6CCS3PRJ Final Year

RIMPiler

Towards a Comprehensive Framework for Reversible Imperative Languages

Final Project Report

Author: Aidan Dakhama

Supervisor: Professor Maribel Fernandez

Student ID: 21034945

April 12, 2024

Abstract

Reversible computing[43] is a key area of research, providing wide spanning benefits, from low-power computing[33], to greater debugging features[56] within programming languages. Several languages have been proposed, such as Janus[61] which ensures reversibility by strict rules on how the program can be written. Unlike Janus, RIMP provides a high-level interface to write programs, which can then be transformed into a reversible program. This thesis presents a framework for the development of reversible languages, focusing primarily on the imperative language RIMP. Building upon prior work by Fernández et al.[16], we introduce a new syntax for RIMP and extend its capabilities to support types. We provide implementations in Rust, including an evaluator, an abstract machine, and a compiler targeting the Java Virtual Machine (JVM). Our implementation also allows for the easy expansion of the features of RIMP, allowing the language to grow, and become more expressive in the future.

Originality Avowal

Acknowledgements

I extend my sincere gratitude to several individuals who have been pivotal in the completion of this thesis.

Foremost, I am deeply thankful to my supervisor, Professor Maribel Fernández, for her consistent support and invaluable feedback throughout this project. Her guidance and expertise have been instrumental in shaping my work and enhancing its quality.

I also want to acknowledge the colleagues I've collaborated with during this journey. Their encouragement and support have been a source of motivation and inspiration throughout.

Thank you to all who have contributed to this endeavour. Your support and guidance have been greatly appreciated.

Contents

Chapter 1

Introduction

In this thesis, we aim to provide a framework for the development of reversible languages, focussing on the imperative language RIMP. The overarching objective of this project is to establish a comprehensive system for developers using RIMP, as well as for future developers of RIMP; facilitating seamless modification, and expansion to the language.

The system is envisioned as a cohesive framework of libraries as well as a command-line interface (CLI) which utilises these libraries. This provides developers with tools for navigating the intricacies of development, from conception, to debugging, to release.

We build on the previous work done by Fernández et al.[16], by defining a new syntax for the language, similar to that of familiar languages such as C. Additionally, we extend the language to support multiple types, including integers and floating-point numbers, expanding the expressiveness of the language. In order to enable the use of multiple types, we have had to define a formal type system for RIMP, and alter its structural semantics and its abstract machine to facilitate this. We also provide an implementation for this in Rust.

Central to this thesis are three components: an evaluator, an abstract machine, and a compiler targeting the Java Virtual Machine (JVM). Each part is important for the development of RIMP programs, contributing to a cohesive environment.

The evaluator serves as a fast way to validate the results of your program, allowing developers to iterate and explore their programs, and the RIMP language. Complementary to this, the abstract machine allows for bidirectional debugging of RIMP programs. By utilising the reversible nature of RIMP, developers are able to step forwards and backwards through their programs and introspect the program state in order to debug and understand the computation. The compiler acts as the final step in this process, allowing developers to compile their programs to a common and portable target to be run on many systems while still being reversible.

Through this thesis, we are faced with many challenges. Foremost, are the difficulties of extending the theoretical foundations provided in the seminal RIMP paper[16] to our needs, while also addressing, and overcoming the nuances of a practical implementation.

Chapter 2

Context

2.1 Reversible Computing

Reversible computing is a paradigm where computation is both forward deterministic, and backward deterministic[43]. This allows a reversible program to be run forwards, stopped at any point, and then rerun backwards, completely reconstructing the state in which it started. Unlike traditional computing, which is not backward deterministic, which inevitably leads to information loss. This loss of information is then, in turn, dissipated as heat energy[33]. Reversible computing allows the conservation of information, and therefore energy, throughout computation[5].

Research in this area is diverse, but can be split by their level of abstraction. While there is no widespread use of reversible hardware, there has still been significant research into programming abstractions which may run on top of them.

2.2 Programming Languages

Programming languages serve as abstractions over the underlying computational model, providing a structure easier for humans to reason and communicate instructions with each other and the computer.

The structure of a programming language is largely dictated by the underlying model. Imperative languages abstract over the Turing Machine[53], while Functional languages abstract over Lambda Calculus[9]. Additionally, programming languages are also impacted by the level of abstraction they wish to provide, being split into high-level and low-level languages. A low-level language is one which directly follows the underlying model, without hiding much detail from the user, while high-level languages aim to hide much of this detail.

The area of programming languages is an open and active area of research. This includes type systems, and memory management[22].

2.3 Compilers, Interpreters, & Abstract Machines

Programming languages often cannot be run directly on hardware, and need some form of translation to occur, this can happen through many methods, such as compilers, interpreters, and abstract machines. While they differ in how the program is executed, they share many components.

Additionally, each provide benefits and drawbacks, but work well to complement each other throughout the development process.

2.3.1 Lexing

Lexing is typically performed first in the context of compilation and interpretation[2, 59]. Lexing is the process where a lexer takes some text input, and outputs a list of tokens. Each token corresponds to a part of the original text which makes sense to group together. For example, we may group letters comprising keywords into a keyword token.

This is frequently a crucial step in the process of working with programming languages, as it permits the later stages of the process to disregard irrelevant input elements, such as whitespace and comments.

2.3.2 Parsing

Parsing is the process of transforming some source, typically a token stream, into a hierarchical structure such as an abstract syntax tree (AST)[2, 59]. This is done following some specification for the grammar of the language.

The grammar defines what is a valid set of symbols that can appear, and the order in which they are allowed to appear.

2.3.3 Compilers

A compiler is a system which translates some source language to a target language, allowing the program to be directly executed on some system. This is done through a transformation from a source code language to a machine code language, which can then be directly executed on hardware.

During the process of compilation, there are typically other processes taking place. Optimisation is the process of producing some new program which is semantically identical to the original, however, minimises some property of it, such as run time, or power consumption.

Typically, compilers are placed at the end of the development cycle. Before a program is passed to its users, it will be compiled. This is done for many reasons, such as:

1. Ease of use: The compiled target can often be run directly[28].
2. Efficiency and speed: The compiler typically optimises many factors of the program[3, 29, 35].
3. Protecting IP: The original human-readable source is mutated, typically to an unrecognisable degree[4, 15].

2.3.4 Interpreters

Interpreters are programs which immediately evaluate the program they are given. Unlike compilers, they do not produce a new output which can be run after. Instead, interpreters usually produce an internal representation, such as an abstract syntax tree[1], and evaluate directly from that[47]. Interpreters are, for this reason, faster to start, and so can be a useful tool for fast iteration while developing[47, 60].

However, as interpreters are expected to immediately execute the program they are given, they don't have time to optimise like compilers do, and so are typically far slower to run the same program through when compared to compilers[47, 58].

2.3.5 Abstract Machines

Like interpreters, abstract machines evaluate program representations, however, they diverge in their operational methodologies, and the level of abstraction[1, 13]. Abstract machines function at a lower level of abstraction compared to interpreters. This difference makes abstract machines particularly valuable in debugging, as they allow the developer to observe small changes in program state after each step is evaluated. Additionally, abstract machines can, and often are used in deferred execution of a program, operating on some compiled result of a previous compilation.

[1]In practice, it is often not an AST used for the internal representation.

2.4 Reversible Languages

Reversible languages are languages which in some way enforce the constraints of reversible computing. That is, a program in a reversible language can be stopped at some point, and returned to the exact same initial state. There are many reversible languages, spanning a range of paradigms, including object-oriented, functional, and imperative[24, 52, 61].

2.4.1 Methods of Reversibility

There are several methods languages can employ to achieve reversibility. One such method is state capture, this is where all relevant state is recorded at each execution step, so when reversing the computation, you simply go back to the last command, and refresh the state to the previous point for all cases you can not reconstruct from the commands alone.

Another way to achieve reversibility is to prevent any executions which are not inherently backwards deterministic. For example, in many languages it is common to be able to overwrite a variable with a new value, such as x = 5. However, using this method, we would prohibit use of this kind of assignment as it would lose any state associated with x instead we would prefer statements of the form x += 5[18].

Another consideration of reversible languages is what we refer to as transparency, this can be thought of in a way as how high-level they are. Languages can be classed as transparent if they reflect the underlying reversible model being used, and opaque if they hide this. Essentially, if the developer has to be aware that they are targeting a reversible model of computation, then the language is transparent (or low-level), if not it is opaque (or high-level).

2.4.2 Janus

Janus is a general-purpose, reversible imperative language with support for structures such as functions[61]. Janus ensures reversibility by only allowing the developer to perform non-destructive operations. For example, we cannot destructively reassign a variable to a completely new value, as we would in conventional languages x = 5;. Rather, we must update it, for example through an addition x += 5;. The result is a reversible language, which we will class as transparent, as the developer has to be aware of the limitations of the underlying model they are using, and avoid any destructive operations. This provides the benefit of limiting the amount of information we must store to maintain reversibility, as all operations remain inherently and deterministically reversible, all we need is the program state which would be kept by any conventional program, improving the memory footprint of such a program[42, 62].

Janus has also had many implementations provided and proposed over time, largely being interpreters[14, 19, 62]. There has also been some effort to provide compilation of Janus, including to targets which are inherently reversible themselves, such as Reversible Static Single-Assignment (RSSA)[38], an intermediate representation ideal for optimisations and static analysis to be performed on[32].

However, what remains relatively unexplored in Janus is a complete environment which would allow for development, debugging, and release of software. We currently only have a disparate collection of tools without any common interoperability.

2.4.3 Hermes

Hermes is a domain-specific reversible language targetted at encryption algorithms[39, 40]. It builds on many of the same concepts present in Janus, such as non-destructive assignments. The primary benefit of this is to allow for a developer to write an encryption algorithm within Hermes, and obtain the corresponding decryption algorithm by simply reversing the program. As Hermes is reversible, this reversal is well-defined, unlike in conventional languages. This therefore guarantees that, if the encryption algorithm is correct, your decryption will be, as well as reducing the overhead for the developer.

However, as Hermes also achieves reversibility through developer restrictions, this does add overhead to the development process, increasing the complexity of developing with such a language.

2.4.4 RIMP

RIMP is a simple imperative reversible language. Unlike many other reversible languages such as Janus or Hermes, RIMP allows for destructive assignments, making it an opaque language, obscuring its underlying model from the developer. RIMP maintains reversibility by maintaining runtime values for destructive operations, allowing the values of variables to be tracked back through any destructive assignments that occur[16].

We can see that in figure 2.1, we have two initial assignments, one being 5 to the variable x, the other being 3 to the variable y. We also, within the body of the if statement, have destructive assignments, where the value of x is updated to a value with no correspondence to the original value (either 8 or 0). In this case, RIMPiler remains reversible by building a stack of value changes during runtime, so in this case we would initially obtain the value of x of

```
x := 5;
y := 3;

if x < y do {
    x := 8;
} else {
    x := 0;
}

while x > 0 do {
    y := y + 1;
}
```

Figure 2.1: A basic example of a RIMP program

$(5, +(5, 0))$ where the first element refers to the current value, and the rest refers to the changes made, here being to add 5 to 0 to initialise it with the value 5. When the destructive assignment is made in the else branch, we will update the value and its history to be $(0, -(5, +(5, 0)))$, indicating that the current value is 0, and the previous value was obtained by subtracting 5.

Another consideration with this is that RIMPiler must perform some transformations to ensure the program is reversible before execution. These will mutate if statements and while loops to ensure we can reverse them. For example, in figure 2.1 we can see that when we execute the if statement, the value of x is lost until we undo the assignment within the body, however, until we have done this, we do not know which branch was taken. Similarly, with the while loop, when finished, we have no way of telling how many iterations occurred before the condition was met[2].

Finally, in order to perform the reverse of a program, we need a program inverter, which we will refer to as the reverse function or *rev*. This function should take a program with the appropriate transformations performed as described above, and return a new program which is the semantic inverse of the original. When run, this program should return a program to its initial state before it was run in the forward direction.

[2]For more details on the specifics of these transformations, you may check Fernández et al.[16], or section 4.4

Chapter 3

Requirements

The goal of RIMPiler is to provide a complete, and cohesive framework for developing programs in RIMP, while also being extensible to allow for future development of the language and the tool.

The following lists the metrics and functionalities which this project aims to fulfil, and by which this project will be evaluated. Within the specification, RIMPiler will be used to refer to the software produced in this project.

3.1 Functional Requirements

1. RIMPiler must produce a token stream from a valid RIMP source file.
2. RIMPiler must produce a valid abstract syntax tree from a token stream produced by RIMPiler.
3. RIMPiler must be able to perform transformations on the abstract syntax tree to ensure the reversibility of the language.
4. RIMPiler must be able to produce an abstract syntax tree corresponding to the reverse of another abstract syntax tree.
5. RIMPiler must be able to evaluate an abstract syntax tree, providing information about the value of variables until the reversal point.
6. RIMPiler must be able to be run through an abstract machine.
7. The RIMPiler abstract machine must allow to step forwards and backwards

8. RIMPiler must be able to compile to the JVM providing information about the value of variables until the reversal point.

9. RIMPiler should be able to compile to an intermediate representation in SSA form.

10. RIMPiler should be able to target a reversible backend.

11. RIMPiler should be extended to allow for arrays.

12. RIMPiler should be extended to allow for user defined structures.

13. RIMPiler should be extended to allow for procedures.

14. RIMPiler should be extended to allow for multi-threading.

In order to be considered complete, requirements 1-8 must be completed, as they provide the basis for a development cycle (test-debug-release). The remaining features are nice-to-haves, or future goals of the RIMPiler framework.

3.2 Non-Functional Requirements

1. RIMPiler must compile short programs within 10 seconds on a modern desktop computer.

2. RIMPiler must give some level of feedback when things go wrong to help guide the user to the issue.

3. RIMPiler must be an extensible system which is able to evolve over time.

4. RIMPiler should be easy to learn for a developer coming from conventional languages such as C or Python.

5. RIMPiler should be a portable system, able to run on a range of system.

6. RIMPiler should be easy to develop on and extend.

To be considered a complete project, requirements 1-3 must be met, the remaining are aims, or future extensions to the project[1].

By fulfilling the core requirements, we can ensure that RIMPiler provides a complete system demonstrating the advantages of a multifaceted approach to developing reversible programming languages.

[1]Some of these requirements cannot be meaningfully evaluated in this project, as they would require a survey of users to evaluate.

Chapter 4

Specification

4.1 RIMP's Concrete Syntax

We will define the grammar used in RIMPiler as being an instance of the Program class defined in figure 4.1.

One feature to note from the grammar is the distinction between the Arithmetic[Expression,Term,Factor] and the BArithmetic[Expression,Term,Factor]. This distinction amounts to not being able to use parenthesis within any BArithmetic. While this difference is minor, it was chosen in order to greatly simplify the algorithm used to parse this grammar, which will be explored further in section 5.3. Despite the similarities between the two forms, this does add some inconvenience to developers using RIMPiler, as they may need to move parts of particularly complex arithmetic outside their conditionals, and into variables. However, this grammar is designed to be easily, and non-destructively modifiable to allow for this in the future to improve the quality of development, while still providing a complete system at the present.

