AWK

November 16, 2025

Canine-Table

POSIX Nexus serves as a comprehensive cross-language reference hub that explores the implementation and behavior of POSIX-compliant functionality across a diverse set of programming environments. Built atop the foundational IEEE Portable Operating System Interface (POSIX) standards, this project emphasizes compatibility, portability, and interoperability between operating systems.

Abstract

Contents

I Miscellaneous Module **II**
I Character Classifiers . . . II
I File Presence Checker . . . III
I Environment Variable Normalizer . . . III
I Presence Evaluator . . . IV
I Fallback Selector . . . V
I Conditional Selector . . . V
I Ternary Divergence . . . VI
I Conditional Union . . . VII
I Exclusive Divergence . . . VIII
I Symbolic Comparator . . . X
I Symbolic Equality Evaluator . . . XII
I Associative Pair Swapper . . . XIV
I Filesystem Path Conductor . . . XV
I Unique File Resolver . . . XVII
I Escaped Sequence Matcher . . . XIX
I Next Match Finder . . . XXI
I File Merge Conductor . . . XXIII

II Struct Module **XXVI**
II Bijective Mapper . . . XXVI
II Grid Queue Conductor . . . XXVIII
II Depth-First Traversal Conductor . . . XXX

I Miscellaneous Module

I Character Classifiers

nx_is_space(D)

```awk
function nx_is_space(D) { return D ~ /[ \t\n\f\r\v\b]/ }
```

nx_is_upper(D)

```awk
function nx_is_upper(D) { return D ~ /[A-Z]/ }
```

nx_is_lower(D)

```awk
function nx_is_lower(D) { return D ~ /[a-z]/ }
```

nx_is_alpha(D)

```awk
function nx_is_alpha(D) { return nx_is_lower(D) || nx_is_upper(D) }
```

nx_is_digit(D)

```awk
function nx_is_digit(D) { return D ~ /[0-9]/ }
```

Character Classifiers — The Glyph Filters

- nx_is_space ⇝ Checks for whitespace characters
- nx_is_upper ⇝ Checks for uppercase letters
- nx_is_lower ⇝ Checks for lowercase letters
- nx_is_alpha ⇝ Checks for alphabetic characters
- nx_is_digit ⇝ Checks for numeric digits

I File Presence Checker

nx_is_file(D)

```
function nx_is_file(D)
{
        if ((getline < D) > 0)
                close(D)
        else
                return 0
        return 1
}
```

File Presence Checker — The Path Glyph

- Purpose ⇝ Check if file *D* exists and is readable
- Mechanism ⇝ Attempts `getline`; closes if successful
- Return ⇝ 1 if file is readable, 0 otherwise
- Use Case ⇝ File validation, path probing, or conditional loading

I Environment Variable Normalizer

nx_to_environ(D)

```
function nx_to_environ(D,         m)
{
        D = toupper(nx_trim_str(D))
        gsub(/[ \t]/, "_", D)
```

```
        if (! (m = sub(/^[.]/, "L_", D)))
        if (! (m = sub(/^[*]/, "G_", D)))
        if (! (m = sub(/^[@]/, "NEXUS_", D)))
        if (! (m = sub(/^[%]/, "P_", D)))
              sub(/^[0-9]/, "_\\&", D)
        gsub(/[^0-9A-Z_]/, "", D)
        return D
}
```

Environment Variable Normalizer — The Uppercase Glyph

- **Purpose** ⇝ Normalize string D into a valid environment variable name
- **Trimming** ⇝ Applies `nx_trim_str` and uppercases the result
- **Prefix Rules** ⇝ `.` → L_, `*` → G_, `@` → NEXUS_, `%` → P_
- **Digit Prefix** ⇝ If D starts with digit, prepends underscore
- **Sanitization** ⇝ Removes all non-alphanumeric and non-underscore glyphs
- **Use Case** ⇝ Transforming symbolic identifiers into shell-safe environment keys

I Presence Evaluator

__nx_defined(D, B)

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

Presence Evaluator — The Defined Glyph

- **Purpose** ⇝ Determines whether value D is considered present or truthy
- **Input** ⇝ D: value to check; B: fallback flag
- **Logic** ⇝ Returns true if D is non-empty or if $length(D)$ and B are both true
- **Use Case** ⇝ Used by conditional glyphs to evaluate presence, fallback, or symbolic truth

