

An Infix Operator and a Keyword Escape for C++

Steve Downey <sdowney@gmail.com>

Document #: D4307R1
Date: 2026-10-22
Project: Programming Language C++
Audience: EWG
Source: <https://github.com/steve-downey/backtick-papers/backtick-infix-and-keyword-escape.org>
heads/blank-slate-0-gfdeaeb0-dirty

1 Abstract

2 Revision history

2.1 R1

Blank slate rewrite of prose.

2.2 R0

3 Before / After

3.1 The infix operator

Before	After
<pre>if (approx_equal(dot(u, v), 0.0)) reject(u, v);</pre>	<pre>if (u `dot` v `approx_equal` 0.0) reject(u, v);</pre>
<pre>auto d = std::gcd(std::gcd(a, b), c);</pre>	<pre>auto d = a `std::gcd` b `std::gcd` c;</pre>
<pre>if (intersects(a, b) && !contains(a, b)) handle_partial_overlap(a, b);</pre>	<pre>if (a `intersects` b && !(a `contains` b)) handle_partial_overlap(a, b);</pre>
<pre>// evaluation inverts reading order: // innermost call is the first stage store(validate(parse(s)));</pre>	<pre>// with a two-line pipe helper, // stages read left to right s `pipe` parse `pipe` validate `pipe` store;</pre>

3.2 The keyword escape

Before	After
<pre>// C++17: fine. C++20: ill-formed – // 'requires' became a keyword. bool requires(const License& l); if (requires(user.license)) admit(user);</pre>	<pre>// declares and calls an ordinary // function whose name is "requires" bool `requires`(const License& l); if (`requires`(user.license)) admit(user);</pre>

4 The proposal

Two features, one token, one paper.

The infix operator. “`x ‘f’ y`” is sugar for `f(x, y)` (equivalently `(f)(x, y)`), where the operator slot between the backticks is an arbitrary expression parsed as an assignment-expression, or a type-name, which constructs. The value, type, and semantics are exactly those of the corresponding call. The operator is left-associative and binds tighter than any other binary operator, looser than the unary and postfix operators.

The keyword escape. In any position where the grammar expects a name (a declarator-id, a class, enumeration or namespace name, a template parameter’s name, a label, an operand, after `‘:’`, `‘->’`, or `‘::’`), a backtick pair wrapping a word spelled as an identifier denotes an ordinary identifier whose spelling is that word. Keywords are the case the feature is named for, and the rule is not restricted to them: “`‘foobar’`” and `‘foobar’` are the same identifier, so a declaration may use one spelling and its uses the other. It is purely a source-level construct; the resulting identifier participates in lookup, mangling, and linkage exactly as if the word had never been a keyword.

5 Motivation, beyond readability

5.1 Short-circuiting returns to user code

C++ reserves non-strict evaluation to a fixed set of built-ins: `&&`, `||` and `?:`. The comma operator is the near neighbor, and is not one of them: it evaluates both of its operands, and what it fixes is the **sequencing**. Overloading does not get non-strict evaluation back: an overloaded `operator&&` evaluates both operands, which is the trap everyone has been bitten by, and the reason the standing advice is: don’t. Non-strict evaluation has been closed to user code since C++98.

A backtick helper whose right operand is a **callable** reopens it, because the helper decides whether (and when) that operand runs:

```
// short-circuiting logical implication: p => q == !p || q
inline constexpr auto implies =
    [](bool p, auto&& q) -> bool { return !p || q(); };

p `implies` [&]{ return expensive(); } // runs only when p holds
```

The same shape gives lazy defaults (“`opt ‘or_else’ [&]{ return costly(); }`”), guarded effects, and bespoke control operators. Any binary operation that must not evaluate its right side unconditionally is in reach. Chaining fallible steps is the zero-ceremony special case: the stages are already functions, so no thunk is written and the short-circuit falls out:

```
inline constexpr auto mbind =
    [](auto&& m, auto&& f)
    { return std::forward<decltype(m)>(m)
        .and_then(std::forward<decltype(f)>(f)); };

parse(s) `mbind` validate `mbind` store; // stops at the first error
```

5.2 One helper makes it a pipeline, and the ranges closures already fit

```
inline constexpr auto pipe =
    [](auto&& x, auto&& f) -> decltype(auto)
    { return std::invoke(std::forward<decltype(f)>(f),
        std::forward<decltype(x)>(x)); };

x `pipe` f `pipe` g `pipe` h // h(g(f(x))) – data-flow order
```

This drives the existing range adaptor closures unchanged. `views::filter(pred)` and `views::transform(fn)` are already unary callables (`c | a` is defined as `a(c)`), so `pipe` feeds them directly:

```
r `pipe` views::filter(pred) `pipe` views::transform(fn)
// identical result and laziness to:
r | views::filter(pred) | views::transform(fn)
```

Same closure objects, same lazy views, verified identical output on the built compiler. The per-library `operator|` overloads exist only to choose the `‘|’` syntax; the closures themselves need nothing, so they work under backtick for free.

