Posix-Nexus AWK

Canine-Table

April 14, 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_num_map(V)**: Initializes an associative array (**V**) with mappings for digits (0-9), where each digit maps to itself. Ideal for digit-based validations or transformations.
- **__nx_lower_map(V)**: Initializes an associative array (**V**) with mappings for digits (0-9) and lowercase letters (a-z), leveraging **nx_bijective** for the letter mappings. Ideal for scenarios involving alphanumeric parsing or transformations.
- **__nx_upper_map(V)**: Extends **__nx_lower_map** by adding bijective mappings for uppercase letters (A-Z) to an associative array (**V**), ensuring full alphanumeric coverage.
- **__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_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.

II Miscellaneous

- __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.
- __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_upper_map

__nx_upper_map(V)

```
function __nx_upper_map(V, i) {
    __nx_lower_map(V)
    for (i = 36; i < 62; i++)
        nx_bijective(V, i, "", sprintf("%c", i + 29))
}
```

function __nx_upper_map(V, i) __nx_lower_map(V) for (i = 36; i < 62; i++) nx_bijective(V, i, "", sprintf("

^ II __nx_upper_map

The **__nx_upper_map** function extends the mappings established by **__nx_lower_map** by adding bijective mappings for uppercase English letters (A–Z). It combines numerical digit mappings (0–9), lowercase letters (a–z), and uppercase letters (A–Z) into the associative array (**V**).

- **V**: An associative array passed by reference. After execution, it contains mappings for digits (0–9), lowercase letters (a–z), and uppercase letters (A–Z).

II __nx_num_map

__nx_num_map(V)

```
function __nx_num_map(V, i) {
    for (i = 0; i < 10; i++)
        V[i] = i
}
```

function __nx_num_map(V, i) for (i = 0; i < 10; i++) V[i] = i

II __nx_num_map

The **__nx_num_map** function initializes an associative array (**V**) with mappings for numerical digits, where each digit maps to itself (e.g., 0 maps to 0, 1 maps to 1, etc.). This utility is useful for tasks involving digit-based validations or transformations.

- **V**: An associative array passed by reference, populated with digit mappings (0 – 9).

II __nx_lower_map

__nx_lower_map(V)

```
function __nx_lower_map(V, i) {
    __nx_num_map(V)
    for (i = 10; i < 36; i++)
        nx_bijective(V, i, "", sprintf("%c", i + 87))
}
```

function __nx_lower_map(V, i) __nx_num_map(V) for (i = 10; i < 36; i++) nx_bijective(V, i, "", sprintf("

^ II __nx_lower_map

The **__nx_lower_map** function initializes an associative array (V) with mappings for numerical digits (0-9) and lowercase English letters (a-z). Numerical digits map to themselves, while lowercase letters are bijectively mapped using their ASCII values.

- V: An associative array passed by reference. After execution, it includes digit-to-digit mappings (0-9) and bijective mappings for lowercase letters (a-z).

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

```
}
```

function __nx_str_map(V) nx_bijective(V, ++V[0], "upper", "A-Z") nx_bijective(V, ++V[0], "lower", "a-z") nx_bijective(V, ++V[0], "xupper", "A-F") nx_bijective(V, ++V[0], "xlower", "a-f") nx_bijective(V, ++V[0], "digit", "0-9") nx_bijective(V, ++V[0], "alpha", V["upper"] V["lower"]) nx_bijective(V, ++V[0], "xdigit", V["digit"] V["xupper"] V["xlower"]) nx_bijective(V, ++V[0], "alnum", V["digit"] V["alpha"]) nx_bijective(V, ++V[0], "print", "20-7e") nx_bijective(V, ++V[0], "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) result is true, as B1 is true (1) regardless of 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) result is false, as B1 is an empty string, which fails the defined check

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

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) result is true, as B1 (0) has a length, even though it evaluates as false in conditions

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) result is true, as B2 ("fallback") is defined and compensates for B1 being empty

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) result is true, as B1 is true (1), overriding the fallback condition of B2 (0)

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) result is true, as B1 ("abc") is valid and overrides the empty fallback (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) result is true, as B1 fails the condition, falling back to B2 (42)

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) result is true, as the pattern "[a-z]+" matches B2 ("hello"), overriding the fallback (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) result is "True Case", as B1 is true (1), returning the second argument

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) result is "Condition Met", as B1 is non-empty and therefore true, returning the second argument

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) result is "Greater", as B1 evaluates to true (5 > 3)

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) result is "Pattern Matches", as B1 evaluates to true due to the regex match

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) result is "Fallback Case", as B1 is false (0), returning the third argument

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) 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

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) 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

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) result is false, as none of the relationships between B1, B2, and B3 fulfill the XOR conditions after fallback adjustments with __nx_else

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) result is true, as __nx_defined(B1, B4) and __nx_defined(B2, B5) are true, and the adjusted relationships satisfy the condition logic

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]+/) result is true, as N matches the first regex pattern for signed integers (-123)

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 $/^{[}0-9]+[.][0-9]+/)$ result is true, as N matches the first regex pattern for signed decimals (123.45)

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) result is true, as B1 is true (1) and B2 is false (0), making XOR true

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) result is true, as B2 satisfies the fallback condition (B4), while B1 is false, fulfilling XOR

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) result is false, as B1 and B2 both satisfy their respective truth and length conditions, violating XOR

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

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) result is false, as length("hello") < length("world!")