I Fallback Selector

__nx_else(D1, D2, B)

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

Fallback Selector — The Else Glyph

- **Purpose** ⇝ Returns $D1$ if defined or truthy, otherwise returns fallback $D2$
- **Input** ⇝ $D1$: primary value; $D2$: fallback value; B: optional presence flag
- **Logic** ⇝ Uses `__nx_defined(D1, B)` to determine presence
- **Use Case** ⇝ Used in conditional chains, defaulting logic, or symbolic substitution

I Conditional Selector

__nx_if(B1, D1, D2, B2)

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

Conditional Selector — The If Glyph

- **Purpose** ⇝ Returns $D1$ if condition $B1$ is true or defined, otherwise returns fallback $D2$
- **Input** ⇝ $B1$: primary condition; $D1$: value if true; $D2$: value if false; $B2$: optional presence flag
- **Logic** ⇝ Uses `__nx_defined(B1, B2)` to evaluate symbolic truth
- **Use Case** ⇝ Conditional rendering, symbolic branching, or fallback substitution in AWK emitters

I Ternary Divergence

__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))
}
```

Ternary Divergence — The Elif Glyph

- **Purpose** ⇝ Evaluates a three-way conditional divergence based on symbolic presence and equality
- **Input** ⇝ $B1$, $B2$, $B3$: primary conditions; $B4$, $B5$, $B6$: optional fallback values
- **Fallback** ⇝ Each fallback is resolved via `__nx_else` to normalize symbolic presence
- **Logic** ⇝ Returns true if $\texttt{defined}(B1) = \texttt{defined}(B2)$ and $\texttt{defined}(B3) \neq \texttt{defined}(B1)$
- **Use Case** ⇝ Used in chained conditional branches where symbolic presence and divergence must be tested

I Conditional Union

__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)))
}
```

Conditional Union — The Or Glyph

- **Purpose** ⇝ Evaluates symbolic union of conditions with fallback normalization
- **Input** ⇝ $B1$, $B2$, $B3$: primary conditions; $B4$, $B5$, $B6$: optional fallback values
- **Fallback** ⇝ Each fallback is resolved via `__nx_else` to normalize symbolic presence
- **Logic** ⇝ Returns true if `defined(B1)` and `defined(B2)` or `defined(B3)` and not `defined(B1)`
- **Use Case** ⇝ Used in conditional branching, symbolic union, or fallback-aware logic overlays

Logical OR with both primary values defined

- **Inputs** ⇝ Left: `"yes"`, Right: `"ok"`
- **Operation** ⇝ Both values are defined
- **Expected Result** ⇝ Returns `true`

```
__nx_or("yes", "ok", "", "", "", "")
```

Logical OR with only fallback defined

- Inputs ⇝ Fallback: `"fallback"`, Primaries: `""`
- Operation ⇝ Evaluates fallback since primaries are undefined
- Expected Result ⇝ Returns `true`

```
__nx_or("", "", "fallback", "", "", "")
```

Logical OR with all values undefined

- Inputs ⇝ All arguments empty
- Operation ⇝ No defined values to satisfy OR condition
- Expected Result ⇝ Returns `false`

```
__nx_or("", "", "", "", "", "")
```

I Exclusive Divergence

__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)))
}
```

Exclusive Divergence — The Xor Glyph

- **Purpose** ⇝ Returns true if exactly one of the two symbolic conditions is defined or truthy
- **Input** ⇝ $B1$, $B2$: primary conditions; $B3$, $B4$: optional fallback values
- **Fallback** ⇝ $B4$ is resolved via `__nx_else(B4, B3)`
- **Logic** ⇝ Returns true if one is defined and the other is not
- **Use Case** ⇝ Used in symbolic branching, exclusive logic, or markup divergence testing

Exclusive-or with only the first value defined

- **Inputs** ⇝ Left: `"yes"`, Right: `""`
- **Operation** ⇝ One side defined, the other undefined
- **Expected Result** ⇝ Returns `true`

```
__nx_xor("yes", "", "", "")
```

Exclusive-or with both values defined

- **Inputs** ⇝ Left: `"yes"`, Right: `"ok"`
- **Operation** ⇝ Both sides defined
- **Expected Result** ⇝ Returns `false`

```
__nx_xor("yes", "ok", "", "")
```

Exclusive-or with only the fallback defined

- Inputs ⇝ Fallback: `"fallback"`, Others: `""`
- Operation ⇝ One side defined via fallback, the other undefined
- Expected Result ⇝ Returns `true`

```
__nx_xor("", "", "fallback", "")
```

Exclusive-or with only the first value defined

- Inputs ⇝ Left: `"yes"`, Right: `""`
- Operation ⇝ One side defined, the other undefined
- Expected Result ⇝ Returns `true`