5.3 Reusable, point-free composition

Compose stages into a named pipeline once, apply it many times:

```
inline constexpr auto then =
  [](auto f, auto g)
  { return [=](auto&&... a) -> decltype(auto)
    { return g(f(std::forward<decltype(a)>(a)...)); }; };

auto clean = trim `then` lower `then` dedup; // a reusable callable
clean(s);
```

Left association gives left-to-right composition, mirroring a reusable adaptor chain.

5.4 These are motivation, not proposals

This is not a library paper, no library facilities are proposed.

6 Prior art

6.1 Infix application of named callables

- Haskell, PureScript, and other Haskell descendents have backtick for infixing.
- Kotlin has an `infix` keyword.
- R has `%op%` operators.
- Fortress has named operators.

6.2 Escaping keywords as identifiers

- C# `@-verbatim`.
- F# double backtick names.
- Nim backtick stropping.
- Kotlin backtick identifiers.
- Swift ``class``.
- Rust raw identifiers.

7 Design choices and decisions

7.1 It desugars to the call, and that is the whole design

7.2 Precedence: the highest binary operator, and why not higher

7.3 Left-associative

7.4 The operator slot is an assignment-expression

7.5 Bare nesting is chaining; real nesting takes parentheses

7.6 Evaluation order is the call's

7.7 A type name in the slot constructs

7.8 The escape yields an ordinary identifier

7.9 One spelling

7.10 Two uses, one paper; no library

8 Anticipated objections

8.1 "Define `*` on a type instead"

The objection: named binary operations do not need an infix spelling, because C++ already has one, which is to overload an operator on a type. Saturating multiplication does not need `a `mul_sat` b`; it needs a `Saturating<double>` whose `operator*` saturates.

Anyone having tried this knows all the problems.

You can't overload for `operator*(double, double)`, you need a `Saturating<double>` in play. This is why we already added `std::mul_sat`. We, rightly, did not ship a `Saturating<T>`.

The scope is too broad, and it is painfully easy to use saturating math in places you did not expect to. The ergonomics of a single `*` operator make different behavior depending on the argument types a trap.

9 Fitting the grammar, without collision

9.1 The token is free

9.2 The productions

```
user-infix-expression:
  cast-expression
  user-infix-expression ` backtick-operator ` cast-expression

backtick-operator:
  assignment-expression
  simple-type-specifier
  typename-specifier
```

The second and third alternatives are the type slot, and the type reading wins exactly when lookup finds a type or a class template. The rule does not follow from the expression slot: a bare type-name is not an *assignment-expression*, so a grammar with only the first alternative would contradict the section above rather than imply it.

user-infix-expression slots between *cast-expression* and *pm-expression*: the pointer-to-member productions consume a *user-infix-expression* where they consumed a *cast-expression*, and everything above them is unchanged. In implementation terms this is one new top level in each compiler's binary operator precedence table. The left recursion gives left associativity; the *cast-expression* operands give the symmetric prefix binding argued for above.

The new level is **not** a *fold-operator*: `(... `f` N)` is ill-formed. Both implementations reject it, each with an ordinary parse error that says nothing about why, which is the reason for stating the exclusion here: nothing else would say it was chosen. Excluding costs one clause in the predicate that already decides which operators may be folded over; admitting would require a fold-expression node that can hold an arbitrary slot expression, which today's cannot. Nothing is foreclosed; every program a later revision would newly accept is one this proposal rejects.

The keyword escape is a new *identifier* alternative in name positions:

```
escaped-identifier:
  ` identifier `
```

yielding an identifier token whose spelling is the word between the backticks. The *identifier* in that production is the lexical one, the production in `[lex.name]`, which every keyword matches; `[lex.key]` is what makes a keyword out of one of those matches, and inside an escape it does not apply. So the escape admits any word spelled as an identifier, and what it yields is an ordinary identifier. It may appear wherever the grammar uses *identifier* as a terminal, and nowhere else: a *declarator-id*, a *class-head-name*, an *enum-name*, an enumerator, a *namespace-name*, a template parameter's name, a *mem-initializer*, a label, a *primary-expression*, an *id-expression* after `.`, `->` or `::`.

