NormCode Guide

1. Introduction

NormCode is a formal language used to construct a **plan of inferences**. It is designed to represent complex reasoning and data processing tasks not as a single, monolithic script, but as a structured combination of multiple, distinct inferences.

Each **inference** within the plan is a self-contained logical operation. The entire Normcode script orchestrates how these individual inferences connect and flow to achieve a larger goal.

2. Core Syntax: Inferences, Concepts, and Sequences

The fundamental unit of a Normcode plan is the **inference**. An inference is defined by a functional concept and its associated value concepts.

- **Functional Concept (<=)**: This is the cornerstone of an inference. It "pins down" the inference by defining its core logic, function, or operation. Crucially, the functional concept is responsible for **invoking a sequence** (e.g., `quantifying`, `imperative`), which is the underlying engine that executes the inference's logic.
- **Value Concept (<-)**: This concept provides the concrete data for the inference. It specifies the inputs, outputs, parameters, or results that the functional concept operates on or produces.

The entire plan is represented in a hierarchical, vertical format. An inference begins with a root concept, followed by an indented functional concept (<=) that defines the operation. The value concepts (<-) are then supplied at the same or a more deeply nested level.

A line in Normcode can also have optional annotations for clarity and control:

```
_concept_definition_ | _annotation_ // _comment_
```

- **`_concept_definition_`** : The core functional (<=) or value (<-) statement.

- **`_annotation_`**: Optional metadata following the `|` symbol. This can be a **flow index** (e.g., `1.1.2`), an intended **data reference**, or the name of the invoked **sequence**.
- **`// _comment_`**: Human-readable comments.

Example Structure of an Inference:

```
_concept_to_infer_ // The overall goal of this inference
    <= _functional_concept_defining_the_operation_ |
    quantifying // This invokes the 'quantifying' sequence
    <- _input_value_concept_1_ // This is an input for the
    operation
    <- _input_value_concept_2_
```

Example of a Concrete Inference:

The following snippet from an addition algorithm shows an `imperative` inference.

```
<- {digit sum} | 1.1.2. imperative
    <= ::(sum {1}<$({numbers})%_> and {2}<$({number})%_> to get {3}?
    <- [all {unit place value} of numbers]<:{1}>
    <- {carry-over number}<:{2}>
    <- {sum}?<:{3}>
```

- **Goal**: The overall goal is to produce a `{digit sum}`.
- **Functional Concept (<=)**: The core of the inference is the `::` (imperative) concept, which defines a command to sum two numbers. This invokes the `imperative` sequence.
- **Value Concepts (<-)**: It takes two inputs (`[all {unit place value} of numbers]` and `{carry-over number}`) and specifies one output (`{sum}`).

3. Concept Types

Concepts are the building blocks of NormCode and are divided into two major classes: **Semantical** and **Syntactical**. The core inference operators (`<=` and `<-`) are explained in the Core Syntax section above.

3.1. Semantical Concept Types

Semantical concepts define the core entities and logical constructs of the domain. They can be subdivided into typically non-functional and functional types.

3.1.1. Typically Non-Functional Concept Types

These concepts represent entities, their relationships, or their roles in the inference.

Symbol	Name	Description
`{}`	Object	Represents a generic object or entity.
`<>`	Statement	Represents a proposition or a state of affairs (non-functional).
`[]`	Relation	Represents a relationship between two or more concepts.
`:S:`	Subject	Marks the subject of a relation or statement.
`:>:`	Input	A special type of Subject that marks a concept as an input parameter.
`:<:`	Output	A special type of Subject that marks a concept as an output value.

Examples: - **Object (`{}`):** `{new number pair}` declares a concept that will hold the state of the two numbers as they are processed. - **Statement (<>):** `<all number is 0>` represents a condition that can be evaluated. The `judgement` sequence will determine if this statement is true or false. - **Relation (`[]`):** `[all {unit place value} of numbers]` defines a collection that will hold the digits from a specific place value of the numbers being added.

3.1.2. Typically Functional Concept Types

These concepts define operations or evaluations that often initiate an inference.

Symbol	Name	Description
`({})` or `::()`	Imperative	Represents a command or an action to be executed.

Symbol	Name	Description
`<{}>` or `<...>`	Judgement	Represents an evaluation that results in a boolean-like assessment.

Examples: - **Imperative (`::()`):** `::(get the {1}?<$({remainder})%_> of {2} <$({digit sum})%_> divided by 10)` issues a command to a tool or model to perform a calculation. This is the heart of an `imperative` inference. - **Judgement (`<...>`):** `<= :%(True):<{1}<$({carry-over number})%_> is 0>` evaluates whether the carry-over is zero. This is the core of a `judgement` inference. Note the use of `<...>` as a common syntax variant for a judgement.

3.2. Syntactical Concept Types

Syntactical concepts are operators that control the logic, flow, and manipulation of data within the plan of inferences. They are grouped by their function.

3.2.1. Assigning Operators

Symbol	Name	Description
`$=`	Identity	Assigns a value to a concept, often used for state updates.
`$.`	Specification	Specifies or isolates a particular property of a concept.
`$%`	Abstraction	Creates a general template from a concrete instance for reuse.
`$+`	Continuation	Appends or adds to a concept, often used in loops to update state.

Examples: - **Identity (`$=`):** `<- {number pair}<$={1}>` is used to give a stable identity (`1`) to the `{number pair}` concept across multiple steps of the algorithm. - **Specification (`$.`):** `<= $.({remainder})` specifies that this part of the inference is focused solely on defining the `{remainder}`. - **Abstraction (`$%`):** `<$([all {unit place value} of numbers])%_>` takes the concrete list of digits and abstracts it as an input parameter for the `sum` imperative. - **Continuation (`$+`):** `<= $+({number pair to append}:{number pair})` defines an operation that updates the `{number pair}` for the next iteration of a loop.

