Technical Specifications: Omega-Code Pseudocode Meta-Language (v1.4 - Core System Entities & Inherent Actions)

Document Version: 1.4 (Final Formal Specification with Comprehensive EBNF, Semantics, and Core Entities)
Date: June 15, 2025
Author: LLM-Driven System Specification Generator
Purpose: This document provides the comprehensive technical specifications for 'Omega-Code', a formally verifiable pseudocode meta-language. It details the language's syntax using Extended Backus-Naur Form (EBNF), defines its formal semantics, and provides illustrative examples. This specification serves as the foundational blueprint for understanding, writing, and implementing Omega-Code.

1. Introduction & Project Goal

Omega-Code is a highly ambitious pseudocode language designed for the unambiguous specification and reasoning about complex, adaptive, and autonomous systems. Its purpose spans diverse domains, including the design of operating systems, programming languages, robotics, swarm intelligence, deep space probes, and even hypothetical communication with extraterrestrial intelligence. A core objective is future-proofing, ensuring flexibility for unanticipated needs.
Omega-Code adheres to the principles of **Unambiguous Specification**, **Engineered for Evolution**, and **Inherent Governance**. It is grounded in **principled realism**, formally declaring its boundaries and constraints. Its core architecture is maximally simple and universal, supporting extensibility and meta-cognition.

2. Overall EBNF Grammar

2.1. EBNF Notation Conventions

This document uses the following Extended Backus-Naur Form (EBNF) notation:

- ::= : "is defined as" or "consists of"
- | : "or" (separates alternatives)
- [item] : Optional item (appears zero or one time)
- { item } : Item that can be repeated zero or more times
- (item1 item2) : Grouping of items
- 'literal' : Terminal symbol (must appear exactly as shown)
- <non_terminal> : Non-terminal symbol (defined by another rule)
- (* comment *) : Comments within the EBNF grammar itself

2.2. Complete Omega-Code EBNF Grammar

```
(* Top-level Structure *)
OmegaCode ::= { ModuleDefinition | Statement | Comment }

(* Module Definition *)
ModuleDefinition ::= 'MODULE' <ModuleName> <Block> 'END MODULE'
ModuleName ::= <Identifier>

(* Block Structure: A sequence of statements or comments *)
Block ::= { Statement | Comment }

(* Statements can be simple or compound. Comments can appear anywhere. *)
Statement ::= SimpleStatement ';'
        | CompoundStatement
        | PrimitiveCall ';' (* Primitive calls are standalone statements *)

SimpleStatement ::= Assignment
            | ReturnStatement
            | FunctionCall
            | ActionCall (* For inherent operations like LOG, INCREMENT *)
            | 'BREAK'
            | 'CONTINUE'

CompoundStatement ::= IfStatement
              | LoopStatement
              | FunctionDefinition
              | VariableDeclaration (* Variable declarations can be top-level statements *)

Assignment ::= <VariableName> 'ASSIGN' <Expression>
ReturnStatement ::= 'RETURN' [ <Expression> ]

(* Control Flow *)
IfStatement ::= 'IF' <BooleanExpression> 'THEN' <Block> [ 'ELSE' <Block> ] 'END IF'
LoopStatement ::= 'LOOP' ( 'WHILE' <BooleanExpression> | 'FOR' <VariableName> 'IN' <Range>
) <Block> 'END LOOP'
Range ::= <Expression> 'TO' <Expression> | <Identifier> (* e.g., '1 TO 10', 'list_of_items' *)

(* Function Definition *)
FunctionDefinition ::= 'FUNCTION' <FunctionName> '(' [ <ParameterList> ] ')' [ 'RETURNS'
<ReturnType> ] <Block> 'END FUNCTION'
FunctionName ::= <Identifier>
ParameterList ::= <Parameter> { ',' <Parameter> }
```

```
Parameter ::= <ParameterName> ':' <DataType>
ParameterName ::= <Identifier>
ReturnType ::= <DataType> | 'VOID'

(* Variable Declaration *)
VariableDeclaration ::= ( 'DECLARE' | 'VAR' ) <VariableName> [ ':' <DataType> ] [ 'AS'
<InitialValue> ]
VariableName ::= <Identifier>
InitialValue ::= <Expression>

(* Primitive Data Types and Expressions *)
DataType ::= 'BOOLEAN' | 'INTEGER' | 'FLOAT' | 'STRING' | <SchemaID> | <Identifier> (* for
user-defined types *)
Expression ::= <Literal> | <VariableName> | <FunctionCall> | <OperatorExpression>
Literal ::= 'TRUE' | 'FALSE' | <IntegerLiteral> | <FloatLiteral> | <StringLiteral>
IntegerLiteral ::= <Digit> { <Digit> }
FloatLiteral ::= <IntegerLiteral> '.' <IntegerLiteral>
StringLiteral ::= '"' { <CharInString> } '"' (* Changed to CharInString *)
Char ::= (* any Unicode character *)
Digit ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
Letter ::= 'a' | ... | 'z' | 'A' | ... | 'Z'
Identifier ::= <Letter> { <Letter> | <Digit> | '_' }

OperatorExpression ::= <Operand> <Operator> <Operand> | <UnaryOperator> <Operand>
Operand ::= <Expression> | '(' <Expression> ')'
Operator ::= '+' | '-' | '*' | '/' | '==' | '!=' | '<' | '>' | '<=' | '>=' | 'AND' | 'OR' | 'NOT'
UnaryOperator ::= 'NOT' | '-'

FunctionCall ::= <FunctionName> '(' [ <ArgumentList> ] ')'
ArgumentList ::= <Expression> { ',' <Expression> }

ActionCall ::= <ActionID> '(' [ <ArgumentList> ] ')' (* For inherent operations like LOG,
INCREMENT *)

(* Boolean Expressions (for Conditions and Predicates) *)
BooleanExpression ::= <Expression> ( '==' | '!=' | '<' | '>' | '<=' | '>=' ) <Expression>
                    | 'NOT' <BooleanExpression>
                    | <BooleanExpression> 'AND' <BooleanExpression>
                    | <BooleanExpression> 'OR' <BooleanExpression>
                    | '(' <BooleanExpression> ')'
                    | <FunctionCall> (* if function returns BOOLEAN *)

(* Numeric Expressions (for Quantities and Priorities) *)
```

```
NumericExpression ::= <IntegerLiteral> | <FloatLiteral>
                    | <VariableName> (* resolves to numeric type *)
                    | '(' <NumericExpression> ')'
                    | <NumericExpression> <ArithmeticOperator> <NumericExpression>
                    | <UnaryNumericOperator> <NumericExpression>
ArithmeticOperator ::= '+' | '-' | '*' | '/'
UnaryNumericOperator ::= '-'

(* Characters for specific contexts *)
CharInString ::= (* any Unicode character except '"' *)
CharInComment ::= (* any Unicode character except newline, or '*' followed by ')' for
BlockComment *)

(* Comments *)
Comment ::= BlockComment | LineComment
BlockComment ::= '(*' { <CharInComment> } '*)'
LineComment ::= '--' { <CharInComment> } <Newline>
Newline ::= '\n' | '\r\n' | '\r'

(* Identifiers for Primitives and Entities *)
ContextID ::= <Identifier>
EntityID ::= <Identifier>
TemporalRelationTypeID ::= <Identifier>
IntervalDefinitionID ::= <Identifier>
BoundID ::= <Identifier>
ResourceTypeID ::= <Identifier>
MetricID ::= <Identifier>
PolicyID ::= <Identifier>
InterfaceID ::= <Identifier>
InteractionTypeID ::= <Identifier>
SchemaID ::= <Identifier>
ActionID ::= <Identifier> (* Named inherent language operation *)
ProtocolID ::= <Identifier>
RuleID ::= <Identifier>
ScopeID ::= <Identifier>
Value ::= <NumericExpression> (* for Priority and other simple values where numeric is
expected *)
PointID ::= <Identifier>
TargetTypeID ::= <Identifier>
SelfReferencePointID ::= <Identifier>
TransformAction ::= <ActionID> (* Refers to a named action for mutation *)
ModelID ::= <Identifier> (* for UncertaintyModel *)
```

```
(* Revised Schema Definition: defines fields within a data type schema *)
SchemaDefinition ::= '(' { FieldDefinition } ')'
FieldDefinition ::= ( 'VAR' | 'FIELD' | 'PROPERTY' ) <FieldName> ':' <DataType> [ 'AS'
<InitialValue> ] ';'
FieldName ::= <Identifier>
Quantity ::= <NumericExpression> (* for Threshold *)


(* Primitive Calls - Detailed in Section 5 *)
PrimitiveCall ::= ContextRuleCall
                | TemporalRelationCall
                | ResourceBoundCall
                | EnvironmentInterfacePointCall
                | DataTypeSchemaCall
                | StateTransitionCall
                | TrustElementCall
                | GovernanceRuleCall
                | SelfReferencePointCall
                | MutationRuleCall
                | PerceptionMapCall
                | LearningAxiomCall
                | MetaDefinitionRuleCall


ContextRuleCall ::= 'CONTEXT_RULE' <ContextID> ':' 'MODALITY' '{' <ModalityType> { ','
<ModalityType> } '}' [ 'INCOHERENCE_TOLERANCE' <Expression> ]
ModalityType ::= <Identifier>

TemporalRelationCall ::= 'TEMPORAL_RELATION' <RelationID> ':' 'SUBJECT_A' <EntityID> ','
'SUBJECT_B' <EntityID> ',' 'TYPE' <TemporalRelationTypeID> [ 'INTERVAL_A'
<IntervalDefinitionID> ] [ 'INTERVAL_B' <IntervalDefinitionID> ]

ResourceBoundCall ::= 'RESOURCE_BOUND' <BoundID> ':' 'SUBJECT' <EntityID> ',' 'TYPE'
<ResourceTypeID> ',' 'METRIC' <MetricID> ',' 'THRESHOLD' <NumericExpression> ','
'VIOLATION_POLICY' <PolicyID>

EnvironmentInterfacePointCall ::= 'ENVIRONMENT_INTERFACE_POINT' <InterfaceID> ':'
'SUBJECT' <EntityID> ',' 'EXTERNAL_REFERENT' <EntityID> ',' 'INTERACTION_TYPE'
<InteractionTypeID> ',' 'DATA_SCHEMA' <SchemaID> [ 'UNCERTAINTY_MODEL' <ModelID> ]

DataTypeSchemaCall ::= 'DATA_TYPE_SCHEMA' <SchemaID> ':' 'DEFINITION'
<SchemaDefinition> [ 'SEMANTIC_PROPERTIES' '{' <PropertyID> { ',' <PropertyID> } '}' ]
```

