SUSSEX UNIVERSITY

Automated Translation of an Irreversible to Reversible Programming Language

Author:
Barnaby KLEINMAN

Supervisor:
Dr. Ian MACKIE

A report submitted as part requirement for the degree of BSc Computer Science

in the

Department of Informatics

May 2020

Declaration

This report is submitted as part requirement for the degree of Computer Science at the University of Sussex. It is the product of my own labour except where indicated in the text. The report may be freely copied and distributed provided the source is acknowledged.

Signed: [illegible]

Date: 26/05/2020

Acknowledgements

I would like to express my appreciation to my project supervisor Dr. Ian Mackie for his continued support throughout the development of this project, offering invaluable advice and guiding me through finding a suitable implementation.

Abstract

Reversible programming is a paradigm that describes the ability for a program to be executed both forwards and backwards. This can be put into practice for many reasons, including energy-optimal computing and debugging. This project outlines the creation of a new reversible language, a compiler which translates a source language into this reversible language, and an emulator to execute the output reversible programs. The main areas covered in this report are:

- An introduction to the concept of reversible computation.
- Background information surrounding reversible programming languages and compilers.
- How the language, compiler and emulator were implemented.
- How the language is used in practice.
- An evaluation of the project as a whole.

Contents

List of Figures

Chapter 1

Introduction

1.1 Reversible Programming Languages

In the last fifty years, programming languages have evolved and been improved on, with languages such as C and Java taking the lead. These examples, alongside all other major languages contain sets of instructions which allow the programmer to modify data in certain ways. Traditional languages often contain a mix of what is known as 'destructive' and 'non-destructive' operations. When a destructive operation is performed, there will be a certain amount of information which is lost or destroyed at that computational step. This information loss is what the introduction of reversible computation aims to remove.

Typically, programming languages are deterministic when executed forwards, meaning a given operation will always produce the same output when run on the same data. However, the same is not true in reverse. Due to information being destroyed, it is impossible to deterministically go from the result of a destructive operation, back a step to before it was performed. Reversible programming languages are languages in which their entire instruction set consists of operations which are non-destructive. This means that because there is no information destroyed at each step, it is possible to deterministically go both forwards and backwards in execution.

In 1961, Rolf Landauer [1] discovered that in order to erase data, energy is needed. This led to a lot of research into reversible computation and non-destructive operations. Resetting the value of a bit to 0 (called bit erasures) is the only operation which fundamentally consumes energy in computation [2]. Reversible programming languages allow computation to take place without any bit erasures, and as such energy-optimal computation is one main application of these languages.

1.2 Examples of Reversible Programming

Generally, programming languages contain a mix of both reversible and irreversible operations. The simplest example of an irreversible arithmetic operator is addition. An example of this would be that if you take the operation 3+5, once evaluated by the computer, it is obvious that the result is 8. Trying to work backwards however, the computer is incapable of knowing which two numbers were added together to make 8 - it could have been 4 and 4, 2 and 6, or any other combination.

Conversely, these languages also contain operations which can be considered reversible. The simplest reversible arithmetic operator is the increment. An example of this would be that if you take the operation 5+=3, once evaluated by the computer, it is obvious that the result is 8. Working backwards, it is clear that the reversal of the increment operator would be the decrement, so to reverse this statement it is as simple as doing 8-=3, to arrive back at the original number of 5.

In order for a language to be reversible it must ensure that not only the individual statements can be executed backwards, but also that programming constructs such as conditionals and loops have a way to be reversed. For example, an if-statement will begin with a condition and depending on the result of this condition, a specific branch of code will be selected and executed. Attempting to run this conventional construct backwards leads into immediate issues. The split of control flow which was introduced on the forwards execution, introduces ambiguity when trying to run it backwards – the computer does not know which branch needs to be executed. It is not as simple as testing the starting condition on the reverse execution as if any of the variables involved in the condition were modified within the branch, the condition may not produce the same result.

For example:

```
if x > 5 then
    x = 0
else
    x = x * 10
end
```

If this very simple conditional is run using the initial value of x=3, on the forwards execution the 'else' branch will be selected, and x will be multiplied by 10 to make 30. Attempting to run this in reverse, it is first apparent that it is unknown to the computer which branch should be run. If the condition was tested to try and choose a branch, due to x now being more than 5, the incorrect branch would be executed. One solution to making conditionals reversible, is to add a post condition to the construct. After the branch, this post-condition is required to evaluate to being equal to the pre-condition before the branch. This ensures that the post- condition can be relied on to instruct the computer in which branch should be selected. This, alongside other approached will be discussed in further detail in chapter 4.

1.3 Applications of Reversible Programming Languages

Reversible computation has many possible areas of application. Following, is a list of a few of these areas:

- **Energy-Optimal Computing** - Rolf Landauer discovered that the theoretical lower bound of energy use for computation is directly linked to the reversibility of the computation [1]. Following this, it was shown by Charles Bennett [3], [4] that in an ideal situation to perform any given computation, no energy need be lost. Thus, energy-optimal computing is tightly linked to reversible computation.

- **Debugging** - Reversible computing is very useful in terms of debugging programs efficiently. When running a program if an unexpected result occurs, the program can be executed backwards step-by-step until it is discovered where exactly the undesired result occurred, and this can be used to overcome the issue.

- **Fault Tolerance** - Fault tolerance is the ability to continue execution in a system despite runtime failures. Fault tolerance revolves around the ability to return to a previous point in execution when it is informed of a fault. This is a fundamental feature of reversible execution, and as such reversible programming can be used as an efficient method of fault tolerance.

- **Quantum Computing** - Quantum computing put simplistically is a series of unitary operations of the computer state [5]. Unitary matrices are by definition reversible, and as such the entire series of operations is reversible by nature.

While in the scope of this project, the application area that will be focused is debugging. The emulator built for this project will allow the user to step forwards and backwards in execution to find the point in which an error occurred in their program.

1.4 Project Aims and Objectives

This project aims to introduce a new reversible programming language for which a defined source language will be compiled into. This reversible language needs to support a set of exclusively reversible programming constructs. In order to achieve this, the project had two primary objectives which had to be met:

- A compiler which generates reversible code from an initial irreversible source-code.
- An emulator which mimics a reversible machine in order to execute reversible programs on.

The project also had two secondary objectives to be completed, given there was sufficient time for them to be implemented:

- Modify the compiler such that it has the ability to output the source-code of reversed input program. This output program can then be executed to perform the reversed computation
- Optimise the compiler, allowing for faster execution of the output code. This could include for example the fusion of loop operations which intend on iterating the same number of times.

1.5 Target Users

The primary demographic for which my project is aimed at is researchers and students who have an interest in learning about, and using a reversible language. This project has the ability to display the strengths and weaknesses of reversible languages, and can act as a good introduction to those who are just learning.

Alternatively, this project could be used by a programmer as a means to debug code simplistically with the ability to execute their code both forwards and backwards. The limiting factor with this second user would be that the language in its current state has a limited set of instructions, making it only suitable for simpler algorithms. With considerable extensions made to the operation set and optimisation of the language, it could be considered a suitable option for lightweight implementations of programs in industry.

Chapter 2

Professional and Ethical Considerations

This project does not involve any human participants, and as such the ethical considerations that need to be made are limited. However, the BCS Code of Conduct has been read and understood, and the following considerations have been made:

1.a. You shall have due regard for public health, privacy, security and wellbeing of others and the environment.

The program will not store information of any kind regarding the person using it, how they are using it or on what system they are using it on. This allows for complete privacy and no potential leaks of any sensitive user information.

