BOVNAR · V1.1

Bovnar — EBNF Grammar

Bovnar (BVNR) v1.1 documentation · EBNF Grammar · 2026-06-21

```
(* ================================================================= *)
(*  BOVNAR  –  EBNF derived from parser/lexer/validator implementation *)
(*  Version 1.1           Notation: ISO/IEC 14977:1996                *)
(*    { }  = zero or more       [ ]  = optional                       *)
(*    ( )  = grouping           |  = alternation                      *)
(*    ,    = concatenation      ;  = rule terminator                  *)
(*    "…"  = literal terminal   (*…*) = comment                       *)
(*  Updated: 2026-06-17 – drop the residual "1.1 (draft)" label (spec   *)
(*    1.1 is released); note datetime also accepts a width parameter;    *)
(*    reframe base-unit as a lookup into the full 163-unit / 487-spelling *)
(*    table (NOT the closed SI subset it previously claimed).             *)
(*  2026-06-16 – consistency pass vs the implementation:               *)
(*    allow ws/comments before the assignment ";" and the "/" array    *)
(*    separator; move the inline unit to a value-only scalar-with-unit  *)
(*    rule so array elements are correctly excluded; bound \u{…} to 1–6 *)
(*    hex digits and version-int to no-leading-zero; inline type-spec;  *)
(*    split the array-homogeneity / duplicate-key constraints into the  *)
(*    LEXER/VALIDATOR tier (the streaming reader) vs the DOM-BUILDER     *)
(*    tier (only a DOM parse rejects ragged siblings, mixed kinds, and  *)
(*    duplicate keys; the streaming reader accepts them).               *)
(*  2026-06-15 – spec 1.1 (additive over the frozen 1.0):              *)
(*    version directive (#!bovnar M.N), datetime type family,         *)
(*    \xHH / \u{…} string escapes, reference array indexing.          *)
(*  Earlier (2026-05-29): null/true/false/on/off keywords, bool       *)
(*    type family, special numbers respelled nan / inf / ninf      *)
(* ================================================================= *)


(* ──────────────────────────────────────────────────────────────────  *)
(*  1. Top-level stream                                                *)
(* ──────────────────────────────────────────────────────────────────  *)

(* Valid end-of-stream states (lexer states where EOF is accepted):   *)
(*   undefined       – nothing has been read yet (empty stream)       *)
(*   first_comment_intro – inside a leading comment that runs to EOF  *)
(*                     with no trailing newline (comment-only stream) *)
(*   first_bom       – BOM or first-line comment (its newline seen),  *)
(*                     no assignment yet                              *)
(*   value_outro     – the semicolon of the last assignment consumed  *)
(* In all four cases struct_nesting_level must be zero.               *)

(* Events emitted to the validator callbacks (bvnr_event_t):         *)
(*                                                                    *)
(*   ev_stream_start  – once at the very start of bvn_lex_run   *)
(*   ev_stream_end    – once at the very end of bvn_lex_run,          *)
(*                      after the last value's terminator is consumed *)
(*   ev_assignment_start   – on the identifier token of an assignment *)
(*   ev_struct_start       – on "{"                                   *)
(*   ev_struct_end         – on "}"                                   *)
(*   ev_array_row_start      – on "[" opening a bracket row             *)
(*   ev_array_row_end      – on "]" closing a bracket row             *)
(*   ev_array_dim_start        – on "/" between rows; fired immediately   *)
(*                           when "/" is consumed, before the next    *)
(*                           "[" opens the next row                  *)
(*   ev_octet_stream_start – on the NUL byte that enters binary mode  *)
(*   ev_octet_stream_end   – on the 0x00 tag ending an octet stream   *)
(*                                                                    *)
(*   For every typed value (including values whose type annotation is  *)
(*   synthesised by the validator – see §4) the following sequence    *)
(*   is emitted before ev_data:                                       *)
(*                                                                    *)
(*     ev_type_annotation_start             – once, before params    *)
(*     ev_type_annotation_type_family       – the family name        *)
(*     ev_type_annotation_type_family_parameter  – zero to three      *)
(*                       times, once per present parameter (width,     *)
(*                       base, unit), in that order                    *)
```

```
(*     ev_type_annotation_end               – once, after params     *)
(*     ev_data                              – the value token        *)
(*                                                                   *)
(*   ev_type_annotation_start is emitted to on_unverified first      *)
(*   (raw bytes only), then to on_verified (with value_type and      *)
(*   value_unit filled in).  All other type-annotation events and    *)
(*   ev_data are emitted to both callbacks simultaneously.           *)
(*   For bool keywords (true/false/on/off) the same default type-    *)
(*   annotation sequence is emitted, synthesising <bool> when no      *)
(*   explicit annotation was given (no width/base/unit params).       *)
(*                                                                   *)
(*   ev_data is also emitted for null values, symbols, references,   *)
(*   and octet-stream chunks without a preceding type-annotation      *)
(*   sequence. Struct values do NOT produce ev_data; they are         *)
(*   signaled by ev_struct_start and ev_struct_end.                  *)

(* Only blank characters — NOT comments — may precede the version      *)
(* directive: it is recognised solely as the very first comment, so an  *)
(* earlier comment would demote it to an ordinary one (see below).      *)
stream       = [utf8-bom] , {ws-char} , [version-directive] , ws ,
               {assignment , ws} ;

(* UTF-8 BOM: legal only as the very first three bytes of the stream. *)
utf8-bom     = "\xEF\xBB\xBF" ;

(* Version directive (spec 1.1): an OPTIONAL leading comment of the    *)
(* form "#!bovnar <major>.<minor>" declaring the spec version the      *)
(* document targets. It is recognised only as the very first comment   *)
(* (after the optional BOM and whitespace); a "#!bovnar …" on any       *)
(* later line is an ordinary comment. Lexically it IS a comment, so a   *)
(* spec-1.0 reader skips it transparently — the declaration is purely   *)
(* additive. A 1.1+ reader parses it, exposes the version, and (in      *)
(* strict mode) rejects a version it does not support. Each component   *)
(* is a decimal integer with no leading zero (a bare "0" excepted).     *)
(* A malformed version after the "!bovnar" prefix is                    *)
(* error_invalid_spec_version.                                          *)
(* The directive accepts ONLY horizontal whitespace (HT or SP) around   *)
(* its components — not the LF/CR/VT/FF that "ws-char" also covers.      *)
version-directive = "#" , "!bovnar" , hspace , {hspace} ,
                    version-int , "." , version-int ,
                    {hspace} , ("\x0D" | "\x0A" | (* eof *)) ;