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) result is true, as 42 > 24 (numeric comparison)

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) result is false, as "abc" != "def"

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) result is true, as "abc123" matches the regex "[a-z]+[0-9]+"

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) result is true, as 10 <= 20 (numeric comparison with B4 = 1)

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) result is true, as "Hello" == "hello" (case-insensitive)

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) result is true, as 5 > 3

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) result is true, as "hello" == "hello"

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) result is true, as "abc123" matches the regex "[a-z]+[0-9]+"

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) result is true, as 7 <= 10

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) Result: V = [10, 40, 30, 20], as the values at indices 1 and 3 are swapped

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) Result: V = ["cherry", "banana", "apple"], as the values at indices 0 and 2 are swapped

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) Result: V = [1, 2, 3], as swapping the same index has no effect

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) Result: V = "a": "y", "b": "x", as the values at keys "a" and "b" are swapped

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) Result: V = [[5, 6], [3, 4], [1, 2]], as the nested arrays at indices 0 and 2 are swapped

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_bijective(V, D1, D2, D3)**: Updates a bijective mapping stored in V, based on three keys (D1, D2, D3). Creates or modifies circular references if all keys are provided, swaps references if two keys are defined, or performs deletions and adjustments based on the keys. Handles edge cases dynamically.

- **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, S1, S2, V2, D2, B1, B2)**: Tokenizes the input string (D1) into segments based on custom delimiters and pair boundaries. The primary delimiter (S1, defaulting to a comma) and an optional secondary delimiter (S2) are used together with a quote map (V2) and an optional extra pattern (D2) to drive token separation. Flags (B1 and B2) control pairing behavior and trimming of tokens, respectively. The found tokens are stored in the array (V1) with the token count in V1[0].

- **nx_vector(D, V1, S, V2)**: Processes a data string (D) by tokenizing it into parts using a delimiter (S) and associative array mappings (V2). Uses **__nx_quote_map** for initialization and **nx_tokenize** for parsing.

- **nx_trim_vector(D, V1, S, V2)**: Tokenizes a data string (D) into parts using delimiters and mappings (V2), then trims each token to remove unnecessary whitespace or characters.

- **nx_uniq_vector(D, V1, S, V2, B1, B2)**: Constructs a unique vector (V1) from input data (D), eliminating duplicates, optionally counting occurrences, and leveraging auxiliary vectors (V2) for processing.

- **nx_length(V, B)**: Calculates the length of elements within a vector (V) and returns the largest or smallest length based on the logical condition (B).

- **nx_boundary(D, V1, V2, B1, B2)**: Filters elements in a vector (V1) that match boundary conditions with a given string (D), and stores the results in another vector (V2). Provides options to specify matching direction (B1) and optionally delete the input vector (B2).

- **nx_filter(D1, D2, V1, V2, B)**: Filters elements from an input vector (V1) based on a flexible equality condition (D1, D2) and stores the results in an output vector (V2). Provides an option (B) to delete the input vector after processing.

III Structures

- **nx_option(D, V1, V2, B1, B2)**: Determines a selected index from an input vector (**V1**) based on boundary conditions (**D**) and advanced filtering logic. Stores intermediate results in a secondary vector (**V2**), applying conditions (**B1**, **B2**) to refine the output.

- **nx_trim_split(D, V, S)**: Splits a trimmed input string (**D**) into parts stored in a vector (**V**), using a customizable delimiter (**S**). Defaults to a comma (,) when no delimiter is provided.

III nx_bijective

nx_bijective(V, D1, B, D2)

```
function nx_bijective(V, D1, D2, D3)
{
    if (D1 != "") {
        if (D2) {
            if (D3 != "") {
                V[D1] = D2
                V[D2] = D3
                V[D3] = D1
            } else {
                V[D1] = D2
                V[D2] = D1
            }
        } else if (D3 != "") {
                V[V[D1]] = D3
            if (D2 != "")
                delete V[D1]
        }
    }
}
```

function nx_bijective(V, D1, D2, D3) if (D1 != "") if (D2) if (D3 != "") V[D1] = D2 V[D2] = D3 V[D3] = D1 else V[D1] = D2 V[D2] = D1 else if (D3 != "") V[V[D1]] = D3 if (D2 != "") delete V[D1]

bŸ", V["Y"] = "X"

Modify mappings

```
print nx_bijective(V, "P", 1, "Q")
# Result: V[V["P"]] = "Q", V["P"] deleted, V["Q"] = V1
```

print nx_bijective(V, "P", 1, "Q") Result: V[V["P"]] = "Q", V["P"] deleted, V["Q"] = V1

No operation

```
print nx_bijective(V, "", "M", "N")
# Result: (no changes made to V)
```

print nx_bijective(V, "", "M", "N") Result: (no changes made to V)

Empty mapping

```
print nx_bijective(V, "", "", "")
# Result: (no changes made to V)
```

print nx_bijective(V, "", "", "") Result: (no changes made to V)

III nx_find_index

nx_find_index(D1, S, D2)

```
function nx_find_index(D1, S, D2,     f)
{
    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) 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) result is 8 Explanation: The first occurrence of "o" in "hello world" is excaped, the next occurence is at index 8.

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) result is 0 Explanation: Since "z" doesn't exist in the string, the function returns 0.

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) result is 5 Explanation: The default pattern ‘S‘ is a space character, and the first space is at index 5.

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) result is 5 Explanation: The function navigates the string while respecting escape constraints and finds the first valid match.

III nx_next_pair

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

```
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) 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.

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 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) Result: V2[1] = 21, V2[1_s] = "" V2[2] = "", V2[2_e] = "" Explanation: Sets the fallback start index as the length of D1 + 1 since no match was found.

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) 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.

