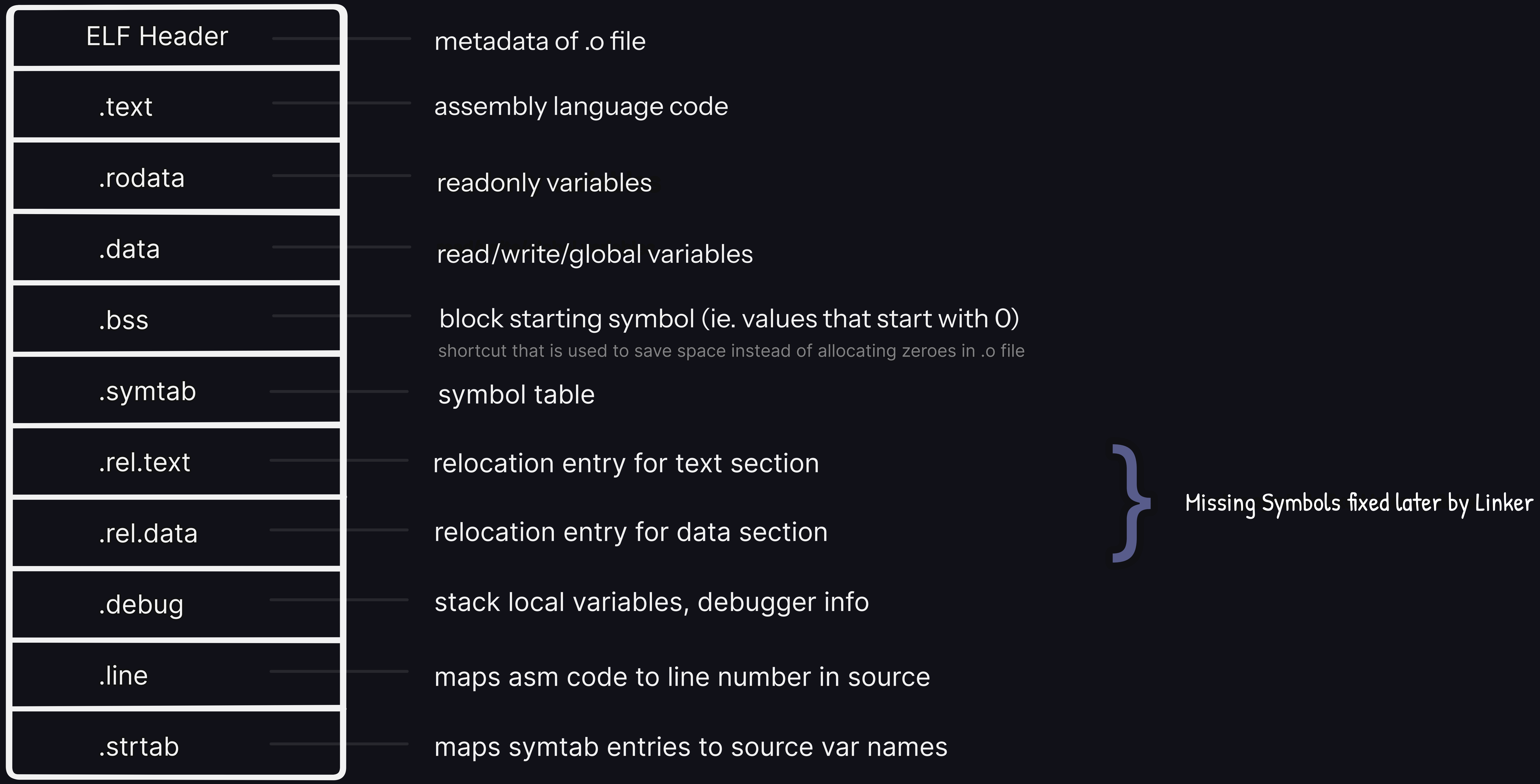
Inside **ELF** - **E**xecutable & **L**inkable **F**ormat



ELF Header	metadata of .o file
.text	assembly language code
.rodata	readonly variables
.data	read/write/global variables
.bss	block starting symbol (ie. values that start with 0) shortcut that is used to save space instead of allocating zeroes in .o file
.symtab	symbol table
.rel.text	relocation entry for text section
.rel.data	relocation entry for data section
.debug	
.line	
.strtab	

Missing Symbols fixed later by Linker

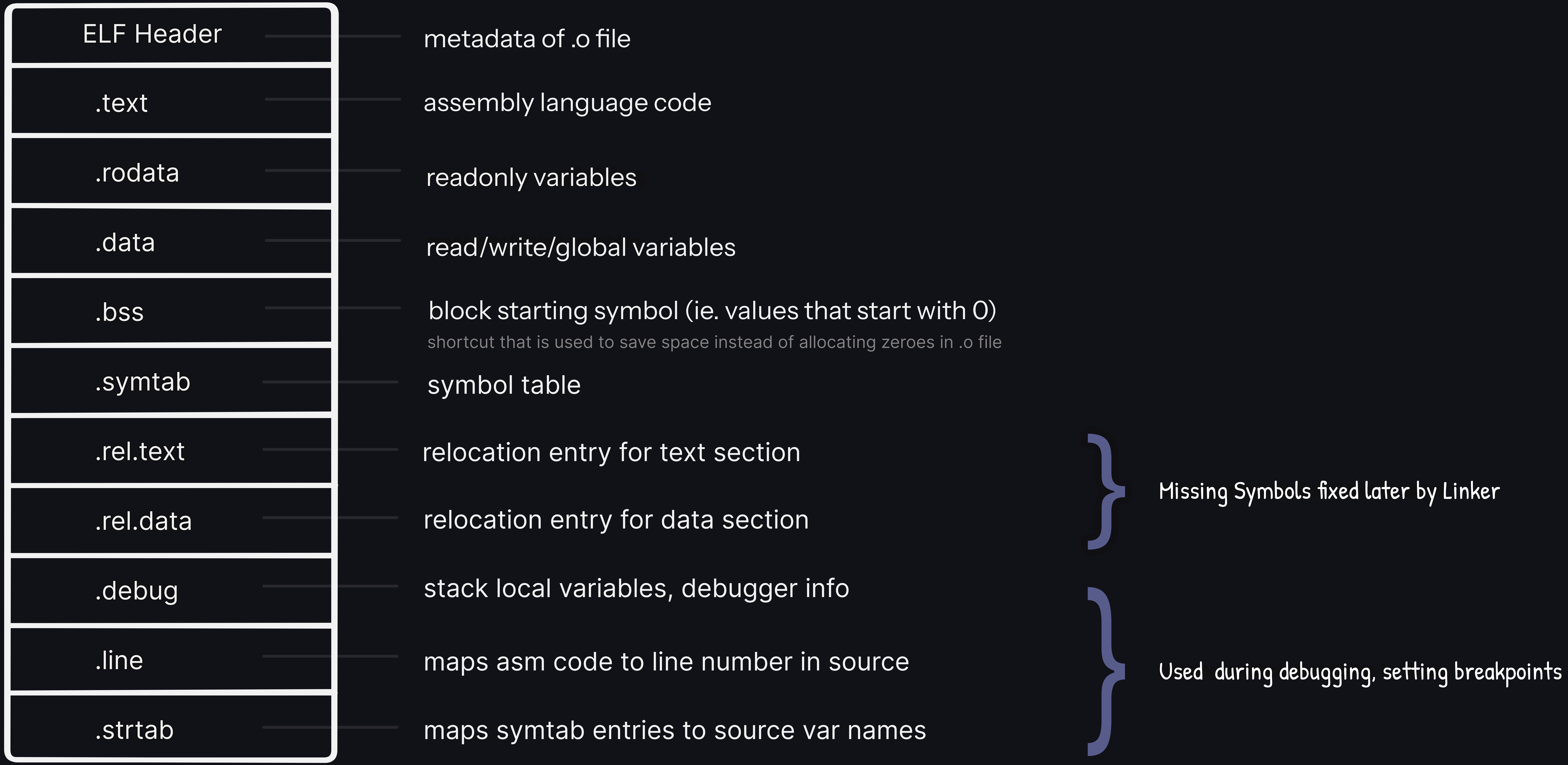
Inside **ELF** - **E**xecutable & **L**inkable **F**ormat



