Principia Nota

Henry Blanchette

Version 0.0

Contents

1 Introduction

The point of this compilation is to record and formalize my notations. I am very intrigued by the aesthetics of code, where I mean code in the most general sense:

[DEFINITION] A **code** is a set of consistent and formal rules for recorded expression.

Note that I didn't restrict code to only *computer* code. I think that, although there are obvious and important distinctions between code meant to be read by humans and code meant to read by machines, the distinctions are merely superficial. In my taxonomy, I label *Machine Language* the section for programming languages (usually written by humans) and machine/assembly-esque code.

There is a distinction between what I mean by *code* and what I mean by *language*. In fact, the term *language* actually has a technical meaning in computer science.

2 Rich Text

2.0.1 Header

2.0.2 Plain

2.0.3 Italic

2.0.4 Bold

2.0.5 Bold and Italic

2.0.6 Underline

2.0.7 Math

2.0.8 Code

3 Archaic

4 Philosophy

4.1 Logic

4.1.1 Variable

This is really the fundamental concept of what *variable* really means (although the term is often appropriated for other uses).

[DEFINITION] A **variable** is a

4.1.2 Truth Table

4.2 Metaphysics

4.3 Epistemology

4.4 Ethics

5 Mathematics

5.1 Names

5.1.1 Greek

5.1.2 Latin

5.1.3 Diacratic

5.1.4 Special Symbol

5.1.5 Capitalization

5.1.6 Abbreviation

5.1.7 Length

5.1.8 Scale

5.2 Set Theory

Set Theory is one of the most fundamental theories in mathematics. As per this important role, it reserves some very iconic and widespread notation.

[DEFINITION] A **set** is an collection of things. There is no necessary relation between the things. For things $s_1, s_2, \dots$ the set of just these things is denotes $\{s_1, s_2, \dots\}$. A set with no things is denoted $\{\}$ or $\varnothing$.

Global-scope sets are usually named with capitalized, single letters. This easily leads to the elements of a globally-scoped set usually named with variants on the lower-case of that letter. For example, we could write $S = \{s\}$, or $S = \{s_1, s_2, \dots, s_n\}$. Note that a member of a set could also be a set. This naming convention stays consistent because of the "globally-scoped" aspect. By a *globally-scoped* set name, I mean that the context of the name doesn't frequently refer to or make reference to the set's being a member of other sets.

The kinds of things that can go into a given set are unrestricted. If s is in S, then we write "s is a member of S", "s is an element of S", or with mathematical symbols,

$$s \in S$$

The symbol "$\in$" is derived from a standard "ϵ". In *Arithmetices prinicipia nova methodo exposita* Giuseppe Peano used "ϵ" to abbreviate "est", which is latin for "in" (note that the text was entirely in latin). Later in Bertrand Russel's *Principia Mathematica*, the same meaning was used but the printed ended up looking more like "$\in$", a intended to be a more modern-looking epsilon. The new symbol was adopted over time as distinct from "ϵ" and delegated exclusively to meaning "...is a member of the set ...".

Dealings with sets also get their own notations. Symbols can be reflected across their y-axis to yield the symbol working in the opposite direction. For example, $A \subset B$ can be written as $B \supset A$. The most important set-related notations are