III nx_tokenize

nx_tokenize(D1, V1, S1, S2, V2, D2, B1, B2)

```
function nx_tokenize(D1, V1, S1, S2, V2, D2, B1, B2, v1, v2, c, s, i, l, t) {
    if (D1 != "") {
        if (! length(V2))
            __nx_quote_map(V2)
        S1 = __nx_else(S1, ",")
        V2[S1] = ""
        if (S2) {
            V2[S2] = ""
            nx_bijective(v1, S1, S2)
            c = v1[S1]
        }
        while (D1) {
            i = nx_next_pair(D1, V2, v2, D2, 1, B1)
            t = substr(D1, v2[i], v2[i "_" v2[i]])
            l = v2[i] + v2[i "_" v2[i]]
            s = s substr(D1, 1, v2[i] - 1)
            if (V2[t] == "" || s == D1) {
                s = __nx_if(B2, nx_trim_str(s), s)
                if (S2 && (t in v1 || s == D1)) {
                    if (c == t || s == D1) {
                        if (c == S2)
                            V1[++V1[0]] = s
                        else
                            V1[V1[V1[0]]] = s
                        c = v1[c]
                    } else {
                        V1[++V1[0]] = s
                        if (t == S1 || (s == D1 && c = S1))
                            V1[V1[V1[0]]] = 1
                    }
                } else {
                    V1[++V1[0]] = s
                }
                s = ""
            } else {
                s = s substr(D1, l, v2[++i])
                l = l + v2[i] + v2[i "_" v2[i]]
            }
            D1 = substr(D1, l)
        }
        delete v1
        delete v2
        return V1[0]
    }
}
```

function nx_tokenize(D1, V1, S1, S2, V2, D2, B1, B2, v1, v2, c, s, i, l, t) if (D1 != "") if (! length(V2)) __nx_quote_map(V2) S1 = __nx_else(S1, ",") V2[S1] = "" if (S2) V2[S2] = "" nx_bijective(v1, S1, S2) c = v1[S1] while (D1) i = nx_next_pair(D1, V2, v2, D2, 1, B1) t = substr(D1, v2[i], v2[i "_" v2[i]]) l = v2[i] +

v2[i "_" v2[i]] s = s substr(D1, 1, v2[i] - 1) if (V2[t] == "" || s == D1) s = __nx_if(B2, nx_trim_str(s), s) if (S2 (t in v1 || s == D1)) if (c == t || s == D1) if (c == S2) V1[++V1[0]] = s else V1[V1[V1[0]]] = s c = v1[c] else V1[++V1[0]] = s if (t == S1 || (s == D1 c = S1)) V1[V1[V1[0]]] = 1 else V1[++V1[0]] = s s = "" else s = s substr(D1, l, v2[++i]) l = l + v2[i] + v2[i "_" v2[i]] D1 = substr(D1, l) delete v1 delete v2 return V1[0]

III nx_tokenize

Tokenizes an input string (**D1**) into multiple segments by finding token boundaries using a customizable delimiter setup. The primary delimiter (**S1**) defaults to a comma (via __nx_else), and if a secondary delimiter (**S2**) is provided, a reciprocal mapping is created using nx_bijective. A mapping array (**V2**) – automatically populated by __nx_quote_map if empty – drives pair-based token detection (via nx_next_pair). Depending on the flag (**B2**), tokens are optionally trimmed (using nx_trim_str), and the tokens are stored in the array (**V1**) with the count in V1[0].

- **D1**: The input string to be tokenized.
- **V1**: An array (destination table) in which tokens are accumulated; V1[0] holds the token count.
- **S1**: The primary delimiter (defaults to "," if not provided).
- **S2**: An optional secondary delimiter for creating bidirectional token mappings.
- **V2**: A mapping array for quotes/delimiters; if empty, it's populated by __nx_quote_map.
- **D2**: An optional pattern used in token boundary detection.
- **B1**: A flag passed on to nx_next_pair for controlling pairing behavior.
- **B2**: A flag that, when true, applies trimming to tokens via nx_trim_str.

Basic tokenization with comma

```
# Suppose V1 and V2 are pre-declared (e.g., V1 = [] and V2 = [])
print nx_tokenize("apple,banana,cherry", V1, ",", "", V2, "", 0, 0)
# Expected Result: 3
# Where V1[1] = "apple", V1[2] = "banana", V1[3] = "cherry"
```

Suppose V1 and V2 are pre-declared (e.g., V1 = [] and V2 = []) print nx_tokenize("apple,banana,cherry", V1, ",", "", V2, "", 0, 0) Expected Result: 3 Where V1[1] = "apple", V1[2] = "banana", V1[3] = "cherry"

Tokenization with trimming

```
# Here B2 is set to 1 (true) so that leading/trailing spaces are trimmed.
print nx_tokenize("  dog , cat ,  bird  ", V1, ",", "", V2, "", 0, 1)
```

```
# Expected Result: 3
# With trimmed tokens: V1[1] = "dog", V1[2] = "cat", V1[3] = "bird"
```

Here B2 is set to 1 (true) so that leading/trailing spaces are trimmed. print nx_tokenize(" dog , cat , bird ", V1, ",", "", V2, "", 0, 1) Expected Result: 3 With trimmed tokens: V1[1] = "dog", V1[2] = "cat", V1[3] = "bird"

III nx_length

nx_length(V, B)

```
function nx_length(V, B,     i, j, k)
{
    if (length(V) && 0 in V) {
        for (i = 1; i <= V[0]; i++) {
            j = length(V[i])
            if (! k || __nx_if(B, k < j, k > j))
                k = j
        }
        return int(k)
    }
}
```

function nx_length(V, B, i, j, k) if (length(V) 0 in V) for (i = 1; i <= V[0]; i++) j = length(V[i]) if (! k || __nx_if(B, k < j, k > j)) k = j return int(k)

III nx_length

Calculates the length of elements within an input vector (**V**) and determines the largest or smallest length based on a given logical condition (**B**).

- **V**: The input vector containing elements whose lengths need to be evaluated.
- **B**: A logical flag that determines whether to return the smallest length (if ‘false‘) or the largest length (if ‘true‘).

Finding the largest length

```
V[0] = 3
V[1] = "short"
V[2] = "longer"
V[3] = "lengthiest"
B = 1
result = nx_length(V, B)
# Result: 10
# Explanation: Evaluates the lengths of elements in V and returns the largest
↪value, which is 10 (from "lengthiest").
```

V[0] = 3 V[1] = "short" V[2] = "longer" V[3] = "lengthiest" B = 1 result = nx_length(V, B) Result: 10 Explanation: Evaluates the lengths of elements in V and returns the largest value, which is 10 (from "lengthiest").

