

UNICODE OPERATORS

STEVE DOWNEY

Created: 2026-09-23 Wed 14:24

THE PROPOSAL

Declare it —

```
Vec operator⊕(Vec const& a, Vec const& b);
```

write it —

```
a ⊕ b
```

- Operators drawn from [UTR #25](#), *Unicode Support for Mathematics* — frozen by the paper, confusables removed.
- Otherwise ordinary functions: ordinary lookup, ordinary overload resolution.

THE TOKEN SET

- Pattern_Syntax, non-ASCII, in nine math and arrow blocks, gc Sm or So, minus a named exclusion list.
- Applied once, against UCD 17.0.0. Frozen by enumeration — the table does not track later Unicode versions.
- 1,381 code points, 32 contiguous ranges. Plus 75 rejected: the blocks partition exactly.

THE 32 RANGES

Range	Count	Glyphs	Block(s)
U+2190-U+21CF	64	← ... ➤	Arrows
U+21D1	1	↑	Arrows
U+21D3	1	↓	Arrows
U+21D5-U+2201	45	↕ ... ⤿	Arrows → Mathematical Operators
U+2203-U+2206	4	∃ ... Δ	Mathematical Operators
U+2208-U+2211	10	∈ ... ∑	Mathematical Operators
U+2213-U+2214	2	∓ ±	Mathematical Operators
U+2216	1	\	Mathematical Operators
U+2218	1	◦	Mathematical Operators
U+221A-U+221D	4	√ ... ∝	Mathematical Operators
U+221F-U+2222	4	⊥ ... ⋈	Mathematical Operators
U+2224-U+2235	18	‡ ... ∴	Mathematical Operators
U+2237-U+2263	45	:: ... ≡	Mathematical Operators
U+2266-U+22C4	95	≤ ... ♦	Mathematical Operators
U+22C6-U+2307	66	* ... }	Mathematical Operators → Miscellaneous Technical
U+230C-U+2319	14	⌈ ... ⌋	Miscellaneous Technical
U+231C-U+2328	13	⌞ ... ⌠	Miscellaneous Technical
U+232B-U+23E8	190	⏏ ... 10	Miscellaneous Technical
U+23ED-U+23EF	3	⏮ ⏭ ⏯	Miscellaneous Technical
U+23F1-U+23F2	2	⌚ ⌚	Miscellaneous Technical
U+23F4-U+23FF	12	◀ ... ◁	Miscellaneous Technical
U+27C0-U+27C4	5	⌞ ... ⏏	Misc Mathematical Symbols-A
U+27C7-U+27E5	31	∀ ... □	Misc Mathematical Symbols-A
U+27F0-U+27FF	16	↗ ... ↘	Supplemental Arrows-A
U+2900-U+2982	131	➡ ... ∞	Supplemental Arrows-B → Misc Mathematical Symbols-B
U+2999-U+29D7	63	⋮ ... ✖	Misc Mathematical Symbols-B
U+29DC-U+29FB	32	∞ ... ≡	Misc Mathematical Symbols-B
U+29FE-U+2B1A	285	+ ... □	Misc Mathematical Symbols-B → Supplemental Mathematical Operators → Misc Sym/Arrows
U+2B1D-U+2B4F	51	• ... ↓	Miscellaneous Symbols and Arrows
U+2B51-U+2B54	4	* ... ⬠	Miscellaneous Symbols and Arrows
U+2B56-U+2B73	30	⊙ ... ↓	Miscellaneous Symbols and Arrows
U+2B76-U+2BFF	138	↶ ... ⚡	Miscellaneous Symbols and Arrows

WHAT CAME OUT

- Confusables, multi-character: $\Leftarrow \Rightarrow \Leftrightarrow \leq \geq$ read as $<= => <=> <= >=$.
- Confusables, single: $- / * \cdot \cdot : |$ read as $- / * \cdot \cdot : |$.
- Identifier characters, by TR31 §7.1: $\partial \nabla \infty$.
- Paired brackets, gc Ps and gc Pe: $[] [[] \langle \rangle$ — delimiters.
- Emoji presentation, by the TR31 §7.2 carve-out: 🕒 🕒 ⌚.

