Dive into the Rust Compiler

崔漠寒 2022.05.17

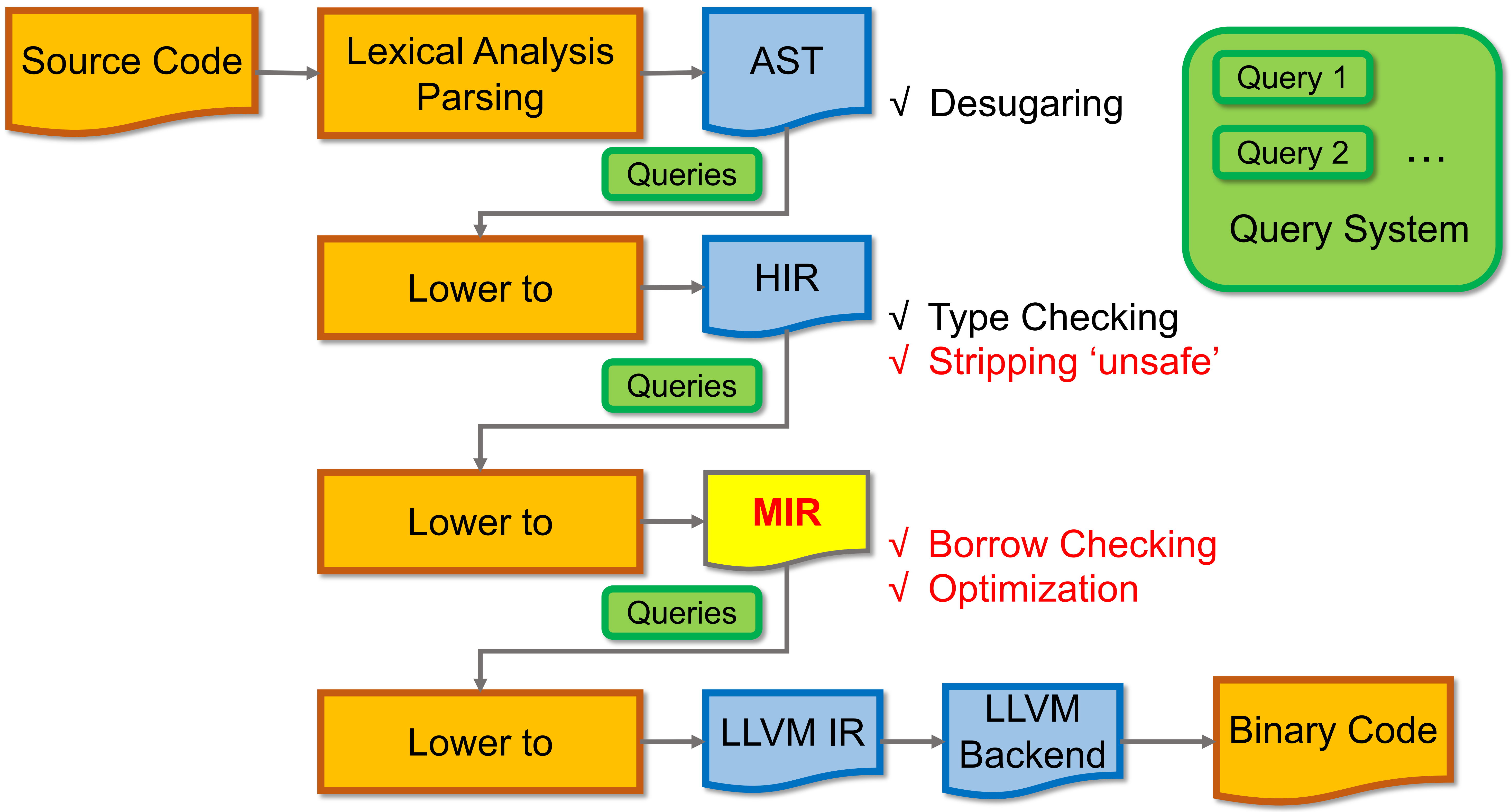
Content

Diving into Rust Compiler

- CH1 High-level Compiler Architecture
- CH2 Source Code Representation

High-level Compiler Architecture

Overview: Framework



Queries: demand-driven compilation

- Progress:
  - still transitioning from a **traditional "pass-based"** setup to a **"demand-driven" system**
- Idea: simple
  - ❎ instead of entirely **independent passes** (parsing, type-checking, etc.)
  - ✅ a set of **function-like *queries*** compute information about the input source