Finding the smallest length

```
V[0] = 3
V[1] = "short"
V[2] = "longer"
V[3] = "lengthiest"
B = 0
result = nx_length(V, B)
# Result: 5
# Explanation: Evaluates the lengths of elements in V and returns the smallest
↪value, which is 5 (from "short").
```

V[0] = 3 V[1] = "short" V[2] = "longer" V[3] = "lengthiest" B = 0 result = nx_length(V, B) Result: 5 Explanation: Evaluates the lengths of elements in V and returns the smallest value, which is 5 (from "short").

III nx_boundary

nx_boundary(D, V1, V2, B1, B2)

```
function nx_boundary(D, V1, V2, B1, B2,      i)
{
    if (length(V) && 0 in V && D != "") {
        for (i = 1; i <= V1[0]; i++) {
            if (__nx_if(B1, V1[i] ~ D "$", V1[i] ~ "^" D))
                V2[++V2[0]] = V1[i]
        }
        if (B2)
            delete V1
        return V2[0]
    }
}
```

function nx_boundary(D, V1, V2, B1, B2, i) if (length(V) 0 in V D != "") for (i = 1; i <= V1[0]; i++) if (__nx_if(B1, V1[i] D "", $V1[i]$ "" $D))V2[++V2[0]] = V1[i]if(B2)deleteV1returnV2[0]$

III nx_boundary

Filters elements from an input vector (**V1**) that match the boundary conditions specified by a string (**D**). Results are stored in an output vector (**V2**), with the option to prioritize start or end boundary matching (**B1**). The function can optionally delete the input vector (**B2**) after processing.

- **D**: The string used to define boundary conditions for filtering elements.
- **V1**: The input vector containing elements to be filtered.
- **V2**: The output vector to store elements matching the boundary conditions.
- **B1**: A logical flag to specify boundary matching. If ‘true‘, matches elements ending with (**D**); if ‘false‘, matches elements starting with (**D**).
- **B2**: A flag to delete the input vector (**V1**) after filtering, if set to ‘true‘.

Filtering elements ending with a boundary

```
V1[1] = "boundary_end"
V1[2] = "no_match"
V1[3] = "another_end"
D = "end"
B1 = 1
nx_boundary(D, V1, V2, B1, 0)
```

```
# Result:
# V2[1] = "boundary_end"
# V2[2] = "another_end"
# Explanation: Filters elements in V1 that end with "end" and stores them in
↪V2.
```

V1[1] = "boundary_end" V1[2] = "no_match" V1[3] = "another_end" D = "end" B1 = 1 nx_boundary(D, V1, V2, B1, 0) Result: V2[1] = "boundary_end" V2[2] = "another_end" Explanation: Filters elements in V1 that end with "end" and stores them in V2.

Filtering elements starting with a boundary

```
V1[1] = "start_boundary"
V1[2] = "no_match"
V1[3] = "start_another"
D = "start"
B1 = 0
nx_boundary(D, V1, V2, B1, 0)
# Result:
# V2[1] = "start_boundary"
# V2[2] = "start_another"
# Explanation: Filters elements in V1 that start with "start" and stores them
↪in V2.
```

V1[1] = "start_boundary" V1[2] = "no_match" V1[3] = "start_another" D = "start" B1 = 0 nx_boundary(D, V1, V2, B1, 0) Result: V2[1] = "start_boundary" V2[2] = "start_another" Explanation: Filters elements in V1 that start with "start" and stores them in V2.

Deleting the input vector after filtering

```
V1[1] = "boundary_delete"
V1[2] = "no_match"
V1[3] = "delete_end"
D = "delete"
B1 = 1
B2 = 1
nx_boundary(D, V1, V2, B1, B2)
# Result:
# V2[1] = "delete_end"
# V1: Cleared
# Explanation: Filters elements in V1 matching the boundary condition "delete"
↪(ending with "delete"), stores "delete_end" in V2, and clears V1 after
↪processing due to B2 being set to true.
```

V1[1] = "boundary_delete" V1[2] = "no_match" V1[3] = "delete_end" D = "delete" B1 = 1 B2 = 1 nx_boundary(D, V1, V2, B1, B2) Result: V2[1] = "delete_end" V1: Cleared Explanation: Filters elements in V1 matching the boundary condition "delete" (ending with "delete"), stores "delete_end" in V2, and clears V1 after processing due to B2 being set to true.

III nx_filter

nx_filter(D1, D2, V1, V2, B)

```
function nx_filter(D1, D2, V1, V2, B,   i, v1, v2)
{
    if (length(V1) && 0 in V1) {
        for (i = 1; i <= V1[0]; i++) {
            if (__nx_equality(D1, D2, V1[i]))
                V2[++V2[0]] = V1[i]
        }
        if (B)
            delete V1
        return V2[0]
    }
}
```

function nx_filter(D1, D2, V1, V2, B, i, v1, v2) if (length(V1) 0 in V1) for (i = 1; i <= V1[0]; i++) if (__nx_equality(D1, D2, V1[i])) V2[++V2[0]] = V1[i] if (B) delete V1 return V2[0]

III nx_filter

Filters elements from an input vector (**V1**) based on a flexible equality condition defined by (**D1**) and (**D2**). Matching elements are appended to an output vector (**V2**). The function supports optional deletion of the input vector (**V1**) after filtering to optimize memory usage.

- **D1**: The primary value or pattern used for comparison.
- **D2**: The secondary value or pattern used to refine the comparison.
- **V1**: The input vector containing elements to be filtered.
- **V2**: The output vector to store elements that meet the equality condition.
- **B**: A flag to delete the input vector (**V1**) after processing if set to ‘true‘.

