VERTIGO PROGRAMMING LANGUAGE: A COMPREHENSIVE REFERENCE MANUAL

INTRODUCTION TO VERTIGO LANGUAGE

Vertigo is a stack-based programming language designed with an emphasis on simplicity, explicit control flow, and flexibility in low-level and procedural programming tasks. Its design is influenced by the Lua 5.4 language manual style, combining readable syntax with powerful stack and register operations to enable versatile execution models.

PURPOSE AND DESIGN PHILOSOPHY

The Vertigo language was created to provide a flexible tool for developers working within constrained environments or requiring fine control of procedural execution with stack-centric semantics. It employs a minimalistic instruction set, allowing direct manipulation of multiple named stacks and registers while supporting subroutine modularity and interaction. Vertigo's design philosophy prioritizes clarity, extensibility, and predictable behavior, making it suitable for developers familiar with stack-based or low-level programming languages.

FILE FORMATS AND COMMENT SYNTAX

Vertigo source files use the `.vtgo` extension, clearly identifying them as programs written in the Vertigo language. During execution or debugging, Vertigo can produce dump files with the `.vtd` extension, which contain serialized runtime state or logs useful for analysis and diagnostics.

Comments in Vertigo begin with a semicolon (`;`) and continue to the end of the line. This allows straightforward annotation of source code without interfering with execution.

CORE LANGUAGE COMPONENTS

Understanding Vertigo requires familiarity with its primary components, which provide the structure and control mechanisms of the language:

- **Stacks:** Vertigo supports multiple, named stacks which hold runtime data. These stacks allow last-in-first-out (LIFO) operations such as `PUSH`, `POP`, and `DUP`. Stacks provide the foundational data storage mechanism, enabling complex computations by combining stack entries with instructions.
- **Registers:** The language features a small set of registers offering quick access to data and control flags. Key registers include `ODA` (Output Data Area), `IDA` (Input Data Area), `CLI` (Loop Counter), and `LTM` (Loop Termination Marker). Registers allow temporary storage and interaction with stack data, facilitating arithmetic and logical operations.
- **Labels:** Labels are symbolic names defined within the program using the `POINT` instruction. They allow precise control flow changes such as jumps and conditional branching, providing structured navigation through instructions.
- **Subroutines:** To promote modularity and code reuse, Vertigo supports subroutines, which can be defined inline with `SUB` and `ENDSUB` directives or imported from external libraries using `BRING` and `IMPORT`. Subroutines streamline program organization and execution.

GENERAL SYNTAX OVERVIEW

Vertigo programs consist of sequences of instructions, each optionally followed by arguments. Instructions are typically written in uppercase for clarity and consistency, such as:

```
PUSH 42
POP ODA
MATH ADD & ODA 10
JUMP loop_start
; This is a comment
```

This clear and concise syntax enables straightforward reading and writing of Vertigo code, encouraging good documentation and maintainability styles similar to the Lua manual format.

SUMMARY

In essence, Vertigo offers a compact, expressive language focused on stack and register manipulation with robust flow control, subroutine capabilities, and extensible modular libraries. It utilizes `.vtgo` for source code and `.vtd` for dump files, with comments denoted by semicolons. The core abstractions of stacks, registers, labels, and subroutines form the backbone of Vertigo's programming model, providing developers with a flexible and powerful toolset for a variety of procedural programming tasks.

BASIC SYNTAX AND FILE FORMAT

Vertigo source code is organized into text files with the `.vtgo` extension. Each source file consists of individual lines, where each line generally contains a single instruction followed by its arguments if applicable. The language syntax is designed to be line-oriented, making the positioning of newlines and whitespace essential for correct parsing and execution.

COMMENTS

Comments in Vertigo begin with a semicolon (`;`) and extend to the end of the line. Any text following the semicolon is ignored by the interpreter, allowing developers to add annotations, explanations, or disable code segments without affecting the program's behavior. For example:

```
; This is a full line comment
PUSH 10  ; Push the number 10 onto the current stack
```

INSTRUCTION STRUCTURE

Each instruction consists of an opcode mnemonic, optionally followed by one or more arguments. The opcode and its arguments are separated by spaces or tabs. Vertigo treats whitespace as a delimiter, so multiple spaces between tokens are allowed, but tokens must not contain unescaped spaces unless enclosed in double quotes.

String literals must be enclosed in double quotes (`"`), enabling inclusion of spaces and special characters without breaking tokenization. For example:

```
PUSH "Hello, Vertigo!"
CONCAT ODA "Hello, " "World!"
```

Instructions and arguments are case-insensitive; `push`, `PUSH`, and `Push` are treated identically.

LABELS AND THE `POINT` INSTRUCTION

Labels are symbolic identifiers used to mark locations in code for control flow operations such as jumps and branches. A label is declared using the `POINT` instruction followed by a unique name:

```
POINT loop_start
```

Labels serve as destinations for `JUMP` and conditional jump instructions (`JUMPEQ`, `JUMPGT`, etc.). They enable structured flow control by allowing execution to transfer to different parts of the program dynamically.

INSTRUCTION POINTER AND EXECUTION FLOW

Vertigo processes instructions sequentially, starting from the first line and advancing one line at a time (incrementing the instruction pointer). Unless explicitly redirected by jump or call instructions, this linear execution model continues until the program ends or halts.

Jump instructions alter the instruction pointer to the line number corresponding to a declared label. For example:

```
JUMP loop_start
```

Here the instruction pointer will transfer control to the line marked by `POINT loop_start` instead of continuing sequentially.

WHITESPACE AND FORMATTING

While Vertigo allows flexible usage of spaces and tabs for separating tokens, strict adherence to placing one instruction per line is required for clear program structure. Empty lines are permitted and ignored, which can help improve readability.

Whitespace within string literals is preserved exactly as written, enabling strings with spaces and special characters. However, no instructions or labels may span multiple lines; each must be fully contained on a single line.

SUMMARY OF BASIC SYNTAX RULES

- Source files have the extension `.vtgo` and are line-oriented text files.
- Comments start with a semicolon `;` and continue to the end of the line.
- Each line typically contains one instruction mnemonic followed by its arguments, separated by whitespace.
- Instruction names are case-insensitive.
- String arguments must be enclosed in double quotes `" "` to include spaces or special characters.
- Labels are declared using the `POINT` instruction and identify jump targets.
- The instruction pointer advances sequentially unless modified by jump or call instructions.
- Whitespace outside tokens serves only as separators and is otherwise ignored.

STACKS AND STACK OPERATIONS

Vertigo's core data management revolves around multiple named stacks. Unlike many languages that rely solely on variables or registers for data storage, Vertigo uses explicit stack structures identified by unique names. This approach allows programmers to organize data flows modularly and operate on several distinct data sets concurrently. Each stack follows the traditional Last-In, First-Out (LIFO) principle, but Vertigo enhances flexibility by enabling dynamic stack creation, selection, and manipulation via dedicated instructions.

NAMED STACKS AND CURRENT STACK CONTEXT

Throughout program execution, Vertigo maintains a notion of a **current stack**, which is the target for most stack-related operations like pushing and popping values. The current stack is set implicitly by naming the stack on a line by itself and is crucial for determining where stack operations apply.

Without selecting a current stack, any stack operation that requires a stack context will fail and raise a runtime error. This design encourages explicit management of stack usage and prevents unintended manipulation of undefined data areas.

