Rust programming language

Andrea Franceschi

March 26, 2024

Abstract

This is a personal guide related to Rust programming language from multiple source: Rust.org and papers just included at the end of this guide also the code snipped with light background are taken from "learn rust by example", so take it as it is.

1 Introduction

1.1 Types

- i8 / i16 / i32 / i64 / isize
- u8 / u16 / u32 / u64 / usize
- f32 / f64
- bool
- char (4-byte unicode)
- **Type inference** for variables `declarations` with `let`: so the compiler can infer the type of a variable according to the context and the value assigned.
- No overloading for literals: type annotations to disambiguate, operators or functions cannot be overloaded for literals. Cannot define different behaviors for an operator or a function based on the type of literal being used. For instance, you cannot define a different implementation of the + operator to add integers and floating-point numbers.
- Tuples
- Arrays: with fixed length. Runtime check for out-of-bound!

2 Memory management

2.1 ownership

- Each value in Rust has an owner.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped.
- Stack: for activation records
- Heap: for dynamic allocated data structures
- No implicit boxing: programmer must be aware where data are stored
  - `let x = 500;` *//x is on the stack*
  - `let y = Box::new(500);` *//y is a reference to 500 on the heap*
  - `println!("x == y is ", x == *y); // "x == y is true"`
- by the default variables are immutable (cannot change value), can be mutable using `mut`
  - `let mut z:  u8 = 600;`

2.2 RAII

No garbage collector or heap managed by the programmer, Rust uses: **RAII** Resource Acquisition Is Initialization, **resources are freed when not needed anymore**, schema: initialization, resource use, destruction

- RAII programming idiom is a software design principle that states that **resource allocation** occurs during **object initialization**, through the constructor, while **resource deallocation** occurs during **object destruction**, through the destructor.
- this means that when an object is created, the necessary resources are acquired and initialized within the object itself, usually in the constructor.
- These resources can be memory, files, network connections, or any other resource managed by the operating system or the application.
- When the *object is no longer needed* or goes **out of valid scope** where it was declared, it is automatically **destroyed**, and the object's destructor is invoked. This `destructor` is responsible for `releasing the resources` acquired during the object's initialization. This way, resources are released properly and safely, avoiding memory leaks or resource leaks.
- Since **objects** are **bound** to the **scope** (function, block) where they are declared, when the scope closes, objects are destroyed, and any owned resource is released.
- **Each resource has a unique owner**, meaning that the object that acquired the resource is the **only one responsible for its release**. This ensures that resources are released properly and that there are no resource management conflicts.
- **Ownership can be borrowed as reference** in a mutable or in an immutable manner with lifetime rules (see Figure **??**).
- The variable is responsible for managing the lifetime and deallocation of the memory it owns.
- **Borrowing** comes with several restrictions, enforced by **lifetime** rules. These rules ensure that **references cannot outlive the data they refer to and that mutable references cannot be aliased**, preventing data races and ensuring memory safety
- **Raw pointers** similar to references, but without the safety guarantees enforced by the ownership system. Operations involving raw pointers that could potentially violate memory safety, such as dereferencing, are marked as unsafe and require explicit handling with the `unsafe` keyword.

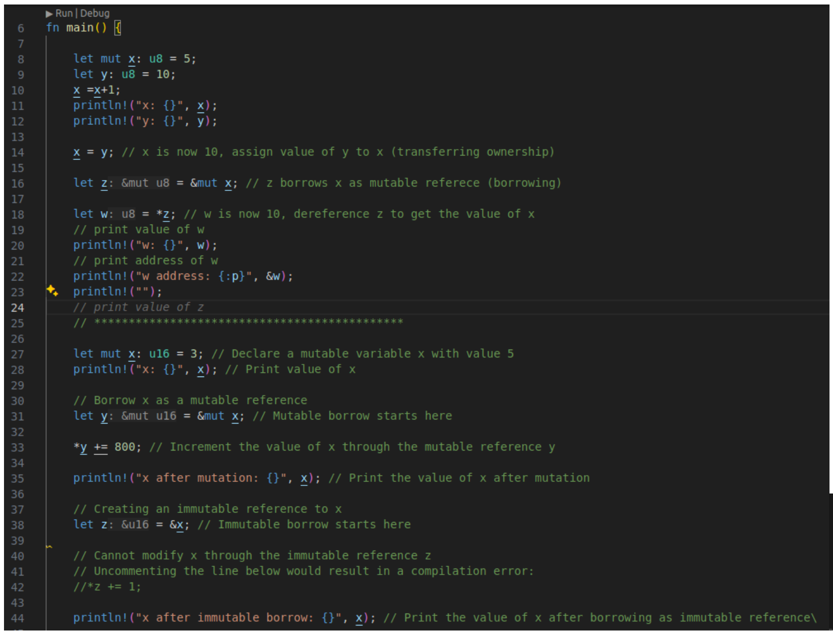
Various mechanisms, including the use of **traits** (shared behavior across different types), to ensure the RAII (Resource Acquisition Is Initialization) principle. The main traits used to ensure this principle are `Copy` trait for stack-only data and `Drop` trait for others.
If the value has Copy trait, then Rust will duplicate (copy) it in the stack and the older variable is still usable. Otherwise, it will transfer (move) the ownership, thus the older variable is no longer available.
These traits determine the way for generating `lvalue` from `rvalues` in assignments (left-hand side and right-hand side of an assignment operation), parameter passing, and value returning.