9.3 The same-delimiter problem

The open and close delimiter are the same token, so while parsing the operator slot, a naive expression parser would take the closing backtick as the start of a second, nested backtick operator. C++ has, of course, been here before: `>` inside a template-argument list is a closer, not an operator, and `vector<vector<int>>` is handled by a parser flag — Clang's `GreaterThanIsOperator`, GCC's `greater_than_is_operator_p` — that turns the operator meaning off in that context. We do the same thing: a `BacktickIsOperator` flag, false while parsing the slot, restored inside any nested parentheses or brackets so that parenthesized nesting works. Both implementations are modeled line-for-line on their compiler's existing `>` handling.

9.4 Operator and escape never meet

C++ expression grammar strictly alternates between wanting an operand and wanting an operator. The escape lives exclusively in operand positions; the infix operator lives exclusively in the post-operand position. The positions are mutually exclusive, so one lexical token serves both uses with no lookahead and no ambiguity, the same strategy the language already uses to give `*`, `&`, and `<` their multiple readings.

“`cpp int 'new'(int, int); / declarator-id -> escaped identifier "new" 'new'(a, b); / primary -> call to the function named "new" obj.delete(); / after ':' -> member named "delete" x 'f' y; / post-operand -> infix: f(x, y) x 'new' y; // escaped callee -> ('new')(x, y), parenthesized slot`”

There might seem to be a second, independent signal here: that an escape wraps a keyword, which is never a valid callee expression, and a slot wraps an expression, which is never a bare keyword. Since the escape wraps any identifier, that signal does not exist. The design does not rest on it, and the case where the two token sequences coincide shows why. Read as an escape and read as a slot, ``f`` names the same thing, `f` the two readings differ only over whether `f` is an operand or a callee, and that is precisely what the position states. Position alone suffices, as it does for `*`, `&`, and `<`.

Because the escape yields an ordinary identifier, nothing downstream of the parser changes: no new lookup rules, no mangling scheme, no ABI surface. `void `new`();` links as a function named `new`.

10 Implementation experience

Implemented in both GCC and Clang. There were no particular implementation difficulties for either. Implementing both did partly dictate the implementation strategy for the identifier escape, but did not make it unmanagably more complicated.

10.1 The forks

- **Clang**, two branches carrying the same feature diff, one on the LLVM 23 release branch ([steve-downey/llvm-project @ backtick-23](#)) and one on trunk ([@ backtick-trunk](#)): lexer token, the new binary precedence level, a parser branch modeled on the ternary operator (the closest existing delimited-middle operator), Sema desugaring to a `CallExpr`, the driver flag, a transparent AST wrapper so `-ast-print` round-trips the surface syntax, and `clang-format` support for both uses (canonical spacing, and a break policy that hard-forbids a break next to either backtick). A bare unqualified name in the slot reaches the call builder unresolved, as an `UnresolvedLookupExpr`, so ADL is the call's. Tests live under `clang/test/` (the `backtick-*` files in `Parser/`, `Lexer/`, `SemaCXX/`, `AST/`, `Analysis/` and `CIR/CodeGen/`, plus `Driver/` and the `Format` unit tests), and the full `check-clang` regression gate stays green with the flag off and on.
- **GCC** ([steve-downey/gcc @ backtick](#)): `libcpp` token (replacing today's stray ```` in program diagnostic), parser precedence level and slot handling, desugaring via `finish_call_expr`, the same flag, and ADL on the slot for both unqualified forms, a plain name and a template-id. Tests live under `gcc/testsuite/g++.dg/backtick/`, with a module pair under `g++.dg/modules/` checking that an escaped name streams through a compiled module interface and mangles as the ordinary module-attached name it is.

11 The design, in one place

Everything argued above reduces to the following, and the wording that follows says no more than this. Every item is decided, and every item is implemented in both compilers except where noted.

The infix operator.

- `x `f` y` is `f(x, y)`. It is desugared in the front end, before overload resolution, to the call node the compiler would have built for the call; overload resolution, ADL, templates, `constexpr`, conversions, value categories and code generation are the call's. ADL fidelity is normative: the slot reaches the call builder unresolved.
- The slot is an *assignment-expression*, or a *simple-type-specifier* or *typename-specifier*, in which case `x `T` y` is `T(x, y)` and CTAD applies. The type reading wins when lookup finds a type or a class template. (Clang implements the type slot; GCC does not yet.)

