

Reflecting Function Bodies: Expression and Statement Introspection

Document #: P4395R0 [[Latest](#)] [[Status](#)]
Date: 2026-10-23
Project: Programming Language C++
Audience: SG7
Reply-to: Siqi Yuan
<syuan147@bloomberg.net>
Alex Yesmanchuk
<ayesmanchuk@bloomberg.net>
Polina Vyshnevskaya
<pyshnevskaya@bloomberg.net>
Lucas Rey
<lrey4@bloomberg.net>
Thomas Prideaux-Ghee
<tprideauxghe@bloomberg.net>
Philip Craig
<pcraig30@bloomberg.net>

Contents

1	Introduction	1
1.1	Background	2
2	Motivation	2
2.1	Automatic differentiation as a library	2
2.2	Getting more information from test assertions	3
2.3	ORMs and mapping C++ to query languages	4
2.4	Mutation testing	4
2.5	Hardware-friendly code transformation for HPC	4
3	Proposed features	5
3.1	Function body reflection	5
3.2	Statement reflection	5
3.3	Declaration statement reflection	6
3.4	Return value reflection	6
3.5	Expression reflection	6
3.6	Operators	8
4	Design decisions & discussion	8
4.1	Abstraction levels	8
4.2	The order of the transformations	8
4.3	Expression splicing	9
5	Implementations and demonstrations	9
5.1	Automatic differentiation library	9
5.2	AD benchmarks	9
6	References	10

1 Introduction

[\[P2996R13\]](#) gives C++26 reflection over a program’s declarations — the types, functions, members, and

enumerators that make up an interface. It stops at the boundary of a function body: one can reflect that `f(double x, double y)` is a function taking two `doubles`, but not that its body computes `x*y + sin(x)`.

We propose to extend this capability to include read-only reflection of the code inside a function presented as a collection of statements that can be traversed down to individual operators and operands. The scope of this paper will be limited to declaration, expression, and return statements inside function bodies (see [Design decisions & discussion](#)).

This would enable applications performing source-level program analyses and transformations, including automatic differentiation, to move from compiler tooling into familiar (if `consteval`) libraries. Our flagship demonstration is a compile-time automatic differentiation (AD) engine that turns a plain `double f(double)` into its exact derivative, generated during compilation.

The core features proposed in this paper are implemented both in a fork of the P2996 reference compiler, Clang (see [Implementations and demonstrations](#)) and in a GCC fork.

1.1 Background

Reflection of statements and expressions has been discussed before. This paper builds on these works and proposes a concrete API for traversing statements and expressions within function bodies. Previous works include:

- [\[P2996R13\]](#) reflects declarations and types, and explicitly defers expression reflection to a future paper (its authors anticipate a `^^(expr)` spelling). It proposes no statement or function-body reflection.
- [\[P2040R0\]](#) (2020) explores reflection of arbitrary expressions to support facilities such as lazy evaluation and expression decomposition. It predates the current reflection model and does not cover statement traversal or function-body reflection.
- [\[P3435R0\]](#) proposes direct reflection of expressions. Some of the facilities considered in that paper’s authors’ earlier work are left out, such as “statements traversal, expressions traversal, and querying function bodies.”
- [\[P3437R1\]](#) states design principles and guidance and does not propose an actionable implementation API.

2 Motivation

2.1 Automatic differentiation as a library

There are existing automatic differentiation tools today, but in C++ they live only outside the language, and where AD has been built into a language (Swift), it is implemented as a compiler transformation rather than as a library:

Tool class	How it works today	Cost
Compiler-based AD ([clad] , [Enzyme])	compiler plugin: Clang AST source transformation ([clad]) or LLVM IR transformation ([Enzyme])	compiler-specific tooling or additional transformation step
Language-integrated AD ([Swift] @differentiable)	built into the compiler; derivatives synthesized by a transformation on [SIL] ([Swift] Intermediate Language)	requires dedicated language features and compiler support; operates on an IR rather than on the source-level function
Operator-overloading AD (dual numbers, tapes; [PyTorch] autograd)	specialized types / runtime recording	changes how the function is evaluated; runtime bookkeeping
Tracing / staged AD ([JAX])	function executed on abstract tracer values to record a functional IR then JIT-compiled ([XLA])	host-language control flow must be rewritten with library primitives; tracing and compilation at run time

```
double poly(double x) { return 2*x*x*x + x; } // an ordinary function

// derivative generated at compile time by reading poly's body:
double dydx = ad::forward_derivative<poly>(3.0); // == 6*x*x + 1 at x=3 + 55
```

`ad::forward_derivative` is a library of `consteval` functions that reflects the function body, walks the statements and expressions to build a small graph of the computation, applies the chain rule, and emits the derivative.

