Fluxion Language Specification 0.1

Part I: Syntax & Semantics

Ege Özkan

September 25, 2020

Contents

Chapter 1

Introduction

This chapter includes the design philasophy of and concepts behind the Fluxion Language, and a defence of the Fluxion Language, that is, why it is needed.

1.1 A Defence of Fluxion

First thing one has to understand when it comes to language is its usefulness. As the readers may ask, *why* should Fluxion exist? Don't we already have many scientific programming langauges like Python or Julia?

The answer to these questions can be given twofold: First, languages live and breath alongside with their ecosystem, even though their existence might be meaningful without a certain use case, and some even develop use cases they are originally not created for, in many cases, languages with a clear goal are easier to design.

Domain Specific Languages, are languages created to answer problems arising from a domain. Fluxion is such a language, its chief primary concern is to create a language, that can adequetly represent mathmatical constructs and expressions from many differing fields, to simplify, solve or to manuplate these constructs for the benefit of the users, who may wish to use Fluxion in their systems for a variety of purposes. These may range from pedogocial concerns, such as to teach pupils about certain mathematical concepts to proving actual mathematical problems.

When it comes to languages like Python, Julia or even Matlab, though they are useful, they are far to complex to be simple enough for users to actually engage with the symbolic mathematical underlayings of formulas, theorems and proofs. Fluxion aims to be simpler, much closer to mathematical notation, while still retatining *some* but not all of their power. Fluxion does not aim to be a general purpose programming language. It does not even aim to be *a* programming language. Fluxion is a Symbolic Computation Language, and it aims to excel at that one field.

Second, Fluxion does not aim to replace these languages, though some users will benefit it from it, to such a degree that in some use cases, they may prefer Fluxion, Fluxion will be written in such a way that it can be embedded and itnerloop with many of these languages, which will allow users to concantrate on problems and expressions like never before.

When designing Fluxion, first aim was to create a language that is easy to understand, extremely close to mathematical notation even if it went against previously established conventions of Computer Science, or if it adhered to conventions lesser known, this allows people litterate in Mathematics to easilly pick up Fluxion.

The second aim was to make sure that mistakes were limited. For instance, unlike many other languages, star imports does not exist in Fluxion to avoid namespace cluttering, in the reverse side of the import, module members are by default private, and must be exported explicitly using the `export` keyword.

Third aim of the Fluxion's *design* is for it to be extendable. Using the `introduce` keyword, users and introduce Fluxion new *contexts*, that can be about different domains of science using mathematics. A Physics domain may introduce Physics related constants and functions, a Mechanics domain may introduce functions and constants related to the mechanical engineering. This allows Fluxion to be tailored towards whatever purpose the user wishes it to be.

Fourth and final aim of the Fluxion is to be a free and open source alternative to the already established systems that are not as readilly as accessible or perhaps extendable as the Fluxion language itself.

1.1.1 Fluxion is not a Programming Language

Fluxion is not a programming language. It is a symbolic computation language. However, this specification will refeer to the language as *the Fluxion Language* for keeping things short.

1.1.2 the Name Fluxion

Fluxion was the original name given to the instantenous rate of change of time sensitive function (or a Fluent Quantity, a *Fluent*) by Sir Isaac Newton[3]. In essance, Fluxion is an obscure term for derivatives[1]. Given that Fluxion arised from a Clojure program that was made to take derivatives, I felt it is a good name as any to give it.

1.2 Contents of the Specification

Subsequent chapters of this specification will talk about different components of Fluxion. Each chapter will talk about a component, its syntax, semantics and behaviour. Syntaxes of these components will be expressed via the Extended Backus-Naur Form[2] with very minor augmentations.

Next chapter contains the lexical structure of the language via Extended BNF, including the operators of the language, statements, expressions and literals. Third hapter contains these basic literal expressions, as well as the type system of Fluxion. Fourth chapter contains the operations on these literals. Chapter five discusses assignment and function definitions as well as built-in functions of the language. Chapter six discusses reductions, chapter seven discusses differentiation of various algabraic data types and finally, chapter eight discussess the various statements, such as `introduce` and `in` the language has.

This is the first part of a two parter language specification. As such, the standard library facilities will be introduced in the second part of this specification. Implementation of the standard library is necessary for a complete implementation of the Fluxion language, however, a minimal implementation may leave it out.

