Pseudo-Language: "PseudoScript"

This language is designed to be simple, readable, and focus on logic and structure rather than complex syntax. It borrows from common scripting languages.

Goal: Express algorithms and data manipulation clearly, usable in a technical challenge that can be reasoned about by both humans and AI.

Key Concepts:

Variables:

Declared with `LET`. Dynamically typed for simplicity.

```
LET counter = 0
LET message = "Processing data"
LET is_valid = TRUE
```

Data Types:

`NUMBER`, `STRING`, `BOOLEAN` (`TRUE`/`FALSE`), `LIST` (ordered collection), `MAP` (key-value pairs).

```
LET names = ["Alice", "Bob", "Charlie"]
LET config = { "threshold": 0.75, "retries": 3 }
```

Sometimes challenges require custom models. You could allow defining lightweight

STRUCT types:

```
STRUCT User
  name: STRING
  age: NUMBER
  active: BOOLEAN
ENDSTRUCT

LET new_user = User { name: "Alice", age: 30, active: TRUE }
```

Comments: Use

```
# This is a comment
```

Operators

Basic arithmetic (+, -, *, /)
Comparison (==, !=, >, <, >=, <=)
Logical (AND, OR, NOT).

Control Flow

```
IF condition THEN
            # code block
      ELSE IF condition THEN
            # code block
      ELSE
            # code block
ENDIF
```

SWITCH/CASE (for readable multi-condition branching)

```
SWITCH value
  CASE 1
    PRINT("One")
  CASE 2
    PRINT("Two")
  DEFAULT
    PRINT("Other")
ENDSWITCH
```

```
LOOP WHILE condition
      # code block
      # Use BREAK to exit loop,
      CONTINUE to skip to next iteration
ENDLOOP
```

```
LOOP FOR item IN collection
      # code block using 'item'
ENDLOOP
```

```
LOOP FOR INDEX i IN 0 TO LENGTH(collection) - 1
      LET item = collection[i]
      # Do something with item
ENDLOOP
```

```
LOOP UNTIL finished # Opposite of WHILE
      # Do stuff
ENDLOOP
```

Functions: Definition and calling.

```
FUNCTION calculate_average(numbers_list)
      LET sum = 0
      LET count = 0
      LOOP FOR num IN numbers_list
            sum = sum + num
            count = count + 1
      ENDLOOP

      IF count == 0 THEN
            RETURN 0
      ENDIF

      RETURN sum / count
ENDFUNCTION

# Calling the function:
LET scores = [85, 90, 77]
LET avg_score = CALL calculate_average(scores)
```

MODULES (optional: grouping functions and variables)

```
MODULE MathUtils
  FUNCTION add(a, b)
    RETURN a + b
  ENDFUNCTION
ENDMODULE
```

```
IMPORT MODULE (conceptual, even if pre-loaded)
```

Input/Output (Conceptual)

Assume functions exist for basic I/O if needed for a challenge.

`READ("input_file.txt")` -> returns content (e.g., as a LIST of strings)
`WRITE(data, "output_file.txt")`
`PRINT(value)` -> outputs to console/standard output.

Array (LIST) Operations

Useful built-in LIST operations:

- `.APPEND(value)`
- `.INSERT(index, value)`
- `.REMOVE(value)`
- `.REMOVE_AT(index)`
- `.SORT(order = "ASC"|"DESC")`
- `.REVERSE()`
- `.FIND(value)` → returns index or -1
- `.FILTER(function)` → returns filtered list

Text (STRING) Manipulation

Common STRING operations:

- `.UPPER()`
- `.LOWER()`
- `.TRIM()`
- `.CONTAINS(substring)`

- `.STARTS_WITH(prefix)`
- `.ENDS_WITH(suffix)`
- `.SPLIT(delimiter)` → returns LIST
- `.JOIN(list, delimiter)` → returns STRING
- `.REPLACE(old, new)`