hspace       = "\x09" (* HT *) | "\x20" (* SP *) ;
(* version-int: a decimal integer with no leading zero (a bare "0" is the *)
(* only form starting with "0"); each component must fit in a uint16.     *)
version-int  = "0"
             | ( ("1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9") , {DIGIT} ) ;

(* Whitespace: blank characters and comments, freely intermixed.     *)
ws           = {ws-char | comment} ;

(* At least one whitespace character, followed by optional further    *)
(* whitespace and comments.  Used as the mandatory separator before   *)
(* an inline unit suffix.                                             *)
ws-mandatory = ws-char , {ws-char | comment} ;
ws-char      = "\x09"             (* HT  *)
             | "\x0A"             (* LF  *)
             | "\x0B"             (* VT  *)
             | "\x0C"             (* FF  *)
             | "\x0D"             (* CR  *)
             | "\x20" ;           (* SP  *)

(* Comment: "#" through the next LF, CR, or end of input.            *)
(* Control bytes 0x00–0x08, 0x0E–0x1F, and 0x7F are hard errors     *)
(* inside a comment body.                                            *)
comment      = "#" , {comment-char} , ("\x0D" | "\x0A" | (* eof *)) ;
comment-char = "\x09"                              (* HT            *)
             | "\x0B" | "\x0C"                     (* VT, FF        *)
```

```
             | printable-byte                          (* 0x20–0x7E      *)
             | high-byte ;                             (* 0x80–0xFF      *)


(* ──────────────────────────────────────────────────────────────────  *)
(*  2. Assignments                                                      *)
(* ──────────────────────────────────────────────────────────────────  *)

(* The leading "." is a syntactic sigil; it is not stored in the key. *)
(* At least one id-start character after "." is required;             *)
(* ".=" is error_empty_identifier.                                    *)
(* Comments are permitted between the last key character and "=".     *)

assignment   = "." , key , ws , "=" , ws , value , ws , ";" ;

key          = id-start , {id-body-char} ;

(* First character of an identifier, symbol, or reference segment:   *)
(* A–Z, a–z, "_", and UTF-8 leader bytes 0xC3–0xF4.                  *)
(* 0xC2 is explicitly rejected (ACT_NONE) at all token-start         *)
(* positions, so code points U+0080–U+00BF cannot begin these tokens.*)
id-start     = ALPHA | "_" | utf8-leader ;

(* Body characters of an identifier:                                  *)
(* Accepted: "+", "-", digits, alpha, "_",                            *)
(*   UTF-8 leader bytes 0xC3–0xF4, UTF-8 continuation bytes 0x80–0xBF.*)
(* Hard-error bytes: 0x00–0x1F, 0x7F, 0xC2, and ASCII punctuation:  *)
(*   " # , . / ; < > [ ] { } = ! $ % & ' ( ) * : ? @ \ ^ ` | ~    *)
(* "=" terminates the body (ACT_value_intro).                         *)
(* Whitespace terminates the body (ACT_to_identifier_outro), after    *)
(* which only "=" is accepted (with comments permitted between).      *)
id-body-char = "+" | "-"
             | DIGIT
             | ALPHA
             | "_"
             | utf8-leader         (* 0xC3–0xF4 *)
             | utf8-continuation ; (* 0x80–0xBF *)


(* ──────────────────────────────────────────────────────────────────  *)
(*  3. Values                                                           *)
(* ──────────────────────────────────────────────────────────────────  *)

(* A value is the right-hand side of an assignment or an element      *)
(* inside an array row; the optional type annotation applies in both  *)
(* contexts.                                                          *)
(*                                                                    *)
(* Typed null elements: in array context, a type annotation not       *)
(* followed by any raw value token (i.e. the next byte is "," "]"    *)
(* or ";") is a null value carrying the annotated type.  This is      *)
(* handled by ACT_type_null_then_new_array_value,                     *)
(* ACT_type_null_then_array_outro, and ACT_type_null_then_value_outro  *)
(* from the type_outro state.                                         *)

value        = [type-annotation , ws] , ( scalar-with-unit | raw-value ) ;

(* A bare number or string scalar may carry an inline unit suffix — but ONLY  *)
(* at value position, never inside an array element (array-elem uses raw-      *)
(* value, which has no unit option).  The lexer reaches the inline-unit state  *)
(* only from number_outro / string_outro, so no other value form takes one.    *)
scalar-with-unit = ( number | string ) , ws-mandatory , inline-unit ;

raw-value    = null-value
             | bool-value
             | special-number
             | datetime-literal              (* spec 1.1 *)
             | number
```

```
              | string
              | symbol
              | reference
              | array
              | struct
              | octet-stream ;

(* datetime-literal (spec 1.1): an ISO-8601 UTC timestamp accepted in     *)
(* place of the integer epoch-seconds carrier.  The lexer accumulates the *)
(* shape (dtlit_* states, entered from a digit run followed by "-"); the  *)
(* validator strictly range-checks the fields and converts the UTC civil  *)
(* instant to the integer carrier on the declared epoch (unix when a bare *)
(* literal infers <datetime:64,unix>).  Whole-second carrier: a frac-     *)
(* tional part (any digit count) does not change the value but is kept    *)
(* verbatim and re-emitted so the literal round-trips.  A "+HH:MM" /      *)
(* "-HH:MM" offset is folded to true UTC before conversion.              *)
(* The atomic GNSS epochs (gps/galileo/glonass/beidou) reject a literal.  *)
datetime-literal = year , "-" , month , "-" , day ,
                   [ "T" , hour , ":" , minute , ":" , second ,
                     [ "." , dt-fraction ] , [ dt-zone ] ] ;
dt-fraction  = digit , { digit } ;          (* kept verbatim; carrier is whole seconds *)
dt-zone      = "Z" | ( "+" | "-" ) , hour , ":" , minute ;  (* tz offset *)
year         = digit , digit , digit , digit ;
month        = digit , digit ;   (* 01..12                                 *)
day          = digit , digit ;   (* 01..28/29/30/31 per month              *)
hour         = digit , digit ;   (* 00..23                                 *)
minute       = digit , digit ;   (* 00..59                                 *)
second       = digit , digit ;   (* 00..60 (60 = UTC leap second)          *)

(* null-value: either the reserved keyword "null" or the empty        *)
(* production.                                                        *)
(* At assignment level:  ".foo = ;" and ".foo = null;" are equivalent. *)
(* In array context: a leading or trailing comma, two consecutive     *)
(* commas, a bare "null", or a type annotation immediately followed   *)
(* by "," / "]" all denote a null element.                            *)
null-value   = "null" | (* empty *) ;

(* bool-value: a reserved keyword in value position.  "on" is an      *)
(* alias for "true" and "off" for "false"; the validator collapses    *)
(* all four spellings to a vt_bool value (canonical text "true" /     *)
(* "false") and synthesises a <bool> type annotation when none is     *)
