

```
(lam x [x x])
```

Blazing-Fast Lambda Calculus via **Metaprogramming**

SHAMELESS PLUG

Lantr builds **Scalus**: Cardano smart contracts in Scala 3



Scala → UPLC

A compiler plugin turns plain Scala 3 into Plutus scripts.



One codebase

The same code runs on-chain and off-chain, on the JVM and in JavaScript.



Fast tooling

Evaluator, testkit, emulator — and the JIT from this talk.

THE PROBLEM

A smart contract is a pure function that thousands of computers must agree on

```
def validator(datum: Data, redeemer: Data, ctx: ScriptContext): Unit
// returns → the transaction is valid · fails → it is rejected
```

Deterministic

Same answer on every node, every OS, forever.

Metered

Every step costs CPU and memory budget, paid in fees.

Correct

Bugs lock real money. We want proofs.

Hot

L2s, emulators, indexers and tests re-run the same scripts all day.

WHY LAMBDA CALCULUS

Want proofs? Ask Phil Wadler



With Phil Wadler · IOHK Summit 2019

- 01 **Propositions as Types:** programs are proofs. Lambda calculus is where logic and computation meet.
- 02 Three constructs, one rule. The whole semantics fits on a **napkin**.
- 03 Mechanized: Plutus Core metatheory in Agda, and a CEK machine in Lean 4 that proves theorems about compiled scripts.

Real compiler output: **fib** from Scala

```
@Compile object FibContract:  
  def fib(n: BigInt): BigInt = if n ≤ 1 then n else fib(n - 1) + fib(n - 2)
```

↓ Scalus compiler plugin, toUplcOptimized, CSE off (names shortened)

```
[(lam fib [fib fib])  
  (lam fib  
    (lam n  
      (case [(builtin lessThanEqualsInteger) n (con integer 1)]  
        [(builtin addInteger)  
          [fib fib [(builtin subtractInteger) n (con integer 1)]]  
          [fib fib [(builtin subtractInteger) n (con integer 2)]]]  
      n)))]
```

TODAY'S MINI-UPLC: EXACTLY WHAT FIB NEEDS

The smallest UPLC, as one Scala 3 enum

```
enum Term:
  case Var(name: String)
  case LamAbs(name: String, body: Term)
  case Apply(fun: Term, arg: Term)
  case Constant(value: Const)
  case Builtin(fn: DefaultFun)
  case Case(scrutinee: Term, branches: List[Term])
  case Error

enum Const:
  case Integer(value: BigInt)
  case Bool(value: Boolean)
```

Strict: arguments are evaluated before the call.

case on a Bool is lazy: False → branch 0, True → branch 1.

error: the only way to fail.

BigInt: overflowing a Long would change what a contract means.

Builtins: add, subtract, multiply, equals, lessThanEquals. Real Plutus V3 adds delay/force, constr, bytestrings, Data and ~100 builtins.

