Spring 2023

MacRuby: User Defined Macro Support for Ruby

Arushi Singh
San Jose State University

MacRuby: User Defined Macro Support for Ruby

A Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Arushi Singh

May 2023

The Designated Project Committee Approves the Project Titled

MacRuby: User Defined Macro Support for Ruby

by

Arushi Singh

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2023

ABSTRACT

MacRuby: User Defined Macro Support for Ruby

by Arushi Singh

Ruby does not have a way to create custom syntax outside what the language already offers. Macros allow custom syntax creation. They achieve this by code generation that transforms a small set of instructions into a larger set of instructions. This gives programmers the opportunity to extend the language based on their own custom needs.

Macros are a form of meta-programming that helps programmers in writing clean and concise code. MacRuby is a hygienic macro system. It works by parsing the Abstract Syntax Tree(AST) and replacing macro references with expanded Ruby code. MacRuby offers an intuitive way to declare macro and expand the source code based on the expansion rules.

We validated MacRuby by adding some features to the Ruby language that didn't exist before or were removed as the user base for the feature was small. We fulfilled this by creating a library using the simple and easy syntax of MacRuby, thus demonstrating the library's utility.

ACKNOWLEDGMENTS

I want to thank Dr. Thomas Austin for his guidance on this project.

TABLE OF CONTENTS

CHAPTER

1 Introduction 1

2 Background 4

2.1 Macros 4

2.1.1 Advantages of Macros 5

2.1.2 Macro Hygiene 7

2.2 C Preprocessor 9

2.3 Taint Analysis 11

2.3.1 Taint Analysis in Ruby 14

2.3.2 Tainted Objects in Ruby 15

2.3.3 Why was Taint Analysis Removed from Ruby 16

2.4 Lisp 16

2.5 Whitequark Ruby Parser 17

3 Design 19

3.1 Adding Macro Support 20

3.2 Hygiene in Ruby 21

4 Implementation 24

4.1 Macro Implementation 24

4.2 Hygiene Implementation 31

5 Experiment 34

5.1 Taint Tracking 34

5.2 Swap 37

5.3 Log 37

6 Conclusion and Future work 42

6.1 Future Work 42

6.1.1 Extending Macro Library 42

6.1.2 Improving Taint Tacking Macro 43

6.1.3 Modularize Ruby Macro 43

LIST OF REFERENCES 44

APPENDIX

CHAPTER 1

Introduction

Macro transforms a small set of instructions into a larger set of instructions. As of Version 3.2.2 the Ruby programming language does not have support for macros.

In maintaining larger systems a lot of time is spent by developers in reading and understanding existing source code. Therefore building mechanisms for enhancing the readability of the source code can be useful for any programming language. While programming, programmers often find certain patterns that repeat. Programmers should have the ability to express these patterns in a concise form. This can make the program look cleaner and more readable, which can in turn increase productivity. Macros give the ability to represent commonly used patterns in a concise format. When used wisely macros also help in making code look cleaner. Furthermore, they also allow developers to evolve the language.

```
1  a = 1
2  b = 2
3  puts(a, b) #Output: 1 2
4  swap(a, b) #swap macro call
5  puts(a, b) #Expected Output:2 1
```

Listing 1.1: Before Macro Expansion

```
1  a = 1
2  b = 2
3  puts(a, b) #Output: 1 2
4  (temp =a; a =b; b =temp) #Expansion
5  puts(a, b) #Expected Output:2 1
```

Listing 1.2: After Macro Expansion

Listings 1.1 and 1.2 demonstrate this benefit. This program creates a `swap` macro that takes in two variables and swaps the values between them. In Listing 1.1, line 4 calls the swap macro that is able to expand into the `swap` logic shown in line 4 of Listing 1.2. The macro code is expanded into the calling code context, which allow the expanded code to use the `"a"` and `"b"` variables declared in the calling code. If `swap` was a function instead of macro, the `swap` logic would not have worked as expected because the variables `"a"` and `"b"` would be replaced in the function's scope but not in the calling code's scope.

This project has added macro support to the Ruby language by modifying the lexer and parser of the language. Furthermore, it builds a library of some useful macros. The macro support allows programmers to create their own macros. Programmers can move the commonly used pattern into a macro definition and use a concise macro call to expand that pattern. The library does this expansion by finding macro calls in the code and replacing them with predefined patterns.

```
1 defmacro swap(num1, num2)
2     temp = num1
3     num1 = num2
4     num2 = temp
5 end
```

Listing 1.3: Swap Macro

Listing 1.3 shows how to define the `swap` macro used in Listings 1.1 and 1.2. The user needs to use the keyword `defmacro` as shown in line 1. Following the keyword is the macro name `swap`, next are the variables `"a"` and `"b"` that the user wants to pass to the macro, and finally there is the macro body. Macro expansion maps the local context variables to the macro arguments. In this case the variables `"a"` and `"b"` are mapped to `"num1"` and `"num2"`, respectively.

Many languages have built-in macro support like C [1] and Racket [2]. Adding built in macro support for the Ruby language is inspired by the Sweet.js [3] macro library for JavaScript and the Pantry [4] macro library for Python. Both of these libraries provide an easy and intuitive structure to define a macro. MacRuby has followed a similar strategy to give its users an intuitive structure for generating macros.

This report is divided in to 5 sections. Section 2 introduces macros, explains different types of macro, goes in depth of various aspects of macros, talks about different languages that use macros like the C preprocessor and Lisp. It also gives a background of taint tracking for which a library was created using MacRuby. Finally, it also gives a background of *Whitequark* [5], which is the Ruby parser we used to create the macro library. Section 3 gives a high level design of MacRuby and also discusses hygiene considerations. Section 4 goes into depth on the implementation details of our macro library. It discusses each and every aspect of how macro support was added to the Ruby language. Section 5 showcases how the macro support created in this project works in practice. Finally, the last section gives the summary of the project and also discusses future directions for the project.

CHAPTER 2

Background

To understand how macro support was added a sound understanding of what macros are and their advantages will be helpful.

2.1 Macros

Macros are a type of metaprogramming. With macros a computer program takes code as data input and the output is rewritten code. A macro is defined using the syntax or structure provided by the language. When the code is compiled the macro reference is replaced by actual code. This process is known as macro expansion. There are two major forms of macros, *text-substitution macros* and *syntactic macros*.

Text-substitution macros are used by languages like C and some assembly level languages. They have a preprocessor like the C preprocessor which works before tokenization. This is a simplistic macro system that helps in making code more readable. The preprocessor in figure 1 [6] substitutes any reference to the macro with the macro definition. After the preprocessing is done there is no reference to macros. All the references are replaced by user defined code that the compiler can understand. While this method works, it does not consider the surrounding scope. So the code might look fine and make sense on its own but when expanded in the main program, it might have issues. Bad hygiene is one such issue, which happens when variables in the macro and variables in the expanded source code collide.