1. **Assignments**: when assigning a value to a variable, the value on the right-hand side (rvalue) is copied or moved to the location of the variable on the left-hand side (lvalue), depending on the behavior defined by the types involved and their traits.
   - If the value being assigned implements the `Copy` trait, it will be copied to the destination location, allowing both the original and the copy to be independently usable.
   - If the value being assigned does not implement `Copy` but implements the `Drop` trait, it will be moved to the destination location. This means that the original variable loses ownership of the value, and it cannot be used afterward.
2. **Parameter passing**: Similar to assignments, passing values as function arguments involves moving or copying the values, depending on their types and traits.

    - If a value is passed by value (not by reference) and it implements `Copy`, it will be copied to the function's parameter, allowing the function to work with an independent copy of the original value.
    - If a value is passed by value and it does not implement `Copy` but implements `Drop`, it will be moved to the function's parameter, transferring ownership to the function and preventing the caller from using the value afterward.

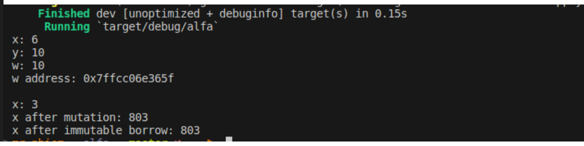
3. **Value returning**: When a function returns a value, it can either return the value itself or move it out of the function's scope.
    - If a function returns a value that implements `Copy`, it will be copied and returned to the caller, allowing the caller to use the returned value independently of the original.
    - If a function returns a value that does not implement `Copy` but implements `Drop`, it will be moved out of the function's scope, transferring ownership to the caller and preventing the function from using the value afterward.

- **Drop**: for types that require special handling when they are no longer needed. Types implementing the Drop trait have associated destructor logic that is executed when the variable goes out of scope. This allows resources to be automatically released at the end of their valid scope, similar to the RAII (Resource Acquisition Is Initialization) pattern in C++
- **Copy**: for types that can be safely copied by simply duplicating their values on the stack. Types implementing the Copy trait are stack-only data types, and their values are copied when assigned to another variable or passed as function arguments. The older variable remains usable after the copy operation.



```rust
fn main() {

    let mut x: u8 = 5;
    let y: u8 = 10;
    x =x+1;
    println!("x: {}", x);
    println!("y: {}", y);

    x = y; // x is now 10, assign value of y to x (transferring ownership)

    let z: &mut u8 = &mut x; // z borrows x as mutable referece (borrowing)

    let w: u8 = *z; // w is now 10, dereference z to get the value of x
    // print value of w
    println!("w: {}", w);
    // print address of w
    println!("w address: {:p}", &w);
    println!("");
    // print value of z
    // *******************************************

    let mut x: u16 = 3; // Declare a mutable variable x with value 5
    println!("x: {}", x); // Print value of x

    // Borrow x as a mutable reference
    let y: &mut u16 = &mut x; // Mutable borrow starts here

    *y += 800; // Increment the value of x through the mutable reference y

    println!("x after mutation: {}", x); // Print the value of x after mutation

    // Creating an immutable reference to x
    let z: &u16 = &x; // Immutable borrow starts here

    // Cannot modify x through the immutable reference z
    // Uncommenting the line below would result in a compilation error:
    //*z += 1;

    println!("x after immutable borrow: {}", x); // Print the value of x after borrowing as immutable reference\
```