(* given (see §4).  An explicit <bool> annotation accepts only these  *)
(* keywords as its value.                                             *)
(*                                                                    *)
(* Lexically these keywords — together with "null" — are ordinary     *)
(* symbol tokens (§7); the validator reserves the exact words and     *)
(* reclassifies them.  A longer word that merely starts with one      *)
(* (e.g. "truthy", "nullable") remains a plain symbol.                *)
bool-value   = "true" | "false" | "on" | "off" ;


(* ------------------------------------------------------------------ *)
(*  4. Type annotations                                               *)
(* ------------------------------------------------------------------ *)

(* Lexer: a keyword state machine (tf_* states) recognises the eight  *)
(* family names via six keyword paths (float_fix / float_dec share    *)
(* the "float" prefix; see §4 below).  After the family keyword the   *)
(* raw bytes of the ":…"                                              *)
(* parameter string are accumulated into type_data by copy_type_byte. *)
(*                                                                    *)
(* Validator: bvn_parse_type_annotation parses the accumulated string *)
(* and the validator emits the structured type-annotation event       *)
(* sequence described in §1.                                          *)
(*                                                                    *)
(* Default synthesis: when a number or string value carries no        *)
(* explicit type annotation (value_type is still vt_plain), the       *)
```

```
(* validator synthesises a default before emitting ev_data:           *)
(*   strings / special-numbers → <utf8>  /  <float:64,_10,no_unit>   *)
(*   floats (has "." or "e"/"E") → <float:64,_10,no_unit>            *)
(*   negative integers           → <sint:64,_10,no_unit>             *)
(*   plain integers              → <uint:64,_10,no_unit>             *)
(*   bool keywords (true/false/on/off) → <bool>                      *)
(* The full type-annotation event sequence is always emitted for      *)
(* number and string values, whether annotated explicitly or not.     *)

type-annotation = "<" , ws , param-type , ws , ">" ;

(* Types with an optional parameter list after ":".                   *)
(* Parameters are identified by class and may appear in any order;    *)
(* at most one of each class (width, base, q, unit) is permitted.     *)
(*                                                                    *)
(* "utf8" goes to type_body_outro after the keyword; the lexer        *)
(* accepts ":" and the parameter bytes, but utf8 is a parameterless    *)
(* family — any width, base, q, or unit parameter is rejected by       *)
(* bvn_parse_type_annotation as error_illegal_value_type (see bool).   *)
(*                                                                    *)
(* "float_fix" and "float_dec" are not separate keyword paths in the  *)
(* lexer state table.  The lexer fires ACT_tf_float_done after the    *)
(* shared "float" prefix (states tf_f → tf_fl → tf_flo → tf_floa     *)
(* → ACT_tf_float_done), which stores "float" in type_data and        *)
(* transitions to type_body_outro.  The subsequent bytes "_fix" or    *)
(* "_dec" are accumulated via copy_type_byte.  The final string        *)
(* ("float", "float_fix", or "float_dec") is dispatched in           *)
(* bvn_parse_type_annotation.                                         *)
(* "bool" and "utf8" are parameterless families: any width, base, q,  *)
(* or unit parameter is error_illegal_value_type.  bool's only valid   *)
(* values are the bool keywords (see bool-value, §3).                 *)
(* "datetime" (spec 1.1) is a timestamp: a signed integer seconds count    *)
(* validated like sint.  It accepts an optional width parameter (as sint)   *)
(* plus one epoch-name parameter (unix|tai|gps|mjd|ntp|galileo|glonass|     *)
(* y2000|beidou; default unix), e.g. <datetime:64,unix>.  A numeric base,   *)
(* q, or unit parameter is error_illegal_value_type.  Available only when   *)
(* the document declares "#!bovnar 1.1" or newer.                           *)
param-type   = ("uint" | "sint" | "float" | "float_fix" | "float_dec" | "utf8"
                | "bool" | "datetime")
             , [ws , ":" , ws , type-param-list] ;

(* The parameter list is accumulated byte-by-byte into type_data by   *)
(* the copy_type_byte / type_body_outro states, and later re-parsed   *)
(* by bvn_parse_type_annotation.                                      *)
(*                                                                    *)
(* Bytes accepted in the accumulated string:                          *)
(*   A–Z, a–z, 0–9, "*", "+", ",", ".", "/", "-", ":", "^", "_"      *)
(*   UTF-8 continuation bytes 0x80–0xBF                               *)
(*   UTF-8 leader bytes 0xC2–0xF4  (0xC2 IS accepted here, unlike     *)
(*   identifier / symbol / reference token starts)                    *)
(* Whitespace is accepted but not accumulated; it transitions to       *)
(* type_body_outro, which accepts the same bytes as copy_type_byte    *)
(* plus "#" (0x23, entering comment_intro), while treating whitespace *)
(* as ignored rather than re-triggering the outro transition.         *)
(* ">" always terminates the annotation.                              *)
type-param-list = type-param , {ws , "," , ws , type-param} ;

type-param   = width-param          (* e.g. 32, 64                      *)
             | base-param           (* e.g. _16, _64, _85               *)
             | q-param              (* e.g. q8, q16  — float_fix only    *)
             | unit-param ;         (* e.g. m~s, Gi~B, m^2              *)

(* Width: one or more decimal digits.  0 means "no width restriction".*)
(* float:      0, 16, or any multiple of 32 up to BVN_FLOAT_MAX_PREC.*)
(* float_fix,  *)
(* float_dec:  0, 16, 32, 64, 128, 256.                               *)
width-param  = DIGIT , {DIGIT} ;
```

```
(* Base: underscore followed by one or more decimal digits.           *)
(* Valid values: 2–62, 64 (standard Base64), 85 (Ascii85).           *)
(* Bases 64 and 85 are uint-only: their alphabets use '+'/'-' as       *)
(*   digits, leaving no sign character, so sint+64/85 is illegal.      *)
(* "float" accepts only base 10 (or absent, → 10) or base 16;        *)
(*   other bases → error_illegal_value_type.                          *)
(* "float_fix" and "float_dec" reject the base parameter entirely:   *)
(*   any base-param → error_illegal_value_type.                       *)
(* A non-decimal base with uint/sint should use a quoted string value; *)
(* a bare literal is not rejected by the validator but will be parsed  *)
(* as a symbol token, causing a type/value mismatch instead.          *)
base-param   = "_" , DIGIT , {DIGIT} ;

(* Q~parameter: lowercase "q" followed by one or more decimal digits. *)
(* Specifies the number of fractional bits for the float_fix Q-format.*)
(*   Value = raw_signed_integer × 2^(-Q).                             *)
(* Valid only for "float_fix"; rejected for all other families.       *)
(* Valid range: 0 ≤ Q < effective_width (width 0 → effective 64).    *)