Queries: Memorized

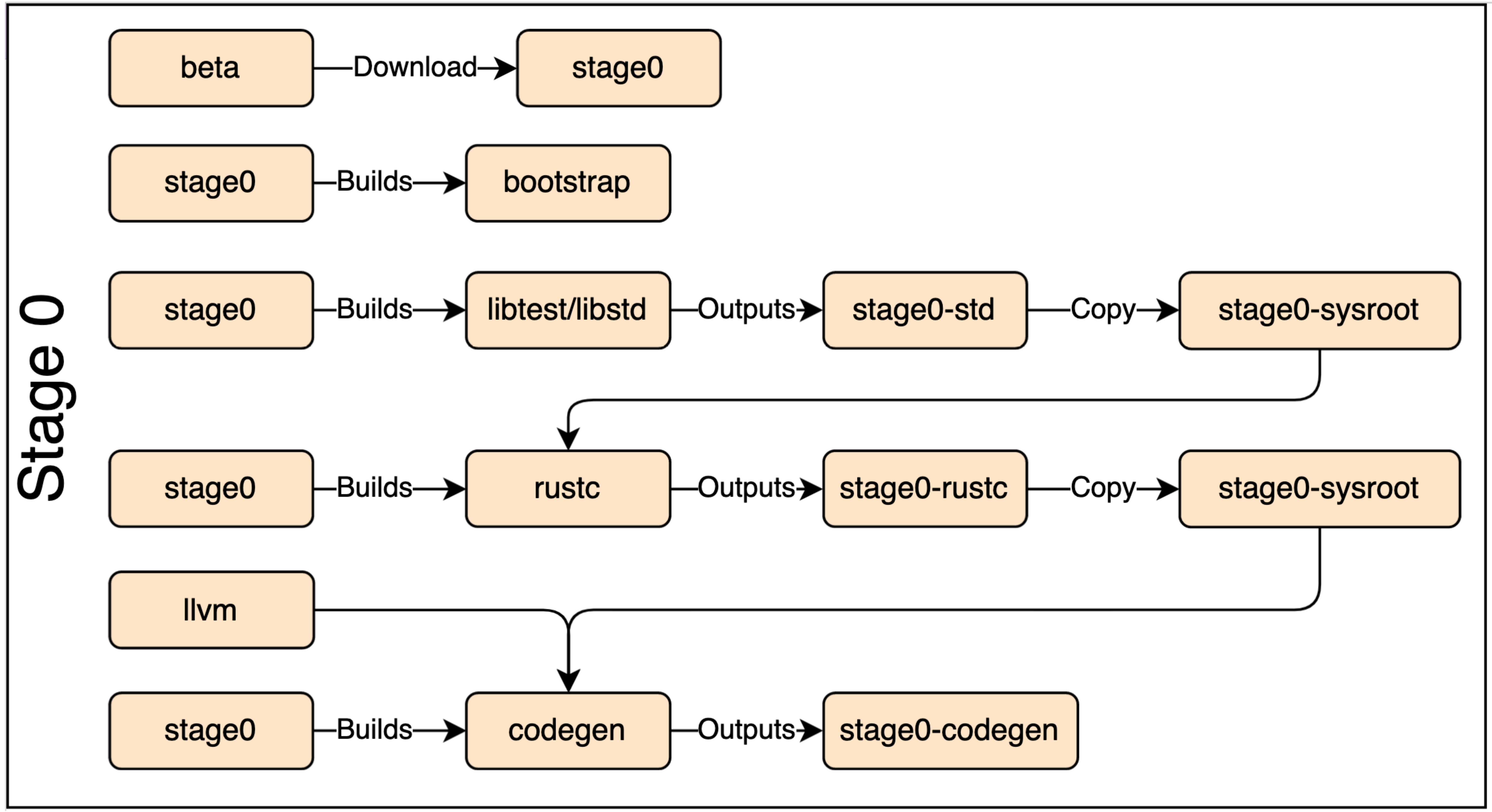
- Keypoint: **memorized**
  - the **first** time invoking a query, it will go do the **computation**
  - the **next** time, the result is returned from a **hashtable**
    - roughly: the result ***may*** be returned by **loading stored data from disk**
  - query execution fits nicely into ***incremental computation***

```
let ty = tcx.type_of(some_def_id);
```