ELF Header	metadata of .o file
.text	assembly language code
.rodata	readonly variables
.data	read/write/global variables
.bss	block starting symbol (ie. values that start with 0) shortcut that is used to save space instead of allocating zeroes in .o file
.symtab	symbol table
.rel.text	relocation entry for text section
.rel.data	relocation entry for data section
.debug	stack local variables, debugger info
.line	maps asm code to line number in source
.strtab	maps symtab entries to source var names

Missing Symbols fixed later by Linker

Inside **ELF** - **E**xecutable & **L**inkable **F**ormat



ELF Header	metadata of .o file
.text	assembly language code
.rodata	readonly variables
.data	read/write/global variables
.bss	block starting symbol (ie. values that start with 0) shortcut that is used to save space instead of allocating zeroes in .o file
.symtab	symbol table
.rel.text	relocation entry for text section
.rel.data	relocation entry for data section
.debug	stack local variables, debugger info
.line	maps asm code to line number in source
.strtab	maps symtab entries to source var names

.rel.text, .rel.data: Missing Symbols fixed later by Linker

.debug, .line, .strtab: Used during debugging, setting breakpoints

Lets make a `main.rs` that calls the `foo` and `bar` functions.

main.rs

```
extern "C" {
    fn foo();
    fn bar();
    static mut Global: i32;
}

fn main() {
    unsafe {
        foo();
        bar();
        println!("Global: {}", Global);
    }
}
```

Lets make a `main.rs` that calls the `foo` and `bar` functions.

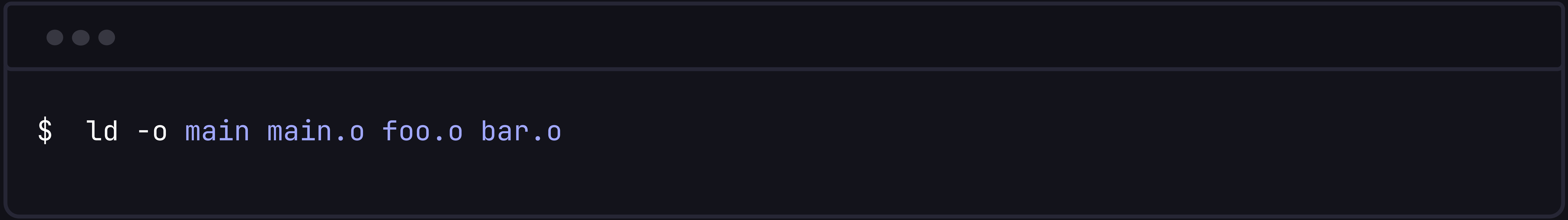
main.rs

```rust
extern "C" {
    fn foo();
    fn bar();
    static mut Global: i32;
}

fn main() {
    unsafe {
        foo();
        bar();
        println!("Global: {}", Global);
    }
}
```

Let's compile the `main.rs` file and emit an object file like before:

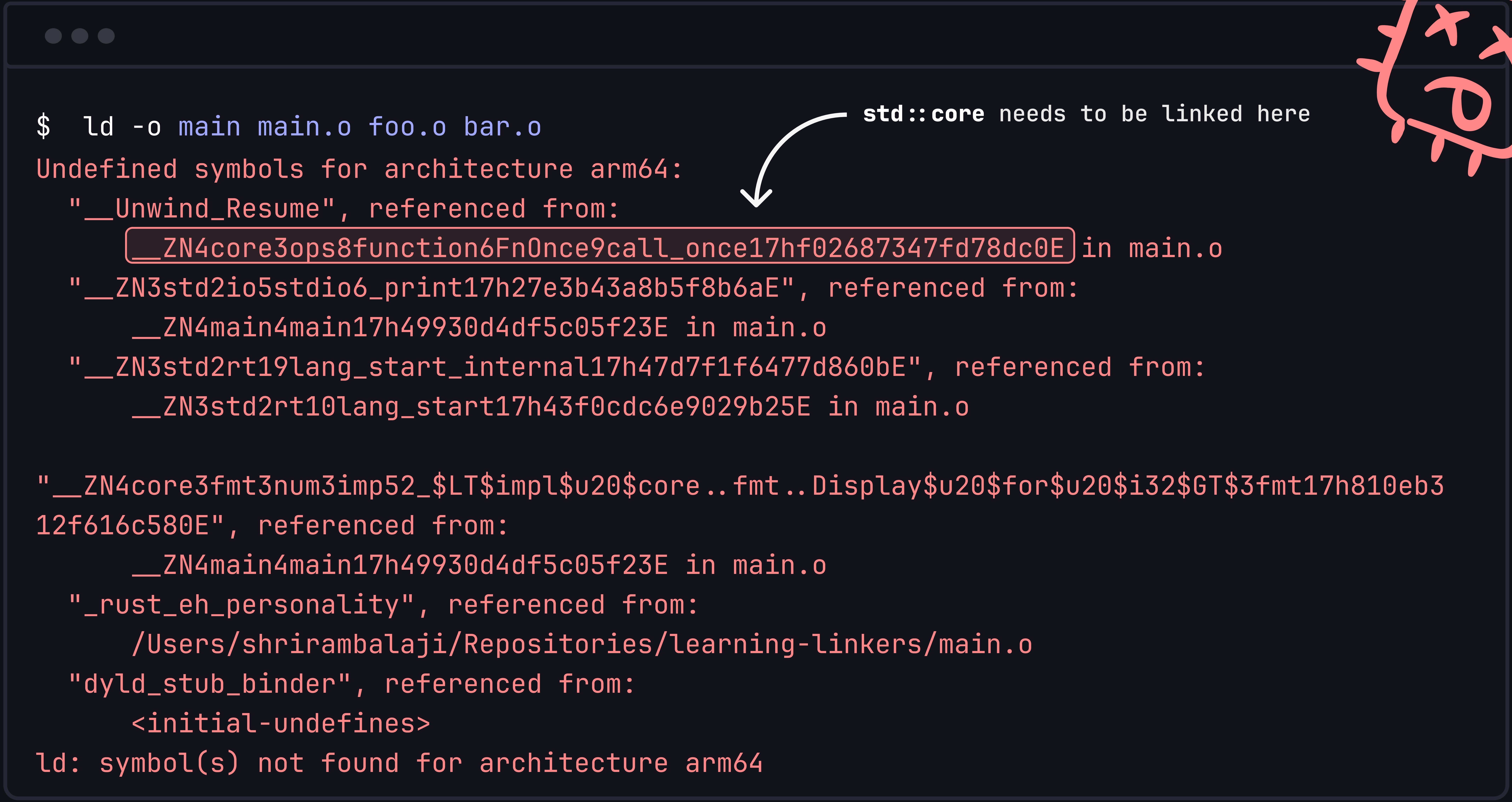
```
$ rustc --emit=obj -o main.o main.rs
```