StateTransitionCall ::= 'STATE_TRANSITION' <TransitionID> ':' 'SUBJECT' <EntityID> ',' 'PRECONDITION' <BooleanExpression> ',' 'POSTCONDITION' <BooleanExpression> ',' 'ACTION' <ActionID> ['REVERSION_PROTOCOL' <ProtocolID>]

TrustElementCall ::= 'TRUST_ELEMENT' <ElementID> ':' 'SUBJECT' <EntityID> ',' 'PREDICATE' <BooleanExpression> ',' 'OBJECT' <EntityID> ['PROOF_PROTOCOL' <ProtocolID>]

GovernanceRuleCall ::= 'GOVERNANCE_RULE' <RuleID> ':' 'SCOPE' <ScopeID> ',' 'PREDICATE' <BooleanExpression> ',' 'ENFORCEMENT_CONTEXT' <ContextID> ['PRIORITY' <NumericExpression>]

SelfReferencePointCall ::= 'SELF_REFERENCE_POINT' <PointID> ':' 'TARGET_TYPE' <TargetTypeID> ',' 'ACCESS_PROTOCOL' <ProtocolID>

MutationRuleCall ::= 'MUTATION_RULE' <RuleID> ':' 'TARGET_REFERENCE' <SelfReferencePointID> ',' 'CONDITION' <BooleanExpression> ',' 'TRANSFORM_ACTION' <ActionID> ['APPROVAL_POLICY' <PolicyID>]

PerceptionMapCall ::= 'PERCEPTION_MAP' <MapID> ':' 'INPUT_INTERFACE' <InterfaceID> ',' 'OUTPUT_SCHEMA' <SchemaID> ',' 'TRANSFORMATION_FUNCTION' <ActionID> ['UNCERTAINTY_MODEL' <ModelID>]

LearningAxiomCall ::= 'LEARNING_AXIOM' <AxiomID> ':' 'INPUT_SCHEMA' <SchemaID> ',' 'OUTPUT_SCHEMA' <SchemaID> ',' 'OBJECTIVE_METRIC' <MetricID> ',' 'CONSTRAINT_SET' '{' <ResourceBoundID> { ',' <ResourceBoundID> } '}' ',' 'KNOWLEDGE_UPDATE_RULE' <RuleID>

MetaDefinitionRuleCall ::= 'META_DEFINITION_RULE' <RuleID> ':' 'TARGET_TYPE' <TargetTypeID> ',' 'DEFINITION_SCHEMA' <SchemaDefinition> ',' 'VALIDATION_PROTOCOL' <ProtocolID>

3. Lexical Structure

Omega-Code specifications are composed of a sequence of tokens, which are the smallest meaningful units of the language.