Symbol Instance	Description
$a \in A$	Set membership. a is an element of A.
$A \cup B$	Set union. The smallest set with all the elements of A and all the elements of B. The symbol resembles a "linking" of the two sets, like a chain.
$A \cap B$	Set intersection. The largest set with only the elements of A that are also elements of B. The symbol resembles a filtering of the two sets, like a sifting apparatus or funnel.
$A \setminus B$	Set difference. The set with all the elements of A that are not elements of B. The symbol resembles a long and tilted minus sign.
$A \subset B$	Proper subset. The symbol is an alteration on the $\cup, \cap$, with the sideways tilt indicating that A unioned with some other set would yield B. Specifically, $A \subset B$ usually implies that $A \neq B$. However, some take the convention of allowing $A = B$, in which case their is no technical difference between $A \subset B$ and $A \subseteq B$. The only difference is an emphasis on the possibility that $A = B$.
$A \subseteq B$	Included subset. The symbol is an alteration on $\subset$, adding half of a $=$ underneath to indicate the possibility of $A = B$.
$A = B$	Set equality. Sets are equal if and only if $A \subseteq B$ and $B \subseteq A$.
$\{a \in A \mid a \text{ has property } P\}$	Set definition idiom. This translates to "the largest subset of A in which every element has property P". Note that P is extremeley flexible. This notation for defining sets is extremeley useful and concise. Sometimes, the "$\in A$" to the left of "$\mid$" is not written and is either implied by "P" or assumed to be unrestricted to any particular set.
$A \times B$	Cartesian Product. Formally, $A \times B = \{(a, b) \mid a \in A, b \in B\}$. $A \times B$ is pronounced "the Cartesian cross-product of A and B," but the "Cartesian" adjective is implicit. The "cross" idea is to imagine that A and B are perpendicular axes, forming a cross, and yielding a grid where each point in the grid is labeled by an (a, b) where $a \in A, b \in B$. This is the *ordered pairs* cross the sets. An ordered pair written as (a, b) which is a notation for $\{a, \{a, b\}\}$. The set construction indicates ordering by having the "second" element only appear in the set that is the second element of (a, b). This set construction works well for ordered pairs, but it difficult to extend to the more generic *n-tuple*. I will cover this later.
A/R	Set quotient, where A is a set and $R : A \to A$ is an equivalence relation. With these, $A/R = \{\{a' \in A \mid r(a) = a' \mid a \in A\}$. In English, A/R is the set of subsets of A in which each element is related to every other by R (*equivalence class*).

Many of these symbols are the most fundamental symbols of Mathematics. This reflects well Set Theory's place within the modern mathematical discipline. Almost everything can be expanded out to a set representation. In terms of notation, this is very fortunate. Many new mathematical ideas can use shorthand from or inspired by Set Theory. This provides a sort of "atomization" of mathematical notation, where there is a genealogy of similarity from the basic, set-theoretical terms to extremely specific and contextual notation. For example, in algebra, many symbols such as related:

$$\times, /, =$$

The advantages of the Set Theory framework are intuitiveness, universality, and its atomic-nature as I mentioned. Are there things, however, that expand beyond the reach set-theoretical notation? There is always Russell's famous [1]

$$PARADOX := \{A \mid A \text{ is a set that is not a member of itself}\}.$$

The question to ask of *PARADOX* is "is *PARADOX* a member of itself?"

5.3 Function

5.4 Measure Theory

5.5 Number Theory

5.6 Algebra

5.7 Analysis

5.8 Calculus

5.9 Graph Theory

6 Computarelogy

etymology of the word computer, as well as inspiration for the new word for computer science on the mathematical side. Perhaps "Computarelogy"?

6.1 Binary

Binary is the base-2 number-representation format. The name "binary" comes from latin, *binarius*, which means "consisting of two". Indeed, the symbols allowed in binary numbers is a *binarius* set: $\{0, 1\}$. Note that binary numbers don't have different *values* from, from example, decimal numbers. The convention for number-representation formats using the arabic numerals is that the value of each number increases from right to left. The rightmost repesents a number of 2^0s, and for each slot 2^i, the slot immediately to the left of it is 2^{i+1}. You may abstractly think of a number in this kind of format as a string that stretches infinitely leftward from a rightmost origin, where each slot contains a numeral (in the case of binary, a 0 or a 1).

[DEFINITION] A **bit** is a single 0 or 1, not necessarily interpreted in the context of representing a longer binary-formatted number.

In terms of notation, this abstract representation is inefficient because we most often represent numbers that require only a few slots, and be waste our time always considering the infite string of 0s on the left-tail of each number-representation when we can represent all of those zeros consicely by not writing them at all. In general written notation, any zero that is the leftmost numeral in a number-representation may be removed with no interpretive effect.