Linking object files emitted from `rustc` using `ld`



```
$  ld -o main main.o foo.o bar.o
```

Linking object files emitted from `rustc` using `ld`

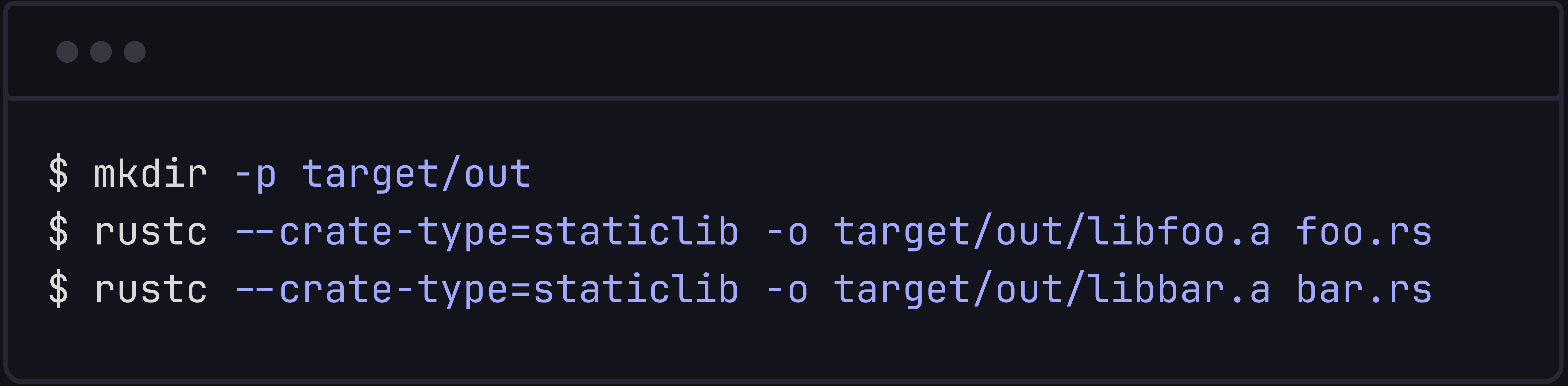


```
$  ld -o main main.o foo.o bar.o
Undefined symbols for architecture arm64:
  "__Unwind_Resume", referenced from:
      __ZN4core3ops8function6FnOnce9call_once17hf02687347fd78dc0E in main.o
  "__ZN3std2io5stdio6_print17h27e3b43a8b5f8b6aE", referenced from:
      __ZN4main4main17h49930d4df5c05f23E in main.o
  "__ZN3std2rt19lang_start_internal17h47d7f1f6477d860bE", referenced from:
      __ZN3std2rt10lang_start17h43f0cdc6e9029b25E in main.o

"__ZN4core3fmt3num3imp52_$LT$impl$u20$core..fmt..Display$u20$for$u20$i32$GT$3fmt17h810eb3
12f616c580E", referenced from:
      __ZN4main4main17h49930d4df5c05f23E in main.o
  "_rust_eh_personality", referenced from:
      /Users/shrirambalaji/Repositories/learning-linkers/main.o
  "dyld_stub_binder", referenced from:
      <initial-undefines>
ld: symbol(s) not found for architecture arm64
```

std::core needs to be linked here

`staticlib` to the rescue

Instead of trying to link the core crate and bring in std dependencies ourselves, we can create a **static library** using `--crate-type=staticlib` from `foo.rs` and `bar.rs`:



```
$ mkdir -p target/out
$ rustc --crate-type=staticlib -o target/out/libfoo.a foo.rs
$ rustc --crate-type=staticlib -o target/out/libbar.a bar.rs
```

The output is a `.a file`, which is a static library / archive in *nix systems. and it contains the .o files we saw previously.

`staticlib` to the rescue

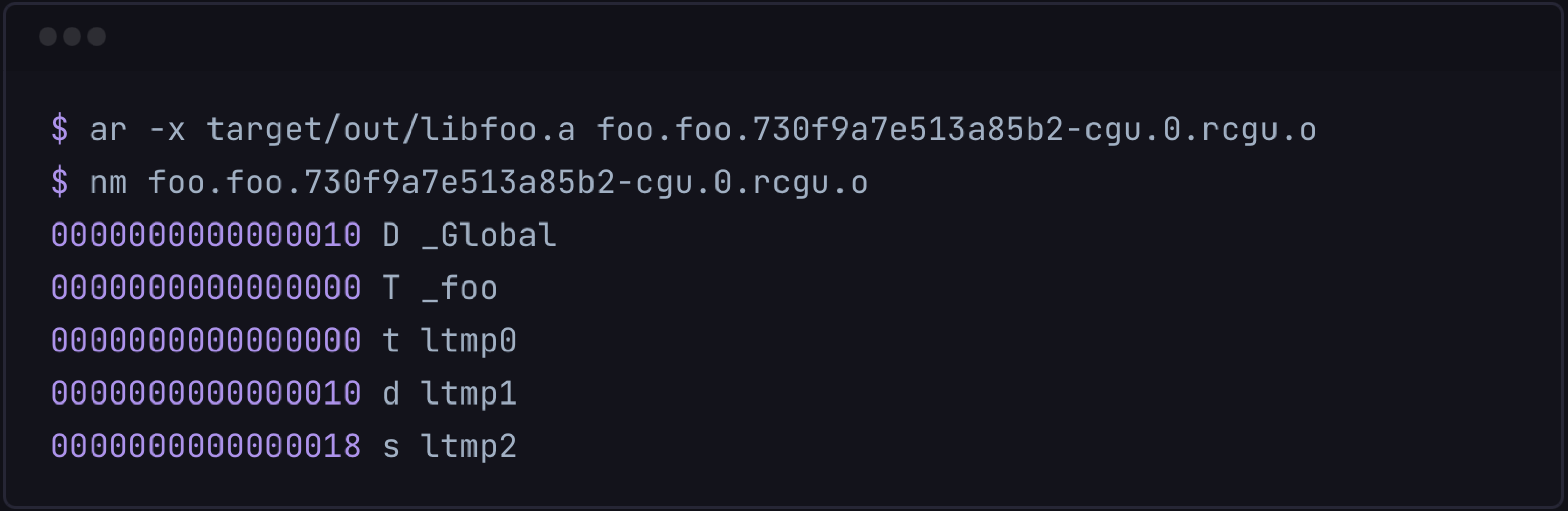
We can use the `ar` command to list the contents of the archive.

```
$ ar -t target/out/libfoo.a | grep foo
foo.foo.730f9a7e513a85b2-cgu.0.rcgu.o
foo.10ftosr6tvdwscdu.rcgu.o
```

Interestingly the `.a` file contains the `.o` files we saw earlier, but with a different name, specifically with `*.rcgu.o` suffix. The `rcgu` stands for “Rust Codegen Unit” and is a unit of code that the compiler generates during Code Generation phase.

