



Mojo compile-time cheat sheet

Reasoning and optimization. What the compiler knows and does with facts.

v1.3dev

MOJO COMPILE-TIME compile-time Mojo is just Mojo

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

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

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

Compile-time computation shifts expensive work out of runtime.

PARAMETERS: TYPES AND VALUES

[] compile-time parameters, () runtime arguments

Use parameterized declarations for functions and structs.

```
def repeat[T: Copyable, n: Int](x: T) -> Array[T, n]:
    ... # T is a type, n is a value

struct Matrix[dtype: DType, rows: Int, cols: Int]:
    ...
```

The compiler creates a concrete implementation for each unique set of parameter values.

WHERE: PROVE, THEN GATE on traits or parameter facts

`where` is a precondition the compiler must prove. For example: trait facts, numeric truths, or a DType's kind, before the call compiles.

An optional message you provide appears in the output.

```
def sort(mut self) where conforms_to(Self.T, Comparable):
    # sort what you can compare

struct Box[T: AnyType](
    Writable where conforms_to(T, Writable)):
    # synthesized representations for `T`'s that support it

comptime Kernel = def[w: Int](Int) thin -> None
where (
    w > 0
) else "'w' must be positive"
```

`where` gates access to code, and the compiler enforces it. A type can shape itself in ways traditional generics can't, such as offering `average()` only when its elements are numbers, and not, for example, `Bool` values.

CONFORMANCES PROVE CAPABILITIES

check and specialize

```
def largest[T: Comparable & Copyable & Deinitable](
    xs: List[T]) -> T:
    # operations that 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

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

A type, method, conformance, or comptime declaration is available when the compiler can prove its conditions.

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. Parameterized 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
```

Move work to compile time. Run expensive computation once while compiling for fixed and free runtime access.

```
def slow_calc() -> Float64:
    ... # an expensive calculation
comptime FACTOR = slow_calc() # compute at compile time
```

For tables and other compile-time data, `global_constant[]()` gives $O(1)$ access without materializing them each time.

COMPILE-TIME NUMERIC PRECISION

 literals stay exact

```
comptime MAX = 2 ** 200 # arbitrary-precision
integer
comptime c = 0.1 + 0.2 # 0.3 exactly: literals stay
exact
var a = 0.1 # a Float64, subject to rounding
var r = a + 0.2 # 0.30000000000000004
```

Float64 rounds values like 0.1, and repeated computations accumulate rounding error.

Compute accuracy-sensitive constants at compile time.

COMPTIME MEMBERS

 comptime lives in 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`.

COMPTIME IF / FOR

 branch and unroll early

```
comptime if is_nvidia_gpu(): # only the live branch
  is kept
  use_nvidia()
else:
  use_fallback()

comptime for i in range(4): # fully unrolled
  process[i]()
```

`comptime if` keeps only the live branch, but calls that depend on parameters must be provable in every branch. `comptime for` unrolls, removing loop overhead.

COMPTIME MATCH

 specialize by pattern

```
comptime match (m, n): # pick a path by pattern
  case (16, 16):
    use_tuned_16x16()
  case (mt, nt) if mt * nt <= 256: # names bind
    as parameters
    use_small_tile(mt, nt)()
  case _:
    use_generic()
```

The subject must be a parameter. A guard is proof for its case, so it satisfies a matching `where` on the call. An unmatched subject skips the match, so add `case _` when every path must return.

INLINING

 force or forbid compiler substitution

```
@inline(.always)
def lerp(a: Float64, b: Float64, t: Float64) ->
Float64:
  return a + (b - a) * t # expanded at every call
site

@inline(.never)
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.

`@inline(.always)` requests inlining

`@inline(.never)` excludes inlining

`@inline(.automatic)` lets the compiler decide

`@inline(.nDEBUG)` inlines and drops inlined debug info

INLINE BY PARAMETER

 each instantiation chooses

One definition, inlined or not per construction.

```
# any comptime InlineLevel, including a parameter
@inline(policy)
def scaled[policy: InlineLevel](x: Int) -> Int:
  return x * 2

scaled[.always](3) # inlined

scaled[.never](3) # kept as a call
```

QUERY THE TARGET

 size, alignment, SIMD width

`sys.info` answers machine questions at compile time.

```
from std.sys.info import simd_width_of, size_of,
align_of
from max.gpu.host import default_accelerator

# lanes that fit a register
comptime w = simd_width_of[DType.float32]()

# layout, at compile time
size_of[T]() ; align_of[T]()

# ask about another target: for example,
# the accelerator, from host code
comptime gpu = default_accelerator()
simd_width_of[DType.float32, gpu]()
```

One source can adapt to every target. Use a `CompilationTarget` to ask about a target other than the one being compiled.

MATERIALIZATION

 bring comptime values to runtime

Materializing brings the values of compile-time literal expressions, tables, and constants to runtime, allowing them to be used there:

```
# a comptime List (heap-backed)
comptime table: List[Int] = [3, 5, 7, 11, 13]

# -> a runtime List; you choose when it allocates
var t = materialize[table]()

# POWERS: a fixed scalar table, read g[i], no copy
ref g = global_constant[POWERS]()
```

Scalars materialize automatically

Heap-backed values (`List`, `Dict`) need `materialize[]()`

`global_constant[]()` (from `std.builtin.globals`) keeps one static copy to index.

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, `comptime if`, or `comptime 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.