# RIMPiler

6CCS3PRJ Background and Specification Progress Report

*Author:* Aidan Dakhama

*Student ID:* 21034945

*Supervisor:* Professor Maribel Fernandez

December 2023

# Introduction

Reversible computing is an area of computer science concerned with the ability to perform computations in reverse[3, 25, 27]. This is useful in a number of applications, such as debugging, and quantum computing[10, 27, 33]. 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[11].

## 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[10, 21, 27].

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.

Compiler designers 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[14], using a DFA construction such as the powerset construction, or by using derivatives[6].

## 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[32].

## 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[20]. Here we will use a mix of a custom top-down parser to handle statements and control flow, and Pratt parsing to handle expressions[26].

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[23, 29].

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

## Compilers

Compilers are an area of very active research, and therefore, there are many different approaches to implementing a compiler. However, typically there are shared components, such as lexing, parsing, and code generation[19].

### Lexing

Lexing is often the first step in a compiler[28], and is the process of taking a stream of characters as input, and outputting a stream of tokens. There are various methods of implementing a lexer, one of the most common being regular expressions. Much research has been done on implementing systems to lex using regular expressions, such as often through the use of NFAs and Thompson's construction, or through derivatives. One of the challenges of lexing using regular expressions is implementing a disambiguation strategy, such as POSIX lexing, which allows us to disambiguate between tokens in a way that is easy to reason about. In the case of NFAs, this can be done using the Disambiguation strategy proposed by Okui et al.[24] through emulating the subset construction. In the case of derivatives, this can be done using the Methods proposed by Sulzmann et al.[30], which uses an injection function to map the matched string to the corresponding regex which matched it, this implicitly also encodes a bias to match the longest leftmost string.

### Parsing

Parsing allows us to construct an unambiguous parse tree from a stream of tokens or characters[19]. We often want to parse based on a context-free grammar such as Backus-Naur Form (BNF), this allows us to avoid problems of ambiguity and left-recursion.

There are two main classes of parsers, top-down parsers, and bottom-up parsers. LR parsers are a class of bottom-up parsers[20]. Recursive descent parsers are a class of top-down parsers[12, 13].

Recursive descent parsers are often used in conjunction with Pratt parsing[26], which 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. This method allows us to easily handle both operator precedence and associativity, however, one of the limitations of pratt parsing is its lack of support for parsing statements and control flow. This is why we will use a mix of a custom top-down parser to handle statements and control flow, and Pratt parsing to handle expressions.

### Code Generation

Code generation is the process of taking an intermediate representation of a program such as an abstract syntax tree, and producing a program in a target language. In modern compilers this is often done in several stages, where we will first target an intermediate representation in SSA form such as LLVM IR. This allows us to apply various optimisations to the program, such as constant propagation, and dead code elimination[8, 9, 34]. Additionally, this allows us to reuse the infrastructure of a common backend, allowing us to target many different architectures by targeting a single common intermediate representation.

There are several ways to go from an abstract syntax tree to an SSA form, such as through the use of a control flow graph, or using CPS conversion[4, 5].

## Reversible Languages

There are several other general purpose reversible programming languages, such as Janus[7, 36], R-While[15], ROOP[16, 18], and RFun[31, 35]. Additionally, there are some Domain Specific Languages such as Hermes[22]. There are also some intermediate representations and machine code formats such as RSSA[23] and PISA[2].

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[11], 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. 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.

### Janus

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.

### Hermes

Hermes is a reversible language intended for cryptography, allowing for the implementation of an encryption algorithm to simply be reversed in order to implement the decryption algorithm.

### PISA

PISA is a reversible machine code designed to be reversible at any point. While hardware is not readily available for PISA, there are emulators available such as PendVM[1, 17].

## Reversible Computing

Optimisations are a major part of compilers, and there are many different optimisations which can be performed. This is often done on an SSA form of the program, as this allows us to easily perform many optimisations such as constant propagation, and dead code elimination[8, 34]. However, some care must be taken to ensure that these optimisations do not break the reversibility of the program[9].

