Posix-Nexus AWK

Canine-Table

April 4, 2025

Contents

I Algorithms

I Algorithms

The following functions provide robust implementations of key algorithms, including sorting, partitioning, and randomization, enabling efficient data processing and manipulation.

- **__pivot(L, R)**: Selects a random pivot index within the range [**L**, **R**], ensuring valid integer bounds.
- **__swap(V, DA, DB)**: Swaps the elements at indices **DA** and **DB** within array **V**.
- **__hoares_partition(V, L, R, B, M)**: Partitions array **V** using Hoare's algorithm, comparing elements based on pivot value and mode **M**, with flags for direction **B**.
- **entropy(P)**: Generates a random alphanumeric string of length **P**, converts it from base-62 to base-10, and returns the result.
- **quick_sort(V, L, R, B, M)**: Implements the QuickSort algorithm to sort array **V** within the range [**L**, **R**], based on comparison direction **B** and mode **M**.

II Boolean Operations

II Boolean Operations

The following functions provide utilities for logical and comparative operations, enabling versatile Boolean checks across various conditions.

- **NOT__(D)**: Returns the logical NOT of D.
- **NULL__(D)**: Returns the logical NOT of D (equivalent to **NOT__**).
- **FULL__(D)**: Determines whether D is full (non-empty).
- **TRUE__(D, B)**: Returns 1 if D is full or valid based on B.
- **FALSE__(D, B)**: Returns the logical NOT of **TRUE__**.
- **OR__(B1, B2, B3)**: Logical OR operation between B1 and B2 based on B3.
- **NOR__(B1, B2, B3)**: Logical NOR operation, the NOT of **OR__**.
- **AND__(B1, B2, B3)**: Logical AND operation between B1 and B2 based on B3.
- **NAND__(B1, B2, B3)**: Logical NAND operation, the NOT of **AND__**.
- **XOR__(B1, B2, B3)**: Logical XOR operation, true if exactly one of B1 or B2 is true.
- **XNOR__(B1, B2, B3)**: Logical XNOR operation, the NOT of **XOR__**.
- **CMP__(B1, B2, B3, B4)**: Compares B1 and B2 based on conditions B3 and B4.
- **NCMP__(B1, B2, B3, B4)**: Logical NOT of **CMP__**.
- **LOR__(B1, B2, B3, M)**: Logical OR based on modes specified in M.
- **EQ__(B1, B2, B3)**: Determines equality between B1 and B2 based on B3.
- **NEQ__(B1, B2, B3)**: Determines inequality (NOT equal) between B1 and B2 based on B3.
- **IEQ__(B1, B2, B3)**: Case-insensitive equality comparison between B1 and B2.

II Boolean Operations

- **INEQ__(B1, B2, B3)**: Logical NOT of **IEQ__**.
- **GT__(B1, B2, B3)**: Returns `true` if B1 is greater than B2.
- **LT__(B1, B2, B3)**: Returns `true` if B1 is less than B2.
- **LE__(B1, B2, B3)**: Returns `true` if B1 is less than or equal to B2.
- **GE__(B1, B2, B3)**: Returns `true` if B1 is greater than or equal to B2.
- **IN__(V, D, B)**: Determines if D is an element of array V and satisfies **TRUE__**.
- **ORFT__(B1, B2, B3)**: Returns true if B1 is false or B2 is true, based on B3.

NOT__(D)

```
function NOT__(D)
{
    return ! D
}
```

function NOT__(D) return ! D

NULL__(D)

```
function NULL__(D)
{
    return NOT__(D)
}
```

function NULL__(D) return NOT__(D)

FULL__(D)

```
function FULL__(D)
{
    return CMP__(D, "", "", 1)
}
```

function FULL__(D) return CMP__(D, "", "", 1)

TRUE__(D, B)

```
function TRUE__(D, B)
{
    if (B)
        return FULL__(D)
    else if (NOT__(NULL__(D)))
        return 1
    return 0
}
```

function TRUE__(D, B) if (B) return FULL__(D) else if (NOT__(NULL__(D))) return 1 return 0

IN__(V, D, B)

```
function IN__(V, D, B)
{
    return D in V && TRUE__(V[D], B)
}
```

function IN__(V, D, B) return D in V TRUE__(V[D], B)

FALSE__(D, B)

```
function FALSE__(D, B)
{
    return NOT__(TRUE__(D, B))
}
```

function FALSE__(D, B) return NOT__(TRUE__(D, B))