(a) code



```
    Finished dev [unoptimized + debuginfo] target(s) in 0.15s
     Running `target/debug/alfa`
x: 6
y: 10
w: 10
w address: 0x7ffcc06e365f

x: 3
x after mutation: 803
x after immutable borrow: 803
```

(b) output

Figure 1

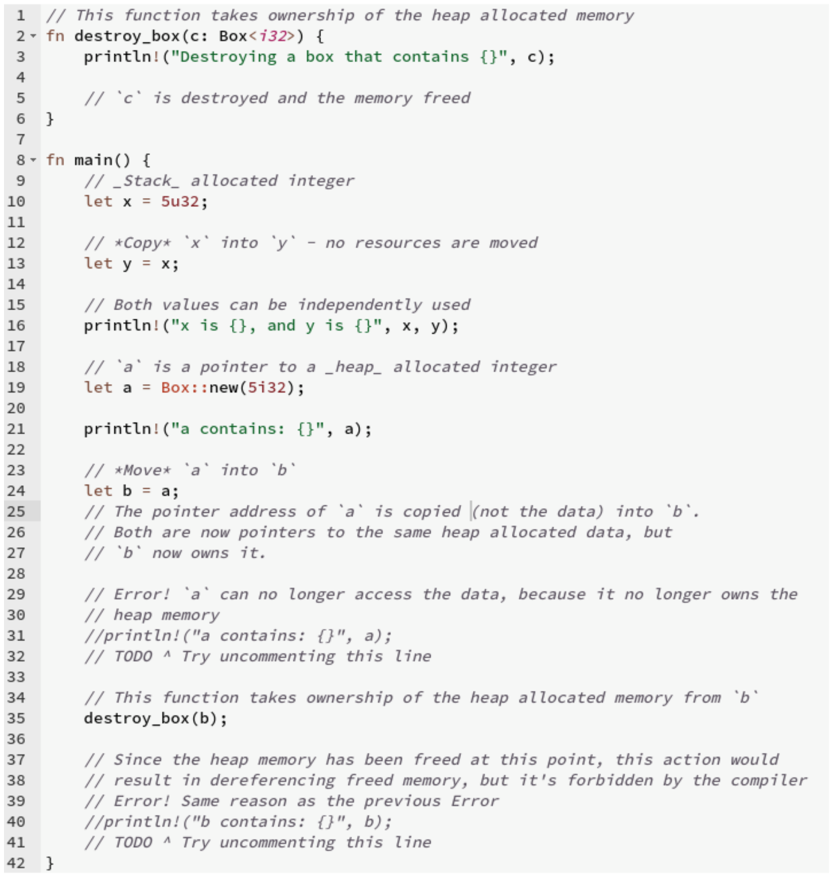
2.3 Ownership

Resource ownership:
In Rust, **every resource** (such as memory allocated on the heap) is associated with a **single owner**. This owner is responsible for deallocating the resource when it's no longer needed. Unlike languages

with garbage collection, Rust's ownership model ensures deterministic resource management. Note that not all variables own resources (e.g. references).

Transfer of Ownership:
When a resource is assigned to another variable (`let x = y`) or passed as a function argument by value (`foo(x)`), ownership of the resource is transferred from the previous owner to the new owner. This transfer of ownership is known as a **move** in Rust terminology.

Preventing Dangling Pointers:
Once a resource is moved, the previous owner can no longer access it. This prevents the creation of dangling pointers, which are pointers that reference deallocated memory. By enforcing this rule, Rust guarantees memory safety and eliminates the risk of undefined behavior due to dangling pointers.