(* Q is stored in value_type_spec_t.base and retrieved via            *)
(* bvn_effective_q; bvn_effective_base always returns 10 for          *)
(* float_fix and float_dec.                                           *)
q-param      = "q" , DIGIT , {DIGIT} ;

(* Unit parameter: everything up to the next "," or ">".             *)
(* Parsed against the unit table by bvn_parse_unit.              *)
(* "no_unit" is a reserved keyword meaning explicitly dimensionless.  *)
(* Valid for: uint, sint, float, float_fix, float_dec.               *)
unit-param   = unit-char , {unit-char} ;
unit-char    = (* any byte accepted by copy_type_byte that is not     *)
               (* "," (0x2C) or ">" (0x3E):                          *)
               (* A–Z, a–z, 0–9, "*", ".", "/", "-", ":", "^", "_",  *)
               (* "~", "$", "(", ")", or a UTF-8 multi-byte byte     *)
               (* (0x80–0xF4)                                         *) ;

(* ── Unit sub-grammar (semantic, enforced by bvn_parse_unit) ─── *)
(*                                                                    *)
(* resolved-unit = "no_unit" | unit-expr ;                           *)
(* unit-expr     = unit-factor , {unit-sep , unit-factor} ;          *)
(* unit-factor   = unit-component | "(" , unit-expr , ")" ;          *)
(*                                                                    *)
(* unit-sep      = "*"                        (* product – U+002A  *) *)
(*               | "·"                        (* product – U+00B7  *) *)
(*               | "/" ;                      (* division           *) *)
(*   "·" is the middle dot, encoded as 0xC2 0xB7 (UTF-8 for U+00B7). *)
(*   "*" and "·" are semantically equivalent (multiplication).       *)
(*                                                                    *)
(* Semantics of "/": the first "/" switches every subsequent factor   *)
(* into the denominator (its stored exponents are negated). A "(...)"  *)
(* group is a sub-expression evaluated independently; when it falls in  *)
(* the denominator its net exponents are negated as a whole, so       *)
(* "k~g/(m·s²)" = kg·m⁻¹·s⁻² and "(k~g/m)·s²" = kg·m⁻¹·s². Additional   *)
(* "/" separators at one level never toggle back to the numerator.    *)
(* An explicit separator is required before a group ("m·(s)", not     *)
(* "m(s)"); a group is not (yet) followed by its own exponent; paren   *)
(* nesting is bounded at 16.                                          *)
(*                                                                    *)
(* Empty components or groups (e.g. "m//s", "()", "m*·s"), unmatched  *)
(* parentheses, and more than BVNR_MAX_UNIT_COMPONENTS = 8 total      *)
(* components per unit string all produce error_unit_illegal.         *)
(*                                                                    *)
(* unit-component = [prefix "~"] base-unit [unit-exponent] ;         *)
(*                                                                    *)
(* When a prefix is present the separator "~" is mandatory.          *)
(* A bare base-unit with no prefix requires no separator.            *)
(*                                                                    *)
(* si-prefix  = "Q"|"R"|"Y"|"Z"|"E"|"P"|"T"|"G"|"M"|"k"|"h"|"da"   *)
```

```
(*            | "d"|"c"|"m"|"µ"|"n"|"p"|"f"|"a"|"z"|"y"|"r"|"q" ;  *)
(*   (µ = U+00B5: 0xC2 0xB5; ASCII "u" is an accepted alias)         *)
(*                                                                    *)
(* iec-prefix = "Ki"|"Mi"|"Gi"|"Ti"|"Pi"|"Ei"|"Zi"|"Yi"|"Ri"|"Qi" ; *)
(*   Ri (2^90) and Qi (2^100) were added in IEC 80000-13:2022 ed. 2. *)
(*                                                                    *)
(* base-unit  = a symbol looked up in the fixed unit table by              *)
(*   bvn_parse_unit (src/utils/bovnar_si_units.c): 163 named units with     *)
(*   487 accepted spellings (canonical symbols plus long-form and plural    *)
(*   aliases), spanning SI, imperial / US customary, CGS, radiation,        *)
(*   surveying, culinary, Old German and digital-storage quantities.        *)
(*   Doc 2 (the unit-system spec) is the authoritative registry; the SI     *)
(*   coherent / common subset below is REPRESENTATIVE only, NOT a closed    *)
(*   alternation:                                                           *)
(*     b B s m g A K mol cd Hz N Pa J W V Ω F C S Wb T H lm lx Bq Gy Sv     *)
(*     kat rad sr L l min h d wk yr °C ° degC degrC degrees degree degr     *)
(*     deg t bar eV Da au ha                                                *)
(*   Symbols are case-sensitive and a few spellings collide across roles,   *)
(*   resolved by context: "P" = poise (base unit) vs peta (SI prefix),      *)
(*   "G" = gauss vs giga.  (Ω = U+2126: 0xE2 0x84 0xA6; ° = U+00B0: 0xC2    *)
(*   0xB0)                                                                  *)
(*                                                                      *)
(* unit-exponent = [exp-sign] exp-digit                                 *)
(*               | "^" ["-"|"+"] ASCII-digit ;                          *)
(*                                                                      *)
(* exp-sign  = "⁺" (* U+207A: 0xE2 0x81 0xBA, positive / no-op *)    *)
(*           | "⁻" ; (* U+207B: 0xE2 0x81 0xBB, negate exponent *)   *)
(*                                                                      *)
(* exp-digit = "¹"|"²"|"³"|"⁴"|"⁵"|"⁶"|"⁷"|"⁸"|"⁹" ;              *)
(*   (U+00B9 / 00B2 / 00B3 / 2074–2079)                                 *)
(*                                                                      *)
(* "⁺" is accepted but has no effect (same as absent).                  *)
(* The ASCII form "^[+-]?[1-9]" maps to the same internal constants     *)
(* as the Unicode superscript form. "^0" is not a valid exponent.       *)


(* ──────────────────────────────────────────────────────────────────── *)
(*  5. Numbers                                                          *)
(* ──────────────────────────────────────────────────────────────────── *)

(* The lexer is digit-agnostic: any ASCII digit 0–9 is accepted in a  *)
(* bare number literal without base validation.  Digit validity against*)
(* a non-decimal base is checked later by bvn_acc_parse_number.       *)
(* Leading zeros are not rejected by the lexer ("007" is valid).      *)
(* Only "e"/"E" are accepted as exponent markers in bare literals.    *)
(* "p"/"P" (binary exponent) is recognised by the validator inside a  *)
(* quoted-string number value of a base-16 float, e.g.                *)
(* <float:64,_16> "1.8p+2"; there "e"/"E" are hex mantissa digits, so *)
(* the binary exponent must use "p"/"P" (see spec §6.3).  It is the    *)
(* only context in which "p"/"P" is reachable.                        *)

number       = ["-"] , (int-led | dot-led) , [dec-exponent] ;

(* Integer-led: one or more digits with an optional fractional part.  *)
(* A trailing dot ("123.") without fractional digits is valid.        *)
int-led      = DIGIT , {DIGIT} , ["." , {DIGIT}] ;

(* Dot-led: at least one digit after the dot is required.             *)
(* ".5" and "-.5" are valid; a lone "." is a hard error.              *)
dot-led      = "." , DIGIT , {DIGIT} ;

dec-exponent = ("e" | "E") , ["+" | "-"] , DIGIT , {DIGIT} ;