Filtering elements with a simple equality condition

```
V1[1] = "apple"
V1[2] = "orange"
V1[3] = "apple"
D1 = "apple"
D2 = "="
nx_filter(D1, D2, V1, V2, 0)
# Result:
# V2[1] = "apple"
```

```
# V2[2] = "apple"
# Explanation: Filters elements in V1 that are equal to "apple" and stores them
↪in V2.
```

V1[1] = "apple" V1[2] = "orange" V1[3] = "apple" D1 = "apple" D2 = "=" nx_filter(D1, D2, V1, V2, 0) Result: V2[1] = "apple" V2[2] = "apple" Explanation: Filters elements in V1 that are equal to "apple" and stores them in V2.

Filtering elements with a numeric condition

```
V1[1] = 10
V1[2] = 20
V1[3] = 30
D1 = 15
D2 = ">"
nx_filter(D1, D2, V1, V2, 0)
# Result:
# V2[1] = 20
# V2[2] = 30
# Explanation: Filters elements in V1 greater than 15 and stores them in V2.
```

V1[1] = 10 V1[2] = 20 V1[3] = 30 D1 = 15 D2 = ">" nx_filter(D1, D2, V1, V2, 0) Result: V2[1] = 20 V2[2] = 30 Explanation: Filters elements in V1 greater than 15 and stores them in V2.

Deleting the input vector after filtering

```
V1[1] = "match"
V1[2] = "no_match"
V1[3] = "match"
D1 = "match"
D2 = "="
B = 1
nx_filter(D1, D2, V1, V2, B)
# Result:
# V2[1] = "match"
# V2[2] = "match"
# V1: Cleared
# Explanation: Filters elements in V1 that match "match", stores them in V2,
↪and clears V1 after processing.
```

V1[1] = "match" V1[2] = "no_match" V1[3] = "match" D1 = "match" D2 = "=" B = 1 nx_filter(D1, D2, V1, V2, B) Result: V2[1] = "match" V2[2] = "match" V1: Cleared Explanation: Filters elements in V1 that match "match", stores them in V2, and clears V1 after processing.

III nx_option

nx_option(D, V1, V2, B1, B2)

```
function nx_option(D, V1, V2, B1, B2,    i, v1)
{
    if (length(V1) && 0 in V1) {
        if (nx_boundary(D, V1, v1, B1) > 1) {
            if (nx_filter(nx_append_str("0", nx_length(v1, B2)), "=_", v1, V2,
↪1) == 1) {
                i = V2[1]
                delete V2
                return i
            }
        } else {
            i = v1[1]
            delete v1
            return i
        }
    }
}
```

function nx_option(D, V1, V2, B1, B2, i, v1) if (length(V1) 0 in V1) if (nx_boundary(D, V1, v1, B1) > 1) if (nx_filter(nx_append_str("0", nx_length(v1, B2)), "=_", v1, V2, 1) == 1) i = V2[1] delete V2 return i else i = v1[1] delete v1 return i

III nx_option

Selects a string from an input vector (**V1**) that matches boundary conditions (**D**) and additional logic refinements. The function processes intermediate results using a secondary vector and logical conditions, enabling flexible selection criteria.

- **D**: A string or pattern used as a boundary condition for selection.
- **V1**: The primary input vector containing elements to evaluate.
- **V2**: A secondary vector used for intermediate filtering results.
- **B1**: A logical flag controlling boundary matching (e.g., start or end).
- **B2**: A logical flag influencing the filtering length criteria during refinement.

Selecting a string based on boundary conditions

```
V1[1] = "start_option"
V1[2] = "middle_match"
```

```
V1[3] = "end_option"
V1[0] = 3 # This was populated dynamically earlier
D = "option"
B1 = 1
B2 = 1
result = nx_option(D, V1, V2, B1, B2)
# Result: "end_option"
# Explanation: Filters V1 for elements matching the boundary condition "option"
↪at the end and returns the matching string ("end_option").
```

V1[1] = "start_option" V1[2] = "middle_match" V1[3] = "end_option" V1[0] = 3 This was populated dynamically earlier D = "option" B1 = 1 B2 = 1 result = nx_option(D, V1, V2, B1, B2) Result: "end_option" Explanation: Filters V1 for elements matching the boundary condition "option" at the end and returns the matching string ("end_option").

Returning a single matching string

```
V1[1] = "single_match"
V1[2] = "no_match"
V1[0] = 2 # Populated dynamically via earlier functions
D = "single"
B1 = 0
B2 = 1
result = nx_option(D, V1, V2, B1, B2)
# Result: "single_match"
# Explanation: Matches "single_match" based on the boundary condition "single"
↪at the start and returns the matching string.
```

V1[1] = "single_match" V1[2] = "no_match" V1[0] = 2 Populated dynamically via earlier functions D = "single" B1 = 0 B2 = 1 result = nx_option(D, V1, V2, B1, B2) Result: "single_match" Explanation: Matches "single_match" based on the boundary condition "single" at the start and returns the matching string.

No valid boundary match

```
V1[1] = "no_boundary"
V1[2] = "no_match"
V1[0] = 2 # Managed dynamically by earlier functions
D = "option"
B1 = 0
B2 = 0
result = nx_option(D, V1, V2, B1, B2)
# Result: None
# Explanation: No elements in V1 match the boundary condition "option", so the
↪function returns no result.
```

V1[1] = "no_boundary" V1[2] = "no_match" V1[0] = 2 Managed dynamically by earlier functions D = "option" B1 = 0 B2 = 0 result = nx_option(D, V1, V2, B1, B2) Result: None Explanation: No elements in V1

V1 match the boundary condition "option", so the function returns no result.

III nx_trim_split

nx_trim_split(D, V, S)

```
function nx_trim_split(D, V, S) {
    return split(nx_trim_str(D), V, "[ \v\t\n\f]*" __nx_else(S, ",") "[
↪\v\t\n\f]*")
}
```

function nx_trim_split(D, V, S) return split(nx_trim_str(D), V, "[ˇ]*" __nx_else(S, ",") "[ˇ]*")

^ III nx_trim_split

Splits a trimmed input string (**D**) into parts stored in a vector (**V**), using a customizable delimiter (**S**). Defaults to a comma (,) when no delimiter is provided.