3.2.2. Timing (Sequencing) Operators

Symbol	Name	Description
`@if`	If	Executes if a condition is true.
`@if!`	If Not	Executes if a condition is false.
`@after`	After	Executes after a preceding step is complete.

Examples: - **If / If Not (`@if` , `@if!`):** The combination `@if!(<all number is 0>)` and `@if(<carry-over number is 0>)` forms the termination condition for the main loop, ensuring it continues as long as there are digits to process or a carry-over exists. - **After (`@after`):** `<= @after({digit sum})` ensures that the remainder is only calculated *after* the `digit sum` has been computed in a prior step.

3.2.3. Grouping Operators

Symbol	Name	Description
`&in`	In	Groups items contained within a larger collection.
`&across`	Across	Groups items by iterating across a collection.

Examples: - **Across (`&across`):** `<= &across({unit place value}:{number pair}*1)` is a `grouping` inference that iterates across the two numbers in the `{number pair}` and extracts the `{unit place value}` (the rightmost digit) from each, creating a new group of digits to be summed. - **In (`&in`):** Used to create a collection from explicitly listed value concepts. The elements to be grouped are provided as `<-` concepts within the inference. `Normcode <- [my collection] | grouping <= &in({item}) <- {item A} <- {item B}`

3.2.4. Quantifying (Listing) Operators

Symbol	Name	Description
`*every`	Every	Iterates over every item in a collection (a loop).

Example: - **Every (`*every`):** `<= *every({number pair})` defines the main loop of the addition algorithm. This functional concept invokes the `quantifying` sequence, which will continue to execute its child inferences as long as the termination condition (defined with `@if` operators) is not met.

3.2.5. Concept Markers

These markers can be appended to concepts to modify their meaning.

Symbol	Name	Description
`?`	Conception Query	Appended to a concept to query its value or definition. E.g., `{sum}?`.
`<:_number_>`	Value Position	To link a positional placeholder for values (e.g., `<:{1}>`, `<:{2}>`).
`<$(_concept_)%_>`	Instance Marker	Marks a concept as an instance of an "umbrella" concept. E.g. `<$({number})%_>`
`<$={_number_}>`	Identity Marker	Identifies the same concept across different occurrences. E.g. `<$={1}>`
`%:[_concept_]`	Axis Specifier	Specifies the `by_axis` for an operation. E.g. `%:[{number pair}]`
`@(_number_)`	Quantifier Index	Specifies the index for a quantifier (`*every`) operation. E.g. `@(1)`
`*_number_`	Quantifier Version	Links a concept to a specific quantifier iteration. E.g. `{number pair}*1`

4. Reference of Concepts

While concepts define the structure of a NormCode plan, the actual information is stored in a **Reference**. Understanding the Reference is key to grasping how data is stored, manipulated, and flows through the plan.

A Reference is the container where the information for a concept is kept. Specifically, it is the **semantical concepts** (like `{object}`, `[]`, or `<>`) that have an associated Reference, as they represent the data-holding entities within the plan.

Key Characteristics of a Reference:

- **Multi-dimensional Container**: A Reference is multi-dimensional because a concept can exist in multiple contexts. Each context can introduce a new

dimension, allowing the Reference to hold different instances or elements of the concept's information in a structured way.

- **Named Axes**: Each dimension, often corresponding to a specific context, is represented as a named **axis**. This allows for clear organization and retrieval of information. For example, a concept like `{grade}` could have a Reference with axes named `student` and `assignment`, representing the different contexts in which a grade exists.
- **Shape**: The size of each dimension defines the `shape` of the information within the Reference.
- **Data Manipulation**: The core logic of the **Sequences** (e.g., collecting items with `grouping` or accumulating results with `quantifying`) involves manipulating the information held within these References.

Conceptual Example: The `{grade}` Concept

Imagine you have a concept defined as `{grade}`. This concept represents the idea of a grade in your plan.

- **The `Concept`**: This is the semantic declaration `{grade}`. It's abstract and doesn't hold any specific grade values on its own.
- **The Contexts**: A grade is meaningless without context. It needs to be associated with a `student` and an `assignment`. These two contexts are what give a specific grade its identity.
- **The `Reference`**: The `Reference` for `{grade}` is where the actual grade values are stored. Because the concept has two contexts (`student` and `assignment`), its Reference will be a two-dimensional container with two named axes:
  - `axis: student`
  - `axis: assignment`

  This creates a structure like a table where you can look up a specific grade by providing a value for each axis:

	assignment 1	assignment 2
student A	95	88
student B	72	91

In this structure, the value `88` is the information stored in the `{grade}` concept's Reference at the intersection of `student = student A` and `assignment = assignment 2`. When an inference needs the grade for Student A on Assignment 2, it queries the `Reference` using these axes.

In short, a `Concept` gives information its meaning within the plan, while a `Reference` provides the structure to hold and organize that information.

5. Sequences

Sequences are pre-defined pipelines that are invoked by functional concepts to execute different types of logical operations.

Common sequences include: - **`quantifying`**: Manages loops and iteration (`*every`). - **`grouping`**: Handles data collection (`&across`, `&in`). - **`assigning`**: Manages variable assignment (`$=`, `$+`). - **`imperative`**: Executes complex commands, often with external tools (`::`). - **`judgement`**: Evaluates conditions (`<>`). - **`timing`**: Controls conditional logic (`@if`, `@while`). - **`simple`**: Dummy inference.