Boostraing: Stage 0

- stage0 compiler:
  - the current **beta** rust **compiler** and its associated **dynamic libraries**
  - used **only** to compile rustbuild, **std**, and **rustc**
  - when compiling rustc, the stage0 compiler uses the **freshly compiled std**
    - **compiler** with its set of dependencies and its '**target**' or 'object' libraries (std and rustc)
    - both are **staged**, but in a **staggered** manner

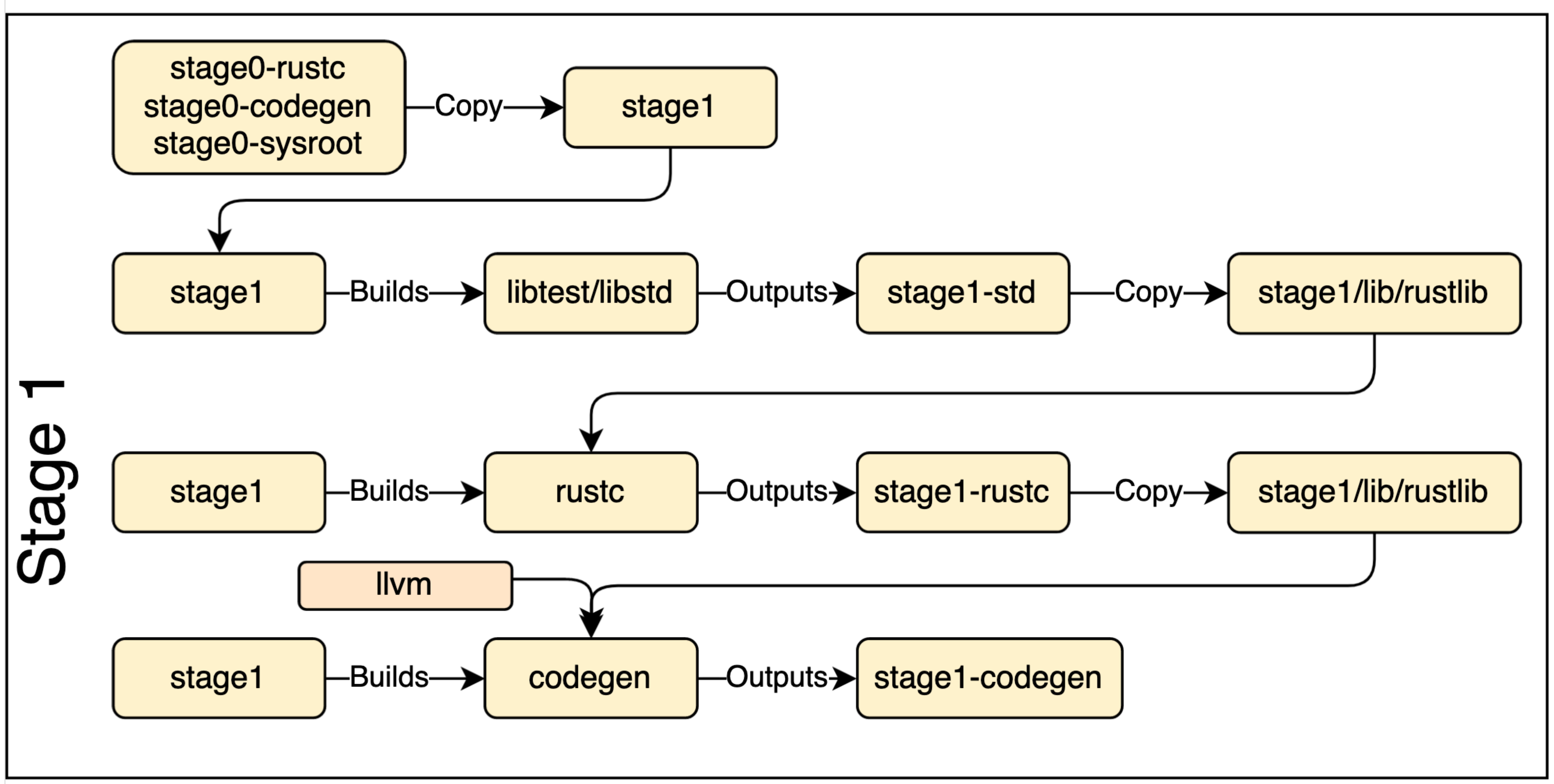
Boostraing: Stage 0



Boostraing: Stage 1

- Stage1 compiler:
  - rebuild **stage1** compiler with ***itself*** to produce the stage2 compiler
- In theory, the stage1 is ~~**functionally identical**~~ to the stage2, but has **subtle differences**:
  - the stage1 was built by stage0 and hence **not by the source** in your working directory *

