Fluxion Language Specification 0.1

Part I: Syntax & Semantics

Ege Özkan

September 28, 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*⟩ | ⟨*logical_condition*⟩ , '`)`'

⟨*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

⟨*line*⟩ := ⟨*expression*⟩*, EOL

⟨*multiline_comment*⟩ := '**;***', ⟨*unicode*⟩*, '***;**'

⟨*single_comment*⟩ := '**;;**', ⟨*unicode*⟩*, EOL

⟨*multiline*⟩ := ⟨*line*⟩, (⟨*tab*⟩ | (⟨*space*⟩ ⟨*space*⟩) ⟨*line*⟩)*

⟨*unit*⟩ := ⟨*multiline*⟩ | ⟨*statement*⟩ | ⟨*assignment*⟩

⟨*program*⟩ := (⟨*unit*⟩ | ⟨*comment*⟩)*, 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`	⟨*boolean*⟩	§2.3.1
`number`	⟨*number*⟩	§2.3.2
`integer`	⟨*integer*⟩	§2.3.2
`decimal`	⟨*decimal*⟩	§2.3.2
`imaginary`	⟨*imaginary*⟩	§2.3.2
`set`	⟨*set*⟩	§2.3.3
`enumerable`	⟨*set*⟩	§2.3.3
`sequence`	⟨*sequence*⟩	§2.3.4
`matrix`	⟨*matrix*⟩	§2.3.5
`vector`	⟨*vector*⟩	§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 Reserved Numerals

Users cannot write some numerals themselves, these numerals may be needed to explain complex topics such as cardinality of infinite sets, irrational numbers, or function convergence.

Irrational Numerals

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.

Cardinality of Infinite Sets

Cardinality of countably infinite sets, that is for a set S, if $|S| = |\mathbb{N}|$ that set is told to be countably infinite, as such, to represent this cardinal, the literal `countable` is used. It is equivalent to $|\mathbb{N}| = \aleph_0$.

Cardinality of uncountably infinite sets is said to be `uncountable`, which is mathematically equivalent to any cardinal numeral larger than $\aleph_0$.

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

Domain	Arity	Position	Operators
Algebraic	Binary	Infix	`+`, `-`, `/`, `*`, `^`, `&`, `in`, `->`, `_`
	Unary	Prefix	`+`, `-`, \
		Postfix	`!`, `'`
		Midfix	`||`
Boolean	Binary	Infix	`<`, `>`, `<=`, `>=`, `=`, \`=`, `&`, `|`
	Unary	Prefix	\
Scope	Binary	Infix	`::`, `:=`

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

Domains	
Numbers	`+`, `-`, `/`, `*`, `^`, `'`, `||`
Integers	`!`
Imaginals	\
Sets	`+`, `-`, `in`, `||`, `&`, `*`
Matrices	`+`, `-`, `&`, `||`, \
Vectors	`*`
Sequences	`+`, `||`, `&`, `-`, `in`

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, `+` acts as the join operation, while `-` is the disjoin operation, returning a set of elements not inside the other set. `&` acts as the intersection operator and `*` acts as cartesian multiplication.

The cardinality operator, `||`, when used on finite sets, returns their length. When used on infinite sets, depending on its countability, it may either return `countable` `uncountable` (§3.3.4).

The contains operator, `in`, does not check for subsets when used with two sets. It still adheres to rules that will be explained in (§4.5.3).

4.4.4 Sequence Operations

Sequences have two operations that differ from sets, they support `+`, used as concatanation, and `||`, which return the length of the sequence if the sequence is a finite sequence, or `countable` if it is infinite. (There are no uncountably infinite sequences.)

Other operators listed under Table 4.4 behave the same way as they do in set operations as shown in the Set Operations subsection (§4.4.3).

4.5 Heterogenous Operations

Heterogeneous operations are operations that can be performed between different types of values.

4.5.1 Containment operator `in`

The `in` operator, when its left-hand operand is any expression and when its right-hand operand is a set or a sequence, checks if the expression (or the the value it evaluates to) is inside the set or the sequence.

4.5.2 Scalar Multiplication Operator `*`

When one of the operands of the operator `*` is a numeric type, and the other a matrix type, the expression evaluates to the same matrix scalarly multiplied with the numeric type.

