Mojo Compile-time

Reasoning and optimization: the facts you give the compiler, and what it does with them

Mojo language reference
v1.0.0b2

keyword type built-in "string" number operator @decorator True/False/None *# comment*

MOJO COMPILE-TIME compile-time Mojo is just Mojo

One language. Compile time and runtime use the same Mojo.

The compiler acts on what it can prove, using constraints and conformance rules in code.

Proven facts enable or disable methods, constructors, and conformances.

Compile-time computation shifts expensive work out of runtime.

PARAMETERS: TYPES AND VALUES [ ] compile-time parameters, ( ) runtime arguments

```
def repeat[T: Copyable, n: Int](x: T) -> InlineArray[T, n]:
    ... # T is a type, n is a value
struct Matrix[dtype: DType, rows: Int, cols: Int]: ...
```

Use parameters to write generic algorithms and structs. The compiler creates a concrete implementation for each unique set of parameter values.

WHERE: PROVE, THEN GATE on traits or parameter facts

```
def sort(mut self) where conforms_to(Self.T, Comparable): ...
struct Box[T](Writable where conforms_to(T, Writable)): ...
def chunk[w: Int]() where w.is_power_of_two(): ...
```

A precondition the compiler must prove, such as trait facts, numeric truths, or a DType's kind, before the call compiles.

CONFORMANCES PROVE CAPABILITIES check and specialize

```
def largest[T: Comparable & Copyable](xs: List[T]) -> T:
    ... # only operations T guarantees will compile
```

A trait conformance guarantees what capabilities the parameters support, so only valid code compiles.

CONDITIONAL AVAILABILITY exists only when it can

```
struct Buffer[T: Copyable, n: Int](
    Writable where conforms_to(T, Writable), # conforms only if T does
):
    def first(self) -> Self.T where Self.n > 0: ... # method only if n >
       0
```

A type, method, conformance, or comptime declaration is available only when the compiler can prove its condition. The API is correct by construction: a missing capability or unmet constraint means calls with invalid parameters won't compile.

REFLECT A TYPE read a type's shape

```
comptime name = reflect[T].name() # also .field_count(), .field_names(),
      ...
comptime t = type_of(x) # the type of an expression
```

reflect[T] reads a type's structure and **type_of** an expression's type, so generic code adapts to any shape.

RUN CODE AT COMPILE TIME any function, no marker

```
def meters(ft: Float64) -> Float64:
    return ft * 0.3048
comptime track = meters(100.0) # runs while compiling, baked in
```

Every fact must be established at compile time.

COMPILE-TIME NUMERIC PRECISION literals stay exact

```
comptime MAX = 2 ** 200 # arbitrary-precision integer
comptime c = 0.1 + 0.2 # 0.3 exactly: a literal, kept exact
var r = 0.1 + 0.2 # a Float64, subject to rounding
```

Literals stay exact. **Float64** rounds values like **0.1**, and repeated computations accumulate rounding error. Compute accuracy-sensitive constants at compile time.

FRONT-LOAD COMPUTE move work to compile time

```
def slow_calc() -> Float64:
    ... # an expensive calculation
comptime FACTOR = slow_calc() # computed while compiling, baked in
```

Run expensive computation once while compiling; the result is baked in, free at runtime. For tables and other compile-time data, **global_constant** gives O(1) access without materializing them each time.

COMPTIME IF / FOR branch and unroll early

```
comptime if is_nvidia_gpu(): # only the live branch compiles
    use_nvidia()
else:
    use_fallback()
comptime for i in range(4): # fully unrolled
    process[i]()
```

comptime if compiles the live branch only; **comptime for** unrolls, removing loop overhead.

QUERY THE TARGET size, alignment, SIMD width

```
comptime w = simd_width_of[DType.float32]() # lanes that fit a register
size_of[T]() align_of[T]() # layout, at compile time
```

sys.info answers machine questions at compile time, so one source adapts to every target.

MATERIALIZATION bring comptime values to runtime

```
comptime table = [3, 5, 7, 11, 13] # a comptime List (heap-backed)
var t = materialize[table]() # -> a runtime List; you choose when it
      allocates
ref g = global_constant[POWERS]() # POWERS: a fixed scalar table, read
      g[i], no copy
```

Scalars materialize automatically; heap-backed values (**List**, **Dict**) need **materialize**. **global_constant** keeps one static copy to index.

COMPTIME MEMBERS comptime lives on types too

```
struct Stack[T: Copyable]:
    comptime Element = Self.T # associated type
    comptime capacity = 1024 # comptime value member
comptime Scalar[dt: DType] = SIMD[dt, 1] # parametric alias
```

A type carries its own compile-time members (values, associated types, and parametric aliases), reached through **Self**.

INLINING force or forbid

```
@always_inline
def lerp(a: Float64, b: Float64, t: Float64) -> Float64:
    return a + (b - a) * t # expanded at every call site
@no_inline
def cold_path(): ... # kept as a real call
```

Inlining replaces a function call with the function body, reducing call overhead for small, frequently called functions. Add **@always_inline** to request inlining, **@no_inline** to exclude the option, or let the compiler decide.

THE COMPILE-TIME BOUNDARY what can't happen early

Everything used in compile-time code must be known at compile time. A compile-time value, parameter, **if**, or **for** can't depend on runtime input.

At compile time, you can't perform file I/O, make foreign calls, or call functions that can raise.

Compile-time code runs on the CPU, like all compilation.