OR__(B1, B2, B3)

```
function OR__(B1, B2, B3)
{
        return TRUE__(B1, B3) || TRUE__(B2, B3)
}
```

function OR__(B1, B2, B3) return TRUE__(B1, B3) || TRUE__(B2, B3)

NOR__(B1, B2, B3)

```
function NOR__(B1, B2, B3)
{
        return NOT__(OR__(B1, B2, B3))
}
```

function NOR__(B1, B2, B3) return NOT__(OR__(B1, B2, B3))

ORFT__(B1, B2, B3)

```
function ORFT__(B1, B2, B3)
{
        # Return the result of OR__ with the negation of B1 and the truth value of B2
        return OR__(FALSE__(B1, B3), TRUE__(B2, B3))
}
```

function ORFT__(B1, B2, B3)   Return the result of OR__ with the negation of B1 and the truth value of B2 return OR__(FALSE__(B1, B3), TRUE__(B2, B3))

AND__(B1, B2, B3)

```
function AND__(B1, B2, B3)
{
        return TRUE__(B1, B3) && TRUE__(B2, B3)
}
```

function AND__(B1, B2, B3) return TRUE__(B1, B3) TRUE__(B2, B3)

NAND__(B1, B2, B3)

```
function NAND__(B1, B2, B3)
{
        return NOT__(AND__(B1, B2, B3))
}
```

function NAND__(B1, B2, B3) return NOT__(AND__(B1, B2, B3))

XOR__(B1, B2, B3)

```
function XOR__(B1, B2, B3)
{
        # Return the result of OR__ with the combination of AND__ and AND__
        return OR__(AND__(TRUE__(B1, B3), FALSE__(B2, B3)),
                    AND__(FALSE__(B1, B3), TRUE__(B2, B3)))
}
```

function XOR__(B1, B2, B3) Return the result of OR__ with the combination of AND__ and AND__
return OR__(AND__(TRUE__(B1, B3), FALSE__(B2, B3)), AND__(FALSE__(B1, B3), TRUE__(B2, B3)))

XNOR__(B1, B2, B3)

```
function XNOR__(B1, B2, B3)
{
        return NOT__(XOR__(B1, B2, B3))
}
```

function XNOR__(B1, B2, B3) return NOT__(XOR__(B1, B2, B3))

CMP__(B1, B2, B3, B4)

```
function CMP__(B1, B2, B3, B4)
{
        # If B3 is true
        if (B3) {
                if (B4)
                        return B1 > B2
                if (length(B4))
                        return B1 ~ B2
                return B1 == B2
        # Else if B3 has a length
        } else if (length(B3)) {
                if (B4)
                        return length(B1) > length(B2)
                if (length(B4))
                        return length(B1) ~ length(B2)
                return length(B1) == length(B2)
        } else if (is_digit(B1, 1) && is_digit(B2, 1)) {
                if (B4)
                        return +B1 > +B2
                if (length(B4))
                        return +B1 ~ +B2
                return +B1 == +B2
        } else {
                if (B4)
                        return "a" B1 > "a" B2
                if (length(B4))
                        return "a" B1 ~ "a" B2
                return "a" B1 == "a" B2
        }
}
```

function CMP__(B1, B2, B3, B4) If B3 is true if (B3) if (B4) return B1 > B2 if (length(B4)) return B1 B2 return B1 == B2 Else if B3 has a length else if (length(B3)) if (B4) return length(B1) > length(B2) if (length(B4)) return length(B1) length(B2) return length(B1) == length(B2) else if (is_digit(B1, 1) is_digit(B2, 1)) if (B4) return +B1 > +B2 if (length(B4)) return +B1 +B2 return +B1 == +B2 else if (B4) return "a" B1 > "a" B2 if (length(B4)) return "a" B1 "a" B2 return "a" B1 == "a" B2

NCMP__(B1, B2, B3, B4)

```
function NCMP__(B1, B2, B3, B4)
{
        return NOT__(CMP__(B1, B2, B3, B4))
}
```

function NCMP__(B1, B2, B3, B4) return NOT__(CMP__(B1, B2, B3, B4))