4.6 Boolean Operations

Boolean operations differ significantly from algebraic operations. To start with, they are classified differently. *Pure Boolean Operations* act only on boolean literals, contrasted with *Comparison Operators*, which act only on algebraic types, they themselves can be categorised further, *Closed Comparison Operators*, operators that can compare only between the same algebraic types and *Open Comparison Operators*, operators that can compare between all algebraic types.

Operation Type	Operators
Pure Boolean	`&`, `|`, `\`
Open Comparison	`=`, `\=`
Closed Comparison	`<`, `>`, `<=`, `>=`

Table 4.3: Operators used with different types of boolean operations.

All of these operators, regardless of their types, evaluate to boolean literals `True` and `False`. Closed Comparison Operations, although allowed only within the same type, are also allowed between a main type and one of its subtypes, such as a `vector` and a `matrix`. Or between `decimals` and `integers`, but not between `imaginals` and other numeric types.

4.6.1 Pure Boolean Operators

Pure boolean operators are logical and (`&`, $\wedge$), logical or (`|`, $\vee$) and logical not (`\`, $\neg$) they behave the same way as they do in mathematics.

When evaluating, binary pure boolean operators short-circuit.

4.6.2 Open Comperasion Operators

These operators are the equals (`=`) and not equals (`\`) operators. By definition, if `a = b` evaluates to `True`, `a \= b` will evaluate to `False`.

Equality operator, when both of its operands are not of the same type, evaluates to `False`, unless if one of them is an integer and the other a double of the same value as that integer, in which case, it evaluates to `True`.

The equality operator evaluates to `True` if both operands are:

- integers or decimals, and their values are equal.
- imaginals and their respective imaginary and real parts are equal.
- matrices of same dimensions and their elements in the same indices are equal.
- sets and they contain the same elements, *regardless* of their order, even if they are enumarables.
- sequences and they contain the same elements in the same order.

4.6.3 Closed Comperasion Operators

These operators are the less than (`<`), greater than (`>`), and their ...or equal to contraparts (`<=`, `>=`)

These operators accept:

1. Two operands of the same type that are not matrices, imaginals or sequences.
2. An operand of a main type and an operand of who is a subtype of that main type.
3. An operand that is an integer, and another operand that is a double.
4. An operand that is an integer or a double, and another operand that is an irrational number.

A less than operation evaluates to `True` when two operands are:

- doubles, integers or irrational numerals, and the left hand side has a smaller value.

- sets and the left hand side is a strict subset of the right hand side.

A greater than operation evaluates to `True` when two operands are:

- doubles or integers, and the left hand side has a greater value.
- sets and the left hand side is a strict superset of the right hand side.

Their ...greater than counterparts also evaluate to `True`, if an equals operation between these two operands would evaluate to `True`.

Chapter 5

Variables

This chapter outlines the variable system of Fluxion, including function definitons. As well as outlining a list of built-in functions.

Fluxion's assignment operator ':=' is used to assign variables and functions to expressions. In Fluxion, assignment can only occur in the top level scope of the program.

Functions are also special variables, defined using an identifier that must not be priorly assigned to any value.

When a variable is assigned to an expression, the variable's name in the global scope is now bound to that expression.

5.1 Pure Variables

Pure variables can be assigned to any expression. They can also be redefined, but the redefinitions do not affect other variables, because the expressions in the right hand side of the pure variable assignments are evaluated during assignment. For more information about evaluation, please see (§5.3). For instance:

```
> r := 12
> a := r * 2
> a
| 24
> r := 128
> a
| 24
```

Attempting to use an unbound variable will result in an `Undefined` error literal to be emitted.

```
> x := y + 2
| Undefined: Identifier "y" is not assigned.
```

5.2 Functions

Functions are special forms of variables with their own scope. Functions are evaluated lazily, when a function call occurs. When defining a function, right hand side is an expression, as in pure variables, but the left hand side must be of the form `f(x)` where `f` is a valid identifier and `x` must be of the form of a pure variable or a logical condition (used in partial functions) (§2.5), or a list of functions.

The variable defined in the left hand side of the function, when used in the right hand side, refeers to a currently unbound variable. For more information as well as the resolution of Function calls, see (§5.3.1).

Functions can also be assigned to other functions. But they may not return other functions.

5.2.1 Partial Functions

Partial functions are defined by defining more than one function with the same identifier, but with a logical condition using a variable name.

```
f(x < 0) := 0
f(x >= 0) := 1
g(x, y < 0) := x * y
g(x, y >= 1) := x + y
```

If a partial function is called with a value it is not defined for, an `Undefined` error literal is emitted.