- **D**: The input string to be trimmed and split.
- **V**: The vector where split segments are stored.
- **S**: The delimiter used for splitting. Defaults to a comma (,) if not provided.

Splitting with a custom delimiter

```
D = "apple ; orange ; banana"
S = ";"
result = nx_trim_split(D, V, S)
# Result: V[1] = "apple", V[2] = "orange", V[3] = "banana"
```

D = "apple ; orange ; banana" S = ";" result = nx_trim_split(D, V, S) Result: V[1] = "apple", V[2] = "orange", V[3] = "banana"

Default delimiter

```
D = "apple,orange,banana"
S = ""
result = nx_trim_split(D, V, S)
# Result: V[1] = "apple", V[2] = "orange", V[3] = "banana"
```

D = "apple,orange,banana" S = "" result = nx_trim_split(D, V, S) Result: V[1] = "apple", V[2] = "orange", V[3] = "banana"

Whitespace trimming

```
D = "  apple  ; orange ;   banana   "
S = ";"
result = nx_trim_split(D, V, S)
# Result: V[1] = "apple", V[2] = "orange", V[3] = "banana"
```

D = " apple ; orange ; banana " S = ";" result = nx_trim_split(D, V, S) Result: V[1] = "apple", V[2] = "orange", V[3] = "banana"

IV Strings

IV Strings

The following functions offer robust tools for processing, manipulating, and transforming strings, enabling versatile text-handling operations for a wide range of use cases.

- **nx_reverse_str(D, i, v)**: Reverses the input string (D) by splitting it into characters, reversing the order, and recombining it into a new string.

- **nx_escape_str(D)**: Escapes all characters in the input string (D) by prefixing them with a backslash (‘\‘).$Usefulforpreparingstringsforuseinregularexpressionsorothercontextswhere$

- `nx_join_str(D1, D2, S, D3)`: Joins two strings (D1 and D2) with a separator (S) while enclosing D2 in dynamic quotes ('/") based on the length and value of D3. Returns the concatenated result.

- `nx_slice_str(D, N, B1, B2)`: Slices a string (D) based on position (N), constraints (B1 and B2), and calculated start and end indices. Handles nuanced edge cases, including zero and empty values. Returns the sliced substring or a composite of two substrings.

- `nx_trim_str(D, S)`: Trims unwanted characters (S) from both ends of a string (D). Defaults to whitespace characters if S is not provided. Returns the trimmed string.

- `nx_append_str(D1, N, D2, B)`: Appends the string (D1) to itself N times, starting with an optional prefix (D2). The append direction is determined by (B): if true, appends in reverse; otherwise, appends normally. Returns the concatenated result.

- `nx_cut_str(D1, D2, B)`: Extracts or removes a substring matching the pattern (D2) from the input string (D1). The behavior is controlled by the flag (B): if true, removes the substring before the match; if false with a length, removes the substring after the match; otherwise, returns the matched substring.

- `nx_totitle(D, B1, B2)`: Converts an input string (D) to title case, capitalizing the first letter of each segment and lowercasing the rest. Handles escaped characters (B1) and relies on mappings (B2) for pair segmentation logic. Returns the title-cased string.

IV nx_reverse_str

nx_reverse_str(D, i, v)

```
function nx_reverse_str(D, i, v) {
    if ((i = split(D, v, "")) > 1) {
        D = ""
        do {
            D = D v[i]
        } while (--i)
    }
    delete v
    return D
}
```

function nx_reverse_str(D, i, v) if ((i = split(D, v, "")) > 1) D = "" do D = D v[i] while (–i) delete v return D

IV nx_reverse_str

Reverses the input string (**D**) and returns the reversed result. The function breaks the string into individual characters, reverses the order, and combines them back into a single string.

- **D**: The input string to be reversed.
- **v**: An auxiliary vector used to store the split characters during processing. Automatically cleared after use.

Basic string reversal

```
D = "hello"
result = nx_reverse_str(D)
# Result: "olleh"
```

D = "hello" result = nx_reverse_str(D) Result: "olleh"

Empty string case

```
D = ""
result = nx_reverse_str(D)
# Result: ""
```

D = "" result = nx_reverse_str(D) Result: ""

Palindrome string

```
D = "madam"
result = nx_reverse_str(D)
# Result: "madam" (same as input, as it's a palindrome)
```

D = "madam" result = nx_reverse_str(D) Result: "madam" (same as input, as it’s a palindrome)

IV nx_escape_str

nx_escape_str(D)

```
function nx_escape_str(D) {
    gsub(/./, "\\\\&", D)
    return D
}
```

function nx_escape_str(D) gsub(/./, "

", D) return D

IV nx_escape_str

Escapes all characters in the input string (**D**) by prefixing them with a backslash (‘\‘). Useful for preparing strings for use in regular expressions or other contexts where characters need escaping.

D: The input string whose characters will be escaped.

Basic string escaping

```
D = "hello"
result = nx_escape_str(D)
# Result: "\h\e\l\l\o"
```

D = "hello" result = nx_escape_str(D) Result: "łłø"

Escaping special characters

```
D = "hello$world"
result = nx_escape_str(D)
# Result: "\h\e\l\l\o\$w\o\r\l\d"
```

D = "hello$world" result = nx_escape_str(D) Result : "$\w\Sigma$

Empty string case

```
D = ""
result = nx_escape_str(D)
```

```
# Result: ""
```

D = "" result = nx_escape_str(D) Result: ""

IV nx_join_str

nx_join_str(D1, D2, S, D3)

```
function nx_join_str(D1, D2, S, D3) {
    if (length(D3))
        D3 = __nx_if(D3, "\x27", "\x22")
    if (D1 && D2)
        D1 = D1 S
    return D1 D3 D2 D3
}
```

function nx_join_str(D1, D2, S, D3) if (length(D3)) D3 = __nx_if(D3, "27", "22") if (D1 D2) D1 = D1 S return D1 D3 D2 D3

IV nx_join_str

Joins two strings (**D1** and **D2**) with a separator (**S**) while enclosing **D2** in dynamic quotes (’ / ") based on the length and value of **D3**. Returns the concatenated result.