Otherwise, this grammar closely follows that of C-like languages. This grammar was selected in order to be intuitive to many developers. Further, by mirroring traditional languages, it assists in making RIMP a language which is separated from its underlying model of computation, reducing further the barrier to use.

```
<Program> ::= <Statements>

<Statement> ::= skip
     | <type> identifier =<ArithmeticExpression>
     | identifier =<ArithmeticExpression>
     | if <BooleanExpression> then <Block> else <Block>
     | while <BooleanExpression> do <Block>

<Statements> ::= <Statement> ; <Statement>
     | <Statement> ;

<Block> ::= { <Statements> ; } ;
     | { <Statement> ; } ;

<ArithmeticExpression> ::= <ArithmeticTerm> + <ArithmeticExpression>
     | <ArithmeticTerm> − <ArithmeticExpression>
     | − <ArithmeticTerm>
     | <ArithmeticTerm>

<ArithmeticTerm> ::= <ArithmeticFactor> ∗ <ArithmeticTerm>
     | <ArithmeticFactor> / <ArithmeticTerm>
     | <ArithmeticFactor> ˆ <ArithmeticTerm>
     | <ArithmeticFactor>

<ArithmeticFactor> ::= ( <ArithmeticExpression> )
     | number
     | identifier

<BArithmeticExpression> ::= <BArithmeticTerm> + <BArithmeticExpression>
     | <BArithmeticTerm> − <BArithmeticExpression>
     | − <BArithmeticTerm>
     | <BArithmeticTerm>

<BArithmeticTerm> ::= <BArithmeticFactor> ∗ <BArithmeticTerm>
     | <BArithmeticFactor> / <BArithmeticTerm>
     | <BArithmeticFactor> ˆ <BArithmeticTerm>
     | <BArithmeticFactor>

<BArithmeticFactor> ::= number
     | identifier

<BooleanExpression> ::= <BArithmeticExpression> == <BArithmeticExpression>
     | <BArithmeticExpression> < <BArithmeticExpression>
     | <BArithmeticExpression> > <BArithmeticExpression>
     | <BArithmeticExpression> != <BArithmeticExpression>
     | <BooleanTerm>

<BooleanTerm> ::= <BooleanFactor> && <BooleanExpression>
     | <BooleanFactor >||<BooleanExpression>
     | ! <BooleanExpression>
     | <BooleanFactor>

<BooleanFactor> ::= (<BooleanExpression >)

<type> ::= int
     | float
```

Figure 4.1: Grammar of RIMP

4.2 RIMP's Type System

The aim of RIMPiler's type system is to balance ease of use with error avoidance. For this, we introduce some basic types in the current version of RIMPiler: 32-bit integers, denoted int, and 32-bit floating-point, denoted float. Within this system, we have a form of type polymorphism, where int and float are both treated as subtypes of an abstract numeric type, which in the future will allow for the extension of this numeric type to encompass more numerics, as well as additional types in a similar manner.

Currently, RIMPiler has a basic type system consisting of only numerics and booleans. We formally define it in figure 4.2.

$$\text{numeric-binop}\,\frac{\Gamma \vdash e_1 : \mathrm{T} \qquad \Gamma \vdash e_2 : \text{Numeric}}{\Gamma \vdash e_1\ op\ e_2 : \mathrm{T}}\,\text{where } op \in \{+, -, *, /, \hat{}\,\}$$

$$\text{numeric-unop}\,\frac{\Gamma \vdash e : \mathrm{T}}{\Gamma \vdash op\ e : \mathrm{T}}\,\text{where } op \in \{-\}$$

$$\text{boolean-relational-binop}\,\frac{\Gamma \vdash e_1 : \text{Numeric} \qquad \Gamma \vdash e_2 : \text{Numeric}}{\Gamma \vdash e_1\ op\ e_2 : \text{Boolean}}\,\text{where } op \in \{>, <, ==, !=\}$$

$$\text{boolean-logical-binop}\,\frac{\Gamma \vdash e_1 : \text{Boolean} \qquad \Gamma \vdash e_2 : \text{Boolean}}{\Gamma \vdash e_1\ op\ e_2 : \text{Boolean}}\,\text{where } op \in \{\&\&, ||\}$$

$$\text{boolean-unop}\,\frac{\Gamma \vdash e : \text{Boolean}}{\Gamma \vdash op\ e : \text{Boolean}}\,\text{where } op \in \{!\}$$

Figure 4.2: Formal definition of the type system within RIMPiler

In order to maintain ease of use, we also employ type inference and coercion. From the type rules, we can obtain the method to perform type inference. We aim to apply type inference and coercion to only certain cases in which it would be cumbersome to have to denote the types, such as in arithmetic operations. For example, we aim to avoid cases like $\text{int}\ \text{x} = (\text{int}\ \text{x}) * (\text{int}\ 2)$, preferring $\text{x} = \text{x} * 2$. We do this by applying the rules for numeric−binop, numeric−unop. We only use this on these two rules as to avoid any unintended type coercions, such as an integer being interpreted as a boolean within a conditional, which may lead to logical errors. Further, we aim to restrict this further as more types will be introduced, to ensure programs written in RIMPiler are free of type errors, or logical errors, only applying coercions to certain type groups such as numerics.

4.3 RIMP's Semantics

We present the complete semantics of RIMPiler we use, extending the existing semantics of RIMP for types. We will denote the application of an arbitrary operator as $\overline{op}$.

$$\text{const}\ \frac{}{\langle c, s\rangle \Downarrow \langle c, s\rangle \text{ if } \Gamma \vdash c : \text{Numeric}} \qquad \text{var}\ \frac{}{\langle !l, s\rangle \Downarrow \langle n, s\rangle \text{ if } l \in s \wedge s(l) = n}$$

$$\text{skip}\ \frac{}{\langle \mathit{skip}, s\rangle \Downarrow \langle \mathit{skip}, s\rangle} \qquad \text{seq}\ \frac{\langle C_1, s\rangle \Downarrow \langle \mathit{skip}, s'\rangle \quad \langle C_2, s'\rangle \Downarrow \langle \mathit{skip}, s''\rangle}{\langle C_1; C_2, s\rangle \Downarrow \langle \mathit{skip}, s''\rangle}$$

$$\text{unary boolean}\ \frac{\langle E_1, s\rangle \Downarrow \langle b_1, s\rangle}{\langle op\ E_1, s\rangle \Downarrow \langle b, s\rangle \text{ if } op \in \{\neg\} \wedge \Gamma \vdash b_1 : \text{Boolean} \wedge b = \overline{op}\ b_1}$$

$$\text{unary numeric}\ \frac{\langle E_1, s\rangle \Downarrow \langle n_1, s\rangle}{\langle op\ E_1, s\rangle \Downarrow \langle n, s\rangle \text{ if } op \in \{-\} \wedge \Gamma \vdash n_1 : \text{Numeric} \wedge n = \overline{op}\ n_1}$$

$$\text{numeric}\ \frac{\langle E_1, s\rangle \Downarrow \langle n_1, s\rangle \quad \langle E_2, s\rangle \Downarrow \langle n_2, s\rangle}{\langle E_1\ op\ E_2, s\rangle \Downarrow \langle n, s\rangle \text{ if } op \in \{+, -, /, *, \hat{}\ \} \wedge \Gamma \vdash n_1, n_2 : \text{Numeric} \wedge n = n_1 \overline{op}\ n_2}$$

$$\text{logical}\ \frac{\langle E_1, s\rangle \Downarrow \langle b_1, s\rangle \quad \langle E_2, s\rangle \Downarrow \langle b_2, s\rangle}{\langle E_1\ op\ E_2, s\rangle \Downarrow \langle b, s\rangle \text{ if } op \in \{\&\&, ||\} \wedge \Gamma \vdash n_1, n_2 : \text{Boolean} \wedge b = b_1 \overline{op}\ b_2}$$

$$\text{relation}\ \frac{\langle E_1, s\rangle \Downarrow \langle n_1, s\rangle \quad \langle E_2, s\rangle \Downarrow \langle n_2, s\rangle}{\langle E_1\ op\ E_2, s\rangle \Downarrow \langle b, s\rangle \text{ if } op \in \{>, <, ==, != \} \wedge \Gamma \vdash n_1, n_2 : \text{Numeric} \wedge b = n_1 \overline{op}\ n_2}$$

$$:=\ \frac{\langle E, s\rangle \Downarrow \langle n, s\rangle}{\langle l := E, s\rangle \Downarrow \langle \mathit{skip}, s[l \to (n, +(n_1, s(l)))]\rangle \text{if } l \in s \wedge \Gamma \vdash n : \text{Numeric} \wedge n_1 = n - s(l)}$$

$$=:\ \frac{\langle E, s[l \to (n - n_1, v)]\rangle \Downarrow \langle n, s[l \to (n - n_1, v)]\rangle}{\langle l =: E, s[l \to (n, +(n_1, v))]\rangle \Downarrow \langle \mathit{skip}, s[l \to (n - n_1, v)]\rangle \text{if } l \in s_1 \wedge \Gamma \vdash n : \text{Numeric}}$$

$$\text{if true}\ \frac{\langle E, s\rangle \Downarrow \langle \mathit{true}, s\rangle \quad \langle C_1, s\rangle \Downarrow \langle \mathit{skip}, s'\rangle}{\langle \text{if } E \text{ then } C_1 \text{ else } c_2, s\rangle \Downarrow \langle \mathit{skip}, s'\rangle \text{if } \Gamma \vdash E : \text{Boolean}}$$

$$\text{if false}\ \frac{\langle E, s\rangle \Downarrow \langle \mathit{false}, s\rangle \quad \langle C_2, s\rangle \Downarrow \langle \mathit{skip}, s'\rangle}{\langle \text{if } E \text{ then } C_1 \text{ else } c_2, s\rangle \Downarrow \langle \mathit{skip}, s'\rangle \text{if } \Gamma \vdash E : \text{Boolean}}$$

$$\text{while true}\ \frac{\langle E, s\rangle \Downarrow \langle \mathit{true}, s\rangle \quad \langle C, s\rangle \Downarrow \langle \mathit{skip}, s'\rangle \quad \langle \text{while } E \text{ do } C, s'\rangle \Downarrow \langle \mathit{skip}, s''\rangle}{\langle \text{while } E \text{ do } C, s\rangle \Downarrow \langle \mathit{skip}, s''\rangle \text{if } \Gamma \vdash E : \text{Boolean}}$$

$$\text{while false}\ \frac{\langle E, s\rangle \Downarrow \langle \mathit{false}, s\rangle}{\langle \text{while } E \text{ do } C, s\rangle \Downarrow \langle \mathit{skip}, s\rangle \text{if } \Gamma \vdash E : \text{Boolean}}$$

Figure 4.3: Axioms and semantic rules defining RIMPiler's evaluation, extending that of Fernández et al.[16]

From the semantic system presented in figure 4.3, we can see the enforcing of our type rules within the construction of expressions, ensuring that the program will be free of runtime errors.

4.4 Semantic Transformations for Reversibility

To ensure the reversibility of RIMP, we must apply the following semantic transformations: if remapping, and while counters[1].

4.4.1 If Remapping

if remapping must ensure that any variables used within the condition of an if statement are not modified within the body, ensuring we can reconstruct which branch was taken.

For example, if we had the following program

```
int x := 4;
if x < 5 do {
    x := x + 1;
} else {
    x := x − 1;
};
```

Figure 4.4: Example of an if statement which needs to be transformed

From this, we observe that when executing the program, whichever branch is taken results in the value of X changing, however, this transformation will also need to apply if only one branch modifies the variable.

We will define a function modifies which will check if a RIMP program modifies a given variable, and a function get_variables which will return all variables contained within a RIMP program. We can then check if an if statement needs to be remapped through the following.

```
define check_if_statement(statement) {
    if statement is if_statement {
        let variables = get_variables statement.condition

        for variable in variables {
            if (modifies statement.if_block variable) {
                return true
            } else if (modifies statement.else_block variable) {
                return true
            }
        }
    }

    return false
}
```

Figure 4.5: Pseudocode to check if an if statement needs to be remapped

[1]This follows closely the definitions provided by RIMP, however, is included here for reader's convenience, and clarity.

If this is true on a given statement, we must then perform a remapping to the if statement. To do this, we must exchange the variables in the condition with new variables which remain unchanged. This swap preserves the semantics, but also allows us to ensure the correct branch of the if statement is taken when reversing the program by maintaining unmodified copies of the variables after the branch. Using the example from 4.4 we will obtain the following.

```
int x := 4;
int reserved_x := x;
if reserved_x < 5 do {
    x := x + 1;
} else {
    x := x − 1;
};
```

Figure 4.6: Transformation result of 4.4

4.4.2 While Counters

While transformations ensure a counter is associated with each while loop. With each while loop in RIMP, we associate an index i with it, allowing us to identify other structures associated with it. This index is unique to this while loop, and can simply be the number of while loops preceding it. With this, we can then insert two commands, the first command must come before the while statement, and initialise a counter to 0. This counter must also be associated with the same index as the while loop we are currently transforming. We then add in an increment statement to the end of the body of the while loop, incrementing its associated counter variable.

For example, we may have the following program.

```
int x := 4;
while x > 0 do {
    x := x − 1;
}
```

Figure 4.7: Example of a while loop which needs to be transformed

Through the while counter transformation, we will obtain the following modified program.

```
int x := 4;
int counter_0 = 0;
while x > 0 do {
    x := x − 1;
    counter_0 := counter_0 + 1;
}
```

Figure 4.8: Transformation result of 4.7

We will then find that by the end of the execution of this program, the value at counter_0 will be 4, as expected from x.

4.5 Reverse Function

Within RIMPiler, we will make use of two similar, but subtly different reverse functions. These are used in either the conventional backend or in the abstract machine. In the case of the former, we only ever reverse at the end of the forward computation, while the later may reverse at any point.

In the general case used by the abstract machine, we must use an additional structure to perform the reverse function, which is the *while_table*. This while table maps an instance of *while* indexed on some i to the expression it has in the forward direction. This store acts as a way to recover the initial condition of while loops to restore a reversed program to its original form.

$$\begin{aligned}
rev(E) =&\quad E && \text{(expression)}\\
rev(x := E) =&\quad x =: E && \text{(assignment)}\\
rev(x =: E) =&\quad x := E && \text{(reverse assignment)}\\
rev(\text{skip}) =&\quad \text{skip} && \text{(skip)}\\
rev(\text{if } E \text{ then } C_1 \text{ else } C_2) =&\quad \text{if } E \text{ then } rev(C_1) \text{ else } rev(C_2) && \text{(if)}\\
rev(\text{while}_i\ E \text{ do } C) =&\quad \text{while}_i\ !counter_i > 0 \text{ do } C \quad if\ counter_i \notin E && \text{(while forward)}\\
rev(\text{while}_i\ !counter_i > 0 \text{ do } C) =&\quad \text{while}_i\ E \text{ do } C \quad if\ while\_table(i) = E && \text{(while backward)}
\end{aligned}$$

Figure 4.9: The generic reverse function function

RIMPiler simplifies on this somewhat in the implementation of the conventional compilation targets. In RIMPiler's conventional targets, we only reverse these targets once, during compilation, to obtain the reverse of the entire program. For this reason, we will never have to perform anything of the form $rev(rev(P))$. Because of this, we can simplify the generic case of the *rev* function by dropping all cases which cannot occur in the forward direction of RIMP. We obtain a new definition without reverse assignment and while backward. This has the additional benefit of not requiring the *while_table* in these cases.