`staticlib` to the rescue

If we extract the `.o` file and look, we can see the same symbols we saw earlier.



```
$ ar -x target/out/libfoo.a foo.foo.730f9a7e513a85b2-cgu.0.rcgu.o
$ nm foo.foo.730f9a7e513a85b2-cgu.0.rcgu.o
0000000000000010 D _Global
0000000000000000 T _foo
0000000000000000 t ltmp0
0000000000000010 d ltmp1
0000000000000018 s ltmp2
```

Doing things the rust way - `cargo’s` back!

Until now, we ignored poor cargo and were relying on rustc.

Ideally, we should leverage cargo as its meant to be

`cargo` build script

We can add a build script in a **`build.rs`** that goes in the project's root. This will link the static libraries from the previous step together.

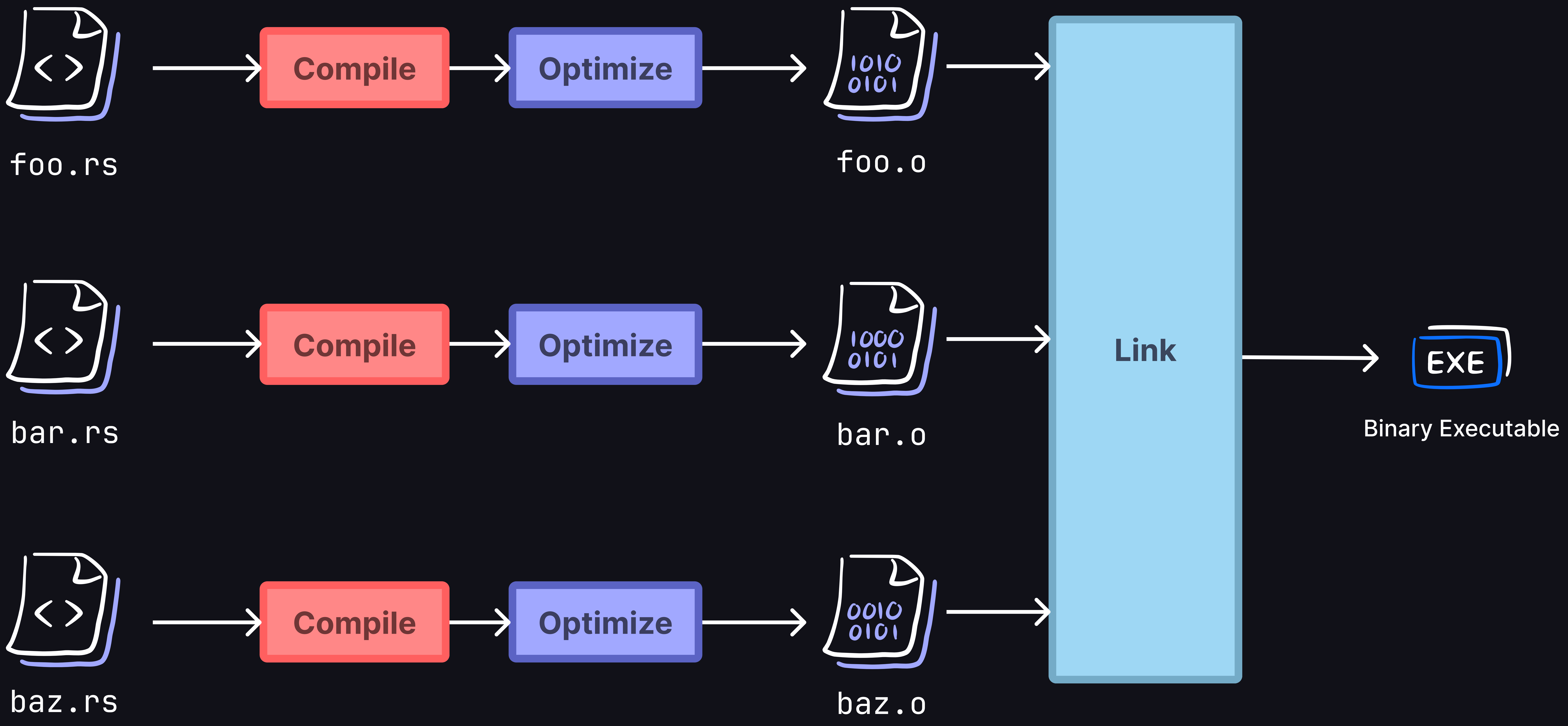
build.rs

```rust
fn main() {
    println!("cargo:rustc-link-search=native=target/out");
    println!("cargo:rustc-link-lib=static=foo");
    println!("cargo:rustc-link-lib=static=bar");
}
```

- **`cargo:rustc-link-search=native=target/out`** instruction tells the compiler to search for the static libraries in the **`target/out`** directory
- **`cargo:rustc-link-lib=static=foo`** and **`cargo:rustc-link-lib=static=bar`** tells the compiler to link the **`foo`** and **`bar`** static libraries. As an alternative to the linking these in the build script, we can also use the **`#[link](https://doc.rust-lang.org/reference/items/external-blocks.html#the-link-attribute)`** attribute directly in **`main.rs`**

