# RIMPiler

6CCS3PRJ Background and Specification Progress Report

*Author:* Aidan Dakhama

*Student ID:* 21034945

*Supervisor:* Professor Maribel Fernandez

December 2023

Reversible programming languages do something...

# Introduction

Reversible computing is an area of computer science concerned with the ability to perform computations in reverse. This is useful in a number of applications, such as debugging, and quantum computing. Currently, the availability of real world tools such as compilers and interpreters are sparse and limited to few languages. This project aims to implement a complete compiler for the reversible programming language RIMP.

## Background and Context

### Reversible Computing

Reversible computing is a model of computing where all operations are reversible, and therefore, we can always recover the previous state of the system. This has a number of applications, such as debugging, quantum computing, and overcoming the Landauer limit.

The Landauer limit is a theoretical limit on the amount of energy required to perform a computation, and is proportional to the number of bits erased during the computation. Overcoming this limit would allow for much more energy efficient computers, and while we are still far from achieving this, reversible computing is a necessary step towards it.

Quantum computing is a model of computing which uses quantum mechanical phenomena such as superposition and entanglement to perform computations, allowing for much more efficient algorithms for certain problems such as Shor's algorithm for factoring large numbers.

The ability to perform computations in reverse is also useful for debugging, as it allows us to step backwards through the execution of a program, and see how the state of the system changes. this allows us to easily navigate the control flow of a program, and see how the state of the system changes with various data changes.

### Programming Languages

Programming languages are an abstraction over some model of computation, allowing developers to write programs in a more human-readable, and human-reasonable way. A programming language is often comprised of both a specification, formally describing the syntax and semantics of the language, and a compiler or interpreter, which allows us to work with the language without having to perform manual translations to a lower level language. Additionally, a programming language may allow for abstractions over the underlying model of computation, or specific architecture, allowing the same program to target different systems which may work in drastically different ways.

In the case of reversible computing and RIMP, the semantics of the language allow us to embed reversibility in a syntax which otherwise resembles a standard imperative programming language, allowing the developer to program in a way which is familiar to them, while still being able to take advantage of the benefits of reversible computing.

### Lexers

Syntax is the set of rules which define the structure of a language, namely, what set of strings are valid strings, without necessarily defining the meaning of those strings. A lexer is a program which takes a stream of characters as input, and outputs a stream of tokens based on some syntax. This is useful for compilers and interpreters, as it allows us to not only break down the problem into smaller steps, but it can also act as a filter, removing any unwanted parts of the input, such as whitespace and comments, as well as catching any potential errors such as a string sequence which cannot correspond to any valid token.

We often aim to produce POSIX lexers, as they allow us to disambiguate between tokens in a way that is easy to reason about.

There are many ways to implement a lexer, including finite state machines, and regular expressions. In the case of regular expressions, we can further explore methods of implementing them, such as using the standard NFA construction such as through Thompson's construction, using a DFA construction such as the powerset construction, or by using derivatives.

### Regular Expressions

Regular expressions are a way of describing a set of strings, and are often used in lexers to describe the syntax of a language. One way of implementing regular expressions is through derivatives, where we can take the derivative of a regular expression with respect to a character, and use this to determine whether a string is in the language described by the regular expression. This can be extended to allow for POSIX lexing.

### Parsers

Parsers take a stream of tokens as input, and output a parse tree, which represents the structure of the program in an unambiguous way. This tree then encodes the semantics of the program, and we can then perform further analysis on the tree, or perform various transformations on it. There are many ways to implement a parser, some methods being top down, such as recursive descent, and bottom up, such as LR parsing. Here we will use a mix of a custom top-down parser to handle statements and control flow, and Pratt parsing to handle expressions.

Pratt parsing is a method of parsing expressions, where we assign a precedence to each token, and then use this to determine how to parse the expression. It is a form of top-down operator precedence parsing, and is often used in conjunction with recursive descent parsing.

### Single Static Assignment

Single static assignment is a form of intermediate representation, where each variable is assigned exactly once, and each assignment is given a unique name. This allows for many optimisations, such as constant propagation, and dead code elimination to easily be performed on the program.

