Blue Nexus
.coding

Language Specification v0.2.a

Contents:

0. Quick summary

BlueNexus is a symbolic stack language whose active program surface is single-character commands. Only variables use multi-character bracketed blocks (`>x<...<`) and conditionals use `?[type]{cond}(cmd)`. Whitespace and commas are ignored by the interpreter and only exist to help the human read code.

Execution model: linear token stream → tokens identified by tokenizer → tokens executed on a stack machine with 52 named variables (`A–Z`, `a–z`).

Values on the stack are never popped implicitly; they are only removed explicitly by ▼ (Pop) or `↡[n]` (Pop N). Arithmetic and comparison operators read values without consuming them.

---

1. Data Model

- **Character encoding**: ASCII integer codes for all character data.
- **Stack** — The stack can only contain single values, which are either **Number** (the default) or an **ASCII Integer Code** (representing a single character). The concept of a "String List" only exists within named variables.
- **Variable slots** — 52 named slots: `A-Z`, `a-z`. Each variable stores:
  - **Number** (default)
  - **String List** (an ordered list of ASCII integer codes, e.g. `[72,101,108,108,111]`)

Types & Promotion

- A variable is promoted to a **String List** if it is assigned a "" literal or if a single ASCII Integer Code is appended to it via `.$` (dot append).
- The stack only ever holds **Number** or **ASCII Integer Code** values.

2. Tokenization rules (interpreter pre-step)

1. **Whitespace and commas are ignored.**
2. Token boundaries:
    - Single-char commands (e.g. ▲, +, #) are tokens.
    - Multi-char constructs are grouped by delimiters:
        - Variable block: `> < ... <` — e.g. `>x<...<` is one block token.
        - Conditional block: `?[type]{cond}(cmd)` — entire `?[...] {...} (...)` is one token.
        - Bracket matching is strict — missing closers produce BN-02.
3. The tokenizer outputs a token list annotated with **absolute character indices** (0-based) so each token carries a starting character position for error reporting and debug jump.

---

3. Core command set (backbone)

Use UTF-8 glyphs shown. Each command operates on the stack, variables, or I/O as described.

Stack / stack control

- ▲ Push — pushes:
    - literal value following ▲ (if used as ▲123 in a tokenized literal), or
    - a variable if immediately followed by a variable char in tokenized form (e.g. ▲x inside an execution context) — pushes the current **first element** of x if it's a string list, or the numeric value if x is number. Rotates the first element (which has been pushed) to the back of the string list if the variable is a string list.
    - ▲& waits for input and pushes user input (numeric if numeric, else 0).
- ▼ Pop — removes the top stack value. `▼[n]` or `↡[n]` recommended for bulk pop (see below).
- ⇅ Duplicate — duplicates the top-most value (pushes a copy).
- ↔ Swap — swaps top two stack values.
- `↡[n]` Pop N — pops n values (e.g. `↡[3]`). `↡[.]` clears whole stack.

Arithmetic / numeric

- `+` Add — reads top two values (without popping) and pushes numeric result.
- `-` Subtract / Append (contextual) — numeric: `second - top`. In variable blocks, used syntactically for `>x-5<` meaning `x = x - 5`.
- `×` Multiply — multiplies top two numbers, pushes result.
- `÷` Integer divide — `second / top` (integer division), pushes result.
- `%` Modulo — `second % top`, pushes remainder.

Note: arithmetic operators assume numeric operands. If operands are string lists or mixed, interpreter raises BN-03 (Type Mismatch).

Comparison / logic

- `>   Greater than`
- `<   Less than`
- `>=  Greater than or equal to`
- `<=  Less than or equal to`
- `=   Equal to`
- `!=  Not equal to`

I/O & printing

- `#` Print Number — Prints the top-of-stack value as a number (without popping).
- `$` Print Character — Prints the top-of-stack value after interpreting it as an **ASCII Integer Code** (without popping).
- **Behavior on Stack:** `$` and `#` read and print the top stack value based on the chosen format (`#` as number, `$` as character). Since the stack only contains single numerical values (either a number or a code), they always operate on the single value at the top. Both can also be used to print variables (e.g. `#x` or `%x`)
- `¦` New line — prints newline to console.

Variable & string operations (multi-char blocks & simple forms)

- Variable block general form:
  - `>x< ... <` — execute `...` in a context targeting `x`. The **result** of executing `...` (a single value or list) becomes the new value of `x`. The block can contain any BlueNexus tokens.