Chapter 2

Grammar

This chapter includes the relevant grammar for statements, expressions and literals of the Fluxion language in Extended Backus-Naur Form.

2.1 Notation

The notation included in this specification is ISO14977 Extended Backus-Naur Form, with the addition of the kleen-star $\langle a \rangle$*, this symbol is used in place of the $\{\langle a \rangle\}$, and represents that a certain lexeme may be repeated zero or more times.

2.2 Common Lexical Units

Below are some lexemes, either in description, or in extended BNF that will be used as the building blog of more complex lexemes. first of all, let us define some production rules informally:

$\langle$*letter*$\rangle$ As it is used in the ISO14977, an English character within the ASCII, lowercase or uppercase.

$\langle$*unicode*$\rangle$ A character that is either a `letter` or any unicode string that is not an operator.

$\langle$*digit*$\rangle$ As it is defined in the ISO14977, a digit is a base ten numeral between 0 and 9.

$\langle$*space*$\rangle$ Space, or more formally the ASCII character number thirty two.

$\langle$*tab*$\rangle$ Tab, or more formally ASCII character number nine.

$\langle$*EOF*$\rangle$ End of file character. May also be substituted for end of string.

$\langle$*EOL*$\rangle$ End of line character.

2.3 Literals

Literals are the rough equivalents of rvalues in the C programming language. They are used to define "types".

2.3.1 Boolean Literals

$\langle$*boolean*$\rangle$:= '`True`' | '`False`'

2.3.2 Numerical Literals

$\langle$*unsigned*$\rangle$:= $\langle$*digit*$\rangle$ [, $\langle$*digit*$\rangle$*]

$\langle$*integer*$\rangle$:= [$\langle$*sign*$\rangle$], $\langle$*unsigned*$\rangle$

$\langle$*decimal*$\rangle$:= $\langle$*integer*$\rangle$, '`.`', $\langle$*unsigned*$\rangle$

⟨*imaginary*⟩ := ⟨*number*⟩, ⟨*sign*⟩, ⟨*number*⟩, ‘i’

⟨*number*⟩ := ⟨*decimal*⟩ | ⟨*integer*⟩ | ⟨*imaginary*⟩

2.3.3 Sets and Enumerables

Sets and Enumerables share the same grammar they differ on semantics. Therefore, both will share the ⟨*set*⟩ production rule.

⟨*finite*⟩ := ‘{’, ⟨*expression*⟩*, ‘}’

⟨*builder*⟩ := ‘{’, ⟨*identifier*⟩, ’|’, ⟨*logical_condition*⟩, ‘}’

⟨*set*⟩ := ⟨*finite*⟩ | ⟨*builder*⟩

2.3.4 Sequences

Sequences are either built when ⟨*finite*⟩ meets a condition or with their own notation.

⟨*sequence*⟩ := ⟨*finite*⟩ | ([⟨*finite*⟩], ⟨*identifier*⟩ ‘->’ ⟨*expression*⟩)

2.3.5 Matrices and Vectors

Vectors are subset of Matrices, so much so that, there is no special production rule for vectors.

⟨*row*⟩ := ⟨*expression*⟩, (⟨*whitespace*⟩, ⟨*expression*⟩)*

⟨*matrix*⟩ := ‘[’, ⟨*row*⟩, (‘|’, ⟨*row*⟩)*, ‘]’

2.3.6 General Literal Grammar

⟨*literal*⟩ := ⟨*boolean*⟩ | ⟨*number*⟩ | ⟨*set*⟩ | ⟨*matrix*⟩ | ⟨*sequence*⟩

2.4 Operators

2.4.1 Binary Operators

⟨*algebraic_operator*⟩ := ‘+’ | ‘-’ | ‘/’ | ‘*’ | ‘^’ | ‘->’ | ‘_’

⟨*comperasion_operator*⟩ := ‘<’ | ‘>’ | ‘<=’ | ‘>=’ | ‘\=’ | ‘=’

⟨*logical_operator*⟩ := ‘|’ | ‘in’ | ⟨*comperasion_operator*⟩

⟨*hybrid_operator*⟩ := ‘&’

⟨*operator*⟩ := ⟨*algebraic_operator*⟩ | ⟨*logical_operator*⟩ | ⟨*hybrid_operator*⟩

2.4.2 Unary Operators

⟨*sign*⟩ := ‘+’ | ‘-’

⟨*prefix*⟩ := ⟨*sign*⟩ | ‘\’

⟨*postfix*⟩ := ‘'’ | ‘!’

