Project Report - 13 January 2025

Vec3 Maths Bytecode Interpreter

Group members:
Jamie Wales, Jacob Edwards.

School of Computing Sciences, University of East Anglia

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

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.

Chapter 2

Background

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.

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.

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.

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 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]`. Other compound data types such as tuples and records were also added.

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.

3.9.1   Grammar in BNF

3.10   Sprint 10: Control flow

In this sprint we added control flow to the language, with `if` expressions and recursive
bindings.

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>
```

```
<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>
```

3.11   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. 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. 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>
                | <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>
<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:

```fsharp
/// <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.
```

13

```fsharp
        /// </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

4.2.1   Code Editor

4.2.2   Notebook View

4.2.3   Plot View

4.3   Notable Features

Alongside 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:

```fsharp
// 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
```

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(lis
```

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)
```

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.

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 precedence 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 Machine4.11, 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).

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 an other 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*.

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
```

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 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 $type1$ **is** *row variable* **then**
            $type1 \leftarrow type2$
        end if
        if $type2$ **is** *row variable* **then**
            $type2 \leftarrow type1$
        end if
        if $type1$ **is** *record type and* $type2$ **is** *record type* **then**
            unify row variables
        end if
        if $type1$ **is** not equal to $type2$ **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)4.14:

```
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, 2024) (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* (4.13) 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 view4.15 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

The core of the Vec3 interpreter was a transition from a tree-walk interpreter to a more efficient stack-based virtual machine. This involved two primary components: a compiler to translate Vec3 source code into bytecode, and a virtual machine to execute that bytecode. The fundamental goal was to achieve faster execution speeds by working with a compact and streamlined instruction set.

The bytecode itself was a sequence of instructions, each represented by an operation code (opcode) and, potentially, operands that the opcode would act upon. These opcodes, defined in the `OP_CODE` type, covered a range of operations necessary for a fully functional language. This included instructions for pushing constants onto the stack (`CONSTANT`, `CONSTANT_LONG`), performing arithmetic (`ADD`, `SUBTRACT`, `MULTIPLY`, `DIVIDE`, `NEGATE`), managing control flow (`RETURN`, `JUMP`, `JUMP_IF_FALSE`, `LOOP`), handling boolean logic (`NIL`, `TRUE`, `FALSE`, `NOT`, `EQUAL`, `GREATER`, `LESS`), manipulating the stack (`POP`), working with global and local variables (`DEFINE_GLOBAL`, `GET_GLOBAL`, `SET_GLOBAL`, `GET_LOCAL`, `SET_LOCAL`), outputting values (`PRINT`), and calling functions (`CALL`). Two functions, `opCodeToByte` and `byteToOpCode`, handled the conversion between these symbolic opcodes and their corresponding byte representations, ensuring a compact bytecode format.

Compiled code, along with associated data, was organized into `Chunk` structures. Each chunk contained a `Code` array, holding the bytecode instructions as a sequence of bytes. A `ConstantPool` array stored constant values referenced by the instructions, enabling efficient reuse of values like numbers and strings. To aid in debugging, a `Lines` array mapped bytecode offsets to their corresponding line numbers in the original source code. Functions like `emptyChunk`, `writeChunk`, `addConstant`, `writeConstant`, and `getLineNumber` provided an interface for creating and manipulating chunks.

The compiler's role was to transform the abstract syntax tree (AST) representation of Vec3 code into this bytecode format. It maintained a `CompilerState` to track the chunk being generated, local variables within the current scope, the current scope's nesting depth, and the line number being processed. The compilation process involved a recursive descent through the AST. Functions like `compileStmt` and `compileExpr` recursively processed statements and expressions, respectively. For each AST node encountered, the compiler emitted corresponding bytecode instructions using helper functions like `emitByte`, `emitBytes`, `emitConstant`, and `emitOpCode`.

