Rust Programming Language

Faculty of Computing and Information Technology, King Abdulaziz University

CPCS-301: Programming Languages

Dr. Rania Molla

Group Project - Spring 2022

Group No: 5

Student Name	**Section**
Anfal sultan alshehri (leader)	**GAR**
Aisha Saeed alhrbi	**IAR**
Areej Abdullah Suleman	**GAR**
Danah Saleh Almalki	**GAR**
Jana Faisal Altalhi	**GAR**
Layan Turki Zaafarani	**GAR**

Table of Contents

List of figures

List of tables

Rust Programing Language

Programing languages are just tools to set up the software. Rust especially is designed to build software where performance and correctness are essential.

Rust programing language is a system programing language that delivers high-level language simplicity with low-level performance.

It is a popular choice for building systems where performance is critical. Such as databases, compilers, or operating systems. Rust was a side project of Graydon Hoare in 2007. It was sponsored by Mozilla in 2009. Rust has been ranked the most loved programing language every year since 2016 according to stack overflow *(Stack Overflow Developer Survey 2019).*

Moreover, Rust is a multi-paradigm language that allows the most suitable programming style for a task. it takes a lot of its influences from programming languages like C++ and Haskell. The compiler in Rust is very powerful it catches things like Dandling pointers, and data races. Rust is a modern language as well, but it is already being used by Amazon and Sky box *(Companies That Use Rust in Production,2021).*

In this report, we will cover:

- Historical overview of Rust
- Implementation method in Rust
- Domain of Rust
- BNF and EBNF in Rust
- The evaluation of the Rust language
- Types of variables, Array and Scope in Rust
- The code of the Student Grads Program in Rust

1. Historical overview of Rust

Rust is a systems programming language, like C and C++, that is used to develop system software like operating systems, filesystems, memory management, device drivers, etc. *(Blandy, 2021)*

Since programming languages grant control over memory and maintain a close relationship between the primitive operations of the language and the hardware, programmers should consider their code's cost and proceed with caution to avoid bugs. This is where the need for rust as a new programming language presents itself. As Rust combines memory safety and the performance of C and C++ while preventing bugs that are common in these two languages. *(Blandy, 2021)*

As mentioned earlier Rust started as a side project for Graydon Hoare, a language engineer at Mozilla in 2006. And in 2009 Mozilla got involved as a sponsor for the project, setting up a team of developers to work on this language. The interest in rust was because of "servo", a long-term project web engine that shares code with Mozilla's infamous browser "Firefox". Their goal was to build "servo" around safer and easier technologies than C++. *(Avram, 2012)*

After a series of alpha and beta releases, Rust got its first stable release in 2015, "Rust 1.0". (*Announcing rust 1.0: Rust blog* 2015)

2. Implementation method

If we think about how does a computer program works on the computer? The first thing that comes to our mind is that the Operating System translates our code into a machine code, which is true. Operating Systems work with another software called Programming language implantation system to execute the program. There are three general methods to implement computer programs: Compilation, Interpretation, and Hybrid Implementation Systems *(Sebesta,2019).*

The methods implanted in Rust programing language to execute the programs are Compilation and Interpretation.

2.1. Compilation

The translation of source programs from high-level language to machine codes then executes the programs. The compilation has certain phases as shown in Figure 1. *(Sebesta,2019)*

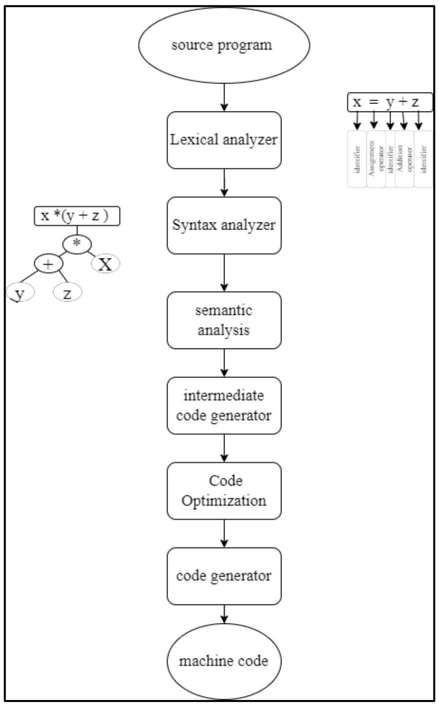


Figure 1:Compilation in programing language

2.2. Interpretation

The interpretation method executes the program without translating to machine language using software called an interpreter. It is faster than Compilation but needs large memory space. *(Sebesta,2019)*

3. Domains of Rust

Rust is a general-purpose systems language, and it is used for applications and programs where security, reliability, and speed are prioritized. Common uses of Rust:

3.1. Programming:

From general programming projects to third-party applications for a wide range of operating systems.

3.2. Web Development:

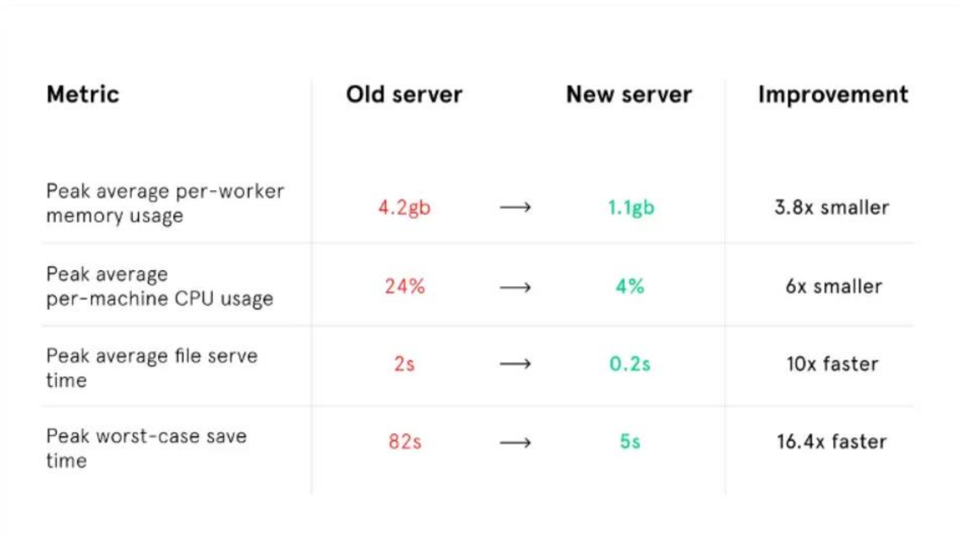
An important application since web servers and sites are used by many users every day and that number could grow the more popular a web server/site gets, and this large number of users requires better data and memory management for speed and security's sake.

Figma, a web design tool that allows real-time collaboration, decided to rewrite parts of their multiplayer server in Rust to solve problems with syncing and lack of parallel operations in their old server. Figure 2 demonstrates the improvement in run time and data usage for Figma after rewriting parts of their multiplayer server in Rust. *(Wallace, 2018)*

Figure 2 shows the improvement in run time and data usage for Figma after rewriting their servers in rust. *(Wallace, 2018)*

3.3. Game Development:

Rust has a big potential for game development, there are already game engines



Metric	Old server		New server	Improvement
Peak average per-worker memory usage	4.2gb	⟶	1.1gb	3.8x smaller
Peak average per-machine CPU usage	24%	⟶	4%	6x smaller
Peak average file serve time	2s	⟶	0.2s	10x faster
Peak worst-case save time	82s	⟶	5s	16.4x faster

Figure 2: Rust at Production at Figma

like Amethyst written in rust which allow for complete games to be written in Rust. *(Gillen, 2020)*

4. BNF and EBNF

Backus Naur Form or BNF was first established by John Bakus along with Peter Naur in the 60s. BNF is a formal mathematical method to specify context-free grammar; a way to clear ambiguity and be precise. After BNF, an extended Backus-Naur form or EBNF was introduced. EBNF is a few extensions added to the original BNF that makes expressing grammar more convenient. Although everything that can be expressed using EBNF can be expressed using BNF as well. (*BNF and Ebnf - DePaul University*)

4.1. Identifiers

Identifiers in Rust should only start with a letter or an underscore "_", and after that, it can be any combination of letters, digits, or underscores. Note that a letter can be an identifier on its own, but an underscore should be followed by any other character and cannot be an identifier on its own.

Rust is a case-sensitive language. Layan and layan will be different identifiers in rust. *(Klabnik & Nichols,2019).*

To create identifiers, we follow the following BNF:

BNF of Identifiers.
<identifier> ::= <letters> | <letters> (<letters> | <digits> | "_")+ | "_" (<letters> | <digits> | "_")+
<letters> ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"
<digits> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

Table 1:BNF of Identifiers for Rust

4.2. Declaration of Variables

To declare a variable in Rust, first, you need to start with the word "let" followed by the variable name, then an assignment operator "=" followed by the value of the variable. The default data type for the variables is i32 which is a signed integer that has 32 bits. Or you can assign a datatype manually by typing the data type right after the variable name.

You can also declare the variable first along with its type, then assign a value to it later on another line. But take note that variables in rust are immutable. Once you assign a value to a variable you cannot change its value.

