CMP-6048A Advanced Programming

Project Report - 13 January 2025

Vec3 Maths Bytecode Interpreter

Group members:
Jamie Wales, Jacob Edwards.

School of Computing Sciences, University of East Anglia

Version 2.0

Abstract

Vec3 is a bytecode interpreted maths language complete with a GUI, plotting, full static type inference and various maths functions. The language is designed to be simple to use and understand, with a focus on strict mathematical expressions and plotting, but with more powerful constructs such as recursive bindings, first class functions and static type inference. The language and GUI are written in F#, using Avalonia(Avalonia, 2024) for the GUI and ScottPlot(ScottPlot, 2024) for plotting.

The language is compiled to a custom bytecode, which is then interpreted by a virtual machine. The language also has the ability to transpile to C, which can then be compiled and run as a standalone executable, allowing for faster execution of the code.

It is a functional-style language, with a focus on immutability, recursion, expressions and correctness.

Some of the features of the language include:

- Plotting of functions and data
- Recursive bindings
- First class functions
- Static type inference
- Strongly typed vector and matrix types
- Transpilation to C
- Async functions
- A GUI
- Lots of maths utilities, many of which are implemented in the language itself

We used an agile approach during development, where we would work on features in small iterations, with regular meetings and discussions to ensure we were on track.

Chapter 1

Introduction

1.1 Project statement

Vec3 is a bytecode interpreted maths language complete with a GUI, plotting, full static type inference and various maths functions. The language is designed to be simple to use and understand, with a focus on strict mathematical expressions and plotting, but with more powerful constructs such as recursive bindings, first class functions and static type inference. The language and GUI are written in F#, using Avalonia(Avalonia, 2024) for the GUI and ScottPlot(ScottPlot, 2024) for plotting.

It also has the ability to transpile to C, which can then be compiled and run as a standalone executable, allowing for faster execution of the code.

1.2 Aims and objectives

The aim of the project is to create an easy to use maths language with a focus on plotting and mathematical expressions, but with more powerful constructs such as recursive bindings, first class functions and static type inference, and ensuring as many features, both optional and mandatory, are implemented as possible and to a high standard.

Table 1.1: MoSCoW

Priority	Task	Comments
Must	Arithmetic operations	$+, -, /, *, \%,$
	Plotting	Plotting of functions and data
	Functions	User defined functions
	GUI	A GUI for the language with a text editor
	Maths tools	Lots of maths utilities (cos, etc)
Should	Rational numbers	Rational number support
	Complex numbers	Complex number support
	Static typing	Static typing and analysis
	First class functions	First class functions
	Control flow	If statements, loops
	Compound data types	Vectors, matrices, records
Could	Compilation	ByteCode compilation and interpretation
	Transpilation	Transpilation to C
	Async	Async functions
	Static type inference	Hindler-Miller type inference
	Error handling	Error handling
	Recursion	Recursive bindings
	Drawing	Drawing on the GUI
	Importing	Importing of external libraries
Should not	Compilation	Compilation directly to ASM
	Transpilation	Transpilation to other languages
	GUI	A web-based GUI
	FFI	Foreign function interface

Chapter 2

Background

2.1 The Task

The task was to create a maths language with a focus on plotting and mathematical expressions, as well as an integrated GUI, with possible support for rational and complex numbers, control flow and functions.

2.2 Other languages

Many similar systems are available, and inspiration was taken from many, such as MathWorks (2024), R (2024), Desmos Studio PBC (2023), Haskell (2024a) and Python (2024).

MATLAB is a popular language for mathematical computing, with a focus on matrix operations and plotting. It is widely used in academia and industry, and has a large number of built-in functions for mathematical operations.

R is a language and environment for statistical computing and graphics. This is more of a general purpose programming language than MATLAB, but is still widely used in academia and industry, particularly in the field of data science. It is quite a powerful language, with a functional style and notable interesting features such as lazy evaluation. We enjoyed the functional style and maths focus.

Desmos is a web-based graphing calculator, which allows users to plot functions and data. It is very user-friendly, and has a simple and intuitive interface. We liked the idea of being able to add dynamically plotted functions to the graph, giving instant feedback to the user.

Haskell is a functional programming language with a strong, static type system. It is a very powerful language, with a focus on correctness and expressiveness, and is based on a branch of mathematical logic called the lambda calculus. We liked the idea of a strong, static type system, ensuring correctness and the expressiveness of the language, and felt it translated almost directly to mathematical expressions.

Chapter 3

Development History

3.1 Sprint 1: Basic expressions

This sprint focused on implementing a lexer and parser for the language, with precedence rules for the arithmetic operators, parsed with Pratt parsing.

3.1.1 Grammar in BNF

```
<expr> ::= <term> | <term> '+' <expr> | <term> "-" <expr>
<term> ::= <factor> | <factor> "*" <term> | <factor> "/" <term>
<factor> ::= <number> | "(" <expr> ")"
<number> ::= <int> | <float>
<int> ::= <digit> | <digit> <int>
<float> ::= <int> "." <int>
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

3.2 Sprint 2: Variable assignment

In this sprint we added variable assignment to the parser, with the ability to bind an expression to a variable name, as well as a few new operators such as == for equality and % for modulo, as well as unary operators.

Variable bindings are in the form `let` *`identifier = expr`*.

3.2.1 Grammar in BNF

```
<stmtlist> ::=  <stmt>
              | <stmt> <stmtlist>
<stmt> ::=  <expr>
          | "let" <identifier> "=" <expr>

<expr> ::=  <term>
          | <term> "+" <expr>
          | <term> "-" <expr>
          | <term> "==" <expr>
          | <term> "!=" <expr>
          | <term> "<" <expr>
          | <term> ">" <expr>
          | <term> "<=" <expr>
          | <term> ">=" <expr>
          | <term> "&&" <expr>
          | <term> "||" <expr>
<term> ::=  <factor>
          | <factor> "*" <term>
```

```
          | <factor> "/" <term>
          | <factor> "%" <term>
<factor> ::=  <number>
            | <identifier>
            | <unaryop> <factor>
            | "(" <expr> ")"
            | <factor> "^" <factor>

<unaryop> ::= "-" | "!" | "+" | <userop>

<number> ::= <int> | <float>
<int> ::= <digit> | <digit> <int>
<float> ::= <int> "." <int>
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

<identifier> ::= <letter> | <letter> <identifier>
<letter> ::=  "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m"
            | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"
```

3.3 Sprint 3: Interpreter

In this sprint a basic interpreter was implemented, with the ability to evaluate expressions and variable bindings. We used a simple environment to store variable bindings (a map of *string* name to *expr*), and a recursive evaluation function to evaluate expressions. It was a REPL style interpreter, where the last expression of a statement list was evaluated and printed, and ran through the command line. We did not change the grammar in this sprint.

3.4 Sprint 4: Functions

In this sprint we added the ability to define and call functions, with a simple lambda syntax of the form `(`*`args`*`) -> `*`expr` and function calls of the form *`funcName / lambda`*`(`*`args`*`)`. Call by value semantics were used, with a new environment created for each function call, consisting of the arguments bound to the parameter names and the parent environment.

We also added an *assert* statement, allowing for simple tests to be written in the language and support for rational numbers was added.

Rational numbers are in the form `1/2` (note the lack of spaces around the `/`), and allowed for more precision in arithmetic operations, compared to floating point numbers.

3.4.1 Grammar in BNF

```
<stmtlist> ::=  <stmt>
              | <stmt> <stmtlist>
<stmt> ::=  <expr>
          | "let" <identifier> "=" <expr>

<expr> ::=  <term>
          | <term> "+" <expr>
          | <term> "-" <expr>
          | <term> "==" <expr>
          | <term> "!=" <expr>
          | <term> "<" <expr>
          | <term> ">" <expr>
          | <term> "<=" <expr>
          | <term> ">=" <expr>
```

```
          | <term> "&&" <expr>
          | <term> "||" <expr>
<term> ::=  <factor>
          | <factor> "*" <term>
          | <factor> "/" <term>
          | <factor> "%" <term>
<factor> ::=  <number>
            | <unaryop> <factor>
            | <identifier>
            | "(" <expr> ")"
            | <factor> "^" <factor>
            | <factor> "(" <exprlist> ")"
            | <lambda>
<lambda> ::= "(" <exprlist> ")" "->" <expr>

<unaryop> ::= "-" | "!" | "+" | <userop>

<number> ::= <int> | <float> | <rational>
<int> ::= <digit> | <digit> <int>
<float> ::= <int> "." <int>
<rational> ::= <int> "/" <int>
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

<identifier> ::= <letter> | <letter> <identifier>
<letter> ::=  "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m"
            | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"

<exprlist> ::= <expr> | <expr> "," <exprlist>
```

3.5 Sprint 5: Static type checking

In this sprint we added static type checking to the language, with a simple type inference system based on Hindley-Milner.

We go over this system in more detail in section 4.7.

The concept of types was introduced, with the types `Int`, `Float`, `Bool`, `Function` and `Never`.

3.5.1 Grammar in BNF

```
<stmtlist> ::=  <stmt>
              | <stmt> <stmtlist>
<stmt> ::=  <expr>
          | "let" <identifier> "=" <expr>
          | "let" <identifier> ":" <type> "=" <expr>

<expr> ::=  <term>
          | <term> "+" <expr>
          | <term> "-" <expr>
          | <term> "==" <expr>
          | <term> "!=" <expr>
          | <term> "<" <expr>
          | <term> ">" <expr>
          | <term> "<=" <expr>
          | <term> ">=" <expr>
```

```
          | <term> "&&" <expr>
          | <term> "||" <expr>
<term> ::=  <factor>
          | <factor> "*" <term>
          | <factor> "/" <term>
          | <factor> "%" <term>
<factor> ::=  <number>
            | <identifier>
            | <unaryop> <factor>
            | "(" <expr> ")"
            | <factor> "^" <factor>
            | <factor> "(" <exprlist> ")"
            | <bool>
            | <lambda>
<lambda> ::=  "(" <typedexprlist> ")" "->" <expr>
            | "(" <typedexprlist> ")" ":" <type> "->" <expr>


<unaryop> ::= "-" | "!" | "+" | <userop>

<bool> ::= "true" | "false"

<number> ::= <int> | <float> | <rational>
<int> ::= <digit> | <digit> <int>
<float> ::= <int> "." <int>
<rational> ::= <int> "/" <int>
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

<identifier> ::= <letter> | <letter> <identifier>
<letter> ::=  "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m"
            | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"

<exprlist> ::= <expr> | <expr> "," <exprlist>
<typedexprlist> ::=  <expr> ":" <type>
                   | <expr> ":" <type> "," <typedexprlist>
                   | <expr> "," <typedexprlist>

<type> ::= "int" | "float" | "bool" | "(" <typelist> ")" "->" <type>

<typelist> ::= <type> | <type> "," <typelist>
```

3.6 Sprint 6: Bytecode

In this sprint the interpreter was rewritten to use a bytecode interpreter, with a stack based virtual machine as well as a simple bytecode compiler, allowing for more efficient evaluation of expressions.

The VM and compiler were written in F# and used a simple stack based architecture, with a small instruction set. Additionally, in order to simplify the VM, binary and unary operators were implemented as functions in the language, instead of separate AST nodes for *BinaryOp* and *UnaryOp*. This helped to simplify not only the AST and the bytecode, but the architecture of the whole system. For example, it prevented a lot of code repetition during type inference.

The grammar was not changed in this sprint.

3.7 Sprint 7: GUI

A simple GUI was developed in order to allow easier testing of the language, with a text box for input and output and a decompiler output for debugging. The GUI was written in F# using Avalonia(Avalonia, 2024). We did not change the grammar in this sprint.

3.8 Sprint 8: Plotting

In this sprint we added the ability to plot lists of points, with a simple plotting function that took a list of x coordinates and a list of y coordinates and plotted them on a graph using ScottPlot (2024). Naturally, we had to add a new type, `List`, to the language, and as an extension of this, we added the ability to define lists using the syntax `[1, 2, 3, 4]`. Lists are type encoded with their length and the type of their elements, and are immutable. A few functions were added such as *dot* and *cross* for vector operations, which both make use of the length encoding to prevent runtime errors.

Other compound data types such as tuples and records were also added, with the syntax `(1, 2, 3)` and {`x=1, y=2`} respectively.

3.8.1 Grammar in BNF

```
<stmtlist> ::=  <stmt>
              | <stmt> <stmtlist>
<stmt> ::=  <expr>
          | <vardecl>
          | <assertion>

<vardecl> ::= "let" <identifier> "=" <expr>
            | "let" <identifier> ":" <type> "=" <expr>

<assertion> ::= "assert" <expr> | "assert" <expr> <string>


<expr> ::=  <term>
          | <term> "+" <expr>
          | <term> "-" <expr>
          | <term> "==" <expr>
          | <term> "!=" <expr>
          | <term> "<" <expr>
          | <term> ">" <expr>
          | <term> "<=" <expr>
          | <term> ">=" <expr>
          | <term> "&&" <expr>
          | <term> "||" <expr>
<term> ::=  <factor>
          | <factor> "*" <term>
          | <factor> "/" <term>
          | <factor> "%" <term>
<factor> ::=  <literal>
            | "(" <expr> ")"
            | <factor> "^" <factor>
            | <unaryop> <factor>
            | <factor> "(" <exprlist> ")"
            | <factor> "." <identifier>
            | <factor> "[" <expr> "]"
            | <factor> "." <identifier>
```

