Posix-Nexus AWK

Canine-Table

April 8, 2025

Contents

I Getting Started

I Variable Naming Convention

I Getting Started

In this implementation, variable names follow a structured format based on their type and scope to minimize contention and maximize readability:

- **V**: Vector
- **D**: Data
- **S**: Separator
- **O**: Output Separator
- **B**: Boolean
- **N**: Number

This naming convention ensures that variables are intuitive to identify and minimizes ambiguity in complex functions. Users can infer the type and role of each variable from its name without inspecting the underlying code.

II Miscellaneous

II Miscellaneous

The following functions provide generic utilities for handling values and conditional operations, offering flexible solutions for various logical scenarios.

- **__nx_quote_map(V)**: Initializes an associative array (**V**) with mappings for common quote characters (double quotes, single quotes, and backticks). Useful for handling quotes in parsing, escaping, or validation tasks.

- **__nx_bracket_map(V)**: Initializes an associative array (**V**) with mappings for common opening and closing brackets (e.g., `[` to `]`, `{` to `}`, and `(` to `)`). Useful for parsing, validation, and other operations involving balanced brackets.

- **__nx_str_map(V)**: Initializes an associative array (**V**) with mappings for string character classes. These include ranges for uppercase and lowercase letters, digits, hexadecimal characters, alphanumeric characters, printable ASCII characters, and punctuation. Useful for validation, parsing, and string processing.

- **__nx_escape_map(V)**: Initializes an associative array (**V**) with mappings for common escape sequences (`\x20`, `\x09`, `\x0a`, `\x0b`, `\x0c`), each assigned to an empty string. Useful for removing whitespace and control characters during string processing.

- **__nx_defined(D, B)**: Validates whether **D** is defined or evaluates as truthy under additional constraints provided by **B**.

- **__nx_null(D)**: Checks whether **D** is null or has a length of zero.

- **__nx_else(D1, D2, B)**: Returns **D1** if it is truthy or satisfies the condition set by **B**, otherwise returns **D2**.

- **__nx_if(B1, D1, D2, B2)**: Returns **D1** if **B1** is truthy or satisfies the condition set by **B2**, otherwise returns **D2**.

- **__nx_elif(B1, B2, B3, B4, B5, B6)**: Evaluates multiple conditions and their relationships, returning a boolean value based on comparisons of the results of **__nx_defined** for the provided inputs.

- **__nx_or(B1, B2, B3, B4, B5, B6)**: Evaluates logical OR conditions between multiple inputs, incorporating constraints applied via **__nx_defined** and fallback adjustments from **__nx_else**.

- **__nx_xor(B1, B2, B3, B4)**: Evaluates exclusive OR (XOR) conditions between multiple inputs, incorporating constraints applied via **__nx_defined** and fallback adjustments from **__nx_else**.

II Miscellaneous

- **__nx_compare(B1, B2, B3, B4)**: Compares two inputs based on type, length, or specified comparison rules, leveraging **awk**'s dynamic behavior and optional constraints.
- **__nx_equality(B1, B2, B3)**: Evaluates the equality or relational conditions between **B1** and **B3** based on the operator specified in **B2**, leveraging **awk** behavior for dynamic comparisons.
- **__nx_swap(V, D1, D2)**: Swaps the values of two specified indices in the provided array or associative array. Utilizes a temporary variable to ensure the operation is safe and lossless.

II __nx_quote_map

__nx_quote_map(V)

```
function __nx_quote_map(V) {
    V["\""] = "\""
    V["'"] = "'"
    V["`"] = "`"
}
```

function __nx_quote_map(V) V["̈"] = "̈" V["’"] = "’" V["‘"] = "‘"

II __nx_quote_map

Initializes an associative array (**V**) with common quote characters (double quotes, single quotes, and backticks), mapping each to itself. This is useful for handling quotes in parsing, escaping, or validation tasks.

- **V**: An associative array passed by reference. After execution, it contains mappings for the following characters:
  - " (double quote)
  - `’ (single quote)`
  - `‘ (backtick)`

II __nx_bracket_map

__nx_bracket_map(V)

```
function __nx_bracket_map(V) {
    V["\x5b"] = "\x5d"
    V["\x7b"] = "\x7d"
    V["\x28"] = "\x29"
}
```

function __nx_bracket_map(V) V["5b"] = "5d" V["7b"] = "7d" V["28"] = "29"

II __nx_bracket_map

Initializes an associative array (V) with mappings for common bracket characters. Each opening bracket is mapped to its corresponding closing bracket, using their ASCII hexadecimal representations.

V: An associative array passed by reference. After execution, it contains the following mappings:

- \x5b (ASCII for [) maps to \x5d (ASCII for]).
- \x7b (ASCII for {) maps to \x7d (ASCII for }).
- \x28 (ASCII for () maps to \x29 (ASCII for)).

II __nx_str_map

__nx_str_map(V)

```
function __nx_str_map(V) {
    V["upper"] = "A-Z"
    V["lower"] = "a-z"
    V["xupper"] = "A-F"
    V["xlower"] = "a-f"
    V["digit"] = "0-9"
    V["alpha"] = V["upper"] V["lower"]
    V["xdigit"] = V["digit"] V["xupper"] V["xlower"]
    V["alnum"] = V["digit"] V["alpha"]
    V["print"] = "\x20-\x7e"
    V["punct"] = "\x21-\x2f\x3a-\x40\x5b-\x60\x7b-\x7e"
}
```

```
function __nx_str_map(V)  V["upper"] = "A-Z" V["lower"] = "a-z" V["xupper"] = "A-F" V["xlower"] = "a-f" V["digit"] = "0-9" V["alpha"] = V["upper"] V["lower"] V["xdigit"] = V["digit"] V["xupper"] V["xlower"] V["alnum"] = V["digit"] V["alpha"] V["print"] = "20-7e" V["punct"] = "21-2f3a-405b-607b-7e"
```

II __nx_str_map

Initializes an associative array (**V**) with mappings for string character classes. These mappings include ranges for uppercase letters, lowercase letters, digits, hexadecimal characters, printable characters, and punctuation.