```
__nx_xor("yes", "", "", "")
```

I Symbolic Comparator

__nx_compare(B1, B2, B3, B4)

```
function __nx_compare(B1, B2, B3, B4)
{
    if (! B3) {
        if (length(B3)) {
            B1 = length(B1)
            B2 = length(B2)
        } else if (nx_digit(B1, 1) && nx_digit(B2, 1)) {
            B1 = +B1
            B2 = +B2
        } else {
            B1 = "a" B1
            B2 = "a" B2
        }
        B3 = 1
    }
    if (B4)
        return __nx_if(nx_digit(B4), B1 > B2, B1 < B2) ||
                __nx_if(__nx_else(nx_digit(B4) == 1, tolower(B4) ==
↪"i"), B1 == B2, 0)
    if (length(B4))
        return B1 ~ B2
```

```
    return B1 == B2
}
```

Symbolic Comparator — The Compare Glyph

- **Purpose** ⇝ Compares two values $B1$ and $B2$ using symbolic or numeric logic
- **Input** ⇝ $B1$, $B2$: values to compare; $B3$: mode flag; $B4$: comparison operator
- **Mode** ⇝ If $B3$ is empty, auto-selects: length, numeric, or string coercion
- **Operator** ⇝ If $B4$ is set: `digit` → numeric compare, `i` → equality, else regex match
- **Use Case** ⇝ Used in symbolic evaluation, numeric comparison, or pattern matching logic

Compare two numeric strings using digit-based operator

- **Inputs** ⇝ Left: `"5"`, Right: `"10"`, Mode: `""`, Operator: `"1"`
- **Operation** ⇝ Numeric comparison with coercion via digit detection
- **Expected Result** ⇝ Returns `true` since `5 < 10`

```
__nx_compare("5", "10", "", "1")
```

Match string against pattern using regex operator

- **Inputs** ⇝ Left: `"abc"`, Operator: `" "`, Right: `"b"`
- **Mode** ⇝ Regex match against pattern `"b"`
- **Expected Result** ⇝ Returns `true` since `"abc"` contains `"b"`

```
__nx_equality("abc", "~", "b")
```

Compare two strings using alphabetic equality operator

- Inputs ⇝ Left: `"abc"`, Operator: `"=a"`, Right: `"abc"`
- Mode ⇝ Alphabetic equality check
- Expected Result ⇝ Returns `true` since both strings are equal

```
__nx_equality("abc", "=a", "abc")
```

I Symbolic Equality Evaluator

__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)
}
```

Symbolic Equality Evaluator — The Equality Glyph

- **Purpose** ⇝ Evaluates symbolic equality or comparison between $B1$ and $B3$ using operator $B2$
- **Input** ⇝ $B1$: left value; $B2$: operator string; $B3$: right value
- **Operators** ⇝ Supports >, <, =, with optional suffixes (a, _, etc.)
- **Logic** ⇝ Delegates to `__nx_compare` with inferred mode and operator
- **Use Case** ⇝ Used in symbolic evaluation, conditional logic, or markup comparison overlays

Compare two numeric strings using digit-based operator

- **Inputs** ⇝ Left: `"5"`, Operator: `">0"`, Right: `"3"`
- **Mode** ⇝ Numeric comparison using digit coercion
- **Expected Result** ⇝ Returns `true` since `5 > 3`

```
__nx_equality("5", ">0", "3")
```

Match string against pattern using regex operator

- **Inputs** ⇝ Left: `"abc"`, Operator: `"~"`, Right: `"b"`
- **Mode** ⇝ Regex match against pattern `"b"`
- **Expected Result** ⇝ Returns `true` since `"abc"` contains `"b"`

```
__nx_equality("abc", "~", "b")
```

Compare two strings using alphabetic equality operator

- **Inputs** ⇝ Left: `"abc"`, Operator: `"=a"`, Right: `"abc"`
- **Mode** ⇝ Alphabetic equality check
- **Expected Result** ⇝ Returns `true` since both strings are equal

```
__nx_equality("abc", "=a", "abc")
```

I Associative Pair Swapper

__nx_swap(V, D1, D2, t)

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

Associative Pair Swapper — The Swap Glyph

- **Purpose** ⇝ Swaps the values of keys $D1$ and $D2$ in associative array V
- **Input** ⇝ V: associative array; $D1$, $D2$: keys to swap
- **Mechanism** ⇝ Temporarily stores $V[D1]$, then performs the exchange
- **Use Case** ⇝ Used in sorting, reordering, or symbolic mutation of key-value pairs