In the context of Computarelogy however, we sometimes do desire a few extra "trailing" zeros in our notation. This is because here, number values exist in the context of memory which is limited and usually pre-designated. If a value is 8-bits, then we want to write 000010 rather than 10 to indicate its 8-bit designation.

[DEFINITION] An **n-bit** value is one that could only be one of 2^n possible values. This yields that is a bijection between each of these possible values and the numbers that can be represented using at maximum 8 slots in binary-form: $0, \ldots, 2^n - 1$.

Note that there is not just one binary format. Any bijection between n-bit values and the numbers $0, \ldots, 2^n - 1$.

6.2 System

6.2.1 File System

6.2.2

6.3 Pseudocode

6.4 Programming

This section focusses on the standards and conventions in computer programming. Many standards are enforced by programming languages and many conventions are not, and visa versa. Many conventions come in different flavors. However, for any particular program the convention choices are expected to be consistent across all of the code.

I focus here on standards and conventions that are widespread across many languages and use cases, rather than language specific ones. However, as I reoterate later on, many language-specific constructions are inspired by broader coded themes.

6.4.1 Special Symbol

Notation in programming languages often uses special symbols that resemble the look and function of mathematics ones, as well as adding a few new function to available ASCII characters. Some symbols have multiple distinct meanings, which are seperated by commas in the right column of the table below.

Note that many programming languages devise their own special symbols or combinations of symbols that are language-specific and non-standard. This list records only the standard associations of the symbols across languages and academia. Additionally, many language-specific instances draw on these standards. For example, Python uses `//` for integer division, whereas it is standardly used to begin in-line comments. Why do languages differ is such odd, arbitrary ways? I don't know, however I may address this question later.

Symbol	Meaning
+	numeric addition, list join
-	numeric subtraction, cabob-case
/	numeric division
*	numeric multiplication, type product, pointer type
^	numeric power-of, bitwise XOR
%	numeric modulo
&	bitwise AND, reference-of (pointers context)
|	bitwise OR, case matching divider
&&	boolean AND
||	boolean OR
!	boolean NOT
~	bit inversion
<<	bits shift left
>>	bits shift right
<	numberic less than
<=	numeric less than or equal to
>	numeric greater than
>=	numeric greater than or equal to
?	conditional ternary
=	value assignment, token equality
:=	value assignment
==	value equality
->	function type constructor
=>	anonymous function, "maps to"
,	tuple
;	end of statement
#	in-line comment begin
//	in-line comment begin
/* */	multi-line comment block
\	begin escaped word
' '	a single of character type, unformatted string
" "	formatted string
( )	function call with enclosed parameters, associative grouping
[ ]	indexing operator (comes after target, encloses index value)
{ }	block with new scope, object body
< >	special parameters, type parameters
$	associative divide
.	scope tree name divider

6.4.2 Naming

In programming almost everything needs a name. And every name carries with it much significance not encompasses solely by its pronunciation. The important factors to consider when judging a naming style are

1. **Readability** is how easily one can recognize the important characteristics of names.

2. **Accuracy** is how closely names correspond to the full literal descriptions, including correct grammar and syntax, of their representees.
3. **Brevity** is how efficiently name represent the relevant features of their representees.

Casing. When first reading someone's code, the first thing you will most likely notice is their style of casing. Casing is one of the easiest ways for source-code to convey abstract meaning tied to programming constructs. The dominant flavors of casing are

A) **Camel Case** has the first letter of each word (except, by default, the first word) in a name be upper case, whereas all other letters are lower case. For example, `thisIsAName`. As for the first word, its capitalization is delegated to contextual judgement. This case infamously has discontinuities with words that generally require capital letters and acronyms. For example, `XMLHttpRequest` is a commonly-used javascript function. Javascript standardly adheres to camel case, but this example demonstrates both an inconsistency with camel case and also with its interal casing. Compare it to `XmlHttpRequest` and `XMLHTTPRequest`. The truth is that none of them look particularly aesthetic. The problem is that XML and HTTP are both fully-capitalized acronyms, but together they cross a line of acceptable readability (especially as a function to be used as often as it is). So, the result trades accuracy for readability. In this case, the trade seems to be fine because as a common function people have gotten used to it without much of a problem. For conflicts like this in non-standard code however, instances of this problem become quite a pain on the eyes.