⟨*unary_operator*⟩ := ⟨*prefix*⟩ | ⟨*postfix*⟩

⟨*hybrid_operator*⟩s are named as such as they can either be parsed as ⟨*logical_operator*⟩s or as ⟨*comperasion_operator*⟩s depending on the literals they are used with. In cases where ambiguity can arise, their algebraic versions have precedence.

2.5 Variables and Expressions

⟨*identifier*⟩ := [⟨*identifier*⟩, ‘::’]⟨*letter*⟩, ⟨*unicode*⟩*

⟨*function*⟩ := ⟨*identifier*⟩, ‘(’, (⟨*variable*⟩ | ⟨*literal*⟩ | ⟨*expression*⟩) , ‘)’

⟨*variable*⟩ := ⟨*identifier*⟩ | ⟨*function*⟩

⟨*cardinality*⟩ := ‘|’, ⟨*expression*⟩, ‘|’

⟨*expression*⟩ := ⟨*cardinality*⟩ | [⟨*prefix*⟩], (⟨*literal*⟩ | ⟨*variable*⟩) [, ⟨*operator*⟩, ⟨*expression*⟩], [⟨*postfix*⟩]

⟨*logical_condition*⟩ := [‘\’], (⟨*boolean*⟩ | ⟨*variable*⟩) [, ⟨*logical_operator*⟩ ⟨*logical_condition*⟩]

⟨*assignment*⟩ := ⟨*variable*⟩, ‘:=’, ⟨*expression*⟩

Here the ⟨*function*⟩ production rule covers the function definition, as well as the function calls.

2.6 Statements

⟨*introduce*⟩ := `introduce`, ⟨*identifier*⟩

⟨*solve*⟩ := ‘?’, ⟨*expression*⟩

⟨*export*⟩ := ‘`export`’, ⟨*identifier*⟩, (‘,’ ⟨*identifier*⟩)*

⟨*statement*⟩ := ⟨*solve*⟩ | ⟨*introduce*⟩ | ⟨*export*⟩

2.7 General Program Grammar

$\langle line \rangle$:= $\langle expression \rangle$*, EOL

$\langle multiline\_comment \rangle$:= '`;*`', $\langle unicode \rangle$*, '`*;`'

$\langle single\_comment \rangle$:= '`;;`', $\langle unicode \rangle$*, EOL

$\langle multiline \rangle$:= $\langle line \rangle$, ($\langle tab \rangle$ | ($\langle space \rangle$ $\langle space \rangle$) $\langle line \rangle$)*

$\langle unit \rangle$:= $\langle multiline \rangle$ | $\langle statement \rangle$ | $\langle assignment \rangle$

$\langle program \rangle$:= (($\langle unit \rangle$ | $\langle comment \rangle$)*, EOF

Multiline statements can be achived by putting two spaces or a tab character in a line, in such case, programs treat this line as the continuation of the previous line.

Chapter 3

Fluxion's Type System

This chapter includes the explanation of the Fluxion's type system from a programming language perspective.

3.1 An Overview of the Fluxion's Type System

Fluxion is a dynamically typed language. Where the type of a variable is inferred by the language. Semantics of the operators change depending on which types they are acting upon in a given context.

All Fluxion types are immutable. As such, variables and literals when passed to functions, only pass by reference and never by value.

Fluxion has five main types. `boolean`, `number`, `set`, `matrix` and `sequence`. Of these five types, `number` is split into `integer`, `decimal` and `imaginary`; a subset of `set` types are `Enumerable` types and a subset of `matrix` types are `vectors`. `number` and `matrix` types can interract directly within an expression depending on the circumstances (which will shortly be described). Members of `sequences` and `enumerables` can also be called upon to interract with `number` and `matrix` types, however, `boolean` types cannot interract with other types directly. Therefore, `number`, `set` and `matrix` types are called *algebraic* types.