2.b. You shall not claim any level of competence that you do not possess

The author has been studying Computer Science and has covered a variety of modules of which allows for a suitable level of competence to complete what has been proposed in this report.

The remaining points in the BCS Code of Conduct do not apply to this project due to the lack of human participants, and as such this project can be considered to conform to the code.

Chapter 3

Requirements Analysis

A list of both functional and non-functional requirements was constructed before the design stage of this project. The functional requirements specify what the software must do, whereas the non-functional requirements describe the general characteristics of the system.

3.1 Functional Requirements

In order to produce a system which provides a stable, functional language, and an appropriate method of execution, the following criteria must be met:

- Define an irreversible language for the input of the compiler. This language should be a simple imperative language that is Turing complete, containing basic program constructs such as while-loops, conditionals and assignments.
- Define a reversible language for the compiler to output. This language should allow for modified versions of the constructs in the irreversible language, which allow for a Turing complete reversible language.
- Implement a Lexer in Java that allows for the lexing of keywords, identifiers, integers and special characters. The lexer must return errors when the input does not match what the language allows.
- Implement a Recursive Decent Parser in Java. The parser will need to check that the program is syntactically correct, and build an Abstract Syntax Tree (AST) for each program. If the syntax is incorrect, the parser must return an error with information regarding the cause.
- Define how irreversible constructs must be converted such that they become reversible.
- Implement a reversible machine emulator in order to execute reversible code. The emulator must include a stack of instructions which can be executed line by line, forwards or backwards.

3.2 Non-Functional Requirements

- The language should be reliable and consistently produce the same output from the same given input.
- The emulator should run on Windows, Mac, or Linux allowing the reversible language to be portable across different machines.
- The display should be easy to use and understand, and provide a simplistic method for editing and executing code.

- The display should allow for easy debugging of a users code, both with a display of which line of execution it is at, and if any, error reports.

Chapter 4

Background Information

4.1 The Problem of Reversible Computation

The problem of reversible computation is concerned with allowing the execution of a program to be paused at any point, and then have the execution continue going in the opposite direction.

4.2 Forms of Reversible Computation

Reversible computation exists in many different forms, spanning from the high level reversible languages, down to low level reversible machine code and logic gates [6]. This project is focusing its aims in the production of a high level reversible language, and a compiler which translates an irreversible source code into that language.

4.3 Issues Surrounding Reversing Execution

In order to successfully provide a language which is entirely reversible, the control structure and operations must be redefined in a way which ensures a deterministic execution both forwards and backwards. The areas which will require amending are detailed below.

4.3.1 Sequence Reversal

A conventional program would follow a sequence structure of [S_1, S_2, ..., S_{n-1}, S_n], where each statement S_i must be one of the statements described in the following sections. For a program of this kind to be reversed, both the order of execution must be reversed, and each of the individual statements must be replaced with their inverse. The resulting sequence would be as follows [S_n^{-1}, S_{n-1}^{-1}, ..., S_2^{-1}, S_1^{-1}].

4.3.2 Variable Updates

A destructive update is any statement which updates a variable to a new value which cannot be undone with the inverse of that statement. A simple example of this would be x = 5, there is no way of recovering the previous value of x and therefore this is destructive.
Conversely, there are also variable updates known as constructive updates. These are the opposite of destructive, in the sense that the value of the variable before the update is easily recovered by the inverse of the statement. A simple example of this would be a swap statement, x $\Leftrightarrow$ y. As none of the data is being lost in this statement, and it can be easily reversed the other way, this is a constructive update.
In order for a program to be reversible, all destructive variable updates must be either excluded from

the code, or they must be translated in such a way that they become constructive. The way this was achieved will be discussed in the implementation section.

4.3.3 Arithmetic

In regular programming languages arithmetic is largely destructive, which would automatically disqualify it from a reversible language. For example, the expression $5 + 3$ would be considered irreversible because after execution the result would be 8, which upon reversal is impossible to try and recover the original two operands. It is not possible to know which two operands were used to make up the 8, it could have been 8 and 0, 7 and 1, or any other combination that makes up 8.
Alternatively, the most simple constructive arithmetic expressions would be the increment and decrement operations (+=, -=). By definition, these are the inverse operation of each other. Due to this, an operation such as 5+=3 can easily be reversed by doing 5-=3.
Constructive arithmetic expressions will not need any conversion to be allowed to run in a reversible manner, however any of the conventional destructive expressions will need to be translated in such a way that they are made constructive.
Integer division is a special case in reversible arithmetic which will have to be disallowed in this reversible language. This is due to it loosing information upon execution. For example, the integer division of 4/3 returns a value of 1. All of the information which would have followed the decimal point is lost, and as such, when trying to reverse this execution, doing 1*3 will return 3, which is not equal to the original operand. This can be accounted for by forcing all division to take place using real numbers.

4.3.4 Conditionals

When a conditional is executed forwards, the condition will be tested at the start, and depending on the result one branch will be selected and executed. This type of branching is what will cause a problem in the reverse execution of conditionals. Upon reverse execution of a standard conditional, it is not possible to know which branch was originally selected to be executed. Selecting a branch in reverse is not as simple as testing the initial condition again. This is because any variables in the condition could be modified within the branch. In this case the condition may produce different results. This can be seen here:

```
if x > 5 then
    x = 0
else
    x = 10
end
```

If this conditional is executed forwards with any initial value of x, the value of x after each branch is such that the opposite condition will be true on the reverse execution, resulting in the incorrect branch being executed in reverse.
There are two potential solutions to this issue of reversibility using conditionals:

- **Variable restrictions** - As the issue of reversibility only presents itself once a variable from the condition is used within the branch, it is possible to ensure reversibility if it is forced that this never occurs. This can be achieved either by leaving it down to the user of the language, or the compiler can replace the variables used in the condition.

- **Post condition** - In addition to the usual condition (referred to as a pre-condition), a post-condition is added once the branches rejoin. The semantics of this post-condition require that in the forwards execution, at the end of the branching, it must evaluate to the same value as the pre-condition before the branching. This way, the post-condition can be used when executing backwards to correctly determine which branch needs to be executed.

4.3.5 Loops

Two types of loops are featured in most traditional programming languages; variable iteration (while loops) and fixed iteration loops (for loops). Fixed iteration loops are reversible by nature, as on both the forwards and backwards execution, the number of iterations can be calculated using the counter variable. On the other hand, variable iteration loops are not reversible as the number of iterations that were performed on the forwards execution are not able to be recovered on the reverse execution. There are a number of ways of resolving this, however the main two that were considered for this project are:

- **Restricted syntax** - The first way that it could be ensured that loops are made reversible would be to simply remove the option of using variable iteration loops, and only allow fixed loops. Leaving the programmer with only the option of fixed iteration loops would be very restrictive in terms of how many different algorithms could be created in this language, however it does provide a solution to the problem.
- **Counter-based** - The alternative to restricting the syntax would be to use counter based reversal. This involves allowing the compiler to make a temporary variable upon entering a loop which counts how many iterations are performed on the forwards execution. From there, it works similar to a fixed loop in the reverse execution, looping the number of times that was stored in the counter.

4.3.6 Control Flow

There are a number of issues surrounding the reversal of different elements of control flow such as jump instructions, function calls, and exception handling. However for this project the only type of irregular control flow that will be present is the jump instruction, also known as goto. Goto statements are not reversible by nature as it breaks the control flow. This can be fixed by introducing an inverse to this statement, the *comefrom* statement. These statements must come in pairs, and wherever the goto statement points to, there needs to be a comefrom statement which states where the jump came from.

