AZRA Cipher Syntax

Imvoke(:”system/64/user….”)

World-type(:Azra , [ip = 1234])

name int, float

-- } space for main code

Extract : 123456 → int, float

Extract(:Hello World) → *for more than 50 character use ”str ”

str

Delog{int, float, str…} → This is used to import the variables we need into the code; otherwise, we can remove this line.

Submit(:1.azr)

AZRA Programming Language

Variable Definition & Syntax Rules

Version 1.0

Document Type: Official Language Specification

---

1. General Structure of Variable Definitions

In AZRA, every variable must follow a strict and recognizable pattern.

This structure allows the system to correctly detect and interpret variable definitions.

1.1 General Format

-<VariableNumber><VariableName> = <Value>-

The leading and trailing dash characters (-) act as structural boundaries.

If either dash is missing, the system cannot detect the definition and a Syntax Error will occur.

---

2. Basic Rules

Rule 1 — Mandatory Leading Dash

Every variable definition must begin with a dash (-).

Correct:

-0x = 2-

Incorrect:

0x = 2

Rule 2 — Mandatory Variable Numbering

Each variable must have a numerical index placed directly before the variable name.

- **Indexing starts at 0.**
- **Each new variable must increment sequentially.**
- **No gaps are allowed.**

Example (third variable):

-2msg = Hello-

Note:

Floating-point values do not affect numbering rules.

Rule 3 — No Extra Numbers Before the Name

The only number allowed before the variable name is the variable index.

Incorrect:

-42A = Hello-

Rule 4 — Numbers Inside Variable Names

Numbers may appear inside the variable name only if they come after a letter.

Correct:

-1Am1r = 10-

-2x2 = 5-

Rule 5 — Mandatory Trailing Dash

Every variable definition must end with a dash (-).

Correct:

-3b = Hello World3-

Incorrect:

-3b = Hello World3

Missing the closing dash results in an Open Variable Syntax Error.

---

3. String Rules

Rule 6 — Strings Shorter Than 50 Characters

Strings with fewer than 50 characters may be written without quotation marks.

Examples:

-1y = Hello World-

-2z = Hi-

Rule 7 — Strings With 50 or More Characters

Strings with 50 or more characters must be enclosed in double quotation marks (").

Example:

-3longMsg = “This is a long text containing more than fifty characters...”-

---

Rule 8 — Single Quotes Are Not Allowed

Single quotes (' ') are not recognized by AZRA and will cause a Syntax Error.

Incorrect:

-1x = ‘Hello’-

Rule 9 — Numbers Within Strings

If a string begins or ends with a number-letter combination, it is still treated as a string.

Valid Examples:

-1y = 1Hello World-

-2z = Hello 1 World-

-3b = Hello World3-

---

4. Value Assignment Rules

Rule 10 — Automatic Type Detection

The system determines value types using the following logic:

- **Value without quotes and without letters → Numeric**
- **Value inside double quotes → String**
- **Value containing both letters and numbers → String**
- **Value containing @ or ref: → Symbol / Reference**

Rule 11 — Reference / Symbol Rules

References such as @2 are not standalone values.

They must be wrapped inside parentheses when used in supported commands (e.g., Extract).

Correct:

Extract(:@2)

Incorrect:

Extract : @2

Rule 12 — Multi-Word Strings

If a string contains multiple words, it is recommended to use double quotes for clarity, even if under 50 characters.

Recommended:

-0name = “Amir Hosseini”-

---

5. Extended Rules

Rule 13 — Spacing Around Equals Sign

Spacing around = is flexible and stylistic.

All are valid:

-0x = 2-

-0x = 2 -

-0x=2-

Rule 14 — One Definition Per Line

Each variable definition must be written on a single line.

Splitting across lines or placing multiple definitions on one line is prohibited.

Rule 15 — No Skipping Numbers

Variable numbering must be sequential.

Incorrect:

-0x = 5-

-2y = 10-

(Index 1 is missing.)

Rule 16 — No Negative Index Numbers

Variable indices must be non-negative integers.

Incorrect:

–1x = 5-

Rule 17 — Variable Name Cannot Start With a Number

After the variable index, the variable name must begin with a letter.

Incorrect:

-0 1name = 2-

-3 12x = ok-

Rule 18 — Allowed Characters in Variable Names

Variable names may contain:

- **Uppercase letters (A–Z)**

• Lowercase letters (a–z)

• Digits (only after a letter)

• Underscore _

No other characters are permitted.