THE REFERENCE INTERPRETER

The CEK machine — on a napkin

C

Control

the term being evaluated

E

Environment

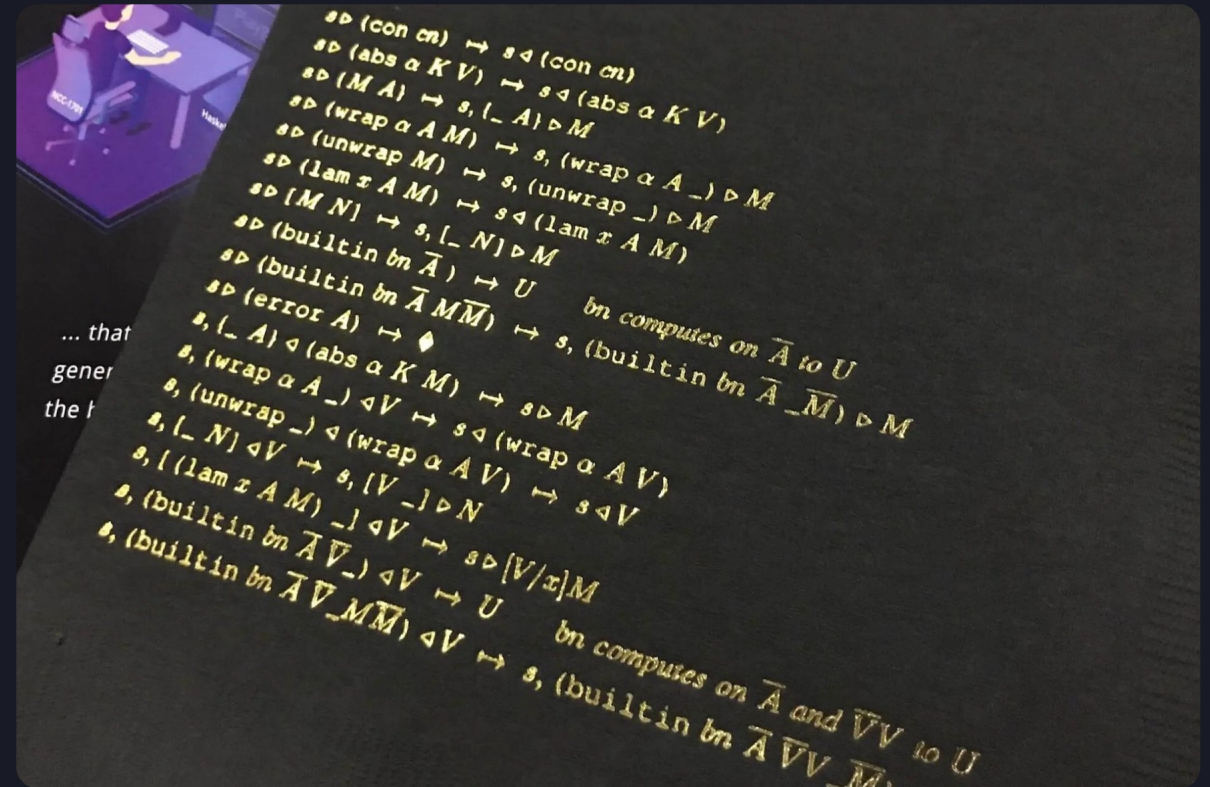
values of its free variables

K

Kontinuation

what to do with the result — a stack of frames on the heap

The Plutus spec *is* a CEK machine. Heap stack → never overflows.



PlutusFest 2018 napkin · Photo: IOHK

Compute a term, return a value. Repeat.

```
@tailrec def run(s: State, k: List[Frame]): Value = s match
  case Compute(Var(x), env)          ⇒ run(Return(lookup(env, x)), k)
  case Compute(LamAbs(x, b), env)    ⇒ run(Return(VLam(x, b, env)), k)
  case Compute(Apply(f, a), env)     ⇒ run(Compute(f, env), ApplyArg(a, env) :: k)
  case Compute(Case(c, bs), env)     ⇒ run(Compute(c, env), CaseFrame(bs, env) :: k)
  // ... constants, builtins, error
  case Return(v) ⇒ k match
    case Nil                ⇒ v
    case ApplyArg(a, env) :: up ⇒ run(Compute(a, env), ApplyTo(v) :: up)
    case ApplyTo(f) :: up    ⇒ run(applyValue(f, v), up) // λ: extend env, run body
    case CaseFrame(bs, env) :: up ⇒ run(Compute(bs(branchIndex(v))), env), up)
```

≈ 80 lines. One case per napkin rule, @tailrec makes it a loop. Correct, stack-safe, easy to audit — and it re-examines every node on every run.

PLACE YOUR BETS · HANDS UP

Same interpreter, same contracts, one machine. Which one is faster?

A Haskell

plutus-core: the reference CEK machine, tuned for years with GHC.

B Scala on the JVM

Scalus: the same CEK machine, with the JVM's JIT and garbage collector behind it.

89 real Cardano smart contracts from a public benchmark. Steady state.

THE ANSWER: A, HASKELL, BY 17 %

Haskell CEK 220 μ s · Scala CEK 256 μ s

Then 3× faster

than that Haskell VM, on the same JVM —
once we stop interpreting. Up to 9× on some contracts.

Geometric mean, all 89 plutus_use_cases scripts, same machine, 2026-10-11: Scalus JIT 71 μ s vs plutus-core CEK 220 μ s. Steady state, after compilation and warm-up.



What if we don't interpret at all?

Turn UPLC lambdas into **JVM lambdas**, builtins into **JVM arithmetic** —
at runtime.