Operating in native C++ avoids the need to switch to the IR level when debugging AD programs, something that might be necessary when using external tools such as compiler plugins. Such plugins, which operate after the optimizations are done by the compiler, can struggle to reconstruct the types used in source code, e.g., small struct returned by value becoming an IR aggregate after inlining [EnzymeIssue2931]. This and other differences are expressed in the table above, where reflection of function bodies can help alleviate the costs listed.

Python’s experience is instructive. Google’s [Tangent] performed source-to-source AD on Python’s syntax tree, but without the type and overload information that semantic analysis provides; it is no longer maintained. The ecosystem instead converged on two libraries that dominate AD in machine learning today, [PyTorch]’s autograd and [JAX]. However, neither reads the function body. [PyTorch] records a tape of tensor operations as they execute and rebuilds it on every call. [JAX]’s grad, vmap, and jit are composable library transformations on a functional IR, obtained by tracing (running the function on abstract values): [JAX] sees only the operations performed on its own array types, not the function’s control flow, and data-dependent branches and loops must be rewritten with library primitives (`lax.cond`, `lax.scan`).

Body reflection provides the information that [Tangent] lacked and that [PyTorch] and [JAX] obtain through other mechanisms: a semantically analyzed function body, with types, overloads, and control flow resolved, available to a library at compile time and over ordinary types such as `double`. In addition, the computation graph is built as compile-time data rather than encoded in types, avoiding the instantiation and symbol growth of expression-template approaches.

2.2 Getting more information from test assertions

Test frameworks construct detailed failure diagnostics by decomposing comparison expressions so that individual operands can be evaluated and displayed (Catch2’s `REQUIRE(a == b)` provides an excellent example of this) [Catch2]. Current implementations rely on operator-overloading mechanics that capture top-level operations into temporary proxy types. This pattern restricts introspection to the immediate macro argument, failing when handling nested logical operations, subexpression function invocations, or custom operator overloads.

An example of what an assertion using reflection on expressions could look like is shown below. Here the `MY_ASSERT` macro takes a reflection over the function to be asserted on at compile time, and, in the case of failure, can walk the syntax tree, producing more informative output [Assertion example, Godbolt].

```
bool in_range(double x, double lo, double hi) {
    return lo <= hi && x >= lo && x <= hi;
}

int main() {
    double x = 5, lo = 10, hi = 1;
    MY_ASSERT(in_range(x, lo, hi));
}

>> test.exe

MY_ASSERT(in_range(x, lo, hi)) failed at line 340
  where x = 5, lo = 10, hi = 1
  inside in_range(x = 5, lo = 10, hi = 1):
    lo <= hi                false
    x >= lo                 (not evaluated)
    lo <= hi && x >= lo     false
```

```
x <= hi                                (not evaluated)
lo <= hi && x >= lo && x <= hi  false
```

2.3 ORMs and mapping C++ to query languages

Existing C++ Object-Relational Mapping (ORM) frameworks, such as [\[sqlpp11\]](#), translate C++ syntax to SQL statements through operator overloading. These implementations capture expression trees by wrapping operands in custom proxy types during runtime evaluation. While effective for overloadable operations, this approach fails for language constructs that cannot be overloaded, including conditional expressions, and for operators such as `&&` and `||` whose short-circuit evaluation is lost once overloaded.

Reflecting on expressions can address these limitations. The example below shows how we could use reflection upon a conditional operator to convert native C++ code to SQL.

```
bool overdue(int days_open, bool member) {
    return days_open > (member ? 60 : 30);
}
static_assert(sql::where<^overdue>() ==
    "days_open > CASE WHEN member THEN 60 ELSE 30 END");
```

2.4 Mutation testing

Mutation testing is a technique that changes source code and checks whether the tests now fail. It is used to quantify test quality. There are existing mutation testing tools for C++ that use LLVM-based approaches [\[Mull\]](#). With reflection on expressions, these mutations could be generated at compile time, simplifying the steps required to set up such tools and lowering the barrier to adopting this technique.

```
int fib(int n) { return n < 2 ? n : fib(n - 1) + fib(n - 2); }
void test_fib() {
    Assert::equal(fib(30), 832040);
    Assert::equal(fib(1), 1);
    Assert::equal(fib(0), 0);
}
mut::mutation_test<^test_fib>();
```