```
            | <factor> "[" <expr> ":" <expr> "]"
            | <factor> "[" <expr> ":" "]"
            | <factor> "[" ":" <expr> "]"

<unaryop> ::= "-" | "!" | "+" | <userop>

<literal> ::= <number> | <identifier> | <bool> | <list> | <lambda> | <string> | "()" | <t
<string> ::= '"' <charlist> '"' | '""'
<charlist> ::= <char> | <char> <charlist>
<list> ::= "[" <exprlist> "]"
<tuple> ::= "(" <exprlist> ")"
<record> ::= "{" <recordlist> "}"
<recordlist> ::=  <identifier> "=" <expr>
                | <identifier> "=" <expr> "," <recordlist>
                | <identifer> ":" <type> "=" <expr>
                | <identifier> ":" <type> "=" <expr> "," <recordlist>

<lambda> ::=  "(" <typedexprlist> ")" "->" <expr>
            | "(" <typedexprlist> ")" ":" <type> "->" <expr>

<bool> ::= "true" | "false"

<number> ::= <int> | <float> | <rational>
<int> ::= <digit> | <digit> <int>
<float> ::= <int> "." <int>
<rational> ::= <int> "/" <int>
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

<identifier> ::= <letter> | <letter> <identifier>
<letter> ::=  "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m"
            | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"

<exprlist> ::= <expr> | <expr> "," <exprlist>
<typedexprlist> ::=  <expr> ":" <type>
                   | <expr> ":" <type> "," <typedexprlist>
                   | <expr> "," <typedexprlist>

<type> ::=  "int" | "float" | "bool"
          | "(" <typelist> ")" "->" <type>
          | "[" <type> "]" | "(" <typelist> ")"
          | "{" <recordtypelist> "}"

<recordtypelist> ::= <identifier> ":" <type> | <identifier> ":" <type> "," <recordtypelis

<typelist> ::= <type> | <type> "," <typelist>
```

3.9 Sprint 9: Maths Functions

In this sprint we added a number of maths functions to the language, including `sin`, `cos`, `tan`, `asin`, `acos` and other, including vector operations, and added the ability to plot functions, both built in and user defined. Support for complex numbers was also added in the form `1+2i`. Due to the complication of parsing complex numbers, the type system was relaxed to allow integers to unify with any other number. This was necessary as complex number parsing was often ambiguous as to whether *3i + 2* should return a complex number directly or a sum of a complex number and an integer. We chose the

latter, as it was more intuitive and simpler to implement.

3.9.1 Grammar in BNF

```
<stmtlist> ::=  <stmt>
              | <stmt> <stmtlist>
<stmt> ::=  <expr>
          | <vardecl>
          | <assertion>

<vardecl> ::= "let" <identifier> "=" <expr>
            | "let" <identifier> ":" <type> "=" <expr>

<assertion> ::= "assert" <expr> | "assert" <expr> <string>


<expr> ::=  <term>
          | <term> "+" <expr>
          | <term> "-" <expr>
          | <term> "==" <expr>
          | <term> "!=" <expr>
          | <term> "<" <expr>
          | <term> ">" <expr>
          | <term> "<=" <expr>
          | <term> ">=" <expr>
          | <term> "&&" <expr>
          | <term> "||" <expr>
<term> ::=  <factor>
          | <factor> "*" <term>
          | <factor> "/" <term>
          | <factor> "%" <term>
<factor> ::=  <literal>
            | "(" <expr> ")"
            | <factor> "^" <factor>
            | <unaryop> <factor>
            | <factor> "(" <exprlist> ")"
            | <factor> "." <identifier>
            | <factor> "[" <expr> "]"
            | <factor> "." <identifier>
            | <factor> "[" <expr> ":" <expr> "]"
            | <factor> "[" <expr> ":" "]"
            | <factor> "[" ":" <expr> "]"

<unaryop> ::= "-" | "!" | "+" | <userop>

<literal> ::= <number> | <identifier> | <bool> | <list> | <lambda> | <string> | "()" | <t
<string> ::= '"' <charlist> '"' | '""'
<charlist> ::= <char> | <char> <charlist>
<list> ::= "[" <exprlist> "]"
<tuple> ::= "(" <exprlist> ")"
<record> ::= "{" <recordlist> "}"
<recordlist> ::=  <identifier> "=" <expr>
                | <identifier> "=" <expr> "," <recordlist>
                | <identifer> ":" <type> "=" <expr>
                | <identifier> ":" <type> "=" <expr> "," <recordlist>
```

```
<lambda> ::=  "(" <typedexprlist> ")" "->" <expr>
            | "(" <typedexprlist> ")" ":" <type> "->" <expr>

<bool> ::= "true" | "false"

<number> ::= <int> | <float> | <rational> | <complex>
<int> ::= <digit> | <digit> <int>
<float> ::= <int> "." <int>
<rational> ::= <int> "/" <int>
<complex> ::= <float> "+" <float> "i" | <float> "-" <float> "i" | <float> "i"
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

<identifier> ::= <letter> | <letter> <identifier>
<letter> ::=  "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m"
            | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"

<exprlist> ::= <expr> | <expr> "," <exprlist>
<typedexprlist> ::=  <expr> ":" <type>
                   | <expr> ":" <type> "," <typedexprlist>
                   | <expr> "," <typedexprlist>

<type> ::=  "int" | "float" | "bool"
          | "(" <typelist> ")" "->" <type>
          | "[" <type> "]" | "(" <typelist> ")"
          | "{" <recordtypelist> "}"

<recordtypelist> ::= <identifier> ":" <type> | <identifier> ":" <type> "," <recordtypelis

<typelist> ::= <type> | <type> "," <typelist>
```

3.10 Sprint 10: Control flow

In this sprint we added control flow to the language, with `if` expressions and recursive bindings. If statements were in the form `if` ***expr*** `then` ***expr*** `else` ***expr***, and required that the condition be a boolean and each branch be of the same type. The *else* branch can be omitted if the *then* is of type `Unit`, allowing for simple if expressions such as `if x > 0 then print(x)`. Additionally, if expressions are expressions, meaning they return a value, allowing for more concise code. They are compiled as jump instructions in the bytecode compiler.

Recursive bindings were implemented using a technique called *trampolining*, where a function is defined recursively by passing the function as an argument to itself, allowing for recursion without the need for a stack. This prevents stack overflows in the case of deep recursion, and is a common technique in functional programming.

3.10.1 Grammar in BNF

```
<stmtlist> ::=  <stmt>
              | <stmt> <stmtlist>
<stmt> ::=  <expr>
          | <vardecl>
          | <assertion>

<vardecl> ::= "let" <identifier> "=" <expr>
            | "let" <identifier> ":" <type> "=" <expr>
```

```
            | "let rec" <identifier> "=" <lambda>

<assertion> ::= "assert" <expr> | "assert" <expr> <string>

<expr> ::=  <term>
          | <term> "+" <expr>
          | <term> "-" <expr>
          | <term> "==" <expr>
          | <term> "!=" <expr>
          | <term> "<" <expr>
          | <term> ">" <expr>
          | <term> "<=" <expr>
          | <term> ">=" <expr>
          | <term> "&&" <expr>
          | <term> "||" <expr>
<term> ::=  <factor>
          | <factor> "*" <term>
          | <factor> "/" <term>
          | <factor> "%" <term>
<factor> ::=  <literal>
            | "(" <expr> ")"
            | <factor> "^" <factor>
            | <factor> "(" <exprlist> ")"
            | <factor> "." <identifier>
            | <unaryop> <factor>
            | <factor> "[" <expr> "]"
            | <factor> "[" <expr> ":" <expr> "]"
            | <factor> "[" <expr> ":" "]"
            | <factor> "[" ":" <expr> "]"
            | <if>
            | "{" <stmtlist> "}"

<unaryop> ::= "-" | "!" | "+" | <userop>

<if> ::=  "if" <expr> "then" <expr> "else" <expr>
        | "if" <expr> "then" <expr>

<literal> ::= <number> | <identifier> | <bool> | <list> | <lambda> | <string> | "()" | <t
<string> ::= '"' <charlist> '"' | '""'
<charlist> ::= <char> | <char> <charlist>
<list> ::= "[" <exprlist> "]"
<tuple> ::= "(" <exprlist> ")"
<record> ::= "{" <recordlist> "}"
<recordlist> ::=  <identifier> "=" <expr>
                | <identifier> "=" <expr> "," <recordlist>
                | <identifer> ":" <type> "=" <expr>
                | <identifier> ":" <type> "=" <expr> "," <recordlist>

<lambda> ::=  "(" <typedexprlist> ")" "->" <expr>
            | "(" <typedexprlist> ")" ":" <type> "->" <expr>

<bool> ::= "true" | "false"

<number> ::= <int> | <float> | <rational> | <complex>
<int> ::= <digit> | <digit> <int>
```

```
<float> ::= <int> "." <int>
<rational> ::= <int> "/" <int>
<complex> ::= <float> "+" <float> "i" | <float> "-" <float> "i" | <float> "i"
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

<identifier> ::= <letter> | <letter> <identifier>
<letter> ::=  "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m"
            | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"

<exprlist> ::= <expr> | <expr> "," <exprlist>
<typedexprlist> ::=  <expr> ":" <type>
                   | <expr> ":" <type> "," <typedexprlist>
                   | <expr> "," <typedexprlist>

<type> ::=  "int" | "float" | "bool"
          | "(" <typelist> ")" "->" <type>
          | "[" <type> "]" | "(" <typelist> ")"
          | "{" <recordtypelist> "}"

<recordtypelist> ::= <identifier> ":" <type> | <identifier> ":" <type> "," <recordtypelis

<typelist> ::= <type> | <type> "," <typelist>
```

3.11 Sprint 11: Optimisation

In this sprint we added a simple optimisation pass to the bytecode compiler, which removed unnecessary stack operations and combined constant expressions. Details of this are discussed in section 4.8. The grammar was not changed in this sprint.

3.12 Sprint 12: Transpiler

In this sprint we added the ability to transpile the bytecode to C, which could then be compiled and run as a standalone executable. Details of this are discussed in section 4.16. The grammar was not changed in this sprint.

3.13 Sprint 13: Finalisation

In this sprint we finalised the language, adding a few more functions such as matrix operations like *transpose* and *invert*, and tidied the codebase. The grammar was not changed in this sprint.

Chapter 4

Final deliverable

4.1 Final BNF

```
<stmtlist> ::=  <stmt>
              | <stmt> <stmtlist>
<stmt> ::=  <expr>
          | <vardecl>
          | <assertion>
          | <typealias>

<vardecl> ::= "let" <identifier> "=" <expr>
            | "let" <identifier> ":" <type> "=" <expr>
            | "let rec" <identifier> "=" <lambda>
            | "let async" <identifier> "=" <lambda>

<assertion> ::= "assert" <expr> | "assert" <expr> <string>

<typealias> ::= "type" <identifier> "=" <type>

<expr> ::=  <term>
          | <term> "+" <expr>
          | <term> "-" <expr>
          | <term> "==" <expr>
          | <term> "!=" <expr>
          | <term> "<" <expr>
          | <term> ">" <expr>
          | <term> "<=" <expr>
          | <term> ">=" <expr>
          | <term> "&&" <expr>
          | <term> "||" <expr>

<term> ::=  <factor>
          | <factor> "*" <term>
          | <factor> "/" <term>
          | <factor> "%" <term>
<factor> ::=  <literal>
            | "(" <expr> ")"
            | <factor> "^" <factor>
            | <factor> <userop> <factor>
            | <unaryop> <factor>
            | <factor> "(" <exprlist> ")"
            | <factor> "." <identifier>
            | <factor> "[" <expr> "]"
```

```
            | <factor> "[" <expr> ":" <expr> "]"
            | <factor> "[" <expr> ":" "]"
            | <factor> "[" ":" <expr> "]"
            | <factor> "." <identifier>
            | <factor> ":" <type>
            | <range>
            | <if>
            | "${" <expr> "}"
            | "{" <stmtlist> "}"
            | <cast>

<cast> ::= <expr> ":" <type>

<unaryop> ::= "-" | "!" | "+" | <userop>
<userop> ::= <opchar> | <opchar> <userop>
<opchar> ::=  "!" | "@" | "#" | "$" | "%" | "^" | "&"
            | "*" | "-" | "+" | "=" | "<" | ">" | "?" | ":" | "|" | "~"

<range> ::= "[" <expr> ".." <expr> "]"

<if> ::=  "if" <expr> "then" <expr> "else" <expr>
        | "if" <expr> "then" <expr>
        | <expr> "if" <expr> "else" <expr>

<literal> ::= <number> | <identifier> | <bool> | <list> | <lambda> | <string> | "()" | <t
<string> ::= '"' <charlist> '"' | '"""'
<charlist> ::= <char> | <char> <charlist>
<list> ::= "[" <exprlist> "]"
<tuple> ::= "(" <exprlist> ")"
<record> ::= "{" <recordlist> "}"
<recordlist> ::=  <identifier> "=" <expr>
                | <identifier> "=" <expr> "," <recordlist>
                | <identifer> ":" <type> "=" <expr>
                | <identifier> ":" <type> "=" <expr> "," <recordlist>

<lambda> ::=  "(" <typedexprlist> ")" "->" <expr>
            | "(" <typedexprlist> ")" ":" <type> "->" <expr>
            | "(" <typedexprlist> ")" "{" <stmtlist> "}"
            | "(" <typedexprlist> ")" ":" <type> "{" <stmtlist> "}"