4.4 Existing Applications of Reversible Computation

4.4.1 Janus

An early attempt at a general-purpose reversible programming language is the Janus language [7]. The language was designed to exclude any irreversible features, and forces local reversibility on every statement.
For variables, Janus allows only integer data types in either a scalar or array data structure. All variables are initialised to zero, and can only be modified using constructive variable updates and arithmetic (as described in sections 4.3.2, 4.3.3).
Janus implements reversible conditionals with the use of a post-condition (as described in section 4.3.4).
Variable iteration loops are implemented using a post-condition for the loop. The semantics are such that in the forward execution, the pre-condition is true before the loop is entered, but false after the loop starts, and the post-condition is false until the loop is exited.
A simple Janus program which given an integer n, the procedure fib computes the (n+1)-th and (n+2)-th Fibonacci number [8].

```
procedure fib(int x1,int x2,int n)
   if n=0 then
         x1 += 1
      x2 += 1
   else n -= 1
      call fib(x1,x2,n)
      x1 += x2
      x1 <=> x2
   fi x1=x2
```

4.4.2 R Language

Just over ten years after Janus was released, Michael Frank released a reversible language called R [9]. This is a high-level reversible language which was designed to make programs on the Pendulum Instruction Set Architecture (PISA).
Similarly to Janus, R uses only integers for variable data types, and allows those to be in the form of either scalars or arrays. These variables can be modified using only constructive updates and arithmetic, although multiplication is introduced in a reversible way.
R manages to overcome the issues surrounding selection by ensuring that the condition evaluates to the same value both before and after the branching has occurred.
Variable iteration loops are supported using for loops of the following syntax:
`for counter = start to end`
`start` and `end` must remain constant for the duration of the loop, whereas the value of `counter` can be modified within the body to allow for loops of variable iterations.

4.4.3 Limitations of Existing Solutions

The existing solutions have provided a strong implementation of reversible programming, and have heavily influenced the style of the reversible language created for this project, however there are clear limitations that this project will attempt to overcome.
The most prominent limitation of these reversible languages is how difficult and different the languages are to the mainstream programming languages in use today. This project will attempt to

overcome this using a compiler that will convert an initial source language that is similar in nature to the Java syntax, and converting it into it's reversible form. This gives the user the simplicity and familiarity of a Java-like language, while maintaining the power and features that a reversible language provides.

4.5 The Compiler

In order for a program to be executed on a computer, the source code must first undergo some type of translation from its high-level form into some form which the machine running it will understand. A compiler is a program which can be used to achieve this sort of translation. The compiler as it is used in the scope of this project can be broken down into two main sections; lexical analysis and syntactic analysis. These two sections work together to transform the input from the source code into an Abstract Syntax Tree (AST) which makes code generation far easier.

4.5.1 Lexical Analysis

Lexical analysis is the first stage of the compiler, in which the input source code will be transformed into a series of lexical tokens. Firstly, all whitespace and comments are removed as these are meaningless to the compiler and the final execution. The input is then transformed from strings into tokens, for example the following short program:

```
if(x < 5) {
   print(x);
}
```

Will be transformed into the following list of lexical tokens:

```
IF, LEFT_BRACKET, IDENTIFIER("x"), LESS_THAN, INT_LITERAL(5), RIGHT_BRACKET,
    LEFT_CURLY_BRACKET, PRINT, LEFT_BRACKET, IDENTIFIER("x"), RIGHT_BRACKET, SEMICOLON,
    RIGHT_CURLY_BRACKET
```

It is at this stage where errors will be thrown and compilation will stop if any unrecognised symbols are included in the input.

4.5.2 Syntactic Analysis

Syntactic analysis consists of two main sections, building the AST, and confirming that the syntax is correct. An AST is a data structure which represents the program in a way that maintains it's order such that execution is simply a traversal of the tree. While building an AST, tokens such as brackets and semicolons can be excluded, as the structure of the tree will infer these types of tokens.
A context free grammar (CFG) is a set of recursive rules which are used by the parser to produce a language. A simple example of a CFG can be seen bellow (Note: some required rules have been omitted for simplicity) :

```
PROG → BLOCK
BLOCK → { ENE }
ENE → E; | E; ENE
E → INT
   | ID
   | if (E COMP E) BLOCK
```

It is important to note that while the CFG can parse the input and check for syntactic accuracy, it will not be able to check for semantic accuracy. For example, while the following program is syntactically well-formed: `{if(5 == 5){10}}`, semantically, it is the same as just having the expression: `10`.
Going back to the example source code that was demonstrated in the last section, figure 4.1 shows how the CFG rules defined can be used to parse a list of tokens into an AST.

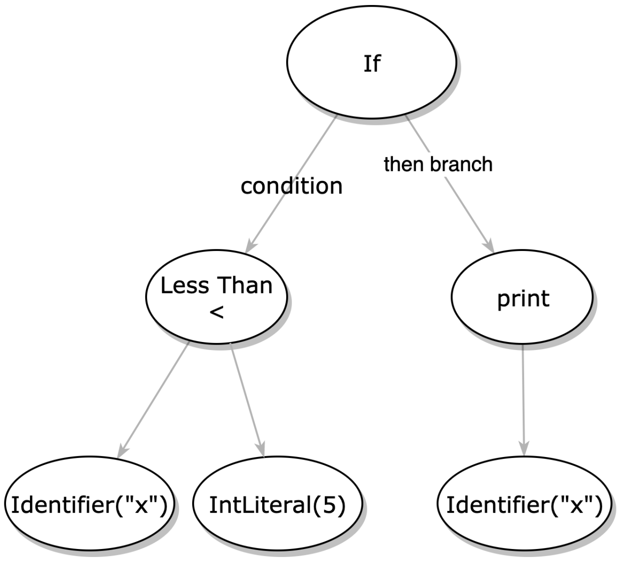


FIGURE 4.1: Example of an Abstract Syntax Tree

4.5.3 AST Transformation

The final stage the compiler must take is to transform the standard AST into a reversible AST. This is the point where additional constructs are added to transform the current irreversible code into code which can be reversed. An example of this can be seen in figures 4.2 and 4.3 using a while loop, with the additional expressions which have been added by the compiler in bold.

```
while(x < 5) {
    x = x + 1;
}
```