2.5 Hardware-friendly code transformation for HPC

High-performance computing applications routinely reorganize algorithms to optimize for underlying hardware layouts: unrolling inner loops to maximize instruction-level parallelism, or converting array-of-structs (AoS) representations into struct-of-arrays (SoA) patterns for vectorized, cache-efficient memory access. Current approaches rely either on manual refactoring, migrating source code to layout-abstraction frameworks like [\[Kokkos\]](#) (which replaces direct accessors with dedicated API syntax rather than transforming existing implementations in place), or on compiler-driven loop transformations, which remain non-portable and opaque when optimization heuristics fail.

Reflecting on statements and expressions enables these optimizations via ordinary `consteval` metaprogramming. Consider a kernel defined over a standard array-of-structs model:

```
struct Particle { double x, y, z; };
void update(Particle& p) { p.x += p.y * 0.5; }
```

By introspecting `update`'s body, we can identify member accesses `p.x` and `p.y`, allowing us to generate an equivalent struct-of-arrays implementation for `update`, which can operate over contiguous `x` and `y` buffers rather than individual `Particle` instances. A fully working library would need expression splicing for that last step; see [Expression splicing](#) for why that's scoped as a later paper building on this one.

3 Proposed features

3.1 Function body reflection

Before we can reflect over function bodies, we need to obtain a function body. We propose adding `body_of`. For a given `std::meta::info r` that is a function (satisfies `is_function(r)`) and has a definition with a reachable body of statements (satisfies `has_body(r)`), `body_of(r)` will return a `std::meta::info` representing the function body.

```
namespace std::meta {
    consteval bool      has_body(info fn);
    consteval info      body_of(info fn);
}
```

3.2 Statement reflection

A function body is a sequence of statements executed in order. We propose that reflection on function bodies should let you introspect each statement in a function.

```
namespace std::meta {
    consteval vector<info> statements_of(info);
}
```

`statements_of` acts on any compound statement, including the function body obtained from `body_of`. We propose adding a `is_compound_statement` metafunction to check if you have a block of statements you can decompose with `statements_of`.

```
namespace std::meta {
    consteval bool      is_compound_statement(info);
}
```

`statements_of` will return a vector of `std::meta::info`, each representing an individual statement. In a given compound statement, `statements_of` will return the statements in the order that they are written in the block. Statements are only returned if they are not discarded. In the example below, `statements_of(body_of(~doSomething<int>))` returns only one statement; the discarded `else` branch is not returned.

```
template <typename T>
void doSomething(T t) {
    if constexpr (std::is_arithmetic_v<T>) {
        std::cout << "value: " << t << '\n';
    } else {
        std::cout << "cannot render value\n";
    }
}
```

We also propose adding an `is_statement` metafunction to identify that a given `std::meta::info` is a statement. Any `std::meta::info r` returned from `statements_of` will satisfy `is_statement(r) == true`.

```
namespace std::meta {
    consteval bool      is_statement(info);
}
```

We propose adding the following metafunctions to help identify the type of each statement returned from `statements_of`. Each statement inside a compound statement is either another compound statement, one of the statement types below, or another kind of statement not yet covered by this proposal (see [Abstraction levels](#)).

```
namespace std::meta {
    consteval bool      is_declaration_statement(info);
    consteval bool      is_expression_statement(info);
}
```

For the rest of this proposal we will only consider declaration and expression statements, with the exception of a return statement. A return statement is a kind of jump statement. The other jump statements will be left for a later paper. However, for AD and other applications, being able to reflect on the function return value is vital. We propose adding an `is_return_statement` metafunction to identify the return statement in a compound statement.

```
namespace std::meta {
    consteval bool      is_return_statement(info);
}
```

The other statement types are outside the scope of this proposal, and reflection over them can be addressed in a follow-up proposal. Within the scope of this proposal, it is possible to get all statements from a function body, and for declarations and expressions, we will describe how to introspect these below. Other types of statements can be identified, but no information is extracted from them at this point.