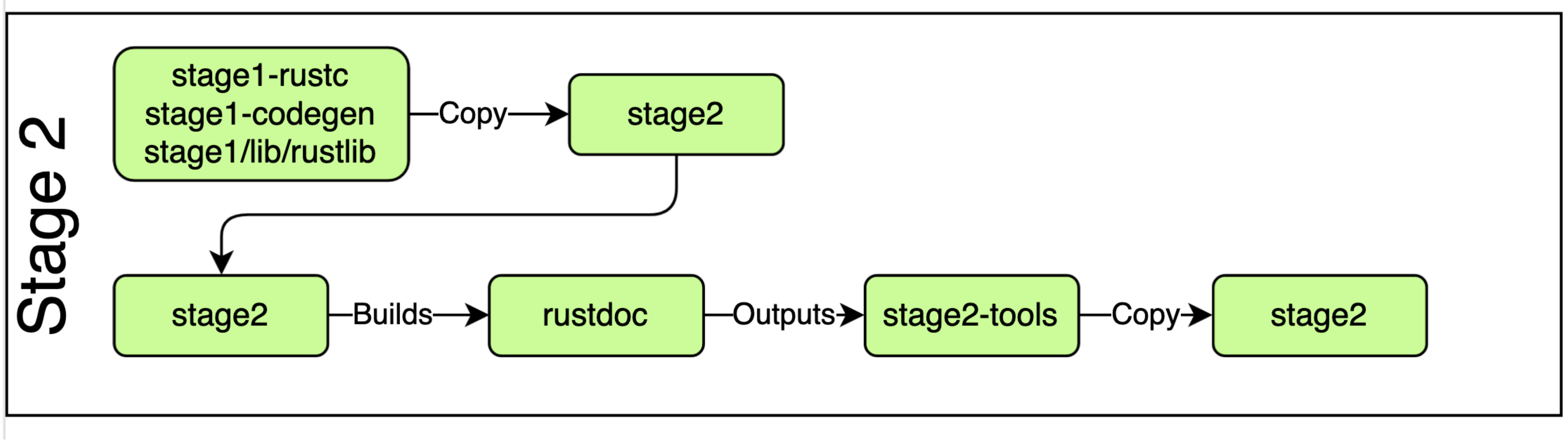
Boostraing: Stage 1



Boostraing: Stage 2

- stage2 compiler:
  - one distributed with **rustup** and all other install methods
  - it takes a **very long time** to build
    - one must **first build the new compiler with an older compiler**
    - then **use that to build the new compiler with itself**

Boostraing: Stage 2



Memory Management: Arena

- Arenas: **arena allocation**
  - create a **LOT** of data structures during compilation
  - allocate from a **global memory pool**
    - each allocated once from a **long-lived** *arena*
  - **reduce** allocations/deallocations of memory
  - allow for easy comparison of types for **equality**:
    - for each interned type x, we implemented ***PartialEq*** => **compare pointers**

Memory Management: TyCtxt

- TyCtxt
- **typing context** / **tcx**: the **central** data structure in the compiler
  - the **context** that uses to perform **all manner of queries**
  - struct ***TyCtxt*** defines a **reference** to the shared context
    - takes a lifetime parameter **‘tcx**
      - when a lifetime ‘tcx => it refers to **arena-allocated data**
      - or data that lives **as long as the arenas**, anyhow

Memory Management: TyS

- TyS
- represent a **type** in the compiler
- Each time we want to **construct** a type, the compiler doesn’t **naively allocate** from the buffer
  - check if that type was **already constructed**.
  - ✅ get the same pointer we had before ❎ make a fresh pointer
    - if two types are the same
    - all we need to do is **compare the pointers** which is efficient

Memory Management: TyS

- TyS
- **TyS** is carefully setup
  - never construct them on the ~~stack~~
  - always allocate them from the **arena**
  - always intern them so they are **unique**

Source Code Representation

Lexing: Overview

- Lexing: ***rustc_lexer***
  - take **strings** and turns into **streams of token**
    - for example, a.b+c => the tokens a, ., b, +, and c
- Token stream:
  - the lexer produces a stream of tokens **directly** from the **source code**
  - is easier for the **parser** to deal with than **raw text**