<bool> ::= "true" | "false"

<number> ::= <int> | <float> | <rational> | <complex>
<int> ::= <digit> | <digit> <int>
<float> ::= <int> "." <int>
<rational> ::= <int> "/" <int>
<complex> ::= <float> "+" <float> "i" | <float> "-" <float> "i" | <float> "i"
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

<identifier> ::= <letter> | <letter> <identifier>
<letter> ::=  "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m"
            | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"

<exprlist> ::= <expr> | <expr> "," <exprlist>
<typedexprlist> ::=  <expr> ":" <type>
```

```
                    | <expr> ":" <type> "," <typedexprlist>
                    | <expr> "," <typedexprlist>

<type> ::=  "int" | "float" | "bool"
          | "(" <typelist> ")" "->" <type>
          | "[" <type> "]" | "(" <typelist> ")"
          | "{" <recordtypelist> "}"
          | <identifier>

<recordtypelist> ::= <identifier> ":" <type> | <identifier> ":" <type> "," <recordtypelis

<typelist> ::= <type> | <type> "," <typelist>
```

This BNF is represented in the F# codebase as an AST, represented by the following type:

```
/// <summary>
/// The AST of the language.
/// </summary>
type Expr =
    | ELiteral of Literal * Type
    | EIdentifier of Token * Type option
    | EGrouping of Expr * Type option

    | EIf of Expr * Expr * Expr * Type option
    | ETernary of Expr * Expr * Expr * Type option

    | EList of Expr list * Type option
    | ETuple of Expr list * Type option

    | ECall of Expr * Expr list * Type option

    /// <summary>
    /// Indexing operation on a list or tensor.
    /// Expr (list or tensor), (index), type
    /// Allows for indexing in the form l[1]
    /// </summary>
    | EIndex of Expr * Expr * Type option

    /// <summary>
    /// Indexing with a range operation on a list or tensor.
    /// Expr (list or tensor), start, end, type
    /// Allows for indexing in the form l[..1] or l[1..2] or l[1..]
    /// </summary>
    | EIndexRange of Expr * Expr * Expr * Type option

    /// <summary>
    /// A lambda expression with a list of arguments, a body, a return type, a pure flag,
    /// </summary>
    | ELambda of (Token * Type option) list * Expr * Type option * bool * Type option * b
    | EBlock of Stmt list * bool * Type option // bool is whether block is part of a func
    | ERange of Expr * Expr * Type option

    | ERecordSelect of Expr * Token * Type option

    /// <summary>
```

```
    /// Records represented recursively as a row type.
    /// </summary>
    | ERecordExtend of (Token * Expr * Type option) * Expr * Type option
    | ERecordRestrict of Expr * Token * Type option
    | ERecordEmpty of Type

    /// <summary>
    /// Unevaluated code block.
    /// </summary>
    | ECodeBlock of Expr

    /// <summary>
    /// A tail call (for tail recursion).
    /// </summary>
    | ETail of Expr * Type option

/// <summary>
/// A statement in the language (something that does not return a value).
/// </summary>
and Stmt =
    | SExpression of Expr * Type option
    | SVariableDeclaration of Token * Expr * Type option
    | SAssertStatement of Expr * Expr option * Type option
    | STypeDeclaration of Token * Type * Type option
    | SRecFunc of Token * (Token * Type option) list * Expr * Type option
    | SAsync of Token * (Token * Type option) list * Expr * Type option
    | SImport of Token option * string * bool * Type option // maybe binding name, module
```

4.2 Final GUI

The final GUI is shown in subsections 4.2.1, 4.2.2 and 4.2.3. It has 3 types of windows: the code editor, the notebook view and the plot view. It has support for loading in a file, buttons for running or transpiling the code, and an interactive repl. The notebook view supports importing or exporting from a proprietary file format, running code blocks and exporting to a PDF.

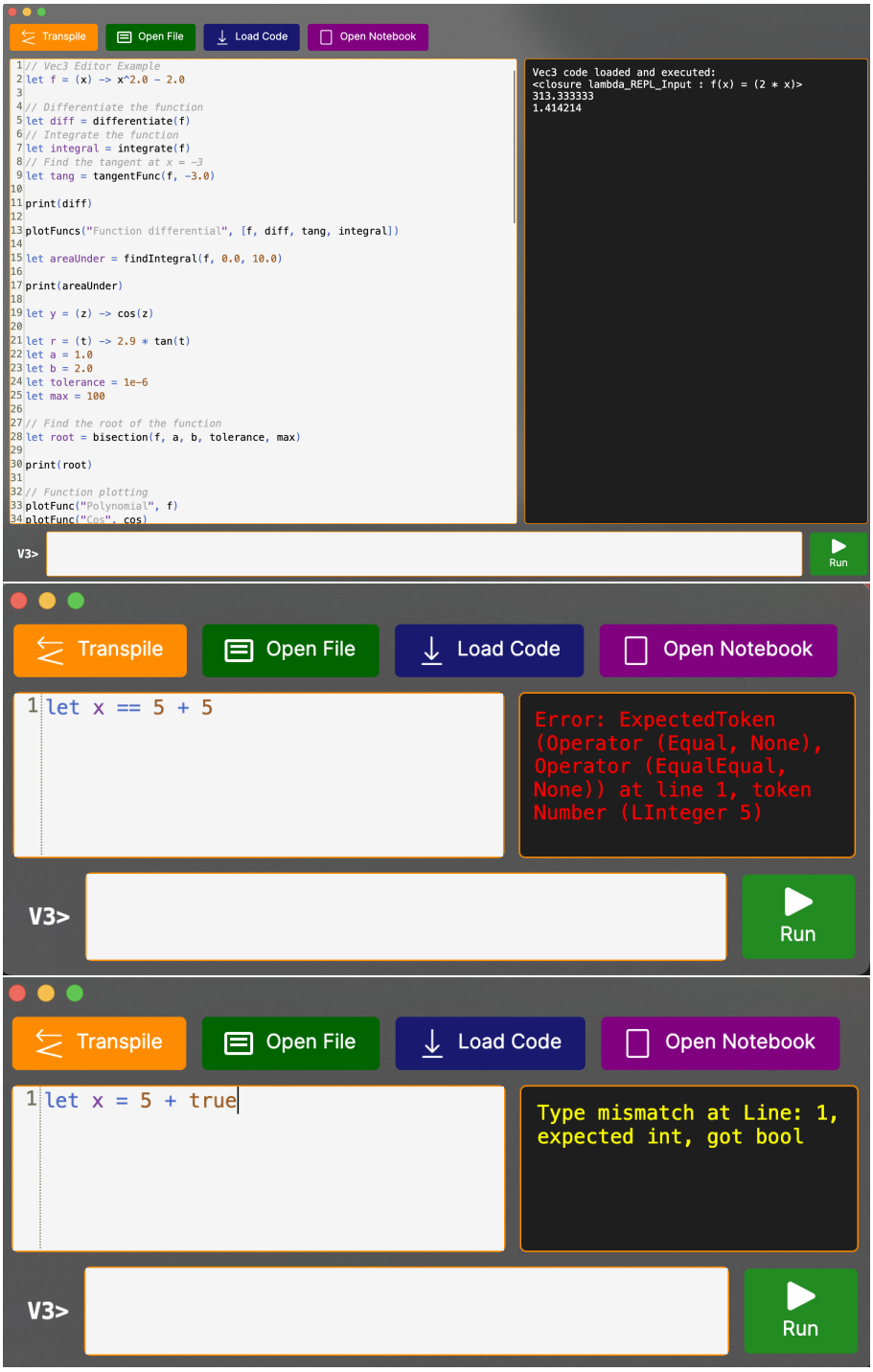


Figure 4.1: Final GUI

4.2.1 Code Editor

The code editor is where the user writes their code, and is the main window of the GUI. It has syntax highlighting, live error checking and a REPL.

Images are given in figure 4.1 of the code editor with output, live parser errors and live type errors.

4.2.2 Notebook View

The notebook view is where the user can write code in a more interactive way, similar to Jupyter notebooks(Project Jupyter, 2023). It has support for importing or exporting from a proprietary file format, running code blocks and exporting to a PDF. An image is shown in figure 4.2. As shown, variable assignment persists between code blocks, the

result of the code block is displayed below the code and plotting functions are supported.

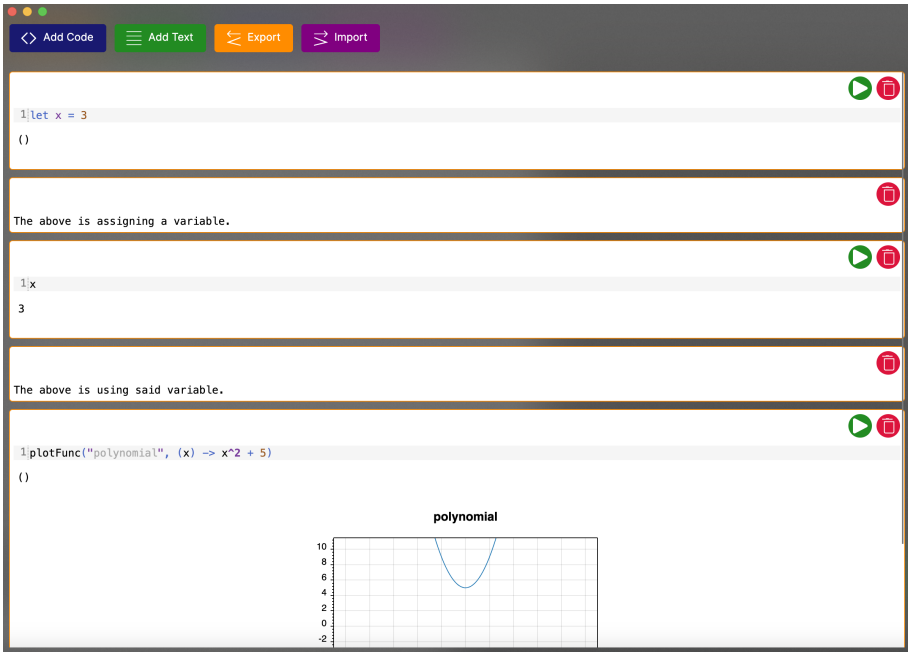


Figure 4.2: Notebook view

An example PDF export from the given image is shown in figure 4.3.

Vec3 Notebook

2024-12-23 19:55

Code

```
let x = 3
```

Output

```
()
```

The above is assigning a variable.

Code

```
x
```

Output

```
3
```

The above is using said variable.

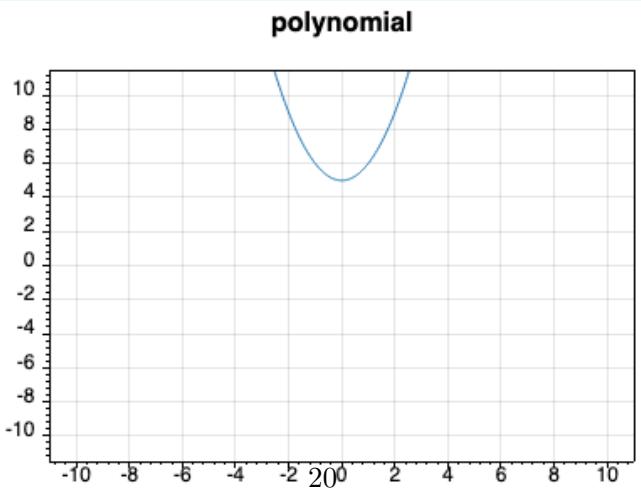
Code

```
plotFunc("polynomial", (x) -> x^2 + 5)
```

Output

```
()
```

Plots



4.2.3 Plot View

The plot view is where the user can view plots generated by the code. It has support for zooming in and out, moving around and adjusting the axes, as well as interactive input for plotting functions. Images and further details are given in section 4.14.

4.3 Notable Features

mintedAlongside standard arithmetic operations, variable assignment, function definitions, and plotting the language also supports the following interesting features:

User defined operators The user can define their own operators, both unary and binary, with the following syntax:

```
// Define a binary operator
let (|+|) = (a: float, b: float) -> a + b

// Define a unary operator
let (|!|) = (a: bool) -> not a
```

This is a powerful feature allowing for the language to be extended in a way that is meaningful to the user.

Static type inference The language uses Hindley-Milner type inference to infer the types of expressions while giving the user the option of specifying types explicitly, improving the correctness of the code and reducing the cognitive load on the user.

```
let f = (x: float) -> x + 1.0
let f: (float) -> float = (x) -> x + 1.0
let f = (x: float) : float -> x + 1.0
```

Compound data types The language supports compound data types such as lists, tuples and records, allowing for more complex data structures to be represented.

```
let l = [1, 2, 3]
let l1 = l[0]

let t = (1, true, "this is a tuple")

let r = { x = 1, y = 2 }
let r1 = r.x
```

Imports The language supports importing other files, allowing for code to be split across multiple files, improving code organisation and reusability. As well as this, a small standard library has been written that can be imported as desired.

```
import "file.vec3"
import "list"

let f = (x) -> func(x) // func defined in file.vec3
let l = [1, 2, 3]
let l1 = head(l) // head defined in list
```

Turing Completeness The language is Turing complete, allowing for the implementation of any algorithm that can be implemented in a Turing complete language. As the language is immutable, this is done through recursion and higher order functions, with control flow managed through if expressions.