3.1.1 Unbound Variable

Since Fluxion is a Symbolic Computation Language, users may wish to leave variables without definition. In such cases, it is not guranteed what a variable may represent. Such variables are called *Unbound variables.* If the variable `x` is undefined, the expression `x^2 + 2` is still a valid expression.

3.1.2 Associated Grammar

Each Fluxion literal type can be produced by using an associated production rule. Types that adhere to the same production rules are special cases of their more general types.

3.2 Booleans

Fluxion has two boolean values, `True` and `False`. These values may arise from logical operations. Unlike many languages, algebraic values themselves do not carry any significance in terms of boolean values.

3.3 Numbers

Numeric types are types that carry numeric values. They can be decimals, integers or imaginary numbers. A numerical literal written without a sign is presumed to be positive.

Table 3.1: Grammar rules associated with each type.

	Rule	Relevant Section
`boolean`	$\langle boolean \rangle$	§2.3.1
`number`	$\langle number \rangle$	§2.3.2
`integer`	$\langle integer \rangle$	§2.3.2
`decimal`	$\langle decimal \rangle$	§2.3.2
`imaginary`	$\langle imaginary \rangle$	§2.3.2
`set`	$\langle set \rangle$	§2.3.3
`enumerable`	$\langle set \rangle$	§2.3.3
`sequence`	$\langle sequence \rangle$	§2.3.4
`matrix`	$\langle matrix \rangle$	§2.3.5
`vector`	$\langle vector \rangle$	§2.3.5

3.3.1 Integers

Integers are whole numbers. Fluxion allows an integer value to be between $-(2^{64}-1)$ and $2^{64}-1$. Trying to assign a variable to an integer literal above this number, or attempting to operate on this number will result in an exception.

```
32
−23
0
```

Snippet 3.1: Example integers

3.3.2 Decimal

Decimals are numbers with a decimal point. They must be stored as fractions of integers within the language, thus allowing for high precision and easier reduction of expressions.

```
3.12
+4.234
−3.12
```

Snippet 3.2: Example decimals

3.3.3 Imaginary numbers

Imaginary numbers are created by summing a number with another number multiplied with $i = \sqrt{-1}$ This also makes `i` a reserved keyword. If `a := b + c*i`, `b` is called the real part and `c` is called the imaginary part. Both `a` and `b` can take any value their type allows. (if `a` is an `integer`, it can be between negative and positive $2^{64}-1$)

```
3 + 2i
a + b*i
-3 - 1i
```

Snippet 3.3: Example imaginals

3.3.4 Irrational Numbers

Irrational numbers are undefinable by normal user programs in Fluxion. However, special Irrational numbers that are often used in math, namely π, τ and e are predefined in the language as `pi`, `tau` and `euler`. Functions that act on these values are hardcoded to resolve them in mathematically correct ways.

For instance, `sin(pi)` resolves to `0` and `ln(euler)` resolves to `1`. In expressions where such resolutions are impossible these variables are retained as variables. For more information, please refeer to §6.1.2.

3.4 Matrices

A matrix consists of rows and columns. A matrix of n rows and m columns is told to be a matrix of size $m \times n$. In Fluxion, matrices may be as big as the memory allows. Moreover, matrices may hold values of arbitary types, including other matrices.

```
[1 2 3 | a b c | (12 + a) | (13 + c) | 3]
```

Snippet 3.4: An example matrix

3.4.1 Vectors

Vectors are Matrices of dimension $m \times n$ where at least one of m or n equals one. Vectors can be horizontal or vertical.

```
[1 2 3]
[1 | 2 | 3]
```

Snippet 3.5: Example vectors

3.5 Sets

Sets are container types that may be infinite or finite. They can be either defined using set builder notation, or by putting elements inside curly brackets. Sets are unorderdered, therefore, one cannot get elements outside a list. But one can check if an element is inside a list, sets also cannot contain two elements that are equal.

```
{1, 2, 3} ; A finite set.
{} ; An empty set.
{x | x in dN} ; A set with set builder notation.
```

Snippet 3.6: Example vectors

3.5.1 Enumerables

Enumerables are special lists, they are either lists that are finite or countably infinite. A set is said to be countably infinite if its members can be mapped one to one with the set of natural numbers. More formally, Enumerables are sets that are enumaratable, for this reason, Enumerables are ordered.