The second form of macro, syntactic macros work on Abstract Syntax Trees(AST). They are associated with the Lisp language. In this method the macro body is a tree like structure that is woven into the main program's Abstract Syntax Tree. After macro expansion the AST does not have macro references in the tree. This is different than text substitution which takes the text of code and replaces macro references in the source code. The major advantage of using the AST is that it is aware of the

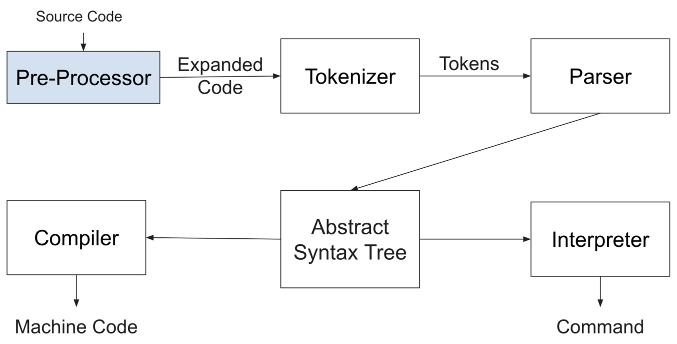

Figure 1: Text substitution macro expansion

syntactic structure of the code. This helps in alleviating invalid expansions and errors that are caused due to duplicate variable names.

2.1.1 Advantages of Macros

In certain languages like Lisp that don't have extensive syntax available, macros are extremely important to evolve the language. However, in Ruby the syntax and pre-defined macros available are very inclusive, which makes understanding the motivation for macros support in Ruby significant.

Macro definitions and macro calls often look like functions; they can even do similar things. Even though macros and functions look very similar, they are very different in how they are designed. The major difference is in context creation. A function creates a new context when it is called. The body of function looses access to the context of calling code. In contrast, macros retain the context of the calling code. The macro is expanded in the context of calling code [7]. Swap is a good example to help explain this process.

```
a = 1
b = 2
print(a, b) #Output: 1 2
a, b = swap(a, b) #Function call
print(a, b) #Expected Output:2 1
```

Listing 2.1: Swap with function

```
a = 1
b = 2
print(a, b) #Output: 1 2
swap(a, b) #Macro call
print(a, b) #Expected Output:2 1
```

Listing 2.2: Swap with macro

Ruby uses call-by-value to make calls to functions. In Listing 2.1 when swap is done using a function call the function has to return the swapped values so that the values can be assigned to the calling code variables. If we just pass variables "a" and "b" and swap them in the function, the variables "a" and "b" will remain unchanged in the calling code. This is because when the function is called a new context is created for the function. The function's new context loses access to original scope. Whereas, in Listing 2.2, swap written as a macro, the values do not have to be returned. We had seen earlier that `swap a b` is extended to `(temp = a; a = b; b = temp)` in the case of macros. The macro scope is extended in the original scope. Therefore, macro calls will never lose access to the original variables "a" and "b".

Often developers run into situations where existing language constructs are not enough. In these situation developers are forced to write complex code that deteriorates the readability and in-turn the understandability of the code. Macros alleviate this problem as they are helpful in generating Domain Specific Languages (DSLs). Further, they give developers the ability to define their own syntax which can be more condensed and readable. A feature request to expose the Abstract Syntax Tree (AST) node to give developers the ability to transform the code was made in Ruby issue number 11475 [8]. [8] explains that exposing AST nodes will help in creating macro libraries, which can prove to be very helpful to its users. Further, experimental semantic constructs can be introduced to users this way and based on their usefulness, they can be introduced natively.

2.1.2 Macro Hygiene

Macros are vulnerable to the *variable capture* problem that can cause extremely subtle bugs. Variable capture arises at the stage of macro expansion. On a high level, variable capture can be explained as an issue in which one variable name clashes with a variable in another context [7]. Variable capture can happen in two situations: *macro argument capture* and *free symbol capture.*

In argument capture a variable is passed to the macro that clashes with a variable that already exists in the macro definition. Listing 2.3 illustrates this issue; `temp` is a variable that was passed as a parameter to the swap macro and `temp` is also a variable used in the macro definition. Having two variables with the same name can cause an incorrect result.

In the case of free symbol capture, the context from which the macro is called has a variable that clashes with the variable defined in the macro definition. When macro expansion happens the two variables conflict leading to errors. In Listing 2.4 `swap` is again being used to explain this. The `temp` variable is used in the code from where the macro is called and `temp` is also declared in the macro body. This will cause the two variables to conflict and result in `temp` in the calling code being modified inadvertently. In calling code `temp` was set to 1 whereas in the macro definition `temp` is used to temporarily hold the value of `a`. After macro expansion the two `temp` variables conflict leading to the calling code `temp` value to be changed from 1 to 2.

```
temp = 1
a = 2
print(temp, a) # Output: 1 2
#Macro expansion from swap(temp,a
    )
temp = temp; temp = a; a = temp
#Swap will not swap values
#Causes argument capture error
```

Listing 2.3: Argument Capture Error

```
temp = 1
a = 2
b = 3
print(a, b) # Output: 1 2
#Expansion below
temp =a; a =b; b =temp;
print(temp) #Expected Output:1
#Free symbol error causes 2
```

Listing 2.4: Free Symbol Error

For a macro system to truly work, the user should not have to think about what variables they can use or the internal implementation of the macro when the user calls the macro. User should be able to use any variable in the calling code and it should not interfere with variables in the macro. This type of problem is called hygiene and a macro system should handle these possible issues.

There are multiple ways to make macros hygienic. One way is to modify all the variables in macros and append some text to make them unique. Even though this will work in most cases there is still a possibility that the modified variable still conflicts with some other variable. A better way to ensure that variables in the macro definition are unique would be to create a list of the variables used. If a macro variable conflicts with any of the variables in the list, the name of the macro variable should be changed to a unique name.

2.2 C Preprocessor

The C preprocessor(CPP) is a software tool that is an essential component of the C programming language. It processes source code before it is compiled. It performs a series of text substitutions on the source code, which helps in making the code easier to read. Depending on the the form of the C preprocessors used, the pre-processor step in Figure 1 can be broken down into four steps; character set, initial processing, tokenization and preprocessing language. In these steps the file is broken down into tokens and then it handles the macro expansion. Once the text substitution is done the code can be handed over for compilation. The pre-processor steps can be different if the form of C preprocessors changes.

There are two forms of the C preprocessors, traditional and standard. The standard CPP is the preprocessor that is commonly used today. The traditional CPP lacks some features that were introduced in the standard CPP. It mainly exists today for backward compatibility. The main difference between the two is in lexical analysis. The standard CPP breaks the input into tokens where as the traditional CPP treats input as stream of text.