Date Operations

Minimal DATE support:

```
LET today = DATE_NOW()
LET parsed_date = DATE_PARSE("2025-04-29")
LET formatted = DATE_FORMAT(today, "YYYY-MM-DD")
LET future = DATE_ADD(today, "DAYS", 5)
LET past = DATE_SUB(today, "MONTHS", 2)
LET diff = DATE_DIFF(today, parsed_date, "DAYS")
```

Functions:

- `DATE_NOW()`
- `DATE_PARSE(string)`
- `DATE_FORMAT(date, format)`
- `DATE_ADD(date, unit, value)`
- `DATE_SUB(date, unit, value)`
- `DATE_DIFF(date1, date2, unit)`

Error Handling Functions

Simple try/catch logic:

```
TRY
  # risky operations
CATCH error
  PRINT("An error occurred: " + error.MESSAGE)
ENDTRY
```

Error object standard fields:

- `error.MESSAGE`
- `error.TYPE`

- `error.CODE`

You can also raise errors manually:

```
RAISE_ERROR("Something went wrong", "CustomError", 400)
```

Network Operations

Basic HTTP functions:

```
LET response = HTTP_GET(url)
LET response = HTTP_POST(url, body)
LET response = HTTP_PUT(url, body)
LET response = HTTP_DELETE(url)
```

```
LET data = { "username": "admin", "password": "1234" }
LET result = HTTP_POST("https://api.example.com/login", data)
PRINT(result["token"])
```

Optional headers could be added later if needed (`HTTP_GET(url, headers)`).

System Operations

Simple built-in system functions:

- `ENV_GET(key)` → get environment variable
- `SLEEP(milliseconds)` → wait/pause
- `EXIT(code)` → exit program
- `LOG(message)` → write to system log

```
CALL LOG("Process started")
LET env = ENV_GET("ENVIRONMENT")
PRINT("Running in " + env)
```

Strong Typing (Optional Annotations)

If needed (e.g., for strict challenges), allow optional type hints:

```
LET counter: NUMBER = 0
LET name: STRING = "Alice"
LET active: BOOLEAN = TRUE
LET users: LIST = ["User1", "User2"]
LET config: MAP = { "mode": "safe", "timeout": 30 }
LET point: TUPLE = (10, 20)
```

Functions can also define parameter types:

```
FUNCTION add_numbers(a: NUMBER, b: NUMBER): NUMBER
  RETURN a + b
ENDFUNCTION
```

More Built-in Utilities

Examples:

- `UUID_GENERATE()`
- `RANDOM(min, max)`
- `HASH(value, algorithm = "SHA256")`
- `ENCODE_BASE64(data)`
- `DECODE_BASE64(data)`

OOP in PseudoScript

1. Define Classes

```
CLASS ClassName
  FUNCTION __init__(self, param1, param2)
    self.field1 = param1
    self.field2 = param2
  ENDFUNCTION

  FUNCTION methodName(self, additionalParam)
    # Access fields using self.field
    RETURN self.field1 + additionalParam
  ENDFUNCTION
ENDCLASS
```

✅ Notes:

- `self` is **always the first argument** inside class methods (to refer to the instance).
- Fields are attached to `self`.
- `__init__` is the constructor (like Python).

---

2. Create and Use Objects

```
LET obj = NEW ClassName(value1, value2)
LET result = CALL obj.methodName(5)
```

✅ Simple: `NEW` keyword + `CALL` on methods.

---

3. Class Inheritance

To allow basic inheritance:

```
CLASS ChildClass EXTENDS ParentClass
  FUNCTION __init__(self, param1, param2, param3)
    CALL super().__init__(param1, param2) # Call parent constructor
    self.field3 = param3
  ENDFUNCTION

  FUNCTION newMethod(self)
    RETURN self.field1 + self.field3
  ENDFUNCTION
ENDCLASS
```

✅ Only *single inheritance* (no multiple inheritance), to keep it simple.