It is also useful for portability, as LLVM IR is in SSA form, and so we can easily target LLVM, and therefore many architectures, by compiling to LLVM IR.

# Literature Review

There are several other reversible programming languages, such as Janus, and R-While, and ROOP. These languages aim at different paradigms, such as ROOP being object-oriented. However, many of these existing languages require the programmer to be aware of the underlying reversible model of computation, which can add overhead to the development process. RIMP, unlike these languages, aims to mirror the syntax of a standard imperative programming language, allowing the programmer to write programs in a way which is familiar to them, while still being able to take advantage of the benefits of reversible computing through semantic transformations.

Janus maintains reversibility by restricting the use of assignments to only be non-destructive, by this, we can always recover the previous state of the system. For example if we have:

x += 1
x −= 4

We can recover the previous state by performing the inverse operations in reverse order:

x += 4
x −= 1

This is often how reversible languages deal with assignments, however, in many conventional languages, programmers are use to using destructive assignments, which can make it difficult to write programs in a reversible language. RIMP allows for both destructive and non-destructive assignments, and uses semantic transformations to ensure that the program is reversible. We achieve this by using a stack to preserve the changes made by destructive assignments, and then using this stack to undo the changes when we need to recover the previous state of the system.

ROOP is an expressive language which allows for rich object-oriented features and typing. It is similar to Janus in that it restricts the use of assignments.

# Requirements and Specification

This project aims to produce a compiler for the reversible programming language RIMP in a modular way, allowing for the use of different backends, and frontends, and to easily alter various aspects of the compilers pipeline.

## Requirements

### Functional Requirements

1. The compiler must be able to produce a token stream from a RIMP source file.
2. The compiler must be able to produce an abstract syntax tree from a token stream.
3. The compiler must be able to perform semantic transformations on the abstract syntax tree to produce a reversible program.
4. The compiler must be able to produce the reverse abstract syntax tree from an abstract syntax tree.
5. The compiler must be able to produce an SSA form from an abstract syntax tree.
6. The compiler must be able to produce an executable file from an input file.
7. The compiler should be able to target a reversible SSA form such as RSSA.
8. The compiler should provide a backend for a reversible SSA form such as RSSA to produce a reversble machine code such as PISA.
9. The compiler could extend the language to include arrays
10. The compiler could extend the language to include functions
11. The compiler could extend the language to include I/O
12. The compiler could extend the language to perform optimisations

### Non-Functional Requirements

1. The compiler should be able to compile in a reasonable amount of time.
2. The compiler should be able to give feedback to the user when compilation fails.
3. The produced code should execute in a reasonable amount of time.

# Specification

Here we will detail the requirements of the project, and prioritise them. In order to have a complete compiler we must implement all high priority requirements. We will also list other requirements which we may implement if time allows, and additionally some which we will not be able to implement.

## Functional Requirements

### 1 produce a token stream from a RIMP source file: High Priority

Given a plaintext file containing a RIMP program, the compiler should be able to read its contents, and produce a token stream corresponding to the program. This token stream should match the concrete syntax of RIMP, and should be sound and complete with respect to the RIMP grammar. This means that no string should produce a token if it is not a valid token in RIMP, and every string which corresponds to a valid token in RIMP should produce a token.

### 2 produce an abstract syntax tree from a token stream: High Priority

Given a token stream, composed only of valid tokens, the compiler should be able to produce an abstract syntax tree corresponding to the program. This abstract syntax tree should match the abstract syntax of RIMP, and should be sound and complete with respect to the RIMP grammar. This means that no token stream should produce an abstract syntax tree if it is not a valid program in RIMP, and every program in RIMP should produce an abstract syntax tree.