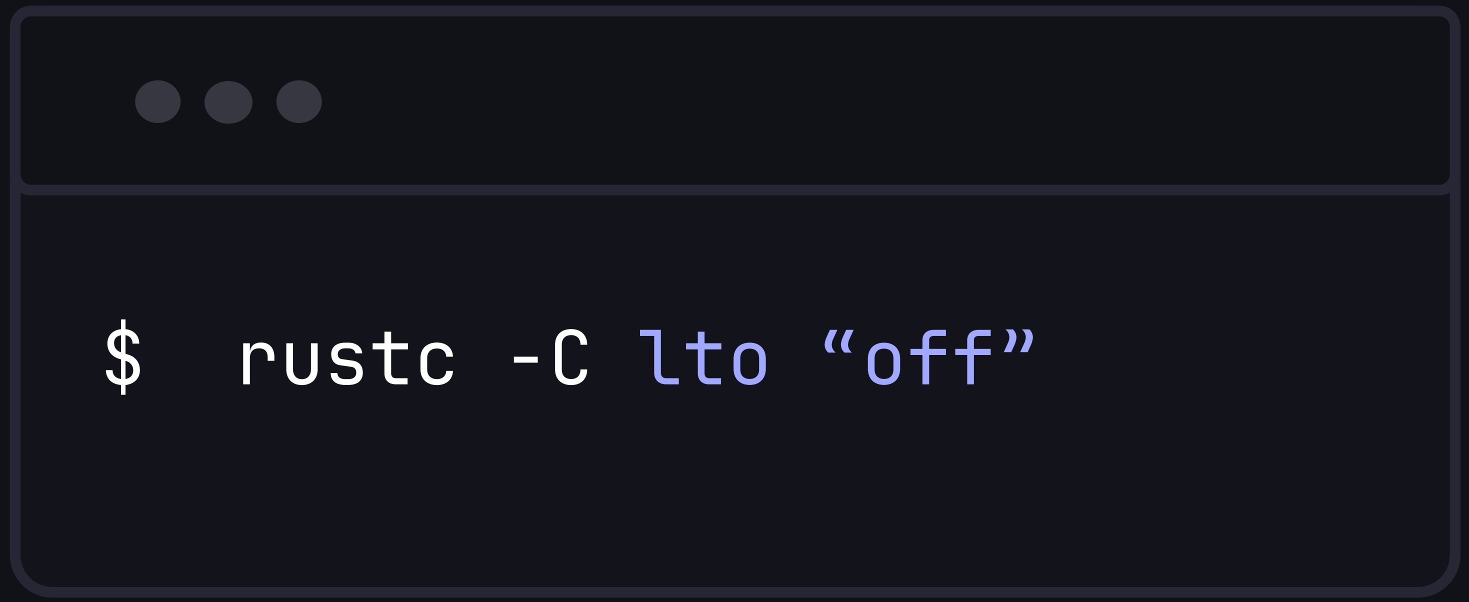
Link **T**ime **O**ptimization

No LTO



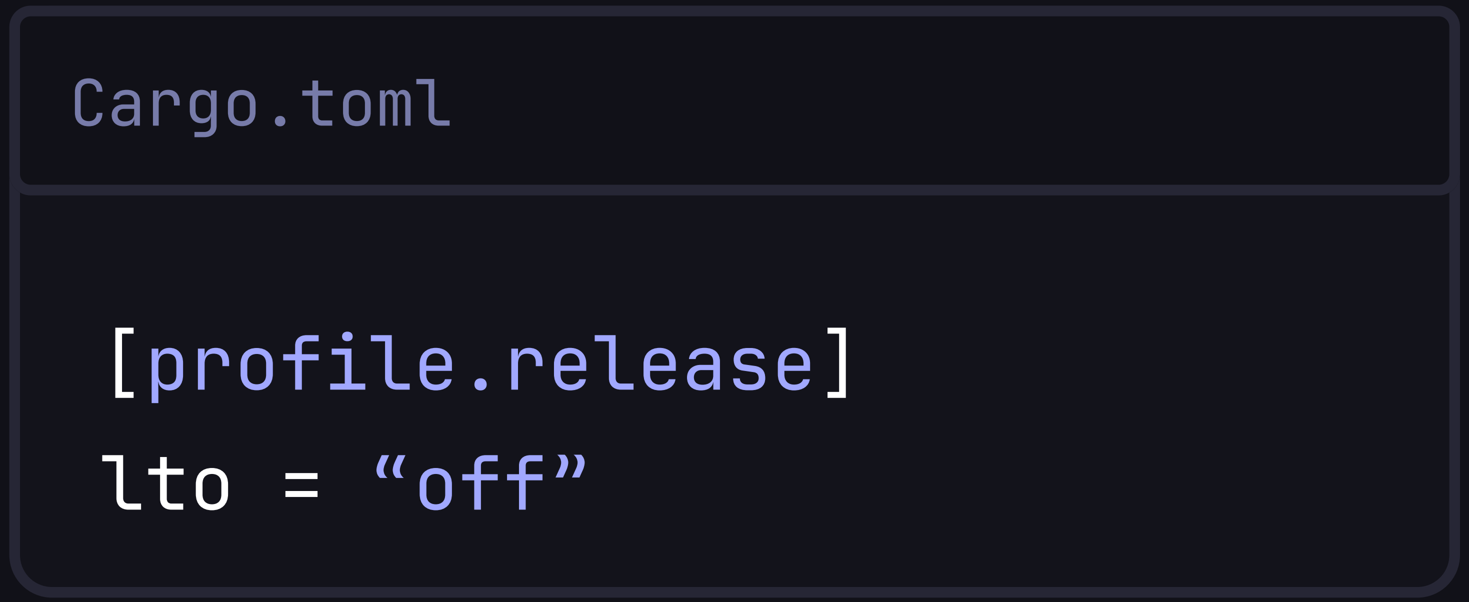
foo.rs
Compile
Optimize
1010
0101
foo.o
bar.rs
Compile
Optimize
1000
0101
bar.o
baz.rs
Compile
Optimize
0010
0101
baz.o
Link
EXE
Binary Executable

No LTO



```
$  rustc -C lto “off”
```

`lto` command-line argument



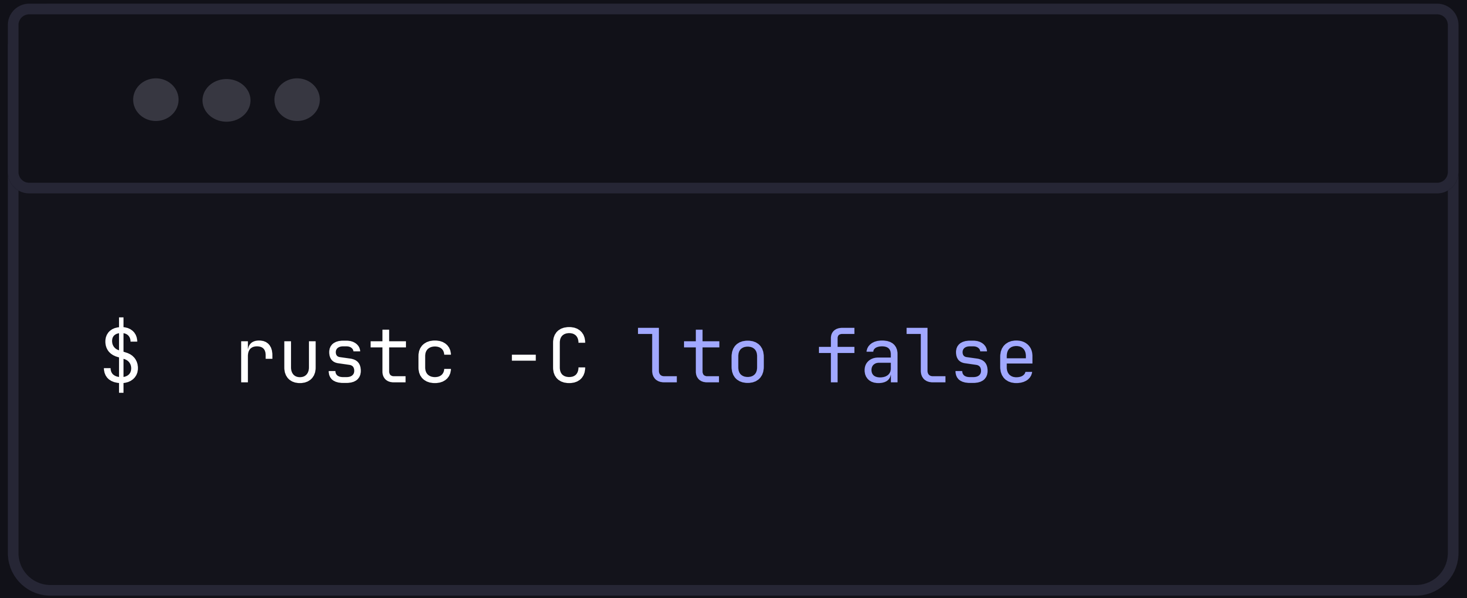
Cargo.toml

```
[profile.release]
lto = “off”
```