(* Special IEEE-754 values are bare reserved keywords: nan, inf, and  *)
(* ninf (negative infinity).  Like null / true / false / on / off they *)
(* are lexed as ordinary symbols and reclassified to numeric special  *)
(* values by the validator; a bare word that is not one of these exact *)
```

```
(* spellings stays an ordinary symbol (e.g. "infinity", "nans").       *)
(* There is no sigil and no inline-unit suffix — a unit is supplied    *)
(* via the type annotation, e.g. <float:64,m/s> inf.                   *)
special-number = "nan" | "inf" | "ninf" ;


(* ──────────────────────────────────────────────────────────────────  *)
(*  6. Strings                                                         *)
(* ──────────────────────────────────────────────────────────────────  *)

(* Two or more adjacent quoted literals with only whitespace/comments  *)
(* between them are concatenated into a single string token by the     *)
(* lexer (string_outro re-enters string_intro on '"').  The combined   *)
(* byte length is bounded by max_string_length (default UINT16_MAX).   *)

string          = string-literal , {ws , string-literal} ;

string-literal  = '"' , {string-char} , '"' ;

string-char  = safe-string-byte
             | escape-seq ;

(* safe-string-byte: whitespace 0x09–0x0D, 0x20–0x7E except '"' (0x22)*)
(* and '\' (0x5C), plus high bytes 0x80–0xFF.                         *)
(* Control bytes 0x00–0x08, 0x0E–0x1F, and DEL (0x7F) are hard       *)
(* errors even inside strings.                                        *)
safe-string-byte = (* byte in [0x09,0x0D] | [0x20,0x21] | [0x23,0x5B] *)
                   (*          | [0x5D,0x7E] | [0x80,0xFF]             *) ;

(* Recognised escape sequences (from bvn_escape_lut):               *)
escape-seq   = "\" , ( "t"   (* \t = 0x09 HT  *)
                     | "n"   (* \n = 0x0A LF  *)
                     | "v"   (* \v = 0x0B VT  *)
                     | "f"   (* \f = 0x0C FF  *)
                     | "r"   (* \r = 0x0D CR  *)
                     | '"'   (* \"            *)
                     | "\"   (* \\            *)
                     | byte-escape       (* spec 1.1, see below *)
                     | unicode-escape ) ; (* spec 1.1, see below *)
(* All other bytes after "\" are error_illegal_escape_sequence.       *)
(* spec 1.1 only — available when the document declares "#!bovnar 1.1"  *)
(* or newer; in a 1.0/unversioned document "x"/"u" after "\" stay      *)
(* error_illegal_escape_sequence.                                     *)
byte-escape    = "x" , hex-digit , hex-digit ;       (* one byte 0xHH    *)
unicode-escape = "u" , "{" , hex-digit ,
                 [hex-digit] , [hex-digit] , [hex-digit] ,
                 [hex-digit] , [hex-digit] , "}" ;
(* 1 to 6 hex digits naming a Unicode scalar.  A 7th hex digit before "}"  *)
(* (or an empty "{}") is error_illegal_escape_sequence — a structural      *)
(* check made before the value is examined.  A value that is a surrogate   *)
(* (D800..DFFF) or above 10FFFF is error_invalid_codepoint; otherwise the   *)
(* scalar is UTF-8 encoded.  A \x byte must leave the whole string valid    *)
(* UTF-8 (else error_invalid_utf8_byte).                                   *)
hex-digit      = DIGIT | "a".."f" | "A".."F" ;


(* ──────────────────────────────────────────────────────────────────  *)
(*  6.5  Inline unit suffix                                            *)
(* ──────────────────────────────────────────────────────────────────  *)

(* An inline-unit may follow a scalar number or string value (but not *)
(* a special-number keyword — give its unit via the type annotation).  *)
(* At least one whitespace character (ws-mandatory) is the sole        *)
(* required separator between the value literal and the unit.          *)
(* Inline units attach only at value position: the value rule has a    *)
(* scalar-with-unit alternative, while an array element (array-elem)   *)
(* uses raw-value, which has no unit option.  At the action level the  *)
```

```
(* lexer additionally rejects ACT_inline_unit_intro when the           *)
(* in_array_element flag is set (a redundant guard).                   *)
(*                                                                     *)
(* INVALID (no whitespace before the unit):                           *)
(*   .bad = 9.81m/s;     (* no separator at all — hard error *)       *)
(*                                                                     *)
(* Examples:                                                           *)
(*   .a = 9.81 m/s;          (* space separator *)                    *)
(*   .b = <float:64> 1.5 k~m; (* with type annotation *)             *)
(*   .c = <float:64,s> inf;  (* special-number unit via annotation *) *)
(*   .d = 1.5e3 k~J;          (* number with exponent *)              *)
(*   .e = "FF" m;             (* string literal with unit *)          *)
(*                                                                     *)
(* State-machine flow:                                                 *)
(*   1. Whitespace from a number-body state fires ACT_to_number_outro *)
(*      → number_outro.  Strings enter string_outro_nosp after the    *)
(*      closing ""; whitespace transitions to string_outro.           *)
(*      Special numbers (nan, inf, ninf) are lexed as symbols and      *)
(*      take no inline unit (give the unit via the type annotation).   *)
(*   2. From number_outro / string_outro (whitespace already seen):   *)
(*        alpha / "_" / "$" / "%" / "(" / UTF-8 leader (0xC2–0xF4)     *)
(*        → ACT_inline_unit_intro                                      *)
(*   3. ACT_inline_unit_intro pushes the first byte and enters        *)
(*      inline_unit_body directly.                                     *)
(*   4. inline_unit_body accumulates unit characters (including ":"  *)
(*      as a valid mid-unit byte) until whitespace (→                 *)
(*      inline_unit_outro) or ";" (→ value_outro via finalize).      *)
(*                                                                     *)
(*   Inside an array element (in_array_element flag set),             *)
(*   ACT_inline_unit_intro immediately rejects with                    *)
(*   error_unexpected_input_byte.                                      *)
(*                                                                     *)
(* Semantic rule (enforced by validator):                              *)
(*   If a type-annotation unit was also given, the inline unit must    *)
(*   parse to the identical value_unit_t; mismatch → error_unit_mismatch. *)

inline-unit       = inline-unit-start , { inline-unit-char } ;

(* A currency component carries a mandatory "$" sigil (spec 1.0): "$USD",      *)
(* "$BTC", or prefixed "<prefix>~$EUR".  Bare codes are no longer currencies,  *)
(* so a currency code can never collide with a physical-unit symbol.           *)
(* "(" lets an inline unit begin with a parenthesised group, e.g. "(m/s)/s",   *)
(* so the inline form accepts the same unit grammar as a type-annotation unit. *)
inline-unit-start = ALPHA | "_" | "$" | "%" | "(" | unit-lead-byte ;

inline-unit-char  = ALPHA | DIGIT | "_" | "$" | "%" | "+" | "-" | "." | "/" | ":" | "^" |
"*" | "~"
                  | "(" | ")" | unit-lead-byte | utf8-continuation ;