The CPP has many pitfalls like misnesting, the operator precedence problem, swallowing semicolons, duplication of side effects, etc. Each one of these are explained in detail in [9]. One of the pitfalls that is commonly encountered by developers is operator precedence problem. This is a situation that arises when the macro arguments are used in body but they don's have parenthesis around it. This can be explained with the divide and roundup example from [9].

```
#define ceil_div(x, y) (x + y − 1) / y //Macro definition
a = ceil_div (b & c, sizeof (int)); //Macro call
a = (b & c + sizeof (int) − 1) / sizeof (int); //Macro expansion
```

Listing 2.5: Operator Precedence Problem

The operator precedence rule will make the above expression equivalent to a = (b & (c + **sizeof** (**int**) − 1)) / **sizeof** (**int**), which is not what is expected. The expectation is #**define** ceil_div(x, y) (((x) + (y) − 1) / (y)). To solve the operator precedence problem the macro definition can be modified so that there are parentheses when referencing any of the macro arguments. This will stop the operator precedence order from changing.

```
#define ceil_div(x, y) (((x) + (y) − 1) / (y)) //Macro definition with parenthesis
```

Listing 2.6: Solution to Operator Precedence Problem

The CPP has many pitfalls but still it is a very powerful tool. Ernst et al. [10] discuss the pitfalls, how the CPP is being used in practice, replacement of the CPP wherever possible and finally why the CPP can't be removed. The most important suggestion is to use existing language constructs and eliminate the use of CPP wherever possible. Ernst et al. suggest that simple changes like using `import` in place of `include` can bring down the CPP usage by 6.2%. Other changes like compilers that do better constant-folding can encourage developers to use more existing language constructs as 42% of macros are defined as constants. Standardising library function names and calling convention can eliminate many conditional statements as 37% of conditional statements are for operating system variance.

2.3 Taint Analysis

Taint analysis is a technique that is used by developers in finding potential vulnerabilities in their code. It works by looking into the flow of data and identifying any malicious code like a script or SQL injection in data and marking the data `tainted`. The major advantage of using taint analysis is that it helps developers add an additional layer of security by identifying potential sources of malicious data. Additionally, taint analysis aids developers in understanding how the program behaves.

Code vulnerabilities are easily introduced through web forms. Let's take an example. Suppose a malicious user is filling a simple web form and the user enters a harmful script in an input field. If the input field data is executed, it can put the entire system in danger. It would be very difficult to detect these issues as the malicious code is being entered into legitimate data. [11] Hansen explains this with example listed below.

```
$name = $_REQUEST['name'];
$greet = 'Hello '.name ;
echo $greet ;
```

Listing 2.7: Code vulnerability Example

Using taint analysis the developers can check all data entry and see if the user is entering anything that could be potentially dangerous for the system. In above case, if there is a script in data then the data can be marked `tainted`.

There are different approaches that have been developed to detect attacks. In [12] Newsome and Song explain most approaches to detect attacks land in one of the two categories *coarse-grained detectors* and *fine-grained detectors. Coarse-grained detectors* look for abnormal activities for detecting an attack such as scanning a port for malicious activity. They often give false positives and do not provide the details of vulnerabilities. Whereas *fine-grained detectors* try to detect any attack on program vulnerability with fewer false positives and also provides information on vulnerability and how it is exploited.

Taint analysis is categorized as a fine-grained detector. Taint analysis can be done in two ways, *static taint analysis* and *dynamic taint analysis*. In [11] Hansen explains what static taint analysis is and how it works in detail. The static taint tracking works at compile time to detect vulnerabilities in code. For performing the static taint analysis on Listing 2.7 the first step is to transform the code so that it is easier to understand how the data flows and how the control flows. A graph of the control flow is built. It is a directed graph called the *control flow graph*. In this graph the instructions are nodes and the edges depict control flow. It is evident that it is

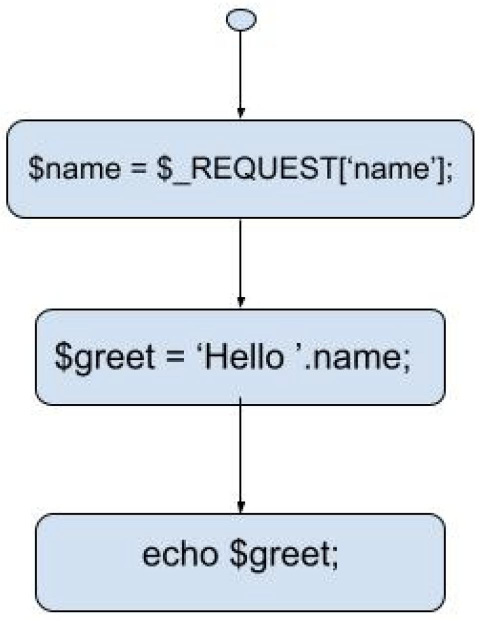

Figure 2: Control Flow Graph

very easy to visualize the code with the graph. Another advantage of using control flow graph is that it is very easy to know which instruction is executed before and which instruction is executed after a particular instruction. This is equivalent to the concept of predecessor and successor nodes.

In the next step, analysis is done to see which variables do not come from a reliable source and such variables are marked tainted. Next, for each instruction the incoming tainted variables are tracked and the outgoing tainted variables are marked `tainted`. Finally, the result of taint analysis is used to see if the code is vulnerable to injection attack. This is done by checking if the variables in output are referencing any tainted variable. So in this example the only output command is `echo $greet` hence greet is checked to be tainted, which is the case. So it can be said that the code snippet is vulnerable to injection attack.

In dynamic taint analysis data that originates from an untrusted source is marked `tainted`. As the program executes, how the tainted data propagates is tracked. An attack is indicated when a tainted data is used in a way that could harm the system. After attack is detected, dynamic taint analysis provides information about the vulnerability.

In [12] Newsome and Song discuss how the information about vulnerability can be used to automatically generate signatures to filter attacks in future. They call this tool *TaintCheck*. TaintCheck runs the code in its own emulation environment. This allows taintCheck to control the program's execution. Newsome and Song designed three components for taint analysis; *TaintSeed*, *TaintTracker* and *TaintAssert*. These three components answer three of the important questions for detecting an attack. First, which input should be marked tainted; second, how the tainted data propagates and third, when tainted data should raise alarm for attack. The situations that can be considered dangerous are jump addresses, system call addresses etc. The authors also suggest another component *exploit analyzer*. Taint analyzer is used for post-analysis to gather information about the attack such as which data caused the attack, semantic information about the payload. This information is used to automatically generate the signature for attack filtering.

2.3.1 Taint Analysis in Ruby

Taint analysis existed in Ruby until version 2.7. In Ruby taint analysis worked much like what is explained in the previous section with the exception of using `$SAFE` global variable. As [13] explains `$SAFE` helps developers in setting security levels for their code so that different source code can run under different security levels. Security levels can be between 0 to 4 with 0 being least secure to 4 being the highest security.

`$SAFE = 0`: Zero is the default security level in Ruby. This level has full permission with access to all resources on the system. This security level is not recommended for untrusted code or code that takes input from user.

`$SAFE = 1`: One has more restrictions and does not allow certain operations. For example, code cannot modify certain global variables or execute methods that can modify the system. This security level is recommended for untrusted code or code that takes input from users.

`$SAFE = 2`: This security level is similar to security level 1 with a few more restrictions. For example, code cannot create objects of certain classes or execute methods that can modify the system. This security level is recommended for untrusted code that is more likely to be malicious.