3.5.2 Predefined Sets and Enumerables

Fluxion programming language comes built-in with certain sets and Enumerables. First of these is the `units` enumerable, which contain first three base vectors in order, $\vec{i}$, $\vec{j}$ and $\vec{k}$.

Number Sets

More usefully, Fluxion comes predefined with sets that represent number sets in math, these are called *domains* (domain of natural numbers, etc.). Some of these, ones that can be mapped to natural numbers via bijection are also Enumerables.

Table 3.2: Domains, their Fluxion equivalents and their enumeration method as Enumerables, if it exists.

	Variable	Enumaration
$\mathbb{N}$	`dN`	$n \to n$
$\mathbb{Z}$	`dZ`	$\{0, -1, 1, -2, \ldots\}$
$\mathbb{Z}^+$	`dZp`	$\{1, 2, 3, 4, 5, \ldots\}$
$\mathbb{Z}^-$	`dZn`	$\{-1, -2, -3, \ldots\}$
$\mathbb{Q}$	`dQ`	§10.3.1
$\mathbb{Q}^-$	`dQn`	§10.3.2
$\mathbb{Q}^+$	`dQp`	§10.3.3
$\mathbb{R}$	`dR`	–
$\mathbb{C}$	`dC`	–

3.6 Sequences

Sequences are container types that are enumarated. They can contain the same element more than once. They either come into existence when the curly bracket syntax is used with a repeating element or with their own special notation.

```
a_sequence := {1, 2, 2}
an_enumarable := {1, 2, 3}
;; Underneath is the fibonacci!
another_sequence := {1, 1} x_n -> x_(n - 1) + x_(n - 2)
```

In this notation, the first finite sequence gives the first elements, then the next part is the production rule by which the next elements are created.

Chapter 4

Operations

This chapter includes how operators act upon different variable types and the results of these operations.

Table 4.1: A classification of operators in Fluxion

<table>
<tr><th>Domain</th><th>Arity</th><th>Position</th><th>Operators</th></tr>
<tr><td rowspan="4">Algebraic</td><td>Binary</td><td>Infix</td><td><code>+</code>, <code>-</code>, <code>/</code>, <code>*</code>, <code>^</code>, <code>&</code>, <code>in</code>, <code>-></code>, <code>_</code></td></tr>
<tr><td rowspan="3">Unary</td><td>Prefix</td><td><code>+</code>, <code>-</code>, \</td></tr>
<tr><td>Postfix</td><td><code>!</code>, <code>'</code></td></tr>
<tr><td>Midfix</td><td><code>||</code></td></tr>
<tr><td rowspan="2">Boolean</td><td>Binary</td><td>Infix</td><td><code><</code>, <code>></code>, <code><=</code>, <code>>=</code>, <code>=</code>, \<code>=</code>, <code>&</code>, <code>|</code></td></tr>
<tr><td>Unary</td><td>Prefix</td><td>\</td></tr>
<tr><td>Scope</td><td>Binary</td><td>Infix</td><td><code>::</code>, <code>:=</code></td></tr>
</table>

4.1 Operator Semantics

As established in §2.4, Fluxion has operators of many types, moreover, semantics of operators may change depending on the type of the value they are acting upon, akin to operator overloading.

Immutability of the Fluxion types mean that, when an operator act upon two expressions of any type, the result evaluates to another expression. This may be a algebraic value, an unbound variable, a smaller operation or the same operation.

Operations are the most top level expression, whose grammar is defined within §2.5 as ⟨*expression*⟩. Operations generally consist of one or more expressions (called operands) bound by an operator. These operators can be categorised in different ways: By their *domain*, the type of variables they act on; by their *arity*, the number of variables they take and by their *position* with respect to the variables they take.

As can be seen in Table 4.1, all Fluxion operators either take one operand, and hence they are *unary* or they take two operands, and hence they are *binary*. They either act upon algebraic and boolean types, although, same symbol maybe used for different meanings, and hence may take occupy more than one category.