4.6 Abstract Machine

The abstract machine is a 4-tuple $\langle c, r, s, b\rangle$ where c is the control stack, r is the result stack, s is the store, and b is the back stack. Due to the nature of the *rev* function, we will also consider the *while_table* to be part of the abstract machine, however, we will omit it from the rules.

The purpose of the control stack is to hold the elements to be executed next, while that of the back stack is to hold the reverse of these. The result stack is used to store intermediate results. The store is used to store variable values and their runtime values.

We will define the tuple elements in figure 4.10 in section A.1.1 of the appendix.

$$
\begin{array}{rl}
c, b ::= & C \\
\\
C ::= & nil \\
| & lab \cdot C \\
| & program \cdot C \\
| & l \cdot C \\
r ::= & nil \\
| & program \cdot r \\
| & l \cdot r
\end{array}
$$

Figure 4.10: Composition of stacks in RIMPiler's abstract machine

The objective is to implement the abstract machine outlined by Fernández et al.[16], incorporating new operators and types. However, during the adaptation process, challenges arose particularly concerning erroneous states, notably within loop structures[2].

Instead, we present a new abstract machine, heavily based on the original, but adapted to overcome these issues. There is a minimal and informal proof for correspondence between the abstract machine presented in this paper and that of the big-step semantics provided in figure 4.3 which can be found in appendix A.1.3.

The main differences provided are as follows:

- Explicit inverse expressions, which require explicit rules. This is to simplify implementation, while allowing expansion of this system more easily in the future.

- Aggregation of unary operators into a set of rules, this then allows for any number of unary operators to extend the current operators with minimal effort.

- Addition of a cleaning step in the *EndW* rule to ensure no additional copies of the while condition and body are left on the result stack[3].

[2]Unfortunately, there was not enough time to thoroughly investigate this issue; please refer to appendix A.1.2 for details

[3]This is to resolve the issue mentioned in footnote 2

In the following rules, we will use an underline to denote the inverse of the symbol, for example E's inverse would be $\underline{E}$. we use an overline to denote the standard application of the operation, so $\overline{op}$ indicates the standard application of the operator op. We will also add an asterisk beside the name of any rule which is significantly updated. It is also important to note that by this point we assume type checking has occurred, and so we do not handle any erroneous types here.

$$
\begin{aligned}
\langle n \cdot c, r, m, b\rangle &\overset{num}{\rightarrow} \langle c, n \cdot r, m, \underline{n} \cdot b\rangle \\
\langle \underline{n} \cdot c, n \cdot r, m, b\rangle &\overset{mun}{\rightarrow} \langle c, r, m, n \cdot b\rangle \\[1em]
\langle !l \cdot c, r, m, b\rangle &\overset{var}{\rightarrow} \langle c, m(l) \cdot r, m, \underline{!l} \cdot b\rangle \\
\langle \underline{!l} \cdot c, n \cdot r, m, b\rangle &\overset{rav}{\rightarrow} \langle c, r, m, !l \cdot b\rangle \\[1em]
\langle (oper\ E) \cdot c, r, m, b\rangle &\overset{unexp*}{\rightarrow} \langle E \cdot oper \cdot c, r, m, unexp \cdot \underline{E} \cdot b\rangle \\
&\quad \text{where } oper = op \text{ or } \underline{op} \\
\langle unexp \cdot \underline{E} \cdot c, r, m, E \cdot oper \cdot b\rangle &\overset{unpxe*}{\rightarrow} \langle c, r, m, (oper\ E) \cdot b\rangle \\[1em]
\langle op \cdot c, n_1 \cdot r, m, \underline{E} \cdot unexp \cdot \underline{E} \cdot b\rangle &\overset{unop*}{\rightarrow} \langle c, n \cdot r, m, (op\ E) \cdot b\rangle \\
&\quad \text{where } n = \overline{op}\ n_1 \\
\langle \underline{op} \cdot c, n_1 \cdot n \cdot r, m, \underline{E} \cdot unexp \cdot \underline{E} \cdot b\rangle &\overset{unpo*}{\rightarrow} \langle c, r, m, (op\ E) \cdot b\rangle \\
&\quad \text{where } n = \overline{op}\ n_1 \\[1em]
\langle (E_1\ oper\ E_2) \cdot c, r, m, b\rangle &\overset{binexp}{\rightarrow} \langle E_1 \cdot E_2 \cdot oper \cdot c, r, m, binexp \cdot \underline{E_1} \cdot \underline{E_2} \cdot b\rangle \\
&\quad \text{where } oper = op \text{ or } \underline{op} \\
\langle binexp \cdot \underline{E_1} \cdot \underline{E_2} \cdot c, r, m, E_1 \cdot E_2 \cdot oper \cdot b\rangle &\overset{binpxe}{\rightarrow} \langle c, r, m, (E_1\ oper\ E_2) \cdot b\rangle \\[1em]
\langle op \cdot c, n_2 \cdot n_1 \cdot r, m, \underline{E_2} \cdot \underline{E_1} \cdot binexp \cdot \underline{E_1} \cdot \underline{E_2} \cdot b\rangle &\overset{binop}{\rightarrow} \langle c, n \cdot r, m, (E_1\ op\ E_2) \cdot b\rangle \\
&\quad \text{where } n = n_1\ \overline{op}\ n_2 \\
\langle \underline{op} \cdot c, n_2 \cdot n_1 \cdot n \cdot r, m, \underline{E_2} \cdot \underline{E_1} \cdot binexp \cdot \underline{E_1} \cdot \underline{E_2} \cdot b\rangle &\overset{binpo}{\rightarrow} \langle c, r, m, (E_1\ op\ E_2) \cdot b\rangle \\
&\quad \text{where } n = n_1\ \overline{op}\ n_2
\end{aligned}
$$

Figure 4.11: Abstract Machine rules for expressions

With the expressions, we can see the only changes made here are to consolidate the neg, $\neg$, and their inverse rules into a generic unary expression set of rules.

We can see from the command rules that we require the rev function, and so we must maintain the *while_table* as part of the abstract machine.

$$\langle skip \cdot c, r, m, b\rangle \overset{skip}{\rightarrow} \langle c, r, m, skip \cdot b\rangle$$

$$\langle (l := E) \cdot c, r, m, b\rangle \overset{asgn}{\rightarrow} \langle E \cdot !l \cdot := \cdot c, l \cdot r, m, asgn \cdot E \cdot b\rangle$$
$$\langle asgn \cdot E \cdot c, l \cdot r, m, E \cdot !l \cdot := \cdot b\rangle \overset{ngsa}{\rightarrow} \langle c, r, m, (l := E) \cdot b\rangle$$

$$\langle := \cdot c, n_2 \cdot n_1 \cdot l \cdot r, m, \underline{!l} \cdot \underline{E} \cdot asgn \cdot E \cdot b\rangle \overset{:=}{\rightarrow} \langle c, r, m[l \mapsto (n_1, +(n_1 - n_2, m(l)))], (l =: E \cdot b\rangle$$

$$\langle (l =: E \cdot c, r, m[l \mapsto (n_1, +(n, v))], b\rangle \overset{asgn^r}{\rightarrow} \langle E \cdot !l \cdot =: \cdot c, l \cdot r, m[l \mapsto (n_1 - n, v)], asgn^r \cdot n \cdot E \cdot b\rangle$$
$$\langle asgn^r \cdot n \cdot E \cdot c, l \cdot r, m, E \cdot !l \cdot =: \cdot b\rangle \overset{ngsa^r}{\rightarrow} \langle c, r, m[l \mapsto (m(l) + n, +(n, m(l)))], (l =: E \cdot b\rangle$$

$$\langle =: \cdot c, n_2 \cdot n_1 \cdot l \cdot r, m[l \mapsto (n_1, +(n_1 - n_2, m(l)))], \underline{!l} \cdot \underline{E} \cdot asgn^r \cdot n \cdot E \cdot b\rangle \overset{=:}{\rightarrow}$$
$$\langle c, r, m, (l := E) \cdot b\rangle$$

$$\langle (C_1; C_2) \cdot c, r, m, b\rangle \overset{seq}{\rightarrow} \langle C_1 \cdot C_2 \cdot ; \cdot c, r, m, seq \cdot b\rangle$$
$$\langle seq \cdot c, r, m, C_1 \cdot C_2 \cdot ; \cdot b\rangle \overset{qes}{\rightarrow} \langle c, r, m, (C_1; C_2) \cdot b\rangle$$
$$\langle ; \cdot c, r, m, rev(C_2) \cdot rev(C_1) \cdot seq \cdot b\rangle \overset{;}{\rightarrow} \langle c, r, m, (rev(C_2); rev(C_1)) \cdot b\rangle$$

Figure 4.12: Abstract Machine rules for commands

$$\langle (if\ E\ then\ C_1\ else\ C_2) \cdot c, r, m, b\rangle \overset{cond}{\rightarrow} \langle E \cdot if \cdot cond \cdot c, C_1 \cdot C_2 \cdot r, m, \underline{cond} \cdot b\rangle$$
$$\langle \underline{cond} \cdot c, C_1 \cdot C_2 \cdot r, m, E \cdot if \cdot cond \cdot b\rangle \overset{donc}{\rightarrow} \langle c, r, m, (if\ E\ then\ C_1\ else\ C_2) \cdot b\rangle$$

$$\langle if \cdot cond \cdot c, true \cdot C_1 \cdot C_2 \cdot r, m, \underline{E} \cdot \underline{cond} \cdot b\rangle \overset{if_T}{\rightarrow} \langle C_1 \cdot cond \cdot c, C_1 \cdot C_2 \cdot r, m, E \cdot \underline{if} \cdot \underline{cond} \cdot b\rangle$$
$$\langle \underline{if} \cdot \underline{cond} \cdot c, true \cdot C_1 \cdot C_2 \cdot r, m, \underline{E} \cdot C_1 \cdot cond \cdot b\rangle \overset{fi_T}{\rightarrow} \langle \underline{E} \cdot \underline{cond} \cdot c, true \cdot C_1 \cdot C_2 \cdot r, m, if \cdot cond \cdot b\rangle$$

$$\langle if \cdot cond \cdot c, false \cdot C_1 \cdot C_2 \cdot r, m, \underline{E} \cdot \underline{cond} \cdot b\rangle \overset{if_F}{\rightarrow} \langle C_2 \cdot cond \cdot c, C_1 \cdot C_2 \cdot r, m, E \cdot \underline{if} \cdot \underline{cond} \cdot b\rangle$$
$$\langle \underline{if} \cdot \underline{cond} \cdot c, false \cdot C_1 \cdot C_2 \cdot r, m, \underline{E} \cdot C_2 \cdot cond \cdot b\rangle \overset{fi_F}{\rightarrow} \langle \underline{E} \cdot \underline{cond} \cdot c, false \cdot C_1 \cdot C_2 \cdot r, m, if \cdot cond \cdot b\rangle$$

$$\langle cond \cdot c, C_1 \cdot C_2 \cdot r, m, rev(C) \cdot E \cdot \underline{if} \cdot \underline{cond} \cdot b\rangle \overset{endif}{\rightarrow} \langle c, r, m, (if\ E\ then\ rev(C_1)\ else\ rev(C_2)) \cdot b\rangle$$

$$\langle \underline{E} \cdot \underline{cond} \cdot c, true \cdot C_1 \cdot C_2 \cdot r, m, b\rangle \overset{ifrexp_T *}{\rightarrow} \langle \underline{cond} \cdot c, C_1 \cdot C_2 \cdot r, m, rev(C_1) \cdot E \cdot b\rangle$$
$$\langle \underline{E} \cdot \underline{cond} \cdot c, false \cdot C_1 \cdot C_2 \cdot r, m, b\rangle \overset{ifrexp_F *}{\rightarrow} \langle \underline{cond} \cdot c, C_1 \cdot C_2 \cdot r, m, rev(C_2) \cdot E \cdot b\rangle$$

Figure 4.13: Abstract Machine rules for conditionals

The conditionals add two additional rules to handle the explicitly inverted expressions, these rules will only be required when reversing midway through an if statement.

$$\langle (while_i\ E\ do\ C) \cdot c, r, m, b\rangle \overset{loop}{\rightarrow} \langle E \cdot while_i \cdot loop_i \cdot c, E \cdot C \cdot r, m, \underline{loop_i} \cdot b\rangle$$

$$\langle \underline{loop_i} \cdot c, E \cdot C \cdot r, m, E \cdot while_i \cdot loop_i \cdot b\rangle \overset{pool}{\rightarrow} \langle c, r, m, (while_i\ E\ do\ C) \cdot b\rangle$$

$$\langle while_i \cdot loop_i \cdot c, true \cdot E \cdot C \cdot r, m, \underline{E} \cdot \underline{loop_i} \cdot b\rangle \overset{loop_T *}{\rightarrow}$$
$$\langle C \cdot (while_i\ E\ do\ C) \cdot c, E \cdot C \cdot r, m, true \cdot \underline{while_i} \cdot \underline{loop_i} \cdot b\rangle$$
$$\text{where } counter_i = counter_i + 1$$
$$if\ counter_i \notin E$$

$$\langle \underline{while_i} \cdot \underline{loop_i} \cdot c, true \cdot E \cdot C \cdot r, m, \underline{true} \cdot C \cdot (while_i\ E\ do\ C) \cdot b\rangle \overset{pool_T}{\rightarrow}$$
$$\langle \underline{E} \cdot \underline{loop_i} \cdot c, true \cdot E \cdot C \cdot r, m, while_i \cdot loop_i \cdot b\rangle$$

$$\langle while_i \cdot loop_i \cdot c, false \cdot E \cdot C \cdot r, m, \underline{E} \cdot \underline{loop_i} \cdot b\rangle \overset{loop_F}{\rightarrow} \langle loop_i \cdot c, E \cdot C \cdot r, m, false \cdot \underline{while_i} \cdot \underline{loop_i} \cdot b\rangle$$

$$\langle \underline{while_i} \cdot \underline{loop_i} \cdot c, false \cdot E \cdot C \cdot r, m, \underline{false} \cdot loop_i \cdot b\rangle \overset{pool_F}{\rightarrow} \langle \underline{E} \cdot \underline{loop_i} \cdot c, fasle \cdot E \cdot C \cdot r, m, while_i \cdot loop_i \cdot b\rangle$$

$$\langle loop_i \cdot c, E \cdot C \cdot r, m, false \cdot \underline{while_i} \cdot \underline{loop_i} \cdot b\rangle \overset{endw_F}{\rightarrow}$$
$$\langle loop_i \cdot c, 0 \cdot rev(while_i\ E\ do\ C) \cdot E \cdot C \cdot r, m, \underline{endw_i} \cdot b\rangle$$

$$\langle \underline{endw_i} \cdot c, 0 \cdot rev(while_i\ E\ do\ C) \cdot E \cdot C \cdot r, m, loop_i \cdot b\rangle \overset{wend_F}{\rightarrow} \langle false \cdot \underline{while_i} \cdot \underline{loop_i} \cdot c, E \cdot C \cdot r, m, loop_i \cdot b\rangle$$

$$\langle loop_i \cdot c, n \cdot rev(while_i\ E\ do\ C) \cdot E \cdot C \cdot r, m, \underline{endw_i} \cdot rev(C) \cdot true \cdot \underline{while_i} \cdot \underline{loop_i} \cdot b\rangle \overset{endw_T}{\rightarrow}$$
$$\langle loop_i \cdot c, n+1 \cdot rev(while_i\ E\ do\ C) \cdot E \cdot C \cdot r, m, \underline{endw_i} \cdot b\rangle$$

$$\langle \underline{endw_i} \cdot c, n+1 \cdot rev(while_i\ E\ do\ C) \cdot E \cdot C \cdot r, m, loop_i \cdot b\rangle \overset{wend_T}{\rightarrow}$$
$$\langle \underline{endw_i} \cdot rev(C) \cdot true \cdot \underline{while_i} \cdot \underline{loop_i} \cdot c, n \cdot rev(while_i\ E\ do\ C) \cdot E \cdot C \cdot r, m, loop_i \cdot b\rangle$$

$$\langle loop_i \cdot c, n \cdot rev(while_i\ E\ do\ C) \cdot E \cdot C \cdot r, m, \underline{endw_i} \cdot b\rangle \overset{endw*}{\rightarrow} \langle c, r, m, rev(while_i\ E\ do\ C) \cdot b\rangle$$

$$\langle c, E \cdot C \cdot r, m, b\rangle \overset{cloop*}{\rightarrow} \langle c, r, m, b\rangle$$

$$\langle \underline{E} \cdot \underline{loop_i} \cdot c, n \cdot E \cdot C \cdot r, m, while_i \cdot loop_i \cdot b\rangle \overset{wrexp*}{\rightarrow} \langle \underline{loop_i} \cdot c, E \cdot C \cdot r, m, E \cdot while_i \cdot loop_i \cdot b\rangle$$