LOR__(B1, B2, B3, M)

```
function LOR__(B1, B2, B3, M,   t)
{
        # Determine mode based on M pattern: length or default
        if (M ~ /^(l(e(n(g(t(h)?)?)?)?)?)$/) # Regex for 'length'
                t = 0
        else if (M ~ /^(d(e(f(a(u(l(t)?)?)?)?)?)?)$/) # Regex for 'default'
                t = 1
        # Full comparison based on t
        if (FULL__(t)) {
                if (B3)
                        return GT__(B1, B2, t) # Greater than comparison
                else
                        return LT__(B1, B2, t) # Less than comparison
        } else {
                # Check if B1 and B2 are digits or M is 'string'
                if (! (is_digit(B1, 1) && is_digit(B2, 1)) || M ~ /^(s(t(r(i(n(g)?)?)?)?)?)$/)
                        t = "a" # Set t to 'a' for ASCII comparison
                if (B3)
                        return GT__(t B1, t B2) # Concatenate t with B1 and B2, compare
                else
                        return LT__(t B1, t B2)
        }
}
```

function LOR__(B1, B2, B3, M, t) Determine mode based on M pattern: length or default if (M /$^{(}l(e(n(g(t(h)?)?)?)?)?)$/) Regex for 'length' t = 0 else if (M /$^{(}d(e(f(a(u(l(t)?)?)?)?)?)?)$/) Regex for 'default' t = 1 Full comparison based on t if (FULL__(t)) if (B3) return GT__(B1, B2, t) Greater than comparison else return LT__(B1, B2, t) Less than comparison else Check if B1 and B2 are digits or M is 'string' if (! (is_digit(B1, 1) is_digit(B2, 1)) || M /$^{(}s(t(r(i(n(g)?)?)?)?)?)$/) t = "a" Set t to 'a' for ASCII comparison if (B3) return GT__(t B1, t B2) Concatenate t with B1 and B2, compare else return LT__(t B1, t B2)

EQ__(B1, B2, B3)

```
function EQ__(B1, B2, B3)
{
        return CMP__(B1, B2, B3)
}
```

function EQ__(B1, B2, B3) return CMP__(B1, B2, B3)

NEQ__(B1, B2, B3)

```
function NEQ__(B1, B2, B3)
{
        return NCMP__(B1, B2, B3)
}
```

function NEQ__(B1, B2, B3) return NCMP__(B1, B2, B3)

III Math

III Math

The following functions provide tools for primality testing, random prime generation, and efficient computational methods for dealing with large numbers under practical constraints.

- **__trim_precision(N1, N2)**: Trims **N2** to **N1** decimal places, removing trailing zeros in the fractional part.
- **pi(N)**: Returns the value of π with **N** decimal places of precision, using the arctangent function.
- **tau(N)**: Returns the value of τ (2π) with **N** decimal places of precision.
- **remainder(N1, N2)**: Computes the remainder when **N1** is divided by **N2**, handling precision and rounding.
- **fibonacci(N, B)**: Calculates the **N**-th Fibonacci number, optionally displaying intermediate sums when **B** is true.
- **factoral(N, B)**: Computes the factorial of **N**, optionally printing the multiplication steps when **B** is true.
- **absolute(N)**: Returns the absolute value of **N**, converting negative numbers to positive.
- **ceiling(N)**: Returns the smallest integer greater than or equal to **N**, rounding up for non-integer values.
- **round(N)**: Rounds **N** to the nearest integer, using standard rounding rules.
- **distribution(N1, N2, N3)**: Calculates the distribution value for **N3** within the range defined by **N1** and **N2**, rounding up to the nearest integer.
- **euclidean(N1, N2)**: Implements the Euclidean algorithm to compute the greatest common divisor (GCD) of **N1** and **N2**.
- **lcd(N1, N2)**: Calculates the least common multiple (LCM) of **N1** and **N2** using their GCD.
- ⌄ **modulus_range(N1, N2, N3)**: Adjusts **N1** to fit within the range [**N2**, **N3**] using modulus operations.
- **modular_exponentiation(N1, N2, N3)**: Efficiently computes $(N1^{N2}) \bmod N3$ using iterative squaring, with **N3** defaulting to 100000007 if not provided.
- **fermats_little_theorm(N)**: Applies Fermat's Little Theorem to estimate primality for **N** by checking divisibility against small primes.