`$SAFE = 3`: This security level is similar to security level 2 with even more restrictions. For example, code cannot execute certain system commands, modify environment or access certain resources. This security level is recommended for untrusted code that is highly likely to be malicious.

`$SAFE = 4`: This is highest level of security in Ruby. Code cannot create object of certain classes, code can only execute methods that are explicitly allowed. This is recommended for running code in a sandbox environment, where the code is not trusted and should be executed with highest level of security.

2.3.2 Tainted Objects in Ruby

[13] discusses tainted objects in Ruby. [13] explains that any object that is derived from an external source is marked as tainted in Ruby. The example of this would be an input field that takes string input from a user. This object would automatically be marked as tainted. Any new object that that derives from a tainted object is marked tainted as well. This is shown in Listing 2.8.

```ruby
str1 = ENV["HOME"] #External input
str1.tainted? #true
str2 = "Welcome!"
str2 = str2 + str1 #Concat with tainted object
str2.tainted? #true
```

Listing 2.8: Tainted Object Example

In above example `str1` is a string object that is derived from an external source therefore `str1` is marked tainted. `str2` is another string object. When `str2` is concatenated with `str1`, `str2` is marked `tainted` as well. This is because after concatenation any malicious code that was entered in `str1` is copied to `str2` as well.

An object can be marked `tainted` by invoking `taint`. Taint can be removed from an object by calling `untaint`. As seen earlier, `tainted?` is used to check if an object is marked `tainted` or not. It returns `true` if the object is marked `tainted` and `false` otherwise.

2.3.3 Why was Taint Analysis Removed from Ruby

There were a number of reasons why taint analysis was removed from Ruby. First, taint analysis was not widely used by the Ruby community. The benefits of taint analysis were not well understood by the developers, which led to fewer developers taking advantage of the tool. Second, taint analysis was hard to maintain. `$SAFE` was complex a complex feature that required careful management or the code would break unexpectedly. Further, the feature was not well documented either which made it even more difficult for developers to use use this rather complex feature effectively. Finally, the third reason was that taint tracking was not an effective solution to tackle modern security threats as these threats evolve very quickly. Ruby wanted developers to use other techniques like input validation to tackle these issues.

2.4 Lisp

Lisp is a group of languages with Scheme being among the most popular derivatives. The name Lisp comes from "List preprocessor" because the source code is made of lists. One of the most important features of Lisp is the use of S-expressions. In Lisp, expressions and data are represented in the same format. For example, `(+ 1 2)` represents the addition of 1 and 2. The parenthesis indicate a list with first item being the name of the function. In this case, the name of function is the addition operator and 1 and 2 are the arguments to the function. In Lisp the lists can be nested such that elements can also be a list. From this the elements can be organized in a tree structure with function at parent and arguments as children. This list structure makes implementing macros in Lisp relatively straightforward.

Lisp uses syntactic macros. Syntactic macros use the grammar and construct of the language in which macro is defined [14]. As explained in Section 2.1 syntactic macros work on Abstract Syntax Trees(ASTs). Macro expansions are performed on the AST that preserves the lexical scope. This makes syntactic macro less error prone as AST transformation takes care of hygiene implementation. Let's take a simple example to understand how macro works in Lisp.

```
(defmacro (square x) '(* x x))
(defmacro (cube x) '(* x x x))
(define y 5)
(if (< y 3) (square y) (cube y))
```

Listing 2.9: Lisp Macro Example

In the above example, based on the value of y either the square or the cube will be performed on y. When evaluating a macro invocation, instead of evaluating the arguments first like a normal function, the evaluator takes the expression of the arguments as-is and passes it as a list to the called macro. The evaluator runs the modified expression returned by the macro. The quote symbol (') is used to specify expressions that should not be evaluated and thus it evaluates to itself [15]. So in the above case, if $y < 3$, the macro returns the expression (* x x) otherwise it returns (* x x x).

2.5 Whitequark Ruby Parser

In order to add support for macros in the Ruby language, we need to modify its grammar and add post processing logic to expand the macro definition. As Ruby's parser is not open source to modify, we will be using the Whitequark parser [5], which is a production-ready Ruby parser written in pure Ruby. It supports parsing a chunk of Ruby code and generates its S-expression as seen in the example below from [5].

```
p Parser::CurrentRuby.parse("2+2")
# (send (int 2) :+ (int 2))
```

Listing 2.10: Whitequark parser Example

Gem library [16] can be used to traverse the abstract syntax tree (AST) generated by this parser and modify it. Each node in the AST has a type and list of children. Library's process method accepts a node and dispatches it to the handler corresponding to its type, and returns an updated variant of the node. A handler class is used to define methods to process each nonterminal node and recursively process all its children nodes. Below is a example from [16] to update node type from add to multiply for a simple calculator language.

```
class ArithmeticsCalculator
  include AST::Processor::Mixin
    ...
    def on_add(node)
        left_expr, right_expr = *node
        node.updated(:multiply, [
          process(left_expr),
          process(right_expr)
        ])
    end
end
```

Listing 2.11: AST Modification Example

CHAPTER 3

Design

To understand how the macro support was added to the Ruby parser we will use the swap example. To create a swap function(without using macros) the arguments to the function have to be passed by reference or the function should return the result of swap as shown in Listing 2.1. If this is not done then the variables will swap in function context but not in the calling source code context. However, with macros there is no need to pass arguments by reference or return the result as the macro adds the swap code in the context of calling source code. Below the swap code snippet gives an example of what the added macro support looks like for Ruby.

```
defmacro swap(num1, num2)
    temp = num1
    num1 = num2
    num2 = temp
end
a = 4
b = 2
swap(a,b) #Swap macro call
```

Listing 3.1: Macro Definition

```
a = 4
b = 2
#After macro expansion
(temp = a; a = b; b = temp)
```

Listing 3.2: Macro Expansion

Listing 3.1 defines the `swap` macro. `defmacro` is the keyword that was added to define a new macro. The `swap` macro code looks a lot like a function but how they work is different. In macro, the line of code that calls macro that is `swap(a,b)` in this case is replaced by the body of macro. The macro body is added in the same context as the calling source code, which is why there is no need to pass arguments by reference. When the code runs after macro expansion there is no reference macro call and the macro definitions are removed as well. The resulting source code can be compiled by any Ruby compiler.

3.1 Adding Macro Support

Next we will discuss on a high level how macro support is added to the Whitequark parser and how macro expansion is performed. The first step to add macro support is to add the ability to define a macro. For this the lexer is modified to add `defmacro` as a token. Next, the token is added to the grammar. After adding the token the structure of macro definition is added to grammar as well. The structure of macro definition looks similar to the function definition structure with the exception of text used to define a macro versus a function. `defmacro` is used for macro whereas for function the text is `def`. Listings 3.3 and 3.4 compares the syntax of a macro and a function.

```
defmacro pretty_print(n1, n2)
    puts("Num1:" + n1.to_s)
    puts("Num2:" + n2.to_s)
end
```

Listing 3.3: Macro Definition

```
def pretty_print(num1, num2)
    puts("Num1:" + n1.to_s)
    puts("Num2:" + n2.to_s)
end
```

Listing 3.4: Function Definition

When the Ruby parser executes it generates the Abstract Syntax Tree(AST). For the parser to be able to create a node for the macro a new macro definition class is created. The macro class initializes a new node for the macro in the AST. Once the AST is successfully created we move to the macro expansion step. For macro expansion we traverse through the AST [16] to keep track of all the macros that are defined in the AST. After this, any time a macro is called, the macro call is replaced with the macro body. An important step in this is argument name substitution. To understand this we will again use the swap example. In Listing 3.1 the `swap` macro definition has argument names `num1` and `num2`. When `swap` is called with variable names `a` and `b`, the variables `num1` and `num2` have to be replaced with `a` and `b`. Only then the macro body will perform as expected when calling code is replaced with macro body.