- Precedence is the highest binary level: tighter than `*`, looser than the unary and postfix operators. Operands are *cast-expressions*, so a prefix operator binds to its own operand and `-a `f` -b` is `f(-a, -b)`.
- Left-associative: `a `f` b `g` c` is `g(f(a, b), c)`. Fixity is fixed, not declared per name and not derived from a spelling.
- Bare nesting is chaining, because the token stream is the same. To nest, parenthesize the slot. Neither compiler diagnoses the bare form, and neither can without rejecting legal chains.
- Evaluation order is the call's: the slot is sequenced before both operands, the operands are unsequenced with respect to each other.
- Operands exclude braced-init-lists. The operator is not a *fold-operator*.
- One spelling. No digraph, no alternative token.

The keyword escape.

- ``kw``, wherever the grammar uses *identifier* as a terminal, is an ordinary identifier spelled `kw`. Lookup, overload resolution, mangling, linkage and ABI are untouched; `void `new`();` links as a function named `new`.
- The escape is spelling and not identity. A printer that names the entity restores the backticks; the AST dump keeps the bare word.
- The positions are enumerated in the grammar section and swept in both compilers: every declaring position, every using position, and either or both ends of a qualified name.

Shared.

- One token, two grammatical positions, no lookahead: operand and name position is the escape, post-operand position is the operator. They never meet, and neither use depends on the other for its disambiguation.
- Both are gated behind an opt-in flag for the proposal period. Flag off, every existing program is untouched, byte for byte.
- No library. `pipe`, `then`, `mbind` and `implies` are motivation, each a few lines of user code, and any that deserve standardizing arrive in a later paper with usage behind them.
- Two uses, one paper, because they share the token and the committee.

Separately pollable. One question: whether the escape is restricted to words that are keywords, so that ``foo`` is ill-formed rather than a second spelling of `foo`. The proposal is the unrestricted rule, for the reason §"The escape yields an ordinary identifier" gives — a restricted escape cannot be written until the standard that takes the word has shipped, so it can repair a break and can never prevent one. Both implementations support either answer, so EWG can still poll it.

12 Wording

The following wording is pro forma. It is intended to make the grammar change concrete; it is not offered as final CWG wording. Wording is relative to the current working draft.

12.1 [lex.operators]

Modify the *operator-or-punctuator* grammar in [lex.operators] paragraph 1 by adding ```:

```
operator-or-punctuator: one of
...
?      ::      .      .*      ->      ->*      ^^      ~      `
...

```

[Note 1: The row shown is [lex.operators]'s third; only the trailing ``` is added. — end note]

12.2 [lex.name]

Add to [lex.name], after the paragraphs defining *identifier*:

escaped-identifier:
` identifier `

- [x] An *escaped-identifier* may appear wherever the grammar uses *identifier*. The token between the ` tokens is treated as an *identifier* even if it would otherwise be a keyword or alternative token. The ` tokens are not part of the identifier. An *escaped-identifier* and an unescaped *identifier* with the same spelling are the same identifier.

[x+1]

[Example 1:

```
void `new`();           // declares a function named new
`new`();              // and calls it
int `count` = 0;
++count;              // names the same object
```

— end example]

12.3 [expr.mptr.oper]

Modify the grammar of [expr.mptr.oper] paragraph 1:

pm-expression:
cast-expression user-infix-expression
pm-expression .* cast-expression user-infix-expression
pm-expression ->* cast-expression user-infix-expression

12.4 [expr.backtick] (new subclause)

Annex A ([gram]) is updated mechanically to match.

Insert a new subclause between [expr.cast] and [expr.mptr.oper]:

Backtick operator [expr.backtick]

user-infix-expression:
cast-expression
user-infix-expression ` *backtick-operator* ` *cast-expression*
backtick-operator:
assignment-expression
simple-type-specifier
typename-specifier

- [1] An expression of the form *E1* `0` *E2* is identical (by definition) to 0(*E1*, *E2*) ([expr.call]).
- [2] If the *backtick-operator* is a *simple-type-specifier* or *typename-specifier* denoting a type *T* or a placeholder for a deduced class type, an expression of the form *E1* `T` *E2* is identical (by definition) to T(*E1*, *E2*) ([expr.type.conv]). A *backtick-operator* whose tokens can be interpreted both as an *assignment-expression* and as a *simple-type-specifier* or *typename-specifier* is interpreted as a *simple-type-specifier* or *typename-specifier*.
- [3] When parsing a *backtick-operator*, the first non-nested ¹ is taken as the ending delimiter, rather than as the first delimiter of a nested *user-infix-expression*.