# 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, a Java class file, or a file which may run on an abstract machine. We will aim to produce several of these, however, we will prioritise producing at least one.

**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

Here is an overview of the system architecture, including various components which will be implemented, and some which may not be implemented.

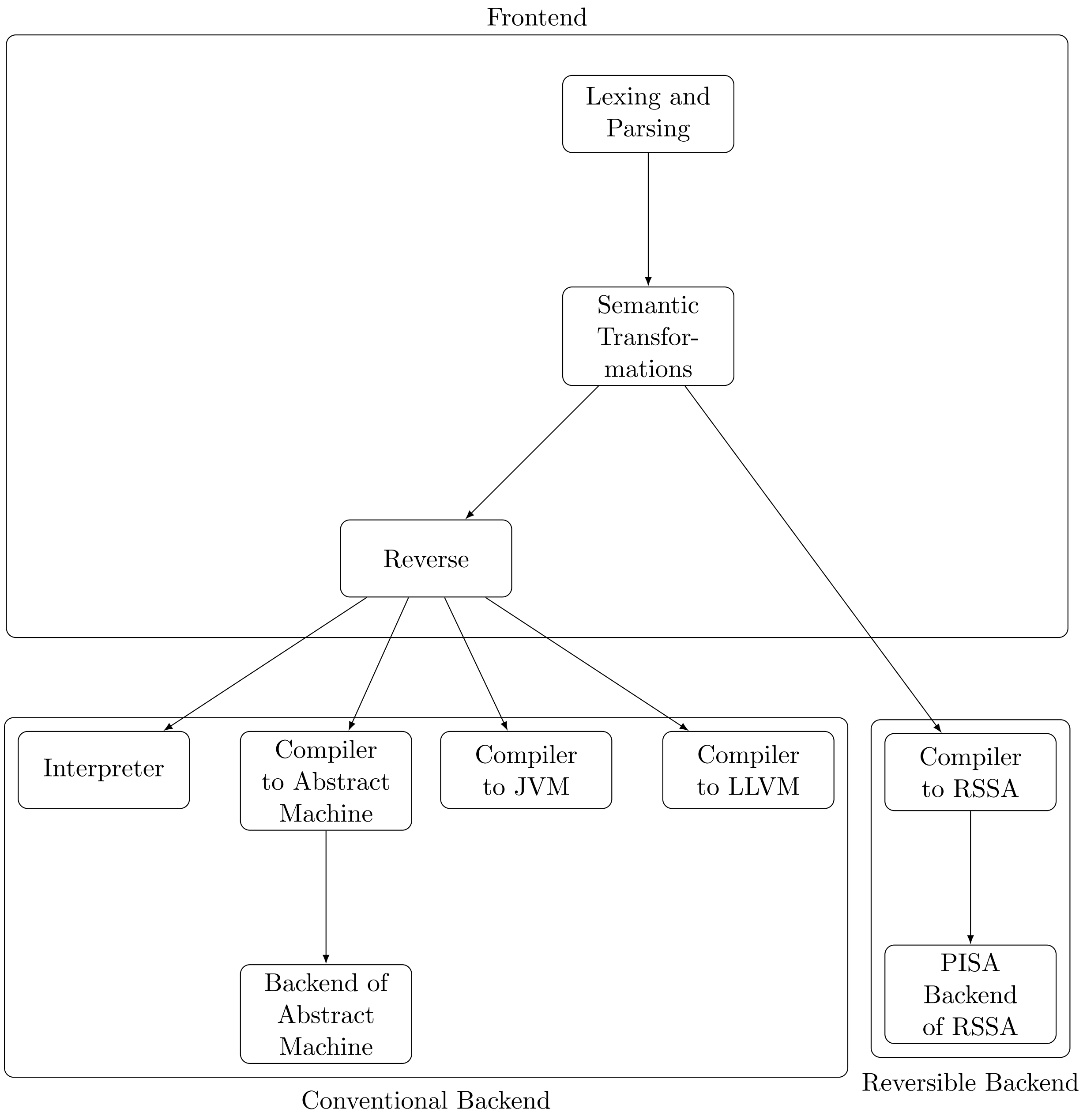

Figure 1: High level overfiew of the system architecture

Here we have the planned layout of the various components of the compiler. I aim for this design to be extensible, and to easily allow for the swapping of various components. I do not aim to implement all of these for this project, however, the architecture should allow for these additional components to be added in the future.