CREATING AND SELECTING STACKS: `NEW` AND STACK NAMING

To create a new named stack, use the `NEW` instruction followed by the desired stack identifier. For example:

```
NEW data_stack
```

This creates an empty stack called `data_stack`. Newly created stacks are empty and ready to store data elements pushed onto them.

To select a current stack, simply write the stack's name on a line by itself. For example:

```
data_stack
```

After this line executes, all subsequent stack operations without explicit stack references affect `data_stack`. Attempting to perform stack operations without a current stack selected results in an error.

BASIC STACK MANIPULATION INSTRUCTIONS

`PUSH`: Adding an Element

The `PUSH` instruction places a value on top of the current stack. Its syntax is:

```
PUSH <value>
```

Here, `<value>` can be a literal, register name, label, or immutable constant. The value is evaluated and then appended to the top of the current stack.

Example:

```
PUSH 42
PUSH "hello"
```

`DUP` : Duplicating the Top Element

`DUP` duplicates the top element of the current stack, pushing the copy onto the stack. This is useful for saving a value temporarily before operations that consume stack data.

Example:

```
DUP
```

If the stack's top was `100`, after `DUP`, the top two elements will both be `100`.

Attempting to `DUP` an empty stack causes a runtime error.

`DROP` (also known as `RM`): Removing the Top Element

The `DROP` instruction removes the top element from the current stack without storing it elsewhere. It effectively discards the most recently pushed item.

Syntax:

```
DROP
```

If the stack is empty or no stack is selected, an error is generated.

`POP` : Removing the Top Element into a Register

`POP` removes the top element of the current stack and stores it in the specified register. Syntax is:

```
POP <register>
```

The `<register>` must be one of Vertigo's predefined registers, such as `ODA`, `IDA`, or user-defined ones. For example:

```
POP ODA
```

If the stack is empty or no stack is selected, or the register name is invalid, an error is raised.

`CLEAR`: Emptying the Current Stack

The `CLEAR` instruction removes all elements from the currently selected stack.

```
CLEAR
```

Attempting to clear without a selected stack produces an error. This instruction is useful to reset or reuse a stack during program execution.

STACK ELEMENT REARRANGEMENT INSTRUCTIONS

`SWAP`: Exchanging Top Two Elements

`SWAP` exchanges the top two elements on the current stack. It requires at least two elements; otherwise, it raises an error.

```
SWAP
```

Example: Suppose the current stack top elements are [A, B] (with B at the very top). After `SWAP`, the top two elements will be [B, A].

`PICK`: Duplicating an Element at a Given Depth

The `PICK` instruction duplicates the element at a specific depth measured from the top and pushes the duplicated value onto the top of the stack.
Syntax:

```
PICK <n>
```

Where `<n>` is a zero-based index indicating how far from the top to pick (e.g., `0` duplicates the top element). Instead of a direct numeric index, `<n>` may also be a register name holding a valid non-negative integer.

Example:

```
5  ; some values pushed before
PICK 2  ; duplicates the 3rd element from the top (index
2) and pushes it
```

If the index is out of range or the stack is too shallow, an error occurs.

`PPICK` : Picking and Removing an Element

Similar to `PICK` , `PPICK` takes an element at a given index from the top of the stack, duplicates and pushes it on top, then removes the original element from its original position. This effectively moves the element to the top of the stack.

```
PPICK <index_or_register>
```

This instruction adjusts stack order efficiently without multiple `SWAP` or manual pops and pushes.

ERROR CONDITIONS AND BEST PRACTICES

Because most stack operations depend on an explicitly selected current stack, it is vital to ensure proper stack context is set before executing such instructions. Failing to select a stack or attempting operations on an empty stack will raise runtime errors, typically of type `LookupError` or `IndexError` indicating no selected stack or insufficient elements.

For example, issuing `PUSH 10` without a current stack selected will trigger an error. Similarly, calling `DUP` on an empty current stack results in an error due to the absence of any element to duplicate.

Best practices include:

- Always define and create your stacks at program start or before use with `NEW`.
- Select the current stack explicitly by naming it on a line by itself before performing stack operations.
- Verify stack contents or sizes when performing operations dependent on stack depth, such as `SWAP`, `PICK`, and `PPICK`.
- Use `CLEAR` to reset stacks when no longer needed or to ensure import subroutines work on clean data.

SUMMARY OF STACK INSTRUCTIONS

Instruction	Syntax	Description	Errors
NEW	`NEW <stack_name>`	Create a new empty stack identified by `stack_name`.	SyntaxError if arguments invalid.
(Stack Selection)	`<stack_name>`	Selects the named stack as the current stack context for subsequent operations.	Error if `stack_name` does not exist is implicit during later ops.
PUSH	`PUSH <value>`	Pushes evaluated `value` onto current stack.	LookupError if no current stack selected.
DUP	`DUP`	Duplicates top element of current stack.	LookupError if no stack; ValueError if stack empty.
DROP / RM	`DROP`	Removes top element of current stack.	LookupError if no stack; ValueError if stack empty.
POP	`POP <register>`	Removes top stack element, stores in register.	LookupError if no current stack; ValueError if empty; NameError if invalid register.
CLEAR	`CLEAR`	Empties all elements of current stack.	ValueError if no stack selected.
SWAP	`SWAP`		LookupError if no stack; IndexError if

Instruction	Syntax	Description	Errors
		Swaps top two elements of current stack.	less than two elements.
PICK	`PICK <index_or_register>`	Duplicates element at index from top and pushes it to top.	LookupError if no stack; IndexError if index invalid; TypeError if register invalid.
PPICK	`PPICK <index_or_register>`	Duplicates element at index and removes original (moves to top).	Same as `PICK`.

REGISTERS AND REGISTER OPERATIONS

In Vertigo, **registers** act as named storage locations that hold single values and provide rapid access to important data during program execution. Registers serve as a bridge between low-level stack manipulations and higher-level control flow constructs, enabling temporary storage, loop control, arithmetic results, and interaction with input and output.

FUNDAMENTAL REGISTERS

Upon program initialization, Vertigo reserves several built-in registers. These include:

- `ODA` (Output Data Area): Used primarily to hold values destined for output or further processing after being popped from a stack or produced by arithmetic operations.
- `IDA` (Input Data Area): Holds user input gathered through the `IN` instruction or results from other data input mechanisms.
- `CLI` (Loop Counter Indicator): Tracks the current iteration count during loop execution, typically manipulated by `LOOP` and `ENDLOOP` instructions.
- `LTM` (Loop Termination Marker): Defines the maximum number of iterations for loops controlled by `LOOP` and `ENDLOOP` instructions.

These predefined registers are core to managing output, input, and iterative control within Vertigo programs.

REGISTER DECLARATION WITH `REG`

Beyond the built-in registers, new registers can be explicitly declared using the `REG` instruction, which follows the syntax:

```
REG <register_name>
```

Declared registers are initialized to `None`, representing an empty state. Registers serve as valid destinations for storage and arithmetic results, and their use is essential for organizing intermediate computations or flags.

Example:

```
REG LIN0
POP LIN0
```

This declares a register named `LIN0` and stores the top element of the current stack into it via `POP`.

STORING VALUES INTO REGISTERS: THE `POP` INSTRUCTION

The `POP` instruction simultaneously removes the top value from the current stack and stores it into a specified register. Its typical form is:

```
POP <register>
```

Where `<register>` is the target register name, such as `ODA`, `ID`, or any previously declared register.

This mechanism allows stack data to be transferred into quick-access storage for subsequent operations or conditional processing. If no stack is selected or the designated register is invalid, an error is thrown.