A solution to the mutability in rust is to add “mut” before the variable name. Once you put “mut” you are allowed to change the variable’s value after assigning an initial value to it. *(Klabnik & Nichols,2019).*

The BNF for the declaration would be something like this:

BNF of variable declaration.

```
        <declaration> ::= "let " <variableName> (" :" | ":" | ": " | " : ") <type> " = " <value> (";"
| " ;")
        | "let " <variableName> " = " <value> (";" | " ;")
        | "let " <variableName> (" :" | ":" | ": " | " : ") <type> (";" | " ;" | "; " | " ; ")
<variableName> " = " <value> (";" | " ;")
        | "let " <variableName> (";" | " ;" | "; " | " ; ") <variableName> " = " <value> (";" | " ;")
        | "let mut " <variableName> " = " <value> (";" | " ;" | "; " | " ; ") <variableName> " = "
<value> (";" | " ;")

        <type> ::= "i8" | "u8" | "i16" | "u16" | "i32" | "u32" | "i64" | "u64" | "i128" | "u128" |
"isize" | "usize" | "98_222" | "0xff" | "0o77" | "0b1111_0000" | "b'A'" | "f32" | "f64" | "bool" |
"tup" | "char"

        <variableName> ::= <identifier>

        <identifier> ::= <letters> | <letters> ( <letters> | <digits> | "_" )+ | "_" ( <letters> |
<digits> | "_" )+

        <letters> ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" |
"O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" | "a" | "b" | "c" | "d" | "e" | "f"
| "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" |
"z"

        <digits> ::= [0-9]+

        <value> ::= ([0-9]+ | ([a-z])+)
```

Table 2:BNF of variable declaration for Rust

4.3. Expressions

In programing language, an expression is an action in the program that creates a new value. Rust is anexpression-oriented language. This means any command in Rust language is an expression including statements*(Blandy & Orendorff,2021).* In this project we will examine these expressions in Rust:

1- Arithmetic Operators
2- Relational Operators
3- Logical Operators
4- Bitwise Operators

4.3.1. Arithmetic operators

The arithmetic operator operates between two digits and returns a digit. The arithmetic Operator operates between two digits and returns a digit.

Assume a=2 and b=1.

Operator	Description	Example
+	Addition Operator	a+b =3
-	Subtraction Operator	a-b=1
*	Multiplication Operator	a*b=2
/	Division Operator	a/b=2
%	Remainder Operator returns the remains from the division operation	a%b=0
-	Negation Operator Return the negative of the digit	-a=-2 -b=-1

Table 3:Arithmetic operators *(Blandy & Orendorff,2021).*

BNF of Arithmetic operators
<exp> ::= <exp> + <exp> | <exp> - <exp> | <exp> * <exp> | <exp> / <exp> | <exp>% <exp> | - <exp> (<exp>)| <digit> <exp> ::=[< digits >..< digits >{<digit>}] <digits> ::= [0-9]+

Table 4:BNF of Arithmetic operators

4.3.2. Relational operators

The relational Operator compares two digits and returns a Boolean value. Whether the expression is true or not *(Blandy & Orendorff,2021).*

Assume a=2 and b=1.

Operator	Description	Example
>	Greeter than	a>b = true
<	Lesser than	a<b = false
>=	Greeter than or equal	a>= b = true
<=	Lesser than or equal	a<=b = false
==	Equality	a==b = false
!=	Not equal	a!=b = true

Table 5:Relational operators

(Blandy & Orendorff,2021).

BNF of Relational operates
<exp> ::= <exp> > <exp> | <exp> < <exp> | <exp> >= <exp> | <exp> <= <exp> | <exp> == <exp> | <exp> != <exp> (<exp>)| <digit> <exp> ::=[< digits >..< digits >{<digit>}] <digits> ::= [0-9]+

Table 6:BNF of Relational operators

4.3.3. Logical operators

A logical operator operates between two Boolean values and returns a Boolean value, whether the expression is true or not.

Assume a=2 and b=1.

Operator	**Description**	**Example**
&&	And	(a>b && a<b) = false
||	OR	(a>b || a<b) = true
!	NOT	!(a>b) = true

Table 7: Relational operators

(Blandy & Orendorff,2021).

BNF of Logical operates
<exp> ::= <exp> && <exp> | <exp> || <exp> | ! <exp> (<exp>)| <Boolean> <exp> ::=[< Boolean >..< Boolean >{< Boolean >}] < Boolean > ::= true | false

Table 8:BNF of Relational operators

4.3.4. Bitwise operates

Bitwise Operator operates between two binary numbers and returns a numeric value*(Blandy & Orendorff,2021)*.

Assume a=2 and b=1.

Operator	Description	Example
&	Bitwise AND	a & b =a
|	Bitwise OR	a | b =b
<<	Left shift	a << 1 = 4
>>	Right shift	a >> 1 = 1
>>>	Right shift with zero	a>>> 1 = 1

Table 9:Bitwise operators

(Blandy & Orendorff,2021).

BNF of Bitwise operates
<exp> ::= <exp> &<exp> | <exp> | <exp> | <exp> << <exp> | <exp> >> <exp> | <exp> >>> <exp> | (<exp>)| <digit> <exp> ::=[< digits >..< digits >{<digit>}] <digits> ::= [0-9]+

Table 10:BNF of Bitwise operates

4.4. Conditional statements

All programming languages have conditional statements like "if" statements, that if a certain condition came true do a certain action*(Blandy & Orendorff,2021)* Rust has three basic conditional statements:

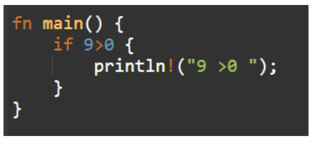
1- If statement.
2- If and else if statement.
3- If and else statement.

4.4.1. If statement

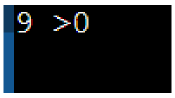
BNF of If statement
<if-statement> ::= if <condition> then <statement>
<statement>::=<assignment> | <if-statement> | < loop>
< condition >::=true | false

Table 11:BNF of If statement

Example



```
fn main() {
    if 9>0 {
        println!("9 >0 ");
    }
}
```



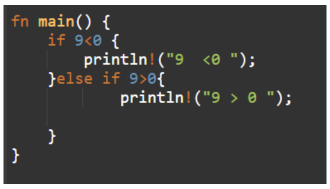
```
9 >0
```

4.4.2. If and else if statement

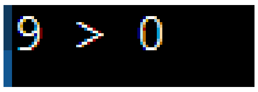
BNF of If and else if statement
<if-statement> ::= if <condition> then <statement>else if< condition > then <statement>
<statement>::=<assignment> | <if-statement> | < loop>
< condition >::=true | false

Table 12:BNF of If and else if statement

Example



```
fn main() {
    if 9<0 {
        println!("9  <0 ");
    }else if 9>0{
            println!("9 > 0 ");

    }
}
```



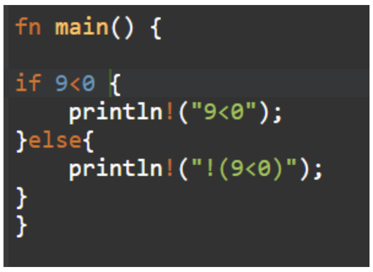
```
9 > 0
```

4.4.3. If and else statement

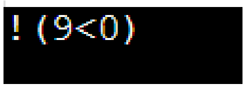
BNF of If and else statement
<if-statement> ::= if <condition> then <statement>else <statement>
<statement>::=<assignment> | <if-statement> | < loop>
< condition >::=true | false

Table 13:BNF of If and else statement

Example:



```
fn main() {

if 9<0 {
    println!("9<0");
}else{
    println!("!(9<0)");
}
}
```



```
!(9<0)
```

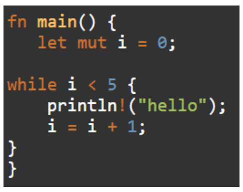
4.5. Loop statements

Loop statements in Rust are a way to iterate a block. There are two to implement a loop in rust for loop or while loop. For loop go over an array or list of numbers element by element. While loop executes block while a certain condition is true *(Blandy & Orendorff,2021)*.

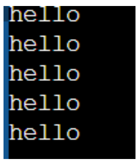
BNF of Loop statements
<loop statement >::= < while loop > | < for loop >
<while loop > ::= while (<condition>) <statement>
< for loop >::=for(<identifier > "in" <list>)<statement>

Table 14:BNF of Loop statements

Example while loop :

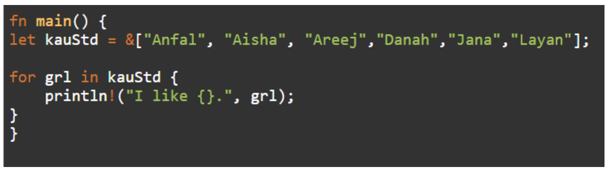


```
fn main() {
   let mut i = 0;

while i < 5 {
    println!("hello");
    i = i + 1;
}
}
```

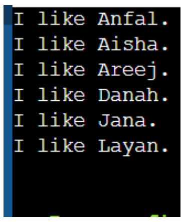


```
hello
hello
hello
hello
hello
```

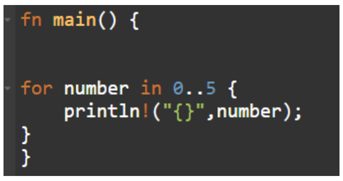
Example for loop:



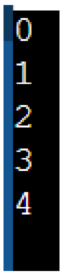
```
fn main() {
let kauStd = &["Anfal", "Aisha", "Areej","Danah","Jana","Layan"];

for grl in kauStd {
    println!("I like {}.", grl);
}
}
```



```
I like Anfal.
I like Aisha.
I like Areej.
I like Danah.
I like Jana.
I like Layan.
```



```
fn main() {

for number in 0..5 {
    println!("{}",number);
}
}
```



```
0
1
2
3
4
```

5. The evaluation of the Rust language

There are three important criteria by which programming languages can be compared: readability, writability, and reliability. These criteria are based on several characteristics that are briefly presented in this section. We will also discuss how Rust meets these criteria by looking at each of these characteristics (Sebesta, R. W. 2019)

5.1. Readability

By the simplicity of the language, syntax design, orthogonality, and ability to identify new data types, we can determine its readability (UKEssays,2018)

5.1.1. Simplicity

The lack of basic structures and concepts in the language indicates its simplicity. The more structures, the more difficult and complex the language is. Moreover, if a language has more than one way to express the same process that will increase its complicity of it. In the other words, if language is multifunctionality.