`lto` profile setting in `Cargo.toml`

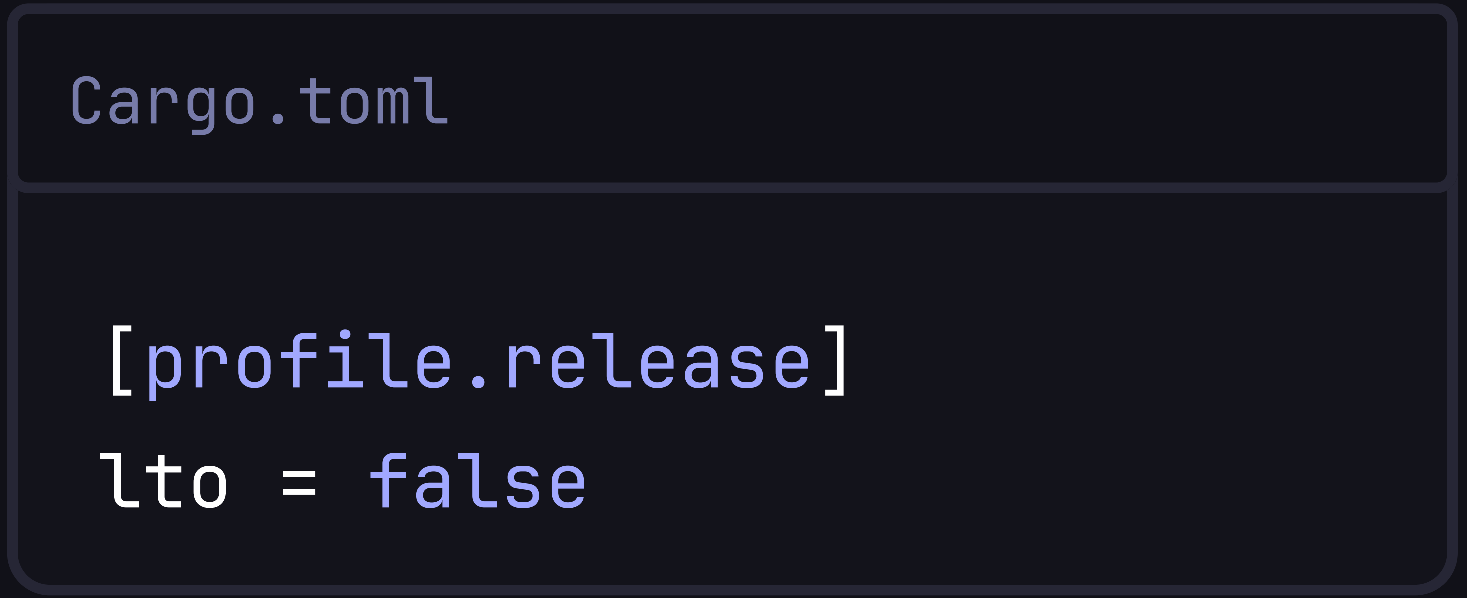
n, no, off are acceptable values for disabling LTO

No LTO



```
$  rustc -C lto false
```

this turns off LTO as expected



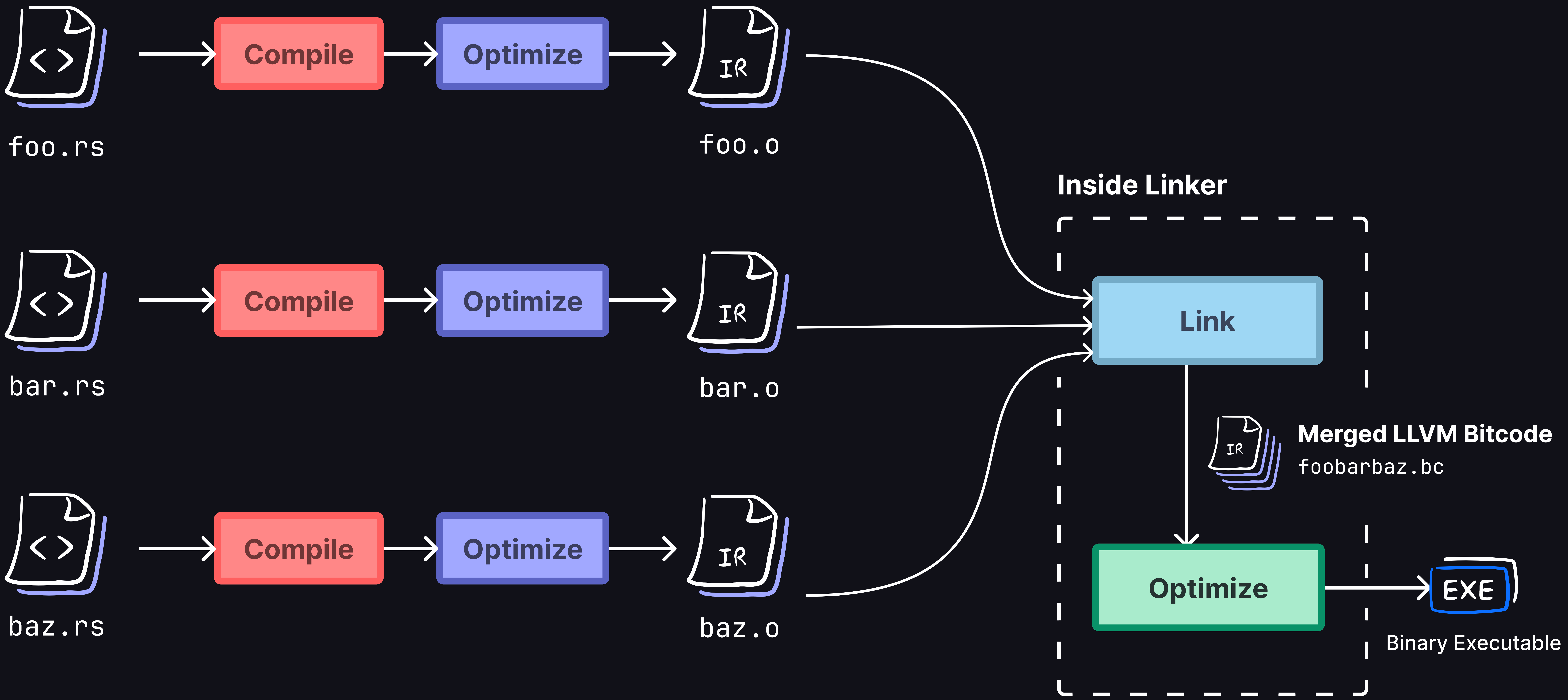
Cargo.toml

```toml
[profile.release]
lto = false
```