3.1. Identifiers

- Identifiers are used for names of modules, functions, variables, and all IDs (e.g., ContextID, EntityID, RuleID).
- They must begin with a letter (a-z, A-Z) and can be followed by letters, digits (0-9), or underscores (_).
- Identifiers are case-sensitive.

- **Formal EBNF Syntax:** <Identifier> ::= <Letter> { <Letter> | <Digit> | '_' }

3.2. Keywords

- Keywords are reserved words that have special meaning in Omega-Code. They are typically written in uppercase.
- Examples: 'MODULE', 'IF', 'THEN', 'ELSE', 'LOOP', 'FUNCTION', 'DECLARE', 'VAR', 'BOOLEAN', 'INTEGER', 'FLOAT', 'STRING', and all the names of the 13 Atomic Primitives.
- **Formal EBNF Syntax (Examples):** 'MODULE', 'IF', 'CONTEXT_RULE', 'SUBJECT_A', etc. (See overall grammar in Section 2.2 for complete list).

3.3. Literals

- **Boolean Literals:** 'TRUE', 'FALSE'
  - **Formal EBNF Syntax:** 'TRUE' | 'FALSE'
- **Integer Literals:** Sequences of digits (e.g., 123, 0, 42).
  - **Formal EBNF Syntax:** <IntegerLiteral> ::= <Digit> { <Digit> }
- **Float Literals:** Integers followed by a decimal point and more digits (e.g., 3.14, 0.5).
  - **Formal EBNF Syntax:** <FloatLiteral> ::= <IntegerLiteral> '.' <IntegerLiteral>
- **String Literals:** Sequences of characters enclosed in double quotes ("). String literals can contain any Unicode character except a double quote itself (unless escaped).
  - **Formal EBNF Syntax:** <StringLiteral> ::= '"' { <CharInString> } '"' (*Updated to use specific character set*)

3.4. Operators

- **Arithmetic:** +, -, *, /
- **Comparison:** == (equal to), != (not equal to), < (less than), > (greater than), <= (less than or equal to), >= (greater than or equal to)
- **Logical:** 'AND', 'OR', 'NOT' (unary)
- **Formal EBNF Syntax (Examples):** '+', '==', 'AND' (See overall grammar in Section 2.2 for complete list).

3.5. Delimiters and Punctuation

- ( ): Parentheses for grouping expressions or function arguments.
- [ ] : Brackets for optional elements in EBNF notation, or for list/array access in pseudocode examples.
- { } : Curly braces for zero or more repetitions in EBNF notation, or for sets/blocks/lists in pseudocode examples.
- , : Comma for separating elements in lists or parameters.
- : : Colon for type declarations or parameter separation within primitive calls.
- ; : Semicolon to terminate statements.
- **Formal EBNF Syntax (Examples):** '(', ';'

3.6. Comments

- **Block Comments:** Begin with (* and end with *). Can span multiple lines.
    - **Formal EBNF Syntax:** <BlockComment> ::= '(*' { <CharInComment> } '*)' (*Updated to use specific character set*)
- **Line Comments:** Begin with -- and extend to the end of the line.
    - **Formal EBNF Syntax:** <LineComment> ::= '--' { <CharInComment> } <Newline> (*Updated to use specific character set*)
- Comments can appear anywhere whitespace is allowed.

4. Inherent Language Features (Syntax, Semantics, Examples)

These foundational capabilities are intrinsic to Omega-Code, providing the scaffolding upon which the primitives operate.

4.1. Syntax & Grammars

- **Purpose:** The Omega-Code inherently possesses a formally defined and extensible syntax and grammar. This ensures the language is unambiguous and machine-readable, crucial for automated processing and formal verification. The grammar itself can be reasoned about and modified via META_DEFINITION_RULE.
- **Formal EBNF Syntax:** Referenced by the <OmegaCode> non-terminal and related lexical rules in Section 2.2.
- **Formal Semantics:** The semantics of "Syntax & Grammars" means that any valid Omega-Code specification *must* conform to the EBNF rules defined in this document (Section 2.2). The processing system (parser) will accept only inputs that are parsable according to this grammar. META_DEFINITION_RULE operates on these very EBNF rules themselves, allowing the grammar to evolve formally.
- **Concrete Example (Conceptual):**

```
(* This isn't Omega-Code itself, but how Omega-Code implicitly handles its own grammar. *)
META_DEFINITION_RULE 'ExtendSyntaxRule' :
  TARGET_TYPE 'GrammarRule' ,
  DEFINITION_SCHEMA (
    VAR NewRuleName : STRING ;
    VAR RuleDefinition : STRING ;
  ) ,
  VALIDATION_PROTOCOL 'EBNF_ComplianceCheck' ;
```

4.2. Basic Control Flow

- **Purpose:** Provides core constructs for sequential execution, conditional branching (IF/ELSE), and iterative execution (LOOP), essential for expressing algorithmic logic.