Once macro expansion is done there are no reference to macro in the AST. All the macro references are replaced with macro body. The final step is to generate Ruby code from the AST. Unparser library [17] is used to generate Ruby code from AST. All that is needed to be done is call the unparse method in the unparser library and pass the AST. The result is Ruby code.

3.2 Hygiene in Ruby

An important aspect of adding macro support is handling hygiene. If a macro is not hygienic it can be difficult for the end user to use it effectively as it will add subtle bugs that are very hard to debug. As discussed in Section 2.1.2, *variable capture* is a hygiene issue that arises when variable names clash. The two scenarios that can cause variable capture are *macro argument capture* and *free symbol capture*. An argument capture error is caused when variables in the argument clash with variables in the macro body as seen in Listing 2.3. Whereas, a free symbol error is caused when the variable in the macro body clashes with variables in the calling source code as seen in Listing 2.4.

Both scenarios have been handled in the macro support for Ruby. This has been handled at the time of macro expansion. When the AST is traversed we track the variables that are in the calling source code. At the time of replacing macro calls with macro bodies any clashing variables in the macro body are replaced with new variable names. The new variable names created are checked against tracked variables to verify the new variable name does not clash as well. This check is done till a unique variable name is found to replace the clashing variable name in macro body. Once found the variable name is replaced and the macro body is added in place of macro call. Examples below show how macro argument capture and free symbol capture have been handled in the macro.

```
defmacro swap(num1, num2)
    temp = num1
    num1 = num2
    num2 = temp
end
temp = 0
a = 4
b = 2
swap(a,b) #Macro call
puts(temp) #Expected 0
```

Listing 3.5: Possible Free Symbol Error

```
temp = 0
a = 4
b = 2
#temp replaced with temp1
#to prevent variable clash
(temp1 = a; a = b; b = temp1)
puts(temp)
#Printes expected value 0
```

Listing 3.6: Free Symbol Error Handled

In Listing 3.5 `temp` variable is clashing as it is present in both the calling code and the macro body. This could possibly cause a free symbol error after macro expansion leading to the value of `temp` inadvertently being changed to 4(value of a). However, this scenario is handled at the time of macro expansion and the `temp` variable name in the macro body was replaced with `temp1` at the time of macro expansion preventing the clash.

Listing 3.7 is a possible scenario for argument capture error as the `temp` argument in the macro call clash with the `temp` variable in the macro body. If this scenario is not handled, the first line of the macro body will be `temp = temp`, which is not what is expected. We are handling this scenario at the time of macro expansion as well. Therefore, the `temp` variable in the macro body is changed to `temp1` as shown in Listing 3.8.

```
defmacro swap(num1, num2)
    temp = num1
    num1 = num2
    num2 = temp
end
temp = 0
a = 4
swap(temp,a)
```

Listing 3.7: Possible Free Symbol Error

```
temp = 0
a = 4
#temp in body replaced with temp1
#to prevent variable clash
(temp1 = temp; temp = a; a =
    temp1)
```

Listing 3.8: Free Symbol Error Handled

CHAPTER 4

Implementation

4.1 Macro Implementation

This section discusses the details of adding macro support to Ruby. We will go over each step to better understand what was done with code snippets. The first step in adding macro support is creating a token that can be used to define a new macro. As mentioned earlier, `defmacro` is the new token that was created for defining a macro. To add the token, the lexer was modified and `defmacro` was added to the list of keywords. Once the `defmacro` was added in the lexer, the keyword name was used to add `defmacro` to the list of tokens in parser. After the keyword is added to the parser, whenever the Ruby parser runs, it is able to recognize `defmacro` as one of the tokens and successfully parse it.

The second step after adding the token is to define the structure for the macro definition. This primarily defines the structure for a new macro. This is added to the grammar so that when the parser runs it is able to parse through the defined grammar and generate the AST. The grammar for defining a new macro looks similar to the structure for defining a new function with the exception that the keyword used for defining a function is `def` whereas for macro it is `defmacro`.

The grammar for macro definition is in Listing 4.2. The grammar looks for any code that matches the pattern

```
defmacro <macro name><argument list>
    <code body>
End
```

Listing 4.1: Macro Definition Pattern

Next, the defmacro_method is called, which generates the AST for the user-defined macro. AST generation is discussed in detail in the next section.

In the grammar `k_defmacro` is the keyword `defmacro` that is followed by `fname`. `fname` is the name of the macro. For example in listing 3.1, `fname` is `swap`. `f_arglist` is the argument list for the macro that is `num1, num2` in listing 3.1. The argument list is followed by `bodystmt`, which is the macro body. At the end `kEND` notes the `end` token.

```
k_defmacro fname {
        local_push
        result = context.dup
        @context.in_def = true
    }
    f_arglist bodystmt kEND {
        result = @builder.defmacro_method(val[0], val[1],
                    val[3], val[4], val[5])
    }
```

Listing 4.2: Macro Definition Grammar

The third step is generating the AST. As the parser runs, the n method is called and the AST is generated. The `n` method takes the type of the node, the children, and the source map, and it returns the AST node. This new node gets added to the AST.

```
def n(type, children, source_map)
  AST::Node.new(type, children, :location => source_map)
end
```

Listing 4.3: n Method Definition

For the `swap` macro from listing 3.1 the AST node looks like listing 4.4. In the output node the node type is `defmacro` with the name set to `swap`. The type and name are followed by the argument list that is shown by `args` in line 2. The argument list has `num1` and `num2`. The source code body follows the argument list.

```
 1  (defmacro :swap
 2    (args
 3      (arg :num1)
 4      (arg :num2))
 5    (begin
 6      (lvasgn :temp
 7        (lvar :num1))
 8      (lvasgn :num1
 9        (lvar :num2))
10      (lvasgn :num2
11        (lvar :temp))))
```

Listing 4.4: AST Node for Swap Macro

All the calls to macros and functions are of type `send`. For example, the call to the `swap` macro looks like `swap(a,b)`. The generated AST node is then `(send nil :swap (lvar :a) (lvar :b))`. In the node, `send` is the type and `a` and `b` are the arguments.