REGISTERS AS DESTINATIONS IN OPERATIONS

Many Vertigo instructions accept registers as destinations for their results. For example, the `MATH` instruction's syntax supports specifying a register to receive mathematical results:

```
MATH ADD <destination> <operand1> <operand2>
```

Here, `<destination>` can be a register like `ODA`, `LIN0`, or the special symbol `&`, indicating pushing the result onto the current stack instead.

Similarly, logical operations (`OPS`) and string manipulation instructions such as `CONCAT` use registers as destinations to store computed results.

LOOP CONTROL REGISTERS: `CLI` AND `LTM`

Registers `CLI` and `LTM` form a crucial part of Vertigo's loop control mechanism:

- `LTM` defines the maximum number of iterations a loop should perform.
- `CLI` tracks the current iteration count, incremented automatically during each loop cycle.

The `LOOP` instruction begins a loop block and initializes or modifies these registers to control iteration counts. Each iteration increments `CLI`. The paired `ENDLOOP` instruction compares `CLI` to `LTM` to determine whether to continue looping or exit.

If the loop counter `CLI` exceeds the termination marker `LTM` (or if `LTM` is zero), the loop terminates; otherwise, execution jumps back to the loop's start instruction pointer.

INTERPLAY OF REGISTERS AND STACKS

Registers and stacks work symbiotically in Vertigo to facilitate flexible data storage and manipulation:

- Registers can serve as sources or destinations for values pushed to and popped from stacks.
- Instructions such as `PUSH` can push register values onto the current stack.
- `POP` transfers the top stack element into a register, enabling non-destructive temporary storage.
- Arithmetic and logical instructions can directly read values from either registers or stacks and write results to registers or the current stack.

This design allows Vertigo programs to control data flow precisely, interleaving stack-based LIFO operations with quick-value retains in registers.

IMMUTABLES: CONSTANTS STORED IN REGISTERS

Vertigo also supports the notion of immutables, which are named constant values stored with a special naming convention using the `+` prefix. Defined with the `IM` instruction:

```
IM <name> <value>
```

For instance, `IM PI 3.14159` defines the immutable `+PI` with the value `3.14159`.

These immutables cannot be redefined during the program execution, providing fixed constants accessible similarly to registers but safeguarded from modification.

Accessing an immutable's value uses its prefixed name (for example, `+PI`) in instructions requiring operands, just like using registers or literals.

Immutables offer a convenient way to include constant data in programs without risk of accidental alteration.

SUMMARY OF KEY REGISTER CONCEPTS

Register	Role	Usage Notes
`ODA`	Output Data Area — stores output and intermediate results.	Common destination for `POP` and results of arithmetic/logical ops.
`IDA`	Input Data Area — holds user input or external input data.	Populated by `IN` instruction.
`CLI`	Loop Counter Indicator — tracks current loop iteration count.	Managed automatically inside `LOOP` / `ENDLOOP` blocks.
`LTM`	Loop Termination Marker — maximum iterations for loops.	Defines loop boundary; zero disables loop repetition.
Other Registers (Declared via `REG`)	General-purpose named storage locations.	Hold temporary values for calculations, flags, or results.

Register	Role	Usage Notes
Immutables (e.g., `+PI`)	Named constants immutable during execution.	Declared with `IM`, accessed like registers but cannot be altered.

ERROR HANDLING FOR REGISTERS

Vertigo enforces strict rules on register usage to ensure program correctness:

- Attempting to `POP` into a nonexistent or undeclared register raises a `NameError`.
- Referencing undefined registers as operands or destinations in operations causes execution errors.
- Immutables cannot be redefined; attempts to do so result in warnings or ignored redefinitions recorded in logs.
- Using registers before declaration (if not built-in) will result in lookup errors or undefined behavior.

Proper register declaration and careful distinction between registers and immutables greatly improve code clarity and stability.

CONTROL FLOW INSTRUCTIONS

Control flow instructions are fundamental to any programming language, determining the order in which instructions are executed. In Vertigo, control flow is managed through explicit jump instructions, conditional branching based on comparison flags, iterative loops, and subroutine calls. This section details the instructions that allow programmers to alter the sequential execution path.

LABELS AND JUMP TARGETS: THE `POINT` INSTRUCTION

The simplest way to alter control flow is through explicit jumps to specific locations within the code. These locations are marked by **labels**, which are symbolic names defined using the `POINT` instruction.

A label declaration consists of the `POINT` instruction followed by the desired label name:

```
POINT <label_name>
```

Labels must be unique within a program. They do not execute any operation themselves but serve purely as markers in the source code. During the first pass of interpretation (often called the "label pass"), the interpreter records the line number corresponding to each label. These line numbers are then used as destinations for jump instructions during execution.

Example:

```
POINT my_loop_start
  ; Instructions here
  ; ...
JUMP my_loop_start  ; This instruction will jump
execution back to the line after POINT my_loop_start
```

Labels are essential for implementing loops, conditional blocks, and modular program structures.

UNCONDITIONAL JUMPS: THE `JUMP` INSTRUCTION

The `JUMP` instruction provides an unconditional transfer of control to a specified label. When the interpreter encounters a `JUMP` instruction, it immediately sets the instruction pointer to the line number associated with the target label, and execution continues from that point.

Syntax:

```
JUMP <label_name>
```

The `<label_name>` must correspond to a label previously defined using the `POINT` instruction. Attempting to jump to an undefined label will result in a runtime error (`LookupError` or `NameError`).

Example:

```
; Some code
JUMP end_program  ; Skip intermediate code
; Code that is skipped
POINT end_program
; Program continues here
```

COMPARISON AND CONDITIONAL JUMPS

Vertigo supports conditional execution based on comparisons between values. This is achieved using comparison instructions that set internal comparison flags, followed by conditional jump instructions that check these flags.

Comparison Instructions: `CMP` and `STRCMP`

The `CMP` and `STRCMP` instructions compare two operands and set internal flags indicating the result of the comparison (equal, greater than, or less than). These flags influence the behavior of subsequent conditional jump instructions.

- **`CMP` (Compare):** Compares two numeric or string operands. Syntax: `CMP <operand1> <operand2>` Operands can be literals, registers, or stack values (using `@`). Both operands must be of the same type (both numbers or both strings).
- **`STRCMP` (String Compare):** Specifically compares two string operands lexicographically. Syntax: `STRCMP <operand1> <operand2>` Operands can be string literals, registers holding strings, or string values from the stack.

Both `CMP` and `STRCMP` set three internal comparison flags:

- Equal flag: Set if `operand1` is equal to `operand2` .
- Greater flag: Set if `operand1` is greater than `operand2` .
- Less flag: Set if `operand1` is less than `operand2` .

These flags are binary (true/false). A subsequent conditional jump instruction checks the state of the relevant flag.

Conditional Jump Instructions

After a comparison instruction (`CMP` or `STRCMP`) has been executed, one of the following conditional jump instructions can be used:

- **`JUMPEQ` (Jump if Equal):** Transfers control to the target label if the Equal flag is set (operands were equal). Syntax: `JUMPEQ <label_name>`
- **`JUMPGT` (Jump if Greater Than):** Transfers control to the target label if the Greater flag is set (operand1 was greater than operand2). Syntax: `JUMPGT <label_name>`

- **`JUMPLT` (Jump if Less Than):** Transfers control to the target label if the Less flag is set (operand1 was less than operand2). Syntax: `JUMPLT <label_name>`
- **`JUMPNEQ` (Jump if Not Equal):** Transfers control to the target label if the Equal flag is NOT set (operands were not equal). Syntax: `JUMPNEQ <label_name>`

If the condition corresponding to the jump instruction is not met (the relevant flag is false), execution continues sequentially to the next instruction.