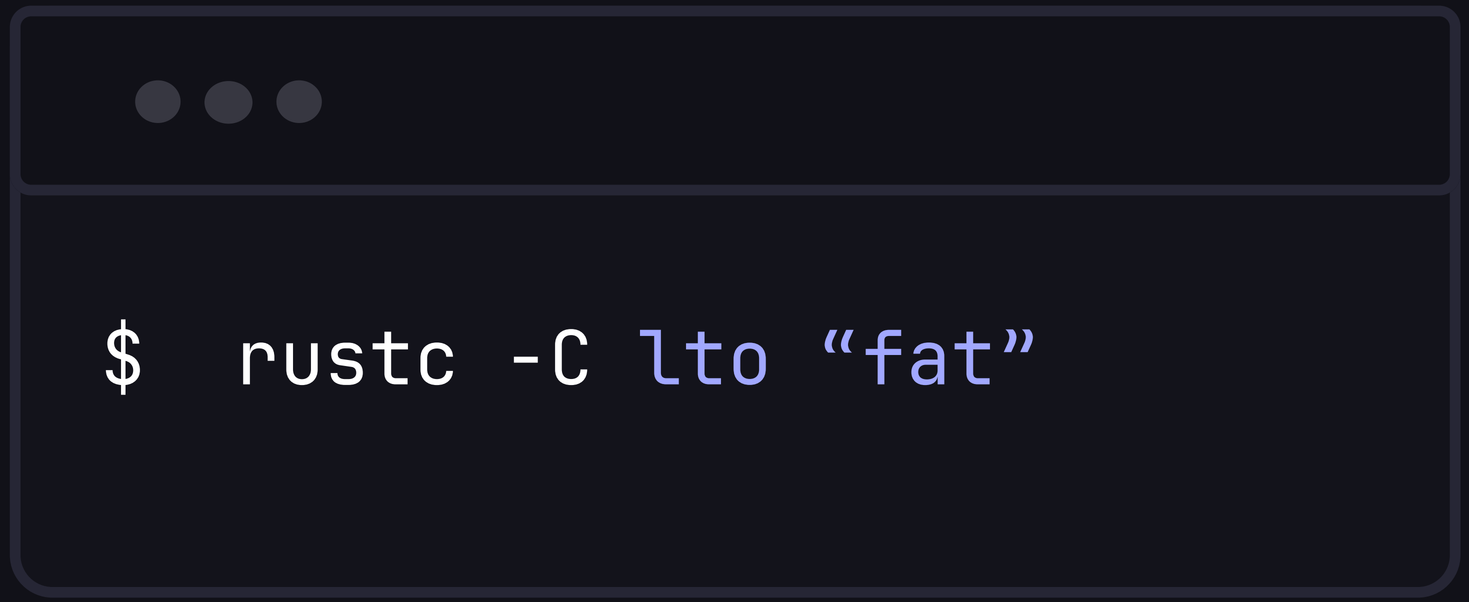
but it does thin-local LTO when set to false in Cargo profiles

false has a peculiar behaviour

Fat / Full LTO

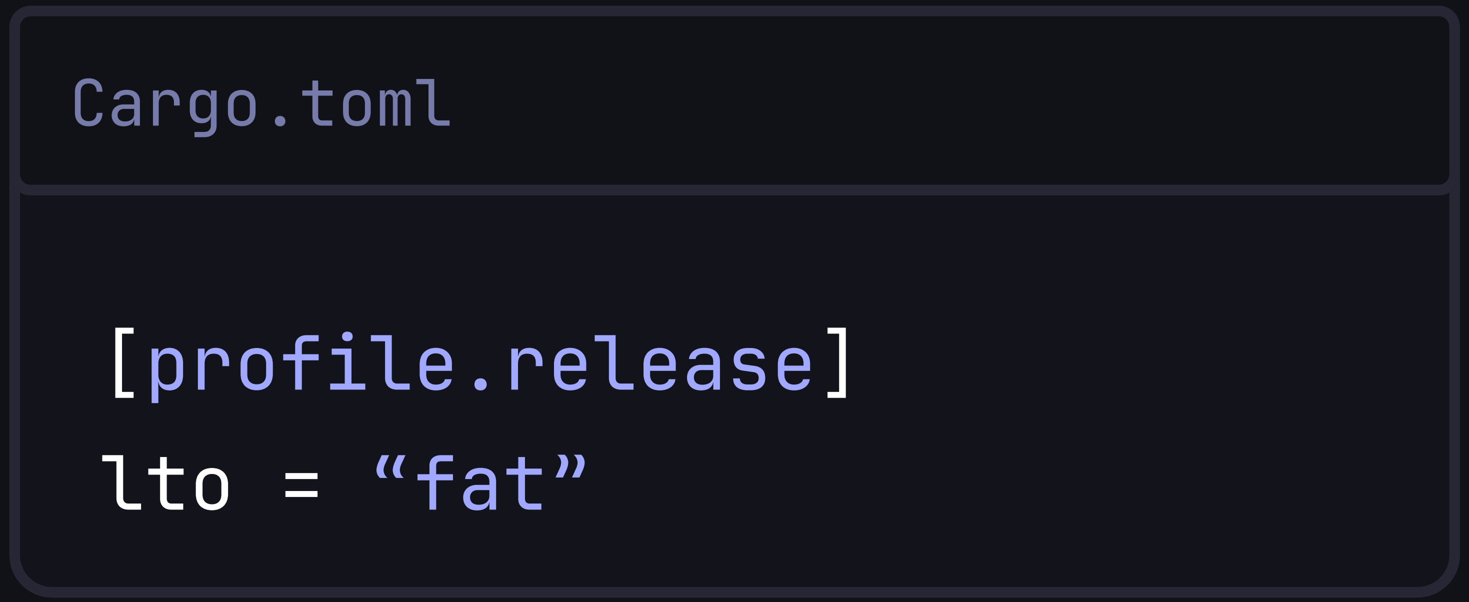

foo.rs
Compile
Optimize
IR
foo.o
bar.rs
Compile
Optimize
IR
bar.o
baz.rs
Compile
Optimize
IR
baz.o
Inside Linker
Link
IR
Merged LLVM Bitcode
foobarbaz.bc
Optimize
EXE
Binary Executable