III Math

- **divisible(N1, N2)**: Determines whether **N1** is divisible by **N2** without a remainder.
- **miller_rabin(N, T, S)**: Performs the Miller-Rabin primality test on **N**, using **T** trials and separating bases with **S**.
- **__load_primes(N, S)**: Loads a set of prime bases for testing **N**, selecting ranges based on the size of **N**, and separating them with **S**.
- **random_prime(N)**: Generates a random prime number with up to **N** digits, defaulting to 8 digits due to POSIX AWK limitations.

III modulus_range

modulus_range(N1, N2, N3)

```awk
# N1: The lower bound of the range
# N2: The upper bound of the range
# N3: The modulus value to adjust the lower bound
function modulus_range(N1, N2, N3)
{
    # If N1 is less than N2, adjust N1 to be within the range [N2, N3]
    if (N1 < N2)
        N1 = N2 + (N1 - N2 + N3) % (N3 - N2 + 1)
    # If N1 is greater than N3, adjust N1 similarly
    else if (N1 > N3)
        N1 = N2 + (N1 - N2) % (N3 - N2 + 1)
    # Return the adjusted value of N1
    return N1
}
```

N1: The lower bound of the range N2: The upper bound of the range N3: The modulus value to adjust the lower bound function modulus_range(N1, N2, N3) If N1 is less than N2, adjust N1 to be within the range [N2, N3] if (N1 < N2) N1 = N2 + (N1 - N2 + N3) If N1 is greater than N3, adjust N1 similarly else if (N1 > N3) N1 = N2 + (N1 - N2) Return the adjusted value of N1 return N1

⌃ III modulus_range

- Ensures N1 stays within the range [N2, N3] using modulus arithmetic.
- If N1 is less than N2, adjusts it upward by computing N2 + (N1 - N2 + N3) % (N3 - N2 + 1).
- If N1 is greater than N3, adjusts it downward by computing N2 + (N1 - N2) % (N3 - N2 + 1).
- Returns the adjusted value of N1, ensuring it remains within bounds.

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.

- **__join_str(D1, D2, S)**: Appends **D2** to **D1**, using the separator **S** if **D1** is not empty.
- **append_str(N, D, B)**: Repeats appending **D** either at the beginning or end, determined by **B**, **N** times.
- **reverse_str(D)**: Reverses the string **D** by rearranging its characters in reverse order.
- **format_str(D1, D2, S, L, R, B)**: Formats **D1** by replacing placeholders in **D2** using delimiters **S**, **L**, and **R**, and optionally removes unmatched placeholders based on **B**.
- **__get_half(D, C, B1, B2)**: Splits **D** at the first occurrence of character **C** and returns either the first or second half based on **B1**, adjusted by **B2**.
- **__first_index(D, V, B)**: Finds the earliest occurrence of any substring in **V** within **D** and returns its position or the length of **D** if no match is found and **B** is set.
- **__load_quote_map(V)**: Initializes **V** with mappings for single (**sq**) and double (**dq**) quotes.
- **__load_str_map(V)**: Populates **V** with character group mappings, such as uppercase, lowercase, digits, and alphanumeric ranges.
- **__load_esc_map(V)**: Sets up **V** with escape character mappings for common whitespace and control characters.
- **__compare_lengths(V, B)**: Compares the lengths of strings in **V** and returns either the maximum or minimum length based on **B**.
- **escape_str(D)**: Escapes all characters in **D** by prefixing them with a backslash.
- **random_str(N, C, S, B)**: Generates a random string of length **N** using character sets defined by **C**, split by **S**, with random device selection influenced by **B**.
- **totitle(D)**: Converts **D** to title case, capitalizing the first letter of each substring and lowering the rest.
- **trim(D, S)**: Trims leading and trailing spaces from **D** and removes excess spaces around the delimiter **S**.

IV Strings

- **match_length(D, B, S, O)**: Filters and sorts substrings in **D** by length based on **B**, joining them with **O**.
- **match_boundary(D1, D2, B, S, O)**: Matches strings in **D2** that start or end with **D1**, based on **B**, and joins them with **O**.
- **match_option(D1, D2, S, O, B1, B2, B3)**: Filters, sorts, and joins strings in **D2** that match **D1** at the start or end based on flags **B2**, **B3**, and optionally clears duplicates with **B1**.
- **even_lengths(V, D1, D2, B)**: Adjusts the lengths of **V[D1]** and **V[D2]** to ensure both are even by trimming the longer one, based on **B**.
- **str_parser(D1, D2)**: Parses a string **D2** based on the format defined in **D1**. Flags and key-value options are extracted, while unrecognized elements are included in the remainder.
- **str_group(D, V)**: Groups strings enclosed in double (‘"‘) or single (‘’‘) quotes within **D** into elements in **V**. Handles spaces and escape sequences.