Example of Conditional Logic:

```
PUSH 10
PUSH 20
POP ODA   ; ODA = 20
POP IDA   ; IDA = 10

CMP IDA ODA ; Compare 10 and 20 (10 < 20)

JUMPGT greater_label  ; Will NOT jump (10 is not > 20)
JUMPEQ equal_label    ; Will NOT jump (10 is not == 20)
JUMPLT less_label    ; Will JUMP (10 is < 20)

POINT greater_label
  ; Code for greater condition

POINT equal_label
  ; Code for equal condition

POINT less_label
  ; Code for less condition
  ; Execution continues here after JUMPLT
```

ITERATIVE LOOPS: `LOOP` AND `ENDLOOP`

Vertigo provides a simple mechanism for controlled iteration using the `LOOP` and `ENDLOOP` instructions, which rely on the built-in `CLI` (Loop Counter Indicator) and `LTM` (Loop Termination Marker) registers.

A loop block is defined by a `LOOP` instruction at the start and an `ENDLOOP` instruction at the end.

```
LOOP
  ; Instructions inside the loop body
  ; ...
ENDLOOP
```

When the `LOOP` instruction is executed, the interpreter stores the current instruction pointer (the line *after* the `LOOP` instruction) as the start of the loop body. The `CLI` register is incremented.

When the `ENDLOOP` instruction is reached, the interpreter checks the value in the `CLI` register against the value in the `LTM` register.

- If `CLI` is less than or equal to `LTM` (and `LTM` is not zero), execution jumps back to the instruction pointer saved at the beginning of the `LOOP` block.
- If `CLI` is greater than `LTM` (or if `LTM` is zero), the loop terminates, and execution continues with the instruction immediately following `ENDLOOP`.

Before entering the loop, you typically set the `LTM` register to the desired number of iterations. The `CLI` register is automatically managed by the loop structure. Note that the loop runs while `CLI <= LTM`.

Example Loop:

```
; Set LTM to 5 for 5 iterations (CLI goes from 1 to 5)
PUSH 5
POP LTM

; CLI is implicitly reset or handled by LOOP/ENDLOOP
logic
; Ensure CLI is 0 before the first LOOP if needed, though
the implementation manages it.

LOOP
  ; This code runs 5 times
  ; PUSH CLI  ; Example: Push the current iteration count
to stack
  ; ...
```

```
ENDLOOP
; Execution continues here after the loop finishes
```

Proper management of `LTM` is crucial for controlling loop termination. A value of 0 in `LTM` will cause the loop to execute zero times (as `CLI` will immediately be greater than `LTM` after the first `LOOP` check, or the check might prevent entry entirely depending on implementation specifics, but in the provided code, `CLI` increments and then is compared to `LTM`, so if LTM is 0, CLI becomes 1, 1 <= 0 is false, and it exits).

SUBROUTINES: `SUB`, `ENDSUB`, AND `CALL`

For code modularity and reusability, Vertigo supports subroutines. A subroutine is a named block of code that can be executed from different parts of the program.

Defining Subroutines: `SUB` and `ENDSUB`

A subroutine definition starts with the `SUB` instruction followed by the subroutine name and ends with `ENDSUB`.

```
SUB <subroutine_name>
  ; Instructions within the subroutine body
  ; ...
ENDSUB
```

Subroutine definitions are processed during the initial pass, similar to labels. The code within a `SUB` / `ENDSUB` block is not executed sequentially during the main program flow. Instead, it is stored and executed only when explicitly called.

Calling Subroutines: `CALL`

The `CALL` instruction is used to execute a defined subroutine.

```
CALL <subroutine_name>
```

When `CALL` is executed, the interpreter saves the current instruction pointer (the line *after* the `CALL` instruction) onto an internal **return stack**.

Execution then jumps to the first instruction within the specified subroutine's body. The subroutine executes line by line. Upon reaching the implicit end of the subroutine's code (or an explicit return mechanism if one were implemented, though the provided code implies execution simply finishes the code block), the interpreter pops the last saved instruction pointer from the return stack and resumes execution from there.

Nested calls are supported because the return stack allows multiple return addresses to be saved.

Example:

```
; Main program flow
PUSH "Starting..."
POP ODA
CALL my_subroutine
PUSH "Resumed after subroutine."
POP ODA
; Program ends

; Subroutine definition
SUB my_subroutine
  PUSH "Inside subroutine."
  POP ODA
  ; No explicit ENDSUB needed here in execution flow,
  ; the definition ends with ENDSUB in the source code.
  ; Execution returns automatically after the last
instruction.
ENDSUB
```

Attempting to `CALL` an undefined subroutine results in a `NameError`.

EXTERNAL CODE INCLUSION: `BRING` AND `IMPORT`

Vertigo allows incorporating code from external files through two mechanisms:

- **`BRING`**: Imports Vertigo subroutine definitions from a library file (a `.vtgo` file formatted in a specific library structure). Syntax: `BRING <library_filename>` The specified library file is expected to contain multiple subroutine definitions separated by colons (`:`). `BRING` reads

these definitions and makes the subroutines available for `CALL`ing in the current program.
- **`IMPORT`**: Loads an external Python module, allowing Vertigo to potentially interact with Python functions or data structures made available by that module. Syntax: `IMPORT <module_name>` This capability bridges Vertigo execution with external Python functionality, although the specifics of this interaction depend on how the imported Python module is designed to interface with Vertigo's runtime environment (e.g., via interrupt handlers).

These instructions facilitate code organization, reuse, and interoperability with other systems.

MATHEMATICAL AND LOGICAL OPERATIONS

Vertigo provides powerful instructions for performing mathematical and logical operations, integral to algorithmic processing and decision-making within programs. These operations are primarily handled through the `MATH` instruction for arithmetic calculations and the `OPS` instruction for logical and boolean evaluations.

THE `MATH` INSTRUCTION: ARITHMETIC OPERATIONS

The `MATH` instruction performs arithmetic operations between two operands and stores the result either in a specified register or pushes it onto the current stack. Its general syntax is:

```
MATH <operation> <destination> <operand1> <operand2>
```

Where:

- `<operation>` is one of `ADD`, `MINUS`, `MUL`, `DIV`, `MOD`, `POW`.
- `<destination>` specifies where the result goes: either a register name like `ODA`, `LIN0`, or the special symbol `&` meaning the current stack.
- `<operand1>` and `<operand2>` are the two numerical values to be used in the operation, which can be literals, registers, immutables, or stack elements.

Supported Operations

- **ADD:** Adds `operand1` and `operand2`.
- **MINUS:** Subtracts `operand2` from `operand1`.
- **MUL:** Multiplies `operand1` by `operand2`.
- **DIV:** Divides `operand1` by `operand2`. Division by zero raises a runtime error.
- **MOD:** Computes the modulo (remainder) of `operand1` divided by `operand2`.
- **POW:** Calculates `operand1` raised to the power of `operand2`.

Operand Types and Evaluation

Both operands must be numeric values (integers or floating-point numbers). Vertigo evaluates operands as follows:

- Literals (e.g., `42`, `-3.5`) are used directly.
- Registers return their stored numeric value.
- Immutables return their constant value.
- Stack elements can be referenced using special notations (e.g., using the `@` prefix).