B) **Snake Case** has each word in a name seperated by a _ character. For example, `this_is_a_name`. Snake case is allowed in almost all languages, but is especially standard in C-like languages. Most C/C++ libraries make use of this casing. This casing is unparalled in its support for both readability and accuracy. Since words are completely seperated and _ is a (almost-)universally unrestricted character in names, snake case yields the most standard scheme for casing. All special-casings are also supported by this casing, like screaming-case, capital-case, and all-lower-case. However, snake case can also promote longer names with several words due to just the ease of it. In functional programming, snake case words can sometimes be hard to tell apart due to the _ being in the place of a space, for example `activate_function on_x with_also_y`. Note that, depending on your naming semantic-style, this can be a disadvantage *or* an advantage.

C) **Kebab Case** (aka Cobol Case) has each word in a name is seperated by a `-` character. For example, `this-is-a-name`. Famously, CSS standardizes and Agda requires kebab casing. The immediately-obvious drawback to this casing is the sacrifice of `-`. "How could a language with kebab casing parse subtraction?" you might wonder. Well, its definitely a problem that requires a solution. In CSS names are never occur nakely next to calcultion expressions, so kebab casing never yields any ambiguity. But this is special for CSS, as simply a styling language. For a more fully-featured language like Agda, kebab casing's tendrils influence and are infuenced by the very foundations of the language itself. In order to remove parsing-ambiguity or even reader-ambiguity, Agda requires that all names be seperated by spaces or other restricted characters. You could never write `a-b` to mean subtraction, you must write `a - b` or `(a)-(b)`. A reason for this enforcement is another feature of Agda: using the _ to ease infix notation. Rather than annotating an infix for an operator like `+`, Agda allows _+_ where each _ represents and input around the use of the function.

Capitalization. These are the rules you use when deciding how to capitalize particular letters of a name. Different groups of letters in the word are used in seperate semantic spaces of convention.

1. **Proper Case** is where the first letter of each word in a name is upper-case. Capitalizing these letters signifies that this name is of a container- or structure-like thing. Specifically, the thing named either has children or parts, or is instanciable. Common use-cases are class names, module names, project names. The idea is that proper-cased names are of large, important objects that hold or control many smaller parts. Specifically in functional programming, this capitalization is used for the names of types. This choice is motivated by the facts that types and tokens are extremeley important to distinguish in a functional setting, as they are often used near name references. Additionally, the idea of a type being a sort of "container" of its tokens carries over from the more generic use of proper-casing. Overall, this casing is extremeley standard and useful, probably because it implictly rides on associations humans have with capitalized words.

2. **Screaming Case** is where all letters in a name are upper-case, in the like of how a millenial (or younger) internet-forum participant indicates that they intend their words to be *screamed* at the reciever. Screaming casing is almost universally adopted as a standard for naming either enumerated names or constant, global names. These are namse that do not change value and are defined usually in a global, static, or singleton-class context. This case conflicts heavily with camel casing. If restricted to very specific and consistent uses, this case is very useful and effectively seperates "global" and "local" sections of your code.

3. **Lower Case** is where all letters in a name are lower-case. This casing is looked down upon because it sacrifices readability for seemingly no reason. Nevertheless, it makes its way into standard libraries across many languages. It is typically used for untility functions or temporary variable names. Note that one-word names do not count as lower-case unlese the scheme they are written in is consistently lower-case even in multi-worded names.

6.4.3 Comment

6.5 Language

[DEFINITION] A **letter** is a member of an alphabet. Letters are denoted by single ASCII characters. σ is associated with a generic letter.