13 Appendix A: alternative spellings, and the lexical inventory

Backtick is the sole proposed spelling. This appendix carries the analysis behind that decision: why alternatives get raised, the lexical filter any candidate must pass, the candidates themselves, and the wider inventory of what ASCII actually remains, so that if the spelling question is raised it can be settled against this record in this paper instead of reopened in a future one.

1) A ` that appears within a matching pair of parentheses, brackets, or braces is nested.

13.1 A.1 Why alternatives get raised

Two reasons, both real, both minor. A single backtick is Markdown’s inline-code delimiter, so `x `f` y` in running prose fights the markup. And backtick is a dead key or awkward on some non-US keyboard layouts; it was one of the ISO-646-variant characters, alongside `# [ ] { } | ~ ^ \`.

The Markdown objection is on the record in WG21, made against this exact character by seven authors. P3381R0 (Revzin et al. 2024) evaluated twelve single-character spellings for the reflection operator, and four multi-character ones, the backtick among them, and rejected it:

The backtick has the advantage that it’s pretty small, even smaller than `^`. But it has the disadvantage that backtick is used by Markdown everywhere inline code blocks, and not all Markdown implementations properly give you mechanisms to escape it. While not necessarily a show-stopper, we also just don’t think it’s good enough to reasonably pursue.

The objection is granted, and the reasoning is right for the operator it was reasoning about. However, that operator is not this one. A reflection operator is written once per reflection and sits inside dense expression text, where a backtick operator is written where the name of an operation would go; the CommonMark cost falls on different text, at a different density, and the same span form handles it either way.

Neither is a capability gap. CommonMark’s multi-backtick spans already delimit code containing backticks (writing ``` x `f` y ``` renders as `x `f` y`), and fenced blocks, the dominant case for code, are unaffected entirely. The span form renders correctly on GitHub and on Mattermost, the committee’s own chat server (author-verified). In the very forum where the operator would most often be typed in running text, the friction is a solved problem. What an alternative spelling offers is ergonomics for the minority case, inline prose.

The fix is the one CommonMark already supplies, and its cost is small and bounded. In running prose the operator is written as a multi-backtick span: doubled delimiters with a padding space at each end, so `x `f` y` renders with its backticks intact. Each inline example pays those extra delimiters and the padding, and nothing else does. A document whose every other line infixes a backtick is the worst case the objection can construct, and the span form carries even that.

13.2 A.2 The filter a candidate must pass

An alternative spelling is an additional token lexed by maximal munch, like the existing digraphs. To be viable, the sequence must never appear adjacent in a valid current program. Three traps, each of which has bitten a real token before:

1. **Maximal-munch theft.** After a binary operator or `<`, a unary-capable character is already valid: `a <-b`, `a * *p`, `!!x`. Minting the two-character form silently changes meaning.
2. **The `::` neighborhood.** `a<:b` needed the `<::` carve-out in [\[lex.pptoken\]](#) because `vector<::std::string>` broke. Any new `<-`-prefixed token lives next to that scar.
3. **Universal-character-name munch.** `\uXXXX` can begin an identifier, so a candidate whose second character is `\` can split a UCN, a non-obvious break.

13.3 A.3 The candidates

Spelling	Lexically clean?	Verdict
<code>\< ... \></code>	yes — <code>\</code> is no token today, and <code>\<</code> cannot start a UCN	front-runner, if ever forced
<code>< ... ></code>	yes	blocked socially: <code> ></code> is P20.
<code><\ ... \></code>	no — UCN munch (trap 3)	inferior twin of <code>\< ... \></code> ; rej
<code>(...)</code>	yes	heavy; Haskell ”banana bra
<code>x \op\ y</code>	yes	visually too light; symmetr
<code>\$... \$</code>	no — <code>\$</code> is an identifier character under default-on <code>-fdollars-in-identifiers</code>	reject
<code>@ ... @</code>	clean in C++	the Objective-C sigil, in a l
<code><: :> <% %></code>	—	already digraphs for bracke

13.4 A.4 The front-runner is not an alias

One candidate is different: `\< ... \>` is lexically bulletproof *and* asymmetric. Distinct open and close tokens would eliminate the same-delimiter problem outright (no `BacktickIsOperator` flag) and with it the nesting

rule, since `x <f <g> h> y` parses unambiguously with no parentheses. That is, on engineering grounds, a better-designed operator than the backtick.