Figure 4.14: Abstract Machine rules for loops

Finally, with the loop rules, we have four main modifications. As with the conditional, we have an explicit inverse expression rule. We also explicitly mark the counter modification within $loop_T$. In order to overcome the issues with looping state, we have modified the *endw* rule in order to also remove the $\underline{endw_i}$ from the back stack, and then introduced a new rule *cloop* to clean the remaining copies of the loop's condition and body from the result stack.

Chapter 5

Design & Implementation

5.1 Architecture

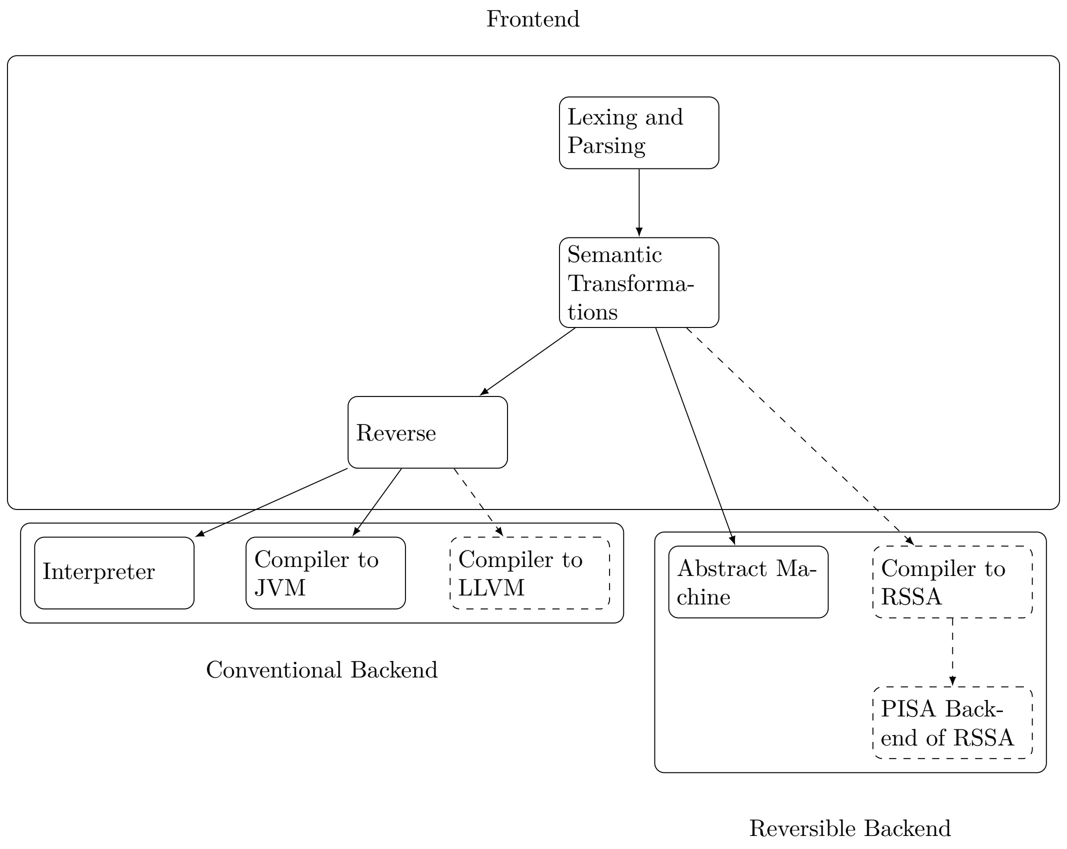

Figure 5.1: High level overview of the system architecture, dotted lines are used to indicate systems not implemented within the current version of RIMPiler, but which are logical future steps to build.

RIMPiler is intended as a complete framework for the language RIMP. For this reason, its architecture has modularity in mind, allowing each component to share infrastructure built by others. Additionally, it allows for new infrastructure to be built and included fairly easily.

Broadly, we have three systems, a frontend, a conventional backend, and a reversible backend.

5.1.1 Frontend

The frontend comprises the functionality shared between most backend systems.

The lexing and parsing component handles taking a RIMP program, and producing a plain abstract syntax tree (AST). This tree closely resembles the structure expected of a conventional language, and does not yet embed the required infrastructure for reversibility.

The semantic transformations component handles embedding the infrastructure for reversibility into the AST produced by the previous step. It then produces an AST which can be used by future components.

The reverse component is required for all conventional backends. The reverse component takes the transformed AST, and applies the reverse function to it, producing the reverse execution of the program. As conventional backends are not implicitly reversible, they require this additional step in order to reverse the computation at the end of the forward computation.

5.1.2 Conventional Backend

Each component of the conventional backend accepts an AST which has the requisite semantic transformations applied, and which includes the reverse execution of the program. The job of each component in the backend is to evaluate the program directly, or produce compiled code for the AST. Within RIMPiler, an interpreter and a compiler to the JVM is included. Other systems may be included within this larger component in the future, such as a backend targeting LLVM.

5.1.3 Reversible Backend

The reversible backend mirrors the conventional backend, however, the components it encompasses are inherently reversible. For this reason, they do not require the reverse component from the frontend, and can handle the reversal of the program independently.

Within RIMPiler, an abstract machine component is provided, with the ability to step through execution.

5.2 Lexing Algorithm

There are many methods by which lexers can be implemented, each with certain benefits and drawbacks. One common consideration, however, is to produce a lexer which is POSIX compliant. A POSIX compliant lexer is a lexer in which the leftmost longest string is always matched[27]. This helps keep the rules we use to match strings simple, and disambiguate potential matches.

There are several ways to implement POSIX lexers, two of the most common approaches being using finite automata backed regular expression matchers[7, 36], and using derivative backed regular expression matchers[50, 54].

Finite automata backed regular expression matchers are some of the most common implementations of regular expression engines. They are often available as a default in many programming languages[21, 45]. As they are readily available, they are relatively easy to use within lexing algorithms. However, one issue with these kinds of matchers, is that they may suffer from catastrophic backtracking. Catastrophic backtracking occurs due to the fact that many of the finite automata's used here are actually non-deterministic. This non-determinism, in certain situations, inevitably leads to the need to backtrack in certain matching cases.

Unlike finite automata regular expression matchers, derivative based matchers do not suffer from backtracking. Derivative based matchers are based on the process of reducing a regular expression based on each character observed until you either reach a match, or finish the characters[8, 50, 54]. In addition, with derivative matchers, there is no need to construct any automata. This reduces both the computation needed up front, and the space used to store the automata, making them an efficient choice of matcher. Derivative matchers are, however, relatively uncommon still, not appearing in common language's default implementation.

Derivative based matching was chosen in order to ensure the speed of the lexing phase[50].

RIMPiler uses the algorithm described by Sulzmann and Lu[50] for derivative based matching. This algorithm broadly follows two phases, as seen in figure 5.2. The first phase is a continual application of the derivative function. This function reduces the input string to be matched until there are no remaining characters to match. The second phase is repeated application of the injection function, the job of this chain is to reconstruct the string which was matched in the first phase, allowing us to build tokens from the strings we match.

$$\begin{tikzcd}
r1 \arrow[r, "der a"] \arrow[d, dotted] & r2 \arrow[r, "der b"] \arrow[d, dotted] & r3 \arrow[r, "der c"] \arrow[d, dotted] & r4 \arrow[d, "mkeps"] \\
v1 & v2 \arrow[l, "injection a"'] & v3 \arrow[l, "injection b"'] & v4 \arrow[l, "injection c"']
\end{tikzcd}$$

Figure 5.2: Lexing with derivatives. This diagram is inspired by Urban 2023[51, 54], but included here for clarity.

The exact method of the derivative function is omitted for brevity. In general, the derivative function works by finding the regular expression which represents what is left to match after matching the current regular expression with the current character. This function does also rely on another supporting function, which is nullable. Nullable indicated if a regular expression is able to match the empty string or not.

One issue with lexing however, is RIMPiler is intended to be a system which is easy to evolve over time. And as such, it was intended to be easy for changes to this lexer to be made. For this reason, the lexing system was largely divided in two. One underlying system which acts as a general purpose tokeniser and regular expression matcher, and another system within RIMPiler which simply calls this system. This acts similar to a metamodel of a tokenising system, making it very flexible, and simple to extend by someone who is not directly familiar with the implementation of the underlying tokenisation algorithm.

This allows RIMPiler to define the grammar using regular expressions, tokens, and lex within two lines as seen in figure 5.3.

```
let lexer = Lexer::new(self.rimp);
let result = lexer.tokenise::<tokens::RIMPToken>(&input);
```

Figure 5.3: Call to the lexing system

5.3 Parsing Algorithm

Broadly, there are two kinds of parsing algorithms, top-down[11, 17, 44], and bottom-up[30]. Top-down parsing algorithms aim to start from the root grammar rule, and match the program with this rule. Bottom-up, on the other hand, aims to construct a valid grammar starting at the terminals, and ending on the root rule.

Bottom-up parsers are relatively common parsing techniques and often provide excellent performance. However, in the case of RIMPiler we prefer the use of top-down parsing algorithms. Top-down algorithms are typically clearer to understand and easier to extend or adapt in the initial stages.

In the case of RIMPiler, a combination of two general methods of top-down parsing were used — Pratt parsing, and a kind of recursive descent parser.

Pratt parsing is used purely on expressions within RIMPiler, as it allows for a clear operator precedence to be set, and to be easily extended in the future with new operators if needed. However, implementing a pure Pratt parser for statements as well is non-trivial, and difficult to understand. This is the reason for the hybrid approach. In the case of statements, a recursive descent parser is used instead, maintaining a simple structure to the parsing algorithm.

Like this, the problem of parsing a RIMP program is reduced into few simple functions.

```
fn parse_program(&mut self, tokens: &mut Tokens) -> Result<Program>;

fn parse_statement(&mut self, tokens: &mut Tokens) -> Result<Statement>;

fn parse_block(&mut self, tokens: &mut Tokens) -> Result<Block>;

fn parse_arithmetic_expression(
        &mut self,
        tokens: &mut Tokens,
        min_binding_power: u8,
    ) -> Result<ArithmeticExpression>;

fn parse_boolean_expression(
        &mut self,
        tokens: &mut Tokens,
        min_binding_power: u8,
    ) -> Result<BooleanExpression>;
```

Figure 5.4: Overview of the parsing algorithm for RIMPiler

Here we have two groups of functions, one for expressions, and one for program structures. the program structures are then parsed by a look ahead, where we check the following instruction, and from that, we can decide what structure we expect to see. For example, given the program:

```
int x := 4;

while x > 0 do {
    if x > 5 then {
        x := x - 4;
    } else {
        x := x - 1;
    }
}
```

We can see that after tokenising the program, the first token will be a token representing the *int* symbol. Due to the limited nature of RIMP's grammar, from this we know that what we expect to see next is an assignment where the right side is an integer. The same approach is used when we come across the *while* token and *if* token. If, while performing this algorithm, we find something unexpected we simply reject the program as being malformed, providing information about where the error originates.

5.3.1 Type System

The type system within RIMPiler is implemented as a by-product of parsing. We do this by attaching extra type information to the AST as we construct it. This is performed on every expression we construct, and aims to ensure that the program being parsed is consistent under the type rules utilising the type inference rules.

```
int x = 5.5;
```

For example, in the parsing of the previous program, we would initially see that x is the variable being declared with the *int* keyword, and as such, we will construct it with the information that it is an *int* attached. When we reach the 5.5 we will attempt to consolidate this with the type information previously gathered using all type inference rules and type checks. In this case, we would fail, as we do not allow type inference from a *float* to an *int* within assignments.

Intuitively, we can think of the type system as looking at the left-hand side of all expressions, and attempting to match the right-hand side to it. If we succeed, additional type annotations will be added to the AST, and if we fail we will treat it as a parsing error due to a malformed program and return a message describing the issue.

5.4 Semantic Transformations for Reversibility

Semantic transformations are implemented as an independent post-parse process. This allows it to easily be integrated or removed as needed by the target.

We have two transformations we aim to apply, the if remapping, and the while transformation. In both cases, we need to apply this anywhere in the program where we have an if statement or a while loop. So the first step is a recursive tree traversal, which steps through all branches of the tree looking for the given statements.

Once we find an if statement, we can perform the transformation through the help of some additional functions *get_variables_in_block* and *get_variables_in_boolean_expression*. Using these, we can check if there is any overlap between the variables in the condition, and the if statements body, if not, no transformation needs to be performed. If we find that a transformation is needed, we will use an additional helper function *remap_variables_in_boolean_expression* to map the names of variables in the condition to new variable names defined in a map, and finally, we initialise these new variable names before the if statement.

In the case we find a while loop, we must always perform the transformation, for this, we generate a new name for the while counter, and insert an initialisation declaration before the loop $int\ counter_i\ =\ 0;$. We then increment this counter at the end of the while body.

5.5 Reverse Function

In RIMPiler the reverse function is required by all conventional backends and is used to reverse the entire program after it has finished executing. As such, we never need to reverse a subset of the program, or reverse an already reverse program. This allows us to simplify the reverse function significantly. The reserve function is implemented by a tree walk, at each node we perform the reversal. We omit the implementation of reversing operations like =: or while loops with their reversed counters, as these are not permitted in an ordinary RIMPiler program, and will only occur when reversed. We also simplify the design further by relying on the fact that the program provided is in a format expected after semantic transformations are performed, this is because we also always need these transformations for conventional backends. For example, given a while loop, due to the semantic transformations, we know that the variable preceding it is the counter for that while loop, and so we reverse it with that in mind.