## Grammar

First we will define the formal grammar of RIMP in Backus-Naur Form (BNF).

```
<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> ^ <ArithmeticTerm>
    | <ArithmeticFactor>

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

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

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

<BooleanFactor> ::= (<BooleanExpression>)

<type> ::= int
```

Where a number is defined as a sequence of digits without leading zeros. An identifier is defined as an upper or lower case number, followed by a sequence of upper or lower case letters, and numbers.

## Use Cases

## Frontend

### Lexing

There are two common approaches to lexing, the first is to use a lexer generator, this is a tool which takes a description of the patterns to match, often as regular expressions, and then produces a lexer. The benefits of this is that it is often fast to set up, and well documented. However, the downside is that it is not under the control of the language developer, and may not always be well optimised for your use case. Given the relatively small size of RIMP, this is why we will use the second approach, which is to implement a lexer manually.

Again, manual lexers are still often implemented using regular expressions, and this is how we will implement our lexer. The benefit of regular expressions is that they can be easily read and understood. For example, we defined identifiers as "an upper or lower case number, followed by a sequence of upper or lower case letters, and numbers." The regular expression for this is `[a-zA-Z][a-zA-Z0-9]*`. Another benefit of regular expressions is that they are easy to modify, unlike if we were to write an NFA by hand for example, changes can be made locally without having to worry about the rest of the lexer.

We can now either use prebuilt regular expression libraries, which provide infrastructure for lexing, such as the ability to match a regular expression, and then return the matched string, or we can implement our own. Similar to the lexer generators, the benefit of using a prebuilt library is that it is often fast to implement, and well documented. However, popular regular expression engines also often have a downside, where they suffer from catastrophic backtracking, which can lead to exponential time complexity. Additionally, many implementations are not always sound and complete with respect to the language of the regular expression. As we aim for a sound and complete lexer, as well as relatively fast compilation times, we will then implement our own regular expression engine.

For this implementation, we will utilise the method presented by Sulzmann et al.[30], which uses derivatives and an injection function to map the matched string to the corresponding regex which matched it, this method is both efficient, POSIX compliant, and has a proof of correctness[32].

More details about derivatives can be found in Brzozowski's paper on derivative based regular expressions[6].

We define a language as a set of strings, where a string $s$ is a finite or infinite sequence of characters from some alphabet $\Sigma$. We will also define the operation of string concatenation denoted by @, where $s_1@s_2$ is the concatenation of strings $s_1$ and $s_2$. Similarly, we can extend this to the concatenation of languages, where $L_1@L_2 = \{s_1@s_2 | s_1 \in L_1, s_2 \in L_2\}$. Finally, we define the power of a language $L^n$ as the concatenation of $n$ copies of $L$, where $L^0 = \{\epsilon\}$, and $\forall n \geq 0\ L^n = L^{n-1}@L$.

We then define a regular expression $r$ over some alphabet $\Sigma$ which accepts a language $L(r)$. Regular expressions are then defined recursively as follows:

- 0, where $L(0) = \emptyset$
- 1, where $L(1) = \{\epsilon\}$
- $a$, where $a \in \Sigma$ and $L(a) = \{a\}$