3.3 Declaration statement reflection

For now we will only consider reflection over simple declarations. A simple declaration here is a list of declarators with their initializers. We propose adding an `is_simple_declaration` metafunction to determine if a given declaration statement is a simple declaration. Other types of declaration can be considered separately in the future, and can be distinguished using similar approaches to `is_simple_declaration`.

```
namespace std::meta {
    consteval bool      is_simple_declaration(info);
}
```

For a `std::meta::info` that satisfies `is_declaration_statement` and `is_simple_declaration`, we propose adding `declarators_of` and `initializers_of`, which return a vector of `std::meta::info` representing reflections over the declared variables, and the initializers of those variables respectively. Structured bindings declarations (`auto [a, b] = pair;`) are excluded for now, since all the declarators share the same initializer; supporting them is deferred to future work.

```
namespace std::meta {
    consteval vector<info>      declarators_of(info);
    consteval vector<info>      initializers_of(info);
}
```

3.4 Return value reflection

For a given `std::meta::info` that satisfies `is_return_statement`, we propose adding a `return_operand_of` metafunction. Return statements either return an expression or a braced initializer list. `return_operand_of` will return a `std::meta::info` representing the returned expression, or, when a braced initializer list is returned instead, the resolved constructor call. How to represent the initializer list itself is not yet defined by this proposal; this is an open question left for future work.

```
namespace std::meta {
    consteval info      return_operand_of(info);
}
```

3.5 Expression reflection

We start by proposing the ability to extract the expression from an expression statement. Given a `std::meta::info` that satisfies `is_expression_statement`, `expression_of` will return a `std::meta::info` for the expression itself.

```
namespace std::meta {
    consteval info      expression_of(info);
}
```

We propose adding an `is_expression` metafunction to determine if a given `std::meta::info` is an expression at all. Expressions returned from `expression_of` will satisfy this, along with expressions returned from `return_operand_of`, and expressions in `initializers_of` for declarations.

```
namespace std::meta {
    consteval bool      is_expression(info);
}
```

An expression is a sequence of operators and operands. Given an expression, we propose adding the ability to extract its top-level operator and operands. This top-level operator can be a recognized operator type, or a function. We propose adding `is_operator_expression` and `is_function_call` to distinguish between the two.

In the case that `is_operator_expression` returns true, `expression_operator_of` will return an enum from `std::meta::operators`. We propose `expression_operator_of` as a separate function rather than overload `operator_of` from P2996, since they have distinct semantics despite both relating to operators. `std::meta::operators` only covers operators that can be overloaded, so it cannot represent member access (`.` or `->`) or casts. These will likely need their dedicated predicates in a future revision.

In the case that `is_function_call` returns true, `callee_of` will return a `std::meta::info` representing the function expression being called. It is possible for a given `std::meta::info` to satisfy both `is_function_call` and `is_operator_expression`, for example, `operator+(int, int)`. In this case, or for an overloaded operator for a custom class, the `callee_of` metafunction will return a reflection for the built-in/overloaded operator function, and the `expression_operator_of` will identify which operator it is.

`callee_of` always returns the raw callee expression [P2825R5], and resolving it into a declaration requires using `is_id_expression` and `declaration_of`, the same way any other name can be resolved.

`operands_of` will return the arguments of the operator or function call. In a call that uses default arguments, `operands_of` returns one operand for each function parameter, including defaulted arguments.

```
namespace std::meta {
    consteval bool      is_operator_expression(info);
    consteval operators  expression_operator_of(info);
    consteval bool      is_function_call(info);
    consteval info       callee_of(info);

    consteval vector<info> operands_of(info);
}
```

Each operand will be a `std::meta::info`. We propose adding an `is_literal` metafunction to check if a given `std::meta::info` operand is a literal value, and `is_id_expression` to check if an operand is an identifier. If an operand is an identifier, we propose adding `declaration_of` to extract the variable from that (which would match the `std::meta::info` returned from `declarators_of` for declarations). Operands can also be other expressions. The proposed `is_expression` metafunction can be used to identify these and extract the respective operands too.

```
namespace std::meta {
    consteval bool      is_literal(info);
    consteval bool      is_id_expression(info);
    consteval info       declaration_of(info);
}
```

Consider the conditional operator. The conditional operator can be thought of as an operator with three operands.

```
int max(int x, int y) { return (x > y) ? x : y; }

constexpr info e = return_operand_of(statements_of(body_of(^~max))[0]);

static_assert(is_expression(e));
static_assert(operands_of(e).size() == 3);
static_assert(is_expression(operands_of(e)[0]));
```

3.6 Operators

`expression_operator_of` returns an enum from `std::meta::operators`. We propose extending this enum to add all possible operators, such as the conditional operator and eventually cast operators.