5.6 Interpreter

The interpreter within RIMPiler is implemented via direct execution of the AST. In order to do this, we need an additional component, being the memory store.

The memory store is a component which allows us to handle assigning and unassigning of the various variables in a RIMPiler program, currently being integers and floating-point numbers. This abstraction is provided in order to allow for easy access to values without the evaluator needing to consider the type of the variables, as this has already been handled by the type checker.

With the memory store, we can perform a modified post order traversal of the AST in order to evaluate each statement. Figure 5.5 demonstrates the result of evaluation on different structures, each can either return a value, as with expressions, or mutate memory, as with statements.

$$
\begin{array}{rlr}
\mathit{interpret}(v, s) = & v & \text{(value)} \\
\mathit{interpret}(!l, s) = & s(m) & \text{(variable)} \\
\mathit{interpret}(E_1 \ \mathit{op} \ E_2, s) = & \mathit{interpret}(E_1, s) \ \overline{\mathit{op}} \ \mathit{interpret}(E_2, s) & \text{(binary operation)} \\
\mathit{interpret}(\mathit{op} \ E, s) = & \overline{\mathit{op}} \ \mathit{interpret}(E, s) & \text{(unary operation)} \\
\mathit{interpret}(l := \mathit{op} \ E, s) = & \mathit{assign}(s, \mathit{interpret}(E, s)) & \text{(assign)} \\
\mathit{interpret}(l =: \mathit{op} \ E, s) = & \mathit{unassign}(s, \mathit{interpret}(E, s)) & \text{(unassign)} \\
\mathit{interpret}(\mathit{skip}, s) = & \mathit{skip} & \text{(skip)} \\
\mathit{interpret}(C_1; C_, s) = & \mathit{interpret}(C_1, s); \mathit{interpret}(C_2, s') & \text{(sequence)} \\
\mathit{interpret}(\mathit{if} \ E \ \mathit{then} \ C_1 \ \mathit{else} \ C_2, s) = & \mathit{if} \ \mathit{intepret}(E, s) & \\
 & \quad \mathit{interpret}(C_1, s) & \\
 & \mathit{else} & \\
 & \quad \mathit{interpret}(C_2, s) & \text{(if)} \\
\mathit{interpret}(\mathit{while} \ E \ \mathit{do} \ C, s) = & \mathit{while} \ \mathit{intepret}(E, s) & \\
 & \quad \mathit{interpret}(C, s) & \text{(while)}
\end{array}
$$

Figure 5.5: structure of interpreter

As you can see, in certain cases such as if statements and while loops, we do not have to perform a full post-order traversal, rather skipping the branches as needed depending on the evaluation of the condition, or repeated traversal in the case of a true loop.

Figure 5.5 simplifies the actual structure by ignoring the details of mutating the structure as we evaluate, rather showing the store s as an immutable store. We rather pass to each function a mutable reference to the shared store, allowing them to update it directly, and then optionally returning either a result, or void.

In order to actually allow the developer to check their results, before executing the reversed part of the program, we capture the values of all variables, as well as their runtime values in order to display them to the developer.

5.7 Compiler

The compiler is one of the key components of RIMPiler. It is intended as the final part of any development cycle.

Compilers can be implemented in many ways, such as by targeting many intermediate representations. This is a common approach, as different intermediate representations provide different advantages in terms of optimisations and type checking.

Targetting intermediate representations also provides additional benefits, in that more infrastructure can be shared between more target outputs. For example, many languages will first compile to a custom intermediate, perform some operations relevant to that language while in this representation, and then compile to targets, such as LLVM IR, X86, ARM, or others. The reason for this is it is often far easier to work with an intermediate representation than the source, allowing for simpler code, and more efficient algorithms. Another approach is to ignore this intermediate representation, this greatly simplifies the compiler, however, may limit, or interfere with future operations which must be performed such as language specific optimisations.

RIMPiler currently has no internal intermediate representations. This was done due to time constraints, and in order to maintain simplicity. Rather, RIMPiler directly targets the JVM, directly generating byte code from the AST. This is a compromise, as it restricts the abilities of RIMPiler to what is easy to do on an AST. However, the JVM is just one target, and, is a relatively simple target to rewrite from an intermediate language, so this is a minor compromise in the current version of RIMPiler.

The choice to initially target the JVM was made due to the relative simplicity as a compilation target, as well as its universality. As RIMPiler targets the JVM, any device which is capable of running Java is also capable of running RIMPiler, making it easily accessible to many users. We currently provide no target for LLVM, this is due to the significant complexity it would add to this initial implementation, likely requiring some kind of CPS implementation in order to construct an SSA form from our AST[6]. RIMPiler does, however, enable this to be added in the future with relative ease, allowing for a more performant implementation, and further extensions through intermediate representations.

The implementation of the code generator for the JVM is simply a post-order traversal of the AST. As the JVM is a stack machine, its operation follows exactly the structure of the AST we have generated for RIMP. Like this, we avoid any complications that can arise from compiling to targets such as LLVM IR, in which we would need to use methods discussed such as CPS.

One complication, however, is due to the nature of variables with RIMP. A variable is not simply a value, as it is in the JVM. In order to store the back stack of values, we have to introduce a wrapper for variables.

In RIMPiler, we have two possible variables, Integers, and Floats. As these are both singlevalued variables, unlike for example arrays, they are largely treated the same.

We provide a library implementation of RIMPInt and RIMPFloat, using these in place of raw integers and floats within the generated byte code. These provide the functions needed to assign, and unassign values, as well as additional features for debugging, and displaying output.

```
public class RIMPInt {
    String name;
    int value;
    Stack<Integer> history;

    boolean debug = false;

    public RIMPInt(String name);

    public void assign(int value);

    public void unAssign();

    public int get();

    public void print();
}
```

Figure 5.6: Interface of RIMPInt utility class for the JVM

We provide four main functions, as well as a debug flag which is used to log information during RIMPiler development. assign simply handles updating the value, as well as the history stack. unAssign, similarly, updates the value and history stack. get acts as a dereference operator, returning the value of the variable, allowing it to be used in expressions. print simply prints the value and history of the variable, along with its name, this is intended to be used at the reversal point of RIMPiler to act as an output for programs. The interface for *intergers* can be seen in figure 5.6, the implementation is similar for floats, however, in that case we store floats rather than integer values.

Once we have produced a file containing the Java assembly, we must then assemble this to the byte code format. For this, we use Krakatau[49], which is an assembler written in rust. This was chosen due to its easy interoperability with the current build system used by RIMPiler, reducing the complexity, and improving the ease for developers of the framework. Alternatives do exist, such as Jasmin[48], which provide some benefits over Krakatau, however, they would be harder to integrate into the build system. One drawback of Krakatau is its limited implementation, working on only a subset of Java's virtual machine, and not always functioning with Java's verification system, requiring the "–noverify " flag to circumvent this.

5.8 Abstract Machine

The abstract machine is a target which, unlike the conventional backends, does not require semantic transformations to be performed, except for the if transformation, which still must occur. So initially, we must perform only the if transformation to the AST.

As described in 4.6, the abstract machine works off a stack of instructions, and so we must initially transform our AST into a stack. Additionally, we expect each while loop to have an index associated with it. We will do this we construct a stack by splitting the tree on all sequences. So C1;C2;C3; becomes $C1 \cdot C2 \cdot C3$. An important detail being that we do not decompose any other parts of the programs, if one of the statements is an if, or a while, we maintain it as such.

In order to execute the abstract machine, we also need to track five structures:

- Control stack: This is the stack containing the commands (or labels) to be executed.
- Result stack: The stack which contains values currently being worked on.
- Store: Where variables are stored.
- Back stack: Where we store the reverse computation.
- while table: Where we map while loops expressions to indices.

All of these must be tracked throughout the execution of the abstract machine. RIMPiler implements a general purpose stack to be used by the control stack, result stack, and back stack, and uses a map to map loop indices to expressions, and reuses the store provided by the interpreter for the abstract machine.

The abstract machine then divides the execution into two steps, first we must decide which rule needs to be applied, then we must apply the rule. In order to decide which rule to apply, we have a function which looks at the minimum contents of the various stacks to differentiate them, this is not always the entire contents as described in the rules. For example, in order to check if we must apply the ; rule, it is sufficient to check the top of the control stack for a ; label, and disregard any other information. This simplification relies on the assumption that the abstract machine never reaches an invalid state. This assumption will hold in our case, as we only construct abstract machines for valid RIMP programs, and perform valid transitions on these.

Once we have obtained the transition to be applied, we apply it through sequential poping and pushing to the various stacks, as well as updating the relevant variables. This too relies on

```
fn check_while(&self, i: index, E: Program) -> bool {
    let e = self.while_table.get(i);
    if let Some(expression) = e {
        return E == expression;
    } else {
        // unreachable
        return false;
    }
}
```

Figure 5.7: Simplified version of the *check_while* function in RIMPiler

the simplification assumption, as at no point do we validate what we pop from the stacks, we only check the values which we use, and all others are assumed valid. Additionally, we simplify the rules further by removing the *cloop* transition. This transition can only ever occur after the *endw* transition, and so can be incorporated within this instruction through a loop, this simplifies the rule selection process as well as the application process significantly.

When applying the $loop_T$ rule, we must also consider the while counter. For this, we need the *while_table*: This table is constructed when the abstract machine is constructed. We do this by iterating through the initial control stack, and upon finding a while loop, we insert the expression associated with it into the *while_table* using the index associated with it as its key. When we are executing the original counter, we must increment the value as this rule is applied, however, if we are reversing the loop, we should instead be decrementing it. In order to do this, when we are in this rule we must check if the loop is the original loop, or the reversed one, for this we have a function *check_loop* which is defined as follows:

Here, we first attempt to fetch e from our *while_table* using the loops index. This should return the initial expression associated with the loop. We can then compare that value to E which is the current expression within the loop. If the expressions match, then we know the loop is the initial version, and thus return *true*, otherwise it is the reverse, and so we return $false$.

Chapter 6

Legal, Social, Ethical & Professional Issues

6.1 BCS Code of Conduct

Throughout this project, I aim to ensure all relevant standards of the BSC Code of conduct are upheld, and as such below is an overview of how I have followed the standards.

1. Public Interest

   (a) have due regard for public health, privacy, security and wellbeing of others and the environment: *N/A.*

   (b) have due regard for the legitimate rights of Third Parties: *Throughout the course of the project, I have ensured to follow all copyright notices and licences on software used.*

   (c) conduct your professional activities without discrimination on the grounds of sex, sexual orientation, marital status, nationality, colour, race, ethnic origin, religion, age or disability, or of any other condition or requirement: *N/A.*

   (d) promote equal access to the benefits of IT and seek to promote the inclusion of all sectors in society wherever opportunities arise: *Not directly applicable to the development process, however, section 6.4 and 6.3 have further details.*

2. Professional Competence and Integrity

   (a) only undertake to do work or provide a service that is within your professional competence: *I have ensured to extensively research existing works to gain relevant competencies.*

   (b) NOT claim any level of competence that you do not possess: *N/A.*

   (c) develop your professional knowledge, skills and competence on a continuing basis, maintaining awareness of technological developments, procedures, and standards that are relevant to your field: *Carried out an extensive literature review covering all areas of research.*

   (d) ensure that you have the knowledge and understanding of Legislation and that you comply with such Legislation, in carrying out your professional responsibilities: *N/A no legislation outside copyright protection is in question with this project.*

   (e) respect and value alternative viewpoints and, seek, accept and offer honest criticisms of work: *N/A.*

   (f) avoid injuring others, their property, reputation, or employment by false or malicious or negligent action or inaction: *N/A.*

   (g) reject and will not make any offer of bribery or unethical inducement: *N/A*

3. Duty to Relevant Authority

   (a) carry out your professional responsibilities with due care and diligence in accordance with the Relevant Authority's requirements whilst exercising your professional judgement at all times: *N/A.*

   (b) seek to avoid any situation that may give rise to a conflict of interest between you and your Relevant Authority: *N/A.*

   (c) accept professional responsibility for your work and for the work of colleagues who are defined in a given context as working under your supervision: *N/A.*

   (d) NOT disclose or authorise to be disclosed, or use for personal gain, or to benefit a third party, confidential information except with the permission of your Relevant Authority, or as required by Legislation: *N/A.*

   (e) NOT misrepresent or withhold information on the performance of products, systems or services (unless lawfully bound by a duty of confidentiality not to disclose such information), or take advantage of the lack of relevant knowledge or inexperience of others: *N/A.*

4. Duty to the Profession

    (a) accept your personal duty to uphold the reputation of the profession and not take any action which could bring the profession into disrepute: *N/A.*

    (b) seek to improve professional standards through participation in their development, use and enforcement: *evaluated and referenced the BSC code of conduct before and after the project to ensure compliance.*

    (c) uphold the reputation and good standing of BCS, the Chartered Institute for IT: *N/A.*

    (d) act with integrity and respect in your professional relationships with all members of BCS and with members of other professions with whom you work in a professional capacity: *N/A.*

    (e) encourage and support fellow members in their professional development: *N/A.*

6.2 Licensing

RIMPiler is available in its entirety under a permissive MIT licence, and all source, and documentation will be made available as soon as possible in furtherance of research and learning in the area of reversible computing.

6.3 Accessibility

RIMPiler aims to be an evolving system in which may contributors could work together to improve the framework. To this aim, RIMPiler aims to follow where possible the concepts of literate programming[31]. This enables the software and techniques to be more accessible to more users, allowing them to contribute, adapt, and learn from the code base.

6.4 Low Power Computing

Reversible computing, while not fully present, promises to enable low-power computing through the reduction of heat loss[33]. As such, the progress and research within this area has the potential to revolutionise many aspects of computing. Two primary considerations are the sustainability of computing and its accessibility to economically disadvantaged communities. Low-power computing could allow for a significant reduction in $C0_2$ emissions[23] primarily by targetting the energy used during the lifetime of the device. Further, due to the reduction in energy consumption, we enable computer access to communities which may not have previously had as much access due to high energy costs. The reduction in costs could help allow access to computing devices to those from disadvantaged backgrounds[25, 26] to access services, education, and financial opportunities[46]. Energy also limits access to computing applications, for example, fuzzing[41] and AI[55], reducing the energy consumption of such tasks would significantly reduce the barrier to entry, enabling people to use this technology in a wider range of areas than previously possible. As such, RIMPiler is a small step in the direction of providing more equal computing.

Chapter 7

Evaluation

7.1 Testing

RIMPiler was largely tested using unit testing, covering a large proportion of the core library features. We reach an average of 80% coverage across all library areas, and 50% across interpreter implementation. This enabled several edge cases to be caught and corrected. Additionally, RIMPiler was tested using semi-manual means through compilation of the example programs across all targets. Unit testing was not used here due to it largely being insufficient to test the future evolving language. This was largely to cover the remaining blind spots of the interpreters tests, as well as to address the lack of tests for the compiler and abstract machine. Through this method of testing, we find that RIMPiler passes all expected tests, providing the correct behaviour in the expected time.