Destination and Storage

The result of the arithmetic operation is either:

- Stored in the register named by `<destination>`, if it is a valid register.
- Pushed onto the current stack if the special symbol `&` is used as `<destination>`.

If no current stack is selected and `&` is specified as the destination, a runtime error occurs.

Error Handling

- **Division by Zero:** Attempting to divide by zero with `DIV` results in a `ZeroDivisionError` at runtime.
- **Operand Type Mismatch:** If either operand is non-numeric or undefined, a runtime arithmetic error occurs.
- **Invalid Destination:** If the destination is neither a registered name nor `&`, an error is raised.

- **Stack Not Selected:** Using `&` without a current stack causes a runtime error.

Examples of `MATH` Usage

```
; Adds 10 and 32, pushes result on current stack
MATH ADD & 10 32

; Multiplies 7 by 6 and stores result in ODA register
MATH MUL ODA 7 6

; Raises 2 to the power 8 and stores in LIN0 register
REG LIN0
MATH POW LIN0 2 8

; Attempts division by zero - will raise an error
MATH DIV ODA 10 0
```

THE `OPS` INSTRUCTION: LOGICAL OPERATIONS

The `OPS` instruction provides logical and boolean operations with flexible argument support. It operates on one or more operands and stores the result in either a register or the current stack, similarly to `MATH`.

```
OPS <operation> <destination> <operand1> [<operand2> ...]
```

Supported Logical Operations

- **AND:** Returns true (1) if all operands are logically true (non-zero); otherwise false (0).
- **OR:** Returns true (1) if any operand is logically true; otherwise false (0).
- **NOT:** Logical negation of a single operand; returns 1 if operand is false (zero), else 0.
- **EQUAL:** Returns 1 if two operands are equal; else 0.
- **NEQUAL:** Returns 1 if two operands are not equal; else 0.

Logical results are always represented as numeric integers `0` (false) or `1` (true).

Operands and Evaluation

Operands for `OPS` are evaluated in the same manner as `MATH`: they may be literals, registers, immutables, or stack values. The number of operands depends on the operation:

- `AND` and `OR`: require two or more operands.
- `NOT`: requires exactly one operand.
- `EQUAL` and `NEQUAL`: require exactly two operands.

Result Destination

The computed logical result is stored similarly as with `MATH`:

- In a register specified by `<destination>`, or
- Pushed onto the current stack if `&` is specified.

An error occurs if the destination is invalid or no current stack is selected when `&` is used.

Error Handling

- **Invalid Operation:** Only the listed logical operations are accepted; unknown operations cause errors.
- **Argument Count Mismatches:** Incorrect number of operands for the given operation produce syntax errors.
- **Invalid Operand Type:** Logical operations treat zero as false and any non-zero value as true; operands of any type convertible to integer logic are accepted.
- **Invalid Destination or Unselected Stack:** Similar to `MATH`, errors occur if destinations are invalid or stack context is missing.

Examples of `OPS` Usage

```
; Logical AND of three values, store result in ODA
OPS AND ODA 1 1 0    ; Result: 0 (false), since one
operand is zero

; Logical OR of three values, push result on current
stack
OPS OR & 0 0 42   ; Result: 1 (true), since 42 is non-
zero
```

```
; Logical NOT of a register LIN0, store result in LIN1
REG LIN0
REG LIN1
PUSH 0
POP LIN0
OPS NOT LIN1 LIN0    ; LIN1 becomes 1 because LIN0 is 0
(false)

; Check if two registers are equal, store on stack
REG A
REG B
PUSH 15
POP A
PUSH 15
POP B
OPS EQUAL & A B   ; Pushes 1 on stack (true)

; Check if two immutables are not equal, store in
register IDA
IM CONST1 100
IM CONST2 200
OPS NEQUAL IDA +CONST1 +CONST2  ; IDA = 1 (true)
```

SUMMARY

Together, the `MATH` and `OPS` instructions provide essential computational capabilities in Vertigo. Arithmetic operations enable numeric calculations with flexible destinations, while logical operations allow complex condition evaluations supporting control flow decisions. Both instructions carefully validate operand types, destination targets, and handle error conditions such as division by zero and invalid argument counts to ensure robust and predictable program behavior.

STRING HANDLING INSTRUCTIONS

Vertigo includes several instructions dedicated to the manipulation and comparison of string data. These instructions treat strings as sequences of characters enclosed within double quotes (`"`), enabling safe inclusion of spaces and special symbols within string literals. String values may also reside

in registers or on stacks, and Vertigo's string operations support these flexible data sources. Below is a detailed description of the primary string handling instructions.

`CONCAT` : CONCATENATION OF TWO STRINGS

The `CONCAT` instruction combines two values into a single string by concatenating their string representations. The resulting string is stored in a specified register. Its syntax is:

```
CONCAT <destination_register> <value1> <value2>
```

Operands and Destination:

- `<destination_register>` : Must be an existing register capable of storing string data.
- `<value1>` and `<value2>` : Can be string literals enclosed in double quotes, registers containing string data, or stack values referenced via special syntax.

Operation: Both operands are converted to strings if necessary, then concatenated in the order they appear. The final combined string is stored in the designated register.

Error Conditions:

- An error is raised if the destination is not a valid register.
- If either operand is not a string or cannot be converted to a string (unusual but possible), a runtime type error occurs.

Example:

```
CONCAT ODA "Hello, " "World!"    ; ODA now contains
"Hello, World!"
```

`STRLEN` : OBTAINING THE LENGTH OF A STRING

The `STRLEN` instruction calculates the length of a given string and stores this integer length in the specified register. The syntax is:

```
STRLEN <destination_register> <string_value>
```

Operands:

- `<destination_register>` : Must be a valid register to hold the integer length.
- `<string_value>` : Can be a string literal, the name of a register containing a string, or a stack value that evaluates to a string.

Operation: Returns the count of characters in the provided string operand.

Error Conditions:

- Raises a type error if the operand is not a string.
- Raises an error if the destination register is invalid or nonexistent.

Example:

```
STRLEN ODA "Vertigo"     ; ODA receives the value 7
```

`STRCMP` : STRING COMPARISON

The `STRCMP` instruction compares two strings lexicographically and sets the internal comparison flags accordingly. These flags are later used by conditional jump instructions to control program flow based on string ordering or equality. Its syntax is:

```
STRCMP <string1> <string2>
```

Operands:

- `<string1>` and `<string2>` : Both operands must be strings, provided either as string literals, registers containing string values, or stack-based string values.

Operation:

- Compares the two strings lexicographically (dictionary order).
- Sets three comparison flags:
  - **Equal:** true if `string1` equals `string2`

    - **Greater:** true if `string1` is lexicographically greater than `string2`
    - **Less:** true if `string1` is lexicographically less than `string2`

Error Conditions:

- Raises a type error if either operand is not a string.

Example:

```
STRCMP "apple" "banana"   ; Sets flags: Equal=false,
Less=true, Greater=false
```

STRING LITERALS AND PARSING

In Vertigo, string literals must be enclosed in double quotes (`"..."`) to differentiate them from identifiers, keywords, and numeric literals. This allows string values to include spaces, punctuation, and other characters without confusion. For example:

```
PUSH "This is a string with spaces and symbols!"
```

Vertigo's parsing logic treats the entire double-quoted sequence as a single token, preserving all internal characters exactly. Escape sequences within strings are not specified, so to include a literal double quote inside a string requires careful handling or avoidance.