V Type Validation

V Type Validation

The following functions provide utilities to classify and manipulate numeric inputs in various formats. The examples demonstrate how to use these functions in practice.

- **__is_signed(N)**: Checks if the input number N has a + or - prefix.
- **__get_sign(N)**: Retrieves the sign (+ or -) of N if it is signed.
- **is_integral(N, B)**: Verifies if N is an integer. The parameter B specifies whether to allow a sign prefix.
- **is_signed_integral(N)**: Checks if N is a signed integer.
- **is_float(N, B)**: Determines if N is a floating-point number, with B controlling the allowance of a sign.
- **is_signed_float(N)**: Validates whether N is a signed floating-point number.
- **is_digit(N, B)**: Checks if N is any numeric value (integer or float).
- **is_signed_digit(N)**: Checks if N is a signed numeric value (integer or float).

__is_signed(N)

```
function __is_signed(N)
{
        return N ~ /^([-]|[+])/
}
```

function __is_signed(N) return N $/^{(}[-]||[+])/$

__get_sign(N)

```
function __get_sign(N)
{
        if (__is_signed(N)) {
                return substr(N, 1, 1)
        }
}
```

function __get_sign(N) if (__is_signed(N)) return substr(N, 1, 1)

is_integral(N)

```
function is_integral(N, B,      e)
{
    if ((B && N ~ /^([-]|[+])?[0-9]+$/) || (! B && N ~ /^[0-9]+$/))
                e = 1
        return e
}
```

function is_integral(N, B, e) if ((B N /^([−]|[+])?[0 − 9]+/) || (! B N /^[0 − 9]+/)) e = 1 return e

is_signed_integral(N)

```
function is_signed_integral(N,          e)
{
        if (__is_signed(N) && is_integral(N, 1))
                e = 1
        return e
}
```

function is_signed_integral(N, e) if (__is_signed(N) is_integral(N, 1)) e = 1 return e

is_float(N, B)

```
function is_float(N, B,         e)
{
        if ((B && N ~ /^([-]|[+])?[0-9]+[.][0-9]+$/) || (! B && N ~ /^[0-9]+[.][0-9]+$/))
                e = 1
        return e
}
```

function is_float(N, B, e) if ((B N /^([−]|[+])?[0 − 9] + [.][0 − 9]+/) || (! B N /^[0 − 9] + [.][0 − 9]+/)) e = 1 return e

is_signed_float(N)

```
function is_signed_float(N,             e)
{
        if (__is_signed(N) && is_float(N, 1))
                e = 1
        return e
}
```

function is_signed_float(N, e) if (__is_signed(N) is_float(N, 1)) e = 1 return e

is_digit(N, B)

```
function is_digit(N, B,         e)
{
        if (is_integral(N, B) || is_float(N, B))
                e = 1
        return e
}
```

function is_digit(N, B, e) if (is_integral(N, B) || is_float(N, B)) e = 1 return e

is_signed_digit(N)

```
function is_signed_digit(N,     e)
{
        if (__is_signed(N) && is_digit(N, 1))
                e = 1
        return e
}
```

function is_signed_digit(N, e) if (__is_signed(N) is_digit(N, 1)) e = 1 return e

VI Numbers Base

VI Numbers Base

The following functions provide utilities for handling base conversions, number constructions, and validations within a customizable range from base 2 to 62, enabling advanced numerical operations.