- **Formal EBNF Syntax:**
  - IfStatement ::= 'IF' <BooleanExpression> 'THEN' <Block> ['ELSE' <Block>] 'END IF'
  - LoopStatement ::= 'LOOP' ('WHILE' <BooleanExpression> | 'FOR' <VariableName> 'IN' <Range>) <Block> 'END LOOP'
  - StatementSequence ::= <Statement> { <Statement> }
  - Refer to Statement and Block rules in Section 2.2.
- **Formal Semantics:**
  - **IF**: Evaluates <BooleanExpression>. If TRUE, executes <Block> after THEN. If FALSE and ELSE is present, executes <Block> after ELSE. Otherwise, continues with the next statement.
  - **LOOP WHILE**: Continuously evaluates <BooleanExpression>. If TRUE, executes <Block>. Repeats until <BooleanExpression> is FALSE.
  - **LOOP FOR**: Iterates <VariableName> through each item in <Range>. For each item, executes <Block>.
  - **BREAK**: Immediately exits the innermost LOOP statement.
  - **CONTINUE**: Skips the rest of the current iteration of the innermost LOOP and proceeds to the next iteration (or re-evaluates condition).
  - **StatementSequence**: Statements are executed one after another in the order they appear.
- **Concrete Examples:**

```
VAR user_active : BOOLEAN AS TRUE;
VAR retry_count : INTEGER AS 0;

LOOP WHILE user_active AND retry_count < 3 THEN
  IF user_input == "quit" THEN
    VAR user_active : BOOLEAN AS FALSE; (* Declare and assign to local scope *)
    BREAK;
  ELSE
    INCREMENT_ACTION (retry_count); (* Call inherent action to increment *)
  END IF;
END LOOP;
```

4.3. Function/Procedure Definition

- **Purpose:** Enables modularity and reusability of logic by encapsulating computational blocks.
- **Formal EBNF Syntax:**
  - FunctionDefinition ::= 'FUNCTION' <FunctionName> '(' [<ParameterList>] ')' ['RETURNS' <ReturnType>] <Block> 'END FUNCTION'
  - ParameterList ::= <Parameter> { ',' <Parameter> }
  - Parameter ::= <ParameterName> ':' <DataType>
  - Refer to <FunctionName>, <Block>, <ParameterName>, <DataType>, <ReturnType>

rules in Section 2.2.

- **Formal Semantics:** Defines a named, callable block of logic. Parameters are passed by value unless otherwise specified (future extension). Execution of the function proceeds through its <Block>. If RETURNS is specified, the function must conclude with a RETURN statement providing a value of the specified <ReturnType>.
- **Concrete Examples:**

```
FUNCTION calculate_area (length : FLOAT, width : FLOAT) RETURNS FLOAT
  VAR area : FLOAT AS 0.0;
  area ASSIGN length * width;
  RETURN area;
END FUNCTION;

FUNCTION log_message (message : STRING)
  LOG_ACTION (message); (* Call inherent action to log *)
END FUNCTION;
```

4.4. Variable Declaration & Scoping

- **Purpose:** Provides standard mechanisms for defining, storing, and managing data, ensuring clarity on data lifecycle.
- **Formal EBNF Syntax:**
  - VariableDeclaration ::= ('DECLARE' | 'VAR') <VariableName> [':' <DataType>] ['AS' <InitialValue>]
  - Refer to <VariableName>, <DataType>, <InitialValue> rules in Section 2.2.
- **Formal Semantics:**
  - **DECLARE / VAR**: Creates a new variable. Variables are lexically scoped (visible only within the block they are declared, and its nested blocks). If AS <InitialValue> is provided, the variable is initialized with that value. If no initial value, it is assigned a default (e.g., FALSE for BOOLEAN, 0 for INTEGER/FLOAT, "" for STRING).
  - Variables are mutable by default.
- **Concrete Examples:**

```
MODULE DataProcessing
  VAR global_counter : INTEGER AS 0; (* Module-scoped variable *)

  FUNCTION process_item (item_value : INTEGER) RETURNS BOOLEAN
    DECLARE temp_result : BOOLEAN; (* Function-scoped variable *)
    temp_result ASSIGN (item_value > 100);
    RETURN temp_result;
  END FUNCTION;
END MODULE;
```

4.5. Module/Namespace System

- **Purpose:** Provides hierarchical organization for large Omega-Code specifications, preventing naming conflicts and aiding in modular design and radical consolidation.
- **Formal EBNF Syntax:**
  - ModuleDefinition ::= 'MODULE' <ModuleName> <Block> 'END MODULE'
  - Refer to <ModuleName>, <Block> rules in Section 2.2.
- **Formal Semantics:** Modules define named organizational units. Identifiers declared within a module are scoped to that module. Modules can be nested. External modules or elements can be referenced using a qualified name (e.g., ModuleName.Identifier).
- **Concrete Examples:**

```
MODULE CoreUtilities
  FUNCTION get_utc_timestamp () RETURNS STRING
    (* ... logic ... *)
  END FUNCTION;
END MODULE;

MODULE SystemMonitor
  FUNCTION check_health () RETURNS BOOLEAN
    VAR current_time : STRING AS CoreUtilities.get_utc_timestamp();
    (* ... logic using current_time ... *)
    RETURN TRUE;
  END FUNCTION;
END MODULE;
```

4.6. Primitive Data Types

- **Purpose:** Fundamental data representations are natively supported, forming the basis for all information representation.
- **Formal EBNF Syntax:**
  - DataType ::= 'BOOLEAN' | 'INTEGER' | 'FLOAT' | 'STRING' | <SchemaID> | <Identifier>
  - Literal ::= 'TRUE' | 'FALSE' | <IntegerLiteral> | <FloatLiteral> | <StringLiteral>
  - Refer to <IntegerLiteral>, <FloatLiteral>, <StringLiteral> rules in Section 2.2.
- **Formal Semantics:**
  - **BOOLEAN**: Represents truth values ('TRUE', 'FALSE').
  - **INTEGER**: Represents whole numbers (e.g., 1, 100, -5).
  - **FLOAT**: Represents real numbers with decimal points (e.g., 3.14, 0.0, -2.5).
  - **STRING**: Represents sequences of characters (e.g., "hello world", "").
  - Other data types can be defined using DATA_TYPE_SCHEMA.
- **Concrete Examples:**

```
VAR is_running : BOOLEAN AS TRUE;
VAR max_retries : INTEGER AS 5;
VAR confidence_score : FLOAT AS 0.95;
VAR user_message : STRING AS "System is operational.";
```

4.7. Entity Identity

- **Purpose:** The language intrinsically supports the concept of a unique EntityID for any identifiable component, agent, or concept within the system and its environment, allowing distinct addressing and referencing.
- **Formal EBNF Syntax:**
  - EntityID ::= <Identifier>
  - Refer to <Identifier> rule in Section 2.2.
- **Formal Semantics:** EntityID is a unique identifier used to refer to distinct entities within the specified system or its environment. This ID allows formal referencing of subjects, objects, interfaces, and other elements across various primitives.
- **Concrete Examples:**

```
(* Defining a rule about a specific user entity *)
GOVERNANCE_RULE UserPrivacyRule :
  SCOPE UserInteractions ,
  PREDICATE ( HasSensitiveData ( user_profile_entity ) ) ,
  ENFORCEMENT_CONTEXT DataHandlingPolicy ,
  PRIORITY 10 ;

VAR system_process_id : EntityID AS Process_XYZ_123;
VAR external_sensor : EntityID AS Sensor_Temp_001;
```