Swapping values between two keys in an associative array

- **Initial State** ⇝ `V["a"] = 1, V["b"] = 2`
- **Operation** ⇝ Swap the values of keys `"a"` and `"b"`
- **Expected Result** ⇝ `V["a"] = 2, V["b"] = 1`

```
V["a"] = 1
V["b"] = 2
```

```
__nx_swap(V, "a", "b")
```

I Filesystem Path Conductor

nx_file_path(D1, B, D2)

```
function nx_file_path(D1, B, D2,    i, j)
{
    D2 = __nx_else(D2, "/")
    if (! sub(/^-/, ENVIRON["OLDPWD"], D1))
    if (! sub(/^~/, ENVIRON["HOME"], D1))
    if (! sub(/^NX_L:/, ENVIRON["NEXUS_LIB"], D1))
    if (! sub(/^NX_C:/, ENVIRON["NEXUS_CNF"], D1))
    if (! sub(/^NX_D:/, ENVIRON["NEXUS_DOCS"], D1))
    if (! sub(/^NX_E:/, ENVIRON["NEXUS_ENV"], D1))
    if (! sub(/^NX_SB:/, ENVIRON["NEXUS_SBIN"], D1))
    if (! sub(/^NX_B:/, ENVIRON["NEXUS_BIN"], D1))
    if (! sub(/^NX_J:/, ENVIRON["NEXUS_LIB"] "java" D2
↪ENVIRON["G_NEX_JAVA_PROJECT"], D1))
        sub(/^NX_S:/, ENVIRON["NEXUS_SRC"], D1)
    gsub(D2 "+", D2, D1)
    gsub(D2 "+$", "", D1)
    i = D1
    if (! sub("[^" D2 "]+$", "", i))
        return D1
    i = length(i)
    j = length(D2)
    if (B == "")
        return substr(D1, i + j)
    if (B == 0)
        return D1
    return substr(D1, 1, i - j)
}
```

Filesystem Path Conductor — The Path Glyph

- **Purpose** ⇝ Resolves symbolic prefixes and returns basename, dirname, or full path
- **Input** ⇝ $D1$: path string; B: toggle; $D2$: separator (default "/")
- **Prefixes** ⇝ Symbols like `NX_C:`, `NX_L:`, etc. expand to environment variables. **Note:** The actual expansion depends on your runtime environment. For example, `NX_C:` may expand to `/opt/posix-nexus/cnf` on one system, but to a different directory elsewhere.
- **Mechanism** ⇝ Expands environment variables, normalizes separators, slices basename/dirname
- **Use Case** ⇝ Used in config resolution, daemon overlays, or IPC path normalization

Return full expanded path when toggle is 0

- **Input** ⇝ `"NX_C:/file7.txt"`, Toggle: 0
- **Expansion** ⇝ Prefix `NX_C:` → environment variable `NEXUS_CNF`
- **Expected Result** ⇝ `$NEXUS_CNF/file7.txt`

```
print nx_file_path("NX_C:/file7.txt", 0)
```

Return basename when toggle is empty

- **Input** ⇝ `"NX_C:/file7.txt"`
- **Expansion** ⇝ Prefix `NX_C:` → environment variable `NEXUS_CNF`
- **Expected Result** ⇝ `file7.txt`

```
print nx_file_path("NX_C:/file7.txt")
```

Return dirname when toggle is non-empty

- Input ⇝ `"NX_C:/file7.txt"`, Toggle: 1
- Expansion ⇝ Prefix NX_C: → environment variable NEXUS_CNF
- Expected Result ⇝ $NEXUS_CNF

```
print nx_file_path("NX_C:/file7.txt", 1)
```

I Unique File Resolver

nx_uniq_file(D1, D2, V, D3)

```
function nx_uniq_file(D1, D2, V, D3,    b, r, d, i)
{
    b = nx_file_path(D1)
    D3 = __nx_else(D3, "/")
    r = D2 D3 D1
    if (nx_is_file(D1)) {
        r = D1
        d = nx_file_path(D1, 1)
    } else if (nx_is_file(r)) {
        d = nx_file_path(r, 1)
    } else {
        for (i = -1; i >= V[-0]; --i) {
            r = V[i] D2 b
            if (nx_is_file(r)) {
                d = nx_file_path(r, 1)
                break
            }
        }
    }
    if (d != "") {
        if (! (d in V))
            V[--V[-0]] = d
        if (! (r in V))
            V[++V[0]] = r
    }
}
```