- Common variable forms:
    - `>x<value<` — set `x` to `value` (literal number or `"<string>"` literal).
    - `>x<~<` — set `x` to the top-of-stack value (represent `~` inside block).
    - `>x<&<` — wait for user input; if numeric input => `x` = number; else => `x` = `0`.
    - `>x<."text"<` — append a string literal to `x` as string elements; if `x` was numeric it is promoted to a string list. Example: `>x<."HELLO"<`.
    - `>x<.value<` — append numeric literal (as string pieces or as numeric append? see numeric append below).
- Indexed/list operations:
    - `$x` — print the **first element** (ASCII code → printed char) of string-list variable `x`, then **rotate** that element to the back of `x` (FIFO rotate). If x is numeric, `x` prints the `ASCII character` corresponding to the numeric value of x (e.g., x=65 prints 'A'). It does not rotate.
    - `$.x` — **invisible rotate**: moves the first element to the back without printing.
    - `▼[x]` — remove the first element of list variable `x`.
    - `▲x` — push first element of x to the stack (and rotates the element to the back of x if x is a string list).

Input placeholder

- `&` — reserved solely as *user input placeholder*. Only used where grammar allows input (e.g. `▲&`, `>x<&<`). `&` never appears inside a literal quote as data unless quoted `"&"` is used in a string literal (then stored as ASCII 38).

Misc

- `_` namespace — `_stack`, `_var=x`, etc. Debug/inspection commands (see §8).

---

4. Variable append rules (explicit)

- `>x<.&<` — **numeric-append-from-input**:
  - If `x` is numeric and input is numeric: append digits (concatenate decimal representation) — `x=12` + `&=34` → `x=1234`.
  - If `x` is a string-list: append the input as **string elements** (convert input to characters).
  - If `x` numeric and input **not** numeric: **do nothing** (emit BN-03 or BN-05 per policy).
- `>x<."value"<` — append the literal `value` as string list entries (always string behavior). Example: `>x<."HELLO"<`.

Variable block general forms:

1. Expression Form: `>x...<` (e.g., >x-5<, >x+1<)
   The content immediately following `>x` is parsed as a direct mutation expression. This is syntactic sugar for common arithmetic and list operations, where x is implicitly the first operand.
   a. **Example:** `>x-5<` is an atomic token meaning `x = x - 5`.

2. Assignment/Execution Form: `>x<...<` (e.g., >x<5<, >x<."text"<)
   The content between `>x<` and the final `<` is tokenized and executed as a sub-program. The final result of this sub-program (a single value or a string list) becomes the new value of `x`.
   a. **Example:** `>x<~<` executes the token `~` (top-of-stack value) and assigns that value to `x`.

5. Conditionals & flow

Conditionals use the precise syntax:

```
?[ !type ]{ condition }( command )
```

- `?` begins a control token.
- `[!type]` selects the flow form. Known types:
  - `!?` — IF (single check; if condition is true execute command)
  - `!∞` — WHILE (evaluate `{condition}`, if condition is true execute `(command)`, then re-evaluate; repeat)
  - `!∑` — FOR (iterates over a range — see example)
- `{ condition }` — These operators are only valid **inside conditional functions** (`?[type]{condition}(command)`).
  They evaluate to **true** or **false** within the condition.
  They **do not** alter the stack — only determine whether the following command block is executed.
- `( command )` — a sequence of BlueNexus statements executed when the flow triggers. This may alter the stack.
- You can have an IF, optionally followed by `??`.
  Only **IF** conditionals (`[!?]`) can be chained this way — **not** FOR (`[!∑]`) or WHILE (`[!∞]`).

  ```
  ?[!?]{condition}(command)??{condition}(command)??(command)
  ```

- `??` in a conditional **NOT** followed by a `{condition}` acts as an **ELSE**.

Examples

If (print 1 when top ≠ 0):

```
?[!?]{~!=0}(▲1#)
```

While (increment `a` until 54):

```
>a<34<?[!∞]{a<54}(>a+1<)
```

- Note: `a<54` inside condition is an expression that executes `(>a+1<)` if `a` < 54.

For-each characters in g:

```
>g<"Hello world!"<?[!∑]{"*g}(▲g$)
```

- (Here `*g` denotes within variable g, and “ denotes for every string in the string list. each loop pushes the first element of g onto the stack (and moves it to the back of variable g string list), then `(▲g$)` prints it.)

---

6. Parsing/semantics details (how the interpreter should act)

- Every token has an **origin char index**; on any error the interpreter must report `[BN-XX]` with that char index and highlight it.
- The interpreter strips whitespace, line breaks and commas **before** tokenization.
- Variable blocks `>x<...<` must be recognized atomically; their inner `...` is tokenized and executed in a *sub-context* where:
  - `x` is the active target,
  - ~ maps to the current top-of-stack (without popping),
  - The final **result** of the block (single number or string-list) becomes `x`.
- Condition tokens `?...` are recognized atomically; their inner blocks are tokenized and executed as subprograms.
- Single-character commands executed in order outside these grouped tokens.