[DEFINITION] An **alphabet** is a finite set of letters. Σ is often used as the name of either a specific or generic alphabet.

[DEFINITION] A **string** is a n-tuple of letters, where n is the length of the string.

[DEFINITION] A **language** is set of strings that all have letters in some alphabet Σ. L is ogten used as the name f either a specific or generic language. $L(X)$ represents the language yielded by the X, where X is an automata, grammar, or other language-yielding construct.

6.5.1 Context-Free Grammar

[Definition] A context-free grammar (CFG) is a set of rules that yield a language (a context-free language (CFL)) with an alphabet Σ. Formally, a grammar G is given by

$$G = (S, \Sigma \cup \{\epsilon\}, V, R)$$

where S is the start token, $\Sigma \cup \{\epsilon\}$ is a set of terminals, V is an alphabet of non-terminals, and R is the set of rules. It is required that $(\Sigma \cup \{\epsilon\} \cap V = \varnothing)$.

The language is yielded by starting with S and then applying each valid string of rules applications in R^* (applying each string's rules consecutively, in order). In other words, a string is in the yielded language if it can be produced exactly by applying a valid string of rules, each rule consecutively, to S.

[Definition] A **token** is a letter in either Σ or V.

[Definition] If a token is **terminal**, then it cannot be manipulated by rule applications. If a terminal is produced by a rule application in a string, then it will persist till the string is completed. When a string contains only terminals, it is completed.

[Definition] If a token is **non-terminal**, then it must be manipulated by rule applications. A string is not completed until all of the non-terminals have been transformed into terminals by rule appliations.

The rules follow a few conventions.

1. Capital letters reprents non-terminals.
2. Rules are writted $A \to x$ where A is a non-terminal and x is a string of tokens. This represents the rule "any A can be replaced with x".
3. The left-hand-side of each rule must be a single non-terminal. The right-hand-side of each rule must be a string of tokens.
4. There must be at least one rule $S \to x$, where x is a string of tokens.
5. There must be at least one rule $A \to x$, where A is a non-terminal and x is a string of only terminals.
6. For each terminal in $A \in V$, there must be at least one rule $A \to x$, where x is a string of tokens.
7. A subset of R of the form $\{A \to x_1, A \to x_2, \ldots, A \to x_n\}$ can be written as $A \to x_1 \mid x_2 \mid \cdots \mid x_n$. $\mid$ are often used as dividers in case rules like this.

6.5.2 Regular Expression

Regular expressions are a rare instance of practicality derived from this subfield of Computarelogy. A regular expression represents a set of strings. The academic standard for regular expressions is the simple grammar

$S \rightarrow$	$\mid \epsilon$	empty string
	$\mid \sigma_i \mid \cdots \mid \sigma_n$	letter of Σ
	$\mid SS$	concatenation
	$\mid (S \cup S)$	set union
	$\mid (S)^*$	Kleene star: $\{x_1 x_2 \cdots x_n \mid \forall i \geq 0, x_i \in S\}$

Where $\Sigma = \{\sigma_i, \ldots, \sigma_n\}$ is the alphabet of the grammar. In addition to these fundamental rules, there are many other common rules that reduce to applications of the fundamental rules:

$*$	equivelant to Σ^*
$S+$	S^* while additionally requiring $i > 0$
$S?$	S^* while aditionally requiring $i = 0 \vee i = 1$
$S \mid S$	set union
$[\sigma_1 \cdots \sigma_n]$	letter union: $\cup_{i=1}^{n} \sigma_i$
$[\sigma_0 - \sigma_1]$	letters from σ_0 to σ_1 in the standard ordering of Σ
.	anything except newline
\$	end of the string

6.5.3 Automata

6.6 Computability

6.7 Complexity

6.8 Artificial Intelligence

7 Game Design

8 Physics

9 Economics

Notes

[1]https://www.britannica.com/topic/Russells-paradox