Lexing: Algorithm

- Algorithm:
- ***rustc_lexer*** crate is responsible for breaking a ***&str*** into chunks constituting tokens
  - **regular expressions** -> **non-deterministic finite automation (NFA)**
  - -> **deterministic finite automation (DFA)**

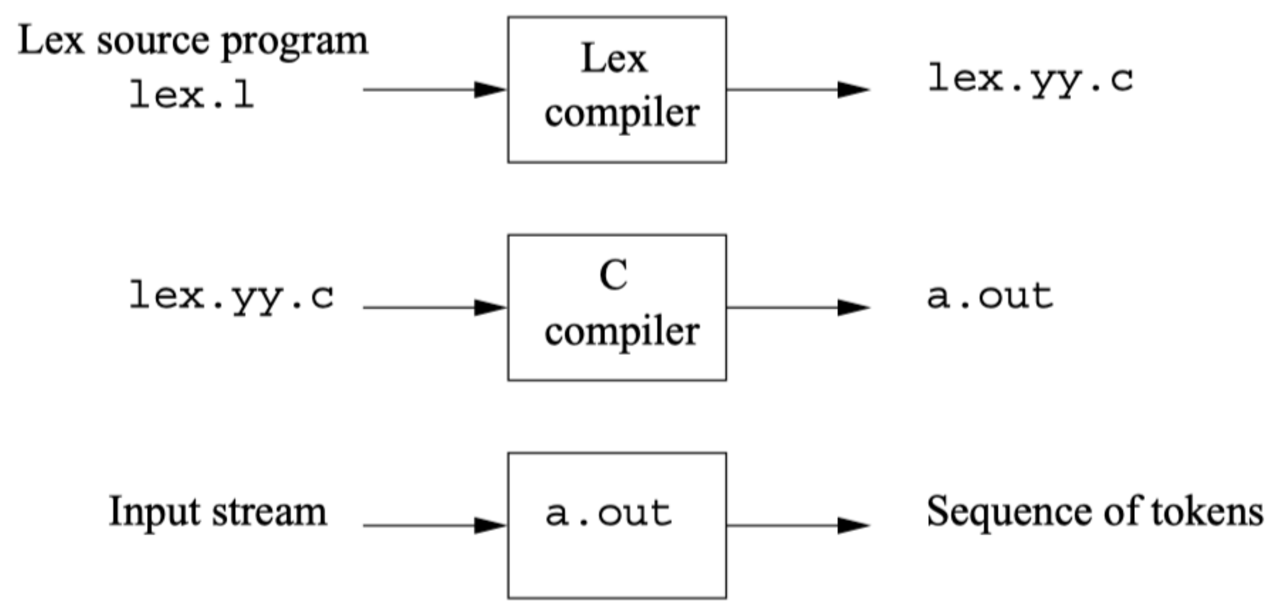


Figure 3.22: Creating a lexical analyzer with Lex

digit	$\rightarrow$	`[0-9]`
digits	$\rightarrow$	$digit^+$
number	$\rightarrow$	*digits* (. *digits*)? (E [+-]? *digits*)?
letter	$\rightarrow$	`[A-Za-z]`
id	$\rightarrow$	*letter* (*letter* | *digit*)*
if	$\rightarrow$	`if`
then	$\rightarrow$	`then`
else	$\rightarrow$	`else`
relop	$\rightarrow$	`<` | `>` | `<=` | `>=` | `=` | `<>`

Figure 3.11: Patterns for tokens of Example 3.8

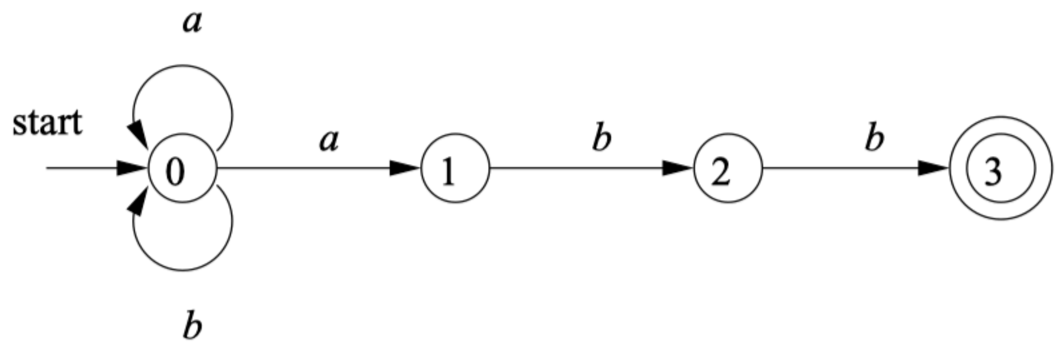


Figure 3.24: A nondeterministic finite automaton

Lexing: Algorithm

- Algorithm:
  - although it is popular to implement lexers as gene
    - the lexer in ***rustc_lexer*** is **hand-written**

```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```

Figure 3.18: Sketch of implementation of **relop** transition diagram