- **V**: An associative array passed by reference. After execution, it contains the following mappings:
  - **"upper"**: Maps to `"A-Z"` (uppercase English letters).
  - **"lower"**: Maps to `"a-z"` (lowercase English letters).
  - **"xupper"**: Maps to `"A-F"` (uppercase hexadecimal characters).
  - **"xlower"**: Maps to `"a-f"` (lowercase hexadecimal characters).
  - **"digit"**: Maps to `"0-9"` (numerical digits).
  - **"alpha"**: Concatenation of **"upper"** and **"lower"**; maps to `"A-Za-z"` (alphabetical characters).
  - **"xdigit"**: Concatenation of **"digit"**, **"xupper"**, and **"xlower"**; maps to `"0-9A-Fa-f"` (hexadecimal digits).
  - **"alnum"**: Concatenation of **"digit"** and **"alpha"**; maps to `"0-9A-Za-z"` (alphanumeric characters).
  - **"print"**: Maps to `"\x20-\x7e"` (printable ASCII characters).
  - **"punct"**: Maps to `"\x21-\x2f\x3a-\x40\x5b-\x60\x7b-\x7e"` (punctuation characters within printable ASCII).

II __nx_escape_map

__nx_escape_map(V)

```
function __nx_escape_map(V) {
    V["\x20"] = ""
    V["\x09"] = ""
    V["\x0a"] = ""
    V["\x0b"] = ""
    V["\x0c"] = ""
}
```

function __nx_escape_map(V) V["20"] = "" V["09"] = "" V["0a"] = "" V["0b"] = "" V["0c"] = ""

II __nx_escape_map

Initializes an associative array (V) with mappings for common escape sequences, assigning each escape sequence to an empty string. This is useful for processing or sanitizing strings by removing whitespace and control characters.

- **V**: An associative array passed by reference. After execution, it contains the following mappings:
  - `\x20`: Maps to an empty string (ASCII for space).
  - `\x09`: Maps to an empty string (ASCII for tab).
  - `\x0a`: Maps to an empty string (ASCII for newline).
  - `\x0b`: Maps to an empty string (ASCII for vertical tab).
  - `\x0c`: Maps to an empty string (ASCII for form feed).

II __nx_defined

__nx_defined(D, B)

```
function __nx_defined(D, B) {
    return (D || (length(D) && B))
}
```

function __nx_defined(D, B) return (D || (length(D) B))

II __nx_defined

Determines whether **D** is defined or evaluates to a truthy value, optionally constrained by **B**. Returns a boolean value accordingly.

- **D**: The variable or value to check for definition or truthiness.
- **B**: An optional additional condition for validation when **D** has length.

Basic truth check

```
B1 = 1
B2 = ""
result = __nx_defined(B1, B2)
% result is true, as B1 is true (1) regardless of B2
```

B1 = 1 B2 = "" result = __nx_defined(B1, B2)

Empty string check

```
B1 = ""
B2 = 1
result = __nx_defined(B1, B2)
% result is false, as B1 is an empty string, which fails the defined check
```

B1 = "" B2 = 1 result = __nx_defined(B1, B2)

String with length check

```
B1 = "hello"
B2 = 0
result = __nx_defined(B1, B2)
```

```
% result is true, as B1 ("hello") is non-empty and therefore defined
```

B1 = "hello" B2 = 0 result = __nx_defined(B1, B2)

Numeric length check

```
B1 = 0
B2 = 1
result = __nx_defined(B1, B2)
% result is true, as B1 (0) has a length, even though it evaluates as false in
↪conditions
```

B1 = 0 B2 = 1 result = __nx_defined(B1, B2)

Combined truth and fallback check

```
B1 = ""
B2 = "fallback"
result = __nx_defined(B1, B2)
% result is true, as B2 ("fallback") is defined and compensates for B1 being
↪empty
```

B1 = "" B2 = "fallback" result = __nx_defined(B1, B2)

II __nx_null

__nx_null(D)

```
function __nx_null(D) {
    return (length(D) == 0)
}
```

function __nx_null(D) return (length(D) == 0)

II __nx_null

Determines whether **D** is null or has a length of zero. This function is useful for validating empty values or strings.

D: The variable or value to check for nullity or zero length.

Null check on an empty value

```
B1 = ""
result = __nx_null(B1)
% result is true, as B1 is an empty string and thus considered null
```

B1 = "" result = __nx_null(B1)

Null check on a numeric zero

```
B1 = 0
result = __nx_null(B1)
% result is false, as B1 (0) is not considered null despite evaluating as false
↪in conditions
```

B1 = 0 result = __nx_null(B1)

Null check on an undefined value

```
B1 = ""
B2 = 1
result = __nx_null(__nx_else(B1, B2))
% result is false, as the fallback B2 (1) is defined, overriding B1's null
↪state
```

B1 = "" B2 = 1 result = __nx_null(__nx_else(B1, B2))

Null check with regex pattern fallback

```
B1 = ""
B2 = "abc123"
B3 = /^[a-z]+[0-9]+$/
result = __nx_null(__nx_else(B1, B2 ~ B3))
% result is false, as B2 matches the regex pattern, making it non-null
```

B1 = "" B2 = "abc123" B3 = /$^{[}a - z] + [0 - 9]+$/ result = __nx_null(__nx_else(B1, B2   B3))

Handling null explicitly in logical flows

```
B1 = ""
B2 = ""
B3 = 0
result = __nx_null(__nx_or(B1, B2, B3))
% result is true, as all inputs to __nx_or resolve to null-like states
```

B1 = "" B2 = "" B3 = 0 result = __nx_null(__nx_or(B1, B2, B3))

II __nx_else

__nx_else(D1, D2, B)

```
function __nx_else(D1, D2, B) {
    if (D1 || __nx_defined(D1, B))
        return D1
    return D2
}
```

function __nx_else(D1, D2, B) if (D1 || __nx_defined(D1, B)) return D1 return D2

II __nx_else

Returns **D1** if it is truthy or satisfies the condition set by **B**. If neither condition is met, **D2** is returned.