(* unit-lead-byte: the UTF-8 lead-byte range accepted inside a unit — the    *)
(* full 0xC2-0xF4, BROADER than utf8-leader (0xC3-0xF4).  Unlike identifier, *)
(* symbol, and reference token starts (where 0xC2 is rejected, see §12), a   *)
(* unit may begin with or contain a 0xC2-led code point: the µ prefix        *)
(* (U+00B5 = 0xC2 0xB5), the ° / °C units (U+00B0 = 0xC2 0xB0), and the      *)
(* "·" product separator (U+00B7 = 0xC2 0xB7).  The lexer maps the whole     *)
(* 0xC2-0xF4 range to ACT_inline_unit_intro / ACT_copy_inline_unit_byte,     *)
(* matching copy_type_byte in a type-annotation unit body.                   *)
unit-lead-byte    = (* byte in [0xC2, 0xF4] *) ;


(* ──────────────────────────────────────────────────────────────────  *)
(*  7. Symbols  (unquoted bare-word values in value position)         *)
(* ──────────────────────────────────────────────────────────────────  *)

(* A symbol starts with id-start and continues with sym-body-char.   *)
(* Differences from the identifier body action table:                 *)
(*   "=" is a hard error (not a state transition)                    *)
```

```
(*   "," / "]" / ";" terminate the symbol cleanly (not errors)       *)
(*   "." is a hard error (same as in identifier body)                *)

(* Reserved keywords: a symbol token whose text is exactly "null",   *)
(* "true", "false", "on", or "off" is reclassified by the validator   *)
(* (see null-value / bool-value, §3) and is NOT delivered as a symbol.*)
(* Any other bare word — including one that merely begins with a      *)
(* keyword, like "ontology" or "nullish" — remains a symbol.          *)
symbol        = id-start , {sym-body-char} ;

(* sym-body-char spans the same byte range as id-body-char; the       *)
(* difference is purely in the termination actions fired for          *)
(* "," (new_array_value), "]" (array_outro), and ";" (value_outro).  *)
sym-body-char = id-body-char ;


(* ──────────────────────────────────────────────────────────────────  *)
(*  8. References                                                      *)
(* ──────────────────────────────────────────────────────────────────  *)

(* The "&" sigil introduces a dotted path to another key.            *)
(* The stored string starts with "." and includes all dots:           *)
(* "&.foo.bar" stores ".foo.bar".                                     *)
(* At least one segment is required; "&" alone or "&." without a      *)
(* following id-start character is an error.                          *)
(* A reference is stored UNRESOLVED — the path is a string token; the *)
(* library never dereferences it, so the target need not exist and    *)
(* cycles are not detected. Resolution is the application's job.       *)

reference    = "&" , ref-segment , {ref-segment | ref-index} ;
ref-segment  = "." , id-start , {ref-body-char} ;

(* ref-index (spec 1.1): array indexing in a reference path, e.g.      *)
(* &.matrix[0][1].  The bytes are captured verbatim into the path      *)
(* string (references are stored unresolved); bvn_dom_lookup           *)
(* interprets them at the DOM layer.  Gated on a "#!bovnar 1.1"         *)
(* declaration — in a 1.0/unversioned document "[" in a reference is   *)
(* error_unexpected_input_byte.  The reference_index sub-state          *)
(* distinguishes a "]" that closes an index from one that closes an    *)
(* enclosing array (which ends the reference).                         *)
ref-index    = "[" , DIGIT , {DIGIT} , "]" ;

(* ref-body-char: same byte set as sym-body-char, except that         *)
(* "." (0x2E) is ACT_copy_reference_dot, which pushes the dot into   *)
(* the token buffer and advances to reference_segment_intro for the   *)
(* next segment's id-start character.                                  *)
ref-body-char = sym-body-char | "." ;


(* ──────────────────────────────────────────────────────────────────  *)
(*  9. Arrays                                                          *)
(* ──────────────────────────────────────────────────────────────────  *)

(* An array is one or more bracket-enclosed rows separated by "/":   *)
(*   .a = [1, 2, 3]/[4, 5, 6];                                        *)
(* "/" emits ev_array_dim_start; the next "[" opens the next row.         *)
(* Whitespace/comments are permitted on both sides of "/" (between the *)
(* closing "]" and "/", and between "/" and the next "[").            *)
(*                                                                    *)
(* The former adjacent-row syntax "[1,2][3,4]" and the comma-between- *)
(* rows form "[1,2],[3,4]" are hard errors at the assignment level.   *)
(* array_outro no longer accepts "[" directly; the only continuation  *)
(* for a further row is "/".                                          *)
(*                                                                    *)
(* Nested arrays are valid element values inside a row:               *)
(*   [[1,2],[3,4]] is a one-row outer array whose two elements are    *)
(*   the inner arrays [1,2] and [3,4].                                *)
```

```
(*                                                                    *)
(* Leading and trailing commas produce null elements:                 *)
(*   [,1,2,]  →  four elements: null, 1, 2, null.                  *)
(*                                                                    *)
(* Semantic constraints — two distinct tiers:                          *)
(*                                                                    *)
(*   a) Enforced by the LEXER/VALIDATOR (the streaming reader): the    *)
(*      "/"-separated dimension rows of a SINGLE array must all have  *)
(*      the same element count; a mismatch is                         *)
(*      error_array_row_size_mismatch (code 2).  The lexer checks each *)
(*      row as it closes, so the error fires at the earliest byte (the *)
(*      overshooting element, or the closing "]" of a short row).     *)
(*      The per-bracket-pair context save/restore mechanism keeps this *)
(*      check scoped to one array instance: a "/"-row width never     *)
(*      leaks across a "," boundary or into a sibling branch.         *)
(*        [1,2,3]/[4,5]   → error (one array's /-rows: 3 vs 2)        *)
(*        [[1,2]/[3,4,5]] → error (inner /-array's rows: 2 vs 3)      *)
(*                                                                    *)
(*   b) Enforced by the DOM BUILDER ONLY (bvn_dom_parse), NOT by the  *)
(*      streaming reader — `loads`/the SAX callbacks accept these and *)
(*      deliver every element; only a DOM parse rejects them.  Array  *)
(*      elements must be HOMOGENEOUS (spec 1.0): the same kind        *)
(*      throughout, `,`-separated sibling sub-arrays must be          *)
(*      rectangular (match in length), and sibling structs must share *)
(*      the same shape.  A ragged sibling set is                      *)
(*      error_array_row_size_mismatch (code 2); a mix of element kinds *)
(*      (e.g. a scalar beside a sub-array) is                         *)
(*      error_array_element_type_mismatch (code 39); sibling structs  *)
(*      with differing shapes is error_struct_shape_mismatch (code 40).*)
(*        [[1,2],[3,4,5]] → DOM error (ragged sibling sub-arrays)     *)
(*        [1,[2,3]]       → DOM error (scalar mixed with a sub-array) *)