When strings are used as operands to instructions like `CONCAT` , `STRLEN` , and `STRCMP` , they are interpreted as string values. Registers used as operands must contain strings; otherwise, a type error results. Stack references (via `@` notation) yielding strings can also be used in these instructions, permitting dynamic string operations based on runtime stack contents.

SUMMARY OF STRING HANDLING BEST PRACTICES

- Always enclose string literals within double quotes to ensure correct parsing.
- Ensure that registers intended for string operations actually contain string data to avoid type errors.

- Use `CONCAT` to build new strings by joining two string values and store results in registers.
- Obtain string lengths using `STRLEN` to facilitate operations that depend on string size.
- Perform lexicographical comparisons using `STRCMP` to guide conditional control flow.

INPUT/OUTPUT AND DEBUGGING

Vertigo provides several instructions and mechanisms to facilitate user interaction, program output, and debugging support during development. This section describes the core input/output instructions and tools designed to assist in monitoring and controlling program execution.

THE `IN` INSTRUCTION: READING USER INPUT

The `IN` instruction enables a Vertigo program to read input from the user interactively. It optionally accepts a prompt string which is displayed before waiting for user input. The input value is then stored in the built-in register `IDA` (Input Data Area).

Syntax:

```
IN [<prompt_string>]
```

If a prompt string is provided (enclosed in double quotes), it is printed to the console to guide the user. The program then reads a line of input from the standard input.

Vertigo attempts to intelligently convert numeric input strings into appropriate numeric types (integers or floating point) before placing the value into `IDA`. Non-numeric input remains as a string. This conversion enables seamless numeric and textual user input handling within the language.

Example:

```
IN "Enter your age: "
; User types 35
; IDA now contains the number 35 (integer)
```

INTERRUPT-BASED OUTPUT: `INT` INSTRUCTION AND `PRINTINT()` FUNCTION

Vertigo supports an internal interrupt mechanism that can trigger specific functions during program execution. The `INT` instruction invokes a predefined interrupt handler based on an interrupt code supplied as an argument.

Syntax:

```
INT <interrupt_code>
```

One common interrupt is the printing of the current contents of the `ODA` register (Output Data Area). The registered interrupt handler `printint()` checks if an internal flag `intpr` is enabled; if so, it prints the value stored in `ODA` to the console and then clears `ODA` by setting it to `None`.

This interrupt-driven output provides a non-intrusive way to generate controlled program output, especially useful in interactive or event-driven scenarios.

THE `DUMP` INSTRUCTION: RUNTIME STATE INSPECTION AND LOGGING

Debugging complex Vertigo programs is facilitated by the `DUMP` instruction, which allows inspection and recording of program runtime state. The behavior of `DUMP` depends on its optional argument:

- `DUMP` without arguments prints the complete state of all stacks to the console, giving a snapshot of all runtime data structures.
- `DUMP @` prints only the currently selected stack's contents, allowing focused inspection of the active working data.
- `DUMP LOGS` writes the accumulated execution log to a timestamped dump file with the `.vtd` extension. The filename includes the current minute, second, and year to ensure uniqueness, e.g., `dump-30242324.vtd`. This log file records a trace of instructions executed and their arguments, proving invaluable for offline analysis or auditing program behavior.

The dump files generated are plaintext and formatted for easy human reading or further automated processing. These facilities enable developers

to identify logic errors, verify stack manipulations, and trace program execution step-by-step.

USING INPUT/OUTPUT AND DEBUGGING FACILITIES IN DEVELOPMENT

Together, the `IN`, `INT`, and `DUMP` instructions form a minimal but effective set of tools for interaction and debugging:

- **User Interaction:** Use `IN` to request input or trigger interactive prompts, storing the result in `IDA` for program logic.
- **Output Data Monitoring:** Use `INT` with the appropriate interrupt code to print important values held in `ODA` without disrupting program flow.
- **Runtime Inspection:** Use `DUMP` commands periodically or on demand to monitor or log the internal stack data, informing debugging or profiling activities.

These instructions are valuable during development, testing, and troubleshooting as they provide visibility into Vertigo's internal state and user-driven data exchange. Leveraging the structured logging of `DUMP LOGS` improves repeatability and postmortem analysis in complex projects.

SUBROUTINES AND MODULAR CODE

DEFINING SUBROUTINES WITH `SUB` AND `ENDSUB`

Vertigo supports modular programming through named subroutines—discrete blocks of code that can be defined once and invoked multiple times from various points in the program. Subroutines encapsulate functionality, promote reuse, and improve code readability.

A subroutine definition begins with the `SUB` instruction followed by a unique subroutine name and ends with the `ENDSUB` instruction. The syntax is:

```
SUB <subroutine_name>
  ; subroutine instructions
ENDSUB
```

The interpreter processes subroutine definitions during the program's first pass. The lines of code within the `SUB` / `ENDSUB` block are stored internally

but are not executed sequentially as the main program runs. Instead, these instructions are invoked only when the subroutine is explicitly called.

Restrictions: Each subroutine name must be unique within a program to avoid conflicts. Redeclaring an existing subroutine causes a runtime `NameError`.

Subroutines have access to the same stacks, registers, and immutables as the main program. They share the program's global state, enabling seamless data manipulation across boundaries.

INVOKING SUBROUTINES WITH `CALL`

To execute a defined subroutine, the `CALL` instruction specifies its name:

```
CALL <subroutine_name>
```

When `CALL` is encountered, Vertigo performs the following actions:

- Saves the current instruction pointer (the line following the `CALL`) onto an internal **return stack**. This mechanism supports nested subroutine calls and proper return sequencing.
- Transfers execution to the first instruction in the specified subroutine's stored code block.
- Executes the subroutine's instructions in order, line by line.
- After reaching the end of the subroutine code block (implicitly upon reaching the last line), Vertigo retrieves the return address from the return stack and resumes execution from that point in the main program.

If the subroutine name is undefined, a `NameError` is thrown. Since subroutines execute with access to all global stacks and registers, they can push, pop, modify, and read shared state, allowing versatile program flow control.

Example:

```
SUB greet
  PUSH "Hello, Vertigo!"
  POP ODA
ENDSUB
```

```
PUSH "Program start"
POP ODA
CALL greet
PUSH "Returned from greet"
POP ODA
```

MODULAR CODE IMPORTING WITH `BRING`

To promote code reuse and modularity beyond a single source file, Vertigo offers the `BRING` instruction. It imports subroutine definitions from an external library file into the current program's environment.

Syntax:

```
BRING <library_filename>
```

The specified `library_filename` should be a text file formatted with multiple subroutine blocks separated by colons (`:`). Each block consists of a subroutine name immediately followed by that subroutine's code lines.

For example, a library file might contain:

```
greet:
PUSH "Hello from library!"
POP ODA
endsub:

farewell:
PUSH "Goodbye from library!"
POP ODA
endsub:
```

The `BRING` instruction parses these blocks and registers their subroutines in the interpreter's subroutine table, making them callable with `CALL` . This approach enables sharing and organizing common routines across multiple projects.

Key Points:

- Subroutines imported by `BRING` retain their original names and cannot overwrite existing subroutines unless explicitly handled by the program logic.
- Imported subroutines gain full access to global stacks, registers, and immutables within the current program context.
- Failure to locate or read the library file triggers a runtime `FileNotFoundError` or `SyntaxError` if the file's structure is malformed.