5.2.2 Function Domains

Function domains are assumed to be in the real numbers unless specifically defined, however, domains of functions can be restricted by the usage of `f(x: S1) -> S2 := ...`, where `S1` and `S2` are sets.

5.2.3 Unary Function Inverter Operation \

When the operator \ is used on a function, it evaluates to a function that is the inverse of that function. However, not all functions have inverses, hence, there are cases when this expression will evaluate to `Undefined` if the functions does not have a one-to-one mapping on its domain.

The calculated inverse function can be assigned to another function, for instance, if there exists a function `f`, `g(x) := \f(x)` will assign `g(x)` to the inverse of `f(x)`, moreover, reverse functions can also be called directly, for instance, by `\f(3)`.

The term `\f(x)` is mathamatically equivalent to $f^{-1}(x)$.

5.2.4 Binary Convergence Operator ->

`->` can be used for taking the limit of a function. `f(x -> 0)` takes the limit for 0, `f(x -> 0+)` takes the function's limit approaching from positive side, and `f(x -> 0-)` takes the functions limit from the left side.

5.3 Expression Evaluation

Expressions are evaluated by first resolving the variables used inside by refeering to the closest scope, and then reducing the expressions as per the reducing rules. In most cases, by evaluating the operations between types.

5.3.1 Name Resolution

Names are resolved by using the value of the variable name in the closest scope. If no such variable exists, the `Undefined` literal is emitted.

When evaluating pure variables, this scope is always the global scope, hence evaluation is easy. However, when evaluating a function, global scope might become another function:

```
f(y) = y + 2
y = 2
g(x) = f(x + 2) + 4
g(y)
```

Here, when `g(y)` is called, first the `y` resolves to `2`, then the expression assigned evaluates to `f(2 + 2) + 4`, which reduces to `f(4) + 4`, `f(y)` is then called, where `y` resolves to `4` of the function call and not the `2` from global scope. Following which, `f(4)` evaluates to `4 + 2`, turning the expression into `4 + 2 + 4`, which finally reduces (in two steps) to `10`.

5.4 Macros

Macros are special functions that act on the semantic representation of the source code. They are defined using the extension facilities of the language. Macros cannot be defined by the Fluxion language itself. Fluxion does not come with any built in variables, but its standard library contains some.

5.5 Built-in Functions

Fluxion comes pre-equipped with multiple functions from various fields of mathematics to facilitate ease of use for the user for everyday mathematical tasks.

These functions are overlaid in Table **??**. In this table, the arguments of the functions are translated to their equivalents as x, y, z, t, u in order.