4.8. Formal Verification Engine Hooks

- **Purpose:** The language intrinsically supports linking to and leveraging external formal verification tools. While the specification of verifiable properties is handled by core primitives, the underlying architecture enables hooks for external verification. This is an architectural feature, not a syntactic construct within Omega-Code itself.
- **Formal EBNF Syntax:** No direct syntactic representation in Omega-Code, as these are external integration points. Omega-Code facilitates formal verification *through its precise semantics*.
- **Formal Semantics:** The presence of BooleanExpressions (used as predicates in various primitives like STATE_TRANSITION, GOVERNANCE_RULE) and the ValidationProtocol in META_DEFINITION_RULE provide the semantic basis for formal verification. An external verification tool would consume the Omega-Code specification and, based on these semantic definitions, prove or disprove properties.

- **Concrete Example (Conceptual):**
  (* This is not Omega-Code, but conceptual integration: *)
  (* VERIFY 'AuthenticationProtocol' USING 'TheoremProverX' AGAINST 'Auth_Logic_Spec.omega' *)

  (* Within Omega-Code, the primitives like TRUST_ELEMENT or GOVERNANCE_RULE provide the verifiable statements *)
  TRUST_ELEMENT AuthSuccess :
    SUBJECT User_Alice ,
    PREDICATE (IsAuthenticated (User_Alice)) ,
    OBJECT System_API_Access ,
    PROOF_PROTOCOL ChallengeResponse ;

4.9. Inherent Actions/Operations

- **Purpose:** These are predefined, atomic operations intrinsic to the Omega-Code execution environment that can be invoked via ActionCalls or referenced by ActionIDs in primitives. They represent fundamental computational or interaction behaviors.
- **Formal EBNF Syntax:** ActionCall ::= <ActionID> '(' [<ArgumentList>] ')' (defined in Section 2.2)
  - ActionID refers to the name of an inherent operation.
- **Formal Semantics (Examples):**
  - **INCREMENT_ACTION (<Variable>)**: Increments an integer or float <Variable> by 1.
  - **DECREMENT_ACTION (<Variable>)**: Decrements an integer or float <Variable> by 1.
  - **LOG_ACTION (<Expression>)**: Records the value of <Expression> to a system log.
  - **SEND_MESSAGE_ACTION (<RecipientID>, <MessageContent>)**: Sends a message to a specified recipient.
  - **PARSE_DATA_ACTION (<RawData>, <SchemaID>) RETURNS <SchemaID>**: Parses RawData according to SchemaID and returns a structured data object.
  - **UPDATE_DB_RECORD_ACTION (<TableID>, <RecordID>, <FieldMap>)**: Updates a database record.
  - **QUERY_DB_ACTION (<TableID>, <QueryPredicate>) RETURNS <ResultList>**: Queries a database table.
  - **ADJUST_THRESHOLD_BY_FACTOR (<BoundID>, <Factor>)**: Modifies the threshold of a RESOURCE_BOUND.
  - **EXTRACT_TAGGED_CONTENT (<Text>, <Tag>) RETURNS <String>**: Extracts content from tagged text.
  - **SEND_UNLOCK_COMMAND (<DeviceID>)**: Sends a command to unlock a device.

    - **UPDATE_ORDER_STATUS (<OrderID>, <NewStatus>)**: Updates the status of an order.
    - *(This list is illustrative and would be expanded as needed for a complete standard library of actions.)*
- **Concrete Examples:** (See Section 4.2 for INCREMENT_ACTION, Section 4.3 for LOG_ACTION, and Section 5.3.3 for ADJUST_THRESHOLD_BY_FACTOR for examples of their usage.)

5. The 13 Atomic Core Primitives (Syntax, Semantics, Examples)

These are the fundamental building blocks of Omega-Code, providing its core expressive power.

5.1. Part A: Foundational Semantics & Constraints (4 Primitives)

5.1.1. CONTEXT_RULE

- **Purpose:** Defines a formal context for reasoning, specifying its modalities (e.g., temporal, probabilistic) and tolerance for inconsistency. ModalityTypes are extensible via META_DEFINITION_RULE.
- **Formal EBNF Syntax:** CONTEXT_RULE <ContextID> ':' 'MODALITY' '{' <ModalityType> { ',' <ModalityType> } '}' ['INCOHERENCE_TOLERANCE' <Expression>] (*Updated to show literal braces for list syntax*)
    - Refer to <ContextID>, <ModalityType>, <Expression> rules in Section 2.2.
- **Formal Semantics:** Establishes a named contextual boundary within which subsequent Omega-Code statements are interpreted. MODALITY defines the nature of the context (e.g., Temporal, Probabilistic, Deterministic, Ethical). INCOHERENCE_TOLERANCE specifies the acceptable degree of logical contradiction within this context before a warning or failure is triggered.
- **Concrete Example:**

```
CONTEXT_RULE RealtimeControlContext :
  MODALITY { Temporal , Deterministic }
  INCOHERENCE_TOLERANCE 0.01 ; (* 1% tolerance for timing errors *)

CONTEXT_RULE EthicalDecisionSpace :
  MODALITY { Normative , Probabilistic } ;
```

5.1.2. TEMPORAL_RELATION

- **Purpose:** Defines a fundamental temporal ordering or relationship between any two entities or events. TemporalRelationTypeID and IntervalDefinitionID are extensible via META_DEFINITION_RULE.