- $[c|c-c]$ where we can define a set of characters and character ranges, where $L([c|c-c]) = \{c | c \in \Sigma \land c \in [c|c-c]\}$
- $r_1 + r_2$, where $L(r_1 + r_2) = L(r_1) \cup L(r_2)$
- $r_1 \cdot r_2$, where $L(r_1 \cdot r_2) = L(r_1)@L(r_2)$
- $r^*$, where $L(r^*) = \bigcup_{i=0}^{\infty} L(r)^i$
- $r^+$, where $L(r^+) = \bigcup_{i=1}^{\infty} L(r)^i$
- $r?$, where $L(r?) = L(r) \cup \{\epsilon\}$
- $s : r$ which allows us to name a regular expression $r$ as $s$

We will define a function *nullable* which takes a regular expression $r$ and returns true if $\epsilon \in L(r)$, and false otherwise. We then have the following:

- $\mathit{nullable}(0) = \mathit{false}$
- $\mathit{nullable}(1) = \mathit{true}$
- $\mathit{nullable}(a) = \mathit{false}$
- $\mathit{nullable}([c|c-c]) = \mathit{false}$
- $\mathit{nullable}(r_1 + r_2) = \mathit{nullable}(r_1) \lor \mathit{nullable}(r_2)$
- $\mathit{nullable}(r_1 \cdot r_2) = \mathit{nullable}(r_1) \land \mathit{nullable}(r_2)$
- $\mathit{nullable}(r^*) = \mathit{true}$
- $\mathit{nullable}(r^+) = \mathit{nullable}(r)$
- $\mathit{nullable}(r?) = \mathit{true}$
- $\mathit{nullable}(s : r) = \mathit{nullable}(r)$

Parallel to this, we will also have the concept of Values, which indicate how a regular expression matched a string. Each value will then correspond to a regular expression which has matched some string. We will define the set of values $V$ as follows:

- 1, which corresponds to the regular expression 1
- $a$, where $a \in \Sigma$ and corresponds to the regular expression $a$ as well as $[a|a-a]$
- $\mathit{Left}(v)$ where $v$ is another value, this corresponds to matching the left hand side of an alternative in a regular expression, such as $r_1 + r_2$ and $v$ corresponds to $r_1$
- $\mathit{Right}(v)$ where $v$ is another value, this corresponds to matching the right hand side of an alternative in a regular expression, such as $r_1 + r_2$ and $v$ corresponds to $r_2$
- $v_1 \cdot v_2$ where $v_1$ and $v_2$ are other values, this corresponds to matching the concatenation of two regular expressions, such as $r_1 \cdot r_2$ and $v_1$ corresponds to $r_1$ and $v_2$ corresponds to $r_2$
- $\mathit{Stars}[vs]$ where $vs$ is a list of values, this corresponds to matching the Kleene star of a regular expression, such as $r^*$ and each $v \in vs$ corresponds to each instance of $r$ matched. Similarly, this also corresponds to $r^+$ and $r?$.

- $s : v$ where $s$ is a string and $v$ is another value, this corresponds to matching a named regular expression, such as $s : r$ and $v$ corresponds to $r$

We will now define the derivative of a language $L$ with respect to a character $c$ as $D_c(L) = \{s | cs \in L\}$. Using this we can define the derivative of a regular expression $r$ with respect to a character $c$ as follows:

- $D_c(0) = 0$
- $D_c(1) = 0$
- $D_c(a) = \begin{cases} 1 & \text{if } a = c \\ 0 & \text{otherwise} \end{cases}$
- $D_c([c'|c' - c'']) = \begin{cases} 1 & \text{if } c' \leq c \leq c'' \\ 0 & \text{otherwise} \end{cases}$
- $D_c(r_1 + r_2) = D_c(r_1) + D_c(r_2)$
- $D_c(r_1 \cdot r_2) = \begin{cases} (D_c(r_1) \cdot r_2) + D_c(r_2) & \text{if } \mathit{nullable}(r_1) = \mathit{true} \\ D_c(r_1) \cdot r_2 & \text{otherwise} \end{cases}$
- $D_c(r^*) = D_c(r) \cdot r^*$
- $D_c(r^+) = D_c(r) \cdot r^*$
- $D_c(r?) = D_c(r)$
- $D_c(s : r) = D_c(r)$

With this we can already match strings by applying the derivative until we reach the end of the string, and then checking if the resulting regular expression is nullable, if it is, then the string matched, otherwise it did not. But this does not allow us to see what part of the regular expression matched the string. For this we also need to define two further functions, *mkeps* and *injection*.

