

|  |
| --- |
| RUST  Interpreter |
|  |
| November 18  Presented to you by:  Authored by:  Aseem Anand Aman Anand Chalavadi Pranav Luvkush Kumar Veerain Sood |

# Our Team and their contribution:

# Aseem Anand: Interpreter class, Function class, Environment class, Callable class ,Resolver class, Abstract syntax tree debugging, Expression class additions and debugging, Token class debugging, If else debugging, for loops debugging, while loops debugging,

# General debugging.

Veerain Sood: Main Class, Token Class , Scanner Implementation, Parser Class, Abstract Syntax Tree generation, Expression class, If else debugging, for loops debugging, while loop debugging ,Token class debugging, general debugging.

Aman Anand:

AST Generation, Automatic AST generator, Rust Info, Debugging

Luvkush Kumar:

Rust Info, Code Debugging, and Knowledge Collection.

Pranav:

Debugging Rust Info.

|  |
| --- |
| Please open it in vs code or any other editor of your choice: Please open the inner most MYOWN folder and then open it    If you encounter(high chances are) that you this error message then please follow the procedure mentioned below:  1) i  In the main file change this directory to your own directory(place the link of text.txt  Like I have done in the pic above)  2) Just click on debug button here (click continue if debug failed)    If this does not fix the issue then:  Please open all the files in vs code in the folder. VS Work!!  most probably opening it in IntelliJ will be all right I don’t know.  Optional  **Here you might need to install**    **This extension and hopefully it works, Please feel free to email us if you face any problems (most probably opening it in IntelliJ will be all right) all this was for vs code**  **As it has some weird bug whenever you send someone some code it requires you to debug it for it to properly work….**  -----------------------------------------------XXXXXXXXXX--------------------------------------------------  ReadMe:  Lets fire that dragon up!!:  Here is a test code:  fn main()  {  let mut a=1;  let mut b=2;  for(i=0;i<10;i=i+1)  {  a=a+1;  b=b+1;  if a<5  {  print!("a=");  print!(a);  }  else  {  print!("b=");  print!(b);  }  }  let mut j=0;  while j<10  {  if j<=5  {  print!("I am Aseem anand");  }  else  {  print!("I am Aman anand");  }  j=j+1;  }  }  main();  **output :**    **Example 2:**  **fn main()**  **{**  **let mut x=10;**  **let mut y=5;**  **let mut z=3.321;**  **print!(x+x\*((y-z)/2));**  **}**  **main();**  **output:** |

Overview

The Rust interpreter implemented in Java is a comprehensive project aimed at interpreting fundamental Rust programming language constructs. The interpreter encompasses tokenization, parsing, and interpretation components to facilitate the execution of Rust code. The primary features include variable declarations, for and while loops, functions, control flow structures, if-else statements, and print statements.

Components and Functionality

1. Tokenization

The `Token` class plays a pivotal role in the tokenization process. It generates tokens representing keywords, identifiers, literals, and operators. The tokenizer processes the source code, breaking it down into meaningful tokens, forming the basis for subsequent parsing.

2. Parsing

The `Parser` class transforms the stream of tokens generated by the tokenizer into a parse tree. This hierarchical representation captures the syntactic structure of the Rust code. The parser enforces Rust language syntax rules and ensures the code's adherence to the language specifications.

3. Interpretation

The heart of the interpreter lies in the `Interpreter` class, responsible for executing Rust code. This component handles variable declarations, loops, functions, control flow structures, if-else statements, and print statements. It interprets the parse tree and executes the code accordingly.

4. Variable Declaration

The interpreter supports the declaration of variables with only “let“ keyword

And by default variables immutable(didn’t implement(due to complexity)( but you do require to write let mut for things to work)

5. Control Flow

Control flow structures, including for loops and while loops, are implemented to enable the execution of code based on specific conditions or iterations. This enhances the interpreter's ability to handle more complex algorithms.

6. Functions

Functions are a fundamental part of Rust, and the interpreter supports their definition and invocation. It manages function parameters and return values, contributing to the overall functionality of the Rust interpreter.

7. If-Else Statements

Conditional statements, utilizing if-else constructs, are implemented. The interpreter evaluates conditions and executes the corresponding code block based on the evaluated results.

8. Print Statements

The interpreter includes the capability to output information to the console. Print statements aid in debugging and provide visibility into the program's execution by displaying relevant information.

Learning Outcomes

1. Language Design and Syntax

The project deepened our understanding of language design principles, especially within the context of Rust. We gained insights into how language features are structured and how syntax influences the interpretation process.

2. Parsing Techniques

Building a parser involved a practical exploration of parsing techniques and algorithms, with a specific focus on recursive descent parsing. We learned how to transform a stream of tokens into a hierarchical representation of the code.

3. Abstract Syntax Trees (AST)

Working with abstract syntax trees provided a tangible understanding of how compilers and interpreters represent and manipulate program structures. The AST served as an essential intermediary step in the interpretation process.

4. Interpreter Architecture

The project emphasized the significance of designing a modular and well-organized interpreter architecture. Separation of concerns among the tokenizer, parser, and interpreter components enhanced code maintainability and readability.

5. Error Handling

Implementing robust error handling mechanisms throughout the tokenization, parsing, and interpretation phases was a critical aspect. This experience enhanced our ability to manage and report errors effectively.

Conclusion

The Rust interpreter project in Java stands as a comprehensive exploration of language implementation and parsing techniques. It provides a solid foundation for understanding the inner workings of the Rust language and interpreting its code. Continued development could involve expanding language support, optimizing performance, and refining error handling mechanisms for a more robust and feature-rich interpreter.

Although we learned the language but the errors were too much to implement specific details of the language like let x: &str = “I am Veerain”;

Or let x: i64 = 1234353465;