It's not easy to learn Rust, it features several basic concepts that are crucial to the use of the language. But its simplicity allows inexperienced users to work on production projects in systems programming while maintaining safety. And what makes Rust language easier to use is its lower multifunctionality. *(Klabnik & Nichols,2019).*

5.1.2. Operator Overloading

Operator overloading is a method by which object-oriented systems allow the same operator's name or symbol to be used for multiple operations. The symbol or operator name can be bound to more than one implementation of the operator. In Rust, many of the operators can be overloaded using traits. Several operators can be used for different tasks depending on their input arguments. (*Operator Overloading, Manifold. Retrieved 2020)*

5.1.3. Orthogonality

Orthogonality means that it is possible to combine all primitive constructs into an ensemble purposefully and comprehensively so that no special keyword is needed for each case. The number of exceptions will decrease the more the design is orthogonal because it is related to simplicity, the fewer exceptions, the easier it is to read and write programs in the programming language and learn it.

Symmetry and consistency are the main parameters. The pointer is an orthogonal concept, for example

In the language of Rust, many keywords can be combined in different ways, so it is considered a very orthogonal language.

5.1.4. Data Types

The ability to define appropriate data types and data structures for different situations also increases readability. In Rust, there are both traits and structs for defining our data types.

5.1.5. Syntax Design

Syntax design refers to the rules that define a language's structure. This refers to the rules that govern the structure of a programming language's symbols, punctuation, and words. It is nearly impossible to understand the meaning or semantics of a language without it.

The way the syntax is designed is important for readability and the meaning of the special words should also be clear. The general syntax of Rust is clean and simple, but since it must convey complex functions, it gets a bit long. Some concepts, such as lifetimes, are not presented very clearly. (*Compilers: Principles, Techniques, and Tools* 2007)

5.2. Writability

is a measure of how easily a language can be used to create programs for a specific problem domain, and A language's writability is defined by its readability, expressivity, and support for abstraction.

If a language is difficult to read, it is also difficult to write, because a programmer must be able to read their code repeatedly. *(Klabnik & Nichols,2019).*

5.2.1. Expressivity

Expressivity means that it is easy to express what you mean. This can conflict with readability because a language can be easy to write if there are many "handy" special expressions for different situations, but this can make it difficult to read if it becomes too much.
Rust is very expressive because it is easy to accomplish something with very little code.

5.2.2. Abstraction

The ability to define and use complicated structures or operations in ways that allow many details to be ignored is referred to as abstraction. support with abstraction, similar to objects in object-oriented languages, makes it easier to design a good program and think about the code in the same way that it is easier to think of objects than just data.

Traits are a form of abstraction in Rust that allows objects with similar properties to be used in similar ways without having to duplicate code. Similarly, generic parameters for functions and data types can be used to allow the use of arbitrary data types instead of concrete types, as long as they meet a set of specified requirements.

5.3. Reliability

The reliability of a programming language is all about how crash-proof the code is written in that language. Although, if it was performed to its specifications under all conditions. Reliability depends largely on how the code is written and only slightly on the language itself. Also, on how easy it is to program in that language.

The reliability of any programming language is defined by its type checking, exception handling, and limited aliasing *(Klabnik & Nichols,2019).*

5.3.1. Type Checking

Type checking is simply the process of testing each program for type errors, either by the compiler or during program execution. It's important to avoid runtime errors, which are very difficult to solve. Performing type checking at compile time rather than at runtime reduces the risk of runtime errors. Rust has a very strict type of checking at compile time.

5.3.2. Exception Handling

Exception handling is to detect errors and take corrective measures, which helps with increasing the reliability as it reduces runtime errors. Rust finds a lot of errors at compile time that would have been exceptions in other languages. During runtime, errors are handled by panic. (*Liskov, B.H.; Snyder, Exception Handling in CLU, 1979)*

5.3.3. Aliasing

Having two or more different variables refer to the same object. This can confuse if the value it contains is changed by some part of the program.

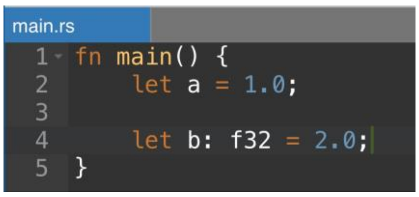
In Rust, aliasing is very restricted; there can never be more than one mutable variable at a time for the same object.

6. Types of variables

A variable is how a storage area within a computer program is referred to and held within a memory location as numbers, texts, or more complex types.

6.1. Declaration variables

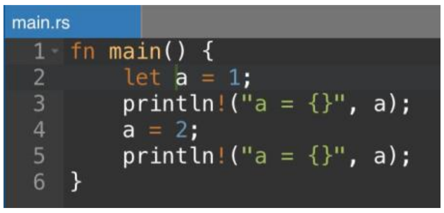
Variable declaration in Rust is as follows:



```
fn main() {
    let a = 1.0;

    let b: f32 = 2.0;
}
```

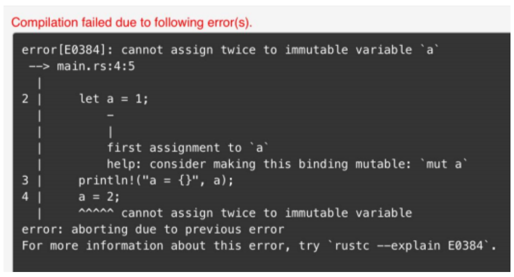
It can be either by mentioning the data type as declaring variable b or without mentioning any data types as declaring variable a *(Klabnik & Nichols,2019).*

Variables in Rust programming language are immutable, that’s means that the value of a variable cannot be changed once it is bound to a name. To understand the immutability, let’s consider the following example of changing the value of the variable a:



```
fn main() {
    let a = 1;
    println!("a = {}", a);
    a = 2;
    println!("a = {}", a);
}
```

When we run this code, we will receive an error message as shown:



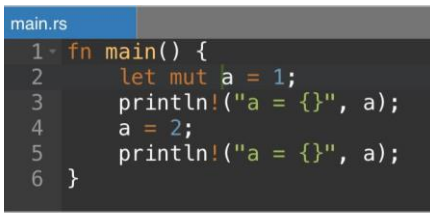
Compilation failed due to following error(s).

```
error[E0384]: cannot assign twice to immutable variable `a`
 --> main.rs:4:5
  |
2 |     let a = 1;
  |         -
  |         |
  |         first assignment to `a`
  |         help: consider making this binding mutable: `mut a`
3 |     println!("a = {}", a);
4 |     a = 2;
  |     ^^^^^ cannot assign twice to immutable variable
error: aborting due to previous error
For more information about this error, try `rustc --explain E0384`.
```

This error occurred because we tried to change or assign a new value of the perused variable a.

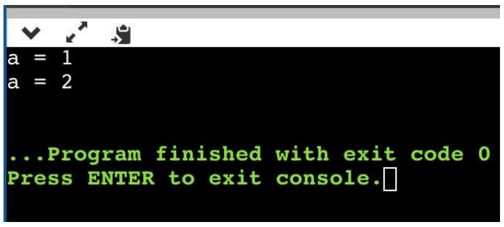
As we saw in the previous example, variables in Rust are declared by using the keyword *let* followed by a name of the variable, the equal operator then values to be assigned.

Although immutability is very useful in preventing the situations that lead to bugs that are difficult to track, the mutability is also very useful as it can make the code simpler and more convenient to write. Now how we can make variables mutable if they are immutable by default? The answer is by adding *mut* before the variable’s name. Let’s try to fix the previous example:



```
main.rs
fn main() {
    let mut a = 1;
    println!("a = {}", a);
    a = 2;
    println!("a = {}", a);
}
```

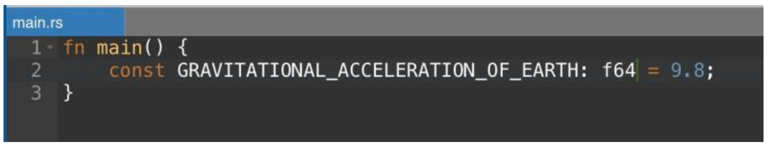
What will be printed in the output is as follows:



```
a = 1
a = 2


...Program finished with exit code 0
Press ENTER to exit console.
```

We can see that the value bound to a is changed successfully to 2 *(Klabnik & Nichols,2019).*

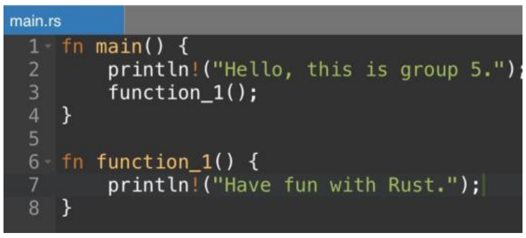
Sometimes we need to bind a value that will never be changed to a name. In this case, it is preferable to use constant instead of variable. Constant declaration syntax is as follows:



```
main.rs
fn main() {
    const GRAVITATIONAL_ACCELERATION_OF_EARTH: f64 = 9.8;
}
```

As we can see, the constant is declared using the *const* keyword instead of the *let* keyword. The name of the constant is in uppercase with underscores between single words. Unlike variable declaration, the data type must be mentioned in the constant declaration, in the above example, the constant is of floating-point type.