Unique File Resolver — The Uniqueness Glyph

- **Purpose** ⇝ Ensures that only unique absolute file paths and their directories are tracked in hashmap V
- **Input** ⇝ $D1$: relative/absolute file path; $D2$: base directory; V: hashmap; $D3$: separator
- **Mechanism** ⇝ Checks if file exists directly, via base directory, or by searching known directories in V
- **Indexes** ⇝ $[-0]$: list of unique directories; $[0]$: list of unique file realpaths
- **Use Case** ⇝ Used in config loaders, IPC daemons, or overlay systems to avoid duplicate path entries

Resolve absolute path from relative input

- **Input** ⇝ `D1 = "file7.txt", D2 = "/opt/posix-nexus/cnf"`
- **Check** ⇝ Concatenate base directory with file → `/opt/posix-nexus/cnf/file7.txt`
- **Expected Result** ⇝ Adds directory `/opt/posix-nexus/cnf` to `V[-0]` and file path to `V[0]`

```
nx_uniq_file("file7.txt", "/opt/posix-nexus/cnf", V)
```

Handle already absolute path

- **Input** ⇝ `D1 = "/etc/nexus/restic.json"`
- **Check** ⇝ File exists directly, no need to prepend base directory
- **Expected Result** ⇝ Adds directory `/etc/nexus` to `V[-0]` and file path to `V[0]`

```
nx_uniq_file("/etc/nexus/restic.json", "/opt/posix-nexus/cnf", V)
```

Search known directories when file not found directly

- Input ⇝ `D1 = "config.json", D2 = "/opt/posix-nexus/cnf"`
- Check ⇝ Iterates over `V[-0]` directories to locate file
- Expected Result ⇝ Once found, adds directory and file path uniquely to V

```
nx_uniq_file("config.json", "/opt/posix-nexus/cnf", V)
```

I Escaped Sequence Matcher

nx_nesc_match(D1, D2, D3)

```
function nx_nesc_match(D1, D2, D3,    f, l)
{
    if (D1 == "")
        return -1
    f = 0
    D2 = __nx_else(D2, " ")
    if ((D3 = __nx_else(D3, "\\\\", 1)) == "\\\\")
        l = 1
    else
        l = length(D3)
    while (match(D1, D2)) {
        f = f + RSTART
        if (! (match(substr(D1, 1, RSTART - 1), D3 "+$") && D3) ||
↪int(RLENGTH % 2) == 0)
            break
        f = f + RLENGTH - l
        D1 = substr(D1, f + 1)
    }
    return f
}
```

Escaped Sequence Matcher — The Escape Glyph

- **Purpose** ⇝ Finds the position of a delimiter in string $D1$, respecting escape sequences
- **Input** ⇝ $D1$: target string; $D2$: delimiter regex (default space); $D3$: escape character regex (default backslash)
- **Mechanism** ⇝ Iteratively searches for delimiter, checks if it is escaped by $D3$, and advances accordingly
- **Return** ⇝ Index of first unescaped delimiter, or -1 if $D1$ is empty
- **Use Case** ⇝ Used in tokenization, argument parsing, or IPC relay parsing where escapes must be honored

Find first unescaped space in a string

- **Input** ⇝ `"foo bar"`
- **Delimiter** ⇝ Space
- **Escape** ⇝ Backslash
- **Expected Result** ⇝ Returns 4, the position of the space

```
print nx_nesc_match("foo bar")
```

Skip escaped space

- **Input** ⇝ `"foo bar baz"`
- **Delimiter** ⇝ Space
- **Escape** ⇝ Backslash
- **Expected Result** ⇝ Returns 9, the position of the unescaped space after `"bar"`

```
print nx_nesc_match("foo\\ bar baz")
```

Handle double escape (escaped backslash before space)

- Input ⇝ `"foo\\\\ bar"`
- Delimiter ⇝ Space
- Escape ⇝ Backslash
- Expected Result ⇝ Returns 5, since the space is not escaped (two backslashes cancel)

```
print nx_nesc_match("foo\\\\ bar")
```

I Next Match Finder

nx_find_next(D1, V, B, D2)

```
function nx_find_next(D1, V, B, D2,    i, f, m)
{
    if (D1 == "")
        return -1
    B = __nx_if(B, ">0", "<0")
    for (i in V) {
        m = nx_nesc_match(D1, V[i], D2)
        if (! f || __nx_equality(m, B, f))
            f = m
    }
    if (f != length(D1) && B == ">0")
        return f + 1
    return f
}i
```