- **Formal EBNF Syntax:** TEMPORAL_RELATION <RelationID> ':' 'SUBJECT_A' <EntityID> ',' 'SUBJECT_B' <EntityID> ',' 'TYPE' <TemporalRelationTypeID> ['INTERVAL_A' <IntervalDefinitionID>] ['INTERVAL_B' <IntervalDefinitionID>]
    - Refer to <RelationID>, <EntityID>, <TemporalRelationTypeID>, <IntervalDefinitionID> rules in Section 2.2.
- **Formal Semantics:** Asserts a specific temporal relationship (TYPE) between SUBJECT_A and SUBJECT_B. Examples of TemporalRelationTypeID could be BEFORE, AFTER, OVERLAPS, CONTAINS. INTERVAL_A and INTERVAL_B define specific time durations or points for the subjects. Crucial for causality, synchronization, and planning.
- **Concrete Example:**
  TEMPORAL_RELATION ProcessOrderBeforeShipment :
    SUBJECT_A OrderProcessing ,
    SUBJECT_B ShipmentDispatch ,
    TYPE BEFORE ;

  TEMPORAL_RELATION SensorReadingInterval :
    SUBJECT_A TemperatureSensorData ,
    SUBJECT_B SystemLogEntry ,
    TYPE OVERLAPS ,
    INTERVAL_A CurrentSecond ; (* Assume CurrentSecond is a defined IntervalDefinitionID *)

5.1.3. RESOURCE_BOUND

- **Purpose:** Declares a formal limit on a quantifiable resource associated with an entity. ResourceTypeID and MetricID are extensible via META_DEFINITION_RULE.
- **Formal EBNF Syntax:** RESOURCE_BOUND <BoundID> ':' 'SUBJECT' <EntityID> ',' 'TYPE' <ResourceTypeID> ',' 'METRIC' <MetricID> ',' 'THRESHOLD' <NumericExpression> ',' 'VIOLATION_POLICY' <PolicyID> (*Updated to NumericExpression*)
    - Refer to <BoundID>, <EntityID>, <ResourceTypeID>, <MetricID>, <NumericExpression>, <PolicyID> rules in Section 2.2.
- **Formal Semantics:** Specifies that SUBJECT is limited in its consumption or availability of a RESOURCE_TYPE measured by a METRIC, not exceeding THRESHOLD. If this THRESHOLD is violated, the VIOLATION_POLICY (e.g., LOG_ERROR, HALT_PROCESS, TRIGGER_ALERT) is invoked.
- **Concrete Example:**
  RESOURCE_BOUND MaxMemoryUsage :
    SUBJECT WebserverProcess ,
    TYPE Memory ,
    METRIC Megabytes ,
    THRESHOLD 2048 ,
    VIOLATION_POLICY SendAlertToAdmin ;

RESOURCE_BOUND MaxApiCallsPerMinute :
SUBJECT LLMAgent ,
TYPE ApiCallRate ,
METRIC CallsPerMinute ,
THRESHOLD 60 ,
VIOLATION_POLICY ThrottleRequests ;

5.1.4. ENVIRONMENT_INTERFACE_POINT

- **Purpose:** Defines an atomic point of interaction between a system's internal entity and an external entity/phenomenon, specifying the nature of the interaction and the type of data exchanged. InteractionTypeID is extensible via META_DEFINITION_RULE. DataSchema refers to a DATA_TYPE_SCHEMA.
- **Formal EBNF Syntax:** ENVIRONMENT_INTERFACE_POINT <InterfaceID> ':' 'SUBJECT' <EntityID> ',' 'EXTERNAL_REFERENT' <EntityID> ',' 'INTERACTION_TYPE' <InteractionTypeID> ',' 'DATA_SCHEMA' <SchemaID> ['UNCERTAINTY_MODEL' <ModelID>]
    - Refer to <InterfaceID>, <EntityID>, <InteractionTypeID>, <SchemaID>, <ModelID> rules in Section 2.2.
- **Formal Semantics:** Designates a named interface where SUBJECT interacts with EXTERNAL_REFERENT. INTERACTION_TYPE describes the nature (e.g., ReadSensor, SendCommand, ReceiveMessage). DATA_SCHEMA defines the expected format of information exchanged. An UNCERTAINTY_MODEL can be specified for probabilistic or noisy interactions.
- **Concrete Example:**
  ENVIRONMENT_INTERFACE_POINT TemperatureSensorInput :
  SUBJECT RoboticsSystem ,
  EXTERNAL_REFERENT PhysicalEnvironment ,
  INTERACTION_TYPE SensorRead ,
  DATA_SCHEMA TemperatureDataSchema ,
  UNCERTAINTY_MODEL GaussianNoiseModel ;

  ENVIRONMENT_INTERFACE_POINT UserCommandOutput :
  SUBJECT UserInterfaceModule ,
  EXTERNAL_REFERENT HumanUser ,
  INTERACTION_TYPE DisplayMessage ,
  DATA_SCHEMA DisplayMessageSchema ;

5.2. Part B: Atomic System Constructs (3 Primitives)

5.2.1. DATA_TYPE_SCHEMA

- **Purpose:** Provides the formal schema for defining structured data types and their intrinsic semantic properties. SchemaDefinition uses inherent primitive types and composition. PropertyID is extensible via META_DEFINITION_RULE.
- **Formal EBNF Syntax:** DATA_TYPE_SCHEMA <SchemaID> ':' 'DEFINITION' <SchemaDefinition> ['SEMANTIC_PROPERTIES' '{' <PropertyID> { ',' <PropertyID> } '}'] (*Updated to show literal braces for list syntax*)
    - Refer to <SchemaID>, <SchemaDefinition>, <PropertyID> rules in Section 2.2.
- **Formal Semantics:** Defines a reusable, named data structure. DEFINITION outlines its fields, types, and constraints (similar to struct/record definitions in programming languages). SEMANTIC_PROPERTIES allows tagging with meta-information about the data's meaning (e.g., PII, FinancialData, Encrypted).
- **Concrete Example:**

```
DATA_TYPE_SCHEMA TemperatureDataSchema :
  DEFINITION (
    VAR value : FLOAT ;
    VAR unit : STRING AS "Celsius" ;
    VAR timestamp : STRING ;
  ) ;

DATA_TYPE_SCHEMA UserProfile :
  DEFINITION (
    VAR userId : INTEGER ;
    VAR username : STRING ;
    VAR email : STRING ;
    VAR signupDate : STRING ;
  )
  SEMANTIC_PROPERTIES { PII_IDENTIFIER , CONFIDENTIAL } ;
```

5.2.2. STATE_TRANSITION

- **Purpose:** Defines an atomic change in the state of an entity. ActionID refers to an inherent language operation. ReversionProtocol specifies how the transition might be undone.
- **Formal EBNF Syntax:** STATE_TRANSITION <TransitionID> ':' 'SUBJECT' <EntityID> ',' 'PRECONDITION' <BooleanExpression> ',' 'POSTCONDITION' <BooleanExpression> ',' 'ACTION' <ActionID> ['REVERSION_PROTOCOL' <ProtocolID>]
    - Refer to <TransitionID>, <EntityID>, <BooleanExpression>, <ActionID>, <ProtocolID> rules in Section 2.2.
- **Formal Semantics:** Describes a state change (Action) on SUBJECT. It asserts that the PRECONDITION must be TRUE before ACTION occurs, and the POSTCONDITION will be

TRUE after ACTION completes. REVERSION_PROTOCOL defines a method to revert the state to its pre-transition condition if needed (for rollback or undo).