All binary operators in Fluxion are also infix operators, which means they are placed in between the expressions they act upon, for instance, the algebraic sum operator `+` is placed between two expressions, as in `a + b`. Prefix unary operators are placed before the expression they act upon, for instance, boolean not operator \, is used as \`True`. On the other hand, postfix unary operators are placed following the expression they act upon, such as `a!`.

The midfix operator is the cardinality operator that is generally used for size and length related calculations. It is used such as `|a|`. The scope operators are operators that somehow modify the variable scope and are outside the scope of this chapter. (§5, §10)

Algebraic Operations (but not operators) can also be classified as homogeneous and heterogeneous operations. Homogeneous operations are either binary operations that act on the same types of operands, or unary operations, whereas the heterogeneous operations are operations that act between two different types of values. For instace, in mathematics, 12! and $\vec{a} \times \vec{b}$ is homogeneous, whereas $12\vec{a}$ is heterogeneous.

4.2 Error Literals

Error literals are literals that are created as a result of an mathematical error. When an error literal is evaluated, the language implementation must inform the user where it happened and if possible offer reasons begind it.

`Overflow` An expression whose absolute value is bigger or equal to 2^{64}.

`Undefined` Using an operator with types that does not support it. Dividing by zero.

`Indeterminate` Dividing zero by zero. May also arise from certain differentiations: §7.

```
> a := True + False
Undefined: + operator undefined between two boolean literals.
> 0/0
Indeterminate: 0/0 is an indeterminate form.
> 21!
Overflow
```

Snippet 4.1: Error Literal Examples

4.3 Code Precedence

Operations on numbers have precedence over operations on matrices, which has precedence over operations on sets, which has precedence on boolean operations. Operations within types are ordered within themselves.

4.4 Homogeneous Operations

<table>
<tr><th>Domains</th><th></th></tr>
<tr><td>Numbers</td><td><code>+</code>, <code>-</code>, <code>/</code>, <code>*</code>, <code>^</code>, <code>'</code>, <code>||</code></td></tr>
<tr><td>Integers</td><td><code>!</code></td></tr>
<tr><td>Imaginals</td><td>\</td></tr>
<tr><td>Sets</td><td><code>+</code>, <code>-</code>, <code>in</code>, \, <code>||</code>, <code>'</code>, <code>&</code>, <code>*</code></td></tr>
<tr><td>Matrices</td><td><code>+</code>, <code>-</code>, <code>&</code>, <code>||</code>, \</td></tr>
<tr><td>Vectors</td><td><code>*</code></td></tr>
<tr><td>Sequences</td><td><code>+</code>, <code>||</code></td></tr>
</table>

Table 4.2: Domains of homogeneous algebraic operators, an homogeneous operator requires all of its operands to be of this type to work correctly.

4.4.1 Numerical Operations

On numbers, precedence of operations is as follows:

1. Complex conjugate
2. Factorial
3. Differentiation
4. Exponantiation
5. Multiplication and Divison
6. Addition and Substraction

Common Operations

Operations of addition (`+`) and substraction(`-`) work commonly across all numeric types. All numbers can be written of form $a + bi$, and hence, summation of `x := a + b*i` and `y := c + d*i` evaluate to `(`(a + c) + (b + d)*i). Substriction of these terms `x - y`, evaluates to `(a - c) + (b - d)*i`

Operation of multiplication (`*`) is trivial mathematical multiplication when both sides are decimals or integers. When one of the sides are an imaginary number, real number part is distributed on both the imaginary and real parts of the number, evaluating into another imaginary number in most cases, such as $x(a+bi) = (ax+bxi)$ mathematically. When both sides are of the imaginary type, the multiplication is expanded mathematically $(a + bi)(c + di) = (ac + (a + b)i - bd) = (ac - bd) + (a + b)i$.

Division `/` works on a similar manner when both sides are real numerals. When the right hand side of the expression is real. When the left hand side is an imaginary number and the right a real number, distrbutive property is applied similar to multiplication. When the left hand side is `1` and right hand side is a complex numeral, the expression evaluates to the complex conjugate of the numeral.

Exponentiation `^` is the trivial mathematical exponentiation for real numbers with the caveat that if a negative real number a is exponantiated with a x such that $-1 < x < 1$, a^x is evaluated as $(-a)^x\sqrt{-1}$, which evaluates to $(-a)^x i$, or in Fluxion code format `(-a)^x*i`.