### 3 perform semantic transformations: High Priority

Given an abstract syntax tree, the compiler should be able to perform semantic transformations on the tree to produce a reversible program. These transformations should be applied to if statements and while loops. If statements should be transformed such that any variables used in the conditional, which are also reassigned in the body of the if statement, remapped to new variables. We will use a function $Vars_{assign}(B)$ to denote the set of variables assigned to in a block $B$. We will use a function $Vars_{used}(E)$ to denote the set of variables used in an expression $E$. Given an if statement comprising a conditional $C$, and an if body $B_{if}$, and an else body $B_{else}$.

$$vars = \{\forall v \in (Vars_{used}(C) \cup Vars_{assign}(B_{if}) \cup Vars_{assign}(B_{else}))\}$$

for all $vars$ we will introduce a new variable $v'$, which before the if statement is assigned to $v$. We will then replace all occurrences of $v$ in $C$ with $v'$.

While loops should be transformed such that each loop maintains a counter variable which is incremented at the end of each iteration.

### 4 produce the reverse abstract syntax tree: High Priority

Given a correct abstract syntax tree, the compiler should be able to produce the reverse abstract syntax tree. This reverse abstract syntax tree should be equivalent to performing the $rev$ function to the original program, and then producing the abstract syntax tree of the resulting program.

### 5 produce an SSA form: High Priority

Given a correct abstract syntax tree, the compiler should be able to produce an SSA form. This SSA form should at least be LLVM IR, however, we may also implement a reversible SSA

form such as RSSA if time allows. The SSA produced should be semantically equivalent to the original program.

#### 6 produce an executable file: High Priority

The compiler must produce a file which at a later time can be executed to run the program. This may be an executable file able to run directly on $x86_64$, $PISA$, $aJavaclassfile$, $orafilewhichmayrunonanabstra$

#### 7 target a reversible SSA form: Medium Priority

The compiler should be able to target a reversible SSA form such as RSSA. This will allow us to target reversible architectures such as PISA, and therefore allow us to run the program on a reversible machine.

#### 8 provide a backend for a reversible SSA form: Medium Priority

The compiler should provide a backend for a reversible SSA form such as RSSA to produce a reversble machine code such as PISA. The PISA produced should be semantically equivalent to the original program.

#### 9 extend the language to include arrays: Low Priority

The compiler could extend the language to include arrays. Arrays must also preserve reversibility. This feature is out of scope for the project given the time constraints, however, it would be a useful feature to have in the future.

#### 10 extend the language to include functions: Low Priority

The compiler could extend the language to include functions. Functions must also preserve reversibility by being able to uncall them. This feature is out of scope for the project given the time constraints, however, it would be a useful feature to have in the future.

#### 11 extend the language to include I/O: Low Priority

The compiler could extend the language to include I/O. This would allow us to have points in programs where optimisations cannot be applied, which would allow us to implement optimisations without potentially reducing the entire program to a trivial set of instructions. This feature is out of scope for the project given the time constraints, however, it would be a useful feature to have in the future.

#### 12 extend the language to perform optimisations: Low Priority

The compiler could extend the language to perform optimisations. This would only be possible if we also extend the language to include I/O, as otherwise we would likely optimise much of the program away as there are no blocks which cannot be optimised. This feature is out of scope for the project given the time constraints, however, it would be a useful feature to have in the future.

### Non-Functional Requirements

#### 1 compile in a reasonable amount of time: High Priority

The compiler should not take longer than a few seconds to compile a simple RIMP program, comprising less than 50 lines.

**2 give feedback to the user when compilation fails: High Priority**

The compiler should give feedback to the user when compilation fails, and should give a useful error message. This message should ideally indicate the location, and indicate some possible causes of the error.

**3 execute in a reasonable amount of time: medium Priority**

The produced code should execute in a reasonable amount of time. This means we should ensure the resulting compiled program does not have significant amounts of unnecessary overhead.

## Limitations

One of the largest limitations to this project outside of time constraints is the lack of I/O provided by RIMP. This means that we cannot implement optimisation without reducing the program significantly. If we were to do this, we would essentially be performing much of the computation at compile time, and then only outputting the result at runtime.

Another limitation is the lack of hardware and emulation support for reversible architectures. This prevents us from easily targeting reversible architectures such as PISA and running the program on a reversible machine due to lack of availability, and documentation making it difficult to implement the compiler for these architectures.

# Design

Identify use cases, with diagrams, give the system architecture, "as low level as possible". Use UML diagrams, data flow diagrams, sequence diagrams, etc.