```rust
fn number(&mut self, first_digit: char) -> LiteralKind {
    debug_assert!('0' <= self.prev() && self.prev() <= '9');
    let mut base = Base::Decimal;
    if first_digit == '0' {
        // Attempt to parse encoding base.
        let has_digits : bool = match self.first() {
            'b' => {
                base = Base::Binary;
                self.bump();
                self.eat_decimal_digits()
            }
            'o' => {
                base = Base::Octal;
                self.bump();
                self.eat_decimal_digits()
            }
            'x' => {
                base = Base::Hexadecimal;
                self.bump();
                self.eat_hexadecimal_digits()
            }
            // Not a base prefix.
            '0'..='9' | '_' | '.' | 'e' | 'E' => {
                self.eat_decimal_digits();
                true
            }
            // Just a 0.
            _ => return Int { base, empty_int: false },
        };
```

Parsing: Overview

- Parsing: ***rustc_parser***
  - take **streams of tokens** and turn into a structured form **Abstract Syntax Tree** (AST)
    - AST mirrors the structure of a Rust program in memory
    - using **span** to **link** a particular **AST node** back to its **source text**
- AST: ***rustc_ast***
  - AST is built from the **stream of tokens** produced by the **lexer**
  - **Marco Expansion**, **Name Resolution**, **#[test] implementation**

Parsing: Algorithm

- Algorithm:
  - LL(1): Top-Down Parsing - Recursive Descent - Predictive Parsers

Example 4.27 : The sequence of parse trees in Fig. 4.12 for the input **id**+**id**$*$**id** is a top-down parse according to grammar (4.2), repeated here:

$$\begin{array}{lcl} E & \rightarrow & T\ E' \\ E' & \rightarrow & +\ T\ E'\ |\ \epsilon \\ T & \rightarrow & F\ T' \\ T' & \rightarrow & *\ F\ T'\ |\ \epsilon \\ F & \rightarrow & (\ E\ )\ |\ \mathbf{id} \end{array} \tag{4.28}$$

```rust
/// Eats `|` possibly breaking tokens like `||` in process.
/// Signals an error if `|` was not eaten.
fn expect_or(&mut self) -> PResult<'a, ()> {
    if self.break_and_eat(token::BinOp(token::Or)) { Ok(()) } else { self.unexpected() }
}
```

High-level IR: Overview

- High-level IR (HIR): ***rustc_hir***
  - sort of desugared AST
  - close to what the **user** wrote **syntactically**, but have some **implicit** things: elided lifetimes
- **Amenable to Type Checking.**
  - **1. Determining the type of each expression.**
  - **2. Resolving methods and traits.**
  - **3. Guaranteeing that most type rules are met.**

High-level IR: Type Collection

- Type Collection: ***rustc_typeck***
  - ✅ pass over all items and **determine** their type: from ***hir::Ty*** to ***TyS***
  - ❎ examine their “**innards**”
    -
```
struct Foo { }
fn foo(x: Foo, y: self::Foo) { ... }
//        ^^^      ^^^^^^^^^
```
    - x and y have the **same type**, but have distinct ***hir::Ty*** nodes
      - those nodes have different **spans**, and they encode the **path** somewhat differently
      - once "collected" ***TyS*** into nodes, they will be represented by the exact **same internal type**

High-level IR: Type Check

- Type Check: ***rustc_typeck***
  - **Variance Inference**
    - compute the variance of each **parameter**
  - **Coherence Check**
    - for **overlapping** or **orphaned** impls
  - Check each **function body** at a time
    - **Inference**: supply types wherever they are **unknown**.

Typed High-level IR: Overview

- Typed HIR (THIR):
  - between HIR and MIR
  - like the HIR but it is **fully typed**: generated **after type checking**
  - **only** used for **MIR Construction** and **Exhaustiveness Checking**

Mid-level IR: Overview

- Middle-level IR (MIR):
  - MIR is basically a **Control-Flow Graph** (CFG)
    - a bunch of **basic blocks** with simple **typed statements** inside
    - control flow **edges** to other basic blocks
  - **borrow checking** and **dataflow-based** checks (uninitialized values)
  - a series of **optimizations** and for **constant** evaluation
  - still **generic**, more efficient than after monomorphization

Mid-level IR: Syntax

- MIR always describes the execution of a **single fn**
  - a series of **declarations:** the **stack storage** that will be required
  - a set of **basic blocks**
- **user-declared** bindings have a **1-to-1** relationship with the variables
- **temporaries** are introduced by the compiler in various cases
  - &foo() -> introduce a temporary to store the result of foo()
- **temporaries** are **single-assignment**
  - can be **borrowed** (may be mutated after assignment)
  - it is not **pure SSA**

Listing 1. Core syntax of Rust MIR.