However, for $x, z \in \mathbb{C}$ and $z = a + bi$, expression `x^z` (x^z) evaluates to $x^a(\cos(b \ln |x|) + i \sin(b \ln(x)))$ (§10.3.1). Moreover, since the logarithms are the inverse of the exponential function, their values are only unique to a value of $2k\pi$ and as such $e^{\ln |a| - 2k\pi}(e^{2k\pi} \cos(b \ln |x|) + e^{2k\pi} i \sin(b \ln(x)))$ [4], where any value of $K \in \mathbb{Z}$ is a valid answer to this question. As such, the Fluxion evaluates an exponantiation operation where the exponent is an imaginary number to a sequence of answers, ordered the same way as $\mathbb{Z}$ (`dZ`).

Differentiation operator `'`, always returns 0. As derivative of constants are 0.

Cardinality operator `||` acts as the absolute value operator for numbers, although trivial for real numbers, if the operand is an imaginary number of form $z = a + bi$, this `|z|` will evaluate to $\sqrt{a^2 + b^2}$.

```
> 1 + 2
3
> 12 * 3
36
> 3i + 12 + 23 + 1i
35 + 4i
```

Snippet 4.2: Example usages of operations

Operations Exclusive to Integers

The factorial operation, `!` is the equivalent to factorial in mathematics, and hence only evaluates with integer values. It is used such as `12!`, the highest evaluatable factorial is 20!. Larger factorials *may* evaluate to `Overflow`. §6.

Operations Exclusive to Imaginals

The complex conjugate operation, \, which is used as `\(3 + 5i)` evaluates to the complex conjugate of the imaginary numerals. `3 - 5i`.

4.4.2 Matrix Operations

Common Operations

Matrices and vectors share many operations together. `+` and `-` act by summing elements in the same cells if the matrices are of the same dimensions, otherwise informs the user.

\ takes the transpose of a matrix, M^T. While `||` Takes the norm of a matrix. `&` is the matrix multiplication, otherwise known as cross product, $M \times N$ or $\vec{v} \times \vec{m}$.

Operations Exclusive to Vectors

When used between two vectors, `*` means the dot product of these vectors, $\vec{x} \cdot \vec{y}$.

4.4.3 Set Operations

Unlike other main types that has subtypes, all sets have a uniform set of operations that act on them. In sets,

Chapter 5

Mathematical Proofs and Functions

5.1 Exponentiation by Complex Numerals

Let $w, z \in \mathbb{C}$ and $z = a + bi$. Evaluate w^z

$$\begin{aligned}
w^z &= e^{\ln|w^z|} \\
&= e^{z\ln|w|} \\
&= e^{(a+bi)\ln|w|} \\
&= e^{a\ln|w|+bi\ln|w|} \\
&= e^{a\ln|w|}e^{bi\ln|w|} && \text{(Using } a^x a^y = a^{x+y}\text{)} \\
&= \left(e^{\ln|w|}\right)^a e^{bi\ln|w|} && \text{(Using } \left(a^b\right)^c = a^{bc}\text{)} \\
&= |w|^a e^{bi\ln|w|} && \text{(Since } e^{ln|a|} = |a|\text{)} \\
&= |w|^a e^{b\ln|w|i} && \text{Observe } e^{b\ln|w|i} \text{ is of form } e^{\theta i} \\
&= |w|^a\left(\cos\left(b\ln|w|\right) + i\sin\left(b\ln|w|\right)\right) && \text{(Using Euler's Formula, } e^{i\theta} = \cos\theta + i\sin\theta\text{)}
\end{aligned}$$

Bibliography

[1] Collins English Dictionary. fluxion.

[2] Information technology - Syntactic metalanguage - Extended BNF. Standard, International Organization for Standardization, Geneva, CH, December 1996.

[3] Isaac Newton. *the Method of Fluxions and Infinite Series: With Its Application to the Geometry of Curve-lines*. Printed by Henry Woodfall; and sold by John Nourse, 1736.

[4] robjohn (https://math.stackexchange.com/users/13854/robjohn). Complex numbers as exponents. Mathematics Stack Exchange. URL:https://math.stackexchange.com/q/1859299 (version: 2016-07-14).