(*        [{.x=1;},{.y=2;}] → DOM error (sibling structs, differing   *)
(*                            shapes)                                 *)
(*        [[1,2]/[3,4], [5,6]/[7,8]] → accepted (same-shape blocks)   *)
(*                                                                    *)
(*   c) Array nesting depth ≤ max_array_nesting (default 64, cap 255).*)

array        = array-row , {ws , "/" , ws , array-row} ;

array-row    = "[" , ws , [row-content] , ws , "]" ;

(* row-content is OPTIONAL: "[]" (nothing between the brackets) is an   *)
(* EMPTY row — zero elements, no ev_data. It is distinct from "[null]"  *)
(* (one null element) and "[,]" (two null elements). When present,      *)
(* row-content has at least one element position; leading / trailing /  *)
(* consecutive commas each expand to a null-value element. An empty row  *)
(* has width 0, a valid row width for /-dimension consistency.          *)
row-content  = array-elem , {ws , "," , ws , array-elem} ;

(* array-elem has the same structure as a top-level value, except that an  *)
(* inline unit suffix is not permitted: it uses raw-value (no unit option), *)
(* not the value rule's scalar-with-unit alternative (see §6.5).            *)
array-elem   = [type-annotation , ws] , raw-value ;


(* ──────────────────────────────────────────────────────────────────  *)
(*  10. Structs  (nested key-value scopes)                            *)
(* ──────────────────────────────────────────────────────────────────  *)

(* A struct "{" … "}" contains zero or more assignments. "{}" is an    *)
(* empty struct (zero members). After "}" the struct behaves like any  *)
(* completed value: followed by ";" at the top level, or "," / "]" in  *)
(* array context.                                                      *)
(*                                                                    *)
(* Key uniqueness is a DOM-BUILDER constraint, NOT a grammar one: the   *)
(* lexer/validator (the streaming reader) accepts a repeated key and    *)
(* delivers BOTH assignments.  A DOM parse (bvn_dom_parse) is stricter — *)
```

```
(* a duplicate key within one scope (a struct, or the top-level         *)
(* document) is error_duplicate_struct_key.  The same key in different  *)
(* scopes is always unrelated.                                          *)
(* Nesting depth: struct_nesting_level ≤ max_struct_nesting           *)
(* (default 64 when the flag is 0; hard cap 255).                      *)
(* An unmatched "}" is error_illegal_struct_close.                    *)

struct       = "{" , ws , {assignment , ws} , "}" ;


(* ──────────────────────────────────────────────────────────────────  *)
(*  11. Octet streams  (binary-framed escape from the text layer)     *)
(* ──────────────────────────────────────────────────────────────────  *)

(* A NUL byte (0x00) appearing where a value is expected switches from*)
(* text mode to binary chunk mode.  The UTF-8 stream validator is     *)
(* suspended for the duration.                                        *)
(*                                                                    *)
(* Binary protocol (bvn_read_octet_stream):                           *)
(*   Tag 0x01 → chunk follows (2-byte LE length + data bytes)        *)
(*   Tag 0x00 → end of stream                                         *)
(*   Any other tag → error_octet_stream_out_of_sync                   *)
(*                                                                    *)
(* After ev_octet_stream_end the lexer enters octet_stream_outro,     *)
(* which accepts the same terminators as any other completed value.   *)

octet-stream = "\x00" , {os-chunk} , os-end ;

os-chunk     = "\x01" , os-length , os-data ;
os-end       = "\x00" ;

(* 16-bit little-endian length; 0x0000 encodes 65536 bytes.          *)
os-length    = byte , byte ;    (* little-endian uint16; 0x0000 → 65536 *)

os-data      = {byte} ;         (* exactly os-length bytes; not CF     *)


(* ──────────────────────────────────────────────────────────────────  *)
(*  12. Lexical primitives                                            *)
(* ──────────────────────────────────────────────────────────────────  *)

DIGIT        = "0" | "1" | "2" | "3" | "4"
             | "5" | "6" | "7" | "8" | "9" ;

ALPHA        = "A"|"B"|"C"|"D"|"E"|"F"|"G"|"H"|"I"|"J"|"K"|"L"|"M"
             | "N"|"O"|"P"|"Q"|"R"|"S"|"T"|"U"|"V"|"W"|"X"|"Y"|"Z"
             | "a"|"b"|"c"|"d"|"e"|"f"|"g"|"h"|"i"|"j"|"k"|"l"|"m"
             | "n"|"o"|"p"|"q"|"r"|"s"|"t"|"u"|"v"|"w"|"x"|"y"|"z" ;

(* utf8-leader here denotes the range used at identifier, symbol, and  *)
(* reference token-start and -body positions: 0xC3–0xF4.              *)
(* 0xC2 is the leader for U+0080–U+00BF.  It is explicitly overridden *)
(* to ACT_NONE at every identifier, symbol, and reference boundary,   *)
(* so code points U+0080–U+00BF cannot start or appear in those       *)
(* tokens.  0xC2 IS accepted inside quoted strings and type parameter  *)
(* bodies (copy_type_byte), where the full range 0xC2–0xF4 applies.  *)
utf8-leader      = (* byte in [0xC3, 0xDF] | [0xE0, 0xEF] | [0xF0, 0xF4] *) ;

utf8-continuation = (* byte in [0x80, 0xBF] *) ;

(* Printable ASCII: 0x20–0x7E *)
printable-byte = (* byte in [0x20, 0x7E] *) ;

(* High byte: any byte 0x80–0xFF.  0xC0, 0xC1, 0xF5–0xFF are         *)
(* structurally in this range but rejected by the parallel UTF-8      *)
(* validator (error_invalid_utf8_byte).                               *)
high-byte    = utf8-continuation | utf8-leader
```

```
              | (* 0xC0 | 0xC1 | 0xF5–0xFF — rejected by UTF-8 check *) ;

byte          = (* any single byte 0x00–0xFF *) ;


(* ──────────────────────────────────────────────────────────────────  *)
(*  13. Constraints not expressible in context-free EBNF               *)
(* ──────────────────────────────────────────────────────────────────  *)

(*
  a) UTF-8 validity
     Every byte in the text layer (outside octet-stream regions) must
     form valid UTF-8 when fed through bvn_utf8_feed.  The UTF-8
     validator runs in parallel with the action state machine; a
     non-UTF-8 byte causes error_invalid_utf8_byte regardless of which
     lexer state the parser is in.  Overlong encodings and surrogate
     halves are rejected.

  b) BOM uniqueness
     The UTF-8 BOM (EF BB BF) is legal only at byte offset 0.
     If the stream begins with "#" instead (a first-line comment), the
     special first_comment_* state machine watches the comment body for
     the exact byte sequence 0xEF 0xBB 0xBF and rejects it as
     error_invalid_byte_order_mark.  This detection applies only while
     the first_comment_* states are active (i.e. until the first
     newline); subsequent regular comments (comment_intro state) have
     no BOM detection, so U+FEFF encoded as 0xEF 0xBB 0xBF appearing
     in any later comment body or string literal is valid UTF-8 and is
     accepted.

  c) 0xC2 at token boundaries
     The byte 0xC2 (UTF-8 leader for U+0080–U+00BF) is mapped to
     ACT_NONE at identifier, symbol, and reference token-start and
     token-body positions, forbidding code points U+0080–U+00BF in
     those tokens.  0xC2 is accepted inside quoted strings and in type
     parameter bodies (copy_type_byte / type_body_outro).