We define *mkeps* as a function which takes a regular expression $r$ where $\mathit{nullable}(r) = \mathit{true}$, and returns a value $v$ which corresponds to $r$. We define *mkeps* as follows:

- $\mathit{mkeps}(1) = 1$
- $\mathit{mkeps}(r_1 + r_2) = \begin{cases} \mathit{Left}(\mathit{mkeps}(r_1)) & \text{if } \mathit{nullable}(r_1) = \mathit{true} \\ \mathit{Right}(\mathit{mkeps}(r_2)) & \text{otherwise} \end{cases}$
- $\mathit{mkeps}(r_1 \cdot r_2) = \mathit{mkeps}(r_1) \cdot \mathit{mkeps}(r_2)$
- $\mathit{mkeps}(r^*) = \mathit{Stars}[]$
- $\mathit{mkeps}(r^+) = \mathit{Stars}[\mathit{mkeps}(r)]$
- $\mathit{mkeps}(r?) = \mathit{Stars}[]$
- $\mathit{mkeps}(s : r) = s : \mathit{mkeps}(r)$

This implicitly encodes a POSIX matching strategy where we prefer the leftmost match, and the longest match.

finally, we define *injection* as a function which takes a regular expression $r$, a value $v$, and a character $c$ and returns a value $v'$ which corresponds to how the character $c$ matched the regular expression $r$. We define *injection* as follows:

- $injection(r^*, v1 \cdot Stars([vs]), c) = Stars([injection(r, v1, c)] :: vs)$
- $injection(r1 \cdot r2, Left(v1 \cdot v2), c) = injection(r1, v1, c) \cdot v2$
- $injection(r1 \cdot r2, v1 \cdot v2, c) = injection(r1, v1, c) \cdot v2$
- $injection(r1 \cdot r2, Right(v), c) = mkeps(r1) \cdot injection(r2, v, c)$
- $injection(r1 + r2, Left(v), c) = Left(injection(r1, v, c))$
- $injection(r1 + r2, Right(v), c) = Right(injection(r2, v, c))$
- $injection(c, 1, c) = c$
- $injection([c|c - c], 1, c) = c$
- $injection(r^+, v1 \cdot Stars([vs]), c) = Stars([injection(r, v1, c)] :: vs)$
- $injection(r^?, v, c) = \begin{cases} Stars([]) & \text{if } v == 1 \\ Stars([v]) & \text{otherwise} \end{cases}$
- $injection(s : r, v, c) = s : injection(r, v, c)$

We then have the following overall structure:

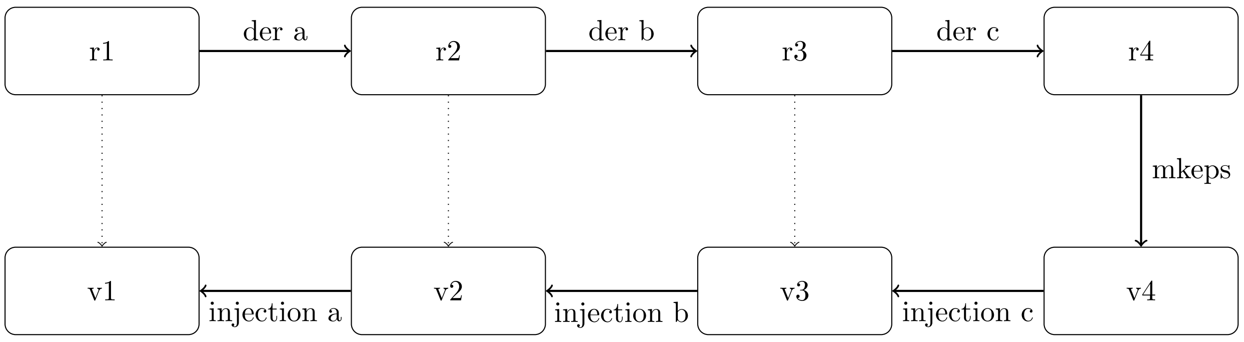

Figure 2: Lexing with derivatives

We take the derivative of the regular expression using the string we are matching until we reach a nullable regular expression, we then travel backwards building values in order to find what part of the regular expression matched the string. Like this we are able to create a regular expression for each token, such as identifiers, and check exactly which identifier matched the string.