Once the AST is successfully generated, we move to the fourth step of macro expansion. In macro expansion the macro calls are replaced with the macro body. The AST is traversed to find all the macros that are defined. In the Whitequark parser at the time of processing the "on" method of node type is called. The "on" method of type `defmacro` in listing 4.4 in `on_defmacro`. At the time of processing the AST for macro expansion when on_defmacro is called, the name of all the macros and the corresponding macro body is tracked for replacing macro calls with macro body at later stage. To keep track of all the macros a map is created with macro name and their corresponding macro AST node. In case of swap the map entry will have swap as the name and the value will be the AST node that is shown in listing 4.4. Listing 4.5 shows the on_defmacro method that is called for all defmacro type nodes. @custom_macros_map in line 3 is the map that keeps track of all the macros.

```
1  def on_defmacro(node)
2      name, args_node, body_node = *node
3      @custom_macros_map[name]= node
4      ...
5  end
```

Listing 4.5: Method for Macro Tracking

Once the list of all the the macros is ready, the next step is to move on to macro expansion. As discussed earlier, macro calls are of type send. For macro expansion the on_send method associated with send is modified. The on_send method is called for each macro and function call. When the on_send method is called for a macro or function, the @custom_macros_map that has the names of all the macro is checked. This check is to verify if on_send was called previously for a macro or function. If it was called for a macro then macro expansion is performed. Listing 4.6 shows the on_send code.

```
def on_send(node)
    receiver_node, method_name, *arg_nodes = *node
    if @custom_macros_map.has_key?(method_name)
        macro_name, macro_args_node, macro_body_node = *
            @custom_macros_map[method_name]
        @marco_arg_to_send_arg = Hash.new
        if macro_args_node.respond_to?(:children)
            index = 0
            macro_args_node.children.each do |child|
                macro_arg_name = *child
                arg_name = *arg_nodes.at(index)
                @marco_arg_to_send_arg[macro_arg_name] = arg_name
                index = index + 1
            end
        end
    @processing_macro = true
    update_macro_body = process(macro_body_node)
    @processing_macro = false
    @marco_arg_to_send_arg = Hash.new
    return update_macro_body
end
...
```

Listing 4.6: Method for Processing Calls

Line 3 checks if the `on_send` call was made for a macro or a function by checking if `@custom_macros_map` list has the macro name. For this reason a macro and a function cannot have the same name. If it is a macro call then macro expansion is done. One of the critical part of macro expansion is mapping the calling arguments to the macro definition arguments. For example, the `swap` definition looks like `defmacro swap(num1,num2)` and call to `swap` might be `swap(a,b)`. Therefore, `num1` and `num2` need to get mapped to `a` and `b`. Line 8 loops through each each argument in the macro definition and maps it to the macro call argument. The mapping is created in `@marco_arg_to_send_arg` list where the key is the macro definition argument name and the value is the macro call argument name. Once the mapping is created, we have to change the variables in the macro body.

The call to `process` is where the change of variable names happens in the macro body. The `process` method calls `on_var` and `on_vasgn` on the macro body. `on_var` is responsible for resolving a variable and `on_vasgn` is responsible for handling variable assignments. Both these methods use the mapping list `@marco_arg_to_send_arg` created in `on_send` to map the variable names. Listing 4.7 and 4.8 show how the variable name replacement happens. Note that these code snippets are simplified versions that do not take into account the hygiene to eliminate variable clashes. The detailed implementation with hygiene is explained in Section 4.2. Just like `on_send`, `on_var` and `on_vasgn` are called by functions as well, therefore it is again important to check if a macro or a function is being processed. The check is shown in Listing 4.7 on line 3 and Listing 4.8 on line 3 where `@processing_macro` is checked to be `true` for macro processing. The `@processing_macro` is set to `true` in `on_send` on line 16 before `process` is called. Once `process` returns with with updated macro body, `@processing_macro` is set to `false`.

```
def on_var(node)
    name = *node
    if !name.nil? && @processing_macro
        && @marco_arg_to_send_arg.has_key?(name)
      node.updated(nil, @marco_arg_to_send_arg[name])
    else
      node
    ...
```

Listing 4.7: Method for Variable Processing

```
def on_vasgn(node)
    name, value_node = *node
    if !value_node.nil? && @processing_macro
        && @marco_arg_to_send_arg.has_key?([name])
      updated_name = @marco_arg_to_send_arg[[name]].first
      node.updated(nil, [updated_name, process(value_node)])
    elsif !value_node.nil?
        node.updated(nil, [name, process(value_node)])
    ...
```

Listing 4.8: Method for Assignment Processing

It can be seen on both the Listings 4.7 and 4.8 above that that they get the updated name of the variable from the map `@marco_arg_to_send_arg`. `node.updated` updates the variable names in the AST. Once the AST node for the macro is updated, the `on_send` macro node can be replaced with the macro body node. The macro definition can be removed as well. Once all four steps are done, the AST will not have any reference to the macro. Any Ruby compiler will be able to process this AST. The final step is generating Ruby code from the AST. This can easily be done by using the unparser library introduced in Section 3.1. As mentioned earlier the macro is not hygienic yet, so the variables may still clash. The next section discusses how hygiene is implemented.

4.2 Hygiene Implementation

It is very important for a macro system to be hygienic. If a macro system is not hygienic it can introduce subtle bugs that are very hard to debug. Furthermore, a macro user should not have to worry about how a macro library is implemented. If the macro system is not hygienic it may force the user to understand the implementation details. There are multiple ways for implementing hygiene in a macro system, Pantry [4] and Lisp [7] both use the `gen_sym()`. Whereas, Flatt [18] uses scope to implement hygiene. Similar to Pantry, MacRuby uses a variable name replacement method to implement hygiene.

The variable name replacement technique can be implemented in two ways. The first technique is to automatically replace the clashing variable name in the macro definition. The second technique is to replace the variable at the time of macro expansion when the macro is called. MacRuby uses the second approach. This approach keeps a list of all the variables used in the program and at the time of macro expansion it renames that variable if any of the variable in macro body clashes with the list.

For variable replacement, it is important to check that the replaced variable does not clash as well. For this reason when replacing variables we have to verify that the new variable name is unique. If it is not then we keep changing the variable name until a unique name is found. To create a unique name we suffix the variable name with an integer value. The integer value is incremented by 1 until a unique variable name is found. Listing 4.9 and 4.10 shows an example of how hygiene implementation changes variable names. In listing 4.9 since temp and temp1 are already present in the source code(line 6 and 7), the variable name `temp` in the macro definition is replaced by `temp2` for the first macro expansion in line 10. After the macro is expanded for line 10, `temp2` becomes a variable already in use. Therefore, for the second macro expansion in line 13 `temp` is replaced with `temp3` in macro body.

```
 1  defmacro swap(num1, num2)
 2      temp = num1
 3      num1 = num2
 4      num2 = temp
 5  end
 6  temp = 0
 7  temp1 = 1
 8  a = 4
 9  b = 2
10  swap(a,b)
11  c = 6
12  d = 8
13  swap(c,d)
```

Listing 4.9: Before Swap Macro Expansion

```
1  temp = 0
2  temp1 = 1
3  a = 4
4  b = 2
5  (temp2 = a; a = b; b = temp2)
6  c = 6
7  d = 8
8  (temp3 = c; c = d; d = temp3)
```