Function	Equivalent	Domain
`sum(from, to, sequence)`	$\sum_x^y z$	$\{\forall x, y \in \mathbb{N} \wedge z \subseteq \mathbb{C} \vert y \geq x\} \to \mathbb{C}$
`prod(from, to, sequence)`	$\prod_x^y z$	$\{\forall x, y \in \mathbb{N} \wedge z \subseteq \mathbb{C} \vert y \geq x\} \to \mathbb{C}$
`isum(sequence)`	$\sum_0^\infty x$	$\forall z \subseteq \mathbb{C} \to \mathbb{C} + \{-\infty, \infty, \texttt{Indeterminate}$
`iprod(sequence)`	$\prod_0^\infty x$	$\forall z \subseteq \mathbb{C} \to \mathbb{C} + \{-\infty, \infty, \texttt{Indeterminate}$
`log(base, value)`	$\log_x y$	$\mathbb{R}^+ \to \mathbb{R}$
`ln(value)`	$\ln x$	$\mathbb{R}^+ \to \mathbb{R}$
`root(base, value)`	$\sqrt[x]{y}$	$\mathbb{C} \to \mathbb{C}$
`sqrt(value)`	$\sqrt{x}$	$\mathbb{C} \to \mathbb{C}$
`sin(x)`	$\sin x$	$(-\infty, +\infty) \to [-1, 1]$
`cos(x)`	$\cos x$	$(-\infty, +\infty) \to [-1, 1]$
`tan(x)`	$\tan x$	$\mathbb{R} - \{\forall k \in \mathbb{Z} \vert \frac{\pi}{2} + k\pi\} \to (-\infty, +\infty)$
`cot(x)`	$\cot x$	$\mathbb{R} - \{\forall k \in \mathbb{Z} \vert k\pi\} \to (-\infty, +\infty)$
`sec(x)`	$\sec x$	$\mathbb{R} - \{\forall k \in \mathbb{Z} \vert \frac{\pi}{2} + k\pi\} \to \mathbb{R} - (-1, 1)$
`csc(x)`	$\csc x$	$\mathbb{R} - \{\forall k \in \mathbb{Z} \vert k\pi\} \to \mathbb{R} - (-1, 1)$
`arcsin(x)`	$\sin^{-1} x$	$[-1, 1] \to [-\frac{pi}{2}, \frac{\pi}{2}]$
`arccos(x)`	$\cos^{-1} x$	$[-1, 1] \to [0, \pi]$
`arctan(x)`	$\tan^{-1} x$	$(-\infty, infty) \to (-\frac{pi}{2}, \frac{\pi}{2})$
`arccot(x)`	$\cot^{-1} x$	$(-\infty, infty) \to (0, \pi)$
`arcsec(x)`	$\sec^{-1} x$	$(-\infty, -1] \cup [1, \infty) \to [0, \frac{\pi}{2}) \cup (\frac{\pi}{2}, \pi]$
`arccsc(x)`	$\csc^{-1} x$	$(-\infty, -1] \cup [1, \infty) \to [-\frac{pi}{2}, 0) \cup (0, \frac{\pi}{2}]$
`Re(complex)`	$\Re(x)$	$\mathbb{C} \to \mathbb{R}$
`Im(complex)`	$\Im(x)$	$\mathbb{C} \to \mathbb{R}$
`gamma(value)`	$\Gamma(x)$	$\{\forall x \in \mathbb{C} \vert \Re(x) > 0\} \to \mathbb{C}$
`floor(num)`	$\lfloor x \rfloor$	$\mathbb{R} \to \mathbb{Z}$
`ceil(num)`	$\lceil x \rceil$	$\mathbb{R} \to \mathbb{Z}$
`modulo(base, num)`	$c := y \mod x$	$\forall y \in \mathbb{R}, x \in \mathbb{Z} \to \mathbb{C}$
`even(num)`	$x = 0 \mod 2$	$\mathbb{Z} \to \{\texttt{True}, \texttt{False}\}$
`maclaurin(function)`	Maclaurin expansion.	–
`taylor(function)`	Taylor expansion.	–
`gcd(x, y)`	$\gcd(x, y)$	$\mathbb{Z}^+ \to \mathbb{Z}^+$
`defInt(function, var, from, to)`	$\int_z^u x dy$	–
`int(function, var)`	$\int x dy$	–

Table 5.1: Builtin functions.

Chapter 6

Reductions

This chapter includes how the Fluxion language reduces expressions to simpler expressions.

Reductions comes in many shapes and forms, some act on simple operation expressions, whereas others may be able to actually reduce builtin functions. Therefore, the reduction rules are covered section by section. Reduction rules are described via mathematical formulae, the characters α, β, λ etc are used to describe any sort of expression unless otherwise specified. A base expression, followed by a $\Rightarrow$ followed by the reduced expression.

Moreover, not all of these reduction rules are valid for all the Fluxion types and in all cases, see the Table 6 conditions for applying these rules.

Rule	Condition
6.1	When α, β is the same type and α, β, λ matrices or integers.
6.2	Same as 6.1
6.3	All numerical
6.4	All numerical, $\lambda \neq 0$
6.5	All numerical.
6.6	$\alpha, \beta \in \mathbb{N}, \beta \geq \alpha$
6.7	$\alpha, \beta \in \mathbb{N}, \alpha \geq \beta$
6.8	$f : S_1 \to S_2 \to f^{-1} : S_2 \to S_1$
6.9	$\lambda \in \mathbb{Z}^+$
6.10	$x, \lambda \in \mathbb{Z}^+$
6.26	$\lambda \in \mathbb{Z}^+$

Table 6.1: Conditions for applying the reduction rules.