ADDITIONAL MODULAR LOADING: `IMPORT`

Besides importing Vertigo subroutines, the `IMPORT` instruction allows integrating external Python modules into the Vertigo environment:

```
IMPORT <python_module_name>
```

This instruction reads and executes the specified Python module (with `.py` extension) at runtime, enabling Vertigo programs to leverage Python code functionalities as extensions or libraries. This powerful interoperability makes Vertigo extensible and suitable for embedding in more comprehensive toolchains.

SUMMARY

Vertigo's subroutine system, centered on the `SUB`, `ENDSUB`, and `CALL` instructions, forms the foundation for modular and reusable code structures. They enable developers to define named blocks of logic that can be invoked from multiple locations with preserved execution context. The `BRING` instruction substantially enhances modularity by importing collections of subroutines from external library files, facilitating code sharing and maintenance. Finally, `IMPORT` bridges Vertigo with external Python modules, further broadening its capabilities and integration potential.

ADDITIONAL UTILITIES AND INSTRUCTIONS

Beyond core stack and control flow instructions, Vertigo provides a set of additional utilities designed to enhance program flexibility and control. These instructions focus on interpreter settings, execution timing, interrupt handling, dynamic register declaration, and stack element rearrangement.

Collectively, they support robust and safe program behavior through explicit error checking and well-defined semantics.

`SET` : CONFIGURING INTERPRETER SETTINGS

The `SET` instruction allows runtime modification of certain interpreter settings. Its syntax is:

```
SET <setting_name> <value>
```

Currently, the primary setting controlled by `SET` is `intpr`, which governs whether the interpreter automatically prints the contents of the `ODA` register after instructions that modify it.

Setting `intpr` to `True` enables automatic printing via internal interrupt-based mechanisms, reducing the need for explicit output instructions. Setting it to `False` disables this feature, requiring manual output commands.

Example:

```
SET intpr True  ; Enable interrupt-driven output printing
```

Error Conditions:

- Raising a `NameError` if the specified setting name is unrecognized.
- Rejecting invalid values that cannot be evaluated or cast appropriately.

`WAIT` : PAUSING EXECUTION

The `WAIT` instruction pauses program execution for a specified fraction of a second, facilitating timing control and synchronization with external events or user interaction.

Syntax:

```
WAIT <duration>
```

Where `<duration>` is interpreted as hundredths of a second (1 unit = 0.01 seconds). For example, `WAIT 50` pauses execution for 0.5 seconds.

Example:

```
WAIT 100  ; Pause for one full second
```

Robustness: The interpreter ensures that numerical arguments are valid and non-negative. Invalid or missing arguments cause runtime errors, guarding against unintended infinite pauses or crashes.

`INT` : TRIGGERING INTERRUPTS

The `INT` instruction invokes a predefined interrupt handler tied to an internal interrupt code, enabling integration with built-in interpreter functions or external event-driven routines.

Syntax:

```
INT <interrupt_code>
```

Currently, the main interrupt supported is printing the output value stored in the `ODA` register, controlled by the internal flag `intpr`. When this interrupt is triggered, if enabled, the interpreter prints `ODA` and then clears it.

Example:

```
INT 1  ; Trigger the print interrupt if enabled
```

Rigorous error checking ensures that invalid interrupt codes result in immediate runtime errors, preventing undefined behavior or security issues.

`REG` : DECLARING REGISTERS DYNAMICALLY

While Vertigo defines several built-in registers by default, the `REG` instruction allows programmers to declare new, named registers during execution.

Syntax:

```
REG <register_name>
```

The newly declared register is initialized with a `None` value, ready to be used as a destination for `POP`, `MATH`, or any instruction that stores data to registers.

Example:

```
REG LIN0  ; Define a new register LIN0 for temporary
storage
```

Attempts to redeclare an existing register do not produce errors but simply leave the register unchanged, promoting idempotent register management.

STACK ELEMENT ROTATION: `RROT` AND `ROT`

To facilitate complex stack rearrangements without multiple pops and pushes, Vertigo provides two specialized rotation instructions that manipulate the top three elements of the current stack:

- **`RROT` (Right Rotate):** Rotates the top three elements so that the top element moves to the bottom, and the second and third elements shift up accordingly.
- **`ROT` (Left Rotate):** Rotates the top three elements so that the third element becomes the top, moving the top and second elements down by one position.

These transformations change the order of stack elements as follows (top is rightmost):

Original Top 3 Elements	After `RROT`	After `ROT`
[A, B, C]	[C, A, B]	[B, C, A]

Usage Example:

```
RROT
ROT
```

Robustness and Error Checking:

- Both instructions require exactly three or more elements on the current stack; otherwise, an `IndexError` is raised.

- Execution fails with a `ValueError` if no current stack is selected, ensuring predictable stack context.

ERROR HANDLING AND EXECUTION MODEL

OVERVIEW OF ERROR HANDLING IN VERTIGO

Vertigo's interpreter employs a rigorous error handling model designed to ensure robust and reliable execution of programs. Errors are detected during parsing, instruction dispatch, and runtime execution phases, and are reported with detailed contextual information to facilitate debugging. Common categories of errors include **syntax errors**, **lookup errors** (such as undefined labels or registers), **type errors**, and **runtime errors** like division by zero or stack underflow.

SYNTAX ERRORS

Syntax errors are generally detected when the interpreter encounters unknown instructions, incorrect argument counts, or malformed instruction formats during parsing or initial validation. For example, an instruction with missing or extra arguments triggers a `SyntaxError`. Additionally, any unknown opcode mnemonic not recognized by the interpreter causes immediate termination with an error message specifying the offending instruction and line number.

The interpreter reports syntax errors by printing a descriptive message including the program file name and the specific line number where the error occurred. This precise feedback helps developers quickly locate and fix textual or format-related mistakes in their `.vtgo` source code.

LOOKUP ERRORS

Lookup errors arise when instructions refer to identifiers that have not been defined or initialized. Typical cases include:

- **Undefined Labels:** Jump instructions (`JUMP` , `JUMPEQ` , etc.) attempting to transfer control to a label that was never declared using `POINT` will raise a `LookupError`.
- **Invalid Registers:** Instructions like `POP` or any operation expecting a valid register as a destination will fail with a `NameError` if the given register name is invalid or undeclared.

- **Unselected Current Stack:** Many stack operations require a currently selected stack context. If no stack has been named or selected prior to a stack operation like `PUSH` or `DUP`, a `LookupError` indicates missing stack context.

Lookup errors are reported with clear error messages that include the missing identifier and the corresponding program line to guide immediate correction.

TYPE ERRORS

Vertigo enforces careful type checking to ensure operations receive compatible operands. Type errors are raised in several scenarios:

- Arithmetic instructions such as `MATH` require numeric operands. Providing non-numeric values (e.g., strings where numbers are expected) causes a `ArithmeticError` or `TypeError`.
- String operations (`CONCAT`, `STRLEN`, `STRCMP`) demand string operands. Passing incompatible types generates a `TypeError`.
- Logical operations (`OPS`) check operand count and validity per operation and reject invalid input types.
- Comparison instructions like `CMP` require both operands to be of the same type, either both numeric or both strings, otherwise a `TypeError` occurs.

These constraints promote predictable behavior and prevent nonsensical computations, with errors specifying the nature of the type incompatibility.

RUNTIME ERRORS

Several runtime errors may occur during execution due to invalid program state or data conditions:

- **Division by Zero:** The `MATH DIV` operation explicitly guards against division by zero, raising a `ZeroDivisionError` to prevent undefined arithmetic.
- **Stack Underflow:** Operations that pop or access elements from an empty or insufficiently sized stack (e.g., `POP`, `DUP`, `SWAP`, `PICK`) generate `IndexError` or `ValueError` errors indicating an underflow condition.
- **Invalid Stack Indices:** Instructions that access stack elements by index (such as `PICK` or `PPICK`) check for valid non-negative indices within stack bounds, raising errors if violated.