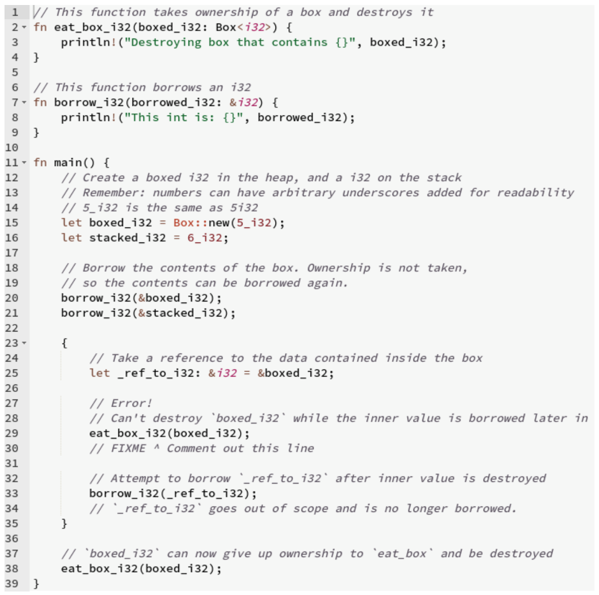
```rust
// This function takes ownership of the heap allocated memory
fn destroy_box(c: Box<i32>) {
    println!("Destroying a box that contains {}", c);

    // `c` is destroyed and the memory freed
}

fn main() {
    // _Stack_ allocated integer
    let x = 5u32;

    // *Copy* `x` into `y` - no resources are moved
    let y = x;

    // Both values can be independently used
    println!("x is {}, and y is {}", x, y);

    // `a` is a pointer to a _heap_ allocated integer
    let a = Box::new(5i32);

    println!("a contains: {}", a);

    // *Move* `a` into `b`
    let b = a;
    // The pointer address of `a` is copied (not the data) into `b`.
    // Both are now pointers to the same heap allocated data, but
    // `b` now owns it.

    // Error! `a` can no longer access the data, because it no longer owns the
    // heap memory
    //println!("a contains: {}", a);
    // TODO ^ Try uncommenting this line

    // This function takes ownership of the heap allocated memory from `b`
    destroy_box(b);

    // Since the heap memory has been freed at this point, this action would
    // result in dereferencing freed memory, but it's forbidden by the compiler
    // Error! Same reason as the previous Error
    //println!("b contains: {}", b);
    // TODO ^ Try uncommenting this line
}
```

Figure 2: This is an example of ownership.

2.4 Borrowing

Rust allow to **access data without taking ownership** of it. This is where borrowing comes into play. Borrowing allows us **to pass objects by reference instead of by value**. In Rust, a reference is denoted by `&T`, where `T` is the type of the object being referenced.
The borrowing mechanism in Rust ensures memory safety by statically guaranteeing, through its **borrow checker**, that references always point to valid objects. This means that as long as references to an object exist, the object cannot be destroyed or deallocated. In other words, the compiler ensures that references remain valid and that the referenced data is not modified in unexpected ways.

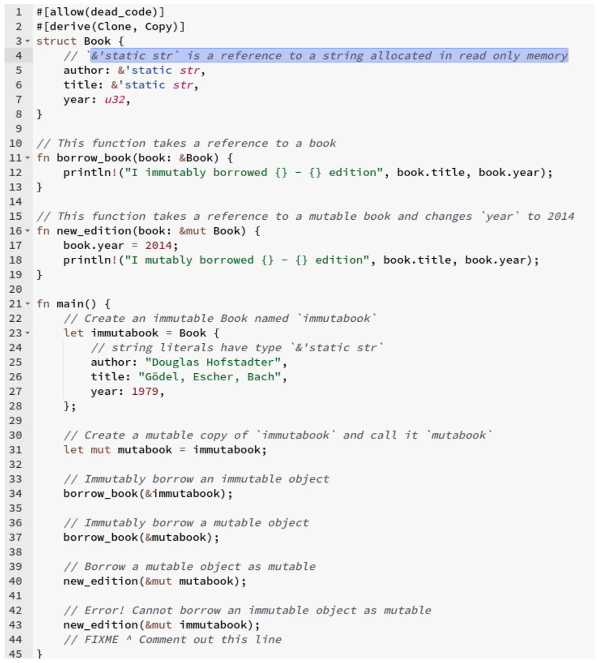
In Rust, **mutable data** can be **mutably borrowed** using `&mut T`. This is known as a **mutable reference**, and it provides read/write access to the borrower. With a mutable reference, the borrower can both read and modify the data it references.
On the other hand, `&T borrows` the data via an **immutable reference**. With an immutable reference, the **borrower can read** the data but cannot modify it. Immutable references provide **read-only** access to the data they reference, ensuring that the data remains unchanged while the reference is active.



```rust
// This function takes ownership of a box and destroys it
fn eat_box_i32(boxed_i32: Box<i32>) {
    println!("Destroying box that contains {}", boxed_i32);
}

// This function borrows an i32
fn borrow_i32(borrowed_i32: &i32) {
    println!("This int is: {}", borrowed_i32);
}

fn main() {
    // Create a boxed i32 in the heap, and a i32 on the stack
    // Remember: numbers can have arbitrary underscores added for readability
    // 5_i32 is the same as 5i32
    let boxed_i32 = Box::new(5_i32);
    let stacked_i32 = 6_i32;

    // Borrow the contents of the box. Ownership is not taken,
    // so the contents can be borrowed again.
    borrow_i32(&boxed_i32);
    borrow_i32(&stacked_i32);

    {
        // Take a reference to the data contained inside the box
        let _ref_to_i32: &i32 = &boxed_i32;

        // Error!
        // Can't destroy `boxed_i32` while the inner value is borrowed later in
        eat_box_i32(boxed_i32);
        // FIXME ^ Comment out this line

        // Attempt to borrow `_ref_to_i32` after inner value is destroyed
        borrow_i32(_ref_to_i32);
        // `_ref_to_i32` goes out of scope and is no longer borrowed.
    }

    // `boxed_i32` can now give up ownership to `eat_box` and be destroyed
    eat_box_i32(boxed_i32);
}
```