Next Match Finder — The Match Glyph

- **Purpose** ⇝ Finds the next delimiter match in string $D1$ across a set of candidate patterns V
- **Input** ⇝ $D1$: target string; V: array of delimiter regexes; B: comparison toggle; $D2$: escape character regex
- **Mechanism** ⇝ Iterates over all patterns in V, uses `nx_nesc_match` to locate matches, compares positions with `__nx_equality`
- **Comparison** ⇝ If B is empty, defaults to `">0"` (find next greater); otherwise `"<0"`
- **Return** ⇝ Index of the next matching delimiter, or -1 if $D1$ is empty
- **Use Case** ⇝ Used in tokenization, parsing overlays, or IPC relays where multiple delimiters may apply

Find next space or comma in a string

- **Input** ⇝ `"foo, bar baz"`
- **Delimiters** ⇝ `V[1] = ",", V[2] = " "`
- **Escape** ⇝ Default backslash
- **Expected Result** ⇝ Returns 4, the position of the comma

```
V[1] = ","
V[2] = " "
print nx_find_next("foo, bar baz", V)
```

Skip escaped delimiter

- Input ⇝ `"foo\,bar baz"`
- Delimiters ⇝ `V[1] = ",", V[2] = " "`
- Escape ⇝ Backslash
- Expected Result ⇝ Returns 8, the position of the space after `"bar"`

```
V[1] = ","
V[2] = " "
print nx_find_next("foo\\,bar baz", V)
```

Handle multiple delimiters with comparison toggle

- Input ⇝ `"alpha|beta gamma"`
- Delimiters ⇝ `V[1] = "|", V[2] = " "`
- Toggle ⇝ Default `">0"` ensures smallest index chosen
- Expected Result ⇝ Returns 6, the position of the space after `"beta"`

```
V[1] = "|"
V[2] = " "
print nx_find_next("alpha|beta gamma", V)
```

I File Merge Conductor

nx_file_merge(D1, D2, D3, D4)

```
function nx_file_merge(D1, D2, D3, D4,  stk, fls, trk)
{
    if (nx_uniq_file(D1, "", fls) != 1)
        return -1

    D4 = __nx_else(D4, "include", 1)

    # directive name
    trk["dir"] = "nx_" D4

    # directive sigil
```

```
12      trk["sig"] = __nx_else(D2, "#", 1)
13
14      # omit files if listed after directive
15      if (D2 = nx_trim_split(D3, stk, "<nx:null/>")) {
16          do {
17              nx_uniq_file(que[D2], fls[fls["-0"]], fls)
18          } while (--D2 > 0)
19          split("", trk, "")
20      }
21
22      stk["rt"] = "."
23
24      do {
25          while ((getline D2 < D1) > 0) {
26              if (D2 ~ "([ \t]+|^)" trk["sig"] trk["dir"] &&
↪match(D2, trk["sig"] trk["dir"] "[ \t]+")) {
27                  trk["cr"] = substr(D2, 1, RSTART - 1)
28                  D2 = substr(D2, RSTART + RLENGTH)
29
30                  if (match(D2, /^[^ \t]+/)) {
31                      if (nx_uniq_file(substr(D2, RSTART, RLENGTH),
↪fls[fls["-0"]], fls) != -1) {
32                          __nx_file_merge_push(stk, trk["cr"])
33                          trk[++trk[0]] = fls[fls[0]]
34                          trk[fls[fls[0]]] = stk["rt"] ""
↪++stk[stk["rt"] "0"] "."
35                          if ((trk["cr"] = substr(D2, RSTART +
↪RLENGTH)) !~ /^[ \t]*$/)
36                              trk["cr"] = trk["cr"] "\n"
37                          __nx_file_merge_push(stk, trk["cr"])
38                      } else {
39                          __nx_file_merge_push(stk, trk["cr"]
↪substr(D2, RSTART + RLENGTH) "\n")
40                      }
41                  } else if (trk["cr"] !~ /^[ \t]*$/) {
42                      __nx_file_merge_push(stk, trk["cr"] "\n")
43                  }
44              } else if (D2 !~ /^[ \t]*$/) {
45                  __nx_file_merge_push(stk, D2 "\n")
46              }
47          }
48          close(D1)
49
50          D1 = trk[trk[0]]
51          stk["rt"] = trk[trk[trk[0]]]
52      } while (trk[0]-- > 0)
53
54      delete trk
55      delete fls
56      nx_dfs(stk)
57      for (D2 = 1; D2 <= stk[0]; D2++)
58          printf(stk[stk[D2]])
59      delete stk
60  }
```