- **Concrete Example:**
  STATE_TRANSITION ProcessOrder :
  SUBJECT CustomerOrder_123 ,
  PRECONDITION (StatusIsPending (CustomerOrder_123)) ,
  POSTCONDITION (StatusIsProcessing (CustomerOrder_123)) ,
  ACTION UpdateOrderStatusToProcessing ,
  REVERSION_PROTOCOL RollbackOrderStatus ;

  STATE_TRANSITION UnlockDoor :
  SUBJECT SmartLock_A ,
  PRECONDITION (IsLocked (SmartLock_A) AND UserAuthenticated (User_Alice)) ,
  POSTCONDITION (IsUnlocked (SmartLock_A)) ,
  ACTION SendUnlockCommand ;

5.2.3. TRUST_ELEMENT

- **Purpose:** Defines an atomic, verifiable assertion about one entity's relationship or property concerning another. ProofProtocol specifies the method of verification.
- **Formal EBNF Syntax:** TRUST_ELEMENT <ElementID> ':' 'SUBJECT' <EntityID> ',' 'PREDICATE' <BooleanExpression> ',' 'OBJECT' <EntityID> ['PROOF_PROTOCOL' <ProtocolID>]
  - Refer to <ElementID>, <EntityID>, <BooleanExpression>, <ProtocolID> rules in Section 2.2.
- **Formal Semantics:** States a named verifiable assertion where SUBJECT has a PREDICATE relationship/property regarding OBJECT. The PROOF_PROTOCOL (e.g., DigitalSignature, OAuth2, ZeroKnowledgeProof) specifies how this assertion can be verified externally. This forms the basis for security, identity, and distributed consensus.
- **Concrete Example:**
  TRUST_ELEMENT UserIdentityVerified :
  SUBJECT User_Bob ,
  PREDICATE (IsVerified (User_Bob)) ,
  OBJECT AuthenticationService ,
  PROOF_PROTOCOL OAuth2_Flow ;

  TRUST_ELEMENT DataIntegrityChecked :
  SUBJECT DatabaseRecord_X ,
  PREDICATE (HasValidChecksum (DatabaseRecord_X)) ,
  OBJECT ChecksumVerificationSystem ;

5.3. Part C: Meta-Level Control & Evolution (6 Primitives)

5.3.1. GOVERNANCE_RULE

- **Purpose:** Defines an atomic normative statement (obligation, permission, prohibition) applicable within a specific scope. EnforcementContext dictates how the rule is applied.
- **Formal EBNF Syntax:** GOVERNANCE_RULE <RuleID> ':' 'SCOPE' <ScopeID> ',' 'PREDICATE' <BooleanExpression> ',' 'ENFORCEMENT_CONTEXT' <ContextID> ['PRIORITY' <NumericExpression>] (*Updated to NumericExpression*)
    - Refer to <RuleID>, <ScopeID>, <BooleanExpression>, <ContextID>, <NumericExpression> rules in Section 2.2.
- **Formal Semantics:** Specifies a named rule that dictates normative behavior within a SCOPE. If PREDICATE is TRUE, the rule applies. ENFORCEMENT_CONTEXT defines *how* violations are handled (e.g., LogOnly, PreventAction, TriggerAlert). PRIORITY defines its precedence among other rules.
- **Concrete Example:**

```
GOVERNANCE_RULE NoDataDeletionWithoutConsent :
  SCOPE UserManagement ,
  PREDICATE ( AttemptToDeleteUserData ( user_id ) AND NOT ConsentIsGiven ( user_id ) ) ,
  ENFORCEMENT_CONTEXT PreventAction ,
  PRIORITY 99 ;

GOVERNANCE_RULE LogHighRiskOperation :
  SCOPE SystemSecurity ,
  PREDICATE ( IsHighRisk ( operation_type ) ) ,
  ENFORCEMENT_CONTEXT LogOnly ;
```

5.3.2. SELF_REFERENCE_POINT

- **Purpose:** Defines a formal, addressable point within the system's own definition (e.g., its grammar, schemas, rule sets) or runtime state for introspection or modification. TargetTypeID is extensible via META_DEFINITION_RULE.
- **Formal EBNF Syntax:** SELF_REFERENCE_POINT <PointID> ':' 'TARGET_TYPE' <TargetTypeID> ',' 'ACCESS_PROTOCOL' <ProtocolID>
    - Refer to <PointID>, <TargetTypeID>, <ProtocolID> rules in Section 2.2.
- **Formal Semantics:** Creates a named reference (PointID) to a part of the system's own meta-level. TARGET_TYPE specifies what kind of meta-level element is referenced (e.g., GrammarRule, DataSchemaDefinition, GovernanceRuleSet). ACCESS_PROTOCOL defines how to interact with this point (e.g., ReadDefinition, ModifyDefinition, QueryRuntimeState). Essential for meta-cognition.
- **Concrete Example:**

```
SELF_REFERENCE_POINT MainGrammarDefinition :
  TARGET_TYPE GrammarRule ,
  ACCESS_PROTOCOL ReadDefinition ; (* For introspection *)

SELF_REFERENCE_POINT UserPreferenceSchemaRef :
  TARGET_TYPE DataSchemaDefinition ,
  ACCESS_PROTOCOL ModifyDefinition ; (* For evolving user preferences *)
```