IT LOOKS LIKE THIS

```
constexpr int operator⊕(int a, int b) { return a + b; }
static_assert(5 ⊕ 7 == 12);

Matrix operator⊗(Matrix const&, Matrix const&);
a ⊗ b ⊗ c           // operator⊗(operator⊗(a, b), c)

Vec operator⊖(Vec const&); // one parameter: unary prefix
⊖v                     // operator⊖(v)
```

ORDER OF INTERPRETATION

ORDER OF INTERPRETATION

- Tokens, then parse, then lookup. In that order. Always.

ORDER OF INTERPRETATION

- Tokens, then parse, then lookup. In that order. Always.
- The parser identifies an operator with **no knowledge of any declaration.**

ORDER OF INTERPRETATION

- Tokens, then parse, then lookup. In that order. Always.
- The parser identifies an operator with **no knowledge of any declaration**.
- So the set is fixed by the grammar — a frozen enumeration.

ORDER OF INTERPRETATION

- Tokens, then parse, then lookup. In that order. Always.
- The parser identifies an operator with **no knowledge of any declaration**.
- So the set is fixed by the grammar — a frozen enumeration.
- Meaning arrives later, at overload resolution — like `operator+` on a class type.

ORDER OF INTERPRETATION

- Tokens, then parse, then lookup. In that order. Always.
- The parser identifies an operator with **no knowledge of any declaration**.
- So the set is fixed by the grammar — a frozen enumeration.
- Meaning arrives later, at overload resolution — like `operator+` on a class type.
- Julia: fixed parse table, user-defined meanings. Not Swift's declaration-gated parse.

POSITION DECIDES

```
⊖a ⊖ ⊖b      // operator⊖(operator⊖(a), operator⊖(b))  
^      ^      operand position -> prefix  
      ^      post-operand -> infix; decided here, no look in either direction
```

- ^ marks the column where the parse commits, or where it breaks.
- Operand position → prefix. After a complete operand → infix. No postfix.
- Nothing else consulted: not whitespace, not declarations, not the next token.

THE FIXITY PROBLEM

THE FIXITY PROBLEM

- A declared precedence must travel with the name — headers, modules, TUs.

THE FIXITY PROBLEM

- A declared precedence must **travel with the name** — headers, modules, TUs.
- The parse of $a \oplus b \otimes c$ comes to depend on which imports are visible.

THE FIXITY PROBLEM

- A declared precedence must **travel with the name** — headers, modules, TUs.
- The parse of $a \oplus b \otimes c$ comes to depend on which imports are visible.
- Two TUs disagreeing about the grouping: an ODR / IFNDR factory.

THE FIXITY PROBLEM

- A declared precedence must **travel with the name** — headers, modules, TUs.
- The parse of $a \oplus b \otimes c$ comes to depend on which imports are visible.
- Two TUs disagreeing about the grouping: an ODR / IFNDR factory.
- A precedence is a meaning: $\otimes$ is multiplication-like in tensor algebra, backwards in a tropical semiring.

FIXITY, CONCRETELY

```
a ⊕ b ⊗ c      // fixity visible: ⊗ binds tighter -> a ⊕ (b ⊗ c)
a ⊕ b ⊗ c      // fixity not seen:                (a ⊕ b) ⊗ c
  ^      ^
<<<<<<<<<    the look required is *the imports*: headers, modules,
                and their order, before the first token can shift
```

ONE PRECEDENCE ONE ASSOCIATIVITY

$a \oplus b \otimes c$ // $(a \oplus b) \otimes c$ -- everywhere, in every TU
 $\wedge$ $\wedge$ each operator commits at its own token

