VERTIGO: A STACK-BASED PROGRAMMING LANGUAGE OVERVIEW

INTRODUCTION TO VERTIGO

Vertigo is an interpreted, stack-based programming language designed for low-level control and custom scripting. It uses **.vtgo** files to store source code and **.vtd** files for dump outputs, allowing developers to introspect the state of their programs during execution.

At its core, Vertigo operates with multiple named stacks and a set of registers to manage data efficiently. These stacks enable flexible manipulation of program data through commands like PUSH, POP, DUP, and SWAP. Registers provide a way to temporarily hold values such as computation results or input data.

Vertigo programs support labels and various control flow operations, including conditional and unconditional jumps (JUMP, JUMPEQ, etc.), loops, and subroutines that allow modular and reusable code segments. Subroutines can be defined and called dynamically, enabling separation of concerns within scripts.

The language offers robust arithmetic, comparison, and string operations alongside direct control over program execution flow. Its design favors explicit stack management and simple but powerful instructions, making it well-suited for educational purposes and scenarios requiring granular control of execution state.

FILE TYPES AND STRUCTURE

Vertigo primarily uses two file types: **.vtgo** source code files and **.vtd** dump files. The `.vtgo` files contain the program's instructions written in plain text. Each line can include a single command and optional arguments, with comments allowed by using a semicolon (`;`) to ignore the rest of the line. This makes Vertigo code easy to read and edit in any text editor.

Within `.vtgo` files, labels for control flow are defined using the `POINT` instruction. Labels serve as markers that commands like `JUMP` and conditional jumps use to navigate through code logically. This structure helps organize Vertigo programs with clear sections and flow control points.

The `.vtd` files are used for dumping the state of the program during execution—primarily the contents and status of stacks or logging output. These dump files are automatically generated with a timestamp in their filename to timestamp the moment of creation. This feature aids debugging and introspection by allowing programmers to review the exact program state or logs captured at a specific execution time.

CORE COMPONENTS: STACKS AND REGISTERS

Vertigo centers its operations around **multiple named stacks**, which are dynamic data structures that hold values in a last-in, first-out (LIFO) manner. You create a new stack using the `NEW` instruction followed by the stack's name. For example, `NEW myStack` initializes a fresh stack called `myStack`. To work with a specific stack, simply use its name as a standalone command line in the source code to select it as the current target for subsequent instructions.

Once a stack is selected, various commands manipulate its contents:

- `PUSH` adds an item onto the top of the stack.
- `POP` removes the top item from the stack and stores it in a register.
- `DUP` duplicates the top item on the stack, increasing the stack size by one.
- `SWAP` exchanges the positions of the top two stack items.
- `PICK` copies an item from deeper in the stack to the top without removing it.
- `CLEAR` empties the current stack completely.

Alongside stacks, Vertigo uses a set of **registers**, which are named storage locations holding single values. Key registers include:

- `ODA` : Often holds an output or intermediate value for printing.
- `IDA` : Typically used to store input values collected from the user.
- `CLI` and `LTM` : Track loop counters and limits for controlling program flow.

Registers facilitate temporary storage, intermediate computations, and control management, complementing stack operations by providing direct access to individual values.

Additionally, Vertigo supports **immutables**—named constants defined via the `IM` instruction. These hold fixed values that cannot be changed during execution, serving as reliable references within a program. Immutables help maintain clarity and prevent accidental modifications to critical constants.

PROGRAM FLOW AND CONTROL INSTRUCTIONS

Vertigo's program flow is managed through a set of control instructions that manipulate the instruction pointer, which determines the next command to execute in the program sequence. These instructions enable branching, looping, and modular code execution.

LABELS AND BRANCHING

The `POINT` instruction defines a label, marking a specific location in the code. Labels are essential for navigation and must be unique within a program. Once labels are set, various jump instructions allow the program to branch execution to these labeled lines:

- **JUMP** — unconditionally moves the instruction pointer to the given label.
- **JUMPEQ** — jumps if the last comparison flagged equality.
- **JUMPGT** — jumps if the last comparison showed the first operand was greater.
- **JUMPLT** — jumps if the first operand was less.
- **JUMPNEQ** — jumps if the last comparison indicated inequality.

These conditional jumps rely on flags set by the `CMP` instruction, which compares two values—numbers or strings—and updates the flags `equal`, `greater`, and `less`. This mechanism provides the foundation for decision-making and flow control in Vertigo programs.

LOOPS

Iteration is controlled using `LOOP` and `ENDLOOP` instructions, working together with the special registers `LTM` (loop limit) and `CLI` (current loop iteration). When a `LOOP` starts, the instruction pointer position is saved. At `ENDLOOP`, the loop counter increments and, if the limit has not been

reached, the instruction pointer jumps back to the start of the loop, effectively repeating the enclosed instructions. This enables executing blocks of code multiple times in a controlled manner.

SUBROUTINES

Vertigo supports modular and reusable code via subroutines. The `SUB` instruction defines a named subroutine, gathering a block of instructions until an `ENDSUB` marker. These subroutines can be invoked anywhere in the program using `CALL`, temporarily transferring execution to the subroutine's code. Upon completion, control returns to the instruction following the `CALL`, preserving the program's flow. This structure allows abstraction of repeated tasks and cleaner program organization.

Through these control flow instructions—labels, conditional jumps, loops, and subroutines—Vertigo provides flexible and explicit management of execution order. The instruction pointer movement is directly affected by these commands, making Vertigo a powerful language for creating structured and dynamic programs.

OPERATIONS AND DATA MANIPULATION

Vertigo offers a rich set of instructions for arithmetic, logical, and string operations that manipulate data within stacks and registers efficiently.

ARITHMETIC OPERATIONS

The `MATH` instruction supports fundamental arithmetic operations: `ADD`, `MINUS`, `MUL`, `DIV`, `MOD`, and `POW`. You specify the operation, a destination (either a register or the current stack denoted by `&`), and two numeric operands. For example:

```
MATH ADD ODA 5 10
```

This adds 5 and 10, storing the result (15) in the register `ODA`. Alternatively, results can be pushed directly onto the selected stack.

STRING OPERATIONS

Vertigo supports string manipulation through instructions like:

- `CONCAT` : Concatenates two strings and stores the result in a register.
- `STRLEN` : Measures the length of a string, storing the integer length in a register.
- `STRCMP` : Compares two strings, setting internal flags (`equal` , `greater` , `less`) used for conditional jumps.

LOGICAL AND COMPARISON OPERATIONS

The versatile `OPS` instruction family performs logical operations such as `AND` , `OR` , `NOT` , and comparison checks like `EQUAL` and `NEQUAL` . These operate on one or more operands, interpreting them logically (true/false as 1/0), then store results in registers or stacks. These logical tools enable programming decisions and flow control within Vertigo scripts.

INPUT AND OUTPUT

User interaction is handled with the `IN` instruction, prompting the user and saving input into the `IDA` register. Output is primarily produced by printing the `ODA` register, facilitated through the internal `printint` function, which writes the current `ODA` value to standard output when enabled.

SPECIAL INSTRUCTIONS

- **WAIT**: Pauses program execution for a specified time (measured in hundredths of a second), enabling timed delays.
- **DUMP**: Dumps the current state of stacks or logs to a `.vtd` file for debugging and state inspection purposes.
- **BRING** and **IMPORT**: Load external Vertigo subroutine libraries or Python modules, enhancing modularity and reusability of code.