Figure 3: This is an example of borrowing.

Can use `&'static` to create a reference allocated in read only memory:



```rust
#[allow(dead_code)]
#[derive(Clone, Copy)]
struct Book {
    // `&'static str` is a reference to a string allocated in read only memory
    author: &'static str,
    title: &'static str,
    year: u32,
}

// This function takes a reference to a book
fn borrow_book(book: &Book) {
    println!("I immutably borrowed {} - {} edition", book.title, book.year);
}

// This function takes a reference to a mutable book and changes `year` to 2014
fn new_edition(book: &mut Book) {
    book.year = 2014;
    println!("I mutably borrowed {} - {} edition", book.title, book.year);
}

fn main() {
    // Create an immutable Book named `immutabook`
    let immutabook = Book {
        // string literals have type `&'static str`
        author: "Douglas Hofstadter",
        title: "Gödel, Escher, Bach",
        year: 1979,
    };

    // Create a mutable copy of `immutabook` and call it `mutabook`
    let mut mutabook = immutabook;

    // Immutably borrow an immutable object
    borrow_book(&immutabook);

    // Immutably borrow a mutable object
    borrow_book(&mutabook);

    // Borrow a mutable object as mutable
    new_edition(&mut mutabook);

    // Error! Cannot borrow an immutable object as mutable
    new_edition(&mut immutabook);
    // FIXME ^ Comment out this line
}
```

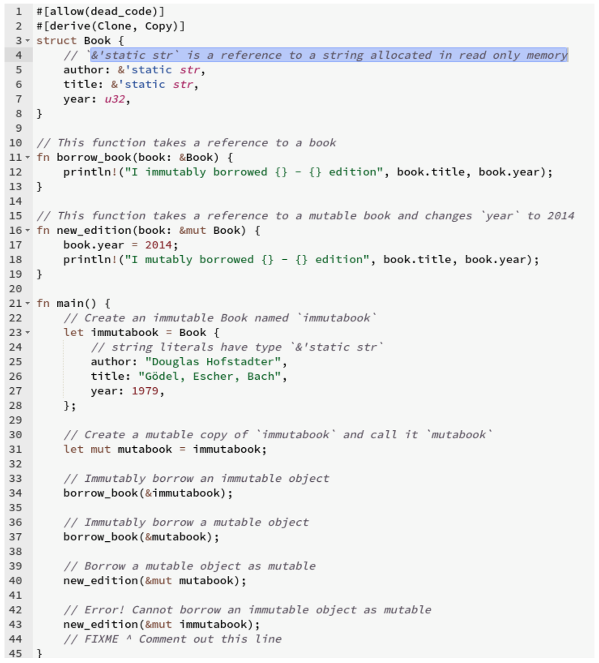
Figure 4: This is an example of borrowing.