However, adopting it would mean choosing a *different primary spelling*, a different operator with a different feel and none of the Haskell lineage. It would not be an alias. The decision taken here is backtick as the single spelling. We considered the better-engineered stranger and chose the familiar borrowed spelling, on purpose.

13.5 A.5 The rebuttals, pre-loaded

1. **It is ergonomics and not capability.** Inline prose works today via CommonMark spans (A.1); fenced blocks cover code.
2. **Two spellings is a permanent tax.** Teaching doubles; clang-format must pick a canonical form and normalize to it; `-ast-print` must choose; grep, linters, and tooling grow a second case, forever, for a cosmetic win.
3. **Direction of travel.** Trigraphs were removed in C++17; digraphs are vestigial and periodically floated for removal. A new alternative token invites "and will you deprecate this one too?"
4. **It fragments the spelling.** The readability case rests on one recognizable form; two camps undercut it.
5. **None reads better.** Backtick is the established infix-quote idiom. The alternatives carry foreign connotations: pipe, escape, banana.
6. **If the markup clash warranted a change, it would argue against the primary and not for a second spelling.** We weighed that and chose backtick-primary anyway. An alias is the worst of both worlds.
7. **"Add it later if needed" is not a cheap option.** A follow-up spelling costs its own paper, an EWG poll, CWG wording, and a ballot cycle, and risks shipping the operator first and bolting a second spelling on after. Deferral buys no option value. The question is settled here.

13.6 A.6 The wider inventory: what ASCII actually remains

The same availability analysis generalizes, and it gets asked in the room. A sequence `XY` is mintable only if `XY` is not a token or token-prefix today *and* `Y` cannot validly follow `X` in a current program. The second clause is the surprising one: after any binary operator or `<`, the unary-capable characters `- + * & ~ !` are already legal, so `<-`, `<+`, `<*`, `<&`, `<~`, `<!` are all blocked; `a * *p` and `!!x` are the cautionary cases. `<|` survives only because `|` is the one bar with no unary form.

Free standalone characters: exactly three, as noted in the grammar section. They are the backtick (claimed by this proposal) and `@` and `$`, compromised by Objective-C and `-fdollars-in-identifiers` respectively. There is a fourth character, `\`, which is free as a *token* and is not counted among the three: it is the line-continuation marker and the universal-character-name lead-in, so it is usable only with care.

Clean two-or-more-character sequences of note: `==>`, `<==`, `<==>`, `<|`, `~>`, and `%>` are mintable today; `=>` and `|>` are lexically clean but spoken for, by P2971's implication operator and P2011's pipeline respectively; `<=>` is spaceship; and `^^` — available by the same analysis until recently — was claimed by reflection (Revzin et al. 2025). Reflection moved from a single `^` to `^^` in P3381R0 (Revzin et al. 2024), after running this exercise, and that paper's candidate table reaches this section's count independently: it calls the backtick the third character recently added to the basic character set, after `$` and `@`. The lesson from that precedent: doubling an operator with no unary form is the reliable way to find clean real estate, and doubling one that has a unary form never is.

13.7 A.7 Why the demand for that inventory evaporates

This proposal is, in effect, a general infix-operator facility: any named binary operation is `x `op` y` with no new punctuation. So the standing demand for new operator tokens, which is what previously justified spending scarce lexical real estate, largely evaporates. `x `implies` y`, `x `pow` y`, `x `dot` y` all work today under the feature, and the table in A.6 can stay unspent.

The residual cases where a dedicated punctuation is still worth minting are the ones a desugar-to-call cannot express: non-strict evaluation with a bare-expression right operand (Walter Brown's short-circuiting `=>` implication (Brown 2025); though the motivation section shows a thunk recovers the capability, so the dedicated operator is an ergonomic win), custom precedence or associativity outside the single backtick level, and operations frequent enough that even ``op`` is too much ceremony, a high bar. Everything else is a

backtick call. The inventory above is what remains technically possible; this proposal removes most of the motivation to spend it.

14 References

Brown, Walter E. 2025. “Implication for C++.” WG21. <https://wg21.link/p2971r3>.

Revzin, Barry, Wyatt Childers, Peter Dimov, Andrew Sutton, Faisal Vali, Daveed Vandevoorde, and Dan Katz. 2024. “Syntax for Reflection.” WG21. <https://wg21.link/p3381r0>.

———. 2025. “Reflection for C++26.” WG21. <https://wg21.link/p2996r13>.