- Teachable and learnable. Parenthesis for grouping is already good practice.

cLang-format MUST WORK

cLang-format MUST WORK

- Formats code it has not compiled: no Sema, no lookup, no imports.

clang-format MUST WORK

- Formats code it has not compiled: no Sema, no lookup, no imports.
- Still has to answer *how does this group?*

cLang-format MUST WORK

- Formats code it has not compiled: no Sema, no lookup, no imports.
- Still has to answer *how does this group?*
- Prototype: the query answered `prec :: Unknown`; chains wrapped wrongly; **every spacing and annotation test passed.**

cLang-format MUST WORK

- Formats code it has not compiled: no Sema, no lookup, no imports.
- Still has to answer *how does this group?*
- Prototype: the query answered `prec : : Unknown`; chains wrapped wrongly; **every spacing and annotation test passed.**
- Teaching it these rules took about forty lines.

cLang-format MUST WORK

- Formats code it has not compiled: no Sema, no lookup, no imports.
- Still has to answer *how does this group?*
- Prototype: the query answered `prec : : Unknown`; chains wrapped wrongly; **every spacing and annotation test passed.**
- Teaching it these rules took about forty lines.
- A table it can consult. A declaration it cannot.

THE MISPARSE

THE MISPARSE

- Postfix was prototyped: after a complete operand, look ahead one token — "can it begin a cast-expression?" → infix; otherwise postfix.

THE MISPARSE

- Postfix was prototyped: after a complete operand, look ahead one token — "can it begin a cast-expression?" → infix; otherwise postfix.
- The witness set contains surprises.

a $\ominus$ && b

```
a  $\ominus$  && b      // intended: postfix  $\ominus$  on a, then logical AND
^ >>          && can begin a cast-expression (GNU address-of-label)
               -> infix:  $\ominus(a, \&\&b)$  ... "use of undeclared label 'b'"
```

- **>** marks a token the rule must examine *ahead* of the **^**.

THE DIAGNOSTIC COST

```
a ̳ ;  
  ^ >
```

; cannot begin a cast-expression -> a valid postfix use
"no matching overload" replaces "expected expression"

- Every missing-operand typo becomes a *valid postfix parse*.
- Every user pays, whether or not they write a postfix operator.

MORE WAYS TO GO WRONG

MAXIMAL MUNCH

```
a &&& b      // today: a && (&b) -- valid, and people write it
  ^^          maximal munch has already produced &&
   ^          the remainder is &, address-of b
```

- Lexing precedes lookup: no declaration can split, or fuse, a token.
- No user &&&, ever. No operator is a prefix of another.

WHITESPACE FIXITY — THE SWIFT RULE

```
a ⊖ b           // spaces both sides           -> infix
a ⊖b           // space before, none after       -> prefix, on b
a⊖ b           // none before, space after       -> postfix, on a
<^>           one column of lookback and one of lookahead, at every use
```

- < marks one the rule must examine *behind* it. So <^> is a column of lookback and a column of lookahead, at every use.

ONE SPACE, DIFFERENT PROGRAM

```
x = a⊖ + b      // postfix ⊖ on a, then binary +  
x = a ⊖ + b     // infix ⊖, unary + on b  
  ^  
    same tokens either side of one space
```

THE ONE TRUE AMBIGUITY

$\boxplus(x)$

^

a call of a function named $\boxplus$? prefix $\boxplus$ applied to (x) ?

- The resolutions on offer: whitespace sensitivity, or a declaration-dependent parse. Both declined — the operator set and the identifier set stay disjoint.

POSITION DECIDES

- The parse of every expression is a function of the expression alone.
- Only the *meaning* of operator $\boxplus$ travels — by ordinary lookup, which already knows how to travel.
- $\wedge$ only: no lookahead, no lookback, no whitespace, no declarations.

QUESTIONS?

THANK YOU!