Listing 4.10: After Swap Macro Expansion

For implementing the unique variable name generation the `on_var` and `on_vasgn` is modified. A part of the `on_var` code is shown below in listing 4.11 to explain this. `on_vasgn` follows similar logic. The loop between line 7-10 checks `@used_variable_names` to generate a unique variable name. Once it finds a unique variable name it breaks out of the loop and the variable is updated in the AST. Once the AST is updated the variable is added to the used variables list shown in line 11.

```
 1  def on_var(node)
 2      name = *node
 3      ...
 4      elsif @processing_macro && @used_variable_names.member?(name[0])
 5          suffix = 1
 6          updated_name = name[0].to_s + suffix.to_s;
 7          while @used_variable_names.member?(updated_name.to_sym)
 8            suffix = suffix + 1
 9            updated_name = name[0].to_s + suffix.to_s;
10          end
11          @used_variable_names_in_macro.add(updated_name.to_sym)
12          node.updated(nil, [updated_name])
13      ...
14    end
```

Listing 4.11: Method for Assignment Processing

CHAPTER 5

Experiment

This section gives examples of how macros are implemented using MacRuby. Further, it also shows how macro can be beneficial for the Ruby language.

5.1 Taint Tracking

Taint tracking in Ruby is used to identify and track potentially dangerous data that can harm the system. It helps in preventing SQL injection and cross-site scripting attacks. Ruby does this by marking data that comes from external sources. The `taint` method marks an object as potentially insecure, while the `untaint` method removes the tainted mark. Data can be checked to be tainted using `tainted?`.

The critical aspect of tracking tainted data is to understand how it impacts other safe data. If untainted data come in contact with tainted data, the untainted data is marked tainted as well. Let us take an example of a string `str1` that is tainted and another string `str2` that is safe. If `str2` is concatenated with `str1`, then the result is marked tainted as well.

Taint tracking was removed by Ruby in version 2.7. Why the taint tracking was removed from Ruby has been explained in section 2.3.3. Even though taint tracking was removed, there was still a minority who use taint tracking. We have added taint tracking to Ruby by using MacRuby to demonstrate how macro can be helpful for users who want something specific for their system. Listing 5.1 shows the macro definition for adding taint tracking for string. The code snippet is simplified for better readability. A new attribute `@tainted` is added to string class to track if a string is tainted or not. New macros are created to perform the tasks related to taint tracking like `taint`, `untaint`, `sanatize`. `tainted?` was added to the string class so that `concat` function can get the value of `@tainted`. The `concat` function is overridden to modify `@tainted` when a tainted string is concatenated with another string.

```
defmacro taint(str)
    str = String.new(str, true)
end
defmacro untaint(str)
    str = String.new(str, false)
end
module MyString
    attr_accessor :tainted
    def initialize(value, attribute=nil)
        @tainted = attribute
        super(value)
    end
    def tainted?
        @tainted || false
    end
    def concat(value)
        result = "#{self} #{value}"
        if !self.tainted? && value.tainted?
          return String.new(result , true)
        end
        result
    end
end
String.prepend(MyString)
```

Listing 5.1: Taint Tracking Macro for String

The taint tracking macro can be used by calling taint method on the string that has to be marked tainted. For example `taint(str1)` will mark `str1` tainted. To get the taint status of a string `tainted?` can be used. For example `str1.tainted?`. Now, if we perform concatenation `str2 = str2 + str1`, then `str2` will be marked tainted as well. Finally, to mark a tainted string, `str2`, safe, we can call `untaint(str2)`. A separate macro is created for `sanitize`. `sanitize` is a simple implementation to remove any reserve words from the string to prevent SQL injection or cross-site scripting.

Some advantages come from using macros instead of functions. A significant advantage is in the case of a frozen string. Ruby allows strings to be marked frozen. A frozen string cannot be modified once it has been created. Modification of a string requires a new string to be created. Therefore, for setting the `@tainted` attribute, a new string is created, as shown in Listing 5.1, lines 12 and 15. With macro, at the time of macro expansion, str will be replaced with the string name that had to be marked tainted. Listing 5.2 and 5.3 explain this with an example. `my_str` is a frozen string that had to be marked tainted. All that user has to do is call taint macro as shown in Listing 5.2 line 3. The macro creates a new string and reassigns the name to the original string, as shown in Listing 5.3, line 3. If taint was a function instead of a macro, the user must write `my_str = taint(my_str)`. Reassigning to `my_str` for marking `my_str` tainted is not intuitive.

```
1  my_str = 'This is my string'
2  my_str.freeze
3  taint(my_str)
```

Listing 5.2: Before Macro Expansion

```
1  my_str = "This is my string"
2  my_str.freeze
3  my_str = String.new(my_str, true)
```

Listing 5.3: After Macro Expansion

Another advantage of using a macro is performance gain. Function calls are more expensive than inline code. In the taint tracking case, we compared taint tracking performance with macro and taint tracking with function in a benchmark test. Taint tracking with macro is 1.10 times faster than taint tracking with function.

5.2 Swap

So far in the discussion, we have seen `swap` macro at many places. `swap` macro has the advantage over `swap` function. For a swap function to switch data between two variables, the function has to return switched values. If the values are not returned, the values will switch in the function context but not the main context. However, this is different for macros because the macro body is added in the expanded source code. `swap` macro and corresponding macro expansion can be seen in listings 3.1 and 3.2.

5.3 Log

Logging is an essential part of an application. There are several types of logging, like information logging, that log common scenarios like a query running successfully. There are warnings where maybe the query ran, but the output is not what is expected, and finally, there are error logs that log exceptions. One such logging library is Google's glog [19].

There are several factors to consider while creating a logging library. First and most important is how efficient the library is. Logs add entries every millisecond in a system. If the logging library is inefficient, then a good amount of efficiency can be lost just for logging, which is not even the actual business logic. An inefficient logging library can slow down a system which will make the library unusable for large systems. Even if a system can use such a library, it will require more CPU, which means the company has to spend more money running the system.

The second important factor is how easy it is to use the logging library. Based on the environment logs are being run on, they are supposed to behave differently. For example, suppose a system is being run in a production environment. In that case, we might want errors and warnings to be logged, as logging everything will cause a loss in efficiency, which is very important in a production environment. However, when running a system in a developer environment, a developer might want to log informational logs as well for debugging purposes. Therefore the logging library should have an easy way to switch between what kind of information should get logged.

Finally, the library code should be clean. The code to use the library should be simple and readable as it goes in with actual business logic. If the logging code is not simple and clean, combined with the source code for business logic, it will make the system's source code unreadable and challenging to understand.