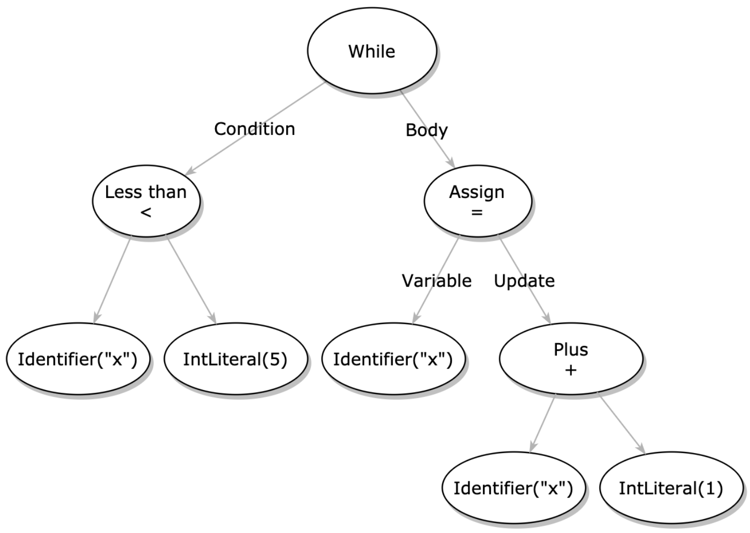


FIGURE 4.2: Example of an Abstract Syntax Tree before transformation

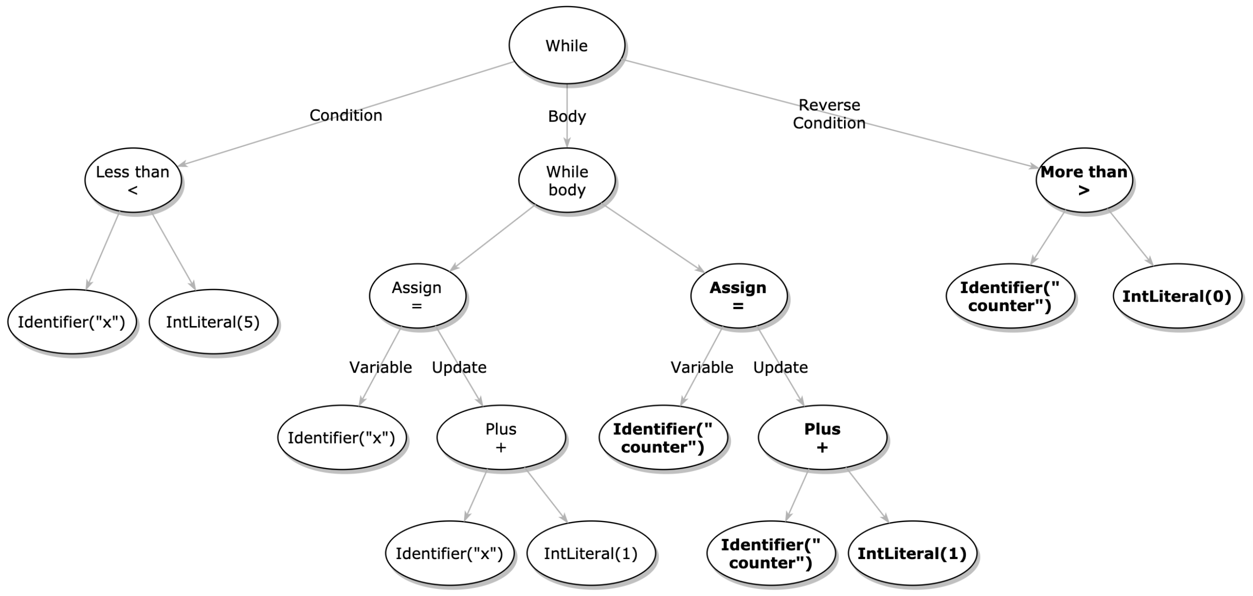


FIGURE 4.3: Example of an Abstract Syntax Tree after transformation

Chapter 5

Implementation

5.1 Source Language

The first stage in designing the new language was to create a Context Free Grammar showing how the source language should be structured. Below is the final, full CFG. It has been altered slightly throughout the implementation of the language, to create room for more expressions, such as the for loop, and allowing for real numbers.

```
DIGIT → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
CHAR → a | b | ... | Y | Z
COMP → == | < | > | <= | >=
BINOP → + | - | * | / | %
PROG → BLOCK
ID → CHAR | ID CHAR
INT → DITGIT | INT DIGIT
DOUBLE → INT . INT
BLOCK → { ENE }
ENE → E; | E; ENE
E → INT
   | ID
   | INT
   | DOUBLE
   | ID = E
   | if (E COMP E) BLOCK
   | if (E COMP E) BLOCK else BLOCK
   | while (E comp E) BLOCK
   | for (E = E : E) BLOCK
   | print(E)
```

This CFG gives rise to a language which can create programs such as:

```
{
    x = 0;
    while(x < 20) {
        if(x == 15) {
            print(x);
        };
    };
    y = x;
}
```

5.2 Reversible Language Implementation

5.2.1 Variable Updates and Arithmetic

It was important to the design of this language to ensure that the user could use it in a way which was syntactically familiar. As such, variable updates and arithmetic needed to use the well known C-like syntax of assignment: `x = 50`, and arithmetic operators: `+, -, *, /`. For the reasons outlined in sections 4.3.2 and 4.3.3, this would not work naturally as these operators are not inherently reversible. The solution to this was to save in memory a stack for each variable which held the operations which had been performed on that variable. The stack was designed such that it held an increment or decrement operation equal to the difference between the current value of a variable, and the desired value of a variable.
For example, in the forwards execution of the code `x = 5; x = 7;`, the stack associated with the variable x would first have the value 5 pushed onto it, as the difference between 0 (the initial value of every variable) and 5 is 5. After the second operation, the stack would then have the value 2 pushed onto it, as the difference between the current value 5, and the desired value 7 is 2.
The reverse of this works by simply popping the top of the stack every time a variable update occurs, which in effect removes a single operation from the memory of a variable.
Evaluating a variable is as simple as summing the contents of the stack.

5.2.2 Conditionals Implementation

As was discussed in section 4.3.4, conditionals could be made reversible by either restricting how the variables are used, or by introducing a post-condition to the conditional. This project achieved reversible conditionals using a slightly modified method of variable restriction. Rather than completely disallowing the use of the condition variables in the body, the compiler modifies the code such that the variable in the condition is a temporary copy of itself. This way, if the value is changed in the body the temporary version remains the same, and can be used in the condition on the reversed execution.
An example of this transformation can be seen below:

```
if(x < 5) {
    print(x);
}
```

After transformation into it's reversible form, would become:

```
x_TEMP = x;
if(x_TEMP < 5) {
    print(x);
}
```

Due to the way that reversible conditionals work, if this temporary variable was used by the programmer in the body of this statement, the code would left with logic errors. For this reason, a better solution to this would be to simply rename the '_TEMP' part of the variable to a character which is undefined in the lexical analysis.

Multiple Branches

Multiple branches within a conditional is seen when there is both a branch to be executed for if the condition is true, and a branch which is executed when the condition fails. This is handled the same way that a single branch is handled, except the compiler must now introduce Goto and Comefrom expressions to ensure that the control flow is followed correctly. An example of what the compiled reversible code would look like for a conditional with multiple branches is as follows:

```
 1| x = 5;
 2| x_TEMP = x;
 3| if(x_TEMP < 10) {
 4|    print(x);
 5|    goto(10)
 6|} else {
 7|    comefrom(3)
 8|    print(x*10);
 9|}
10|comefrom(5, if condition false)
```

Goto expressions are only executed in the forwards execution of the program, and Comefrom expressions are only executed in the reverse execution of the program.

5.2.3 Loop Implementation

In this language, both fixed and variable iteration loops have been implemented. Fixed loops are in the form of for loops, and variable iteration are in the form of while loops.

Fixed Iteration Loops

Fixed iteration loops are trivial to implement and as explained in section 4.3.5, can be reversed by iterating the same number of times that was stated in the expression. As was seen in the R reversible language, it is possible to modify the value of the iteration variable from within the body to either extend or shorten the length of the loop. However, this isn't the most intuitive way for a programmer and as such, variable iteration loops were also implemented.