RIMPiler aims to utilise a grammar based fuzzing approach[20], which could have the potential to catch a larger variety of bugs in the implementation without requiring static test cases or manual testing by producing programs in RIMP using a grammar, and a corresponding program in some oracle language such as C using CompCert[35]. This however was not implemented due to time constraints on the implementation. Another potential approach to fuzz testing would also be through the use of LLM's which has proven valuable in identifying edge cases, or as test oracles[10, 34].

7.2 Requirements

RIMPiler has been able to achieve all required goals, as well as some additional targets, details of which are given below.

7.2.1 Functional Requirements

1. RIMPiler must produce a token stream from a valid RIMP source file.

   RIMPiler is able to produce tokens from a valid RIMP program. Further, RIMPiler is able to provide feedback on unexpected tokens including where the error originates to enable developers to correct mistakes. RIMPiler also uses an extensible system in order to allow for quick adaptation of the language tokens.

2. RIMPiler must produce a valid abstract syntax tree from a token stream produced by RIMPiler.

   RIMPiler is able to produce an AST from a token stream produced by RIMPiler or a third party source, additionally, RIMPiler is capable of providing error messages to indicate where and how a program is malformed. The parsing process, however, uses a top-down algorithm, which could in the future become a bottleneck to performance.

3. RIMPiler must be able to perform transformations on the abstract syntax tree to ensure the reversibility of the language.

   RIMPiler is able to perform all transformations required, or a subset, and is flexible in its application, allowing developer control as needed in future extensions to the framework.

4. RIMPiler must be able to produce an abstract syntax tree corresponding to the reverse of another abstract syntax tree.

   RIMPiler fully sufficiently implements the reverse function, providing a sufficient minimal implementation for reversing correct RIMPiler programs in the conventional backend, and a complete implementation in the abstract machine.

5. RIMPiler must be able to evaluate an abstract syntax tree, providing information about the value of variables until the reversal point.

   The abstract machine of RIMPiler provides fine-grained execution control and probing to the developer, allowing them to view all internal state of the abstract machine between any rule applications of the program.

6. RIMPiler must be able to be run through an abstract machine.

   The RIMPiler abstract machine is capable of running a program. However, the abstract machine can currently only be run when you have access to the RIMP source code, and has no binary form, additionally, there is currently not debug information associating each command in the abstract machine with source code, potentially complicating debugging.

7. The RIMPiler abstract machine must allow to step forwards and backwards

   RIMPiler's abstract machine completely implements the rules allowing for forward stepping and backward stepping.

8. RIMPiler must be able to compile to the JVM providing information about the value of variables until the reversal point.

   RIMPiler is able to produce binaries targetting the JVM, allowing for the compilation, and distribution of RIMPiler programs for later execution. The issue of Java's verification system is present, however, forcing the users to not verify the class file produced.

9. RIMPiler should be able to compile to an intermediate representation in SSA form.

   RIMPiler currently does not support any intermediate representation, however, is designed to be modular as to allow for this to be added in future revisions.

10. RIMPiler should be able to target a reversible backend.

    RIMPiler provides a reversible abstract machine implementation, however, not yet a fully reversible ISA such as PISA[57].

11. RIMPiler should be extended to allow for arrays.

    RIMPiler currently does not support data types other than integers and floats, but provides the facilities to extend this in the future.

12. RIMPiler should be extended to allow for user defined structures.

    RIMPiler currently does not support data types other than integers and floats, but provides the facilities to extend this in the future.

13. RIMPiler should be extended to allow for procedures.

    RIMPiler currently does not provide procedures, in order to implement these, we will need to extend the definitions of RIMPiler to these, being careful to maintain reversibility, and balance the memory consumption, especially in cases where we have recursive functions.

14. RIMPiler should be extended to allow for multi-threading.

RIMPiler provides no facilities for starting threads, or interacting with the underlying operating system.

7.2.2 Non-Functional Requirements

1. RIMPiler must compile short programs within 10 seconds on a modern desktop computer.

   On a system using an i7-1185G7 with 32 GB of RAM, RIMPiler's release build is able to compile all example programs within 0.3 seconds as measured using the time utility[37].

2. RIMPiler must give some level of feedback when things go wrong to help guide the user to the issue.

   RIMPiler provides feedback on errors in order to help a user resolve these. The effectiveness of these messages has not been evaluated yet outside the author.

3. RIMPiler must be an extensible system which is able to evolve over time.

   RIMPiler follows literate programming principles throughout, and so aims to be an approachable system to extend and develop, however, this has not been evaluated outside the author.

4. RIMPiler should be easy to learn for a developer coming from conventional languages such as C or Python.

   RIMPiler's syntax closely follows that of C's with some minor differences, as does its semantics, and as such is likely easy to learn, however, this has not been evaluated outside the author.

5. RIMPiler should be a portable system, able to run on a range of system.

   RIMPiler is written in Rust, and uses no platform specific code, as such, it can be compiled for any target Rust is available, and its compiled results can run on any system capable of running compatible Java versions.[1]

6. RIMPiler should be easy to develop on and extend.

   This requirement is not assessable due to the need of additional developers.

[1]RIMPiler has only been tested on Linux distributions with x86 processors.

7.3 Limitations

While RIMPiler provides a complete system for development, there are several areas in which it could be improved. RIMPiler has a limiting and slightly odd structure around conditionals of branching and looping structures (see 4.1), reducing the ease of use for developers. RIMPiler also is only capable of targetting the JVM for compilation, further limited by the use of the Krakatau assembler, which has limitations on what it can target, and the verification abilities of it. Aside from this, the greater limitation on RIMPiler is the limited nature and feature set of the language, missing many core features modern developers would expect, such as functions, user defined data types, I/O operations[2], and threading. While these features are not present, there are significant barriers to the complexity reasonably achievable by a RIMP program. RIMPiler also lacks sufficient supporting tooling such as a language server, or linter. While the infrastructure exists within the libraries provided by RIMPiler, no such tools have been developed yet.

[2]I/O operations are a particular issue among all existing reversible languages, as the issue of how to maintain reversibility at the I/O layer becomes more complex.

Chapter 8

Conclusion and Future Work

RIMPiler provides a cohesive and complete framework for the development and release of reversible languages. We were able to extend the language to allow for types, and practical implementations, making suitable optimisations on the theory. Unlike existing systems for reversible languages, we impose no restrictions to the developer. We have shown the development using reversible languages can potentially be as easy to use as that of a conventional language, and with extensions could be a practical target, while also allowing for the benefits reversibility has to provide, including enhanced debugging, reduced power consumption, and increased speed[33, 56].

RIMPiler could greatly benefit from future work on extending the language to allow for more common language facilities such as functions, objects, closures, threads, etc. These features would provide a more comprehensive and familiar programming experience for developers using RIMPiler, potentially increasing its practicality in real-world scenarios. Additionally, further research into optimisation techniques[12] specific to reversible languages could enhance the performance and efficiency of RIMP programs, making them more competitive with traditional irreversible languages.

Future work into targetting an SSA form such as RSSA[38], and later a reversible ISA such as PISA[57] could help seed a reversible equivalent to systems like LLVM, providing further common infrastructure to the greater development of reversible languages. Additionally, the targetting of an intermediate SSA form would allow for the implementation of an optimising compiler for RIMPiler, improving its performance[12].

Further, as seen in languages such as Hermes[40], we could aim to produce domain specific variations of RIMP to enhance the development of many tools which may benefit from reversibility such as compression algorithms.

RIMPiler provides a test bed on which easy and fast development and trials of new concepts can be applied to the reversible domain, pushing for the advancement of this promising area.

Chapter 9

Bibliography

[1] Beniamino Accattoli, Pablo Barenbaum, and Damiano Mazza. Distilling abstract machines. *ACM SIGPLAN Notices*, 49(9):363–376, 2014.

[2] V Aho Alfred, S Lam Monica, and D Ullman Jeffrey. *Compilers Principles, Techniques & Tools*. pearson Education, 2007.

[3] David F Bacon, Susan L Graham, and Oliver J Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys (CSUR)*, 26(4):345–420, 1994.

[4] Arini Balakrishnan and Chloe Schulze. Code obfuscation literature survey. *CS701 Construction of compilers*, 19:31, 2005.

[5] Charles H. Bennett and Rolf Landauer. The fundamental physical limits of computation. *Scientific American*, 253(1):48–57, 1985.

[6] Gianfranco Bilardi and Keshav Pingali. Algorithms for computing the static single assignment form. *Journal of the ACM (JACM)*, 50(3):375–425, 2003.

[7] Anne Brüggemann-Klein. Regular expressions into finite automata. *Theoretical Computer Science*, 120(2):197–213, 1993.

[8] Janusz A Brzozowski. Derivatives of regular expressions. *Journal of the ACM (JACM)*, 11(4):481–494, 1964.

[9] Alonzo Church. A set of postulates for the foundation of logic. *Annals of mathematics*, pages 346–366, 1932.

[10] Aidan Dakhama, Karine Even-Mendoza, William B Langdon, Hector Menendez, and Justyna Petke. Searchgem5: Towards reliable gem5 with search based software testing and large language models. In *International Symposium on Search Based Software Engineering*, pages 160–166. Springer, 2023.

[11] Nils Anders Danielsson. Total parser combinators. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, pages 285–296, 2010.

[12] Niklas Deworetzki, Martin Kutrib, Uwe Meyer, and Pia-Doreen Ritzke. Optimizing reversible programs. In *International Conference on Reversible Computation*, pages 224–238. Springer, 2022.

[13] Stephan Diehl, Pieter Hartel, and Peter Sestoft. Abstract machines for programming language implementation. *Future Generation Computer Systems*, 16(7):739–751, 2000.

[14] diku. Janus playground. `http://topps.diku.dk/pirc/janus-playground/`. Accessed on 2024-04-05.

[15] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 ieee symposium on security and privacy (sp)*, pages 472–489. IEEE, 2019.

[16] Maribel Fernández and Ian Mackie. A reversible operational semantics for imperative programming languages. In *Formal Methods and Software Engineering: 22nd International Conference on Formal Engineering Methods, ICFEM 2020, Singapore, Singapore, March 1–3, 2021, Proceedings 22*, pages 91–106. Springer, 2020.

[17] Richard A Frost and Rahmatullah Hafiz. A new top-down parsing algorithm to accommodate ambiguity and left recursion in polynomial time. *ACM SIGPLAN Notices*, 41(5):46–54, 2006.

[18] Robert Glück, Ivan Lanese, Claudio Antares Mezzina, Jarosław Adam Miszczak, Iain Phillips, Irek Ulidowski, and Germán Vidal. Towards a taxonomy for reversible computation approaches. In *International Conference on Reversible Computation*, pages 24–39. Springer, 2023.

[19] Robert Glück and Tetsuo Yokoyama. A linear-time self-interpreter of a reversible imperative language. *Information and Media Technologies*, 11:160–180, 2016.

[20] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation*, pages 206–215, 2008.

[21] Google. Google's v8 engine regex module. `https://github.com/v8/v8/tree/main/src/regexp`. Accessed on 2024-04-08.

[22] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 282–293, 2002.

[23] Udit Gupta, Young Geun Kim, Sylvia Lee, Jordan Tse, Hsien-Hsin S Lee, Gu-Yeon Wei, David Brooks, and Carole-Jean Wu. Chasing carbon: The elusive environmental footprint of computing. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 854–867. IEEE, 2021.

[24] Lasse Hay-Schmidt, Robert Glück, Martin Holm Cservenka, and Tue Haulund. Towards a unified language architecture for reversible object-oriented programming. In *International Conference on Reversible Computation*, pages 96–106. Springer, 2021.

[25] Donna L Hoffman, William D Kalsbeek, and Thomas P Novak. Internet and web use in the us. *Communications of the ACM*, 39(12):36–46, 1996.

[26] Donna L Hoffman and Thomas P Novak. Bridging the racial divide on the internet, 1998.

[27] IEEE. IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(TM)) Base Specifications. *IEEE Std. 1516-2000*, 2017.

[28] Steven C Johnson and Dennis M Ritchie. Unix time-sharing system: Portability of c programs and the unix system. *The Bell System Technical Journal*, 57(6):2021–2048, 1978.

[29] Gary A Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206, 1973.

[30] Donald E Knuth. On the translation of languages from left to right. *Information and control*, 8(6):607–639, 1965.

[31] Donald Ervin Knuth. Literate programming. *The computer journal*, 27(2):97–111, 1984.

[32] Martin Kutrib, Uwe Meyer, Niklas Deworetzki, and Marc Schuster. Compiling janus to rssa. In *International Conference on Reversible Computation*, pages 64–78. Springer, 2021.

[33] Rolf Landauer. Irreversibility and heat generation in the computing process. *IBM journal of research and development*, 5(3):183–191, 1961.

[34] William B Langdon, Shin Yoo, and Mark Harman. Inferring automatic test oracles. In *2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing (SBST)*, pages 5–6. IEEE, 2017.

[35] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. Compcert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016.

[36] Michael E Lesk and Eric Schmidt. *Lex: A lexical analyzer generator*, volume 39. Bell Laboratories Murray Hill, NJ, 1975.

[37] Linux. time utility man page. `https://man7.org/linux/man-pages/man1/time.1.html`. Accessed on 2024-04-10.

[38] Torben Ægidius Mogensen. Rssa: a reversible ssa form. In *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pages 203–217. Springer, 2015.

[39] Torben Ægidius Mogensen. Hermes: a language for light-weight encryption. In *Reversible Computation: 12th International Conference, RC 2020, Oslo, Norway, July 9-10, 2020, Proceedings 12*, pages 93–110. Springer, 2020.

[40] Torben Ægidius Mogensen. Hermes: a reversible language for lightweight encryption. *Science of Computer Programming*, 215:102746, 2022.

[41] Jiradet Ounjai, Valentin Wüstholz, and Maria Christakis. Green fuzzer benchmarking. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1396–1406, 2023.

[42] Luca Paolini, Mauro Piccolo, Luca Roversi, et al. A certified study of a reversible programming language. *LEIBNIZ INTERNATIONAL PROCEEDINGS IN INFORMATICS*, 69:1–21, 2018.

[43] Kalyan S Perumalla. *Introduction to reversible computing*. CRC Press, 2013.

[44] Vaughan R Pratt. Top down operator precedence. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 41–51, 1973.

[45] Python. Python re module. `https://github.com/python/cpython/blob/3.12/Lib/re/_compiler.py`. Accessed on 2024-04-08.

[46] Massimo Ragnedda and Anna Gladkova. *Digital inequalities in the Global South*. Springer, 2020.

[47] Theodore H Romer, Dennis Lee, Geoffrey M Voelker, Alec Wolman, Wayne A Wong, Jean-Loup Baer, Brian N Bershad, and Henry M Levy. The structure and performance of interpreters. *ACM SIGPLAN Notices*, 31(9):150–159, 1996.

[48] Storyyeller. Jonathan meyer, daniel reynaud. `https://jasmin.sourceforge.net/`. Accessed on 2024-04-10.