- **D1**: The primary value to evaluate and potentially return.
- **D2**: The fallback value returned if **D1** does not meet the conditions.
- **B**: An optional constraint applied to **D1** using **__nx_defined**.

Simple fallback adjustment

```
B1 = 1
B2 = 0
result = __nx_else(B1, B2)
% result is true, as B1 is true (1), overriding the fallback condition of B2
↪(0)
```

B1 = 1 B2 = 0 result = __nx_else(B1, B2)

Fallback with string input

```
B1 = "abc"
B2 = ""
result = __nx_else(B1, B2)
% result is true, as B1 ("abc") is valid and overrides the empty fallback (B2 =
↪"")
```

B1 = "abc" B2 = "" result = __nx_else(B1, B2)

Numeric fallback adjustment

```
B1 = ""
B2 = 42
result = __nx_else(B1, B2)
% result is true, as B1 fails the condition, falling back to B2 (42)
```

B1 = "" B2 = 42 result = __nx_else(B1, B2)

Fallback with pattern matching

```
B1 = "[a-z]+"
B2 = "hello"
result = __nx_else(B2 ~ B1, 0)
% result is true, as the pattern "[a-z]+" matches B2 ("hello"), overriding the fallback (0)
```

B1 = "[a-z]+" B2 = "hello" result = __nx_else(B2 B1, 0)

II __nx_if

__nx_if(B1, D1, D2, B2)

```
function __nx_if(B1, D1, D2, B2) {
    if (B1 || __nx_defined(B1, B2))
        return D1
    return D2
}
```

function __nx_if(B1, D1, D2, B2) if (B1 || __nx_defined(B1, B2)) return D1 return D2

II __nx_if

Returns **D1** if **B1** is truthy or satisfies the condition set by **B2**. If neither condition is met, **D2** is returned. This function extends conditional operations by integrating the **__nx_defined** utility.

- **B1**: The primary condition to evaluate for truthiness.
- **D1**: The value returned if **B1** meets the conditions.
- **D2**: The fallback value returned if **B1** does not satisfy the conditions.
- **B2**: An optional additional constraint applied to **B1** using **__nx_defined**.

Basic conditional check

```
B1 = 1
B2 = "True Case"
B3 = "False Case"
result = __nx_if(B1, B2, B3)
% result is "True Case", as B1 is true (1), returning the second argument
```

B1 = 1 B2 = "True Case" B3 = "False Case" result = __nx_if(B1, B2, B3)

Evaluating string-based condition

```
B1 = "non-empty"
B2 = "Condition Met"
B3 = "Condition Not Met"
result = __nx_if(B1, B2, B3)
% result is "Condition Met", as B1 is non-empty and therefore true, returning
↪the second argument
```

B1 = "non-empty" B2 = "Condition Met" B3 = "Condition Not Met" result = __nx_if(B1, B2, B3)

Numeric comparison in conditional check

```
B1 = (5 > 3)
B2 = "Greater"
B3 = "Lesser or Equal"
result = __nx_if(B1, B2, B3)
% result is "Greater", as B1 evaluates to true (5 > 3)
```

B1 = (5 > 3) B2 = "Greater" B3 = "Lesser or Equal" result = __nx_if(B1, B2, B3)

Regex-based condition

```
B1 = ("abc123" ~ /^[a-z]+[0-9]+$/)
B2 = "Pattern Matches"
B3 = "Pattern Doesn't Match"
result = __nx_if(B1, B2, B3)
% result is "Pattern Matches", as B1 evaluates to true due to the regex match
```

B1 = ("abc123" $/^[a - z] + [0 - 9]+/$) B2 = "Pattern Matches" B3 = "Pattern Doesn't Match" result = __nx_if(B1, B2, B3)

Fallback when condition is false

```
B1 = 0
B2 = "Will Not Return"
B3 = "Fallback Case"
result = __nx_if(B1, B2, B3)
% result is "Fallback Case", as B1 is false (0), returning the third argument
```

B1 = 0 B2 = "Will Not Return" B3 = "Fallback Case" result = __nx_if(B1, B2, B3)

II __nx_elif

__nx_elif(B1, B2, B3, B4, B5, B6)

```
function __nx_elif(B1, B2, B3, B4, B5, B6) {
    if (B4) {
        B5 = __nx_else(B5, B4)
        B6 = __nx_else(B6, B5)
    }
    return (__nx_defined(B1, B4) == __nx_defined(B2, B5) && __nx_defined(B3,
↪B6) != __nx_defined(B1, B4))
}
```

function __nx_elif(B1, B2, B3, B4, B5, B6) if (B4) B5 = __nx_else(B5, B4) B6 = __nx_else(B6, B5) return (__nx_defined(B1, B4) == __nx_defined(B2, B5) __nx_defined(B3, B6) != __nx_defined(B1, B4))

II __nx_elif

Evaluates multiple conditions and their relationships. Returns a boolean value based on comparisons of the outputs from **__nx_defined** applied to the provided inputs.

- **B1**: The first condition to validate using **__nx_defined**.
- **B2**: The second condition to validate using **__nx_defined**.
- **B3**: The third condition to validate using **__nx_defined**.
- **B4**: Optional constraint applied to subsequent conditions **B5** and **B6**.
- **B5**: Adjusted condition based on **B4** if provided, otherwise unchanged.
- **B6**: Adjusted condition based on **B5** if provided, otherwise unchanged.