Variable Iteration Loops

Variable iteration loops are implemented using a counter-based approach as described in section 4.3.5. For each loop, a counter is created and initialised to 0. For each iteration in the forwards execution, the counter is incremented. This counter is then used on the reverse execution as the looping condition. An example of what the compiled reversible code looks like is as follows:

```
0| _COUNTER1 = 0;
1| while(x < 5) {
2|     comefrom(7);
3|     print(x);
4|     x = x + 1;
5|     _COUNTER1 = _COUNTER1 + 1;
6|     goto(1)
7| } endwhile(_COUNTER1 > 0)
```

5.3 Lexer Implementation

The lexer for this project is a hand-written Java class which scans the input text character by character to build a list of tokens. Each individual token is represented by it's own object, and as such there is a Token interface for which each of these tokens can implement. The lexer follows the *longest match* rule, which states that you read a single token for as long as you get characters which can belong to it. For example, take the string "form". While it could be read as the keyword 'for', due to the longest match rule it will be read as the identifier 'form'. In order to implement this, there must be a look-ahead, which is where the lexer will look one character ahead of its current position. The lexer is implemented using a switch-case expression, which follows the following basic form:

```
for each character c in input:
    switch c
    case c is a letter:
        loop until c isn't a letter:
            append c to a string word
        if word is a keyword add keyword to token list
        else add word as an identifier to token list
    case c is a digit:
        loop until c isn't a digit:
            append digit to string number
        add number to token list
    case c is a special character (eg. ';', '+'):
        add c's token to token list
    default:
        Unknown character, throw error
```

5.4 Parser Implementation

For this project, a recursive-decent parser has been written in Java in order to build the AST. A recursive-decent parser is a type of top-down parser which uses a set of recursive procedures where each procedure implements one of the non-terminal elements of the context free grammar.

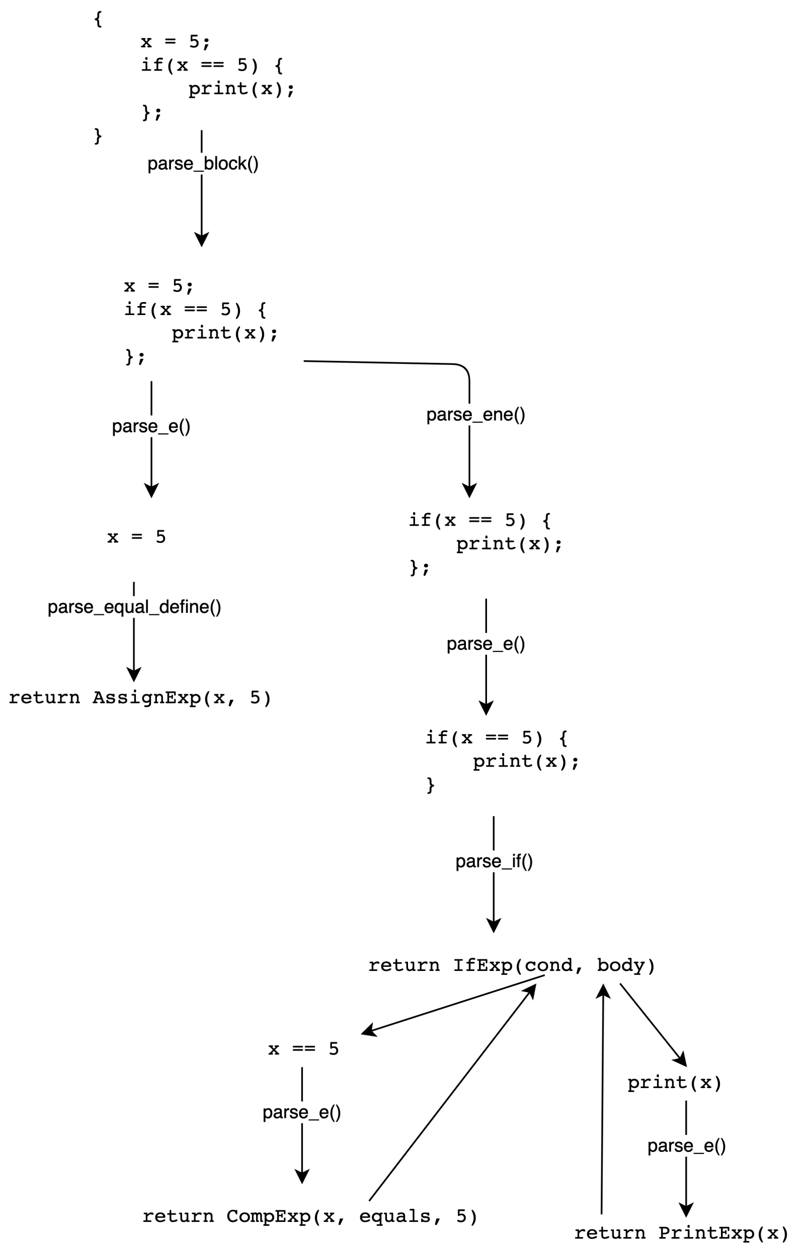
5.4.1 AST Structure

The AST at its root consists of an object of a `Block` class, which contains a list of expressions in the form of `Exp` objects. A subsection of the class signatures which build up the expressions can be seen below:

```
class IntLiteral extends Exp;
class VarExp extends Exp;
class AssignExp extends Exp;
class IfExp extends Exp;
class CompExp extends Exp;
```

5.4.2 Parser Algorithm

The input list of tokens is initially run through two functions; `parse_block` and `parse_ene`. The prior of these removes the curly brackets from the beginning and the end of the program, while the latter splits the remaining program up into a list of individual expressions according to the semi-colons present. Each of the expressions will then be passed into a `parse_e` function, which is used to determine how each input needs to be parsed by looking at the first token in the expression. From here, the token list will be passed into whichever function parses that type of expression, such as `parse_if` or `parse_binop`. Below is a recursive call stack diagram for how the parser will turn an input string of tokens into a list of expressions. The token list is represented in it's code form for simplicity.



```
{
    x = 5;
    if(x == 5) {
        print(x);
    };
}
```

```
x = 5;
if(x == 5) {
    print(x);
};
```

```
x = 5
```

```
return AssignExp(x, 5)
```

```
if(x == 5) {
    print(x);
};
```

```
if(x == 5) {
    print(x);
}
```

```
return IfExp(cond, body)
```

```
x == 5
```

```
return CompExp(x, equals, 5)
```

```
print(x)
```

```
return PrintExp(x)
```

FIGURE 5.1: Example of the recursive nature of the parser

5.4.3 Operator Precedence

In order to remove ambiguity when performing mathematical operations on variables, it is important to define the order in which operations are performed. To ensure this, the Shunting Yard algorithm [10] was implemented. Pseudocode for this algorithm can be seen below, along with a table defining the precedence of each operator:

Operator	Meaning	Precedence
( )	Brackets	5
/	Division	4
%	Modulo	3
*	Multiplication	2
+	Addition	1
-	Subtraction	0

```
For each token t:
    If t is a number add it to queue
    If t is an operator
        While there's an operator on the top of the stack with greater precedence than t:
            Pop operators from the stack onto the queue
        Push the current operator onto the stack
    If t is a left bracket push it onto the stack
    If t is a right bracket
        While there's not a left bracket at the top of the stack:
            Pop operators from the stack onto the queue.
        Pop the left bracket from the stack and discard it
 While there are operators on the stack, pop them to the queue
```

5.5 Emulator Implementation

The final stage of implementation was to emulate a reversible machine to execute the code. This works by traversing the abstract syntax tree which the parser had constructed. It was important to allow the user to run the code either line by line or as a whole program. The emulator works by recursively calling functions which individually dealt with each possible expression. The root of the AST contains a list of expressions, and each of these expressions will be passed into a `executeExp(Exp)` function. This function checks which type of expression it is and in turn passes it to a function of the type `executeWhile(Exp)` or `executeIf(Exp)`. The emulator maintains a list of registers in which all of the variables will be stored, and an integer which represents the current instruction which is being executed. If at any point a given expression fails to be executed, a RuntimeException will be thrown, with a description of what made it fail.