```
let rec map = (list, f) -> if len(list) == 0 then [] else f(head(list)) :: map(tail(list)
```

Calculus The language has built-in functions for finding the integral function, derivative function and tangent function at a given point of a function. As such, a few useful math operations are built around these, such as finding the integral:

```
let findIntegral = (f, a, b) {
    let integral = integrate(f)
    integral(b) - integral(a)
}
```

Vector and Matrix operations The language represents vectors and matrices as lists of numeral types, with operations such as addition, subtraction, multiplication and division defined for them, and functions such as *transpose* and *determinant* defined for matrices.

```
let v1 = [1.0, 2.0, 3.0]
let v2 = [4.0, 5.0, 6.0]
let v3 = v1 + v2

let m1 = [[1.0, 2.0], [3.0, 4.0]]
let trans = transpose(m1)
let det = determinant(m1)
let inv = inverse(m1)
```

Casting The language has first class support for casting between types, allowing for the user to explicitly cast between types as desired. A useful example of this is casting integer lists to float lists:

```
let l = [1..10] : [float]
```

Async functions The language supports asynchronous functions, allowing for non-blocking operations to be performed. This is useful is a long calculation is to be performed, as the function will run in the background until awaited.

```
let async longCalc = sum([1..1000000])

let result = await(longCalc)
```

Range Expressions The language supports range expressions, allowing for a list of numbers to be generated easily. In addition to this, list indexing also supports ranges, allowing for a sublist to be extracted from a list.

```
let l = [1..10]
let l1 = l[1..5] // Extracts the sublist [2, 3, 4, 5, 6]
let l2 = l[1..] // Extracts the sublist [2, 3, 4, 5, 6, 7, 8, 9]
let l3 = l[..5] // Extracts the sublist [1, 2, 3, 4, 5]
```

Function Plotting The language has built-in functions for plotting functions, allowing for easy visualisation of functions. This is done through the *plotFunc* function, which takes in a function and plots it on a graph.

Interactive Plotting The plot windows are interactive, with the user being able to zoom in and out, move around and adjust the axes as desired, as well as inputting functions to be plotted on command.

4.4 Lexer

Initial lexer design was based on a simple regular expression based lexer, but this was later replaced with a more functional approach using pattern matching on the input string. The reason for this change was that the regular expression based lexer was difficult to extend and maintain due to the lack of type safety. For example if we had a more general regex called before a more specific one, the more general one would always match first, even if the more specific one should have matched.

This was solved by using a more functional approach, where the type system of F# would inform us if a case would never be matched due to the order of the cases or otherwise, preventing a class of easily overlooked errors during development. The lexer is now implemented as a recursive pattern matching function that takes a string and returns a list of tokens, complete with their lexeme and position in the input string.

Lexer errors are also accumulated in a list of type *LexerError*, which are displayed to the user in the GUI.

Something of note is that the lexer parses numbers itself, rather than passing them to the parser as strings.

Additionally, due to the permittance of user defined operators, the lexer makes special considerations when lexing special characters, as the distinction between a built-in operator (with precedence) and a user defined operator (currently without taking precedence into account) is made during lexing.

Furthermore, both block comments (*/* */*) and line comments (*//*) are handled by the lexer by ignoring the contents of the comment. In future, it may be interesting represent comments as a token in the AST, allowing for systems such as documentation generation or automatic formatting to be implemented.

4.5 Parser

The parser is implemented using Pratt parsing(Pratt, 1973), which is a top-down operator precedence parsing method that allows for easy extension and modification of the grammar. It works by assigning a precedence to each token, as well as functions specifying how to parse the token when encountering it in a prefix, infix or postfix position.

For example, take the expression $2 + 3 * 4$.

The parser would first encounter the number *2*, which has a precedence of 0 and a prefix function that simply returns the number. Thus, the current state of the parser is 2. The parser would then encounter the operator *+*, which has a precedence of 1 and a left associative infix function that takes the left hand side and the right hand side and returns a binary expression node. The parser then attempts to parse the right hand side of the operator with a precendence level higher than the plus operator, as Pratt parsing must ensure that higher precedence operations (such as multiplication) are parsed first. The parser would then encounter the number *3*, which again is treated as a literal and returned. The parser then encounters the operator ***, which has a precedence of 2 (higher than the plus operator) and as such the parser cannot yet resolve the *+* operator; it must handle the higher precedence multiplication operator first. The parser saves the left hand side (the number 3) and then parses the right hand side of the multiplication operator using a precedence level higher than the multiplication operator. It encounters the number *4*, which is returned as a literal. The parser then returns the binary expression node for the multiplication operator, with the left hand side being the number 3 and the right hand side being the number 4. The parser then returns to the plus operator, which can now be

resolved as the left hand side is the number 2 and the right hand side is the result of the multiplication operator.

This is a simple example, but Pratt parsing can handle more complex expressions with ease, such as nested expressions and function calls.

Using Pratt parsing has improved the extensibility of the parser, as adding new operators or changing the grammar is as simple as adding a new case to the parser.

A slight limitation is during ambiguity, such as the *(* symbol, which can be used for a grouping, a lambda definition, a tuple or a *unit* type when encountered in the prefix position. This is resolved through a state machine approach, where the parser can move around the state at will, allowing lookahead and backtracking in order to reach a point where the ambiguity is resolved.

In order to simplify the Virtual Machine**??**, the parser parses all binary and unary operations as function calls, with the operator as the function name.

In order to make type inference simpler for operators that are overloaded for both unary and binary operations (such as the - operator), the operator itself keeps track of the manner in which it is called (unary or binary) and returns the appropriate AST node. This allows for easier type inference (as the names of the overloaded functions are different), and simplifies the bytecode generation process by removing ambiguity in the AST. This idea could possibly be extended to allow other overloaded function names (with varying numbers of arguments or arguments of different types).

Error handling in the parser is done through the use of the monadic *ParserResult* type, which can either return a successful result or an error message, which is then displayed to the user in the GUI allowing for clear and easy diagnosis of errors.

4.6 AST

The AST of the language is represented as a list of statements, where a statement is either expression, a variable assignment or another statement type. It is typed (after type inference4.7) in order to allow for easier optimisation and bytecode generation.

The AST representation is given in section 4.1.

4.7 Type Inference

Vec3 is a statically typed language, with full type inference. The type inference algorithm is based on Hindley-Milner type inference(Sulzmann, 2000), with some modification to support the non-ML style syntax, and extended to support *row polymorphism*(Morris and McKinna, 2019) (4.7.3), *gradual typing*(Garcia et al., 2016) (4.7.2), *recursive bindings* (4.7.4), *vector length encoding* (4.7.6) and a seemingly unique method of supporting ad-hoc polymorphism named *constraints* (4.7.5).

The reason for implementing strong type inference due to the *Semantic Soundness Theory*(Timany et al., 2024), which states that a *well-typed program cannot go wrong.*

Of course this is not strictly true in practice due to external factors, but it is certainly true that strong typing rules out a large class of errors, most of which human, and as such it is a valuable tool for a maths language to have as the user is less likely to make trivial mistakes.

Another thing to note is that in order to make the language more intuitive to use, the **integer** type will coerce into any other number type (**float, rational, complex**). This allows for expressions such as 5.0^5 type checking successfully, with the result being a float.

4.7.1 Type Inference Algorithm

The algorithm used to infer types is based on Algorithm W(Milner, 1978). The general idea is to assign the widest type possible for a given node in the AST, which is generally a *type variable*, which is a type used to represent a type that can be unified with any other type (a generic type). The node's children are then inferred, and the types of the children

are unified with the parent node. If the types cannot be unified, then the program is ill-typed. Unification is the process of finding the most general type that can be assigned to two types, and is a key part of the algorithm.

For example, unifying *int* and *int* would result in *int*, as this is the most general type that can be assigned to both.

Contrasting this, unifying int and a type variable a, would result in *int*, and then the type variable a would have to be substituted with *int* throughout the program (because *int* is the most general type that can be assigned to a). It works bottom-up as only a few types are known at the start, such as the types of literals and the types of built-in functions.

Algorithm Implementation A simplified version of the algorithm, with some details omitted for brevity, is shown in Algorithm 1.

As shown, it is an incredibly simple yet powerful algorithm, and is the basis for many modern type inference algorithms, such as that of F# and OCaml.

Bindings Generally in implementations of *Algorithm W*, after type inference for a given binding has taken place a process known as *generalisation* occurs. This is the process of replacing all type variables in the type of the binding with *forall* quantifiers, which is a way of saying that the type is polymorphic and can therefore be instantiated with any type.

However, this was not necessary in our implementation as we don't specialise bindings during the instantiation of types (such as during calls), we simply infer the type of the call and check it against the type of the binding, so generalisation is not necessary.

4.7.2 Gradual Typing

Gradual typing is a type system that allows for the gradual transition from dynamic typing to static typing. This is useful in a language like Vec3 as it allows for the user to write code without having to worry about types allowing for quick prototyping, but then add types later to ensure correctness. Users have the option of adding types to their code in the form $let\ x : int = 5$, and the type inference algorithm will check that the type of the expression matches the type given.

The type *any* can also be used, which represents a dynamic type that can be unified with any other type. This disables the safety guarantees of the type system, but can be useful as mentioned above for quick prototyping. One thing to note however is that the *any* type is infectious, meaning that if a type is inferred to be *any* then the type of the parent node will also be *any*.

4.7.3 Row Polymorphism

Row polymorphism is a form of polymorphism that allows for the definition of functions that operate on records with a certain set of fields, but can also operate on records with additional fields. It can be considered both a form of structural typing like that of TypeScript(Bierman et al., 2014), and a form of subtyping.

For example, consider the following function:

```
let f = (x) -> x.a
```

This function takes a record with a field a and returns the value of that field.

Now consider the following record:

```
let r = {a = 5, b: int = 6}
```

The function f can be called with r as an argument as r has a field a, and the function will return 5. This is a powerful feature as it allows for the definition of functions that

Algorithm 1 Type Inference Algorithm

```
1: function unify(type1, type2)
2:     if type1 is type variable then
3:         type1 ← type2
4:     end if
5:     if type2 is type variable then
6:         type2 ← type1
7:     end if
8:     if type1 is function type and type2 is function type then
9:         unify paramTypes
10:        unify returnTypes
11:    end if
12:    if type1 is not equal to type2 then
13:        error
14:    end if
15: end function
16: function infer(expr, env)
17:    if expr is literal then
18:        return type of literal
19:    end if
20:    if expr is variable then
21:        T ← lookup variable env
22:        return type
23:    end if
24:    if expr is function call then
25:        funcType ← infer function
26:        argTypes ← infer arguments
27:        funcType ← unify paramTypes argTypes
28:        returnType ← return type of funcType
29:        return returnType
30:    end if
31:    if expr is binding then
32:        bodyType ← infer body
33:        env ← add binding bodyType environment
34:        return bodyType
35:    end if
36:    if expr is lambda then
37:        argTypes ← new type variables
38:        bodyEnv ← add arguments to environment
39:        bodyType ← infer body with bodyEnv
40:        funcType ← argTypes + bodyType
41:        return funcType
42:    end if
43: end function
```

operate on a wide range of records. The reference algorithm given by Morris and McKinna (2019) was used as a basis for the implementation of row polymorphism in Vec3, however without record restriction as it was not necessary for the language.

The algorithm works by assigning a *row variable* to each record type, which is a type variable that represents the fields of the record, where a record is represented in the type system as an extension of another record, or the empty record. The algorithm then unifies the row variables of the record types, and if the unification is successful then the records are considered to be the same type.

A reference implementation of the algorithm is given in Algorithm 2.

Algorithm 2 Row Polymorphism Algorithm

function $unify(type1,\ type2)$
  if *type*1 **is** *row variable* **then**
    $type1 \leftarrow type2$
  end if
  if *type*2 **is** *row variable* **then**
    $type2 \leftarrow type1$
  end if
  if *type*1 **is** *record type and type*2 **is** *record type* **then**
    $unify$ *row variables*
  end if
  if *type*1 **is** not equal to *type*2 **then**
    error
  end if
end function

With some creativity, row polymorphism can be used to represent semi-algebraic data types or tagged unions. For example, consider the built-in *on* function (used to add event listeners for shapes)4.15:

```
on(shape, Keys.Down, (state) -> ...)
```

The function expects a shape reference, a key, and a function that takes a state.

The implementation of the keys record is hidden from the user, but could well be implemented as a record with a field for each key, where each key is a record that contains a field *43hr4h54j3* (a unique identifer for the keys record) and the *on* function could have a type of:

```
let on = (shape, key: { 43hr4h54j3: int }, func) -> ...
```

This has pretty good type safety, as the function will only accept keys with said field, which is hidden from the user. This doesn't have quite as good safety guarantees as a true algebraic data type, i.e. in the form $databool = True|False$, but is certainly safer than say C enums, preventing mistakes such as using incorrect argument order.

4.7.4 Recursive Bindings

Due to the fact that everything is immutable in Vec3, the simplest way to ensure Turing completeness is to allow for recursive bindings (i.e. functions that call themselves). This is a powerful feature as it allows for the definition of functions that operate on recursive data structures, such as trees and lists.

The type inference algorithm was modified to support recursive bindings, as the standard algorithm would not be able to infer the type of a recursive due to the fact that the binding would not be in the environment when the type of the function was inferred (all functions are lambdas, and therefore are not assigned to a binding until after declaration). Hence, recursive functions were introduced as a separate statement in the grammar of the language, and the type inference algorithm was modified to support them by adding the binding to the environment before inferring the type of the function.

4.7.5 Constraints

Due to the restrictiveness of the standard Hindley-Milner type inference algorithm, it is not possible to support ad-hoc polymorphism (i.e. overloading) without some modification. For example, OCaml(OCaml, 2024) does not support ad-hoc polymorphism and instead uses, for example, the *+* operator for integer addition and the *+.* operator for float addition, which is not ideal for this language as it would be unintuitive for a mathematician.

Examples of ML style languages that do support ad-hoc polymorphism are F#, which uses static member functions on types to achieve operator overloading(Microsoft, 2024), and Haskell, which uses type classes(Haskell, 2024b) (constructs that define behaviour for a type, similarly to interfaces in object-oriented languages). The way this issue was solved in Vec3 is by introducing the concept of *Constraint types*, which could be likened to a slightly less powerful version of type classes in Haskell. A constraint is a type that is defined by a type variable, and a function of type *Type -¿ bool*. During unification, if a type is unified with a constraint type, then the function is called with the type, and if it returns true then the unification is successful and the type variable that the constraint holds is unified with the type.

For example, consider the type of the *+* operator:

```
(+) :: Constraint (a, supportsArithmetic) -> Constraint (a, supportsArithmetic) -> a
```

Then, when the operator is used with say two ints, the first constrain would be unified with the type *int* (as the int type passes the *isArithmetic* function), and the type variable *a* would be replaced with *int*. The second int would then be unified with the type *int*, and the unification would be successful. However, if the operator was used with a rational and a float, the first constraint would be unified with the type *rational*, replacing the type variable *a* with *rational*, and the second constraint would be unified with the type rational, which does not unify with float, and so the unification would fail.

This type constrain acts as a normal type, allowing for user defined ah-hoc functions, such as:

```
let double = (x) -> x + x
```

This function can be called with any type that supports arithmetic, and the type inference algorithm will infer the type of the function as *Constraint (a, supportsArithmetic) -¿ a*.

Something else unique is the concept of a *transformation*, which is a function of the constraint type that transforms the type into another type. This was necessary due to functions such as *append*, which appends two lists. Due to the fact that the length of the list is encoded in the type, without a transformation the arguments could only unify if the lists were of the same length, which is an unnecessary restriction. As such, the transformation function is used to transform the dimensions of the first type into a list without dimension restrictions so unity can occur.

A current limitation of this system is that the user cannot define their own constraints, and the only constraints present are those built into the language (such as operators). This is a feature that could be added in the future, but was not necessary for the current implementation of the language.

4.7.6 Vector Length Encoding

Another key feature present in the type system is the encoding of vector lengths.

The type of a vector looks like *Vector of Type * Dims*, where *Type* is the type of the elements, and *Dims* is an integer representing the number of dimensions. This means that a vector *[1,2,3]* is inferred to be of type *Vector of int * 3*. This allows for the type system to catch errors such as adding two vectors of different lengths, or only allowing the *cross product* function to be called on vectors of length 3.

This is a powerful feature as it allows for the type system to catch errors that would otherwise only be caught at runtime with standard type inference. In its current state, it also allows for slight *refinement types*(Freeman and Pfenning, 1991), which are types that are dependent on values, such as the length of a vector.

Examples of this catching an otherwise runtime error is shown in the following code:

```
let a = [1,2,3]
let b = [1,2]
let c = a + b // Error: Vectors must be of the same length

let d = [1,2,3]
let e = d[4] // Error: Index out of bounds
```

The latter example is currently very simple, and only catches out of bound errors during indexing with constant values, but could be extended to catch more complex errors in the future.

Another use case for this is during matrix operations, where the type system can ensure that the dimensions of the matrices are correct, preventing errors such as finding the transpose of a non-square matrix:

```
let a = [[1,2,3], [4,5,6]]
let b = transpose(a) // Error: Matrix must be square
```

Furthermore, the inner tensors of a matrix are also encoded with their dimensions, allowing for the type system to catch errors such as accidentally creating a matrix with rows of different lengths:

```
let a = [[1,2,3], [4,5]] // Error: Rows must be of the same length
```

One thing to note however is that the dimensions of a vector are lost fairly easily, for example during *cons*, as it is not powerful enough to infer the length of the resulting vector.

Having full dependent types would solve this issue, but would be overkill for this language, and would make the type system much more complex (likely requiring a theorem prover and types as values).

4.7.7 Function Purity

The purity of a function is also determined during type inference, with the type of a function being inferred as pure if it is made up of only pure functions, with the base pure functions being built in. This allows for, for example, the *plotFunc* (4.14) ensuring that only pure functions can be passed to it, preventing a user from passing a function that has side effects (which would likely cause a runtime error otherwise). It also allows for easy dead code elimination, as a call to a function that has no side effects can be removed if the result is not used.

4.8 Optimisation

Before compilation, the AST is optimised by removing dead code and constant folding.

4.8.1 Dead code elimination

Dead code elimination is performed on the AST by removing any statements that are not used. For example, if a variable is declared but never used, the variable declaration is removed or if an expression is written but never used, the expression is removed.

This is accomplished by through static analysis of the AST, where the following process is repeated until no more dead code can be removed:

```
while Dead code can be removed do
    for Each node in the AST do
        if Node is a statement then
            if Statement is not used then
                Remove statement
            end if
        else if Node is an expression then
            if Expression is not used then
                Remove expression
            end if
        else if Node is a binding then
            if Variable is not used then
                Remove binding
            end if
        end if
    end for
end while
```

The process is repeated until no more dead code can be removed, allowing for long chains of dead code to be removed (for example if a variable is used in a function that is never called, the function would first be removed and then the variable). It is to be noted that variable assignments are never removed during DCE when running the code editor due to the attached REPL, as the user may wish to use the variable in the REPL, or when running code blocks in the notebook view4.2.2 as the variable may be used in a later code block. However, DCE can be aggressively performed when transpiling to C4.16, as the user is not expected to interact with the generated C code.

4.8.2 Constant folding

Constant folding is performed on the AST by evaluating constant expressions at compile time, such as $2+2 \rightarrow 4$. This is accomplished using the initial interpreter implementation, which recursively evaluates the AST and replaces constant expressions with their evaluated value. Only constants are evaluated, and thus no variable resolution is performed due to the cost of this operation.

4.9 Initial Design of the Bytecode Virtual Machine and Compiler

4.10 Bytecode Compilation

This section details the design and implementation of the bytecode compiler for the new stack-based virtual machine. The compiler translates Vec3 source code, represented as an Abstract Syntax Tree (AST), into a sequence of bytecode instructions optimized for execution on the VM.

4.10.1 Design Principles

The bytecode and compiler are designed with the following principles in mind:

- **Compactness:** A compact instruction set minimizes code size, leading to faster loading and execution. This is achieved by representing many operations, such as mathematical expressions, as function calls, reducing the need for a large number of specialized opcodes.
- **Efficiency:** The stack-based nature of the VM lends itself to efficient execution of common operations, particularly arithmetic and logical expressions.

- **Modularity:** Bytecode is organized into *chunks*, self-contained units that include code, a constant pool, and line number information. This promotes modularity and simplifies debugging.
- **Tail Call Optimization:** The instruction set and compiler are designed to support tail call optimization, allowing for efficient execution of recursive functions without the risk of stack overflow.

4.10.2 Bytecode Architecture

The bytecode consists of a sequence of instructions, each composed of an *opcode* (operation code) followed by zero or more *operands*. The instruction set is designed for a stack machine, with most instructions operating on values stored on an operand stack.

Instruction Set

Table 4.1: Vec3 Virtual Machine Instruction Set

c

Opcode	Operands	Description	Stack Effect
CONSTANT	index (byte)	Load constant from pool	... $\rightarrow$ ..., val
CONSTANT_LONG	index (3 bytes)	Load constant (long index)	... $\rightarrow$ ..., val
POP	—	Remove top value	..., $val \rightarrow$ ...
NIL	—	Push nil	... $\rightarrow$ ..., nil
TRUE	—	Push true	... $\rightarrow$ ..., $true$
FALSE	—	Push false	... $\rightarrow$ ..., $false$
GET_LOCAL	index (byte)	Load local variable	... $\rightarrow$ ..., val
SET_LOCAL	index (byte)	Store to local variable	..., $val \rightarrow$ ...
GET_GLOBAL	index (byte)	Load global variable	... $\rightarrow$ ..., val
SET_GLOBAL	index (byte)	Store to global variable	..., $val \rightarrow$ ...
DEFINE_GLOBAL	index (byte)	Define global variable	..., $val \rightarrow$ ...
GET_UPVALUE	index (byte)	Load upvalue	... $\rightarrow$ ..., val
SET_UPVALUE	index (byte)	Store to upvalue	..., $val \rightarrow$ ...
JUMP	offset (2 bytes)	Jump forward	... $\rightarrow$ ...
JUMP_IF_FALSE	offset (2 bytes)	Jump if false	..., $val \rightarrow$ ...
LOOP	offset (2 bytes)	Jump backward	... $\rightarrow$ ...
CALL	count, r (bytes)	Call function with args	..., $fn, a_1, ..., a_n \rightarrow$ ..., ret
RETURN	b (byte)	Return from function	..., $val \rightarrow$ ...
CLOSURE	index (byte)	Create closure	... $\rightarrow$ ..., $closure$
CLOSE_UPVALUE	—	Close upvalue	..., $upvalue \rightarrow$ ...
COMPOUND_CREATE	—	Create compound value	... $\rightarrow$ ..., $compound$

Note: Stack effects are depicted from left to right, with '...' representing the remaining stack contents.

Chunk Structure

Bytecode is organized into *chunks*. Each chunk, represented by the `Chunk` data structure in the F# code, encapsulates the following:

- **Code**: A `ResizeArray<byte>` containing the sequence of bytecode instructions.
- **Constant Pool**: A `ResizeArray<Value>` storing constants used within the chunk, such as numbers, strings, and function references. This allows for constant deduplication and efficient storage.
- **Line Numbers**: A `ResizeArray<int>` that maps bytecode instruction offsets to their corresponding line numbers in the original source code. This mapping is crucial for debugging and generating informative error messages.

Functions are represented by the `Function` type, which includes the function's arity, name, its associated chunk, and a list of locals.

4.10.3 Compiler Implementation

The compiler, implemented in F, performs a recursive descent traversal of the AST, generating bytecode instructions corresponding to each node.

Compiler State

The compiler maintains state throughout the compilation process using the `CompilerState` record:

4.10.4 Compiler Implementation

The compiler, implemented in F, performs a recursive descent traversal of the AST, generating bytecode instructions corresponding to each node.

Compiler State

The compiler maintains state throughout the compilation process using the `CompilerState` record: F

type CompilerState = CurrentFunction: Closure CurrentLine: int ScopeDepth: int LocalCount: int

- `CurrentFunction`: Represents the function currently being compiled.
- `CurrentLine`: Tracks the current line number in the source code for error reporting.
- `ScopeDepth`: Indicates the current nesting level, used for managing variable scopes.
- `LocalCount`: Keeps track of the number of local variables within the current scope.

The compiler uses a monadic approach, threading the CompilerState through the compilation process. The CompilerResult type, defined as a Result type, is used to handle potential errors during compilation. The use of a monadic approach with CompilerResult offers several advantages. Firstly, it avoids mutable state variables, making the compiler more modular and easier to reason about. Secondly, it enhances testability by explicitly passing the state as an argument. Finally, it provides type-safe error handling.

Expression Compilation

The `compileExpr` function recursively compiles expressions into bytecode. Here are a few examples:

Literals: Literal values (numbers, strings, booleans) are added to the chunk's constant pool, and a `CONSTANT` or `CONSTANT_LONG` instruction is emitted to push the constant onto the stack. Identifiers: Variable references are compiled into `GET_LOCAL`, `GET_GLOBAL`, or `GET_UPVALUE` instructions, depending on the variable's scope. Function Calls: Function calls are compiled into a sequence of instructions that push the function and its arguments onto the stack, followed by a `CALL` instruction. Lambda Expressions: Compiling a lambda expression involves creating a new `Function` object for the lambda, compiling its body

into the new chunk, and then emitting a `CLOSURE` instruction to create a closure object at runtime. The `CLOSURE` instruction also captures any upvalues (variables from enclosing scopes). Lists and Tuples: Lists and tuples are created by first pushing each element of the list or tuple onto the stack in order, then pushing the length of the list or tuple, then pushing an empty list or tuple onto the stack, and finally executing the `COMPOUND_CREATE` instruction to create the data structure.

Statement Compilation

The `compileStmt` function handles the compilation of statements.

Variable Declarations: Variable declarations are compiled into `DEFINE_GLOBAL` instructions for global variables or by allocating a slot on the stack for local variables. Expression Statements: Expression statements are compiled by first compiling the expression and then emitting a `POP` instruction to discard the result if it's not used. Control Flow: if and while statements are compiled using `JUMP_IF_FALSE` and `JUMP` instructions to control the flow of execution.

4.10.5 Compilation Output: The Function Object

The compilation process culminates in the creation of a `Function` object. This object serves as a self-contained executable unit, encapsulating the generated bytecode along with essential metadata. The structure of the `Function` object is as follows:

1. **Arity**: An integer specifying the number of arguments expected by the function.
2. **Chunk**: This component, detailed in Section 4.10, houses the compiled bytecode instructions, the constant pool, and line number information crucial for debugging.
3. **Name**: A string holding the function's name. While not used during execution, it aids in debugging and provides context when examining compiled code.
4. **Locals**: A list of the function's local variables, primarily for debugging purposes.

The `Function` object effectively represents the entry point for execution within the VM. The VM can readily load this object, instantiate a new `Closure` (a wrapper around the `Function` that also holds references to any necessary upvalues), and subsequently initialize a `CallFrame` to commence execution. The ‘compileProgram‘ function is the main driver of the compilation pipeline. It accepts the root of the AST as input and orchestrates the generation of the final ‘Function‘ object. The process can be summarized as follows:

1. **Initialization**: A new, empty `Function` object is created. A default name, such as "REPL_Input", is typically assigned to this top-level function. An initial `CompilerState` is also created to track the compilation context.
2. **Recursive Compilation**: The compiler recursively traverses the AST. For each statement encountered, the ‘compileStmt‘ function is invoked, which, in turn, may call ‘compileExpr‘ for expressions. This recursive process generates bytecode instructions that are sequentially added to the `Chunk` within the `Function` object.
3. **Finalization**: Upon completing the AST traversal, a `RETURN` instruction is appended to the bytecode sequence, ensuring proper function termination. The completed `Function` object is then returned as the result of the compilation process.

Once the `Function` object is generated, it can be seamlessly loaded into the VM for execution. The VM provides a dedicated ‘loadFunction‘ function to facilitate this. This function performs the following actions:

1. **Closure Creation**: A new `Closure` object is created based on the provided `Function`.
2. **Call Frame Initialization**: A new `CallFrame` is initialized. The instruction pointer (`IP`) is set to 0, pointing to the beginning of the function's bytecode. The `StackBase` is set to the current top of the operand stack, providing the function with its own dedicated stack space.
3. **Stack Push**: The newly created `CallFrame` is pushed onto the VM's call stack, making it the active frame.

Following these steps, the VM can initiate execution by invoking the ‘runLoop‘ function. This method of packaging compiled code into `Function` objects offers several advantages:

1. **Modularity**: The compiled code is encapsulated within a well-defined structure, clearly separated from the internal workings of the compiler.
2. **Flexibility**: The VM is designed to load and execute any valid `Function` object, irrespective of its origin. This opens up possibilities for features such as dynamic code loading and separate compilation.
3. **Simplicity**: The process of loading and executing compiled code within the VM is streamlined and straightforward.

In essence, this mechanism allows the Vec3 interpreter to treat compiled code as first-class executable units, enabling dynamic loading and execution of Vec3 programs within a running VM instance. This capability is fundamental to providing a flexible and interactive development environment.

4.11 Virtual Machine Implementation

The Vec3 virtual machine executes bytecode through a straightforward execution model centered around a stack and call frames. At its heart is a simple loop that fetches, decodes, and executes instructions one at a time, maintaining program state through a carefully designed stack structure.

4.11.1 Core Execution Loop

The VM's main execution loop operates on a series of call frames. Each frame represents a function call and contains its own instruction pointer and stack base. The loop continually fetches the next instruction from the current frame, executes it, and updates the VM state accordingly:

```
let rec runLoop (vm: VM) =
    let frame = getCurrentFrame vm
    let vm, instruction = readByte vm
    let opcode = byteToOpCode instruction
    match executeOpcode vm opcode with
    | Return value ->
        push vm value |> continue
    | Call(args, recursive) ->
        callValue vm args recursive |> continue
    | Continue ->
        continue vm
```

The simplicity of our main execution loop isn't just about clean code - it directly impacts the VM's performance. Since this loop executes for every single instruction in our program, its efficiency is crucial. Each extra check or operation in this loop would be multiplied across millions of instruction executions. Consider what happens when the VM executes a simple addition operation:

```
let rec runLoop (vm: VM) =
let frame = getCurrentFrame vm
let vm, instruction = readByte vm  // Just one byte read
let opcode = byteToOpCode instruction  // Simple array lookup
match executeOpcode vm opcode with    // Direct dispatch
| Return value ->
push vm value |> continue
| Call(args, recursive) ->
callValue vm args recursive |> continue
| Continue ->
continue vm
```

Each instruction follows this exact path: read a byte, convert it to an opcode (a constant time array lookup), and dispatch to the specific handler. No extra checks, no special cases, no branching logic. This matters because modern CPUs can efficiently predict and pipeline these consistent operations. This is why we implement operations like addition as function calls rather than dedicated instructions. While it might seem counterintuitive to turn a simple addition into a function call, this tradeoff means our main execution loop stays fast for all instructions. The small performance cost of occasional function calls is more than offset by the efficiency gained in the core loop that handles every single instruction. The real elegance here is how this design cascades into other benefits. A simple execution loop means fewer places for bugs to hide, easier performance profiling, and more predictable behavior under different workloads. Sometimes the fastest solution is also the simplest one.

4.11.2 Stack Management

The stack serves two purposes: storing temporary values during expression evaluation and holding local variables for function calls. Each function call creates a new frame that marks where its locals begin on the stack. This unified approach means accessing both temporary values and local variables is just array indexing - there's no need for separate variable storage or complex lookup mechanisms. Local variables are accessed through offsets from the frame's base pointer. When executing code like:

```
let x = 10
let y = x + 5
```

The VM pushes 10 onto the stack and stores its position as variable 'x'. When later accessing 'x', it simply reads from that stack position using the frame's base pointer plus the variable's index.

4.11.3 Function Calls and Returns

Function calls demonstrate how all parts of the VM work together. When making a call, the VM:

```
let callValue vm argCount recursive =
    let callee = peek vm argCount  // Get function being called
    let stackBase = vm.Stack.Count - argCount
    let frame = {
        Closure = createClosure callee
        IP = 0
        StackBase = stackBase
    }
    vm.Frames.Add(frame)
```

The new frame's stack base points just past the arguments, setting up the local variable area for the called function. This frame structure means each function has its own view of the stack, with its arguments and locals neatly organized. When a function returns, the VM pops off its frame and any temporary values, preserving only the return value for the caller. This disciplined stack management ensures memory usage stays predictable and bounded.

4.11.4 Tail Call Support

The VM handles tail calls by reusing stack frames rather than allocating new ones. When the compiler identifies a tail call, it sets a flag that tells the VM to reuse the current frame:

```
| Call(args, true) ->  // Tail call
    let stackBase = vm.Stack.Count - args - 1
    let frame = createFrame func stackBase
    vm.Frames.Add(frame)
    executeTailCall vm
```

This means deeply recursive code runs in constant stack space, enabling functional programming patterns without risk of stack overflow.

4.11.5 Closures and Upvalues

To support lexical scoping with nested functions, the Vec3 VM implements closures using a mechanism similar to Lua. A *closure* is a combination of a function and its surrounding environment, captured at the time of the function's definition. This environment is represented by *upvalues*. When a function is defined within another function, and it references variables from the enclosing function's scope, those variables become upvalues for the inner function. The compiler identifies these upvalues and generates appropriate bytecode instructions (‘GET_UPVALUE‘, ‘SET_UPVALUE‘, ‘CLOSURE‘, ‘CLOSE_UPVALUE‘) to manage them. When a closure is created at runtime (via the ‘CLOSURE‘ instruction), the VM creates ‘UpValue‘ objects that point to the relevant variables on the stack. If the enclosing function returns, and the upvalues are no longer on the stack, the ‘CLOSE_UPVALUE‘ instruction is executed, which moves the upvalue's value to the heap, ensuring the closure still has access to its environment even after the outer function has completed. This upvalue mechanism allows inner functions to retain access to the variables of their enclosing scope, even after the outer function has returned. This is essential for implementing higher-order functions and other functional programming paradigms. The use of upvalues and closures adds a layer of complexity to the VM, but it significantly enhances the expressive power of the Vec3 language. By carefully managing the lifetime and accessibility of upvalues, the VM ensures that closures behave correctly and maintain the integrity of lexical scoping.

4.11.6 Built-in Functions

The VM integrates built-in functions through a special value type that can directly manipulate VM state. This enables powerful features like symbolic differentiation and plotting while maintaining a clean interface. Built-ins can access the stack and VM state but must follow the same calling conventions as normal functions:

```
VBuiltin(fun args ->
    match args with
    | [VClosure(_, Some f)] ->
        let diff = differentiate f
        let expr = toExpr diff
        compiledAndRun expr
    | _ -> raise "Type error")
```

The VM executes built-ins directly rather than creating a new frame, but their results flow through the same stack mechanism as regular functions.

4.12 Conclusion

The Vec3 VM achieves its goals through careful attention to fundamentals: a clean execution loop, unified stack management, and consistent handling of function calls. By keeping each component focused and well-defined, we create a VM that's both efficient and maintainable. The design proves that a straightforward approach, well-executed, can handle sophisticated language features without undue complexity.

4.13 Prelude

A prelude is implicitly included in every program, which contains some useful functions defined in the language, as well as wrappers for the built-in functions of the Virtual Machine, such as *cos*, *log*, etc.

Notable functions include:

- *map*, *fold* and *filter* functions for lists.
- *range* function for generating a list of numbers.
- *sqrt*, *cubeRoot* which are specialisations of the *root* function.
- *head*, *tail* and *len* functions for lists.
- *findIntegral* function for finding the integral of a function.

We felt it was useful implementing these in-language functions as it allows for more concise and readable code, as well as showcasing the power of the language.

4.14 Plotting

The plotting system is implemented using ScottPlot(ScottPlot, 2024), a plotting library for .NET.

The functionality is exposed to the user through 3 built-in functions: *plot*, *plotFunc* and *plotFuncs*.

plot takes in a record of configuration options of the following type:

```
type PlotConfig = {
    title: string,
    x: [float],
    y: [float],
    ptype: "bar" | "scatter" | "signal",
}
```

The resulting plot is then displayed in a separate window based on these configuration options.

Here are some examples of the *plot* function in use:

```
let x = [1..10] : [float]
let y = map(x, (x) -> x^2)
let data = {
    title = "Example Plot",
    x = x,
    y = y,
    ptype = "scatter"
}
plot(data)
```

Image 4.4 shows the resulting plot.

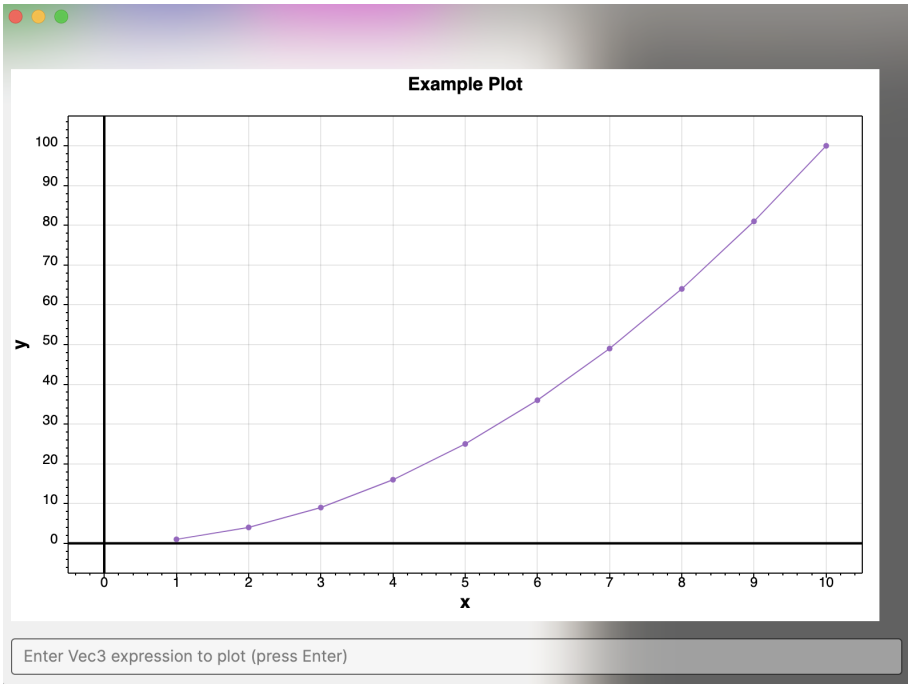


Figure 4.4: Scatter plot

The *ptype* option allows for the user to specify the type of plot, with the options being *bar*, *scatter* and *signal*.

The *bar* type is useful for visualising data, the *scatter* type is useful for plotting functions and the *signal* type is useful for plotting signals. An example of a bar plot is shown in image 4.5.

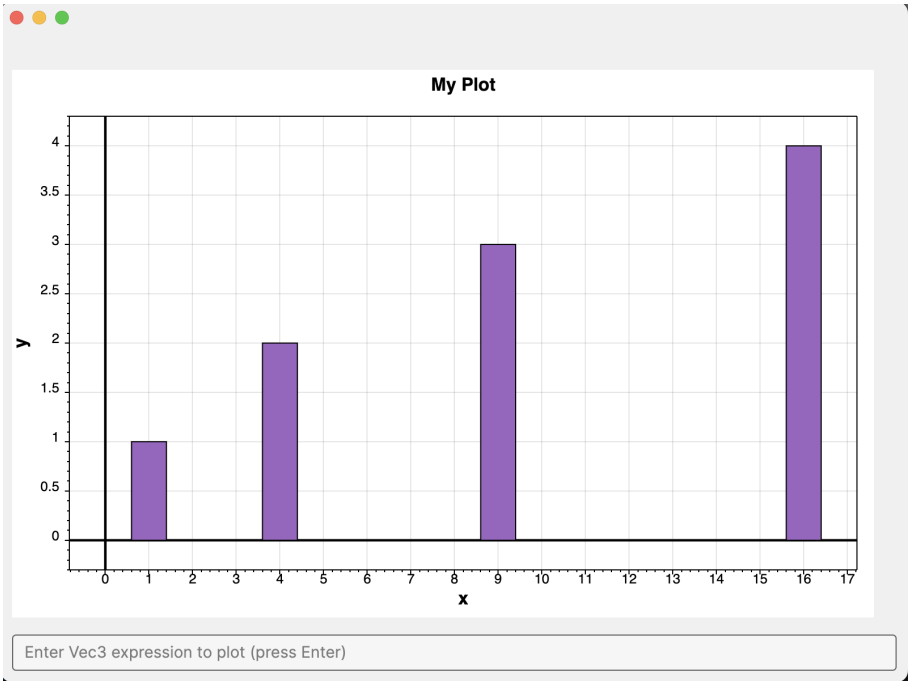


Figure 4.5: Bar plot

The *plotFunc* function takes in a string title and a pure function of type *float -¿ float*. The function is then plotted on the graph with an infinite range of x values. Optionally, the user can also specify two more float values, *start* and *end*, in which case the integral of the function is calculated and displayed on the graph.

For example, given the following snippet:

```
let polynomial = (x) -> x^2 + 2.0 * x + 1.0
plotFunc("Polynomial", polynomial)
```

Image 4.6 shows the resulting plot.

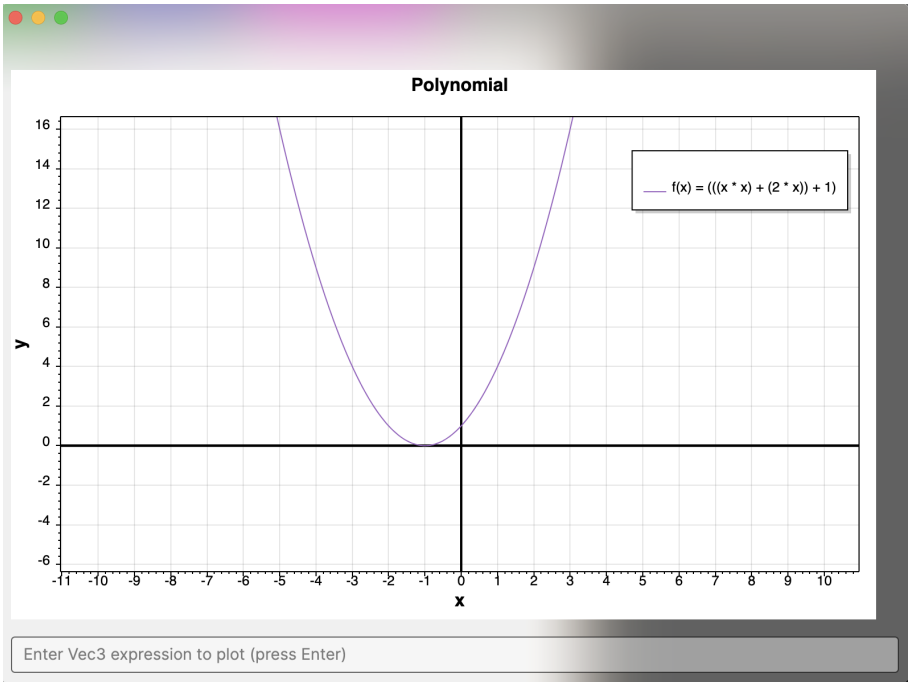


Figure 4.6: Polynomial plot

A point of interest is that for both the *plotFunc* and *plotFuncs* functions, a legend is provided with a textual representation of the function being plotted. This is particularly useful when plotting multiple functions on the same graph, or when dynamically plotting functions (or their derivatives/integrals) based on user input.

The *plotFuncs* function takes in a string title and a list of pure functions of type *float -¿ float*. This allows for multiple plots to be placed on the same window, which we felt was valuable for comparing functions or plotting derivatives.

For example, given the following snippet:

```
let polynomial = (x) -> x^2 + 2.0 * x + 1.0
let derivate = differentiate(polynomial) // find the derivative of the polynomial
let integrand = integrate(polynomial) // find the integral of the polynomial
let tangentFunc = tangentFunc(polynomial, 2.0) // find the tangent function at x = 2.0

plotFuncs("Polynomial", [polynomial, derivate, integrand, tangentFunc])
```

Image 4.7 shows the resulting plot.

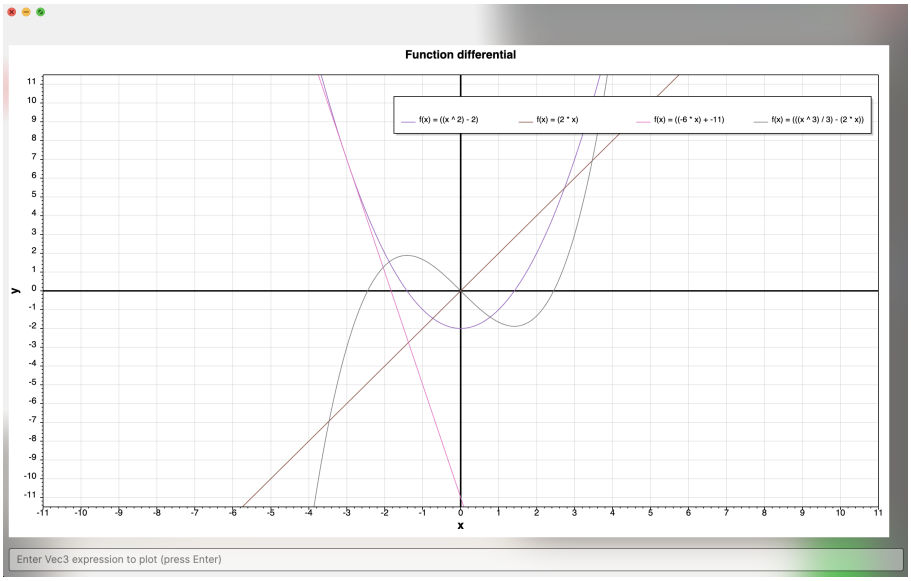


Figure 4.7: Multiple plots

The plots are very interactive, with the user being able to zoom in and out, move around and adjust the axes as desired.

The plot windows also have an input at the bottom, which allows for the user to input a function and have it plotted on command4.8. This is useful for quick visualisation of functions, and allows for a more interactive experience. Note that variable resolution is performed on the input, allowing for the user to use variables defined in the program in the input. Additionally, the legend updates dynamically.

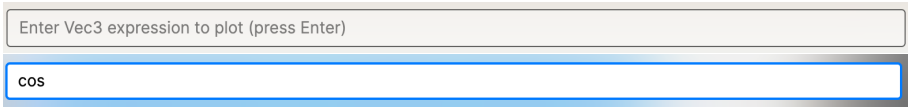


Figure 4.8: Plot input

4.15 Drawing

As well as plotting, the user also has the option of drawing arbitrary shapes on a canvas, and attaching event listeners to them. This is done by means of the *draw* function, which takes in a record of configuration options of the following type:

```
type DrawConfig = {
    x: float,
    y: float,
    width: float,
    height: float,
    color: string,
    shape: "rectangle" | "circle",
    trace?: bool,
}
```

Or a list of the above record type, allowing for multiple shapes to be drawn on the same canvas.

For example, the following will draw a few circles on the canvas:

```
let circle1 = {
    x = 100.0,
    y = 100.0,
```

```
    width = 50.0,
    height = 50.0,
    colour = "red",
    trace = true
}
let circle2 = {
    x = 200.0,
    y = 200.0,
    width = 50.0,
    height = 50.0,
    colour = "blue",
    trace = true
}
draw([circle1, circle2])
```

The result of the above code is shown in image 4.9.

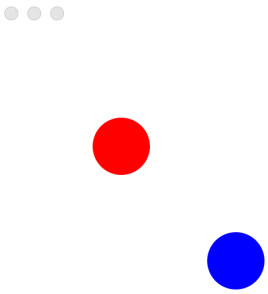


Figure 4.9: Draw circles

The *draw* function then returns a unique identifier for the shape, which can be used to attach event listeners, allowing for movement of the shape through key presses.

The following example attached event listeners to a shape which moves it left and right following the *cos* curve:

```
on(id, Keys.Right, (state) -> { x = state.x + 10.0, y = cos(state.x) * 10.0 + 100.0 })
on(id, Keys.Left, (state) -> { x = state.x - 10.0, y = cos(state.x) * 10.0 + 100.0 })
```

Where keys is a record defined in the prelude of the language (see section 4.13).

Additionally, the *trace* option allows for the shape to leave a trail behind it, which can be useful for animations or visualising movement.

4.16 Transpiler

The user also has the option of transpiling their code to C, which can then be compiled and run as a standalone executable, allowing for faster execution of the code which is important for larger or more computationally intensive programs.

TODO

4.17 Code architecture

The solution is split into 3 separate projects: the GUI, Tests and the main interpreter / compiler project.

The GUI is where the project runs from, and is a standard Avalonia project, with a main window (the code editor), and then some other windows for the plots, drawing and notebook view, as well as some non-UI files such as the a helper module for PDF exports. It also contains a directory containing the prelude and standard library of the language, which are then imported as desired (except the prelude, which is always implicitly imported). The frameworks are libraries used in the GUI project include AvaloniaUI, AvaloniaEdit, FSharp.Core, ScottPlot, TextMate and QuestPDF.

The Tests project mirrors the main project, with a similar structure and file naming for clear and sensible organisation. It uses the NUnit framework for testing, and tests the lexer, parser, compiler and type inference.

The main project is where the main interpreter and compiler are implemented, and is split into the following submodules:

- *Syntax Analysis* contains the lexer and parser, as well as the AST of the language.
- *Type Analysis* contains the type inference engine.
- *Backend* contains the compiler and virtual machine.
- *Optimisation* contains the optimisation passes (dead code elimination and constant folding).

A couple more files, such as the REPL, which contains helper functions for compilation and running of code, and the Symbolic Expression modules, which contain helper functions for maths operations such as simplification of formulas and calculation of derivatives, are also present. No external libraries are used in the main project, as it is a standalone F# project.

We felt this structure was sensible, as it allowed for clear separation of concerns and easy navigation of the codebase.

Chapter 5

Discussion, conclusion and future work

5.1 Discussion and Conclusion

Everything that we set out to do has been completed to a high standard, and more. The language is Turing complete, has a GUI, plotting, static type inference, first class functions, compound data types, recursion, error handling, a transpiler to C, and a standard library. It also has importing, control flow, async functions, and a large number of built-in functions, as well as support for rational and complex numbers.

We feel that the project has been a success, and we are proud of what we have achieved.

5.2 Future Work

An infinite number of more maths functions could be added to the language, such as more matrix operations, more trigonometric functions, and more complex number operations, such as plotting of complex numbers. Additionally, multi-parameter function plotting would be a useful feature, allowing for plotting of custom circles and ellipses, for example.

However, the language is to a standard wherein lots of these functions can be implemented in the language itself, by the user for example, so adding more built-in functions is something that can be done externally and incrementally.

A more IDE like experience, comparable to another maths language like R or MATLAB would be useful, as it would allow users a more complete experience, outside of prototyping. Obviously we have imports, both standard library and user made files, but no way to save progress or set a working directory, so this would be a useful feature to add.

Bibliography

Avalonia (2024). Avalonia website. https://avaloniaui.net/.

Bierman, G., Abadi, M., and Torgersen, M. (2014). Understanding typescript. In *ECOOP 2014–Object-Oriented Programming: 28th European Conference, Uppsala, Sweden, July 28–August 1, 2014. Proceedings 28*, pages 257–281. Springer.

BMITC (2022). Symbolic calculus in f#. https://bmitc.me/articles/symbolic-expressions-in-fsharp.

Desmos Studio PBC (2023). Desmos website. https://desmos.com [Accessed: 30/11/2023].

Freeman, T. and Pfenning, F. (1991). Refinement types for ml. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 268–277.

Garcia, R., Clark, A. M., and Tanter, É. (2016). Abstracting gradual typing. *ACM SIGPLAN Notices*, 51(1):429–442.

Haskell (2024a). Haskell. https://www.haskell.org/.

Haskell (2024b). Haskell documentation. https://www.haskell.org/documentation/.

MathWorks (2024). Matlab. https://www.mathworks.com/products/matlab.html.

Microsoft (2024). F# documentation. https://docs.microsoft.com/en-us/dotnet/fsharp/.

Milner, R. (1978). A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375.

Morris, J. G. and McKinna, J. (2019). Abstracting extensible data types: or, rows by any other name. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–28.

Nonsense, A. (2013). Symbolic calculus in haskell. http://5outh.blogspot.com/2013/05/symbolic-calculus-in-haskell.html.

OCaml (2024). Ocaml documentation. https://ocaml.org/docs/.

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

Project Jupyter (2023). Jupyter website. https://jupyter.org/.

Python (2024). Python. https://www.python.org/.

R (2024). R. https://www.r-project.org/.

ScottPlot (2024). Scottplot website. https://scottplot.net/.

Sulzmann, M. F. (2000). *A general framework for Hindley/Milner type systems with constraints*. Yale University.

SymPy (2024). Symbolic calculus in python. `https://www.sympy.org/en/index.html`.

Timany, A., Krebbers, R., Dreyer, D., and Birkedal, L. (2024). A logical approach to type soundness. *Journal of the ACM*, 71(6):1–75.

Appendix A

Contributions

50/50

A.1 Jacob Edwards

- Lexer
- Parser
- Compiler
- Type inference
- Testing
- Documentation

A.2 Jamie Wales

- GUI
- Plotting
- Transpiler
- Testing
- Documentation
- Parser
- Compiler
- Lexer

Appendix B

Testing

The language has been tested thoroughly using .Net's built in unit testing framework, NUnit. Unit tests have been written for the lexer, parser, compiler, type inference, compiler and transpiler, and are within the Test project.

Additionally, tests have been written in the language itself to test the language's features.

B.1 Arithmetic Expression Testing

Tests can be found in table B.1.

Other tests have been written in the language itself. An example is shown below:

```
// arithmetic tests
assert 1 * 1 == 1, "Multiplication failed"
assert 1 / 1 == 1, "Division failed"
assert 1 % 1 == 0, "Modulus failed"
assert 1 ^ 1 == 1, "Exponentiation failed"
assert 1 == 1, "Equality failed"
```

B.2 Lexer testing

Tests for the lexer are written using NUnit and test the lexer's ability to correctly tokenize the input string. They can be found in the source code or in the documentation.

B.3 Parser testing

Tests for the parser are written using NUnit and test the parser's ability to correctly parse the input string. They can be found in the source code or in the documentation.

B.4 Variable assignment testing

Tests can be found in table B.2.

More variable testing are written in the language itself. An example is shown below:

```
// variable tests
let x = 1
assert x == 1, "Assignment failed"
assert x^2 == 1, "Assignment with expression failed"
assert x^2 + 1 == 2, "Assignment with expression failed"
```

B.5 Function testing

Tests can be found in table B.3.

Further tests are written in the language itself. An example is shown below:

```
// function tests
let f = (x) -> x^2
assert f(3) == 9, "Function failed"
assert f(3) + f(2) == 13, "Function with expression failed"
assert f(3) + f(2) == 13, "Function with expression failed"
```

B.6 Type inference testing

Type inference tests can be found in table B.4.

B.7 Compound data type testing

Tests can be found in table B.5.

B.8 Control flow testing

Tests can be found in table B.6.

B.9 GUI testing

B.10 Plot testing

Table B.1: Arithmetic expression tests. Note that floating pointing values are accurate to three decimal places for the fractional part. ResE is expected result and ResA is actual result.

Expression	ResE	ResA	Pass/Fail	Action/comment
$5 * 3 + (2 * 3 - 2)/2 + 6$	23	23	Pass	BIDMAS
$9 - 3 - 2$	4	4	Pass	left assoc.
$10/3$	3	3	Pass	int division
$10/3.0$	3.333	3.333	Pass	float division
$10\%3$	1	1	Pass	Modulus
$10 - -2$	12	12	Pass	unary minus
$-2 + 10$	8	*	Pass	Unary minus
$3 * 5\hat{}(-1 + 3) - 2\hat{}2 * -3$	87	87	Pass	power test
$-3\hat{}2$	-9(*) or 9	9	Pass	precedence
$-7\%3$	2(*) or -1	-1	Pass	precedence (*)Python
$2 * 3^2$	18	18	Pass	precedence pow ¿ mult
$3 * 5\hat{}(-1 + 3) - 2\hat{} - 2 * -3$	75.750 or 75	75	Pass	Complex expression
$3 * 5\hat{}(-1 + 3) - 2.0\hat{} - 2 * -3$	75.750	-75.750		
$(((3 * 2 - -2)))$	8	8	Pass	
$(((3 * 2 - -2))$	Error	Error: Expected "argument or '(' after call"		syntax error
$-((3 * 5 - 2 * 3))$	-9	-9	Pass	minus expression
$1/2$	1/2	1/2	Pass	Rational number
$1/2 + 1/2$	1/1	1/1	Pass	Rational number
$1/2 + 1/3$	5/6	5/6	Pass	Rational number
$1/2 * 4/7$	2/7	2/7	Pass	Rational number
$4/5 * 2/3 + 1/2/3/4$	6/5	6/5	Pass	Rational number expressions
$1/2 * 4 + 3 - 1/3$	14/3	14/3	Pass	Rational number with integer
$1 + 2i$	1+2i	1+2i	Pass	Complex number
$1 + 2i + 3 + 4i$	4+6i	4+6i	Pass	Complex number
$1 + 2i * 3 + 4i$	1+10i	1+10i	Pass	Complex number

Table B.2: Variable expression tests. Note that floating pointing values are accurate to three decimal places for the fractional part. ResE is expected result and ResA is actual result.

Expression	ResE	ResA	Pass/Fail	Action/comment
$let\ x = 3$ $(2 * x) - x\hat{}2 * 5$	-39	-39	Pass	var assign
$let\ x = 3$ $(2 * x) - x\hat{}2 * 5/2$	-16			
$let\ x = 3$ $(2 * x) - x\hat{}2 * (5/2)$	-12	-12	Pass	
$let\ x = 3$ $(2 * x) - x\hat{}2 * 5/2.0$	-16.5	-16.5	Pass	Conversion
$let\ x = 3$ $(2 * x) - x\hat{}2 * 5\%2$	5	5	Pass	
$let\ x = 3$ $(2 * x) - x\hat{}2 * (5\%2)$	-3	-3	Pass	

Table B.3: Function tests. Note that floating pointing values are accurate to three decimal places for the fractional part. ResE is expected result and ResA is actual result.

Expression	ResE	ResA	Pass/Fail	Action/comment
$let\ f = (x)\ \rightarrow x^2$ $f(3)$	9	9	Pass	function
$let\ f = (x)\ \rightarrow x^2$ $f(3) + f(2)$	13	13	Pass	function use
$let\ f = (x)\ \rightarrow \cos(x)$ $f(0)$	1.000	1.000	Pass	function composition
$let\ f = (x)\ \rightarrow \cos(x)$ $f(0) + f(0)$	2.000	2.000	Pass	function composition
$letf = () \rightarrow 5$ $f()$	5	5	Pass	function no args
$letf = () \rightarrow 5$ $letfa = (x) \rightarrow x * 4$ $f() + fa(3)$	17	17	Pass	function no args
$letf = () \rightarrow 5$ $letfa = (x) \rightarrow x * 4$ $fa(f())$	20	20	Pass	function composition
$let(\mid >) = (x, f) \rightarrow f(x)$ $5 \mid > (x) \rightarrow x * 4$	20	20	Pass	operator overloading
$let(\mid >) = (x, f) \rightarrow f(x)$ $5 \mid > (x) \rightarrow x * 4 \mid > (x) \rightarrow$ $x + 3$	23	23	Pass	operator overloading

Table B.4: Type Inference tests. ResE is expected result and ResA is actual result.

Expression	ResE	ResA	Pass/Fail	Action/comment
$let\ x = 3$	int	int	Pass	int
$let\ x = 3.0$	float	float	Pass	float
$let\ x = 3.0 + 2$	float	float	Pass	addition
$let\ x = 3.0 + 2.0$	float	float	Pass	
$let\ x = 3 + 2.0$	float	float	Pass	
$let\ x = 3 + 2$	int	int	Pass	
$let\ x = (() => 3)()$	int	int	Pass	
$let\ x = (() => 3.0)()$	float	float	Pass	
$let\ x = (() => [1, 2])()$	List[float]	List[float]	Pass	
$let\ x = (() => a = 3)()$	Record[a: int]	Record[a: int]	Pass	
$let\ x = (() => a = 3, b = 4)()$	Record[a: int, b: int]	Record[a: int, b: int]	Pass	
$let\ x = 4 : float$	float	float	Pass	Casting
$let\ x = 4 : int$	int	int	Pass	
$let\ x = 4.0 : int$	int	int	Pass	Casting
$let\ x = 4.0 : float$	float	float	Pass	

Table B.5: Compound DT tests. Note that floating pointing values are accurate to three decimal places for the fractional part. ResE is expected result and ResA is actual result.

Expression	ResE	ResA	Pass/Fail	Action/comment
$[1, 2, 3]$	[1,2,3]	[1,2,3]	Pass	list
$[1, 2, 3][1]$	2	2	Pass	list index
$[1, 2, 3][1..2]$	[2]	[2]	Pass	list index range
$[1, 2, 3][1..]$	[2,3]	[2,3]	Pass	list index range
$[1, 2, 3][..2]$	[1]	[1]	Pass	list index range
$[1..3]$	[1,2,3]	[1,2,3]	Pass	list range
$[1..3][1]$	2	2	Pass	list range index
$[1..3][1..2]$	[2]	[2]	Pass	list range index range
$\{a = 3\}$	a = 3	a = 3	Pass	record
$\{a = 3\}.a$	3	3	Pass	record index
$\{a = 3\}.b$	Error	Error	Pass	record index
$\{a = 3, b = 4\}.b$	4	4	Pass	record index
$\{a = 3, b = 4\}.a$	3	3	Pass	record index
$\{a = 3, b = 4\}.a + a = 3, b = 4.b$	7	7	Pass	record index

Table B.6: Control Flow tests. Note that floating pointing values are accurate to three decimal places for the fractional part. ResE is expected result and ResA is actual result.

Expression	ResE	ResA	Pass/Fail	Action/comment
$if\ 5 == 4\ then\ 3\ else\ 2$	2	2	Pass	false
$if\ 5 == 5\ then\ 3\ else\ 2$	3	3	Pass	true
$if\ 5\ then\ 3\ else\ 2$	Error	Error	Pass	invalid type
$if\ 5 == 5\ then\ 3$	Error	Error	Pass	missing else
$if\ true\ then\ print(\text{"}here\text{"})$	here	here	Pass	print no else (unit branch)
$if\ false\ then\ 1\ else\ 2$	2	2	Pass	false
$let\ rec\ fact = (n) \rightarrow if\ n \leq 1\ then\ 1\ else\ n * fact(n-1)$				
$fact(5)$	120	120	Pass	factorial

Appendix C

Algorithms

This chapter will showcase some of the maths focused algorithms used in the project.

C.1 Root finding

Both the bisection and Newton-Raphson methods are implemented.

Algorithm 3 Bisection method

```
function BISECTION(f, a, b, tol)
    c ← (a + b)/2
    while |f(c)| > tol do
        if f(a) · f(c) < 0 then
            b ← c
        else
            a ← c
        end if
        c ← (a + b)/2
    end while
    return c
end function
```

Algorithm 4 Newton-Raphson method

```
function NEWTONRAPHSON(f, df, x0, tol)
    x1 ← x0 − f(x0)/df(x0)
    while |f(x1)| > tol do
        x0 ← x1
        x1 ← x0 − f(x0)/df(x0)
    end while
    return x1
end function
```

C.2 Calculus

In the language, pure functions are represented in a symbolic expression DSL, making it easy to differentiate and integrate functions, as well as find tangent lines or the tailor series of a function. The symbolic expression DSL created was inspired by the systems found at Nonsense (2013), SymPy (2024) and BMITC (2022).

Algorithm 5 Differentiation

```
function DIFFERENTIATE(f)
    if f is a constant then
        return 0
    else if f is a variable then
        return 1
    else if f is a sum then
        return differentiate(f_1) + differentiate(f_2)
    else if f is a product then
        return f_1 · differentiate(f_2) + f_2 · differentiate(f_1)
    else if f is a power then
        return n · x^(n−1)
    else if f is cos then
        return −sin(x)
    else if f is sin then
        return cos(x)
    else if f is tan then
        return sec^2(x)
    end if
end function
```

C.3 Vector operations

Functions used to find the cross and dot products of two vectors are implemented, as well as standard arithmetic operations.

Algorithm 6 Cross product

```
function CROSSPRODUCT(v1, v2)
    x ← v1[1] · v2[3] − v1[3] · v2[2]
    y ← v1[3] · v2[1] − v1[1] · v2[3]
    z ← v1[1] · v2[2] − v1[2] · v2[1]
    return [x, y, z]
end function
```

Algorithm 7 Dot product

```
function DOTPRODUCT(v1, v2)
    sum ← 0
    for i ← 1 to n do
        sum ← sum + v1[i] · v2[i]
    end for
    return sum
end function
```

C.4 Matrix operations

The language has built-in functions for finding the transpose, determinant and inverse of a matrix.

Algorithm 8 Transpose

```
function TRANSPOSE(m)
    rows ← length(m)
    cols ← length(m[1])
    trans ← emptyMatrix(cols, rows)
    for i ← 1 to rows do
        for j ← 1 to cols do
            trans[j][i] ← m[i][j]
        end for
    end for
    return trans
end function
```

Algorithm 9 Determinant

```
function DETERMINANT(m)
    rows ← length(m)
    cols ← length(m[1])
    if rows ≠ cols then
        return 0
    end if
    if rows = 2 then
        return m[1][1] · m[2][2] − m[1][2] · m[2][1]
    end if
    det ← 0
    for i ← 1 to rows do
        det ← det +m[1][i] · cofactor(m, 1, i)
    end for
    return det
end function
```

Algorithm 10 Inverse

```
function INVERSE(m)
    rows ← length(m)
    cols ← length(m[1])
    det ← determinant(m)
    if det = 0 then
        return error
    end if
    inv ← emptyMatrix(rows, cols)
    for i ← 1 to rows do
        for j ← 1 to cols do
            inv[i][j] ← cofactor(m, i, j)/ det
        end for
    end for
    return inv
end function
```