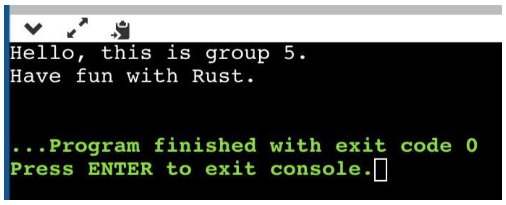
Functions in Rust are defined by the keyword *fn* followed by a function name and then a set of parentheses. Here is an example of defining a function other than the most important function which is the *main()* function:



main.rs

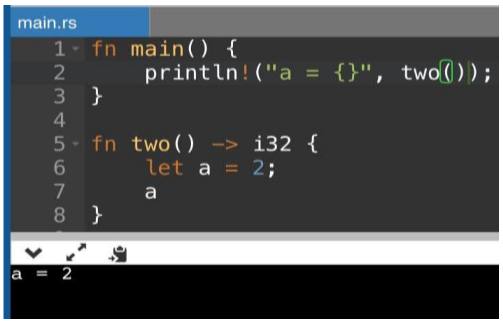
```
fn main() {
    println!("Hello, this is group 5.");
    function_1();
}

fn function_1() {
    println!("Have fun with Rust.");
}
```

In Rust, we can call any function we have defined even if it is defined before or after the calling function. Rust does not care where the function has been defined. As shown above, the calling of *function_1()* is done inside the *main()* function. The output is:



```
Hello, this is group 5.
Have fun with Rust.


...Program finished with exit code 0
Press ENTER to exit console.
```

The above example shows that *function_1()* does not return any values. We can define a function that returns an integer as follows:



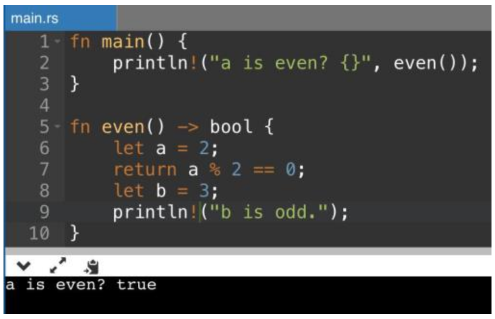
main.rs

```
fn main() {
    println!("a = {}", two());
}

fn two() -> i32 {
    let a = 2;
    a
}
```

```
a = 2
```

(Klabnik & Nichols,2019).

The function that returns a value has the same definition in addition to an arrow followed by the data type of the value to be returned. In Rust, the function returns the value of the last expression in the function's body implicitly even if we did not use the return keyword, but if we want to return a value early, we must use the keyword return. Here we have an example of a function that returns a value that is not in the last expression:

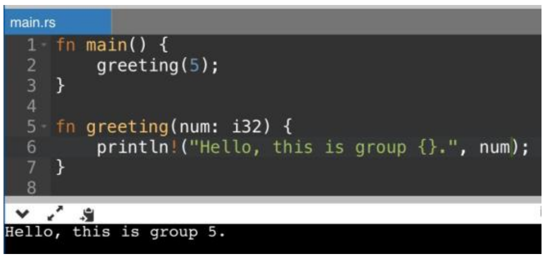


main.rs

```rust
fn main() {
    println!("a is even? {}", even());
}

fn even() -> bool {
    let a = 2;
    return a % 2 == 0;
    let b = 3;
    println!("b is odd.");
}
```

```
a is even? true
```

We also can pass one or more values to a function when we call it. This value represents the parameter of a function which is a part of the function's signature. Here is an example of passing a single value to a function:

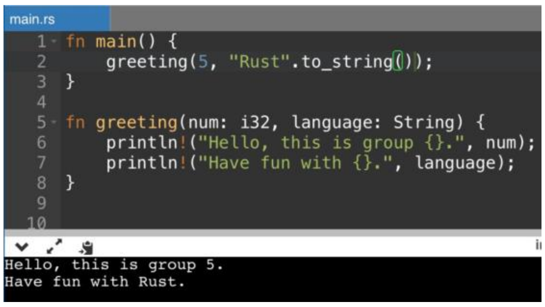


main.rs

```rust
fn main() {
    greeting(5);
}

fn greeting(num: i32) {
    println!("Hello, this is group {}.", num);
}

```

```
Hello, this is group 5.
```

Also, we can pass multiple values as follows:



main.rs

```rust
fn main() {
    greeting(5, "Rust".to_string());
}

fn greeting(num: i32, language: String) {
    println!("Hello, this is group {}.", num);
    println!("Have fun with {}.", language);
}


```

```
Hello, this is group 5.
Have fun with Rust.
```

(Klabnik & Nichols,2019).

6.2. Storage

To understand the lifetime for variables in Rust first we need to go over how Rust manages the memory. In programming languages, there are two common ways to manage the memory either by garbage collector or manual memory management. A high-level language is based on the garbage collector, the programmers are not worried about the memory management. But, On the other hand, there is no control over memory and slow runtime. Low-level programming languages are manual memory management, which means allocating and deallocating memory. The pros of this are full control with memory and small program size. On the other side, the cons are extremely error-prone and slower writing time *(Gilmore, 2007).*

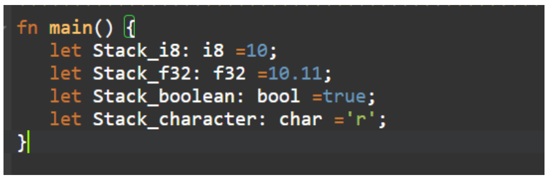
However, Rust is a System language that cares about memory, performance, and safety. Rust can't use garbage collector or manual memory management because as we mentioned in garbage collector there is no control over memory. Also, manual memory management is error-prone *(Klabnik & Nichols,2019).*

Therefore, Rust manages memory using the ownership model. Allows Rust to make memory safety guarantees without of use a garbage collector. Unfortunately, it is slower to write than memory manual management but it is worth it *(Klabnik & Nichols,2019)*.

6.2.1. Stack

During Run time rust makes certain decisions based on if our memory is on the stack or the heap. Stack variables are used for fast memory creation and retrieval to make the program extremely fast. memory management of stack variables is very easy, and the memory of the variables is automatically recaptured by the program after it goes out of scope. Rust uses the stack by default for memory needs. Stack is a fixed size and can't control the size in the running time. As well l stack creates a stack frame for every function. That stores the local variables of the function. variable in the frames lives if the stack frame lives *(Klabnik & Nichols,2019)*.

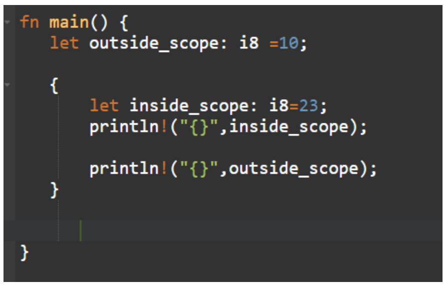
Variables that store on the stack are any scalar types (integers, floating-point numbers, Booleans, characters and array fixed size).



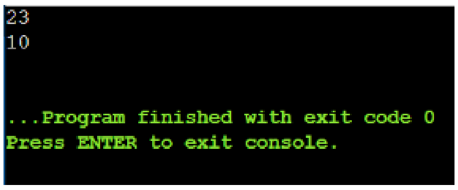
```
fn main() {
    let Stack_i8: i8 =10;
    let Stack_f32: f32 =10.11;
    let Stack_boolean: bool =true;
    let Stack_character: char ='r';
}
```

Each of these has a memory size that is known to Rust at compile time. Even though those types can hold a range of values they will take fixed in size. For example, Stcak_i8 will take 8 bits even if we mutate the value. Rust cleans up all memory associated with the scope once the scope exits. In our example, it’s at the min curly bracket.

Let's do more examples so that you get the idea. any things that have a curly bracket will create a new scope.