---

7. Error codes (character indexed)

Errors are non-fatal by default; they are logged in the runtime console with the form:

```
[BN-XX] <Short Name> @ char <index>: <human message>
```

Code	Short	When triggered
BN-00	OK	No error.
BN-01	Undefined Variable	Accessed variable that is undefined.
BN-02	Syntax Error	Tokenizer found unbalanced or missing delimiter.
BN-03	Type Mismatch	Operation incompatible with variable/operand type.
BN-04	Value Overflow	Numeric append or arithmetic exceeds allowable bounds.
BN-05	Null Input	`>x<.&<` with non-numeric input when `x` is numeric (no-op; optionally BN-05).
BN-06	Unknown Command	Token not recognized.
BN-07	Stack Overflow	Too many stack items or recursion depth exceeded.
BN-08	Runtime Halt	Critical failure that stops execution.
BN-09	Logic Fault	Internal interpreter inconsistency.

Diagnostic features

- On BN errors the console must show the offending char index and **highlight** the token in the source viewer.
- **Jump-to-char**: Developer tool accepts a number `N` and jumps the source cursor to char index `N`. Useful since BlueNexus is position-based rather than line-based.

8. Debug commands

`_stack` — opens a scrollable window showing stack contents with the top at the bottom; values displayed as one-per-line vertically:

```
 54
65
98
34
12
```

- `_var=x` — opens a window titled `VARIABLE: x`. If `x` is numeric: show single number. If string list: show bracketed array of codes or optionally decoded string on hover.

---

9. Examples

Hello world (print each char of a variable g)

```
>g<"Hello!"<?[!∑]{"*g}(▲g$)
```

Explanation:

- `>g<"Hello!"<` sets `g` to `[72,101,108,108,111,33]`
- `?[!∑]{...}(...)` iterates each character; `(▲g$)` pushes first element of `g` then prints it and rotates it to the back.

While increment a from any number (34 in example) to 54

```
>a<34<?[!∞]{a<54}(>a+1<)
```

- `>a<34<` set `a` to 34.
- `?[!∞]{a<54}(... )` while `a < 54`, run `>a+1<` (add 1 to a).

Input numeric append

```
>a<12<
>x<&<     // waits for numeric input; x=enteredNumber or 0
>a<.&<    // if a was numeric and input numeric, a= concat digits
```

Rotating print vs invisible rotate

```
$g    // print first element of g and move it to back
$.g   // move first element to back WITHOUT printing
▼[g]  // remove first element of g
```

---

10. Implementation notes (for the interpreter - short)

- Implement tokenizer to:
  - Remove whitespace & commas.
  - Walk characters left→right, grouping `>x<...<` and `?[...]{...}(...  )` as atomic tokens. Record starting char index per token.
- Execution loop:
  - For each token:
    - Dispatch by token type (single-char → command; grouped token → sub-execution).
    - Provide sub-executions with a sandbox stack/view: variable blocks must return at most one value (or a list).
    - On error, abort token execution, log BN code with token char index, continue or stop per policy.
- Stack semantics: no implicit pops; use explicit `▼` / `↡` for mutation.

---

11. Style & usage rules

- **Commas and spaces** are visually helpful; **ignored by tokenizer**.
- **Quotes**: `"..."` denote string literals. If you want to store an actual `"` char, include it in the literal as `""` (double `"` inside literal).
- **`&` only** stands for user input when used in input positions like `▲&` or `>x<&<`. It is not a general-purpose placeholder elsewhere.

---

12. Appendix — Quick command cheat sheet

- Single-character core (most-used):

    - ▲ Push (var, literal, or ▲& input)
    - ▼ Pop top
    - ⇅ Duplicate top
    - ↔ Swap top two
    - ↡[n] Pop n / ↡[.] clear stack
    - + add
    - - subtract
    - × multiply
    - ÷ integer divide
    - % modulo
    - # print number (top) (or #x prints value of x)
    - $ print char (top) (or $x prints first char of x and rotate)
    - $. invisible rotate (rotate list first->back without printing)
    - ¦ newline
    - & user input placeholder (only in input contexts)
    - >x<...< variable block (set x to result)
    - ?[!type]{cond}(cmd) conditionals (IF/WHILE/FOR types)
    - _... debug namespace (_stack, _var=x)