- **__load_number_map(V1, N1, V2, N2, N3)**: Loads **N1** into **V1** based on base **N2**, storing attributes like sign, fractional part, and validation against the base range 2-62.
- **__construct_number(V, N, B1, B2, B3)**: Reconstructs a number from **V** using its integer, fractional, and sign components based on flags **B1**, **B2**, and **B3**.
- **__get_base(D, V)**: Determines and validates the base (2-62) of **D**, using **V** if necessary.
- **__set_base(N, V)**: Ensures **N** is a valid base and returns it as an integer, defaulting to 10 if invalid.
- **__base_regex(N, V, B)**: Generates a regex for validating numbers in base **N** using **V**, optionally loading the number, lower, or upper maps based on **B**.
- **__load_num_map(V)**: Initializes **V** with digits 0-9 as the number map.
- **__load_lower_map(V)**: Extends **V** with mappings for digits and lowercase letters for bases 10-35.
- **__load_upper_map(V)**: Extends **V** with mappings for digits and uppercase letters for bases 36-62.
- **__base_logarithm(N1, N2)**: Calculates the logarithm of **N1** with base **N2**.
- **__bit_width(N)**: Computes the number of bits required to represent **N** in binary.
- **__pad_bits(V, N1, N2)**: Pads the integer and fractional parts of **N1** in **V** with zeros to align with the bit width of base **N2**.
- **convert_base(N1, N2, N3, N4)**: Converts **N1** from base **N2** to base **N3**, with optional precision **N4** (default: 64).
- **compliment(N1, N2)**: Calculates the base-**N2** complement of **N1**, adjusting digit values accordingly.
- **base_compliment(N1, N2, N3, N4, D, B)**: Computes the base-**N3** complement of **N1** relative to **N2**, accounting for optional sign **D** and precision **N4**.

VI Numbers Base

- **subtract_base(N1, N2, N3, N4, B)**: Subtracts **N2** from **N1** in base **N3**, using optional flags for precision **N4** and default sign behavior **B**.
- **add_base(N1, N2, N3, N4, B)**: Adds **N1** and **N2** in base **N3**, handling fractional parts and sign alignment, with optional precision **N4** and flag **B**.

VII Internet

VII Internet

The following functions provide essential tools for validating, processing, and manipulating IPv4 and IPv6 addresses, including CIDR handling and format conversions, ensuring compliance with modern networking standards.

- **valid_address(D)**: Validates whether **D** is a properly formatted address, checking for exactly five colons, dots, or hyphens, and ensuring it matches the regex pattern for alphanumeric pairs separated by delimiters.
- **valid_prefix(D, S, B)**: Generates a valid prefix for **D**, ensuring correct separator **S**, length adjustments, and casing based on **B**, and validates the result.
- **l2_type(D)**: Identifies the Layer 2 (L2) address type of **D** (e.g., locally/universally administered, multicast/unicast) based on RFC 7042 standards.
- **expand_ipv6(D)**: Converts a compressed IPv6 address **D** into its expanded format, ensuring all segments are present and properly padded with zeros.
- **truncate_ipv6(D)**: Compresses an expanded IPv6 address **D**, removing leading zeros and replacing consecutive zero segments with "::" for a shorter representation.
- **valid_ipv6(D, B)**: Validates whether **D** is a proper IPv6 address, checking segment ranges and patterns, optionally returning the expanded form if valid.
- **valid_ipv4(D, B)**: Validates whether **D** is a properly formatted IPv4 address, ensuring all octets are within the valid range, and optionally processing CIDR notation if **B** is true.
- **load_inet_map(D, V)**: Loads the address configuration map into **V** based on the IP version (**D**), setting the character, tetra size, segments, and base for IPv4 or IPv6.
- **inet(D)**: Processes and validates an IPv4 or IPv6 address, applies CIDR constraints, adjusts segments accordingly, and returns the formatted result.

VIII Miscellaneous

VIII Miscellaneous

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

- **__return_value(D1, D2)**: Returns D1 if it exists; otherwise, returns D2.
- **__return_if_value(D1, D2, B)**: Combines D1 and D2 based on B and returns the result if D1 exists.
- **__return_else_value(D1, D2, B)**: Returns D2 if D1 satisfies the TRUE__ condition for B.
- **__load_value(V, K, DA, DB)**: Assigns DA to V[K] if DA is not "NULL"; otherwise, assigns DB.
- **__load_delim(V, S, O)**: Sets delimiters in V by assigning S and O to keys "s" and "o" with defaults "," and newline.
- **__load_tag(V, L, R)**: Sets tag delimiters in V by assigning L and R to keys "l" and "r" with defaults "<" and ">".

IX Standard Output

IX Standard Output

The following functions enable efficient management of text formatting and color mapping, providing tools for dynamically adjusting style and appearance in outputs.

- **load_symbols(V)**: Populates the array **V** with symbolic representations for text formatting or display purposes, using predefined mappings.
- **text_style_map(D)**: Maps text style descriptors (**D**) like “bold” or “underlined” to their corresponding numerical codes.
- **color_map(D)**: Maps color names or descriptors (**D**) like “red” or “warning” to their corresponding numerical codes for display purposes.