Simple relational checks

```
B1 = 1
B2 = 2
B3 = 3
B4 = 0
B5 = 1
B6 = 0
result = __nx_elif(B1, B2, B3, B4, B5, B6)
% result is false, as the comparisons of __nx_defined(B1, B4), __nx_defined(B2,
↪B5), and __nx_defined(B3, B6)
% do not satisfy the logic for XOR relationships
```

B1 = 1 B2 = 2 B3 = 3 B4 = 0 B5 = 1 B6 = 0 result = __nx_elif(B1, B2, B3, B4, B5, B6)

Pattern matching logic

```
B1 = "hello"
B2 = "world"
B3 = "[a-z]+"
B4 = 0
B5 = ""
B6 = "_"
result = __nx_elif(B1, B2, B3, B4, B5, B6)
% result is false, as __nx_defined(B3, B6) evaluates to true with pattern
↪"[a-z]+",
% but the fallback adjustments for B1 and B2 overlap in truth, violating XOR
↪logic
```

B1 = "hello" B2 = "world" B3 = "[a-z]+" B4 = 0 B5 = "" B6 = "_" result = __nx_elif(B1, B2, B3, B4, B5, B6)

Complex nested XOR conditions

```
B1 = 10
B2 = 20
B3 = 30
B4 = ""
B5 = "a"
B6 = "i"
result = __nx_elif(B1, B2, B3, B4, B5, B6)
% result is false, as none of the relationships between B1, B2, and B3 fulfill
↪the XOR conditions
% after fallback adjustments with __nx_else
```

B1 = 10 B2 = 20 B3 = 30 B4 = "" B5 = "a" B6 = "i" result = __nx_elif(B1, B2, B3, B4, B5, B6)

Nested condition adjustments

```
B1 = "abc"
B2 = "abc"
B3 = ""
B4 = "a"
B5 = "def"
B6 = ""
result = __nx_elif(B1, B2, B3, B4, B5, B6)
% result is true, as __nx_defined(B1, B4) and __nx_defined(B2, B5) are true,
% and the adjusted relationships satisfy the condition logic
```

B1 = "abc" B2 = "abc" B3 = "" B4 = "a" B5 = "def" B6 = "" result = __nx_elif(B1, B2, B3, B4, B5, B6)

II __nx_or

__nx_or(B1, B2, B3, B4, B5, B6)

```
function __nx_or(B1, B2, B3, B4, B5, B6) {
    if (B4) {
        B5 = __nx_else(B5, B4)
        B6 = __nx_else(B6, B5)
    }
    return ((__nx_defined(B1, B4) && __nx_defined(B2, B5)) || (__nx_defined(B3,
↪B6) && ! __nx_defined(B1, B4)))
}
```

function __nx_or(B1, B2, B3, B4, B5, B6) if (B4) B5 = __nx_else(B5, B4) B6 = __nx_else(B6, B5) return ((__nx_defined(B1, B4) __nx_defined(B2, B5)) || (__nx_defined(B3, B6) ! __nx_defined(B1, B4)))

^ II __nx_or

Evaluates logical OR conditions between multiple inputs. Uses **__nx_defined** to validate conditions and applies fallback adjustments using **__nx_else** for specific inputs.

- **B1**: The first condition to evaluate using **__nx_defined**.
- **B2**: The second condition to evaluate using **__nx_defined**.
- **B3**: The third condition to evaluate using **__nx_defined**.
- **B4**: Optional constraint applied to conditions and used in fallback adjustments via **__nx_else**.
- **B5**: Adjusted condition based on **B4** if provided, otherwise unchanged.
- **B6**: Adjusted condition based on **B5** if provided, otherwise unchanged.

OR condition for integer validation

```
B = 1
N = "-123"
result = __nx_or(B, N ~ /^([-]|[+])?[0-9]+$/, N ~ /^[0-9]+$/)
% result is true, as N matches the first regex pattern for signed integers
↪(-123)
```

B = 1 N = "-123" result = __nx_or(B, N /^([−]|[+])?[0 − 9]+/, N /^[0 − 9]+/)

OR condition for decimal number validation

```
B = 0
N = "123.45"
result = __nx_or(B, N ~ /^([-]|[+])?[0-9]+[.][0-9]+$/, N ~ /^[0-9]+[.][0-9]+$/)
% result is true, as N matches the first regex pattern for signed decimals
↪(123.45)
```

B = 0 N = "123.45" result = __nx_or(B, N $/^{([-]|[+])?[0-9]+[.][0-9]+/}$, N $/^{[0-9]+[.][0-9]+/})$

II __nx_xor

__nx_xor(B1, B2, B3, B4)

```
function __nx_xor(B1, B2, B3, B4) {
    if (B3)
        B4 = __nx_else(B4, B3)
    return ((! __nx_defined(B2, B4) && __nx_defined(B1, B3)) ||
↪(__nx_defined(B2, B4) && ! __nx_defined(B1, B3)))
}
```

function __nx_xor(B1, B2, B3, B4) if (B3) B4 = __nx_else(B4, B3) return ((! __nx_defined(B2, B4) __nx_defined(B1, B3)) || (__nx_defined(B2, B4) ! __nx_defined(B1, B3)))

^ II __nx_xor

Evaluates exclusive OR (XOR) conditions between multiple inputs. Uses **__nx_defined** to validate conditions and applies fallback adjustments using **__nx_else** for specific inputs.

- **B1**: The first condition to evaluate using **__nx_defined**.
- **B2**: The second condition to evaluate using **__nx_defined**.
- **B3**: Optional condition used for fallback adjustments via **__nx_else**.
- **B4**: Adjusted condition based on **B3** if provided, otherwise unchanged.