6.1 Reductions on Simple Operations

Assume that, α and β are any expressions, moreover, assume that if α and β are both vectors or matrices, γ is their common integer divisor or $\exists\gamma \in \mathbb{C}, \gamma\alpha = \beta$. If not, $\gamma = 1$.

$$\alpha\lambda + \beta\lambda \Rightarrow \left(\frac{\alpha}{\gamma} + \frac{\beta}{\gamma}\right)\gamma\lambda \tag{6.1}$$

$$\alpha\lambda - \beta\lambda \Rightarrow \left(\frac{\alpha}{\gamma} - \frac{\beta}{\gamma}\right)\gamma\lambda \tag{6.2}$$

$$\alpha\lambda^m \times \beta\lambda^n \Rightarrow \alpha\beta\lambda^{m+n} \tag{6.3}$$

$$\lambda \neq 0, \frac{\alpha\lambda^m}{\beta\lambda^n} \Rightarrow \frac{\alpha}{\beta}\lambda^{m-n} \tag{6.4}$$

$$\alpha^\lambda\beta^\lambda \Rightarrow (\alpha\beta)^\lambda \tag{6.5}$$

$$\frac{\beta!}{\alpha!} \Rightarrow (\beta - \alpha)! \tag{6.6}$$

$$\frac{\beta!}{\alpha!} \Rightarrow \frac{1}{(\beta - \alpha)}! \tag{6.7}$$

6.2 Reductions on Functions

$$f(f^{-1}(\lambda)) \Rightarrow \lambda \tag{6.8}$$

6.3 Reductions on Logarithmic Functions

$$e^{\ln\lambda} \Rightarrow \lambda \tag{6.9}$$

$$x^{\log_x\lambda} \Rightarrow \lambda \tag{6.10}$$

6.4 Reductions on Trigonometric Functions

$$\frac{\sin\lambda}{\cos\lambda} \Rightarrow \tan\lambda \tag{6.11}$$
$$\frac{\cos\lambda}{\sin\lambda} \Rightarrow \cot\lambda \tag{6.12}$$
$$\frac{1}{\sin\lambda} \Rightarrow \csc\lambda \tag{6.13}$$
$$\frac{1}{\cos\lambda} \Rightarrow \sec\lambda \tag{6.14}$$
$$\frac{1}{\tan\lambda} \Rightarrow \cot\lambda \tag{6.15}$$
$$\frac{1}{\cot\lambda} \Rightarrow \tan\lambda \tag{6.16}$$
$$\tan\lambda\cos\lambda \Rightarrow \sin\lambda \tag{6.17}$$
$$\cot\lambda\sin\lambda \Rightarrow \cos\lambda \tag{6.18}$$
$$\tan\lambda\csc\lambda \Rightarrow \sec\lambda \tag{6.19}$$
$$\cot\lambda\sec\lambda \Rightarrow \csc\lambda \tag{6.20}$$
$$\cos^2\lambda + \sin^2\lambda \Rightarrow 1 \tag{6.21}$$
$$2\cos\lambda\sin\lambda \Rightarrow \sin(2\lambda) \tag{6.22}$$
$$\cos^2\lambda - \sin^2\lambda \Rightarrow \cos(2\lambda) \tag{6.23}$$
$$1 - 2\cos^2\lambda \Rightarrow \cos(2\lambda) \tag{6.24}$$
$$2\sin^2\lambda - 1 \Rightarrow \cos(2\lambda) \tag{6.25}$$

6.5 Reductions on Gamma Function

$$\Gamma(\lambda) \Rightarrow (\lambda - 1)! \tag{6.26}$$

6.6 Reductions on Sum and Product

$$\sum_{\alpha}^{\beta}\lambda + \sum_{\alpha}^{\beta}\lambda \Rightarrow 2\sum_{\alpha}^{\beta}\lambda \tag{6.27}$$
$$\sum_{\alpha}^{n=\beta}\gamma\lambda \Rightarrow \gamma\sum_{\alpha}^{\beta}\lambda \tag{6.28}$$
$$\prod_{\alpha}^{n=\beta}\gamma \Rightarrow (\beta-\alpha)!\gamma^{\beta-\alpha} \tag{6.29}$$

6.7 Reductions on Integrals

$$\int \lambda'(x)\mathrm{d}x \Rightarrow \lambda(x) + cons \tag{6.30}$$

$$\int_{\alpha}^{\beta} f(x)\mathrm{d}x + \int_{\beta}^{\delta} g(x)\mathrm{d}x \Rightarrow \int_{\alpha}^{\delta} \left(f(x) + g(x)\right)\mathrm{d}x \tag{6.31}$$

Chapter 7

Mathematical Proofs and Functions

7.1 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).