✅ `super()` is used to call the parent.

---

4. Example of a Full OOP Snippet

```
# Define a class
CLASS Animal
  FUNCTION __init__(self, name)
    self.name = name
  ENDFUNCTION

  FUNCTION speak(self)
    RETURN self.name + " makes a sound."
  ENDFUNCTION
ENDCLASS

# Define a subclass
CLASS Dog EXTENDS Animal
  FUNCTION speak(self)
    RETURN self.name + " barks!"
  ENDFUNCTION
ENDCLASS

# --- Main part ---

LET animal = NEW Animal("GenericAnimal")
LET dog = NEW Dog("Buddy")

PRINT(CALL animal.speak()) # "GenericAnimal makes a sound."
PRINT(CALL dog.speak())    # "Buddy barks!"
```

PseudoScript Coding Best Practices

1. Code Readability First

- **Use clear names** for variables, functions, classes, and modules.

  ```
  LET total_score = 0   # GOOD
  LET ts = 0            # BAD
  ```

- **Use comments (#)** to explain why something is done, especially complex logic.

- **Indent consistently** inside control structures, loops, and functions.

2. Structure Your Code

- Group related functions into modules whenever possible.
- One purpose per function: Functions should be small and focused on a single task.
- Separate logic into classes if modeling real-world objects or systems.

3. Naming Conventions

- Variables and Functions: Use snake_case

```
LET max_value = 100
FUNCTION calculate_average(numbers_list)
```

- Classes: Use PascalCase

```
CLASS PaymentProcessor
```

- Constants (if you define them): Use UPPER_SNAKE_CASE

```
LET MAX_RETRIES = 5
```

4. Error Handling

- Always handle potential errors using TRY/CATCH where operations can fail (e.g., network, file operations).
- Print or log clear error messages, don't just silently fail.

```
TRY
  LET data = READ("config.txt")
CATCH error
  PRINT("Failed to load config: " + error.MESSAGE)
ENDTRY
```

5. Data and Collections

- Prefer LIST methods (.FILTER(), .SORT(), .APPEND()) over manual loops when simple.
- Avoid modifying LISTs while looping over them unless absolutely necessary.
- Use STRUCTs when you need clear, structured data instead of free MAPs.

6. Control Flow

- Keep IF/ELSE blocks short.
- Prefer SWITCH when checking many discrete values.
- Use BREAK/CONTINUE carefully: they should improve flow, not confuse it.

Example:

```
LOOP FOR item IN items
  IF item == "skip" THEN
    CONTINUE
  ENDIF
  IF item == "stop" THEN
    BREAK
  ENDIF
  PRINT(item)
ENDLOOP
```

7. Functions and Methods

- Limit parameters: Prefer <= 4 parameters per function; use MAPs or STRUCTs for many related values.

- Always RETURN a value explicitly if the function is meant to produce output.

- Document your functions with a comment header if complex.

```
# Calculates the average of a list of numbers.
FUNCTION calculate_average(numbers_list)
  ...
ENDFUNCTION
```

8. OOP Best Practices

- Use classes when modeling entities (e.g., Users, Products, Orders).

- Use inheritance only when there's a clear "is-a" relationship (e.g., Dog "is a" Animal).

- Encapsulate related behaviors inside the class. Avoid exposing fields unless necessary.

- Prefer composition over inheritance if complex behavior needs combining.

9. System and External Operations

- Validate input data when reading from files, networks, or user inputs.

- Log important system events (CALL LOG("User logged in")), especially failures.

- Sleep responsibly: Avoid unnecessary SLEEP unless really needed (e.g., retries).

10. Write Clean and Testable Code

- Write modular, testable functions: Pure functions (without side effects) are easier to reason about.

- Prefer small examples or simple scenarios for testing during development.

- Think about edge cases (empty LISTs, NULL values, bad inputs).