[49] Storyyeller. Krakatau. `https://github.com/Storyyeller/Krakatau`. Accessed on 2024-04-10.

[50] Martin Sulzmann and Kenny Zhuo Ming Lu. Posix regular expression parsing with derivatives. In *International Symposium on Functional and Logic Programming*, pages 203–220. Springer, 2014.

[51] Chengsong Tan and Christian Urban. Posix lexing with bitcoded derivatives. In *14th International Conference on Interactive Theorem Proving (ITP 2023)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.

[52] Michael Kirkedal Thomsen and Holger Bock Axelsen. Interpretation and programming of the reversible functional language rfun. In *Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages*, pages 1–13, 2015.

[53] Alan Mathison Turing et al. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[54] Christian Urban. Posix lexing with derivatives of regular expressions. *Journal of Automated Reasoning*, 67(3):24, 2023.

[55] Aimee Van Wynsberghe. Sustainable ai: Ai for sustainability and the sustainability of ai. *AI and Ethics*, 1(3):213–218, 2021.

[56] Germán Vidal. Reversible computations in logic programming. In *International Conference on Reversible Computation*, pages 246–254. Springer, 2020.

[57] Carlin James Vieri. *Pendulum–a reversible computer architecture.* PhD thesis, Massachusetts Institute of Technology, 1995.

[58] David Anthony Watt and Deryck F Brown. *Programming language processors in Java: compilers and interpreters.* Pearson Education, 2000.

[59] Niklaus Wirth, Niklaus Wirth, Niklaus Wirth, Suisse Informaticien, and Niklaus Wirth. *Compiler construction*, volume 1. Addison-Wesley Reading, 1996.

[60] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing ast interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages*, pages 73–82, 2012.

[61] Tetsuo Yokoyama and Robert Glück. A reversible programming language and its invertible self-interpreter. In *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 144–153, 2007.

[62] Tetsuo Yokoyama and Robert Glück. A reversible programming language and its invertible self-interpreter. In *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 144–153, 2007.

Appendix A

Extra Information

A.1 Abstract Machine

A.1.1 Details of Definition

Here we provide the definition of *lab* and *program*.

$$\begin{array}{rl} lab ::= & exp \\ \mid & unexp \\ \mid & op \\ \mid & \underline{op} \\ \mid & asgn \\ \mid & := \\ \mid & asgn^r \\ \mid & =: \\ \mid & seq \\ \mid & ; \\ \mid & cond \\ \mid & if \\ \mid & \underline{cond} \\ \mid & \underline{if} \\ \mid & loop_i \\ \mid & \underline{loop_i} \\ \mid & while_i \\ \mid & \underline{while_i} \\ \mid & \underline{endw_i} \end{array}$$

Figure A.1: Definition of the labels within the abstract machine

$$
\begin{array}{rcl}
\textit{program} ::= & & n \\
& | & \underline{n} \\
& | & !l \\
& | & \underline{!l} \\
& | & (E_1 \ \textit{oper} \ E_2) \\
& | & (\textit{oper} \ E) \\
& | & \underline{E} \\
& | & \textit{skip} \\
& | & (l \ := \ E) \\
& | & (l \ =: \ E) \\
& | & (C_1; C_2) \\
& | & (\textit{if} \ E \ \textit{then} \ C_1 \ \textit{else} \ C_2) \\
& | & (\textit{while}_i \ E \ \textit{do} \ C)
\end{array}
$$

Figure A.2: Definition of the programs in the abstract machine

A.1.2 Issue With Existing System

An issue was encountered upon implementing the abstract machine as provided. An example of this issue is seen below:

```
n := 0;


while n > 0 do {
    n := !n - 1;
};
```

Here we have an example of a RIMPiler program with a loop which we expect to never iterate. The expected behaviour of this would then to conclude with $n \mapsto 0$, and to reverse back to this state.

Below we have the forward execution as described by the original abstract machine.

$$\langle \mathit{while}_0\ !n\ >\ 0\ \mathit{do}\ \ n\ := !n-1\cdot \mathit{nil}, \mathit{nil}, m\{n\mapsto 0\}, \mathit{nil}\rangle \xrightarrow{loop}$$

$$\langle !n>0\cdot \mathit{while}_0\cdot \mathit{loop}_0\cdot \mathit{nil}, !n>0\cdot n := !n-1\cdot \mathit{nil}, m\{n\mapsto 0\}, \underline{\mathit{loop}_0}\cdot \mathit{nil}\rangle \xrightarrow{exp}$$

$$\langle !n\cdot 0\cdot > \cdot \mathit{while}_0\cdot \mathit{loop}_0\cdot \mathit{nil}, !n>0\cdot n := !n-1\cdot \mathit{nil}, m\{n\mapsto 0\}, \mathit{exp}\cdot \underline{!n}\cdot \underline{0}\cdot \underline{\mathit{loop}_0}\cdot \mathit{nil}\rangle \xrightarrow{var}$$

$$\langle 0\cdot > \cdot \mathit{while}_0\cdot \mathit{loop}_0\cdot \mathit{nil}, 0\cdot !n>0\cdot n := !n-1\cdot \mathit{nil}, m\{n\mapsto 0\}, \underline{!n}\cdot \mathit{exp}\cdot \underline{!n}\cdot \underline{0}\cdot \underline{\mathit{loop}_0}\cdot \mathit{nil}\rangle \xrightarrow{num}$$

$$\langle > \cdot \mathit{while}_0\cdot \mathit{loop}_0\cdot \mathit{nil}, 0\cdot 0\cdot !n>0\cdot n := !n-1\cdot \mathit{nil}, m\{n\mapsto 0\}, \underline{0}\cdot \underline{!n}\cdot \mathit{exp}\cdot \underline{!n}\cdot \underline{0}\cdot \underline{\mathit{loop}_0}\cdot \mathit{nil}\rangle \xrightarrow{op}$$

$$\langle \mathit{while}_0\cdot \mathit{loop}_0\cdot \mathit{nil}, \mathit{False}\ \cdot !n>0\cdot n := !n-1\cdot \mathit{nil}, m\{n\mapsto 0\}, (!n\underline{>}0)\cdot \underline{\mathit{loop}_0}\cdot \mathit{nil}\rangle \xrightarrow{loop_F}$$

$$\langle \mathit{loop}_0\cdot \mathit{nil}, !n>0\cdot n := !n-1\cdot \mathit{nil}, m\{n\mapsto 0\}, \mathit{False}\cdot \underline{\mathit{while}_0}\cdot \underline{\mathit{loop}_0}\cdot \mathit{nil}\rangle \xrightarrow{endw_F}$$

$$\langle \mathit{loop}_0\cdot \mathit{nil}, 0\cdot C\cdot !n>0\cdot n := !n-1\cdot \mathit{nil}, m\{n\mapsto 0\}, \underline{\mathit{endw}_0}\cdot \mathit{nil}\rangle \xrightarrow{endw}$$

$$\text{where } C = \mathit{while}_0\ !\mathit{counter}_0\ >\ 0\ \mathit{do}\{n =: !n-1\}$$

$$\langle \mathit{nil}, \mathit{nil}, m\{n\mapsto 0\}, (\mathit{while}_0\ !\mathit{counter}_0\ >\ 0\ \mathit{do}\{n =: !n-1\})\cdot \underline{\mathit{endw}_0}\cdot \mathit{nil}\rangle$$

At this point we finished forward computation, and the reverse is[1].

$$\langle (\mathit{while}_0\ !\mathit{counter}_0\ >\ 0\ \mathit{do}\{n =: !n-1\})\cdot \underline{\mathit{endw}_0}\cdot \mathit{nil}, \mathit{nil}, m\{n\mapsto 0\}, \mathit{nil}\rangle \xrightarrow{loop}$$

$$\langle !\mathit{counter}_0 > 0\cdot \mathit{while}_0\cdot \mathit{loop}_0\cdot \underline{\mathit{endw}_0}\cdot \mathit{nil}, !\mathit{counter}_0 > 0\cdot n =: !n-1\cdot \mathit{nil}, m\{n\mapsto 0\}, \underline{\mathit{loop}_0}\cdot \mathit{nil}\rangle \xrightarrow{expr}$$

$$\langle !\mathit{counter}_0\cdot 0\cdot > \cdot \mathit{while}_0\cdot \mathit{loop}_0\cdot \underline{\mathit{endw}_0}\cdot \mathit{nil}, !\mathit{counter}_0 > 0\cdot n =: !n-1\cdot \mathit{nil}, m\{n\mapsto 0\},$$
$$\mathit{exp}\cdot \underline{!\mathit{counter}_0}\cdot \underline{0}\cdot \underline{\mathit{loop}_0}\cdot \mathit{nil}\rangle \xrightarrow{var}$$

$$\langle 0\cdot > \cdot \mathit{while}_0\cdot \mathit{loop}_0\cdot \underline{\mathit{endw}_0}\cdot \mathit{nil}, 0\cdot !\mathit{counter}_0 > 0\cdot n =: !n-1\cdot \mathit{nil}, m\{n\mapsto 0\},$$
$$\underline{!\mathit{counter}_0}\cdot \mathit{exp}\cdot \underline{!\mathit{counter}_0}\cdot \underline{0}\cdot \underline{\mathit{loop}_0}\cdot \mathit{nil}\rangle \xrightarrow{num}$$

$$\langle > \cdot \mathit{while}_0\cdot \mathit{loop}_0\cdot \underline{\mathit{endw}_0}\cdot \mathit{nil}, 0\cdot 0\cdot !\mathit{counter}_0 > 0\cdot n =: !n-1\cdot \mathit{nil},$$
$$m\{n\mapsto 0\}, \underline{0}\cdot \underline{!\mathit{counter}_0}\cdot \mathit{exp}\cdot \underline{!\mathit{counter}_0}\cdot \underline{0}\cdot \underline{\mathit{loop}_0}\cdot \mathit{nil}\rangle \xrightarrow{op}$$

$$\langle \mathit{while}_0\cdot \mathit{loop}_0\cdot \underline{\mathit{endw}_0}\cdot \mathit{nil}, \mathit{False}\cdot !\mathit{counter}_0 > 0\cdot n =: !n-1\cdot \mathit{nil}, m\{n\mapsto 0\},$$
$$(!\mathit{counter}_0\underline{>}0)\cdot \underline{\mathit{loop}_0}\cdot \mathit{nil}\rangle \xrightarrow{loop_F}$$

$$\langle \mathit{loop}_0\cdot \underline{\mathit{endw}_0}\cdot \mathit{nil}, !\mathit{counter}_0 > 0\cdot n =: !n-1\cdot \mathit{nil}, m\{n\mapsto 0\},$$
$$\mathit{False}\cdot \underline{\mathit{while}_0}\cdot \underline{\mathit{loop}_0}\cdot \mathit{nil}\rangle \xrightarrow{endw_F}$$

$$\langle \mathit{loop}_0\cdot \underline{\mathit{endw}_0}\cdot \mathit{nil}, 0\cdot C\cdot !\mathit{counter}_0 > 0\cdot n =: !n-1\cdot \mathit{nil}, m\{n\mapsto 0\}, \underline{\mathit{endw}_0}\cdot \mathit{nil}\rangle \xrightarrow{endw}$$

$$\text{where } C = \mathit{while}_0\ !n\ >\ 0\ \mathit{do}\{n := !n-1\}$$

$$\langle \underline{\mathit{endw}_0}\cdot \mathit{nil}, \mathit{nil}, m\{n\mapsto 0\}, (C = \mathit{while}_0\ !n\ >\ 0\ \mathit{do}\{n := !n-1\})\cdot \underline{\mathit{endw}_0}\cdot \mathit{nil}\rangle$$

At this point we can only apply either wend_F or wend_T as we have an endw_0 at the head of the control stack, but neither can be applied, as both require contents in the result stack, while we have *nil*, showing the definition provided cannot work in this case.

[1] $\mathit{counter}_0$ will be initialised to 0 in memory but is left from m for brevity

A.1.3 Proof of Correctness

In order to prove the correctness of our implementation of the abstract machine, we must show it follows the results produced by the big-step semantics. To do this there are some important things to note, the big-step semantics to not distinguish between the next operation to produce and the result of an operation, while the abstract machine places results on the result stack, and the next operation on the control stack, so when showing correspondence we must keep this in mind. Additionally, there are many more states possible in the abstract machine, however, only a subset of them are reachable from valid RIMP programs. So to show correspondence, we will show that for all valid starting configurations of the abstract machine, (including those for expressions) we reach the same results expected by the big-step semantics.

$$\text{const}\,\frac{}{\langle c, s\rangle \Downarrow \langle c, s\rangle \text{ if } \Gamma \vdash c : \text{Numeric}}$$

$$\langle c \cdot c, r, m, b\rangle \overset{num}{\rightarrow} \langle c, c \cdot r, m, \underline{c} \cdot b\rangle$$

$$\text{var}\,\frac{}{\langle !l, s\rangle \Downarrow \langle n, s\rangle \text{ if } l \in s \wedge s(l) = n}$$

$$\langle !l \cdot c, r, m, b\rangle \overset{var}{\rightarrow} \langle c, m(l) \cdot r, m, \underline{!l} \cdot b\rangle$$

$$\text{numeric}\,\frac{\langle E_1, s\rangle \Downarrow \langle n_1, s\rangle \quad \langle E_2, s\rangle \Downarrow \langle n_2, s\rangle}{\langle E_1 \; op \; E_2, s\rangle \Downarrow \langle n, s\rangle \text{ if } op \in \{+, -, /, *, \hat{}\} \wedge \Gamma \vdash n_1, n_2 : \text{Numeric} \wedge n = n_1 \overline{op} \; n_2}$$

$$\text{logical}\,\frac{\langle E_1, s\rangle \Downarrow \langle b_1, s\rangle \quad \langle E_2, s\rangle \Downarrow \langle b_2, s\rangle}{\langle E_1 \; op \; E_2, s\rangle \Downarrow \langle b, s\rangle \text{ if } op \in \{\&\&, ||\} \wedge \Gamma \vdash n_1, n_2 : \text{Boolean} \wedge b = b_1 \overline{op} \; b_2}$$

$$\text{relation}\,\frac{\langle E_1, s\rangle \Downarrow \langle n_1, s\rangle \quad \langle E_2, s\rangle \Downarrow \langle n_2, s\rangle}{\langle E_1 \; op \; E_2, s\rangle \Downarrow \langle b, s\rangle \text{ if } op \in \{>, <, ==, !=\} \wedge \Gamma \vdash n_1, n_2 : \text{Numeric} \wedge b = n_1 \overline{op} \; n_2}$$

$$\begin{aligned}
\langle (E_1 \; op \; E_2) \cdot c, r, m, b\rangle &\overset{binexp}{\rightarrow} \langle E_1 \cdot E_2 \cdot oper \cdot c, r, m, binexp \cdot \underline{E_1} \cdot \underline{E_2} \cdot b\rangle \\
&\rightarrow^* \langle oper \cdot c, n_2 \cdot n_1 \cdot r, m, underlineE_2 \cdot \underline{E_1} \cdot binexp \cdot \underline{E_1} \cdot \underline{E_2} \cdot b\rangle \\
&\overset{binop}{\rightarrow} \langle c, n \cdot r, m, (E_1 \; op \; E_2) \cdot b\rangle \\
&\text{Where } n \; = \; n_1 \; \overline{op} \; n_2
\end{aligned}$$

$$\text{unary boolean}\,\frac{\langle E_1, s\rangle \Downarrow \langle b_1, s\rangle}{\langle op \; E_1, s\rangle \Downarrow \langle b, s\rangle \text{ if } op \in \{\neg\} \wedge \Gamma \vdash b_1 : \text{Boolean} \wedge b = \overline{op} \; b_1}$$

$$\text{unary numeric}\,\frac{\langle E_1, s\rangle \Downarrow \langle n_1, s\rangle}{\langle op \; E_1, s\rangle \Downarrow \langle n, s\rangle \text{ if } op \in \{-\} \wedge \Gamma \vdash n_1 : \text{Numeric} \wedge n = \overline{op} \; n_1}$$

$$\begin{aligned}
\langle (op \; E) \cdot c, r, m, b\rangle &\overset{unexp}{\rightarrow} \langle E \cdot oper \cdot c, r, m, unexp \cdot \underline{E} \cdot b\rangle \\
&\rightarrow^* \langle oper \cdot c, n_1 \cdot r, m, underlineE \cdot unexp \cdot \underline{E} \cdot b\rangle \\
&\overset{unop}{\rightarrow} \langle c, n \cdot r, m, (op \; E) \cdot b\rangle \\
&\text{Where } n \; = \; \overline{op} \; n_1
\end{aligned}$$