- **D1**: The first string to be concatenated.
- **D2**: The second string to be concatenated. Enclosed in quotes dynamically.
- **S**: The separator placed between **D1** and **D2**.
- **D3**: Determines the type of quotes (’ / ") to enclose **D2**. Quotes are applied if **D3** has length.

Basic concatenation

```
D1 = "hello"
D2 = "world"
S = " "
D3 = "\""
result = nx_join_str(D1, D2, S, D3)
# Result: hello "world"
```

D1 = "hello" D2 = "world" S = " " D3 = """ result = nx_join_str(D1, D2, S, D3) Result: hello "world"

Single quotes for D2

```
D1 = "key"
D2 = "value"
S = "="
```

```
D3 = "'"
result = nx_join_str(D1, D2, S, D3)
# Result: key='value'
```

D1 = "key" D2 = "value" S = "=" D3 = "’" result = nx_join_str(D1, D2, S, D3) Result: key=’value’

No separator or D1

```
D1 = ""
D2 = "standalone"
S = ""
D3 = "\""
result = nx_join_str(D1, D2, S, D3)
# Result: "standalone"
```

D1 = "" D2 = "standalone" S = "" D3 = """ result = nx_join_str(D1, D2, S, D3) Result: "standalone"

No quotes for D2

```
D1 = "start"
D2 = "end"
S = "-"
D3 = ""
result = nx_join_str(D1, D2, S, D3)
# Result: start-end
```

D1 = "start" D2 = "end" S = "-" D3 = "" result = nx_join_str(D1, D2, S, D3) Result: start-end

IV nx_slice_str

nx_slice_str(D, N, B1, B2)

```
function nx_slice_str(D, N, B1, B2, s, e, l) {
    if (__nx_is_natural(N) && N <= (l = length(D))) {
        if (B1) {
            if (B2) {
                s = N + 1
                e = l - N
            } else {
                s = 1
                e = N
            }
        } else if (length(B1) && N * 2 <= l) {
            if (B2) {
                return substr(D, 1, N) substr(D, l - N + 1)
            } else {
                s = N + 1
                e = l - N * 2
            }
        } else {
            if (B2) {
                s = 1
                e = l - N
            } else {
                s = l - N + 1
                e = N
            }
        }
        return substr(D, s, e)
    }
}
```

function nx_slice_str(D, N, B1, B2, s, e, l) if (__nx_is_natural(N) N <= (l = length(D))) if (B1) if (B2) s = N + 1 e = l - N else s = 1 e = N else if (length(B1) N * 2 <= l) if (B2) return substr(D, 1, N) substr(D, l - N + 1) else s = N + 1 e = l - N * 2 else if (B2) s = 1 e = l - N else s = l - N + 1 e = N return substr(D, s, e)

IV nx_slice_str

Slices a string (**D**) based on position (**N**), constraints (**B1** and **B2**), and calculated start and end indices. Handles nuanced edge cases, including zero and empty values. Returns the sliced substring or a composite of two substrings.

- **D**: The input string to be sliced.
- **N**: The position used for slicing, which must be a natural number.
- **B1**: The first condition influencing slicing logic.
- **B2**: The second condition influencing slicing logic, which affects start and end indices.

Basic slicing

```
print nx_slice_str("abcdefghij", 3, 0, 0)
# Result: "defg" (starts from index 4 and captures 4 characters)
```

print nx_slice_str("abcdefghij", 3, 0, 0) Result: "defg" (starts from index 4 and captures 4 characters)

Slicing with constraints (B1 and B2 both true)

```
print nx_slice_str("abcdefghij", 3, 1, 1)
# Result: "defghij" (starts from index 4 and captures the rest of the string)
```

print nx_slice_str("abcdefghij", 3, 1, 1) Result: "defghij" (starts from index 4 and captures the rest of the string)

Composite slicing

```
print nx_slice_str("abcdefghij", 3, 1, 0)
# Result: "abc" (captures the first N characters of the string)
```

print nx_slice_str("abcdefghij", 3, 1, 0) Result: "abc" (captures the first N characters of the string)

Edge case: B1 empty

```
print nx_slice_str("abcdefghij", 5, "", 0)
# Result: "fghij" (starts from index 6 and captures the rest of the string)
```

print nx_slice_str("abcdefghij", 5, "", 0) Result: "fghij" (starts from index 6 and captures the rest of the string)

Special case: N equals 0

```
print nx_slice_str("abcdefghij", 0, 1, 1)
# Result: "" (returns an empty string as N is 0)
```

print nx_slice_str("abcdefghij", 0, 1, 1) Result: "" (returns an empty string as N is 0)

IV nx_trim_str

nx_trim_str(D, S)

```
function nx_trim_str(D, S) {
    S = __nx_else(S, " \v\t\n\f")
    gsub("(^[" S "]+|[ " S "]+$)", "", D)
    return D
}
```

function nx_trim_str(D, S) S = __nx_else(S, " ˇ ") gsub("([" S"] + |[" S"]+)", "", D) return D

IV nx_trim_str

Trims unwanted characters (**S**) from both ends of a string (**D**). Defaults to whitespace characters if **S** is not provided. Returns the trimmed string.

- **D**: The input string to be trimmed.
- **S**: A set of characters to be removed from both ends. Defaults to common whitespace characters (‘, , , ’).