Basic XOR: Compare B1 and B2

```
B1 = 1
B2 = 0
B3 = ""
B4 = ""
result = __nx_xor(B1, B2, B3, B4)
% result is true, as B1 is true (1) and B2 is false (0), making XOR true
```

B1 = 1 B2 = 0 B3 = "" B4 = "" result = __nx_xor(B1, B2, B3, B4)

Complex XOR with fallback adjustment

```
B1 = 0
B2 = 0
B3 = ""
B4 = "_"
```

```
result = __nx_xor(B1, B2, B3, B4)
% result is true, as B2 satisfies the fallback condition (B4),
% while B1 is false, fulfilling XOR
```

B1 = 0 B2 = 0 B3 = "" B4 = "_" result = __nx_xor(B1, B2, B3, B4)

XOR with both conditions as true

```
B1 = 10
B2 = 20
B3 = 1
B4 = 1
result = __nx_xor(B1, B2, B3, B4)
% result is false, as B1 and B2 both satisfy their respective truth and length
↪conditions,
% violating XOR
```

B1 = 10 B2 = 20 B3 = 1 B4 = 1 result = __nx_xor(B1, B2, B3, B4)

String XOR with adjusted truth conditions

```
B1 = ""
B2 = "def"
B3 = 1
B4 = "a"
result = __nx_xor(B1, B2, B3, B4)
% result is true, as B1 ("") fails the truth check required by B3,
% while B2 ("def") satisfies the length requirement via B4, fulfilling XOR
```

B1 = "" B2 = "def" B3 = 1 B4 = "a" result = __nx_xor(B1, B2, B3, B4)

II __nx_compare

__nx_compare(B1, B2, B3, B4)

```
function __nx_compare(B1, B2, B3, B4) {
    if (! B3) {
        if (length(B3)) {
            B1 = length(B1)
            B2 = length(B2)
            B3 = 1
        } else if (__nx_is_digit(B1, 1) && __nx_is_digit(B2, 1)) {
            B1 = +B1
            B2 = +B2
            B3 = 1
        } else {
            B1 = "a" B1
            B2 = "a" B2
            B3 = 1
        }
    }

    if (B4) {
        return __nx_if(__nx_is_digit(B4), B1 > B2, B1 < B2) || __nx_if(__nx_else(B4 == 1, tolower(B4) == "i"), B1 == B2, 0)
    } else if (length(B4)) {
        return B1 ~ B2
    } else {
        return B1 == B2
    }
}
```

function __nx_compare(B1, B2, B3, B4) if (! B3) if (length(B3)) B1 = length(B1) B2 = length(B2) B3 = 1 else if (__nx_is_digit(B1, 1) __nx_is_digit(B2, 1)) B1 = +B1 B2 = +B2 B3 = 1 else B1 = "a" B1 B2 = "a" B2 B3 = 1
if (B4) return __nx_if(__nx_is_digit(B4), B1 > B2, B1 < B2) || __nx_if(__nx_else(B4 == 1, tolower(B4) == "i"), B1 == B2, 0) else if (length(B4)) return B1   B2 else return B1 == B2

II __nx_compare

Dynamically compares two inputs based on their type, value, and specified behavior. The function leverages **awk**'s dynamic capabilities, adjusting input values and logic based on context. Its flexibility allows for comparisons of numeric values, strings, lengths, or patterns.

- **B1**: The first input to compare.
- **B2**: The second input to compare.
- **B3**: Determines how inputs are normalized for comparison:
  - **1**: Inputs are treated as numeric values.
  - **""**: Inputs are compared as strings.
  - **0**: Inputs engage regex-based comparison.
- **B4**: Specifies the comparison rule. When **B4** is:
  - **"i"**: Performs >= (greater than or equal to).
  - **"a"**: Performs > (greater than) only, as it fails the second logic check.
  - **"1"**: If numeric, performs <= (less than or equal to).
  - **numeric but not "1"**: Performs < (less than).
  - **""**: Engages strict equality comparison (==).
  - **"0"**: Activates pattern matching (~).

Compare lengths of B1 and B2

```
B1 = "hello"
B2 = "world!"
B3 = 1
result = __nx_compare(B1, B2, B3)
% result is false, as length("hello") < length("world!")
```

B1 = "hello" B2 = "world!" B3 = 1 result = __nx_compare(B1, B2, B3)

Compare numeric values of B1 and B2

```
B1 = "42"
B2 = "24"
B3 = 1
result = __nx_compare(B1, B2, B3)
% result is true, as 42 > 24 (numeric comparison)
```

B1 = "42" B2 = "24" B3 = 1 result = __nx_compare(B1, B2, B3)

String comparison of B1 and B2

```
B1 = "abc"
B2 = "def"
B3 = ""
result = __nx_compare(B1, B2, B3)
% result is false, as "abc" != "def"
```

B1 = "abc" B2 = "def" B3 = "" result = __nx_compare(B1, B2, B3)

Pattern matching B1 against B2

```
B1 = "abc123"
B2 = "[a-z]+[0-9]+"
B3 = 0
result = __nx_compare(B1, B2, B3)
% result is true, as "abc123" matches the regex "[a-z]+[0-9]+"
```

B1 = "abc123" B2 = "[a-z]+[0-9]+" B3 = 0 result = __nx_compare(B1, B2, B3)

Relational comparison using B4

```
B1 = 10
B2 = 20
B3 = 1
B4 = "1"
result = __nx_compare(B1, B2, B3, B4)
% result is true, as 10 <= 20 (numeric comparison with B4 = 1)
```

B1 = 10 B2 = 20 B3 = 1 B4 = "1" result = __nx_compare(B1, B2, B3, B4)

Case-insensitive equality comparison

```
B1 = "Hello"
B2 = "hello"
```

```
B3 = ""
B4 = "i"
result = __nx_compare(B1, B2, B3, B4)
% result is true, as "Hello" == "hello" (case-insensitive)
```

B1 = "Hello" B2 = "hello" B3 = "" B4 = "i" result = __nx_compare(B1, B2, B3, B4)