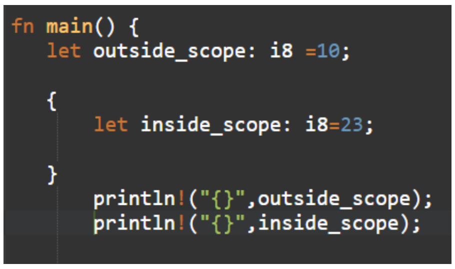
```
fn main() {
    let outside_scope: i8 =10;

    {
        let inside_scope: i8=23;
        println!("{}",inside_scope);

        println!("{}",outside_scope);
    }

}
```



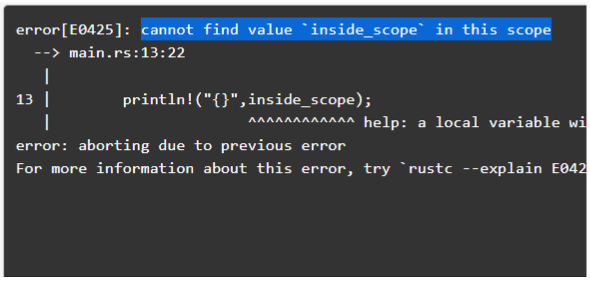
```
23
10

...Program finished with exit code 0
Press ENTER to exit console.
```

The Outside_scope variable will be placed first in the stack since it’s the first variable from the main curly bracket. Then inside_scope variable will be placed on top of the Outside_scope variable. As you can see the Outside_scope variable is still in the stack to our min program run just fine.

After the curly bracket for the inner scope, the stack will pop the inside_scope variable out. So, it’s no longer in the stack.



```
fn main() {
    let outside_scope: i8 =10;

    {
        let inside_scope: i8=23;

    }
        println!("{}",outside_scope);
        println!("{}",inside_scope);
```



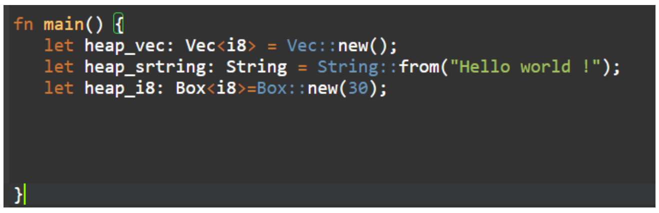
```
error[E0425]: cannot find value `inside_scope` in this scope
  --> main.rs:13:22
   |
13 |         println!("{}",inside_scope);
   |                       ^^^^^^^^^^^^ help: a local variable wi
error: aborting due to previous error
For more information about this error, try `rustc --explain E042
```

The same idea for function any variable that is created inside that function automatically gets cleaned up and the memory is freed once the end curly bracket is hit.

6.2.2. Heap

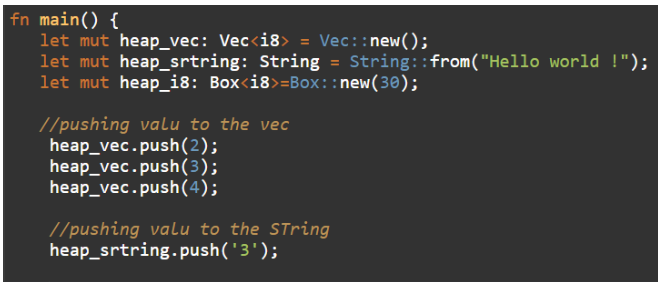
The heap on the other hand is less organized. Give us the flexibility we need in the same situation. We can change the size at the runtime of the heap. Data stored in the heap could be dynamic in size and we can control the lifetime of that variable in the heap. There is a runtime cost to using the heap to allocate memory. Any time the variables need to change in size means they can't live on the stack and must live on the heap. Memory can live beyond of scope that created it. Similar to the stack heap memory will be automatically cleaned up and recaptured when the last owner goes out of scope. In the following section, we will talk about ownership. To allocate memory on the heap using a new keyword *(Klabnik & Nichols,2019)*.

Variables that store on the heap are (box type, String, Vectors, and collection). Or any variable can change in size.



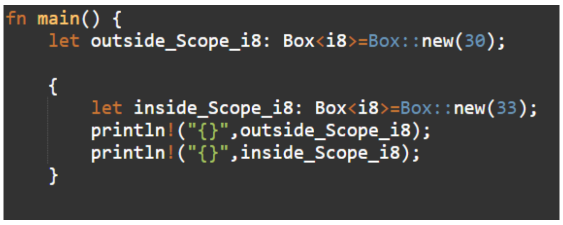
```
fn main() {
    let heap_vec: Vec<i8> = Vec::new();
    let heap_srtring: String = String::from("Hello world !");
    let heap_i8: Box<i8>=Box::new(30);



}
```

As we said Heap is flexible. It's possible to change the value of a variable in compiling.

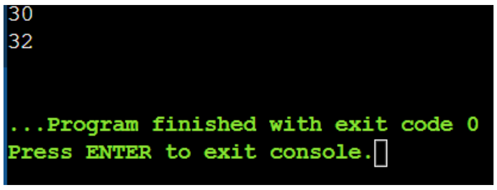


```
fn main() {
    let mut heap_vec: Vec<i8> = Vec::new();
    let mut heap_srtring: String = String::from("Hello world !");
    let mut heap_i8: Box<i8>=Box::new(30);

    //pushing valu to the vec
     heap_vec.push(2);
     heap_vec.push(3);
     heap_vec.push(4);

     //pushing valu to the STring
     heap_srtring.push('3');
```

The outside_Scope_i8 is pushed first to the heap. Then inside_Scope_i8 will be pushed beyond the curly bracket into the inner scope. inside_Scope_i8 will pop out of the heap.

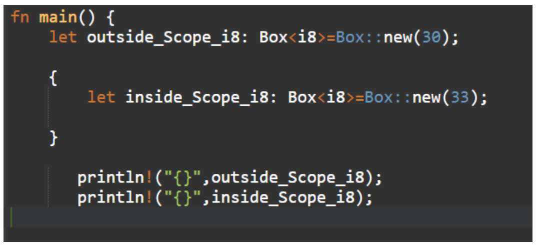


```
fn main() {
    let outside_Scope_i8: Box<i8>=Box::new(30);

    {
        let inside_Scope_i8: Box<i8>=Box::new(33);
        println!("{}",outside_Scope_i8);
        println!("{}",inside_Scope_i8);
    }
```

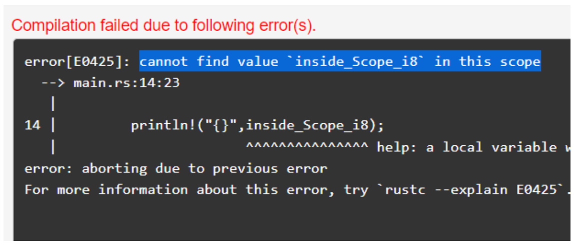


```
30
32


...Program finished with exit code 0
Press ENTER to exit console.
```

Since the inside_Scope_i8 has been popped out. inside_Scope_i8 is no longer stored in the heap



```
fn main() {
    let outside_Scope_i8: Box<i8>=Box::new(30);

    {
        let inside_Scope_i8: Box<i8>=Box::new(33);

    }

       println!("{}",outside_Scope_i8);
       println!("{}",inside_Scope_i8);
```



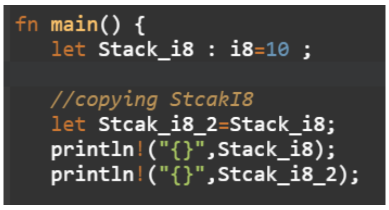
```
Compilation failed due to following error(s).
error[E0425]: cannot find value `inside_Scope_i8` in this scope
  --> main.rs:14:23
   |
14 |        println!("{}",inside_Scope_i8);
   |                      ^^^^^^^^^^^^^^^ help: a local variable w
error: aborting due to previous error
For more information about this error, try `rustc --explain E0425`.
```

6.2.3. Ownership

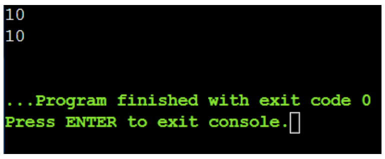
Now let us talk about Ownership. Each value in Rust has a variable called owner, one owner for one value, when the owner gets out of the scope the value will be dropped. There can only be one owner at a time, variable can't have two owners at the same time.

Assume that you want to copy a value to another variable. In stack, Rust has a copy trait as a simple type stored on the stack such as integers. This trait allowed the stack to copy the value. On the other hand, copying values in heap isn't allowed. The value can move but not copy. This means that will happen Rust will drop the variable that is copying from the value. And the other variable will own the value. *(Klabnik & Nichols,2019)*.

Look at this example in the stack variable, the value is copied just fine. Moreover, the variable Stack_i8 is not dropped.

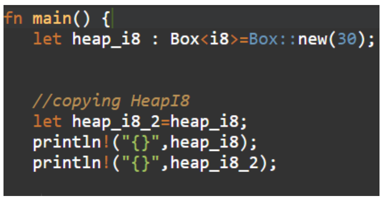


```
fn main() {
    let Stack_i8 : i8=10 ;

    //copying StcakI8
    let Stcak_i8_2=Stack_i8;
    println!("{}",Stack_i8);
    println!("{}",Stcak_i8_2);
```

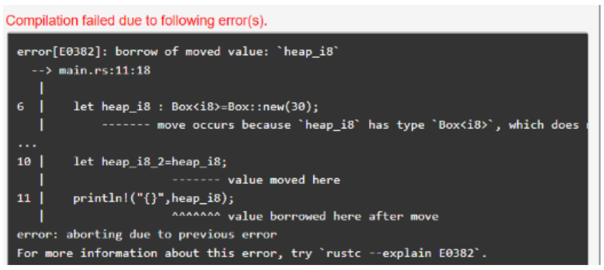


```
10
10


...Program finished with exit code 0
Press ENTER to exit console.
```

Although if we apply the same action to the heap we will find an error, that the value in heap_i8 is removed.