  d) Special-number keywords
     The special floats nan / inf / ninf are bare reserved keywords (no
     sigil).  They are lexed as symbols and reclassified to numeric
     special values by the validator, like null / true / false / on /
     off; any other bare word (e.g. "infinity", "nans") stays a symbol.
     They take no inline unit — a unit is supplied via the annotation.

  e) String concatenation
     Adjacent string literals with only whitespace/comments between
     them are concatenated into a single token by the lexer.  The
     combined byte length is bounded by max_string_length (default
     UINT16_MAX = 65535 bytes).

  f) Number exponent in bare literals vs. quoted strings
     Bare number literals only accept "e"/"E" as exponent markers
     (exp_intro state).

  g) Array row-size consistency (LEXER/VALIDATOR — the streaming reader)
     The "/"-separated dimension rows of a single array must all have
     the same element count; the lexer checks each row as it closes, so
     the violation is reported at the earliest offending byte. The
     per-bracket-pair context save/restore mechanism keeps the /-row
     check scoped to one array instance.
     Violation: error_array_row_size_mismatch
       (e.g. [1,2,3]/[4,5], or [[1,2]/[3,4,5]]).

     Sibling homogeneity (DOM BUILDER ONLY — bvn_dom_parse; the streaming
     reader accepts these). Array elements must be the same kind throughout,
     `,`-separated sibling sub-arrays must be rectangular (match in length,
     so a bare array/matrix is rectangular), and sibling structs must share
```

```
   the same shape. A ragged sibling set is error_array_row_size_mismatch
   (e.g. [[1,2],[3,4,5]]); a mix of element kinds is
   error_array_element_type_mismatch (code 39, e.g. [1,[2,3]]); sibling
   structs with differing shapes is error_struct_shape_mismatch (code 40,
   e.g. [{.x=1;},{.y=2;}]). loads()/the SAX callbacks deliver such arrays
   unchanged; only a DOM parse rejects them.

h) Array and struct nesting limits
   Array nesting (bracket depth): max_array_nesting (default 64 when
     the flag is 0; hard cap 255).
   Struct nesting: max_struct_nesting (default 64 when the flag is 0;
     hard cap 255).

i) Type-annotation / value compatibility (enforced by validator)
   "utf8" requires a string value.
   "uint", "sint", "float", "float_fix", "float_dec" accept a number or
     string value; a quoted string allows special float notation.
   "utf8" and "bool" are parameterless: the lexer accepts the ":"
     and parameter bytes, but bvn_parse_type_annotation rejects any
     width, base, q, or unit parameter as error_illegal_value_type
     (e.g. "<utf8:8>" and "<utf8:,m>" are both errors).
   "uint" rejects negative number literals (error_value_out_of_range).
   Digit characters in the value must be valid for the declared base
     (error_digit_not_in_base).
   The numeric value must fit in the declared bit-width
     (error_value_out_of_range).
   For "float": accepted widths are 0, 16, or any multiple of 32 up to
     BVN_FLOAT_MAX_PREC (32768); accepted bases are 10 (or absent) and
     16; any other base → error_illegal_value_type.
   For "float_fix": accepted widths are 0, 16, 32, 64, 128, 256;
     the base parameter (_N) is forbidden (error_illegal_value_type);
     the Q parameter (qN) must satisfy 0 ≤ Q < effective_width
     (error_illegal_value_type if violated);
     Q stored in value_type_spec_t.base; retrieved via bvn_effective_q.
   For "float_dec": accepted widths are 0, 16, 32, 64, 128, 256;
     the base parameter (_N) is forbidden (error_illegal_value_type);
     the Q parameter is not valid for float_dec.
   Neither "float_fix" nor "float_dec" can be synthesised by the
     validator's default type synthesis; both require explicit annotation.

j) Identifier (key) non-empty
   The grammar requires at least one id-start character after ".".
   An empty key (".=") triggers error_empty_identifier.

k) Struct closing brace without matching open
   A "}" seen when struct_nesting_level == 0 is
   error_illegal_struct_close.

l) Comma in non-array context
   "," triggers ACT_new_array_value, which checks
   array_nesting_level > 0 and returns error_unexpected_input_byte
   if the condition is not met.  A stray "," at the top level is
   therefore an error.

m) Octet stream out-of-sync
   Any byte other than 0x01 (chunk) or 0x00 (end) as the tag byte
   inside an octet stream causes error_octet_stream_out_of_sync.

n) Error recovery (continue_on_error mode)
   When bvnr_read_flags_t.continue_on_error is set, any parse error
   invokes the on_error callback and then enters the resync state
   machine (states: resync, resync_string, resync_string_escape,
   resync_comment).  The resync machine skips bytes while tracking
   bracket and brace nesting depths, and attempts to re-synchronise
   at the next ";" at the current nesting depth.  recovery_count
   (bvnr_reader_get_recovery_count) is incremented immediately when
   an error triggers entry into resync mode, not when resync
```

```
     completes at ";".  EOF inside any resync state propagates
     the original error code rather than overwriting it with
     error_got_incomplete_bvnr_stream.

  o) Inline unit suffix
     An inline-unit follows a scalar number or string in non-array
     context only.  A special-number keyword (nan, inf, ninf) takes NO
     inline unit — it is lexed as a symbol, and symbols accept no inline
     suffix; "inf m/s" is error_unexpected_input_byte.  Supply a unit
     for a special value through the type annotation (<float:64,m/s> inf).
     At least one whitespace character MUST separate the value literal
     from the unit.  The only accepted form is "value unit"; any other
     form (no separator or colon) is a hard error.

     State-machine flow:
       1. Whitespace from number-body states (zero_intro, copy_number_byte,
          fraction_intro, copy_fraction_byte, copy_exp_byte) fires
          ACT_to_number_outro → number_outro.  Strings enter
          string_outro_nosp after the closing quote; whitespace
          transitions to string_outro.  A comment's terminating newline
          also counts as whitespace (comment_outro promotes string_outro_nosp
          → string_outro).
       2. From number_outro / string_outro (at least one ws consumed):
            alpha / "_" / "$" / "%" / "(" / UTF-8-leader (0xC2-0xF4)
            → ACT_inline_unit_intro
       3. ACT_inline_unit_intro pushes the first byte and enters
          inline_unit_body directly.
       4. inline_unit_body accumulates unit characters (including ":"
          as a valid mid-unit byte) until whitespace (→
          inline_unit_outro) or ";" (→ value_outro via finalize).

     Inside an array element (in_array_element flag set),
     ACT_inline_unit_intro immediately rejects with
     error_unexpected_input_byte.

     The validator's bvn_val_receive calls bvn_parse_unit on the
     inline buffer and compares the result to v->parsed_unit using
     memcmp on the full value_unit_t structure (valid because both
     structs are zero-initialised before parsing and all padding is
     zeroed).  When has_annotation_unit is true and the comparison
     fails, the validator returns error_unit_mismatch.
     When no annotation unit was present, the inline unit is silently
     adopted as the effective unit.
*)
```