Fat / Full LTO



```
$  rustc -C lto “fat”
```

`lto` command-line argument



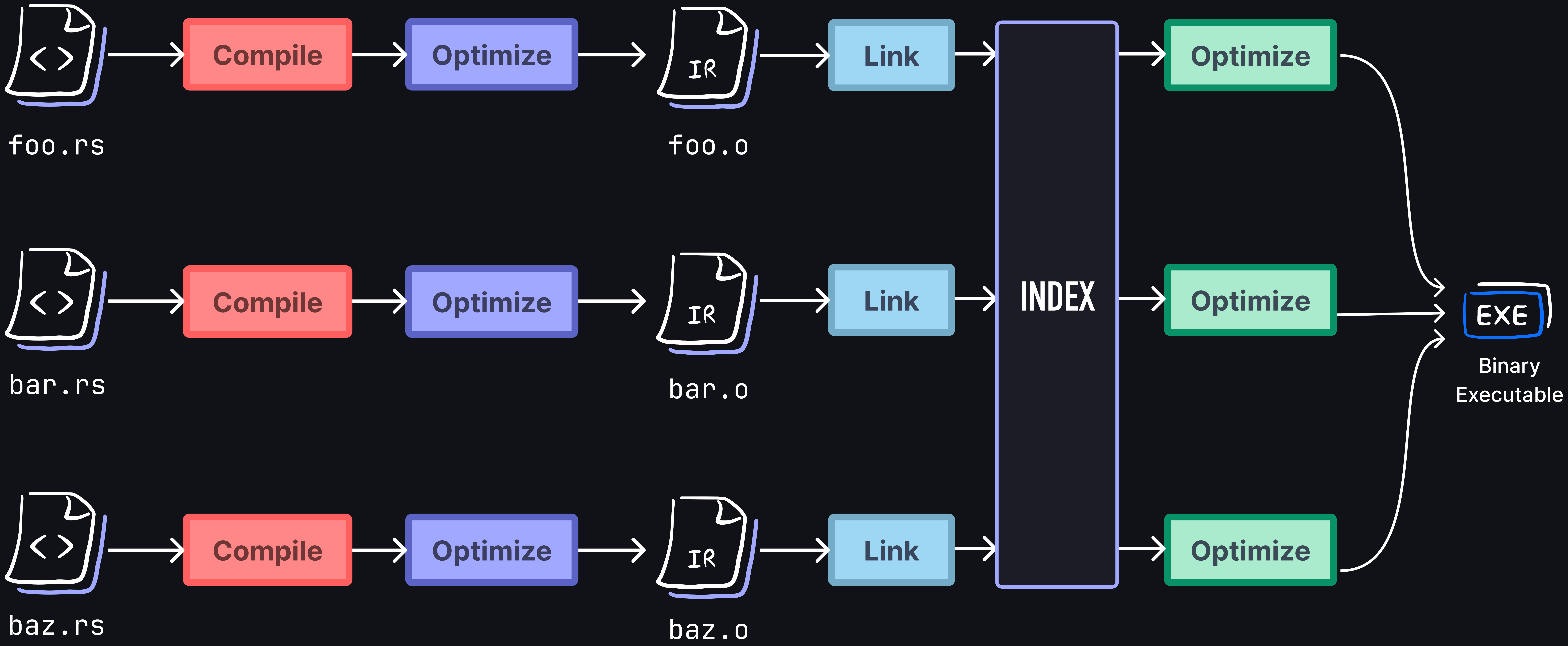
Cargo.toml

```
[profile.release]
lto = “fat”
```

`lto` profile setting in `Cargo.toml`

y, yes, on, true, fat are acceptable values for enabling “fat” LTO

Thin / Parallel LTO



Thin / Parallel LTO

```
$  rustc -C lto “thin”
```

lto command-line argument

Cargo.toml

```
[profile.release]
lto = “thin”
```

lto profile setting in Cargo.toml

Thin-Local LTO

By default, rustc splits a crate into multiple "codegen units for parallel processing by LLVM. However, this prevents some optimizations as code is separated into different codegen units, and is handled independently.

Thin-local LTO will perform thin LTO across the codegen units within a single “local” crate, bringing back some optimizations that would otherwise be lost by the separation. This is the default setting in release profile.

Thin-Local LTO

Cargo.toml

```toml
[profile.release]
lto = false
```

when `lto = false` or when `-C lto` is not specified

STATIC LINKING

All the necessary dependencies are compiled and linked in the final executable-binary **statically**. This enables easier distribution, but the tradeoff being bigger executables.

DYNAMIC LINKING

Dynamic linking allows a program to load external libraries / shared libraries into memory and use their functionalities at runtime, rather than at compile time.

Rust statically links everything including crates by default, except `libc`

But why can’t Rust always link dynamically?

Rust doesn’t have a stable ABI

What does having a stable ABI have to do with linking?

ABI, or **A**pplication **B**inary **I**nterface, specifies data layout in memory and function call mechanics. Function calls involve complexities like register protection and argument passing order, defined by the "calling convention."

In Rust, if you don't specify a representation with `#[repr(_)]` or a calling convention with `extern "_"`, the compiler can optimize these aspects variably, influenced by compiler version and optimization level. This variability poses issues with dynamic linkage, as differing compiler calls may lead to ABI disagreements between software units, complicating linking.

Taking a “Stabby”

Taking a “Stabby”

Stabby is a library that aims to address these challenges by assisting in pinning the ABI for a portion of your program, while preserving some of the layout optimizations provided by rustc's unstable ABI. Additionally, Stabby enables you to mark function exports and imports to validate your dependency versioning for types within `stabby::abi::IStable`.

Taking a “Stabby”

When you annotate structs with `#[stabby::stabby]`, two things happen:

- The struct becomes `#[repr(C)]`. Unless you specify otherwise or your struct has generic fields, stabby will assert that you haven't ordered your fields in a suboptimal manner at compile time.

- `stabby::abi::IStable` will be implemented for your type. It represents the layout (including niches) through associated types. This is key to being able to provide niche-optimization in enums

Taking a “Stabby”

When you annotate structs with `#[stabby::stabby]`, two things happen:

- The struct becomes `#[repr(C)]`. Unless you specify otherwise or your struct has generic fields, stabby will assert that you haven't ordered your fields in a suboptimal manner at compile time.

- `stabby::abi::IStable` will be implemented for your type. It represents the layout (including niches) through associated types. This is key to being able to provide niche-optimization in enums

Taking a “Stabby”

```
#[stabby::import(name = "library")]
extern "C" {
    pub fn stable_fn(v: u8) -> stabby::option::Option<()>;
}

fn main() {
    stable_fn(5);
}
```

LINKS

References

- Blog
- Slides
- Code Snippets on Github
- CS 361 Systems Programming by Chris Kanich
- High Level Compiler Architecture - Rustc Guide
- Rust Borrow Checker - Nell Shamrell-Harrington
- Linkage - Rust Reference
- Visualizing Rust Compilation
- Freestanding Rust Binary - Philipp Oppermann
- Matt Godbolt - The Bits between the Bits
- Link Time Optimization by Ryan Stinnett

Thank You

@shrirambalaji