Can also use the **ref pattern** where `ref` borrow on the left side of an assignment is equivalent to an `&` on the right side:

```rust
let ref ref_c1 = c;
let ref_c2 = &c;
println!("ref_c1 equals ref_c2:  ", *ref_c1 == *ref_c2);
```

2.5 Lifetime

lifetimes are a feature of the type system that ensures references are valid for the duration of their use. Lifetimes specify the scope during which a reference is valid and prevent dangling references or references to deallocated memory.
lifetime is a construct the borrow checker's compiler uses to ensure all borrows are valid.
Specifically, a **variable's lifetime begins when it is created and ends when it is destroyed**. While **lifetimes and scopes are not the same**. For example, the case where we borrow a variable via &. The borrow has a lifetime that is determined by where it is declared. As a result, the borrow is valid as long as it ends before the lender is destroyed. However, the scope of the borrow is determined by where the reference is used.



```rust
#[allow(dead_code)]
#[derive(Clone, Copy)]
struct Book {
    // `&'static str` is a reference to a string allocated in read only memory
    author: &'static str,
    title: &'static str,
    year: u32,
}

// This function takes a reference to a book
fn borrow_book(book: &Book) {
    println!("I immutably borrowed {} - {} edition", book.title, book.year);
}

// This function takes a reference to a mutable book and changes `year` to 2014
fn new_edition(book: &mut Book) {
    book.year = 2014;
    println!("I mutably borrowed {} - {} edition", book.title, book.year);
}

fn main() {
    // Create an immutable Book named `immutabook`
    let immutabook = Book {
        // string literals have type `&'static str`
        author: "Douglas Hofstadter",
        title: "Gödel, Escher, Bach",
        year: 1979,
    };

    // Create a mutable copy of `immutabook` and call it `mutabook`
    let mut mutabook = immutabook;

    // Immutably borrow an immutable object
    borrow_book(&immutabook);

    // Immutably borrow a mutable object
    borrow_book(&mutabook);

    // Borrow a mutable object as mutable
    new_edition(&mut mutabook);

    // Error! Cannot borrow an immutable object as mutable
    new_edition(&mut immutabook);
    // FIXME ^ Comment out this line
}
```

Figure 5: This is an example of lifetime.

- `Scope`: lifetimes are associated with references and indicate the duration for which the referenced data is valid. They are denoted by a tick mark (') followed by a name, such as 'a, 'b, etc.
- `Annotations`: lifetimes can be explicitly annotated in function signatures, struct definitions, and other places where references are used. This helps the compiler verify that references adhere to the specified lifetimes.
- `Lifetime elision`: rules that automatically infer lifetimes in many common situations, reducing the need for explicit annotations. This makes Rust code more concise while still ensuring safety.
- `Borrow checker`: rust's borrow checker analyzes lifetimes to ensure that references are used correctly and safely. It enforces rules such as the borrowing rules, which prevent mutable references from coexisting or overlapping.
- `Parameterized lifetimes`: lifetimes can be parameterized, allowing functions and data structures to work with references of varying durations. This enables generic programming with lifetimes.
- `lifetime bounds`: lifetimes can be bounded to specify relationships between multiple references, such as ensuring that one reference outlives another. This helps prevent dangling references and memory safety issues.

Explicit annotation
Since the borrow checker ensures memory safety by tracking the lifetimes of references and lifetimes define the scope for which a reference is valid and can be safely used, the borrow checker uses explicit lifetime annotations to determine **how long references should be valid**.
These explicit annotations specify the **relationships between the lifetimes of references** in the code. By explicitly defining lifetimes, developers **can ensure that references do not outlive the data they point to** or are not used after the data has been deallocated, thereby preventing memory safety issues like dangling references or use-after-free errors.
`foo<'a> // ‘foo‘ has a lifetime parameter ‘'a‘`
Using lifetimes requires generics and lifetime syntax indicates that the lifetime of the function `foo` may not exceed that of `'a`. Explicit annotation of a type has the form `&'a T` where `'a` has already been introduced.
NI the following example, the lifetime of `foo` cannot exceed that of either `'a` or `'b`:

3 Rust compiler

3.1 crate

Crate is a compilation unit, which can be an entire application, a library, or a module within a larger program. Crates are the primary mechanism for organizing code in Rust and provide a form of modularity and code reuse:

- Compilation Unit: a crate is a single compilation unit, which can contain one or more modules and their corresponding source code. Each crate is compiled separately into an executable binary file or a library.
- Modularity: Crates allow code to be organized into logical modules, making it easier to manage and maintain. Modules can be defined within a crate to separate related functionality and keep the code cleaner and more readable.
- Code Reuse: Crates can be used as dependencies in other Rust projects. This allows existing code to be easily reused, encouraging the creation and sharing of open-source libraries.
- Dependency Management: Rust uses Cargo, its package and dependency manager, to manage crates and project dependencies. Cargo simplifies dependency management, code compilation, and distribution of Rust applications.