```
BasicBlock := {Statement} Terminator
Statement := LValue = RValue | StorageLive(Value)
          | StorageDead(Value) | ...
LValue := LValue | LValue.f | *LValue | ...
RValue := LValue | move LValue
          | & LValue | & mut LValue
          | * LValue | * mut LValue | ...
Terminator := Goto(BB) | Panic(BB) | Drop(Value)
          | Return | Resume | Abort
          | If(Value, BB0, BB1)
          | LValue = (FnCall, BB0, BB1)
          | SwitchInt(Value, BB0, BB1, BB2, ...) | ...
```

Mid-level IR: Syntax

Listing 1. Core syntax of Rust MIR.

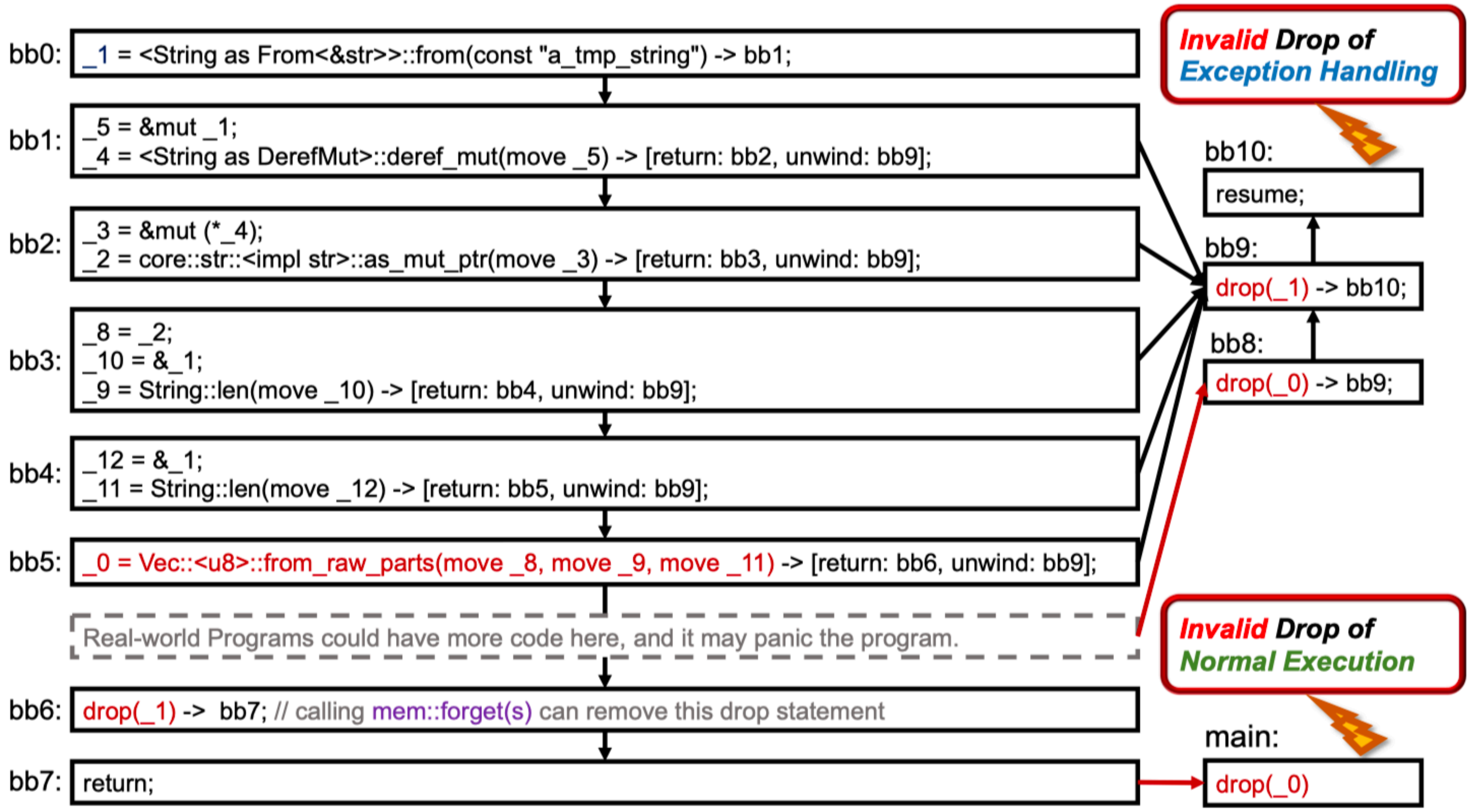
```
BasicBlock := {Statement} Terminator
Statement := LValue = RValue | StorageLive(Value)
        | StorageDead(Value) | ...
LValue := LValue | LValue.f | *LValue | ...
RValue := LValue | move LValue
        | & LValue | & mut LValue
        | * LValue | * mut LValue | ...
Terminator := Goto(BB) | Panic(BB) | Drop(Value)
        | Return | Resume | Abort
        | If(Value, BB0, BB1)
        | LValue = (FnCall, BB0, BB1)
        | SwitchInt(Value, BB0, BB1, BB2, ...) | ...
```

Mid-level IR: Example