## Parsing

There are many approaches to parsing. Here we will use a mix of Pratt parsing, and a custom top down parsing. The reason for this choice is for the ease of use and speed of these methods. Pratt parsing allows for easy changes to be made to the parsing of expressions, such as changes in precedence, but it cannot handle statements and conditionals. For this, we introduce an additional custom top-down algorithm where, we can check the next token, and infer what the rest we should expect is. This choice allows for us to quickly develop a parser which is still very efficient.

### Semantic Transformations

Semantic transformations will be used to take the abstract syntax tree generated from parsing, and it will traverse the tree to perform the transformations. Separating it out allows for us to isolate it from the rest of the system, making it agnostic to how we parsed, or how we will continue. It also allows for a much simpler method than if we were to perform these transformations on the fly while generating the abstract syntax tree.

We will traverse the tree, and when we find an incidence of an if or while statement, we will replace it with the transformed version

### Reverse Function

The reverse function allows us to take a program and compute its semantic inverse. We will implement this as a function which takes an abstract syntax tree, and returns a new abstract syntax tree which is the inverse of the input. The reverse function we are modelling has been defined as to ensure that $rev(rev(x)) = x$, while this could be done here, it should not be needed, and so we will not necessarily implement all needed for this.

## Conventional Backend

### Interpreter

In order to implement the interpreter, we will need to first generate an AST, perform the semantic transformations, and then append the reverse of the AST to the end of the original AST. We can then preform a walk of the tree recursively applying big-step semantics to each node.

Additionally, we will insert operations to output the state once the forward execution has finished, allowing the result of the computation to be seen. Similarly, we will also do this at the end for debugging purposes, however, this should always result in all variables being set to their initial values (0).

### Abstract Machine Compiler

In order to generate the abstract machine code, we can do a post-order traversal of the AST, and generate the corresponding abstract machine code for each node. This will follow the structure of the abstract machine defined for RIMP[11], which uses a stack, allowing us to compile with almost a one to one mapping between the AST and the abstract machine code.

For this we will not need to generate the reverse of the program using the reverse function, as this will be handled by the abstract machine, allowing us to reverse at any point in the execution.

### Abstract Machine Backend

In order to execute generate abstract machine code, we will need to implement an abstract machine. This will consist of:

- A control stack, where we will load the program generated into.
- A results stack, where we store intermediate results.
- A store, which will map variables to their runtime values.
- A backstack in order to reverse the program.

In addition to this the abstract machine will need to be able to evaluate the arithmetic and boolean operations.

## JVM Compiler

In order to generate JVM code, we can do a post-order traversal of the AST, and generate the corresponding JVM code for each node. One consideration is the implementation of the stack for each variable, we have two options here, we can either use an array, which would require us to know the maximum number of changes made to a variable ahead of time, or we can use a list structure, which would allow us to dynamically grow and shrink the stack. In order to ensure the language is usable, we will use a list structure, however, this will come at the cost of compiler complexity, as we will have to interact with Java classes to utilise the list structure.

# Bibliography

[1] Holger Axelsen, Robert Glück, and Tetsuo Yokoyama. "Reversible Machine Code and Its Abstract Processor Architecture". In: Sept. 2007, pp. 56–69. ISBN: 978-3-540-74509-9. DOI: `10.1007/978-3-540-74510-5_9`.