3.2 Invocation

Compilation starts when a Rust source program is written and `rustc` invoked, there are some flags es LLVM-IR to extract the LLVM Intermediate Representation

3.3 Lexing and parsing

The raw source code is initially analyzed by a low-level component called a **lexer**, which is located in a module called `rustc_lexer`. The purpose of the lexer is to `break down the source code` into smaller units known as `tokens`.
`Tokens` are `atomic units` of source code that represent individual elements such as keywords, identifiers, literals (like numbers and strings), punctuation symbols, and operators. These tokens serve as the `building blocks` for the subsequent **stages of compilation**.
The lexer operates on the raw source text character by character, identifying and **categorizing each character into appropriate tokens** based on the Rust language `syntax rules`: in this case lexical-grammar (regular grammar). It recognizes sequences of characters that form keywords, identifiers, literals, punctuation symbols, and operators, among others. An important feature of the lexer is its support for Unicode character encoding, which allows Rust source code to contain characters from a wide range of languages and scripts.
After this, the token stream passes through a **higher-level lexer** located in `rustc_parse`. This higher-level lexer prepares the token stream for the next stage of the compilation process.

At this stage, the `StringReader struct` is employed to perform a series of *validations* and *to convert strings into interned symbols*. **String interning** is a technique used *to store only one immutable copy of each unique string value*. This means that if multiple parts of the code contain the same string literal, they will all *share the same underlying memory representation*, reducing memory usage and improving efficiency.
The lexer in `rustc_parse` has a minimal interface and does not directly rely on the diagnostic infrastructure present in rustc. Instead, it generates diagnostics as plain data, which are later emitted as real diagnostics in `rustc_parse::lexer`. This separation allows for a more modular and decoupled design, where the **lexer can focus solely on its primary task without being burdened by the complexities of diagnostics handling**.
Furthermore, the lexer ensures that it preserves full fidelity information for both Integrated Development Environments (IDEs) and procedural macros (often referred to as "proc-macros"). This means that the lexer provides detailed information about the structure of the code, allowing IDEs to offer features like syntax highlighting, code completion, and error detection. Similarly, procedural macros, which are used to extend the Rust language with custom syntax, can rely on accurate tokenization provided by the lexer to perform their transformations effectively.
Once the token stream generated by the lexer has been produced, it's passed to the **parser**, which translates it into an `Abstract Syntax Tree (AST)`. The AST represents the hierarchical structure of the source code in a more abstract and structured form, making it easier for subsequent stages of the compiler to analyze and manipulate and has several properties that aid the further steps of the compilation process:

- an AST can be edited and enhanced with information such as properties and annotations for every element it contains. Such editing and annotation is impossible with the source code of a program, since it would imply changing it.
- Compared to the source code, an AST does not include inessential punctuation and delimiters (braces, semicolons, parentheses, etc.).
- usually contains extra information about the program, due to the consecutive stages of analysis by the compiler. For example, it may store the position of each element in the source code, allowing the compiler to print useful error messages.
- ASTs are needed because of the inherent nature of programming languages and their documentation. Languages are often ambiguous by nature. In order to avoid this ambiguity, programming languages are often specified as a texttttcontext-free grammar (CFG).

The parser employs a `top-down` parsing, to perform `syntax analysis`: the parser starts from the top-level constructs of the language grammar and recursively breaks down the input token stream into smaller and smaller parts until it constructs the complete AST.
In the Rust compiler codebase, the entry points for the parser are primarily two methods:
`Parser::parse _crate_mod()` and `Parser::parse _mod()`, which are located in the `rustc_parse:: parser::Parser module`.
These methods are responsible for parsing the top-level constructs of a Rust crate or module, respectively.
For `parsing external modules`, there's an entry point located in texttttrustc_expand::module::parse_external_mod: handles the parsing of Rust modules that are external to the current `crate` being compiled. Additionally, the parser also serves as the entry point for parsing macros, with the `Parser::parse_nonterminal()` method handling the parsing of macro invocations and expansions. To aid in the parsing process, the parser utilizes a set of utility methods such as bump, check, eat, expect, and look_ahead. These methods help the parser navigate through the token stream, inspect tokens, and enforce syntax rules defined by the Rust language grammar. Each of these methods plays a specific role in advancing the parsing process and ensuring correctness according to the language specifications.