Rule 19 — String Values Starting With a Number

If a string value begins with a number, it must either:

• Be enclosed in double quotes

OR

• Have a letter immediately after the number

Correct:

-0a = “1Hello”-

-1code = 1xTest-

Incorrect:

-0a = 1-

Rule 20 — Symbols Cannot Be Assigned Directly

Symbols such as @2 cannot be directly assigned to a variable.

Incorrect:

-0ref = @2-

Symbols must be accessed through supported structures:

Extract(:@n)

Conceptual Model & Core Architecture

File Metadata

Target System

The Target System refers to the system on which the code will be executed.

This system has two possible forms:

1. **The system where the code is written.**
2. **A dedicated device specifically built to execute the AZRA language.**

Filename

The file containing the code must be named using numbers only.

Examples:

1.azr

561.azr

No alphabetic characters are allowed in the filename.

Dependencies

An AZRA file can only connect to other files written in the same language (AZR).

If the file needs to connect to external code or a non-AZR file,

that file must first be converted (compiled) into AZR format before use.

Therefore, AZRA files only communicate with files of the same type.

Pre-Execution Metadata (Before Code Starts)

Before the main body of code begins, certain metadata must be defined.

This includes:

- **The invoked system**
- **File type (default: azr)**
- **System ID for execution authorization**
- **The symbol “–” to mark the beginning of the main code section**

Imvoke

At the beginning of every file, the `Imvoke` instruction must appear.

This command is responsible for:

- **Calling the execution system**
- **Defining the system version**
- **Defining access level**
- **Determining which environment is authorized to execute the file**

Example:

Imvoke(:“system/64/user.001.alpha”)

This line informs the system:

- **Which environment should execute the file**
- **Which version is required**
- **What access level is permitted**

If a system other than the specified one attempts to execute the file, execution will be denied.

Without the `Imvoke` instruction, the file is not authorized to run and will be rejected.

World-type

The `World-type` instruction defines the execution environment and world configuration.

Example:

World-type(:Azra | [ip = 1234])

This line specifies that the code runs in the Azra world

with defined environmental parameters (for example, IP address or network identifier).

Important Syntax Note

In AZRA, variable names must never contain spaces.

Incorrect example:

-0amir reza = 2-

This produces a Syntax Error

because there is a space between `amir` and `reza`.

Correct Method for Multi-Word Variable Names

To separate words inside a variable name, the following character must be used:

• Underscore _

Correct version:

-0amir_reza = 2-

This version is also valid:

-0amir-reza = 2-

Note:

In World-type, the underscore rule does not apply because World-type is a core system instruction and is not governed by the standard AZRA variable grammar.

Post-Execution Metadata (End of File Section)

At the end of the file, specific instructions must be added to inform the execution system:

• What data should be processed

• What data should be ignored

This section usually consists of two final commands:

Delog{ ... }

Submit

Delog

General Format:

Delog{int, str, ...}

This instruction specifies which data types from the file's variables

must be recognized during compilation.

Functionality:

• Defines which variable types (int, str, bool, ref, etc.) the program should process

• Instructs the compiler to ignore variables of other types

• Ensures that only the types listed inside {} are included in final processing

If Delog{...} is not present:

All variables in the file are considered valid and required.

The compiler must process every variable.

Example:

Delog{int, str}

This means:

Only numeric and string variables are relevant.

Other types (e.g., bool or ref) will be ignored.

Submit

General Format:

Submit(:<filename>)

Example:

Submit(:1.azr)

This instruction must appear at the end of the file.

Functionality:

• Officially marks the end of the code

- **Confirms successful execution**
- **Registers the file name for reporting and dependency tracking**
- **No executable instructions may appear after `Submit`**

Standard Ending Structure

In a standard AZRA file, the ending typically appears as:

Delog{int, str, ref}

Submit(:123.azr)

Execution flow:

1. **`Delog` determines which data types are processed and which are ignored.**
2. **`Submit` finalizes and officially closes the file execution.**

Final Note

`Delog` and `Submit` are complementary and form the official termination block of an AZR file.

If either of them is missing,

the system will consider the file incomplete or invalid.

AZRA – Dash Block Separator (DBS) Rule Specification

Official Language Rule Document

Introduction

The DBS (Dash Block Separator) rule is one of the core structural principles of the AZRA programming language.