II __nx_equality

__nx_equality(B1, B2, B3)

```
function __nx_equality(B1, B2, B3,    b, e, g) {
    b = substr(B2, 1, 1)
    if (b == ">") {
        e = 2
        g = 1
    } else if (b == "<") {
        e = "a"
        g = "i"
    } else if (b == "=") {
        e = ""
    } else if (b == "~") {
        e = 0
    } else {
        b = ""
    }
    if (b) {
        if (__nx_compare(substr(B2, 2, 1), "=", 1)) {
            b = g
        } else {
            b = e
        }
        e = substr(B2, length(B2), 1)
        if (__nx_compare(e, "a", 1))
            return __nx_compare(B1, B3, "", b)
        else if (__nx_compare(e, "_", 1))
            return __nx_compare(B1, B3, 0, b)
        else
            return __nx_compare(B1, B3, 1, b)
    }
    return __nx_compare(B1, B2)
}
```

function __nx_equality(B1, B2, B3, b, e, g) b = substr(B2, 1, 1) if (b == ">") e = 2 g = 1 else if (b == "<") e = "a" g = "i" else if (b == "=") e = "" else if (b == " ") e = 0 else b = "" if (b) if (__nx_compare(substr(B2, 2, 1), "=", 1)) b = g else b = e e = substr(B2, length(B2), 1) if (__nx_compare(e, "a", 1)) return __nx_compare(B1, B3, "", b) else if (__nx_compare(e, "_", 1)) return __nx_compare(B1, B3, 0, b) else return __nx_compare(B1, B3, 1, b) return __nx_compare(B1, B2)

II __nx_equality

Dynamically evaluates equality or relational conditions between inputs. **B2** specifies the operator (>, <, =, or ~) and controls the type of comparison. The function uses **__nx_compare** for nuanced behavior based on **awk** capabilities.

- **B1**: The first input to compare.
- **B2**: Specifies the operator and additional flags for comparison logic. When **B2** starts with:
  - ">": Relational comparison (greater than).
  - "<": Relational comparison (less than).
  - "=": Strict equality check.
  - " ": Pattern matching (~).
- **B3**: The second input to compare against **B1**.

Compare B1 > B3

```
B1 = 5
B2 = ">"
B3 = 3
result = __nx_equality(B1, B2, B3)
% result is true, as 5 > 3
```

B1 = 5 B2 = ">" B3 = 3 result = __nx_equality(B1, B2, B3)

Compare B1 == B3

```
B1 = "hello"
B2 = "="
B3 = "hello"
result = __nx_equality(B1, B2, B3)
% result is true, as "hello" == "hello"
```

B1 = "hello" B2 = "=" B3 = "hello" result = __nx_equality(B1, B2, B3)

Check if B1 matches regex B3

```
B1 = "abc123"
B2 = "~"
B3 = "[a-z]+[0-9]+"
result = __nx_equality(B1, B2, B3)
% result is true, as "abc123" matches the regex "[a-z]+[0-9]+"
```

B1 = "abc123" B2 = " " B3 = "[a-z]+[0-9]+" result = __nx_equality(B1, B2, B3)

Compare B1 <= B3

```
B1 = 7
B2 = "<="
B3 = 10
result = __nx_equality(B1, B2, B3)
% result is true, as 7 <= 10
```

B1 = 7 B2 = "<=" B3 = 10 result = __nx_equality(B1, B2, B3)

II __nx_swap

__nx_swap(V, D1, D2)

```
function __nx_swap(V, D1, D2, t) {
    t = V[D1]
    V[D1] = V[D2]
    V[D2] = t
}
```

function __nx_swap(V, D1, D2, t) t = V[D1] V[D1] = V[D2] V[D2] = t

II __nx_swap

Swaps the values of two indices within an array or associative array. This ensures flexibility in dynamically rearranging or reordering data structures. A temporary variable (‘t‘) protects against loss during the exchange.

- **V**: The array or associative array containing values to be swapped.
- **D1**: The first index (or key) whose value will be swapped.
- **D2**: The second index (or key) whose value will be swapped.

Basic swap of numeric values

```
V = [10, 20, 30, 40]
D1 = 1
D2 = 3
__nx_swap(V, D1, D2)
% Result: V = [10, 40, 30, 20], as the values at indices 1 and 3 are swapped
```

V = [10, 20, 30, 40] D1 = 1 D2 = 3 __nx_swap(V, D1, D2)

Swap in a string array

```
V = ["apple", "banana", "cherry"]
D1 = 0
D2 = 2
__nx_swap(V, D1, D2)
% Result: V = ["cherry", "banana", "apple"], as the values at indices 0 and 2
↪are swapped
```

V = ["apple", "banana", "cherry"] D1 = 0 D2 = 2 __nx_swap(V, D1, D2)

Swapping the same index (no-op)

```
V = [1, 2, 3]
D1 = 1
D2 = 1
__nx_swap(V, D1, D2)
% Result: V = [1, 2, 3], as swapping the same index has no effect
```

V = [1, 2, 3] D1 = 1 D2 = 1 __nx_swap(V, D1, D2)

Swapping in an associative array

```
V["a"] = "x"
V["b"] = "y"
D1 = "a"
D2 = "b"
__nx_swap(V, D1, D2)
% Result: V = {"a": "y", "b": "x"}, as the values at keys "a" and "b" are
↪swapped
```

V["a"] = "x" V["b"] = "y" D1 = "a" D2 = "b" __nx_swap(V, D1, D2)

Nested array swap

```
V = [[1, 2], [3, 4], [5, 6]]
D1 = 0
D2 = 2
__nx_swap(V, D1, D2)
% Result: V = [[5, 6], [3, 4], [1, 2]], as the nested arrays at indices 0 and 2
↪are swapped
```

V = [[1, 2], [3, 4], [5, 6]] D1 = 0 D2 = 2 __nx_swap(V, D1, D2)

III Structures

III Structures

The following functions provide comprehensive utilities for creating, managing, and manipulating structured data like arrays and hashmaps, enabling efficient operations across indexed elements.