Using MacRuby, we created a logging library that incorporates all the necessary factors mentioned above for a successful logging library. It is efficient, easy to use, and readable. Listing 5.4 shows the source code of the library. Listing 5.5 shows an example of how to use the library. We will explain how the library created is efficient, easy to use, and readable one at a time.

```
defmacro log_info(value)
    if $mode == 0
        puts("info:" + value)
    end
end
defmacro log_warning(value)
    if $mode <= 1
        puts("warning:" + value)
    end
end
defmacro log_error(value)
    if $mode <= 2
        puts("error:" + value)
    end
end
defmacro log_if(value, condition)
    if $mode == 0 && condition
        puts("info:" + value)
    end
end
```

Listing 5.4: Logging Library Macro

The library allows the user to set minimum mode. Zero is for info, one is for warning, and two is for error. `$mode` is used to set the minimum mode. The library prints everything, including and above the number set in `$mode`. If `$mode` is set to

0, the logger will print information, warnings, and errors. If the `$mode` is 1, it will print warnings and errors. Finally, if the `$mode` is set to 2, it will only print errors. For information logs, there is `log_info` macro; for warnings, we have `log_warning`; and for errors, there is `log_error`. There is a fourth macro `log_if` that logs, if the condition passed is met.

```
$mode = 1
employees = [
    { id: 1, name: "John Doe" },
    { id: 2, name: "Jane Smith" }]
id_to_check = 4
begin
    if employees.any? { |employee| employee[:id] == id_to_check }
        info_message = "Employee found"
        log_info(info_message)
    elsif id_to_check.class.to_s != "Integer"
        raise
    else
        warning_message = "Employee not found"
        log_warning(warning_message)
    end
rescue
    error_message = "error occured"
    log_error(error_message)
end
```

Listing 5.5: Using Log Library

To understand how this library is efficient, easy to use, and clean, we will use listing 5.5. The significant efficiency comes from using macros instead of functions to log items. Function calls are more expensive than a macro. If a function was used instead of macro, there had to be a function call for every log, whereas macro replaces code in the calling context, making it behave like inline code. We compared the performance of the logger we created with functions and macros. The logger created with macros is 1.24 times faster than those created with functions. Table 1 shows the performance results from the benchmark test.

Table 1: Function vs Macros Logger Performance

	Function	**Macros**
Iterations/sec	26741093	33146669

The ease of use comes from using the `$mode` variable. For a developer to change what information they want to get logged, they only have to change the value in `$mode`, as discussed earlier. Finally, the source code is readable. All we need to do is call the macro like a function that will expand into an if condition for different modes. The developer will not see the expanded code, which keeps the code clean.

CHAPTER 6

Conclusion and Future work

Adding macro support to a language could be beneficial. Languages like C and Lisp with macro support have evolved a lot because of it. A language that does not have macro support has extensive syntax available, which makes adding macro support a lower priority.

Ruby has 1164 open feature requests [20] for the language. Ruby community has different kinds of suggestions to improve the language. There are requests for adding new features to the language. Some requests suggest how an existing feature can be improved from how it is implemented. These requests help in evolving the Ruby language. However, not all these requests are implemented. Many requests are delayed or rejected. There are also cases where an old feature is removed as there were minority users of that feature, but a small community still benefited from it. So while the Ruby developers wait for these features to be incorporated into the language, macros can be a great solution. Developers do not have to wait for their feature request to be accepted, built, and then get for the next Ruby version. Developers can write a macro and get the feature they want. Further, macros can be a great way to test a new idea, find design flaws, and find if a new feature has a good user base before a feature is implemented into the language. Finally, macros can perform better than functions in cases where a small set of instructions are called frequently.

6.1 Future Work

There are improvements that can be made to MacRuby. Some of them are listed below.

6.1.1 Extending Macro Library

A significant improvement for MacRuby would be to extend the macro library created. Finding different formats for creating different types of macros. The current macro format incorporates macros that look like function calls. If the format is expanded, then it opens up the opportunity for conditional codes like `#if` available in the C preprocessor

6.1.2 Improving Taint Tacking Macro

Another improvement could be in the taint tracking library. The taint tracking library that is implemented requires developers to mark the initial data tainted. For example, all data that comes from external sources can be marked tainted. Once data is marked tainted, it is tracked through the source code to find which other data it impacts and marks that data tainted. Extending the taint tracking library to auto-detect exploits and use that exploit to generate signatures to prevent future attacks, as suggested by Newsome and Song [12], could be beneficial. This will help taint tracking to keep evolving as threats to the systems keep evolving.

6.1.3 Modularize Ruby Macro

Another area for future work is moving the Ruby macro to its module. Right now, the macro support is added only for the Whitequark parser for Ruby. If the macro is converted into its own module, then it can possibly work with different Ruby parsers.

LIST OF REFERENCES

[1] “The c preprocessor.” [Online]. Available: https://gcc.gnu.org/onlinedocs/cpp/

[2] “Macros.” [Online]. Available: https://docs.racket-lang.org/guide/macros.html

[3] T. Disneya, N. Faubion, D. Herman, and C. Flanagan, “Sweeten your javascript: Hygienic macros for es5,” *ACM SIGPLAN Notices*, vol. 5, no. 2, pp. 35--44, 2014.

[4] D. Pang, “Pantry: A macro library for python,” *Master’s Projects. 657*, 2018.

[5] “Parser.” [Online]. Available: https://whitequark.github.io/parser/

[6] T. Austin, “A review of compilers,” p. 10. [Online]. Available: https://www.cs.sjsu.edu/~austin/cs252-spring18/slides/CS252-Day03_BigStepOperationalSemantics.pdf

[7] P. Graham, *On Lisp: Advanced Techniques for Common Lisp.* Prentice Hall, 1994.

[8] “Ast transforms.” [Online]. Available: https://bugs.ruby-lang.org/issues/11475

[9] “Macro pitfalls.” [Online]. Available: https://gcc.gnu.org/onlinedocs/cpp/Macro-Pitfalls.html

[10] M. D. Ernst, G. J. Badros, and D. Notkin, “An empirical analysis of c preprocessor use.” *IEEE Transactions on Software Engineering*, vol. 28, no. 12, 2002.

[11] R. R. Hansen, “Static taint analysis: A brief introduction.” [Online]. Available: https://cybertraining.dk/CodeAnalysis/#/lessons/X74l4Ti2nDufZIF4E9SGAHPDDp0SO62S

[12] J. Newsome and D. X. Song, “Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software,” *Network and Distributed System Security Symposium*, 2005.

[13] “Locking ruby in the safe.” [Online]. Available: https://ruby-doc.com/docs/ProgrammingRuby/html/taint.html

[14] T. Ball, *The Lost Chapter: A Macro System For Monkey.* [Online]. Available: https://interpreterbook.com/lost/

[15] “Macro elegance: the magical simplicity of lisp macros.” [Online]. Available: https://dotink.co/posts/macros/

[16] ‘‘Abstract syntax tree.’’ [Online]. Available: https://whitequark.github.io/ast/

[17] ‘‘Unparser.’’ [Online]. Available: https://github.com/mbj/unparser

[18] M. Flatt, ‘‘Binding as sets of scopes.’’ *ACM SIGPLAN Notices*, vol. 51, no. 1, pp. 705--717, 2016.

[19] ‘‘Google logging library.’’ [Online]. Available: https://github.com/google/glog#user-guide

[20] ‘‘Issue tracking.’’ [Online]. Available: https://bugs.ruby-lang.org/projects/ruby-master