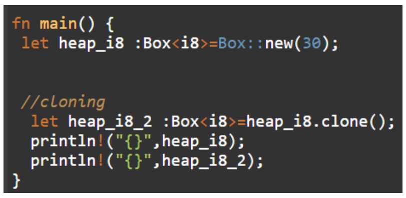
```
fn main() {
   let heap_i8 : Box<i8>=Box::new(30);


   //copying HeapI8
   let heap_i8_2=heap_i8;
   println!("{}",heap_i8);
   println!("{}",heap_i8_2);
```



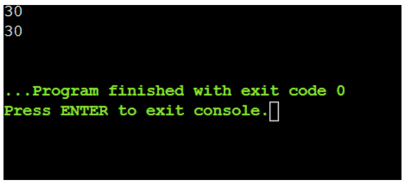
Compilation failed due to following error(s).

```
error[E0382]: borrow of moved value: `heap_i8`
  --> main.rs:11:18
   |
6  |    let heap_i8 : Box<i8>=Box::new(30);
   |        ------- move occurs because `heap_i8` has type `Box<i8>`, which does
...
10 |    let heap_i8_2=heap_i8;
   |                  ------- value moved here
11 |    println!("{}",heap_i8);
   |                  ^^^^^^^ value borrowed here after move
error: aborting due to previous error
For more information about this error, try `rustc --explain E0382`.
```

So how do we copy a value that is sorted on the heap? We can clone value instead of copying it by calling the clone method.



```
fn main() {
 let heap_i8 :Box<i8>=Box::new(30);


 //cloning
  let heap_i8_2 :Box<i8>=heap_i8.clone();
  println!("{}",heap_i8);
  println!("{}",heap_i8_2);
}
```



```
30
30


...Program finished with exit code 0
Press ENTER to exit console.
```

6.3. Life Time

A variable's lifetime begins when it is created and ends when it is destroyed. Lifetimes in Rust is about ensuring that memory does not get cleaned up before a reference can use it. Duo to the memory management tradeoffs Rust lifetime is a difficult topic. Lifetimes enforce a piece of memory is still valid for a reference. The data stored on the heap live longer than on the stack. *(Klabnik & Nichols,2019).*

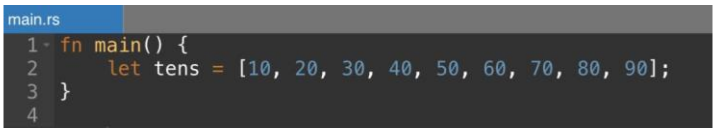
7. Types of Arrays in Rust

An array is a way to have a collection of values. Arrays in rust are fixed in length, which means their size cannot shrink or grow once declared. Also, each element in the array must have the same type. As we mentioned in section 6.2.1 Array type is sorted on the stack because the size is fixed in the Array type.

7.1. Array type

7.1.1. Creating array

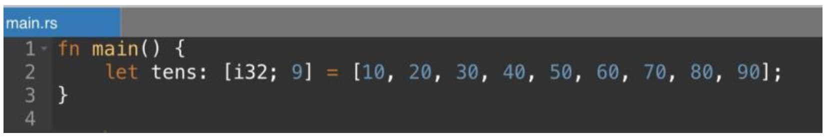
We create an array as follows:



main.rs

```rust
fn main() {
    let tens = [10, 20, 30, 40, 50, 60, 70, 80, 90];
}
```

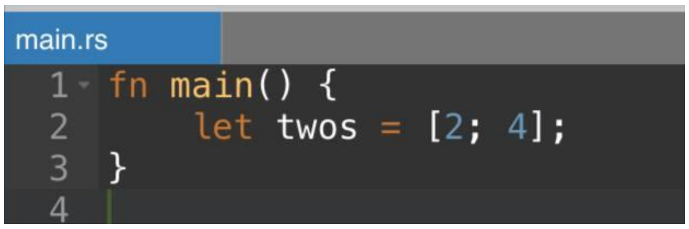
In the above statement, the type and number of elements are implicitly determined. If we want to specify them explicitly, we can define the array as the following statement which is equivalent to the statement above.



main.rs

```rust
fn main() {
    let tens: [i32; 9] = [10, 20, 30, 40, 50, 60, 70, 80, 90];
}
```

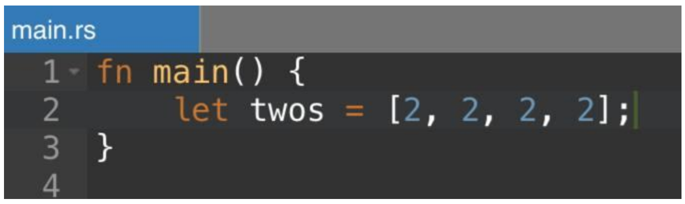
If we want to set all array elements to the same value, we can write this statement:



main.rs

```rust
fn main() {
    let twos = [2; 4];
}
```

Here, there are 4 elements all with the same value which is 2, which is equivalent to this statement:



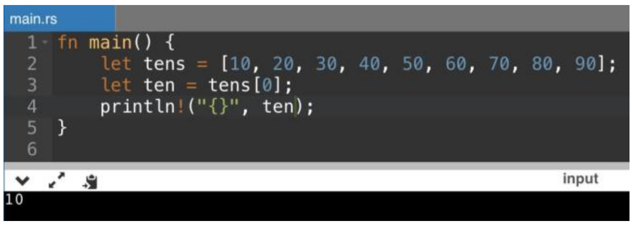
main.rs

```rust
fn main() {
    let twos = [2, 2, 2, 2];
}
```

(Klabnik & Nichols,2019).

7.1.2. Accessing array elements

Array elements can be accessed by index. Here are some examples of accessing elements of an array:

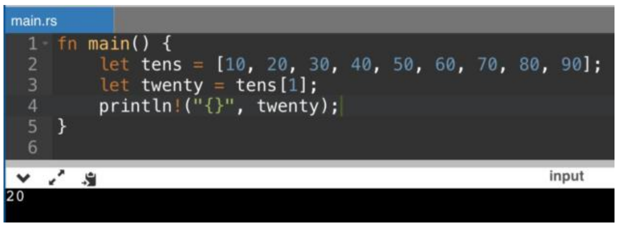


main.rs

```
fn main() {
    let tens = [10, 20, 30, 40, 50, 60, 70, 80, 90];
    let ten = tens[0];
    println!("{}", ten);
}
```

input

```
10
```

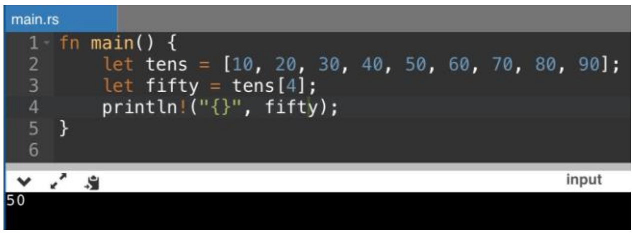


main.rs

```
fn main() {
    let tens = [10, 20, 30, 40, 50, 60, 70, 80, 90];
    let twenty = tens[1];
    println!("{}", twenty);
}
```

input

```
20
```



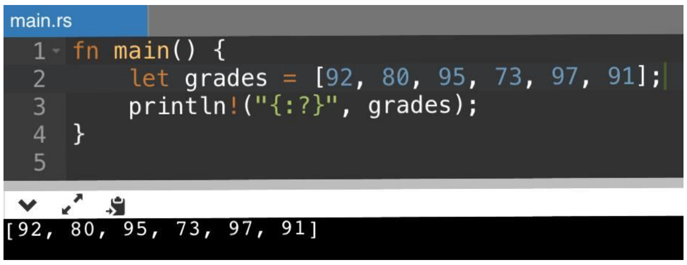
main.rs

```
fn main() {
    let tens = [10, 20, 30, 40, 50, 60, 70, 80, 90];
    let fifty = tens[4];
    println!("{}", fifty);
}
```

input

```
50
```

Also, here is an example of printing all elements in the array.



main.rs

```
fn main() {
    let grades = [92, 80, 95, 73, 97, 91];
    println!("{:?}", grades);
}
```

```
[92, 80, 95, 73, 97, 91]
```

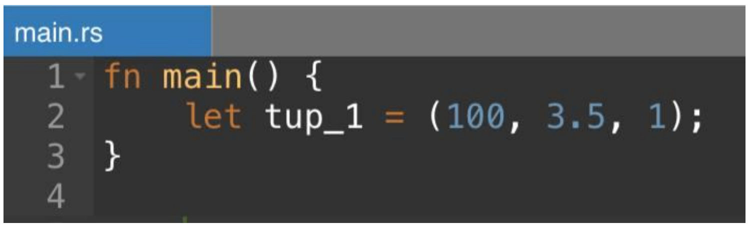
(Klabnik & Nichols,2019).

7.2. Tuple type

An array type is one of two primitive compound types that group multiple values into one type. Another compound type is the tuple type. A tuple is a way to group multiple values. Also, tuples have fixed lengths like arrays. Unlike an array, elements in a tuple could be of different types *(Klabnik & Nichols,2019).*

7.2.1. Creating tuples

We can create a tuple like this:

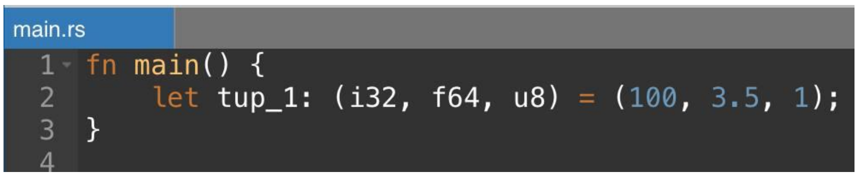


main.rs

```
fn main() {
    let tup_1 = (100, 3.5, 1);
}
```

Or we can create it by annotating the types of each element in the tuple:

(Klabnik & Nichols,2019).