Existing operators have different meanings depending on how they're applied in expressions; the key example is increment/decrement operators that can be applied in prefix or postfix, with different effects. The existing `std::meta::operators` cannot distinguish between these two expressions, so we propose adding `is_prefix` / `is_postfix` for that.

```
namespace std::meta {
    consteval bool    is_prefix(info);        // `++a`/`--a`
    consteval bool    is_postfix(info);       // `a++`/`a--`
}
```

4 Design decisions & discussion

4.1 Abstraction levels

When thinking about the API to propose, we were aiming for something that would enable developers to iterate over the reflected AST and inspect individual nodes, while keeping the distance from the actual AST representation in compilers. The reason for this is twofold: on the one hand, to make sure reflections are implementable on all C++ compilers regardless of their internal representation, and on the other hand to enable future language extensions and modifications without breaking the reflection API.

The introspecting metafunctions `is_*` are extendable. A new one can be added as needed, while the scope of an existing one can be narrowed so that it returns false for cases covered by the new metafunction. However, we also need to consider what happens if the definition of things changes, for example if `is_simple_declaration` changes its scope in future standards. That would mean that after recompilation with a new compiler, the function might return a different value. To some extent it is unavoidable, and has to be dealt with on a case-by-case basis, but trying to avoid overspecification should help.

Some examples of such `is_*` metafunctions that we have considered, but decided to omit for now, are listed below:

```
namespace std::meta {
    consteval bool    is_iteration_statement(info);
    consteval bool    is_jump_statement(info);
    consteval bool    is_labeled_statement(info);
    consteval bool    is_selection_statement(info);
}
```

Keeping `body_of`, `statements_of`, `callee_of`, `operands_of` abstract enough to fit different kinds of implementation while still giving developers the tools to iterate over the AST is one of the objectives of the API design. `is_prefix` / `is_postfix` can fall in this category; the alternative then would be to modify the existing enum entries in `std::meta::operators` to reflect the difference.

`has_body` is another function that we've decided to include to be able to check if `body_of` can be safely applied on a reflected function. Alternatively this can be replaced by checking whether the result of `body_of` is empty, although this may be confused for a function with a genuine empty body.

Another example would be `is_simple_declaration`, which follows <https://en.cppreference.com/cpp/language/declarations> definition.

4.2 The order of the transformations

To avoid potentially confusing situations, reflection on function bodies should start after the template instantiation. Consider the example in [Statement reflection](#). The developer reflecting on `doSomething`'s body here shouldn't have to deal with a discarded if constexpr branch that disappears at compile time. Similarly, template substitutions should already take place before the reflection is evaluated by the compiler, e.g., to resolve operator overloads.

4.3 Expression splicing

This paper deliberately draws a line at code inspection and defers splicing for expressions to future papers. Splicing relocates code from where it was written to a different context, and that relocation raises a distinct cluster of design questions that should be thoroughly investigated in a dedicated paper.

5 Implementations and demonstrations

A basic subset of features including but not limited to `body_of`, `statements_of`, `operands_of`, `return_operand_of` are implemented in a fork of the P2996 Clang reference implementation and in a GCC fork.

5.1 Automatic differentiation library

Using the compiler forks, we have created a library that can be used for automatic differentiation. Our example library demonstrates just one approach to automatic differentiation using the proposed reflection API. It reflects the statements and expressions in a function body and uses them to construct a directed acyclic computation graph (DAG) at compile time. Each node in the DAG represents an expression, its operator and operands for differentiation. By transforming these nodes according to differentiation rules, the library synthesizes a new function corresponding to the exact derivative of the original.