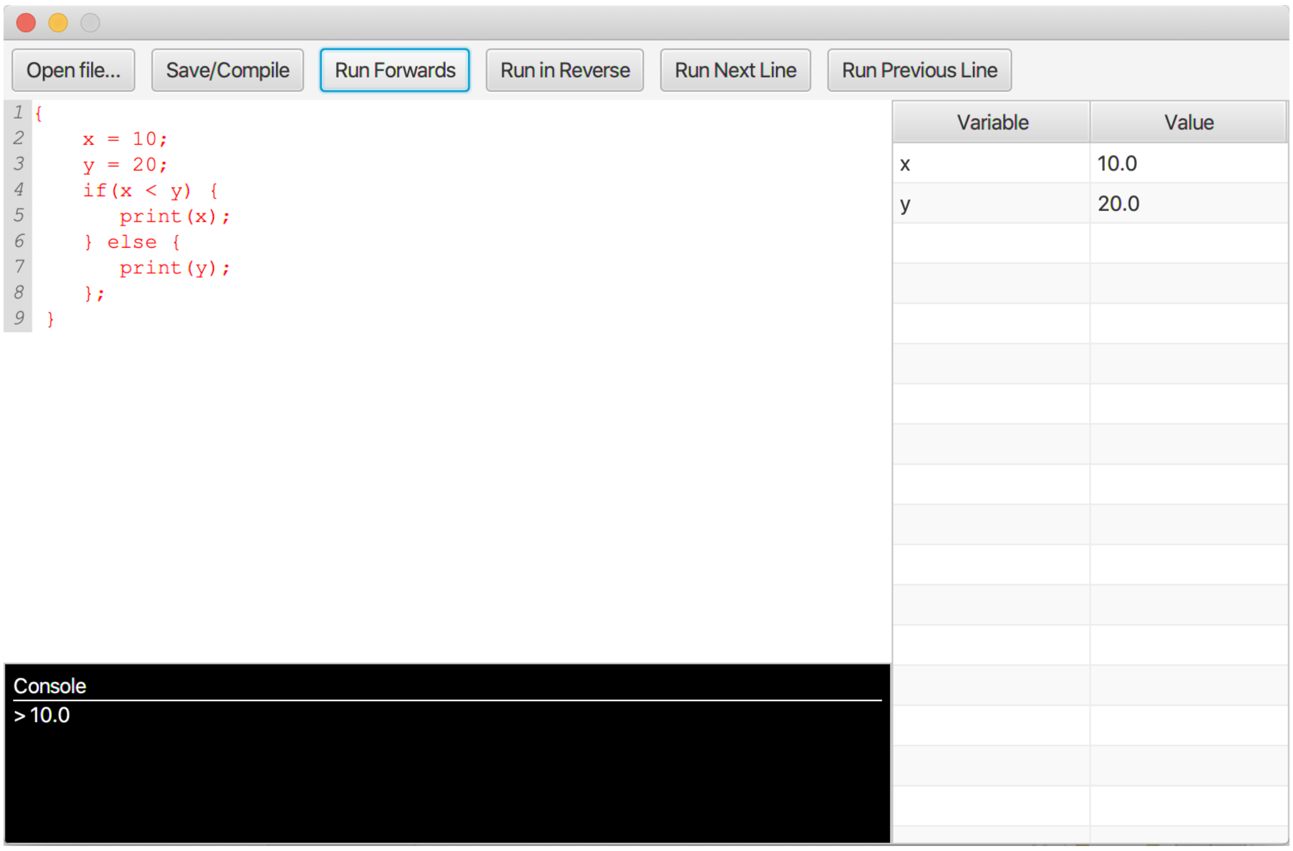
5.6 GUI Design

The following are fundamental features which the GUI must be implemented for full use of the program:

- Main window which allows for programming
- A console where error messages and 'print()' statements will be displayed
- A panel which displays all variables and their current value

- Buttons to allow for forwards and backwards execution both as a whole program and line by line
- A button to open a source code file
- A button to save and compile the source code file

5.6.1 Final design of the GUI



```
{
    x = 10;
    y = 20;
    if(x < y) {
       print(x);
    } else {
       print(y);
    };
}
```

Variable	Value
x	10.0
y	20.0

```
Console
> 10.0
```

FIGURE 5.2: Final GUI design of the program

Chapter 6

Language Usage

In order to demonstrate the practical use of this language, a few different algorithms will be constructed and the source code and output will be documented.

6.1 Finding Largest Number

As an initial trivial use case of the language, it is shown here using a conditional in order to display the larger of the two input numbers.

Source code

```
{
   x = 10;
   y = 20;
   if(x > y) {
      print(x);
   } else {
      print(y);
   };
}
```

Output

```
20
```

6.2 Fibonacci Sequence

As can be seen by the implementation of the Fibonacci sequence in this language, it is almost identical to an equivalent implementation in for example Java.

Source code

```
{
   i = 0;
   j = 1;
   while(i < 30) {
      print(i);
      temp = j;
      j = i + j;
      i = temp;
   };
}
```

Output

```
0
1
1
2
3
5
8
13
21
```

6.3 FizzBuzz

Implementing the FizzBuzz program causes issues for this language for a number of reasons. While it is clear that the implementation of this program in this language is far from optimal, it is a clear demonstration that with sufficient additions to the grammar, reversible languages can be powerful. Issues that had to be overcome:

1. There is no concept of a String, so printing either "fizz" or "buzz" is made impossible. To overcome this, the problem is changed slightly such that on a multiple of three, "3" is printed, on a multiple of five, "5" is printed, and on a multiple of three and five, "35" is printed.

2. If statements may only have one condition, unlike for example in Java where a condition could be "x == 3 && x == 5". To overcome this, nested if statements have been utilised.

3. Unlike a number of languages, this does not allow for the syntax "else if", and as such, further nesting of if statements has been using to get the required result.

Source Code

```
{
   for(i = 10:20) {
      if(i % 3 == 0) {
         if(i % 5 == 0) {
            print(35);
         } else {
            print(3);
         };
      } else {
         if(i % 5 == 0) {
            print(5);
         } else {
            print(i);
         };
      };
   };
}
```

Output

```
5                      (equivalent to buzz)
11
3                      (equivalent to fizz)
13
14
35                     (equivalent to fizzbuzz)
16
17
3                      (equivalent to fizz)
19
```

Chapter 7

Testing

7.1 JUnit Testing

Testing was done using Java's JUnit testing framework in order to check the correctness of the lexer, parser, and execution of code both forwards and backwards. Each function of each of these systems have been broken down into small sections that can be accurately tested according to a hard-coded expected value.

7.1.1 Lexer Testing

For the following tests, the lexer was tested against any possible input it could have received. A pass result was given by comparing the list of tokens which the lexer produces against a hard-coded list of tokens for each of the given inputs.