- **nx_find_index(D1, S, D2)**: Searches for the first occurrence of a pattern within a string, with additional constraints. Returns the index of the match or modifies behavior based on optional parameters.

- **nx_next_pair(D1, V1, V2, D2, B1, B2)**: Retrieves the next pair of start and end indices within a string (**D1**), based on associative array delimiters (**V1**). Outputs indices and their lengths to the result vector (**V2**), while handling escape constraints (**D2**). Logic flags (**B1**, **B2**) control fallback behavior and prioritization during evaluation.

- **nx_tokenize(D1, V1, V2, D2, B1, B2)**: Tokenizes an input string (**D1**) based on start and end delimiters (**V2**). Extracted tokens are stored in the output vector (**V1**). Handles escape sequences (**D2**) and allows prioritization or fallback using logical flags (**B1**, **B2**).

III nx_find_index

nx_find_index(D1, S, D2)

```
function nx_find_index(D1, S, D2,    f, m)
{
    if (__nx_defined(D1, 1)) {
        f = 0
        S = __nx_else(S, " ")
        D2 = __nx_else(__nx_defined(D2, 1), "\\\\")
        while (match(D1, S)) {
            f = f + RSTART
            if (! (match(substr(D1, 1, RSTART - 1), D2 "+$") && D2) || int(RLENGTH % 2) == 0)
                break
            f = f + RLENGTH
            D1 = substr(D1, f + 1)
        }
        return f
    }
}
```

function nx_find_index(D1, S, D2, f, m) if (__nx_defined(D1, 1)) f = 0 S = __nx_else(S, " ") D2 = __nx_else(__nx_defined(D2, 1), "

") while (match(D1, S)) f = f + RSTART if (! (match(substr(D1, 1, RSTART - 1), D2 "+")$D2)||int(RLENGTHbreakf = f + RLENGTHD1 = substr(D1, f+1)returnf$

III nx_find_index

Searches for the first match of a given pattern (**S**) within a string (**D1**) while applying optional constraints (**D2**). The function handles fallback conditions and uses nuanced logic to account for escape characters and repeated patterns.

- **D1**: The input string to search.
- **S**: The primary pattern to search for. Defaults to the space character (‘ ‘).
- **D2**: An optional secondary pattern used to constrain matches (e.g., escape sequences). Defaults to the backslash (‘\‘).

Basic pattern matching

```
D1 = "hell\o world"
S = "o"
```

```
result = nx_find_index(D1, S)
% result is 8
% Explanation: The first occurrence of "o" in "hello world" is excaped, the
↪next occurence is at index 8.
```

D1 = "helløworld" S = "o" result = nx_find_index(D1, S)

No match for the pattern

```
D1 = "hello world"
S = "z"
result = nx_find_index(D1, S)
% result is 0
% Explanation: Since "z" doesn't exist in the string, the function returns 0.
```

D1 = "hello world" S = "z" result = nx_find_index(D1, S)

Default parameters

```
D1 = "this is an example"
result = nx_find_index(D1)
% result is 5
% Explanation: The default pattern `S` is a space character, and the first
↪space is at index 5.
```

D1 = "this is an example" result = nx_find_index(D1)

Complex string with escape sequences

```
D1 = "path\\to\\file"
S = "\\\\"
D2 = "\\\\"
result = nx_find_index(D1, S, D2)
% result is 5
% Explanation: The function navigates the string while respecting escape
↪constraints and finds the first valid match.
```

D1 = "path
to
file" S = "

" D2 = "

" result = nx_find_index(D1, S, D2)

III nx_next_pair

nx_next_pair(D1, V1, V2, D2, B1, B2, s, s_l, e, e_l, f, i)

```
function nx_next_pair(D1, V1, V2, D2, B1, B2, s, s_l, e, e_l, f, i) {
    if (length(V1) && D1 != "") {
        for (i in V1) {
            if ((f = nx_find_index(D1, i, D2)) && (! s || __nx_if(B2, f > s, f < s))) {
                s = f
                s_l = length(i)
                if (length(V1[i]) && (f = nx_find_index(substr(D1, s + s_l + 1), V1[i], D2))) {
                    e = f
                    e_l = length(V1[i])
                } else {
                    e = ""
                    e_l = ""
                }
            }
        }
        if (! s && B1) {
            s = length(D1) + 1
        }
        V2[++V2[0]] = s
        V2[V2[0] "_" s] = s_l
        V2[++V2[0]] = e
        V2[V2[0] "_" e] = e_l
        return V2[0] - 1
    }
}
```

function nx_next_pair(D1, V1, V2, D2, B1, B2, s, s_l, e, e_l, f, i) if (length(V1) D1 != "") for (i in V1) if ((f = nx_find_index(D1, i, D2)) (! s || __nx_if(B2, f > s, f < s))) s = f s_l = length(i) if (length(V1[i]) (f = nx_find_index(substr(D1, s + s_l + 1), V1[i], D2))) e = f e_l = length(V1[i]) else e = "" e_l = ""   if (! s B1) s = length(D1) + 1 V2[++V2[0]] = s V2[V2[0] "_" s] = s_l V2[++V2[0]] = e V2[V2[0] "_" e] = e_l return V2[0] - 1

III nx_next_pair

Retrieves the next pair of start and end indices from the input string (**D1**) based on specified delimiters (**V1**). Stores indices and their lengths in the output vector (**V2**) for subsequent operations. Handles escape sequences (**D2**) and prioritizes pairs based on logical conditions (**B1**, **B2**).