A full example can be seen on Godbolt [\[AD example, Godbolt\]](#). An extract is shown below. Given a function, we can obtain the function parameters, the statements of the function body and its return value to create a DAG that can then be differentiated.

```
namespace m = std::meta;
constexpr std::vector<Node> build(m::info fn) {
    Builder b;
    for (m::info p : m::parameters_of(fn)) {
        b.names.push_back(p);
        b.slots.push_back(b.emit({Op::Input}));
    }
    for (m::info s : m::statements_of(m::body_of(fn))) {
        if (m::is_declaration_statement(s)) { // double v = expr, w = expr2;
            auto decls = m::declarators_of(s);
            auto inits = m::initializers_of(s);
            for (std::size_t i = 0; i < decls.size(); ++i) {
                b.names.push_back(decls[i]);
                b.slots.push_back(b.lower(inits[i]));
            }
        } else // return expr;
            b.lower(m::return_operand_of(s));
    }
    return b.nodes;
}
```

5.2 AD benchmarks

We benchmarked our automatic differentiation implementation using reflection on expressions against [\[Enzyme\]](#) (a Clang-based tool). We did this for the Black-Scholes formula for the call price of an option defined below.

```
inline double total_vol(double v, double T) { return v * std::sqrt(T); }

inline double call_price(double S, double K, double v, double T) {
    double totalvol = total_vol(v, T);
    double d1 = std::log(S / K) / totalvol + totalvol * 0.5;
    double d2 = d1 - totalvol;
    return S * mcsim::CDF(d1) - K * mcsim::CDF(d2);
}
```

The benchmarks below show that the LLVM IR-based [Enzyme] approach has a similar runtime cost to the reflection on expressions approach.

Derivative of <code>call_price</code>	Reflection Approach	Enzyme
Forward mode AD	63 ns	64 ns
Reverse mode AD	73 ns	71 ns

- **AD library, demos, and benchmarks:** https://github.com/CppReflectionAD/cpp_reflection_ad
- **Clang compiler fork** (this proposal’s implementation), branch `expr-reflect`: <https://github.com/siqiyyy/clang-p2996/tree/expr-reflect>
- **GCC compiler fork** (`body_of`, `statements_of` and others implementation), branch `autograd-capable`: <https://github.com/yesmanchyk/gcc-mirror/tree/autograd-capable>

6 References

- [AD example, Godbolt] Example of building a DAG using reflection on expressions for automatic differentiation.
<https://godbolt.org/z/KcMo8cs1b>
- [Assertion example, Godbolt] Custom assertion example using expression reflection.
<https://godbolt.org/z/rxMnqfav4>
- [Catch2] Catch2: A C++ test framework with expression decomposition in assertions.
<https://github.com/catchorg/Catch2>
- [clad] Vassil Vassilev. Clad: Automatic Differentiation for C++.
<https://github.com/vgvassilev/clad>
- [Enzyme] William Moses. Enzyme: High-Performance Automatic Differentiation of LLVM.
<https://github.com/EnzymeAD/Enzyme>
- [EnzymeIssue2931] Cannot deduce type of aggregate phi for small float struct, EnzymeAD/Enzyme issue #2931.
<https://github.com/EnzymeAD/Enzyme/issues/2931>
- [JAX] JAX.
<https://jax.dev>
- [Kokkos] Kokkos: Performance-portable manycore programming model with layout-abstracting containers.
<https://github.com/kokkos/kokkos>
- [Mull] Mull: an LLVM-based mutation testing framework.
<https://mull.readthedocs.io/en/latest/index.html>
- [P2040R0] Corentin Jabot. 2020-01-13. Reflection-based lazy-evaluation.
<https://wg21.link/p2040r0>
- [P2825R5] Gašper Ažman. 2025-03-17. Overload resolution hook: `declcall( unevaluated-call-expression )`.
<https://wg21.link/p2825r5>
- [P2996R13] Barry Revzin, Wyatt Childers, Peter Dimov, Andrew Sutton, Faisal Vali, Daveed Vandevoorde, Dan Katz. 2025-06-20. Reflection for C++26.
<https://wg21.link/p2996r13>
- [P3435R0] Jean-Baptiste VALLON HOARAU, Joel FALCOU. 2024-10-14. Reflection and meta-programming.
<https://wg21.link/p3435r0>
- [P3437R1] Herb Sutter. 2024-11-07. Proposed principles: Reflect C++, generate C++ (by default).
<https://wg21.link/p3437r1>
- [PyTorch] PyTorch.
<https://pytorch.org>
- [SIL] SIL: Swift Intermediate Language.
<https://github.com/swiftlang/swift/blob/main/docs/SIL/SIL.md>
- [sqlpp11] sqlpp11: A type safe SQL template library for C++.
<https://github.com/rbock/sqlpp11>
- [Swift] Swift.
<https://www.swift.org>
- [Tangent] Tangent.
<https://github.com/google/tangent>

[XLA] XLA.
<https://openxla.org/xla>