[2] Holger Bock Axelsen, Robert Glück, and Tetsuo Yokoyama. "Reversible Machine Code and Its Abstract Processor Architecture". In: *Computer Science – Theory and Applications*. Ed. by Volker Diekert, Mikhail V. Volkov, and Andrei Voronkov. Vol. 4649. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 56–69. ISBN: 978-3-540-74509-9 978-3-540-74510-5. DOI: `10.1007/978-3-540-74510-5_9`. (Visited on 12/01/2023).

[3] C. H. Bennett. "Logical Reversibility of Computation". In: *IBM Journal of Research and Development* 17.6 (1973), pp. 525–532. DOI: `10.1147/rd.176.0525`.

[4] Gianfranco Bilardi and Keshav Pingali. "Algorithms for Computing the Static Single Assignment Form". In: *Journal of the ACM* 50.3 (May 2003), pp. 375–425. ISSN: 0004-5411. DOI: `10.1145/765568.765573`. (Visited on 12/10/2023).

[5] Matthias Braun et al. "Simple and Efficient Construction of Static Single Assignment Form". In: *Compiler Construction*. Ed. by Ranjit Jhala and Koen De Bosschere. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 102–122. ISBN: 978-3-642-37051-9. DOI: `10.1007/978-3-642-37051-9_6`.

[6] Janusz A Brzozowski. "Derivatives of regular expressions". In: *Journal of the ACM (JACM)* 11.4 (1964), pp. 481–494.

[7] Lutz Christopher. "JANUS: A Time-Reversible Language". Letter. Apr. 1986. (Visited on 12/01/2023).

[8] Cliff Click and Keith D Cooper. "Combining analyses, combining optimizations". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 17.2 (1995), pp. 181–196.

[9] Niklas Deworetzki et al. "Optimizing Reversible Programs". In: *Reversible Computation*. Ed. by Claudio Antares Mezzina and Krzysztof Podlaski. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 224–238. ISBN: 978-3-031-09005-9. DOI: `10.1007/978-3-031-09005-9_16`.

[10] Jakob Engblom. "A review of reverse debugging". In: *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference*. IEEE. 2012, pp. 1–6.

[11] Maribel Fernández and Ian Mackie. "A Reversible Operational Semantics for Imperative Programming Languages". In: Dec. 2020, pp. 91–106. ISBN: 978-3-030-63405-6. DOI: `10.1007/978-3-030-63406-3_6`.

[12] Richard Frost, Rahmatullah Hafiz, and Paul Callaghan. "Modular and efficient top-down parsing for ambiguous left-recursive grammars". In: *Proceedings of the Tenth International Conference on Parsing Technologies*. 2007, pp. 109–120.

[13] Richard A Frost, Rahmatullah Hafiz, and Paul Callaghan. "Parser combinators for ambiguous left-recursive grammars". In: *Practical Aspects of Declarative Languages: 10th International Symposium, PADL 2008, San Francisco, CA, USA, January 7-8, 2008. Proceedings 10.* Springer. 2008, pp. 167–181.

[14] Dora Giammarresi. *Gluskov and Thompson Constructions : A Synthesis.* 1998.

[15] Robert Glück, Robin Kaarsgaard, and Tetsuo Yokoyama. "From Reversible Programming Languages to Reversible Metalanguages". In: *Theoretical Computer Science* 920 (June 2022), pp. 46–63. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2022.02.024. (Visited on 12/10/2023).

[16] Tue Haulund. *Design and Implementation of a Reversible Object-Oriented Programming Language.* July 2017. DOI: 10.48550/arXiv.1707.07845. arXiv: 1707.07845 [cs]. (Visited on 12/01/2023).

[17] Tue Haulund. *PendVM.* 2018.

[18] Lasse Hay-Schmidt et al. "Towards a Unified Language Architecture for Reversible Object-Oriented Programming". In: *Reversible Computation.* Ed. by Shigeru Yamashita and Tetsuo Yokoyama. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 96–106. ISBN: 978-3-030-79837-6. DOI: 10.1007/978-3-030-79837-6_6.