Test	**Input Data**	**Expected Output**	**Result**
Testing that a non-keyword is lexed as an identifier	"var_1"	T_Identifier("var_1")	Pass
Testing that a number will be lexed as a double	"30.5"	T_Double(30.5)	Pass
Testing that all of the allowed special characters are lexed accordingly	";():-+=/*%==!="	A list of tokens with each item representing each symbol	Pass
Testing that the longest match rule applied correctly	"=== forex whileident"	T_Equal(), T_EqualDefines(), T_Identifier("forex"), T_Identifier("whileident")	Pass
Testing that incorrect characters produce a lexing error	"& $ # ? £ @ ^"	Lexing should fail, and the errors list should have a length of 7	Pass

TABLE 7.1: Table displaying the tests that the lexer underwent

7.1.2 Parser Testing

For the following tests, the list of the instructions that the parser produces was compared to a hard-coded expected value for which the input should have produced. A pass result was given if the expected value of the list was equal to the actual output list that was produced.

Test	Input Data	Expected Output	Result
Testing that a variable is parsed correctly	"{x;}"	VarExp("x")	Pass
Testing that mutliple lines of code are parsed correctly	"{x;y;}"	VarExp("x"), VarExp("y")	Pass
Testing that a for loop is parsed correctly	"{for(x = 1:2){x;};}"	AssignExp(), ForExp(), ComeFromExp(), VarExp(), AssignExp(), GotoExp(), EndForExp()	Pass
Testing that a while loop is parsed correctly	"{while(x<1){x;};}"	AssignExp(), WhileExp(), ComeFromExp(), VarExp(), AssignExp(), GotoExp(), EndWhileExp()	Pass
Testing that an if statement is parsed correctly	"{if(x<1){x;};}"	CompilerExp(), IfExp(), VarExp(), GotoExp(), ComefromExp(), EndIfExp()	Pass
Testing that an if-else statement is parsed correctly	"{if(x<1){x;} else {y;};}"	CompilerExp(), IfExp(), VarExp(), GotoExp(), ComefromExp(), VarExp(), EndIfExp()	Pass
Testing that the print statement is parsed correctly	"{print(5);}"	PrintExp()	Pass
Testing that all of the comparison operators are parsed correctly	"{x<5;x>5;x==5; x!=5;x<=5;x>=5;}"	CompExp(), CompExp(), CompExp(), CompExp(), CompExp(), CompExp()	Pass

Testing that all of the binary operators are parsed correctly	"{x+5;x-5;x*5;x/5;x%5;}"	BinopExp(), BinopExp(), BinopExp(), BinopExp(), BinopExp()	Pass
Testing that equal define is parsed correctly	"{x=5;}"	AssignExp()	Pass

TABLE 7.2: Table displaying the tests that the parser underwent

Invalid Input Parsing Tests

For the following tests, it was tested that the output produced for each input was a SyntaxException describing what the error was. SyntaxException is an error class which extends the Java class Exception. Each of the following tests were produced with the intent to produce errors.

Test	Input Data	Expected Output	Result
Testing inaccurate curly brackets	"{"	SyntaxException("Input not in form "{ <ene> }" ")	Pass
Testing a missing semi-colon	"{x}"	SyntaxException("Missing Semi-colon")	Pass
Testing an incorrectly formatted if-statement	"{if(x<5{x;};}"	SyntaxException("Incorrect Brackets")	Pass
Testing an incorrectly formatted while-loop	"{while(x<5{x;};}"	SyntaxException("Incorrect Brackets")	Pass
Testing an incorrectly formatted for-loop	"{for(x<5){x;};}"	SyntaxException("Error parsing for")	Pass
Testing an empty print statement	"{print();}"	SyntaxException("Token list size is 0")	Pass
Testing an incorrect expression in the form of an assignment	"{5=5;}"	SyntaxException("Error with token list")	Pass

7.1.3 Forwards Execution Tests

For the following tests, the value of a variable were recorded after the execution of the input code. This value was compared to a hard-coded expected value based on what the logic of the program should have produced.

Test	Input Data	Expected Output	Result
Testing that assignment changes variable values correctly	"{x=5;}"	x == 5	Pass
Testing that the less-than operator works correctly in an if-statement	"{x=5; if(x<10){x=10;};}"	x == 10	Pass

Testing that the more-than operator works correctly in an if-statement	"{x=5; if(x>1){x=10;};}"	x == 10	Pass
Testing that the less-than or equals operator works correctly in an if-statement	"{x=10; if(x<=10){x=20;};}"	x == 20	Pass
Testing that the more-than or equals operator works correctly in an if-statement	"{x=10; if(x>=10){x=20;};}"	x == 20	Pass
Testing that the not-equals operator works correctly in an if-statement	"{x=10; if(x!=5){x=20;};}"	x == 20	Pass
Testing that the binary operator addition produces the correct value	"{x = 5 + 3;}"	x == 8	Pass
Testing that the binary operator subtraction produces the correct value	"{x = 5 - 3;}"	x == 2	Pass
Testing that the binary operator multiplication produces the correct value	"{x = 5 * 3;}"	x == 15	Pass
Testing that the binary operator division produces the correct value	"{x = 5 / 2;}"	x == 2.5	Pass
Testing that the binary operator modulo produces the correct value	"{x = 5 % 2;}"	x == 1	Pass
Testing that the equals operator evaluates correctly in an if-statement	"{x=5; if(x==5){x=10;};}"	x == 10	Pass
Testing that branching works correctly, and that the correct branch is chosen for a given condition	"{x=5; if(x==1) {x=10;}else{x=15;};}"	x == 15	Pass
Testing that a for loop executes the body the correct number of times	"{for(x = 1:5){x;};}"	x == 5	Pass
Testing that a while loop executes the body as long as the condition is true	"{x = 1; while(x < 5){x = x + 1;};}"	x == 5	Pass

TABLE 7.4: Table displaying the tests which were run on the forwards execution

7.1.4 Reverse Execution Testing

For the following tests the code displayed in the input column was executed forwards and then backwards again. The value of a variable was compared against a hard-coded expected value both after the forwards execution and again after a backwards execution. The test is marked as passed if both values are tested as equal.

Test	Input Data	Expected Output	Result
Testing the reverse execution of assignment	"{x = 5;}"	x == 5, x == 0	Pass
Testing reverse execution of if-statements	"{x = 5; if(x == 5) {x = 10;};}"	x == 10, x == 0	Pass
Testing reverse execution of if-else-statements	"{x = 5; if(x == 10) {x = 10;} else {x = 20;};}"	x == 20, x == 0	Pass
Testing that for loops are executed the correct number of times in reverse	"{for(x = 1:5){x;};}"	x == 5, x == 0	Pass
Testing that while loops are executed the correct number of times in reverse	"{x = 0; while(x < 5) {x = x + 1;};}"	x == 5, x == 0	Pass
Testing that addition is consistent when executed in reverse	"{x = 5; x = x + 5;}"	x == 10, x == 0	Pass
Testing that subtraction is calculated correctly in reverse	"{x = 10; x = x - 5;}"	x == 5, x == 0	Pass
Testing that multiplication is calculated correctly in reverse	"{x = 5; x = x * 10;}"	x == 50, x == 0	Pass
Testing that division is calculated correctly in reverse	"{x = 5; x = x / 5;}"	x == 1, x == 0	Pass
Testing that modulo is calculated correctly in reverse	"{x = 10; x = x % 3;}"	x == 1, x == 0	Pass

TABLE 7.5: Table displaying the tests that the reverse execution underwent

7.2 GUI Testing

Test	Expected Output	Result
Clicking the 'Open file' button	The system should display a file explorer allowing the user to select a code file they wish to modify	Pass
Opening a file when browsing the file explorer	The system should load the file into the code window and auto-format the code.	Pass
Clicking the 'Save/compile' button	The system should save the code into the file that is open, format the code, and run the code through the lexer and parser. If any errors are present in the code, this is when the errors will be picked up	Pass

Clicking the 'Run forwards' button	The code should be executed as a whole program forwards. The source code should all turn red, indicating the entire program has successfully run. The variable list and console should be displayed according to the program.	Pass
Clicking the 'Run in reverse' button	The code should be executed in reverse as a whole program. The code should be reset to a default black colour indicating the position of the instruction pointer. The variable display and console should be updated accordingly.	Pass
Click the 'run next line' button	The code should be executed forward for a single line. The current line should turn red to indicate it has been executed. The variable list and console should be updated accordingly.	Pass
Click the 'run previous line' button	The code should be executed in reverse for a single line. The current line should turn back to black to indicate it has been un-executed. The variable list and console should be updated accordingly.	Pass

TABLE 7.6: Table displaying the tests that the GUI underwent

Chapter 8

Evaluation

This section will detail to what extent this project has met the requirements outlined in sections 3.1 and 3.2.