Variable declarations were handled by `compileVariableDeclaration`, which added the variable to the `Locals` map in the `CompilerState`. This map stored local variables and their corresponding slot indices on the virtual machine's stack. Control flow instructions, like jumps and loops, were initially emitted with placeholder offsets. These placeholders were later patched with the correct offsets once the target locations were determined.

Error Handling

Error handling during compilation was managed using the `CompilerResult` type. This allowed the compiler to either return a successful result along with an updated `CompilerState` or an error along with a descriptive message and the state at the point of the error.

Debugging

To facilitate debugging, a disassembler was implemented. Functions like `disassembleInstruction` and `disassembleChunk` took the compiled bytecode and produced a human-readable representation, showing the instructions, their operands, and their associated source code line numbers. This was invaluable for understanding the generated bytecode and identifying potential issues.

4.10 Virtual Machine

The Virtual Machine (VM) was responsible for executing the compiled bytecode. It was designed as a stack-based machine, meaning that it used a stack to store intermediate values during computation. The VM's state was represented by the `VM` type, defined as follows:

```
type VM = {
    Chunk: Chunk
    IP: int
    Stack: ResizeArray<Value>
    ScopeDepth: int
}
```

where:

* `Chunk` held the bytecode and associated data (constant pool, line information) currently being executed. * `IP` (Instruction Pointer) was an integer representing the index of the next bytecode instruction to be executed. * `Stack` was a dynamically sized array used to store values during computation. Operations like arithmetic, comparisons, and function calls would push and pop values from this stack. * `ScopeDepth` tracked the current level of scope nesting.

The `createVM` function initialized a new VM instance with a given chunk. The core of the VM was the `run` function, a recursive loop that fetched, decoded, and executed instructions until a `RETURN` instruction was encountered or an error occurred.

Key helper functions included:

* `push`: Pushed a value onto the stack. * `pop`: Popped a value from the stack. * `peek`: Looked at a value on the stack at a given position without removing it. * `readByte`: Read the byte at the current instruction pointer and incremented the IP. * `readConstant`: Read a constant index from the bytecode, fetched the corresponding value from the constant pool, and pushed it onto the stack. * `readConstantLong`: Similar to `readConstant`, but for constants that required a larger index. * `binaryOp`: Performed a binary operation on the top two values on the stack.

The `run` function used a match expression to dispatch to the appropriate code based on the current opcode. Each opcode case handled a specific instruction, potentially manipulating the stack, performing calculations, or managing control flow.

The `interpret` function provided the main entry point for executing a chunk of bytecode. It first disassembled the chunk for debugging purposes, then created a new VM instance and called `run` to start the execution process.

In conclusion, this initial design established a solid foundation for a stack-based virtual machine and its associated compiler. It emphasized a clean separation of concerns between compiling and executing code, a compact bytecode representation, and a focus on essential features for a functional language. The inclusion of debugging tools, a well-defined error-handling mechanism, and a dedicated virtual machine for execution further contributed to the robustness and efficiency of the system.

4.11 Virtual Machine

4.12 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. Initially, defining the following instructions:

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.

4.13   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,
    XValues: [float],
    YValues: [float],
    ptype: "bar" | "scatter" | "signal",
}
```

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

Add images of plots

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.

Add images of plots

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.

The plot windows also have an input at the bottom, which allows for the user to input a function and have it plotted on command. This is useful for quick visualisation of functions, and allows for a more interactive experience.

4.14   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.

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 4.12).

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

Add images of drawings

4.15   Notebook View

The notebook view is a feature that allows the user to write code in a more interactive way, similar to Jupyter notebooks(Project Jupyter, 2023).

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.

4.17   Code architecture

Chapter 5

Discussion, conclusion and future work

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.

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

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.

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/`.

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

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

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.

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

B.1   Lexer testing

B.2   Parser testing

B.3   Expression testing

B.4   Variable assignment testing

B.5   Function testing

B.6   GUI testing

B.7   Plot testing

Appendix C

Syntax