[19] Grace Murray Hopper. "The Education of a Computer". In: *Proceedings of the 1952 ACM National Meeting (Pittsburgh).* ACM '52. Pittsburgh, Pennsylvania: Association for Computing Machinery, 1952, pp. 243–249. ISBN: 9781450373623. DOI: 10.1145/609784.609818. URL: https://doi.org/10.1145/609784.609818.

[20] Donald E. Knuth. "On the translation of languages from left to right". In: *Information and Control* 8.6 (1965), pp. 607–639. ISSN: 0019-9958. DOI: https://doi.org/10.1016/S0019-9958(65)90426-2.

[21] R. Landauer. "Irreversibility and Heat Generation in the Computing Process". In: *IBM Journal of Research and Development* 5.3 (1961), pp. 183–191. DOI: 10.1147/rd.53.0183.

[22] Torben Ægidius Mogensen. "Hermes: A Reversible Language for Lightweight Encryption". In: *Science of Computer Programming* 215 (Mar. 2022), p. 102746. ISSN: 0167-6423. DOI: 10.1016/j.scico.2021.102746. (Visited on 12/10/2023).

[23] Torben Ægidius Mogensen. "RSSA: A Reversible SSA Form". In: *Perspectives of System Informatics.* Ed. by Manuel Mazzara and Andrei Voronkov. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 203–217. ISBN: 978-3-319-41579-6. DOI: 10.1007/978-3-319-41579-6_16.

[24] Satoshi Okui and Taro Suzuki. "Disambiguation in regular expression matching via position automata with augmented transitions". In: *Implementation and Application of Automata: 15th International Conference, CIAA 2010, Winnipeg, MB, Canada, August 12-15, 2010. Revised Selected Papers 15.* Springer. 2011, pp. 231–240.

[25] Kalyan S. Perumalla. *Introduction to Reversible Computing.* CRC Press, Sept. 2013. ISBN: 978-1-4398-7340-3.

[26] Vaughan R. Pratt. "Top down Operator Precedence". In: *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages - POPL '73.* Boston, Massachusetts: ACM Press, 1973, pp. 41–51. DOI: 10.1145/512927.512931. (Visited on 12/10/2023).

[27] Feynman Richard P. *Feynman Lectures on Computation.* CRC Press, 2018.

[28] Dennis M Ritchie. "The M4 Macro Processor Brian W. Kernighan". In: (1977).

[29] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. “Global Value Numbers and Redundant Computations”. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL ’88. New York, NY, USA: Association for Computing Machinery, Jan. 1988, pp. 12–27. ISBN: 978-0-89791-252-5. DOI: `10.1145/73560.73562`. (Visited on 12/10/2023).

[30] Martin Sulzmann and Kenny Zhuo Ming Lu. “POSIX Regular Expression Parsing with Derivatives”. In: June 2014. ISBN: 978-3-319-07150-3. DOI: `10.1007/978-3-319-07151-0_13`.

[31] 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.* IFL ’15. New York, NY, USA: Association for Computing Machinery, Sept. 2015, pp. 1–13. ISBN: 978-1-4503-4273-5. DOI: `10.1145/2897336.2897345`. (Visited on 12/10/2023).

[32] Christian Urban. “POSIX Lexing with Derivatives of Regular Expressions”. In: *Journal of Automated Reasoning* 67.3 (July 2023), p. 24. ISSN: 1573-0670. DOI: `10.1007/s10817-023-09667-1`. (Visited on 12/10/2023).

[33] Germán Vidal. “Reversible Computations in Logic Programming”. In: *Reversible Computation.* Ed. by Ivan Lanese and Mariusz Rawski. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 246–254. ISBN: 978-3-030-52482-1. DOI: `10.1007/978-3-030-52482-1_15`.

[34] Mark N Wegman and F Kenneth Zadeck. “Constant propagation with conditional branches”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13.2 (1991), pp. 181–210.

[35] Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. “Towards a reversible functional language”. In: *International Workshop on Reversible Computation.* Springer. 2011, pp. 14–29.

[36] 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.* 2007, pp. 144–153.