- **Missing or Malformed Library Files:** The `BRING` instruction raises file-related errors if a specified library file is unavailable or incorrectly formatted.

These runtime protections ensure that programs fail fast and with informative diagnostics instead of producing corrupted state or unpredictable outcomes.

EXCEPTION REPORTING AND DEBUGGING INFORMATION

Vertigo's interpreter is configured to catch all uncaught exceptions and report them in a structured format. When an error occurs, the interpreter prints:

- The source file name triggering the error.
- The approximate line number corresponding to the instruction pointer at the error occurrence.
- The type of exception (e.g., `SyntaxError`, `LookupError`, `ZeroDivisionError`, etc.).
- The detailed error message describing the cause.

This consistent reporting mechanism greatly aids debugging by providing immediate contextual clues. Additionally, the interpreter maintains an execution log internally, accessible through the `DUMP LOGS` instruction, enabling offline tracing of execution steps and arguments to pinpoint errors after program runs.

INSTRUCTION POINTER AND EXECUTION FLOW MODEL

Vertigo programs execute instructions sequentially, managed by a central instruction pointer (IP) that starts at the first instruction (line zero) and increments by one after each instruction, moving linearly through the source file.

This sequential model continues uninterrupted unless altered explicitly by control flow instructions:

- **Jump Instructions:** Instructions like `JUMP`, `JUMPEQ`, `JUMPGT`, and their variants directly modify the instruction pointer to the line number recorded for the target label. This causes the interpreter to continue execution from that new location.
- **Loop Constructs:** The `LOOP` and `ENDLOOP` instructions manage instruction pointer jumps within iterative blocks according to loop counters stored in registers `CLI` and `LTM`.

- **Subroutines:** When `CALL` is invoked, the current instruction pointer is saved on a return stack before jumping to the subroutine's first line. Execution within the subroutine proceeds sequentially. After completion, the saved return address is popped, and the instruction pointer resumes from that position, continuing main program execution.

Control flow modifications thus temporarily redirect the instruction pointer but always preserve a reliable mechanism to resume or exit loops and subroutines properly.

IMPORTANCE OF CORRECT INSTRUCTION ORDER AND STATE MANAGEMENT

Because Vertigo's interpreter operates on explicit instruction sequences with stateful stacks and registers, the program's correctness heavily depends on:

- Proper ordering of instructions to maintain logical data flows and avoid premature or unexpected jumps.
- Ensuring stack and register states are correctly established and preserved across instructions, especially when subroutines or loops are involved.
- Explicit selection of the current stack before stack operations to guarantee that value manipulations happen in the intended data context.
- Careful handling of registers and immutables to prevent unintended overwrites or type mismatches.
- Using labels precisely to avoid ambiguous or undefined jump targets, which could cause runtime failures and subroutine dispatch errors.

These factors contribute to the deterministic and predictable execution model that Vertigo provides, enabling developers to write maintainable and debuggable code.

GETTING STARTED: WRITING YOUR FIRST VERTIGO PROGRAM

This section walks you through creating a simple Vertigo program illustrating fundamental concepts such as stack creation, pushing values, performing arithmetic operations, using labels and jumps, and printing output. It also covers basic file structure, comment conventions, and how to run your program with the Vertigo interpreter.

STEP 1: CREATING THE SOURCE FILE (`.VTGO`)

Vertigo programs are saved as plain text files with the `.vtgo` extension. Each line typically contains one instruction and its arguments, and comments begin with a semicolon (`;`) and continue to the end of that line.

Begin by creating a new file named `hello.vtgo` in your preferred text editor.

STEP 2: WRITING THE PROGRAM

Enter the following code into `hello.vtgo`. It demonstrates basic stack operations, arithmetic, labels, jumps, and output:

```
; Create a new stack named 'main'
NEW main

; Select 'main' as the current stack
main

; Push two numeric values onto the stack
PUSH 10
PUSH 32

; Pop the top value (32) into the ODA register for output
POP ODA

; Add 10 and 32 and push the result onto the 'main' stack
MATH ADD & 10 32

; Duplicate the top stack value (42)
DUP

; Pop the duplicated value into ODA to prepare for output
POP ODA

; Label marking the loop start
POINT loop_start

; Push 1 onto the stack
PUSH 1
```

```
; Pop into register LIN0 (declared implicitly)
POP LIN0

; Subtract 1 from LIN0 and push result back on stack
MATH MINUS & LIN0 1

; Pop the result into LIN0 again
POP LIN0

; If LIN0 == 0, jump out of the loop
PUSH LIN0
PUSH 0
CMP @1 @0          ; Compare top two stack elements: LIN0
and 0
JUMPEQ loop_end

; Jump back to loop_start for next iteration
JUMP loop_start

POINT loop_end

; Push final string output to stack
PUSH "Done looping!"

; Pop string into ODA
POP ODA

; Trigger the interrupt to print ODA
INT 1
```

HOW THIS PROGRAM WORKS

- **Stack Creation and Selection:** The `NEW main` instruction creates a stack named `main`, and placing `main` alone on a line selects it as the current stack for subsequent operations.
- **PUSH and POP:** Values are pushed to the stack and popped into registers like `ODA`, which is typically used for output.
- **Arithmetic Operations:** The `MATH ADD` instruction adds two numbers and pushes the result back onto the stack.

- **Labels and Jumps:** The program defines two labels (`loop_start` and `loop_end`) used as jump targets for looping logic.
- **Comparisons and Conditional Jump:** The `CMP` instruction sets comparison flags for the conditional jump `JUMPEQ` , which exits the loop when the counter reaches zero.
- **Output:** The `INT 1` instruction triggers an internal interrupt that prints the content of `ODA` to the console.

STEP 3: RUNNING YOUR PROGRAM

To run a Vertigo program, invoke the interpreter from the command line, passing your `.vtgo` source file as the sole argument:

```
python vertigo_interpreter.py hello.vtgo
```

Replace `vertigo_interpreter.py` with the actual interpreter filename if different.

The interpreter reads the source, processes commands line-by-line, and uses an internal instruction pointer to track execution. Your program's output will appear in the terminal whenever values are pushed to `ODA` and printed via interrupts or explicit output calls.

STEP 4: UNDERSTANDING DUMP FILES (`.VTD`)

During or after execution, using the `DUMP` instruction allows you to inspect current stack states or write detailed logs to a dump file with a `.vtd` extension. This file contains serialized runtime data and logs valuable for debugging or analysis.

For example, adding the instruction:

```
DUMP LOGS
```

generates a timestamped dump file such as `dump-30242324.vtd` containing an execution trace useful during development.

ENCOURAGEMENT TO EXPERIMENT

To deepen your understanding of Vertigo, try modifying the example program:

- Change the numeric values pushed onto the stack.
- Add more arithmetic or logical operations, e.g., `MATH MUL` or `OPS AND`.
- Create and use new stacks with `NEW` and switch between them.
- Define labels and jump to different parts of the program conditionally.
- Use `IN` to accept user input via the terminal.
- Experiment with subroutines by adding `SUB`, `ENDSUB`, and `CALL` blocks.

Each experiment will help familiarize you with Vertigo's stack-based model, control flow, and register interactions. Refer back to earlier sections for detailed instruction syntax and semantics as you extend and refine your programs.