Its main purpose is to enforce organization, enhance readability, and allow structural verification of code blocks during compilation.

Proper use of DBS enables the compiler to precisely identify where each code block starts and ends.

Core Rule

After every multi-line block, there must be a separator line consisting *only* of dash (-) characters.

The number of dashes in that line must be exactly equal to the number of lines contained in the preceding block.

This line acts as a "block signature", confirming the structural integrity and completeness of that code section.

Block Definition

A block is defined as a sequence of consecutive lines that together form a single logical unit within the program.

Common examples of blocks in AZRA include:

• Multiple variable definitions written consecutively

• Several Extract instructions grouped together

• A series of related operations

• Any segment of code perceived by the programmer as one cohesive unit

After every block, inserting a DBS line is mandatory.

Omission of the DBS line will result in a compilation error.

Examples

Example 1: Two-Line Block

-0x = Hello World-

Extract(:Hello World)

Explanation:

Because the block contains 2 lines, the separator line contains 2 dashes (--).

Example 2: Three-Line Block

-1y = 1Hello World-

-2z = Hello 1 World-

-3b = Hello World3-

Explanation:

The block contains 3 lines, thus the separator consists of 3 dashes (---).

Importance of DBS

The DBS is not a mere visual separator—

it plays several critical structural roles within the AZRA language:

1. **File Structure Integrity**

   If even one line within a block is moved, inserted, or removed,

   the compiler will instantly detect the inconsistency by comparing the number of dashes.

2. **Block Order Validation**

   Since block length and separator length are interlocked,

   it becomes practically impossible to forge, manipulate, or partially define a block without detection.

DBS Error Types

The compiler is required to detect and report the following DBS-related errors:

1. DBS-Mismatch

The number of dashes in the separator does not match the number of lines in its preceding block.

---

2. DBS-Missing

The block has ended, but no DBS line has been placed.

---

3. DBS-Invalid

The separator line contains characters other than dashes (-).

Summary

The Dash Block Separator (DBS) rule ensures:

• Structural stability of AZRA code

• Automatic detection of editing or copying errors

• Verification of block completeness during compilation

By enforcing DBS at the end of every multi-line block,

AZRA maintains deterministic structure validation and precise compilation integrity.

AZRA — Specification: Class, Method, Function

Official Structural & Behavioral Definition

1. Class Definition in AZRA

A Class in AZRA is used to define a custom data type.

A class contains:

• The class name

• A list of properties

• Optional method bodies

General Structure

Class(:ClassName | [prop1:type , prop2:type , ...])

• The first parenthesis holds the class name

• The bracket block contains the property list

After defining a class, a DBS line must be placed, containing exactly 1 dash, because the class definition is considered a single structural line.

Example

Class(:Person | [name:str , age:int])

This structure creates a `Person` class with two properties: `name` and `age`.

---

2. Method Definition in AZRA

A Method is a function associated with a class and describes the behavior of an object.

General Structure

Method(:methodName) ->returnType

-[variable definitions and operations]-

turn(:value)

Rules

- **A method must always begin with the keyword** `Method`
- `returnType` **defines the method's output type**
- **Internal variables must follow AZRA Variable Definition rules**
- **Output must be produced using** `turn`
- **The method must end with a DBS line,**

where the number of dashes equals the number of internal lines

Example

Method(:greet) ->str -0msg = "Hello" + name-turn(:msg)

(2 internal lines → --)

---

3. Function Definition in AZRA

A Function (Func) is an independent piece of code located outside of classes, often used for higher-level logic or main program flow.

General Structure

Func(:functionName)

-[variable definitions and operations]-

turn(optional)

Rules

- A function begins with `Func`
- It may contain variables, operations, method calls, and `Extract`
- Output is defined using `turn`; it may be empty
- Ending DBS must contain a number of dashes equal to the number of internal lines

Example

Func(:main) -0p = Person(“Amir” , 20)- -1result = p:greet()- Extract(:@1) turn()

(4 internal lines → ----)

---

4. Method Invocation

Methods are invoked using the : operator.

General Form

object:method()

Example

p:greet()

This executes the `greet` method on object `p`.

---

5. Object Instantiation

Objects are created using the class name followed by arguments.

General Structure

variable = ClassName(arg1 , arg2 , ...)

Example

-0p = Person(“Amir” , 20)-

---

6. turn Instruction

The turn instruction defines the output of a method or function.