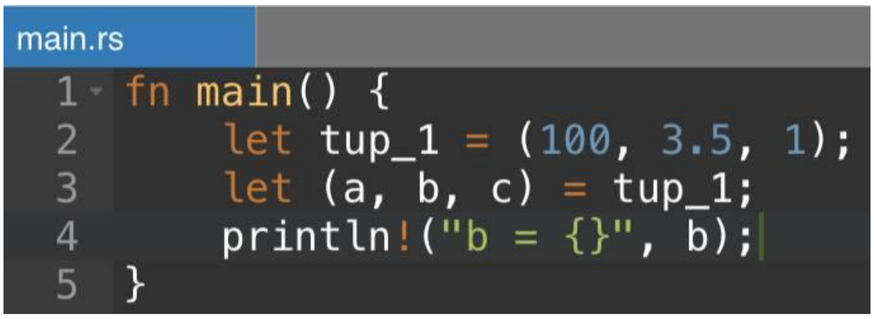


main.rs

```
fn main() {
    let tup_1: (i32, f64, u8) = (100, 3.5, 1);
}
```

7.2.2. Accessing tuple elements

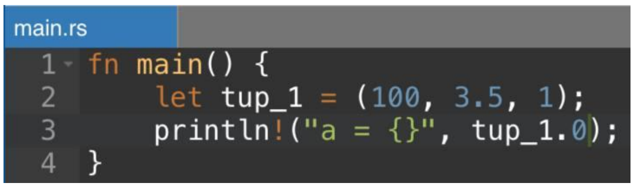
Tuple elements can be accessed by destructuring the tuple values. Destructuring is done by breaking the tuple into parts then every part will be bound to a variable (*Klabnik & Nichols,2019).* The following statements represent an example:



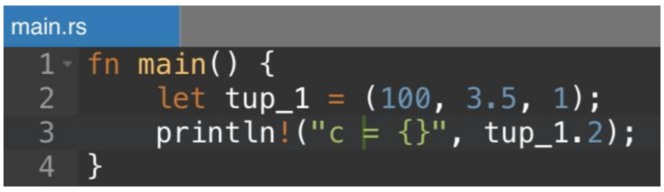
main.rs

```
fn main() {
    let tup_1 = (100, 3.5, 1);
    let (a, b, c) = tup_1;
    println!("b = {}", b);
}
```

100, 3.5, and 1 are bound to a, b and c respectively, then the value of variable b is printed, which is 3.5. Another way of direct access to a tuple element is by indexing. It is done by using a tuple name followed by a dot "." and the index. Here are some examples:



main.rs
```rust
fn main() {
    let tup_1 = (100, 3.5, 1);
    println!("a = {}", tup_1.0);
}
```



main.rs
```rust
fn main() {
    let tup_1 = (100, 3.5, 1);
    println!("c = {}", tup_1.2);
}
```

(Klabnik & Nichols,2019).

7.3. Collection

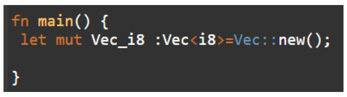
Collection allowed us to store multiple values together. Unlike the Array or tuple, it can grow in size. Collection is stored in the heap; the size of any collection can grow or shrink as needed. The most common collections are Vectors and Strings *(Blandy & Orendorff,2021).*

7.3.1. Vectors

Vectors are beefed up Arrays on steroids. This data structure is where you store a sequence of elements inside the heap *(Blandy & Orendorff,2021).*

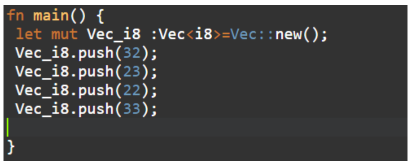
7.3.1.1. Creating Vectors

To make a vector you must set the vector's data type and use the *new* function to allocate the memory on the heap as we point out in section 6.2.2.



```rust
fn main() {
 let mut Vec_i8 :Vec<i8>=Vec::new();

}
```

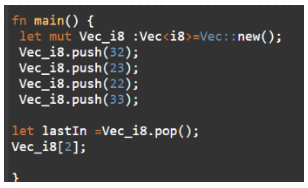
To insert the data inside the vectors, use the *push* keyword.



```rust
fn main() {
 let mut Vec_i8 :Vec<i8>=Vec::new();
 Vec_i8.push(32);
 Vec_i8.push(23);
 Vec_i8.push(22);
 Vec_i8.push(33);

}
```

7.3.1.2. Accessing Vectors

Accessing Vectors can be done in two ways, indexing or pop the last value has been pushed.



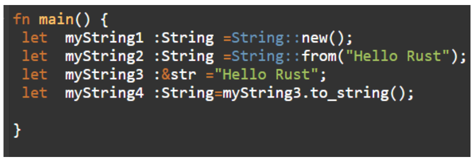
```rust
fn main() {
 let mut Vec_i8 :Vec<i8>=Vec::new();
 Vec_i8.push(32);
 Vec_i8.push(23);
 Vec_i8.push(22);
 Vec_i8.push(33);

let lastIn =Vec_i8.pop();
Vec_i8[2];

}
```

7.3.2. String

Rust string is stored as a collection of utf-8 encoded bytes. utf-8 is a variable with character encoding for Unicode. utf-8 could be represented as one byte, two bytes, three bytes, or four bytes *(Blandy & Orendorff,2021).*

7.3.2.1. Creating String

We have four ways to create String in Rut.



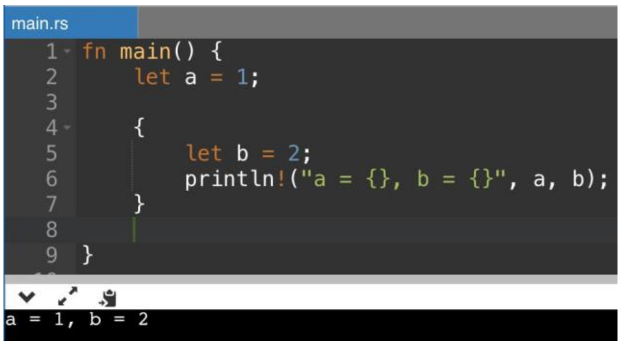
```rust
fn main() {
 let  myString1 :String =String::new();
 let  myString2 :String =String::from("Hello Rust");
 let  myString3 :&str ="Hello Rust";
 let  myString4 :String=myString3.to_string();

}
```

The first way will be by creating an empty string with a function *new* like the Vectors. The second way is using the function *from*. The third way is to define string &str. lastly, turn the string &str into own string using *.to_string()* method.

The difference between &str and String is that &str stores in binary on the stack, while String is stored in utf-8 on the heap as mentioned earlier.

8. Types of Scope in Rust

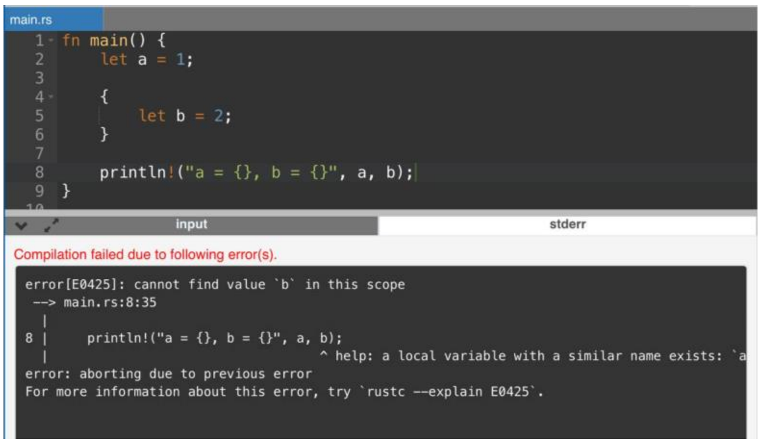
A scope is a section in a program where a variable is visible or valid. Variables in Rust are valid only inside the block where they have been defined which is delimited by curly brackets. So, Rust can be considered as a statically scoped language. Let's see this example:



```
fn main() {
    let a = 1;

    {
        let b = 2;
        println!("a = {}, b = {}", a, b);
    }

}
```

```
a = 1, b = 2
```

From this example, we can see that the inner block can access the variable in the outer block, but the reverse is not possible as shown below:

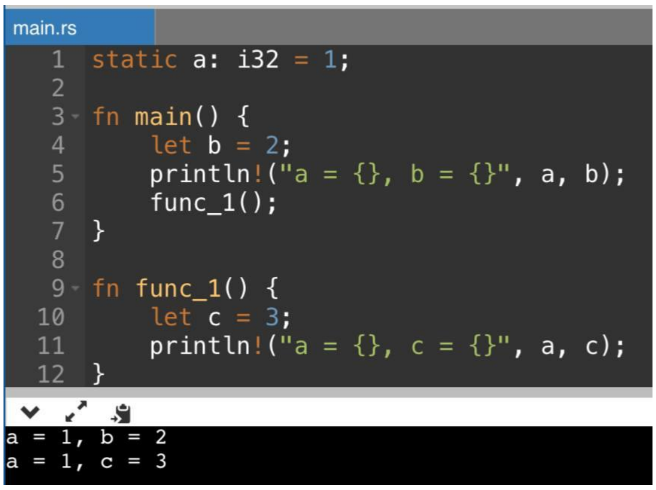


```
fn main() {
    let a = 1;

    {
        let b = 2;
    }

    println!("a = {}, b = {}", a, b);
}
```