```rust
fn genvec() -> Vec<u8> {
    let mut s = String::from("a_tmp_string");
    let ptr = s.as_mut_ptr();
    let v;
    unsafe{
        v = Vec::from_raw_parts(
                ptr, s.len(), s.len());
    }
    // mem::forget(s); // do not drop s
    // otherwise, s is dropped before return
    return v;
}

fn main() {
    let v = genvec();
    // use v -> use after free
    // drop v before return -> double free
}
```

(a) Source code of dropping aliases.



(b) MIR form and CFG of Figure 2a.

Mid-level IR: Type Representation

- ```rust
  struct MyStruct<T>
  ```
- This is one ***TyKind::Adt*** containing the ***AdtDef*** of ***MyStruct*** with the ***SubstsRef***.
- ***AdtDef*** with ***DefId*** for ***MyStruct***
- ***TyKind::Param*** with ***DefId*** for ***T***
- ***SubstsRef*** : **list** ***[GenericArgKind::Type(Ty(T))]***

```rust
pub enum TyKind<'tcx> {
[-]  Bool,
    Char,
    Int(IntTy),
    Uint(UintTy),
    Float(FloatTy),
    Adt(AdtDef<'tcx>, SubstsRef<'tcx>),
    Foreign(DefId),
    Str,
    Array(Ty<'tcx>, Const<'tcx>),
    Slice(Ty<'tcx>),
    RawPtr(TypeAndMut<'tcx>),
    Ref(Region<'tcx>, Ty<'tcx>, Mutability),
    FnDef(DefId, SubstsRef<'tcx>),
    FnPtr(PolyFnSig<'tcx>),
    Dynamic(&'tcx List<Binder<'tcx, ExistentialPredicate<'tcx>>>, Region<'tcx>),
    Closure(DefId, SubstsRef<'tcx>),
    Generator(DefId, SubstsRef<'tcx>, Movability),
    GeneratorWitness(Binder<'tcx, &'tcx List<Ty<'tcx>>>),
    Never,
    Tuple(&'tcx List<Ty<'tcx>>),
    Projection(ProjectionTy<'tcx>),
    Opaque(DefId, SubstsRef<'tcx>),
    Param(ParamTy),
    Bound(DebruijnIndex, BoundTy),
    Placeholder(PlaceholderType),
    Infer(InferTy),
    Error(DelaySpanBugEmitted),
}
```

enum rustc_middle::ty::IntTy
```

LLVM & Codgen: Overview

- LLVM IR:
  - the **standard form** of all input to the LLVM compiler
  - a **standard format** that is used by all compilers that use LLVM
  - a sort of **typed assembly language** with lots of annotations
  - LLVM IR is designed to be **easy** for other compilers to **emit** and also rich enough for LLVM to run a bunch of **optimizations** on it