8.1 Source Language

The source language outlined in section 5.1 in the form of a CFG was implemented in full. This language contains all of the necessary constructs such as while-loops, conditionals and assignments and as such can be classified as a Turing complete language. Thus, this language meets the conditions outlined in the first point of the functional requirements. One issue with this language is that it lacks the ability to have more than two branches with the use of 'else-if' in a conditional. This issue can be seen in the FizzBuzz implementation in this language. While this limitation does not hold back the completeness of this language, extending it to include multiple branches would allow for far easier programming when using this language.

8.2 Reversible Language

As was detailed in section 5.2, the output from the compiler was a fully functional reversible language which had modified versions of all of the constructs available in the source language. As such, the reversible language can also be considered Turing complete. While it is true that each of the irreversible constructs have a one-to-one relationship to a reversible construct which performs the same function, the reversible language requires some memory overhead to allow for it's reversibility. To allow for more flexibility with arithmetic in this language, a history of increments must be maintained for each variable used in a program. This is generally not too damaging, as the memory overhead is relatively small, however if the user has a strict memory usage, the language would need to be scaled down to only allow for increments and decrements. The reverse execution testing shown in section 7.1.4 helped ensure that all of the features are indeed locally reversible. While this is a good start, there is a possibility that different situations could result in a program not being correctly reversible. For example, while each operation has been proved to be locally reversible, on a global scale different operations may affect one another and cause problems. No examples of this have been found, and going forward it can be monitored and troubleshooted as appropriate.

8.3 Lexer Evaluation

The testing of the lexer seen in section 7.1.1 shows it working as planned in the requirements. It correctly identifies which inputs are keywords, identifiers or numbers and outputs the correct token accordingly. As every legal input has been tested for, it is possible to say that the lexer works to it's full capacity and without any errors. For the purpose of this project, the lexers efficiency is of little importance (to a certain degree), however for the lexer to be improved upon the next step would most likely be to use a lexer generator such as JFlex [11] to allow for a much more efficient lexing.

8.4 Parser Evaluation

As can be seen in the testing of the parser in section 7.1.2, the parser works to the full extent of what has been tested. Every construct which was tested upon was parsed into the correct form of an AST, with the correct conversions being made into a reversible AST. Incorrect input was also tested for, and it can be shown to not accept any incorrect syntax. Syntax Error's were thrown on the line that caused the error and a brief description of the error is given in console. As such, the parser can be seen to meet the initial requirements which were set out. However, the relatively small test size which the parser underwent is likely not large enough to entirely rule out that there are no small bugs in the implementation. For example, it is possible that different combinations of expressions being nested within each other could cause the parser to interpret the program incorrectly and either produce an incorrect AST, or incorrectly throw an error.

8.5 Reversible Conversion Evaluation

The conversion from the irreversible programming constructs to reversible ones have been detailed in section 5.2. This can be considered to successfully meet the criteria set out in the requirements, as every construct has a one-to-one relationship with a reversible version of itself. The reversible constructs used were overall a large success, with almost no detriment to the possible logic that can be performed in the language. There is however a downside to some of the conversions that have been selected for this language. For example, both loops and selection require the use of additional variables in order to keep track of the conditions. This comes with a memory overhead that was not present in the initial source code. On a small scale, these additional variables are entirely unnoticeable and have a negligible impact on the performance, although it is possible that should this language be used in industry this memory overhead would need to be considered.

8.6 Emulator Evaluation

The emulator testing in sections 7.1.3 and 7.1.4 displays it correctly evaluating a reversible program by executing it both forwards and backwards. The emulator makes use of a stack of instructions which can be iterated through either one at a time, or as an entire program in both directions. As such, the emulator can be considered to meet the criteria outlined in the requirements. The unit testing for the emulator made sure that the execution of programs occurred as it should, however testing the concept of reversibility is difficult and the current test suit has flaws which would need to be addressed before taking this project any further. Firstly, the test set was too small to entirely rule out the possibility that some programs will not run as expected. Secondly, the only way in which the reversibility is tested was by using hand-crafted programs which modified the value of variables based off what the logic of a program should accomplish. This highlights two possible weaknesses, the first being the possibility for human error in calculating the logic behind the programs. The second being that these variables could have only been the correct values by chance, and it is only testing that the program does *what* it should do, rather than *how* it should do it.

Chapter 9

Conclusion

This project has provided an insight into the issues surrounding reversible computation and has provided a programming language and environment to utilise certain benefits that reversible computation provides.
The primary objectives that the project set out to achieve have been completed in full, and the syntax of the source language makes it ideal for the target audience. The methods displayed in this report show many of the possible ways to overcome the issues presented by reversible computation as well as demonstrating the practical uses of a reversible language.
There are several future improvements which this system could benefit from:

- Increased syntax specifically introducing 'else-if' statements to allow for the same conditional to extend to more than two branches.
- Extension of the data types to include for example booleans and characters.
- Introducing arrays and other collection based data-structures.
- Extend the compiler to output the standalone reversible code. which can be executed at a later point.
- Lexer and Parser improved such that the error messages they produce give more information on the location and reason for the error.
- Language optimisations could include reducing memory overhead, or extending the compiler to perform optimisations on the output code, for example the fusion of for loops which iterate the same number of times.

Chapter 10

References

[1] R. Landauer, "Irreversibility and heat generation in the computing process", IBM Journal of Research and Development, 1961.

[2] K. S. Perumalla, Introduction to Reversible Computing, CRC Press, 2014.

[3] C. H. Bennett, "Logical Reversibility of Computation" in IBM Journal of Research and Development, 1973.

[4] Charles Bennett, "Thermodynamics of computation". International Journal of Physics, 1982.

[5] Steane, Andrew, "Quantum computing" Reports on Progress in Physics 61, 1998.

[6] Thomsen, Michael Kirkedal, Holger Bock Axelsen, and Robert Glück, "A reversible processor architecture and its reversible logic design." In International Workshop on Reversible Computation, 2011.

[7] C. a. D. H. Lutz, "Janus: A time-reversible language. A letter to R. Landauer", 1986.

[8] Tetsuo Yokoyama, "Reversible Computation and Reversible Programming Languages", 2010.

[9] Frank, M.P. and Knight Jr, T.F., "Reversibility for efficient computing", Doctoral dissertation, Massachusetts Institute of Technology, 1999.

[10] E.W Dijkstra, "Algol 60 translation : An algol 60 translator for the x1 and making a translator for algol 60", Stichting Mathematisch Centrum, 1961.

[11] Gerwin Klein, "JFlex User's Manual", 2001