Trim default whitespace

```
print nx_trim_str("   hello world    ", "")
# Result: "hello world"
```

print nx_trim_str(" hello world ", "") Result: "hello world"

Trim custom characters

```
print nx_trim_str("__hello__", "_")
# Result: "hello"
```

print nx_trim_str("__hello__", "_") Result: "hello"

Trim multiple custom characters

```
print nx_trim_str("-*-hello-*-", "-*")
# Result: "hello"
```

print nx_trim_str("-*-hello-*-", "-*") Result: "hello"

Trim multiple custom characters

```
print nx_trim_str("-*-hello-*-", "-*")
# Result: "hello"
```

print nx_trim_str("-*-hello-*-", "-*") Result: "hello"

No characters to trim

```
print nx_trim_str("hello", "")
# Result: "hello"
```

print nx_trim_str("hello", "") Result: "hello"

No characters to trim

```
print nx_trim_str("hello", "")
# Result: "hello"
```

print nx_trim_str("hello", "") Result: "hello"

Empty string

```
print nx_trim_str("", "")
# Result: ""
```

print nx_trim_str("", "") Result: ""

IV nx_append_str

nx_append_str(D, N, s)

```
function nx_append_str(D, N, s) {
    if (__nx_is_natural(N) && __nx_defined(D, 1)) {
        do {
            s = s D
        } while (--N)
        return s
    }
}
```

function nx_append_str(D, N, s) if (__nx_is_natural(N) __nx_defined(D, 1)) do s = s D while (–N) return s

IV nx_append_str

Appends the string (**D**) to itself **N** times, prepending an optional initial string (**s**) at the start. Returns the concatenated result.

- **D**: The string to be appended multiple times.
- **N**: The number of times to append **D**. Must be a natural number.
- **s**: An optional initial prefix to include at the start of the resulting string.

IV nx_append_str

nx_append_str(D1, N, D2, B)

```
function nx_append_str(D1, N, D2, B, s) {
    if (__nx_is_natural(N) && __nx_defined(D1, 1)) {
        if (D2 != "")
            s = D2
        do {
            if (B)
                s = D1 s
            else
                s = s D1
        } while (--N)
        return s
    }
}
```

function nx_append_str(D1, N, D2, B, s) if (__nx_is_natural(N) __nx_defined(D1, 1)) if (D2 != "") s = D2 do if (B) s = D1 s else s = s D1 while (–N) return s

^ IV nx_append_str

Appends the string (**D1**) to itself **N** times, starting with an optional prefix (**D2**). The append direction is determined by (**B**): if true, appends in reverse; otherwise, appends normally. Returns the concatenated result.

- **D1**: The string to be appended multiple times.
- **N**: The number of times to append **D1**. Must be a natural number.
- **D2**: An optional prefix to include at the start of the concatenated result. Defaults to an empty string.
- **B**: A flag controlling the append direction. If true, appends **D1** in reverse order; otherwise, appends normally.

Normal appending with no prefix

```
print nx_append_str("abc", 3, "", 0)
# Result: "abcabcabc"
```

print nx_append_str("abc", 3, "", 0) Result: "abcabcabc"

Normal appending with a prefix

```
print nx_append_str("xyz", 2, "start-", 0)
# Result: "start-xyzxyz"
```

print nx_append_str("xyz", 2, "start-", 0) Result: "start-xyzxyz"

Reverse appending

```
print nx_append_str("123", 2, "", 1)
# Result: "123123"
```

print nx_append_str("123", 2, "", 1) Result: "123123"

Reverse appending with prefix

```
print nx_append_str("456", 3, "end-", 1)
# Result: "456456456end-"
```

print nx_append_str("456", 3, "end-", 1) Result: "456456456end-"

Edge case: Undefined input (D1 = "")

```
print nx_append_str("", 5, "prefix-", 0)
# Result: (no output) (function returns without doing anything as D1 is not
↪defined)
```

print nx_append_str("", 5, "prefix-", 0) Result: (no output) (function returns without doing anything as D1 is not defined)

Edge case: No appending (N = 0)

```
print nx_append_str("repeat", 0, "prefix-", 0)
# Result: (no output) (function returns without doing anything since N = 0 is
↪not a natural number)
```

print nx_append_str("repeat", 0, "prefix-", 0) Result: (no output) (function returns without doing anything since N = 0 is not a natural number)

IV nx_cut_str

nx_cut_str(D1, D2, B)

```
function nx_cut_str(D1, D2, B) {
    if (match(D1, D2)) {
        if (B)
            return substr(D1, 1, RSTART - 1)
        if (length(B))
            return substr(D1, RSTART + RLENGTH)
        return substr(D1, RSTART, RLENGTH)
    }
}
```

function nx_cut_str(D1, D2, B) if (match(D1, D2)) if (B) return substr(D1, 1, RSTART - 1) if (length(B)) return substr(D1, RSTART + RLENGTH) return substr(D1, RSTART, RLENGTH)

IV nx_cut_str

Extracts or removes a substring matching the pattern (**D2**) from the input string (**D1**). The behavior is controlled by the flag (**B**): if true, removes the substring before the match; if false with a length, removes the substring after the match; otherwise, returns the matched substring.

- **D1**: The input string to be processed.
- **D2**: The pattern to search for within **D1**.
- **B**: A flag or value controlling the substring removal behavior.

Substring before the match

```
print nx_cut_str("abcdefghi", "def", 1)
# Result: "abc"
```

print nx_cut_str("abcdefghi", "def", 1) Result: "abc"

Matched substring

```
print nx_cut_str("abcdefghi", "def", 0)
# Result: "def"
```

print nx_cut_str("abcdefghi", "def", 0) Result: "def"

Substring after the match

```
print nx_cut_str("abcdefghi", "def", "")
# Result: "ghi"
```

print nx_cut_str("abcdefghi", "def", "") Result: "ghi"

No match

```
print nx_cut_str("abcdefghi", "xyz", 0)
# Result: (no output)
```

print nx_cut_str("abcdefghi", "xyz", 0) Result: (no output)

- **D**: The input string to be transformed to title case.
- **B1**: A flag to handle escaped characters during segmentation.
- **B2**: Additional mapping information for pair segmentation logic.