3.4 AST lowering

After the Abstract Syntax Tree (AST) has been constructed, it undergoes a transformation into a `High-Level Intermediate Representation (HIR)`, which is a more compiler-friendly representation of the AST. This transformation process is commonly referred to as `lowering`. The AST is

transformated and desugared, which involve expanding and formalizing shortened or abbreviated syntax constructs, such as loops and async functions.
The `primary purpose` of lowering the AST into HIR is *to prepare it for subsequent stages of the compilation process*, where more detailed analysis and manipulation are performed. The HIR representation provides a more structured and uniform representation of the code, making it easier for the compiler to perform tasks like type inference, trait solving, and type checking.
Type inference is the process of automatically `determining` the **types of expressions** in the code without explicit type annotations (i.e.: `let var = 2;` - we have not defined the type of the variable such that `let var:  u8 = 2`). This is particularly useful in Rust, where the type system is strong but allows for flexibility and conciseness in expressing code.
Trait solving: when you use a trait in your code, such as Clone or Display, you're essentially saying, "I want this type to have the behavior defined by this trait." However, traits alone don't provide implementations; they define a set of behavior that types can choose to implement. When the compiler encounters code that uses a trait, needs to find the appropriate implementation of that trait for the specific type being used.
For example, if you have a function that takes a parameter of type T, and you call a method defined by a trait on that parameter, the compiler needs to find the implementation of that trait for type T.
The process of trait solving `ensures` that the `compiler can match the trait reference` (e.g., calling a trait method) with the `correct implementation for the specific type`. This allows for **polymorphic behavior**, meaning that the `same code can work with different types` as long as those types implement the required traits.
Type checking is a critical step where the types found in the HIR, representing what the user wrote, are converted into an internal representation used by the compiler. This internal representation, typically denoted as `Ty<'tcx>`, captures additional semantic information and annotations necessary for further analysis and optimization. Type checking is crucial for verifying the type safety, correctness, and coherence of the types used throughout the program, helping to prevent common errors and ensure robustness in Rust code.

3.5 MIR Lowering

the High-Level Intermediate Representation (HIR) undergoes further transformation into the Mid-Level Intermediate Representation (MIR), which is primarily used for **borrow checking** 4. Before reaching MIR, the HIR is first converted into THIR (Typed Higher-Level Intermediate Representation), an even more desugared form of HIR, used for pattern and exhaustiveness checking.
Once we have the MIR representation, various optimizations are performed on it. MIR is a generic representation, meaning it is not specific to any particular type, which makes it easier to apply optimizations that can improve code generation and compilation speed. Some optimizations are more effective at the MIR level compared to the LLVM-IR level, which is the intermediate representation used by the LLVM compiler backend.
One example of an optimization performed at the MIR level is the simplify_try MIR optimization, which simplifies patterns. While LLVM can perform many optimizations, it may not always recognize certain patterns that can be optimized. By applying optimizations at the MIR level, we can target specific patterns that are more effectively optimized before the code is translated into LLVM-IR.

During code generation, Rust employs a process called monomorphization, which involves generating specialized versions of generic code by replacing type parameters with concrete types. This process ensures that each function or data structure is compiled for every concrete type it is used with, optimizing performance and reducing runtime overhead.

To perform monomorphization, the compiler needs to collect a list of concrete types for which to generate specialized code. This collection process, known as monomorphization collection, occurs at the MIR level. Essentially, the compiler analyzes the MIR representation to identify all the places where generic code is used and determines which concrete types need specialized code generated for them. This ensures that the generated code is efficient and tailored to the specific types used in the program.

4 Borrow checking

4.1 How to add Citations and a References List

You can simply upload a `.bib` file containing your BibTeX entries, created with a tool such as JabRef. You can then cite entries from it, like this: [**?**]. Just remember to specify a bibliography style, as well as the filename of the `.bib`. You can find a video tutorial here to learn more about BibTeX.

If you have an upgraded account, you can also import your Mendeley or Zotero library directly as a `.bib` file, via the upload menu in the file-tree.

Rust.org

References