Compilation failed due to following error(s).

```
error[E0425]: cannot find value `b` in this scope
 --> main.rs:8:35
  |
8 |     println!("a = {}, b = {}", a, b);
  |                                   ^ help: a local variable with a similar name exists: `a
error: aborting due to previous error
For more information about this error, try `rustc --explain E0425`.
```

A variable is said to be a local variable in a block if it is declared in that block. We can also have a global variable that can be accessed by any block. Global variables must be declared outside all blocks using static keywords like this:



```
static a: i32 = 1;

fn main() {
    let b = 2;
    println!("a = {}, b = {}", a, b);
    func_1();
}

fn func_1() {
    let c = 3;
    println!("a = {}, c = {}", a, c);
}
```

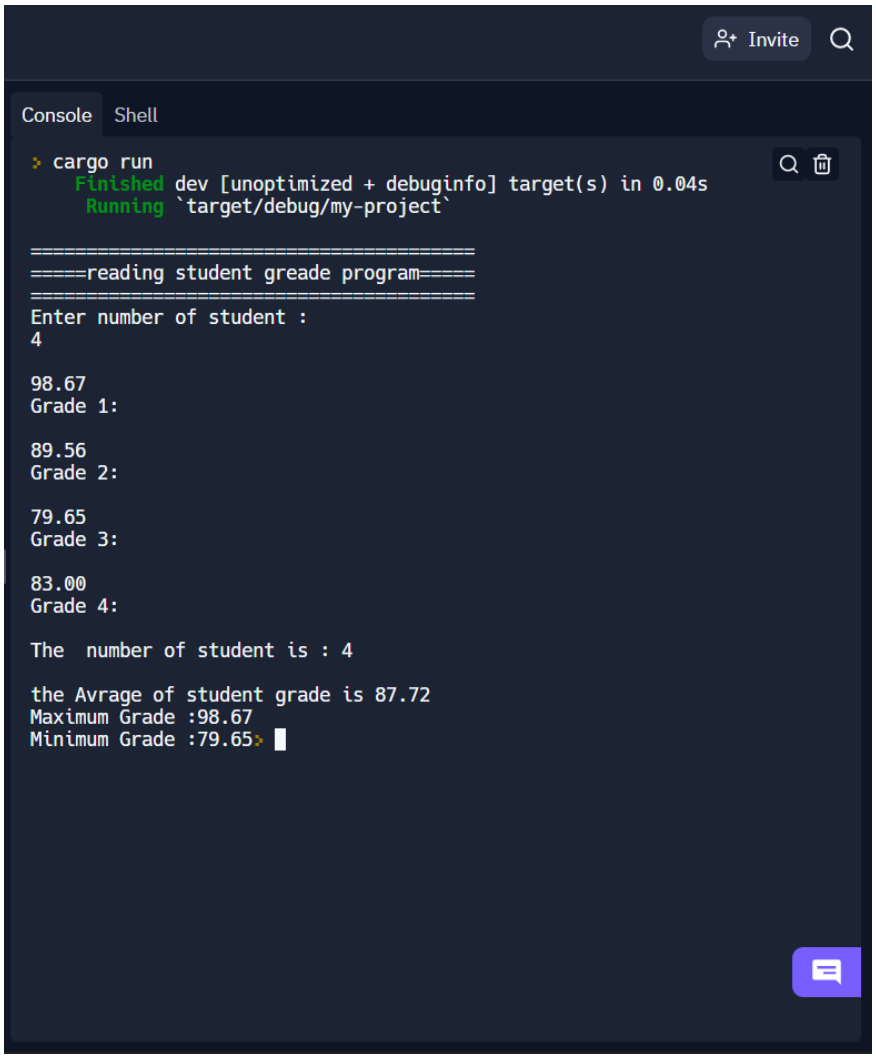
```
a = 1, b = 2
a = 1, c = 3
```

9. Program

9.1. Code

```rust
use std::io; // Rust Standard Library
use io::BufRead; // necessary to have `.read_line()` on
                 // locked stdin available

fn main() { //main function

    let stdin = io::stdin(); // handle to the standered input
print!("\n=======================================");
print!("\n=====reading student greade program=====");
print!("\n=======================================\nEnter number of student :");
print!("\n");
    let mut locked_stdin = stdin.lock();//handel for scoping,reading data
    let mut buffer = String::new(); //red line
    locked_stdin .read_line(&mut buffer).expect("Need number of student");
     //lock buffer to make it like circular array
    let numberofgrades: usize = buffer.trim().parse().expect("Need a number");
   //pointer size and remove the white space and convert string to intger
    buffer.clear(); //set the buffer
    let  mut gred:Vec<f32> = Vec::with_capacity(numberofgrades);
 //create vec with initial size
     //reade the greate
    for x in 0..numberofgrades { //for loop for number of student
        buffer.clear(); //make sure the buffer is clean
        print!("\nGrade {}: ",x+1);
        locked_stdin.read_line(&mut buffer).expect("Need Grade"); //read the greade
        gred.push(buffer.trim().parse().expect("Need a number")); //insert the value in the  vec and remove the white space and convert string to intger

      println!();
    }

   //find  the avg
    let numberofgrades = gred.len() as f32; //the size of the vec as float number
    let mut sum = 0.0;
    // This will consume the numbers.
    for n in  &gred{ //this case we refer to the element of the vec since we dont want to take the ownership
        sum += n;
    }
    let avg:f32=sum/numberofgrades; // find the avg be devied the sum and numberofgrades


   gred.sort_by(|a, b| a.partial_cmp(b).unwrap()); //sort to get easily the max and min

println!("\nThe  number of student is : {}", numberofgrades);
print!("\nthe Avrage of student grade is {:.2}",avg);
print!(" \nMaximum Grade :{}",gred[gred.len()-1]);
print!(" \nMinimum Grade :{}",gred[0]);
}

```

9.2. Run



```
> cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.04s
     Running `target/debug/my-project`

========================================
=====reading student greade program=====
========================================
Enter number of student :
4

98.67
Grade 1:

89.56
Grade 2:

79.65
Grade 3:

83.00
Grade 4:

The  number of student is : 4

the Avrage of student grade is 87.72
Maximum Grade :98.67
Minimum Grade :79.65>
```

References

Avram, A. (2012, August 3). *Interview on rust, a systems programming language developed by Mozilla*. InfoQ. Retrieved May 6, 2022, from https://www.infoq.com/news/2012/08/Interview-Rust/

Blandy, J., Orendorff, J., & Tindall, L. F. S. (2021). *Programming Rust: fast, safe systems development*. O'reilly.

BNF and Ebnf - DePaul University. (n.d.). Retrieved May 7, 2022, from https://condor.depaul.edu/ichu/csc447/notes/wk3/BNF.pdf

Gillen, T. (2020, September 7). Legion ECS - v0.3. *Amethyst Game Engine*. Retrieved May 6, 2022, from https://amethyst.rs/posts/legion-ecs-v0.3

Gilmore, S. (2007). *Advances in Programming Languages: Memory management* [Review of *Advances in Programming Languages: Memory management*]. https://homepages.inf.ed.ac.uk/stg/teaching/apl/handouts/memory.pdf

How Is Readability Important To Writability? (n.d.). Www.ukessays.com. Retrieved May 8, 2022, from https://www.ukessays.com/essays/information-technology/how-is-readability-important-to-writability-information-technology-essay.php#citethis

Klabnik, S., & Carol Nichols, C. (2019). *The Rust Programing Language* [Review of *The Rust Programing Language*]. William Pollock.

Liskov, B. H., & Snyder, A. (1979). Exception Handling in CLU. *IEEE Transactions on Software Engineering*, *SE-5*(6), 546–558. https://doi.org/10.1109/tse.1979.230191

Operator overloading. (2021, January 2). Wikipedia. https://en.wikipedia.org/wiki/Operator_overloading

Stack Overflow Developer Survey 2019. (n.d.). Stack Overflow. Retrieved May 8, 2022, from https://insights.stackoverflow.com/survey/2019?_ga=2.159089018.1567861010.1651979299-1491582577.1649330748#key-results

Sebesta, R. W. (2019). *Concepts of programming languages*. Pearson.

The Rust Core Team. (2015, May 15). Announcing rust 1.0: Rust blog. *The Rust Programming Language Blog*. Retrieved May 6, 2022, from https://blog.rust-lang.org/2015/05/15/Rust-1.0.html

Wallace, E. (2018, May 2). *How Mozilla's rust dramatically improved our server-side* performance. Figma. Retrieved May 6, 2022, from https://www.figma.com/blog/rust-in-production-at-figma/

9 Companies That Use Rust in Production. (n.d.). Serokell Software Development Company. Retrieved December 1, 2021, from https://serokell.io/blog/rust-companies

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). *Compilers Principles, Techniques, & Tools + Gradiance.* Addison-Wesley.