File Merge Conductor — The Merge Glyph

- **Purpose** ⇝ Processes include directives in a file, merging referenced files into a unified output
- **Input** ⇝ $D1$: root file; $D2$: directive sigil (default #); $D3$: omit list; $D4$: directive keyword (default "include")
- **Mechanism** ⇝ Scans lines, detects directives, resolves file paths via `nx_uniq_file`, pushes content into stack, recursively merges
- **Helpers** ⇝ Uses `__nx_file_merge_push` and `__nx_file_merge_rt` to manage stack indices
- **Return** ⇝ Prints merged file content to stdout
- **Use Case** ⇝ Used in config overlays or script loaders where nested includes must be resolved safely

Merge file with include directive

- **Root** ⇝ `main.conf` contains line `#nx_include extra.conf`
- **Operation** ⇝ Directive detected, `extra.conf` merged inline
- **Expected Result** ⇝ Unified output with contents of both files

```
nx_file_merge("main.conf")
```

Handle directive with non-existent file

- **Root** ⇝ `main.conf` contains `#nx_include missing.conf`
- **Operation** ⇝ Directive match but file not found
- **Expected Result** ⇝ Line preserved without directive expansion

```
nx_file_merge("main.conf")
```

Omit files listed in exclusion list

- Root ⇝ `main.conf` with omit list `"extra.conf"`
- Operation ⇝ Directive found but file excluded
- Expected Result ⇝ File skipped, only root content printed

```
nx_file_merge("main.conf", "#", "extra.conf")
```

II Struct Module

II Bijective Mapper

nx_bijective(V, D1, D2, D3)

```
function nx_bijective(V, D1, D2, D3)
{
    if (D1 == "")
        return -1
    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]
    }
}
```

Bijective Mapper — The Mapping Glyph

- **Purpose** ⇝ Creates or mutates bijective relationships between keys in associative array V
- **Input** ⇝ V: associative array; $D1, D2, D3$: keys/values to link
- **Mechanism** ⇝ If three arguments are given, forms a cycle $D1 \to D2 \to D3 \to D1$. If two arguments, forms a symmetric pair $D1 \leftrightarrow D2$. If only $D3$ is provided, mutates the existing mapping of $D1$ to point to $D3$
- **Return** ⇝ No explicit return; modifies array V in place
- **Use Case** ⇝ Used in overlay systems to guarantee reversible mappings and cyclic relationships between symbolic keys

Create symmetric pair mapping

- **Input** ⇝ `D1 = "a", D2 = "b"`
- **Operation** ⇝ Sets `V["a"] = "b"` and `V["b"] = "a"`
- **Expected Result** ⇝ a $\leftrightarrow$ b bijection established

```
nx_bijective(V, "a", "b")
```

Create cyclic triple mapping

- **Input** ⇝ `D1 = "x", D2 = "y", D3 = "z"`
- **Operation** ⇝ Sets x $\to$ y, y $\to$ z, z $\to$ x
- **Expected Result** ⇝ Cycle established among three keys

```
nx_bijective(V, "x", "y", "z")
```

Mutate existing mapping

- Initial State ⇝ `V["a"] = "b", V["b"] = "a"`
- Input ⇝ `D1 = "a", D2 = "", D3 = "c"`
- Operation ⇝ Reassigns `V["b"] = "c"`, optionally deletes `V["a"]`
- Expected Result ⇝ Mapping updated: `b → c`

```
nx_bijective(V, "a", "", "c")
```

II Grid Queue Conductor

nx_grid(V, D, N)

```
function nx_grid(V, D, N)
{
    if (D) {
        if (! (0 in V && "|" in V && "-" in V)) {
            V[0] = 1
            V["|"] = 1
            V["-"] = 1
        }
        if ((N = __nx_else(nx_natural(nx_digit(N, 1)), V[0])) <
↪V["-"])
            N = V["-"]
        while (V[0] < N) {
            if (! (++V[0] in V))
                V[V[0]] = 0
        }
        V[N "," ++V[N]] = D
    } else if (N) {
        while (! V[V[0]] && V["-"] <= V[0])
            delete V[V[0]--]
        if (V["-"] <= V[0]) {
            N = V[V[0] "," V[V[0]]]
            if (D == "")
                delete V[V[0] "," V[V[0]]--]
            return N
        }
    } else {
        while (V[V["-"]] < V["|"] && V["-"] <= V[0]) {
            delete V[V["-"]++]
            V["|"] = 1
        }
        if (V["-"] <= V[0]) {
            N = V[V["-"] "," V["|"]]
            if (D == "")
```

```
                delete V[V["-"] "," V["|"]++]
            return N
        }
    }
}
```