- **D1**: The input string to search for start and end pairs.
- **V1**: An associative array mapping start delimiters (keys) to end delimiters (values).
- **V2**: A vector to store indices and lengths of matched pairs.
- **D2**: Constraints for handling escape sequences or specific delimiters.
- **B1**: A flag to set a fallback start index if none is found.
- **B2**: A logical parameter to prioritize pairs based on index comparison (above or below the previous match).

Matching a single pair of delimiters

```
D1 = "<pair start content end />"
V1["<pair start"] = "end />"
V2[0] = 0
result = nx_next_pair(D1, V1, V2)
% Result:
% V2[1] = 1, V2[1_s] = 11
% V2[2] = 23, V2[2_e] = 6
% Explanation: Finds the start and end delimiters, capturing their indices and
↪lengths.
```

D1 = "<pair start content end />" V1["<pair start"] = "end />" V2[0] = 0 result = nx_next_pair(D1, V1, V2)

Handling fallback start index

```
D1 = "no delimiters here"
V1["<start"] = "end />"
V2[0] = 0
result = nx_next_pair(D1, V1, V2, "", 1, 0)
% Result:
% V2[1] = 21, V2[1_s] = ""
% V2[2] = "", V2[2_e] = ""
% Explanation: Sets the fallback start index as the length of \NexOption{D1} +
↪1 since no match was found.
```

D1 = "no delimiters here" V1["<start"] = "end />" V2[0] = 0 result = nx_next_pair(D1, V1, V2, "", 1, 0)

Multiple pairs with prioritization

```
D1 = "<startA>contentA<endA><startB>contentB<endB>"
V1["<startA>"] = "<endA>"
V1["<startB>"] = "<endB>"
result = nx_next_pair(D1, V1, V2, "", 0, 1)
% Result:
% V2[1] = 1, V2[1_s] = 8
% V2[2] = 20, V2[2_e] = 7
% Explanation: Prioritizes pairs based on index comparison and logical
↪conditions.
```

D1 = "<startA>contentA<endA><startB>contentB<endB>" V1["<startA>"] = "<endA>" V1["<startB>"] = "<endB>" result = nx_next_pair(D1, V1, V2, "", 0, 1)

III nx_tokenize

nx_tokenize(D1, V1, V2, D2, B1, B2)

```
function nx_tokenize(D1, V1, V2, D2, B1, B2, i, j, m, v, s) {
    if (length(V2) && D1 != "") {
        while (D1) {
            i = nx_next_pair(D1, V2, v, D2, 1, B1)
            m = substr(D1, v[i], v[i "_" v[i]])
            j = v[i] + v[i "_" v[i]]
            s = s substr(D1, 1, v[i] - 1)
            if (length(V2[m])) {
                s = s substr(D1, j, v[++i])
                j = j + v[i] + v[i "_" v[i]]
            } else {
                V1[++V1[0]] = s
                s = ""
            }
            D1 = substr(D1, j)
        }
        if (s != "")
            V1[++V1[0]] = s
        delete v
        return V1[0]
    }
}
```

function nx_tokenize(D1, V1, V2, D2, B1, B2, i, j, m, v, s) if (length(V2) D1 != "") while (D1) i = nx_next_pair(D1, V2, v, D2, 1, B1) m = substr(D1, v[i], v[i "_" v[i]]) j = v[i] + v[i "_" v[i]] s = s substr(D1, 1, v[i] - 1) if (length(V2[m])) s = s substr(D1, j, v[++i]) j = j + v[i] + v[i "_" v[i]] else V1[++V1[0]] = s s = "" D1 = substr(D1, j) if (s != "") V1[++V1[0]] = s delete v return V1[0]

III nx_tokenize

Processes an input string (**D1**) to extract and tokenize content into an output vector (**V1**). Tokens are defined based on start and end pairs in (**V2**). Handles constraints (**D2**) and logical prioritization (**B1**, **B2**).

- **D1**: The input string to be tokenized.
- **V1**: A vector to store tokenized outputs. Each entry represents a complete token.
- **V2**: An associative array containing start and end delimiters for tokenization.
- **D2**: Constraints to handle escape sequences or special rules during tokenization.
- **B1**: A flag to determine logical prioritization for the next token (e.g., based on position).
- **B2**: A flag to control fallback behavior or logical prioritization when matching pairs.

Tokenizing with basic delimiters

```
D1 = "token1, token2, token3"
V2[""] = ","
V1[0] = 0
result = nx_tokenize(D1, V1, V2, "")
% Result:
% V1[1] = "token1"
% V1[2] = "token2"
% V1[3] = "token3"
% Explanation: Tokenizes based on the delimiter "," and stores the tokens in
↪\NexOption{V1}.
```

D1 = "token1, token2, token3" V2[""] = "," V1[0] = 0 result = nx_tokenize(D1, V1, V2, "")

Handling nested delimiters

```
D1 = "start1(content1)end1 start2(content2)end2"
V2["start1"] = "end1"
V2["start2"] = "end2"
V1[0] = 0
result = nx_tokenize(D1, V1, V2, "")
% Result:
% V1[1] = "content1"
% V1[2] = "content2"
% Explanation: Matches "start1" with "end1" and "start2" with "end2",
↪extracting content between them.
```

D1 = "start1(content1)end1 start2(content2)end2" V2["start1"] = "end1" V2["start2"] = "end2" V1[0] = 0 result = nx_tokenize(D1, V1, V2, "")

Tokenizing with escapes

```
D1 = "token1\,token2, token3"
V2[""] = ","
V1[0] = 0
result = nx_tokenize(D1, V1, V2, "\\")
% Result:
% V1[1] = "token1\,token2"
% V1[2] = "token3"
% Explanation: Accounts for escaped commas ("\,") and ensures they are not
↪treated as delimiters.
```

D1 = "token1 token2, token3" V2[""] = "," V1[0] = 0 result = nx_tokenize(D1, V1, V2, "
")