Forms

turn(:value)

turn() // no output

---

7. Role of DBS in Classes, Methods, and Functions

All Class, Method, and Function definitions must end with a DBS line.

The count of dashes must exactly match the number of internal logical lines of the block.

This ensures:

- **Structural validation**
- **Detection of missing, extra, or reordered lines**
- **Block integrity at compile-time**

Conditional Structure Specification (is / reply / shoot / wall)

Overview

In AZRA, logical decision-making is done through a four-part conditional chain consisting of the following reserved keywords:

1. **is**
2. **reply**
3. **shoot**
4. **wall()**

This order must never change.

Any modification in ordering results in a syntax error.

---

1. is — Primary Condition

The conditional structure always begins with `is`.

- **Defines the main logical condition**
- **If the condition evaluates to true, its block executes**
- **After execution, the entire chain terminates**
- **If false, execution moves to the next stage (`reply`)**

---

2. reply — Additional Conditional Branch(es)

- **Zero or more `reply` sections may follow `is`**
- **Each `reply` has its own independent condition**
- **A `reply` is evaluated only if all prior conditions failed**
- **If true, its block executes and the chain ends**
- **If false, the next `reply` is tested**

---

3. shoot — Default Path

If none of the previous conditions (is or any reply) evaluate to true:

• Execution enters the shoot block

• shoot acts as the final fallback path

• It always appears after all replies

---

4. wall() — Conditional Chain Terminator

The conditional chain must end with:

wall()

This instruction:

• Marks the official end of the conditional block

• Allows the compiler to close the structure correctly

If wall() is missing, a syntax error occurs because the compiler cannot determine where the decision structure ends.

Execution Flow Summary

1. **Evaluate is• If true → execute → end**
2. **Else, evaluate each reply in order• First true reply → execute → end**
3. **Else, execute shoot**
4. **Execute wall() to officially close the structure**
5. **Finally, a DBS line must follow**

DBS Requirement

After the conditional chain ends (right after `wall()`):

- **A DBS line must appear**
- **The number of dashes must match the total logical lines inside the conditional block**
- **A mismatch results in a DBS error**

Formal Definition of the Conditional Structure

Conditional Structure Specification — AZRA Language

---

1. Structural Components of the Conditional Chain

1.1 Core Elements

The conditional structure in AZRA consists of four components:

“is” , “reply” , “shoot” , “wall()”

These four elements together form the complete logical decision chain.

1.2 The “is” Section

1.2.1

The keyword “is” is the first element of the conditional chain.

1.2.2

The primary condition of the structure is defined within this section.

1.2.3

If the condition inside “is” evaluates to true, its corresponding block is executed.

1.2.4

After execution of the “is” block, the conditional chain terminates immediately.

1.3 The “reply” Section

1.3.1

The keyword “reply” is used to define additional conditions following “is”.

1.3.2

Each “reply” is evaluated only if all preceding conditions have evaluated to false.

1.3.3

Multiple “reply” sections are allowed within a single conditional chain.

1.3.4

“reply” sections are evaluated strictly in the order they are written.

1.3.5

The first “reply” whose condition evaluates to true will execute its block, and the chain terminates at that point.

1.4 The “shoot” Section

1.4.1

The keyword “shoot” defines the fallback (default) path of the conditional chain.

1.4.2

“shoot” executes only if none of the conditions in “is” or any “reply” evaluate to true.

1.4.3

The definition of “shoot” is optional.

1.4.4

If “shoot” is not defined and no condition evaluates to true, the chain terminates without executing any block.

1.4.5

“shoot” has two syntactical forms:

• Full form:

shoot() ->

• Short form:

shoot ->

1.5 The “wall()” Section

1.5.1

The keyword “wall()” marks the official termination of the conditional chain.

1.5.2

The presence of “wall()” at the end of the conditional structure is mandatory.

1.5.3

Omission of “wall()” results in a syntax error.

---

2. Element Ordering Within the Conditional Structure

2.1 Valid and Correct Order

“is” → “reply” (zero or more times) → “shoot” (optional) → “wall()”

2.2 Ordering Rules

2.2.1

The defined order must be strictly followed.

2.2.2

Any reordering, invalid repetition, or incorrect placement of elements results in a syntax error.

2.2.3

Only the first branch whose condition evaluates to true is executed.

All subsequent branches are ignored.