X Structures

X Structures

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

- **insert_indexed_item(V, D, S, N1, N2, N3)**: Inserts items from **D** into the indexed array **V**, handling split delimiters **S** and optional position adjustments using **N1**, **N2**, and **N3**.

- **unique_indexed_array(D, V, S, O, B)**: Creates a unique indexed array or joined string from **D**, splitting using **S**, joining with **O**, and optionally storing results in **V** based on **B**.

- **remove_indexed_item(V, N1, N2, N3, N4)**: Deletes items from the indexed array **V** based on the range and step defined by **N1**, **N2**, **N3**, and **N4**.

- **__join_array(V, S)**: Joins the elements of array **V** into a string, separated by the delimiter **S**.

- **__join_indexed_array(V, S)**: Joins the indexed elements of array **V** into a string, separated by the delimiter **S**.

- **flip_map(V, D1, D2, D3, S)**: Swaps the values in array **V** for the keys defined in **D3**, using **D1** and **D2** to determine the keys to flip.

- **size(V)**: Calculates and returns the size (number of elements) of the array **V**.

- **is_array(V)**: Checks if **V** is an array and returns 1 if true, 0 otherwise.

- **__is_index(N)**: Validates that **N** is a positive integer and returns it if valid, otherwise returns 0.

- **resize_indexed_hashmap(V, N1, N2, S, D)**: Resizes the indexed hashmap **V** to the target size **N1**, redistributing elements starting from **N2**, joining them with **S**, and filling extra slots with **D** if needed.

- **reverse_indexed_hashmap(V, N1, N2, D, S, O)**: Reverses the indexed hashmap **V** between indices **N1** and **N2**, optionally splitting or joining elements with **S** and **O**.

- **stack(V, M, D, S)**: Implements stack operations (push, pop, peek, isempty) on **V**, using **D** for push and optionally splitting data with **S**.

- **queue(V, M, D1, S, D2)**: Implements queue operations (enqueue, dequeue, isempty, size, resize) on **V**, using **D1** and **D2** for data management and **S** for delimiters.

- **clone_array(V1, V2, B)**: Copies elements from array **V1** to **V2**, either preserving keys (**B** set) or copying values directly.

- **trim_split(D, V, S)**: Splits **D** into array **V**, trimming whitespace around delimiters **S**.

X Structures

- **array(D, V, S)**: Populates the array **V** with unique keys from **D**, splitting elements using delimiter **S**.
- **split_parameters(D, V, S1, S2)**: Splits **D** into key-value pairs based on delimiters **S1** (pair separator) and **S2** (key-value separator), storing the result in **V**.
- **compare_arrays(D1, D2, M, S, O)**: Compares arrays **D1** and **D2** based on mode **M** (left, right, intersect, difference) and combines results using delimiters **S** and **O**.

X insert_indexed_item

insert_indexed_item(V, D, S, N1, N2, N3)

```
function insert_indexed_item(V, D, S, N1, N2, N3, j)
{
    if (TRUE__(D, 1)) {
        if (! is_array(V))
            split("", V, "")
        if (! is_integral(N1))
            N1 = size(V)
        if (LT__(+N2, N1))
            N2 = 0
        if (! __is_index(N3))
            N3 = 1
        j = N1
        for (i = 1; i <= trim_split(D, v, S); i++) {
            if (N2)
                j = modulus_range(j, N1, N2)
            V[j] = v[i]
            j = j + N3
        }
        delete v
        if (N2)
            return modulus_range(j, N1, N2)
        return j
    }
}
```

function insert_indexed_item(V, D, S, N1, N2, N3, j) if (TRUE__(D, 1)) if (! is_array(V)) split("", V, "") if (! is_integral(N1)) N1 = size(V) if (LT__(+N2, N1)) N2 = 0 if (! __is_index(N3)) N3 = 1 j = N1 for (i = 1; i <= trim_split(D, v, S); i++) if (N2) j = modulus_range(j, N1, N2) V[j] = v[i] j = j + N3 delete v if (N2) return modulus_range(j, N1, N2) return j

^ X insert_indexed_item

Validation and Initialization

- **trim_split(D, V, S)**: Splits **D** into array **V**, trimming whitespace around delimiters **S**.