Grid Queue Conductor — The Grid Glyph

- **Purpose** ⇝ Implements a grid/queue structure inside associative array V
- **Input** ⇝ V: associative array; D: data element; N: index or control
- **Mechanism** ⇝ Initializes grid if empty, appends data into indexed slots, retrieves or deletes entries depending on arguments
- **Indexes** ⇝ $[0]$: highest row index; $[" - "]$: lowest active row; $["|"]$: column pointer
- **Return** ⇝ When retrieving, returns the stored element at current grid position
- **Use Case** ⇝ Used for queueing, scheduling, or grid-like storage overlays in IPC or config daemons

Insert element into grid at next slot

- **Initial State** ⇝ Empty grid
- **Operation** ⇝ Insert `"alpha"`
- **Expected Result** ⇝ Stored at 1 , 1; grid initialized with $[0] = 1, [" - "] = 1, ["|"] = 1$

```
nx_grid(V, "alpha")
```

Retrieve last inserted element

- State ⇝ Grid contains `"alpha"` at 1 , 1
- Operation ⇝ Call with N=1
- Expected Result ⇝ Returns `"alpha"`

```
print nx_grid(V, "", 1)
```

Iterate forward through grid

- State ⇝ Grid contains multiple entries
- Operation ⇝ Call with no D, no N
- Expected Result ⇝ Returns next element at current [" − "], ["|"] position, advancing column pointer

```
print nx_grid(V)
```

II Depth-First Traversal Conductor

nx_dfs(V, B, trk, stk)

```
function nx_dfs(V, B, trk, stk)
{
    if (! (".0" in V && int(V[".0"]) > 0))
        return -1
    stk[++stk[0]] = 1
    stk[++stk[0]] = V[".0"]
    if (B ~ /^[02]$/)
        V[0] = 0
    do {
        for (; stk[1] <= stk[2]; ++stk[1]) {
            trk["ky"] = trk["rt"] "." stk[1]
            if (B ~ /^[12]$/)
                trk["str"] = trk["str"] "<nx:null/>" V[trk["ky"]]
            if (B ~ /^[02]$/)
                nx_bijective(V, ++V[0], 0, trk["ky"])
            if (trk["ky"] ".0" in V && int(V[trk["ky"] ".0"]) > 0) {
                trk["rt"] = trk["ky"]
                stk[++stk[0]] = stk[1]
                stk[++stk[0]] = stk[2]
                stk[1] = 0
```

```
                    stk[2] = V[trk["rt"] ".0"]
                }
            }
            if (stk[1] > 0 && sub(/[^.]+$/, "", trk["rt"])) {
                sub(/[.]$/, "", trk["rt"])
                stk[2] = stk[stk[0]--]
                stk[1] = stk[stk[0]--] + 1
            }
        } while (stk[0] > 2 || stk[1] <= stk[2])
        if (B ~ /^[12]$/)
            B = substr(trk["str"], 11)
        else
            B = ""
        delete trk
        delete stk
        return B
}
```

Depth-First Traversal Conductor — The DFS Glyph

- **Purpose** ⇝ Performs depth-first traversal over a nested structure encoded in associative array V
- **Input** ⇝ V: tree-like associative array; B: mode selector; trk: traversal record; stk: stack
- **Mechanism** ⇝ Initializes stack with root bounds, iterates children, records traversal string or bijective mapping depending on mode
- **Modes** ⇝ B=0: bijective mapping of traversal order; B=1: build traversal string; B=2: both
- **Return** ⇝ Traversal string if mode includes string building, else empty
- **Use Case** ⇝ Used in overlay systems to walk nested JSON-like structures, build linearized strings, or establish bijective index mappings

Traverse tree and build string

- State ⇝ V[".0"]=2, V["1"]="alpha", V["2"]="beta"
- Mode ⇝ B=1 → string building
- Expected Result ⇝ Returns concatenated string "alpha beta" with <nx:null/> markers

```
print nx_dfs(V, 1, trk, stk)
```

Traverse tree and build bijective mapping

- State ⇝ Nested keys with .0 counts
- Mode ⇝ B=0 → bijective mapping
- Expected Result ⇝ Populates V with bijective index ↔ key mappings for traversal order

```
nx_dfs(V, 0, trk, stk)
```

Traverse tree with both string and mapping

- State ⇝ Tree with multiple children
- Mode ⇝ B=2 → both string and mapping
- Expected Result ⇝ Returns traversal string and updates bijective mapping simultaneously

```
print nx_dfs(V, 2, trk, stk)
```