5.3.3. MUTATION_RULE

- **Purpose:** Defines an atomic rule for the system to formally modify itself. TargetReference points to a SELF_REFERENCE_POINT. ApprovalPolicy specifies governance for the change.
- **Formal EBNF Syntax:** MUTATION_RULE <RuleID> ':' 'TARGET_REFERENCE' <SelfReferencePointID> ',' 'CONDITION' <BooleanExpression> ',' 'TRANSFORM_ACTION' <ActionID> ['APPROVAL_POLICY' <PolicyID>]
  - Refer to <RuleID>, <SelfReferencePointID>, <BooleanExpression>, <ActionID>, <PolicyID> rules in Section 2.2.
- **Formal Semantics:** Specifies a named rule for self-modification. If CONDITION is TRUE, the TRANSFORM_ACTION (an ActionID referring to an inherent operation that performs the modification) is applied to the element referenced by TARGET_REFERENCE. APPROVAL_POLICY governs whether this mutation requires external consent or passes internal checks before execution.
- **Concrete Example:**

```
MUTATION_RULE AdaptiveResourceThreshold :
  TARGET_REFERENCE MaxMemoryUsage , (* Points to a SELF_REFERENCE_POINT *)
  CONDITION ( LowSystemMemory ( current_memory_usage ) ) ,
  TRANSFORM_ACTION AdjustThresholdByFactor , (* Assumed action taking args from context *)
  APPROVAL_POLICY AutomatedInternalApproval ;
```

5.3.4. PERCEPTION_MAP

- **Purpose:** Defines the atomic transformation from raw input received via an ENVIRONMENT_INTERFACE_POINT into structured internal concepts. TransformationFunction refers to an inherent language function.
- **Formal EBNF Syntax:** PERCEPTION_MAP <MapID> ':' 'INPUT_INTERFACE' <InterfaceID> ',' 'OUTPUT_SCHEMA' <SchemaID> ',' 'TRANSFORMATION_FUNCTION' <ActionID> ['UNCERTAINTY_MODEL' <ModelID>]
  - Refer to <MapID>, <InterfaceID>, <SchemaID>, <ActionID>, <ModelID> rules in Section 2.2.

- **Formal Semantics:** Defines a named process for interpreting raw environmental data. It takes data from INPUT_INTERFACE, applies a TRANSFORMATION_FUNCTION (an ActionID referring to an inherent operation like parsing, LLM inference, or sensor interpretation), and maps it to an OUTPUT_SCHEMA for internal representation. An UNCERTAINTY_MODEL can be specified for probabilistic or noisy interactions.
- **Concrete Example:**
  PERCEPTION_MAP ParseRawSensorData :
    INPUT_INTERFACE TemperatureSensorInput ,
    OUTPUT_SCHEMA TemperatureDataSchema ,
    TRANSFORMATION_FUNCTION ParseCSVDataStream , (* Inherent parsing action *)
    UNCERTAINTY_MODEL SensorErrorModel ;

  PERCEPTION_MAP InterpretLLMResponse :
    INPUT_INTERFACE LLMResponseAPI ,
    OUTPUT_SCHEMA StructuredThoughtSchema ,
    TRANSFORMATION_FUNCTION ExtractTaggedContent ; (* Inherent LLM parsing action *)

5.3.5. LEARNING_AXIOM

- **Purpose:** Defines the atomic contract for a learning process: what it consumes, what it produces, what it optimizes, its resource limits, and how it updates internal knowledge. KnowledgeUpdateRule refers to a STATE_TRANSITION or MUTATION_RULE.
- **Formal EBNF Syntax:** LEARNING_AXIOM <AxiomID> ':' 'INPUT_SCHEMA' <SchemaID> ',' 'OUTPUT_SCHEMA' <SchemaID> ',' 'OBJECTIVE_METRIC' <MetricID> ',' 'CONSTRAINT_SET' '{' <ResourceBoundID> { ',' <ResourceBoundID> } '}' ',' 'KNOWLEDGE_UPDATE_RULE' <RuleID> (*Updated to show literal braces for list syntax*)
  - Refer to <AxiomID>, <SchemaID>, <MetricID>, <ResourceBoundID>, <RuleID> rules in Section 2.2.
- **Formal Semantics:** Defines a named learning process. It takes input conforming to INPUT_SCHEMA, produces output conforming to OUTPUT_SCHEMA, optimizes for OBJECTIVE_METRIC, operates within CONSTRAINT_SET (e.g., resource usage), and updates internal knowledge using a KnowledgeUpdateRule (which must be a STATE_TRANSITION or MUTATION_RULE rule).
- **Concrete Example:**
  LEARNING_AXIOM AdaptUserTonePreference :
    INPUT_SCHEMA UserFeedbackSchema ,
    OUTPUT_SCHEMA UpdatedToneModelSchema ,
    OBJECTIVE_METRIC ToneAlignmentScore ,
    CONSTRAINT_SET { CPU_Constraint , Memory_Constraint } ,
    KNOWLEDGE_UPDATE_RULE UpdateInternalToneModelRule ; (* Refers to a MUTATION_RULE or STATE_TRANSITION rule *)

5.3.6. META_DEFINITION_RULE

- **Purpose:** Defines an atomic rule for extending the Omega-Code's own meta-level vocabulary (e.g., defining new ModalityTypes, ResourceTypeIDs, InteractionTypeIDs). DefinitionSchema provides the formal structure for the new definition.
- **Formal EBNF Syntax:** META_DEFINITION_RULE <RuleID> ':' 'TARGET_TYPE' <TargetTypeID> ',' 'DEFINITION_SCHEMA' <SchemaDefinition> ',' 'VALIDATION_PROTOCOL' <ProtocolID>
    - Refer to <RuleID>, <TargetTypeID>, <SchemaDefinition>, <ProtocolID> rules in Section 2.2.
- **Formal Semantics:** Specifies a named rule for extending Omega-Code's built-in type system or conceptual ontology. It defines a new TARGET_TYPE (e.g., NewSensorType, CommunicationPattern). DEFINITION_SCHEMA provides the structure for how instances of this new type are formally defined. VALIDATION_PROTOCOL ensures that new definitions conform to required standards (e.g., SchemaValidation, FormalLogicCheck).
- **Concrete Example:**

```
META_DEFINITION_RULE DefineNewSensorType :
  TARGET_TYPE SensorTypeID ,
  DEFINITION_SCHEMA (
    VAR name : STRING ;
    VAR measurementUnit : STRING ;
    VAR accuracyTolerance : FLOAT ;
  ) ,
  VALIDATION_PROTOCOL SchemaValidation ;

(* Example of using the newly defined SensorTypeID: *)
VAR my_new_sensor : SensorTypeID AS OpticalSensor;
```