Figure A.3: Proofs of correspondence between expressions

From figure A.3 using *const* and *var* as base cases, we can show all expressions correspond to big-step semantics of RIMP. One thing missing here is an explicit proof of the reverse, however, this follows through the application of the as all inverse expressions are just wrapped versions of the original, we can achieve this through one of the unwrapping rules provided with the conditional and loop structures.

$$\text{skip}\,\frac{}{\langle skip, s\rangle \Downarrow \langle skip, s\rangle}$$

$$\langle skip \cdot c, r, m, b\rangle \overset{skip}{\rightarrow} \langle c, r, m, skip \cdot b\rangle$$

$$:=\,\frac{\langle E, s\rangle \Downarrow \langle n, s\rangle}{\langle l := E, s\rangle \Downarrow \langle skip, s[l \rightarrow (n, +(n_1, s(l)))]\rangle \text{if } l \in s \wedge \Gamma \vdash n : \text{Numeric} \wedge n_1 = n - s(l)}$$

$$\begin{aligned}\langle (l\ :=\ E) \cdot c, r, m, b\rangle &\overset{asgn}{\rightarrow} \langle E \cdot !l \cdot := \cdot c, l \cdot r, m, asgn \cdot E \cdot b\rangle\\ &\rightarrow^{*} \langle !l \cdot := \cdot c, n \cdot l \cdot r, m, \underline{E} \cdot asgn \cdot E \cdot b\rangle\\ &\overset{var}{\rightarrow} \langle := \cdot c, m(l) \cdot n \cdot l \cdot r, m, \underline{!l} \cdot \underline{E} \cdot asgn \cdot E \cdot b\rangle\\ &\overset{:=}{\rightarrow} \langle c, r, m[l \mapsto (n, +(n - m(l), m(l)))], (l =: E) \cdot b\rangle\end{aligned}$$

$$=:\,\frac{\langle E, s[l \rightarrow (n - n_1, v)]\rangle \Downarrow \langle n, s[l \rightarrow (n - n_1, v)]\rangle}{\langle l =: E, s[l \rightarrow (n, +(n_1, v)]\rangle \Downarrow \langle skip, s[l \rightarrow (n - n_1, v)]\rangle \text{if } l \in s_1 \wedge \Gamma \vdash n : \text{Numeric}}$$

$$\begin{aligned}\langle (l\ =:\ E) \cdot c, r, m, b\rangle &\overset{asgn^r}{\rightarrow} \langle E \cdot !l \cdot =: \cdot c, l \cdot r, m[l \mapsto (n_1 - n, v)], asgn^r \cdot n \cdot E \cdot b\rangle\\ &\rightarrow^{*} \langle !l \cdot =: \cdot c, n \cdot l \cdot r, m[l \mapsto (n_1 - n, v)], \underline{E} \cdot asgn^r \cdot n \cdot E \cdot b\rangle\\ &\overset{var}{\rightarrow} \langle =: \cdot c, m(l) \cdot n \cdot l \cdot r, m[l \mapsto (n_1 - n, v)], \underline{!l} \cdot \underline{E} \cdot asgn^r \cdot n \cdot E \cdot b\rangle\\ &\overset{=:}{\rightarrow} \langle c, r, m, (l := E) \cdot b\rangle\end{aligned}$$

$$\text{seq}\,\frac{\langle C_1, s\rangle \Downarrow \langle skip, s'\rangle \quad \langle C_2, s'\rangle \Downarrow \langle skip, s''\rangle}{\langle C_1; C_2, s\rangle \Downarrow \langle skip, s''\rangle}$$

$$\begin{aligned}\langle (C_1; C_2) \cdot c, r, m, b\rangle &\overset{seq}{\rightarrow} \langle C_1 \cdot C_2 \cdot ; \cdot c, r, m, seq \cdot b\rangle\\ &\rightarrow^{*} \langle ; \cdot c, r, m, rev(C_2) \cdot rev(C_1) \cdot seq \cdot b\rangle\\ &\overset{;}{\rightarrow} \langle c, r, m, (rev(C_2); rev(C_1)) \cdot b\rangle\end{aligned}$$

Figure A.4: Proofs of correspondence between statements

Similar to before, we use *skip*, :=, and =: as base cases, as well as all expressions, and the *seq* as the recursive step.

$$\text{if true}\frac{\langle E,s\rangle \Downarrow \langle true,s\rangle \quad \langle C_1,s\rangle \Downarrow \langle skip,s'\rangle}{\langle \text{if } E \text{ then } C_1 \text{ else } c_2, s\rangle \Downarrow \langle skip, s'\rangle \text{if } \Gamma \vdash E : \text{Boolean}}$$

$$\begin{aligned}
\langle (if\ E\ then\ C_1\ else\ C_2)\cdot c, r, m, b\rangle &\overset{cond}{\rightarrow} \langle E\cdot if\cdot cond\cdot c, C_1\cdot C_2\cdot r, m, \underline{cond}\cdot b\rangle\\
&\rightarrow^* \langle if\cdot cond\cdot c, true\cdot C_1\cdot C_2\cdot r, m, \underline{E}\cdot \underline{cond}\cdot b\rangle\\
&\overset{if_T}{\rightarrow} \langle C_1\cdot cond\cdot c, C_1\cdot C_2\cdot r, m, E\cdot \underline{if}\cdot \underline{cond}\cdot b\rangle\\
&\rightarrow^* \langle cond\cdot c, C_1\cdot C_2\cdot r, m, \underline{C_1}\cdot E\cdot \underline{if}\cdot \underline{cond}\cdot b\rangle\\
&\overset{endif}{\rightarrow} \langle c, r, m, (if\ E\ then\ rev(C_1)\ else\ rev(C_2))\cdot b\rangle
\end{aligned}$$

$$\text{if false}\frac{\langle E,s\rangle \Downarrow \langle false,s\rangle \quad \langle C_2,s\rangle \Downarrow \langle skip,s'\rangle}{\langle \text{if } E \text{ then } C_1 \text{ else } c_2, s\rangle \Downarrow \langle skip, s'\rangle \text{if } \Gamma \vdash E : \text{Boolean}}$$

$$\begin{aligned}
\langle (if\ E\ then\ C_1\ else\ C_2)\cdot c, r, m, b\rangle &\overset{cond}{\rightarrow} \langle E\cdot if\cdot cond\cdot c, C_1\cdot C_2\cdot r, m, \underline{cond}\cdot b\rangle\\
&\rightarrow^* \langle if\cdot cond\cdot c, false\cdot C_1\cdot C_2\cdot r, m, \underline{E}\cdot \underline{cond}\cdot b\rangle\\
&\overset{if_F}{\rightarrow} \langle C_2\cdot cond\cdot c, C_1\cdot C_2\cdot r, m, E\cdot \underline{if}\cdot \underline{cond}\cdot b\rangle\\
&\rightarrow^* \langle cond\cdot c, C_1\cdot C_2\cdot r, m, \underline{C_2}\cdot E\cdot \underline{if}\cdot \underline{cond}\cdot b\rangle\\
&\overset{endif}{\rightarrow} \langle c, r, m, (if\ E\ then\ rev(C_1)\ else\ rev(C_2))\cdot b\rangle
\end{aligned}$$

Figure A.5: Proofs of correspondence between conditionals

$$\text{while true}\frac{\langle E,s\rangle \Downarrow \langle true,s\rangle \quad \langle C,s\rangle \Downarrow \langle skip,s'\rangle \quad \langle \text{while } E \text{ do } C, s'\rangle \Downarrow \langle skip,s''\rangle}{\langle \text{while } E \text{ do } C, s\rangle \Downarrow \langle skip, s''\rangle \text{if } \Gamma \vdash E : \text{Boolean}}$$

$$\begin{aligned}
\langle (while_i\ E\ do\ C)\cdot c, r, m, b\rangle &\overset{loop}{\rightarrow} \langle E\cdot while_i\cdot loop_i\cdot c, E\cdot C\cdot r, m, \underline{loop_i}\cdot b\rangle\\
&\rightarrow^* \langle while_i\cdot loop_i\cdot c, true\cdot E\cdot C\cdot r, m, \underline{E}\cdot \underline{loop_i}\cdot b\rangle\\
&\overset{loop_T}{\rightarrow} \langle C\cdot (while_i\ E\ do\ C)\cdot c, E\cdot C\cdot r, m, \underline{true}\cdot \underline{while_i}\cdot \underline{loop_i}\cdot b\rangle\\
&\rightarrow *\langle (while_i\ E\ do\ C)\cdot c, E\cdot C\cdot r, m, \underline{C}\cdot \underline{true}\cdot \underline{while_i}\cdot \underline{loop_i}\cdot b\rangle\\
&\rightarrow *\langle c, r, m, (while_i\ E\ do\ rev(C))\cdot b\rangle
\end{aligned}$$

$$\text{while false}\frac{\langle E,s\rangle \Downarrow \langle false,s\rangle}{\langle \text{while } E \text{ do } C, s\rangle \Downarrow \langle skip, s\rangle \text{if } \Gamma \vdash E : \text{Boolean}}$$

$$\begin{aligned}
\langle (while_i\ E\ do\ C)\cdot c, r, m, b\rangle &\overset{loop}{\rightarrow} \langle E\cdot while_i\cdot loop_i\cdot c, E\cdot C\cdot r, m, \underline{loop_i}\cdot b\rangle\\
&\rightarrow^* \langle while_i\cdot loop_i\cdot c, false\cdot E\cdot C\cdot r, m, \underline{E}\cdot \underline{loop_i}\cdot b\rangle\\
&\rightarrow^* \langle c, r, m, (while_i\ E\ do\ rev(C))\cdot b\rangle
\end{aligned}$$

Figure A.6: Proofs of correspondence between loops

With conditionals and loops, we treat all commands and expressions as the base cases.

Note that the proofs given here are informal and do not provide an entirely complete proof, skipping some steps for simplicity. particularly in the case of the while, as the complete proof is very long and cumborsome.

Appendix B

User Guide

B.1 Installation

In order to install RIMPiler you must first obtain a copy of the source. This will include a git folder. The project relies on an external utility called Krakatau, this can be added using git:

```
git submodule update --init --recursive
```

This will pull the project into your local files[1]. In the event you forget this, you will receive an error indicating the build is failing in a future step.

In order to build the project, you then need to install cargo from Rust. This can be found at `https://doc.rust-lang.org/cargo/getting-started/installation.html`, or, on Linux systems you can do the following:

```
curl https://sh.rustup.rs -sSf | sh
```

If you have issues here, please refer to the Rust documentation for help.

You can now, from the root of the source project build RIMPiler using cargo:

```
cargo build --release
```

This will create a release binary in ./target/release/RIMPiler, and also produce a binary for Krakatau in ./target/release/krak2. You can now move both of these files wherever you would like on your system, including adding them to your PATH to access them across your system. If you add them to the PATH please ensure you add both files, RIMPiler has fail safes to try to recover from a missing Krakatau binary, however, this may not work in all cases,

[1]You may need to set up git on your computer for this to work.

especially in locations such as /urs/bin, and so the easiest way to ensure Krakatau is available to RIMPiler in this case is to add it to your PATH.

B.2 RIMP Programs

RIMP's syntax is provided, however, for ease, some explanations, and examples are provided here.

All statements in RIMPiler must be followed by a semicolon, this includes conditionals and loops:

```
int n = 7;
int m = 0;

while n > 0 do {
        n = n - 1;
        m = m + n;
};

if m > 11 then {
    m = m * 2;
} else {
    m = m - 1;
};
```

If statements also require an else branch, if this is not needed you can simply put skip here instead.

```
if n == 1 then {
    skip;
} else {
    skip;
};
```

variable declarations can only occur once, and must be preceded by their type, either `int` or `float`:

```
int x = 0;
float y = 1.2;

x = y * 4;
y = 4.5;
```

It is also suggested to avoid doing any arithmetic in the conditions of loops and conditionals, this is due to the complicated nature of the grammar around this point, and can produce some errors which may be confusing to new users, for this reason, it is suggested to keep to just binary operations here.

For more examples, check the ./examples folder of the project.

B.3 Running RIMPiler

To run RIMPiler you simply call the executable RIMPiler. If you run this without any arguments, you will be greeted with some help information.

To pass an input file to RIMPiler you can use the −i <input> flag. In the event you are compiling, you will use the −c flag to indicate this, you will also then need to provide an output folder, this can be done with −o <output>. If you wish to use the interpreter, you can provide the −r flag. To use the abstract machine, you use the −m flag.

B.3.1 Using the Abstract Machine

If you are running the abstract machine, you have many commands available to you. If at any point you wish to remind yourself of them, while in the abstract machine you can type h or help to bring up a list of available commands.

Some of the most useful commands are as follows:

- r: This will run the entire program through to completion in the same direction it is currently in.
- s: This will perform a single step through the program.
- rv: This will reverse the direction of the program.
- d: This will display the current direction.
- pa: This will print the contents of all stacks and the store.

There are many more options provided for ease of use, please refer to the help menu for more details.

B.3.2 Running Compiled Programs

Once compiled, you should have a folder with the name you provided in the output option. Within there should be a Main.class, this is the entry point.

Due to limitations in the assembler, there are some considerations when running the compiled Java programs.

Please ensure you use the following Java configuration if you have issues:

```
java version "17.0.10" 2024-01-16 LTS
Java(TM) SE Runtime Environment (build 17.0.10+11-LTS-240)
Java HotSpot(TM) 64-Bit Server VM (build 17.0.10+11-LTS-240, mixed mode, sharing)
```

It is likely that many other versions before Java version 18 will work, however this has not been verified.

To run your compiled program, you use:

```
java -noverify -cp <output> Main
```

The −noverify flag is not always necessary, but programs are not guaranteed to run without it.