TIER 1 · NO METAPROGRAMMING

Compile to a tree of closures

```
type Code = Frame ⇒ Any // built once, run many times
def compile(t: Term, scope: List[String]): Code = t match
  case Var(x) ⇒
    val depth = scope.indexOf(x) // resolved at compile time
    env ⇒ env.lookup(depth)
  case LamAbs(x, body) ⇒
    val b = compile(body, x :: scope)
    env ⇒ (arg: Any) ⇒ b(Frame(arg, env)) // a real JVM lambda
  case Apply(f, a) ⇒
    val (cf, ca) = (compile(f, scope), compile(a, scope))
    env ⇒ Runtime.apply(cf(env), ca(env))
```

16× faster already. But the interpreter is still hiding inside: megamorphic `Function1.apply` calls, a runtime environment, and program structure the JVM can't see.

Two operators: `'{ quote }` and `${ splice }`

```
// x^n, with the loop unrolled at runtime
def power(x: Expr[Double], n: Int)
  (using Quotes): Expr[Double] =
  if n == 0 then '{ 1.0 }
  else '{ $x * ${ power(x, n - 1) } }

given staging.Compiler = staging.Compiler.make(...)

val cube: Double => Double =
  staging.run('{ (x: Double) => ${ power('x, 3) } })
// compiled: (x: Double) => x * (x * (x * 1.0))
```

`'{ e }` is the **code** of `e`: an `Expr[T]`, not its value.

`${ e }` inserts generated code into a quote.

staging.run compiles it to bytecode — in the running JVM.

Quotes are type-checked when *your* code compiles.

Specialize an interpreter to one program and you get a compiled program.

Futamura 1971 · the first projection

Rompf & Odersky 2010 · staging in Scala (LMS)

Stucki, Biboudis & Odersky 2018 · Scala 3 quotes

Staging: an interpreter that returns code

```
def gen(t: Term, env: Map[String, Expr[Any]])(using Quotes): Expr[Any] =
  t match
    case Var(x) => env(x) // a JVM local – no environment at runtime
    case LamAbs(x, body) =>
      '{ (arg: Any) => ${ gen(body, env.updated(x, 'arg)) } }
    case Apply(f, a) =>
      '{ val fv = ${ gen(f, env) }; Runtime.apply(fv, ${ gen(a, env) }) }
    case Constant(c) => genConstant(c)
    case Error => '{ Runtime.fail() }
    case IfThenElse(c, t, e) => ... // next slide
    case SaturatedBuiltin(fn, args) => ... // next slide

def compile(t: Term): () => Any = staging.run('{ () => ${ gen(t, Map.empty) } })
```

Pattern-match the idioms away

```
// (case c else then) on a Bool ⇒ a native if, only one branch runs
case IfThenElse(c, t, e) ⇒
  '{ if ${ genBool(c, env) } then ${ gen(t, env) } else ${ gen(e, env) } }

// [(builtin addInteger) a b] ⇒ BigInt arithmetic, no currying
case SaturatedBuiltin(AddInteger, List(a, b)) ⇒
  '{
    val x = ${ gen(a, env) }
    val y = ${ gen(b, env) }
    Runtime.asInteger(x) + Runtime.asInteger(y)
  }
```

Custom unapply extractors recognise shapes in the UPLC tree.

Quotes turn each shape into the Scala you'd write by hand.

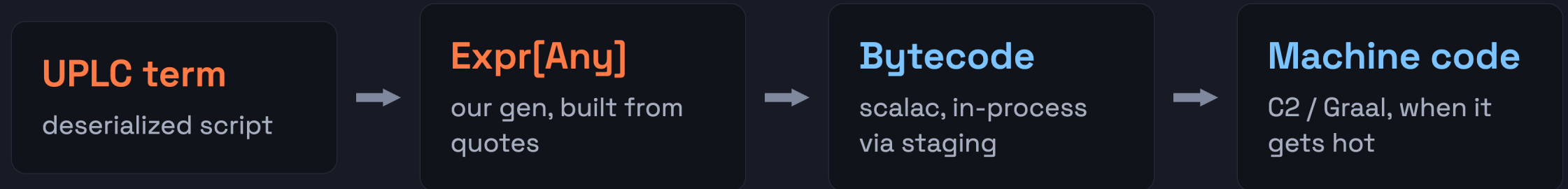
STAGEDJIT.SHOWCODE(FIB) · LIGHTLY TRIMMED

What the staged compiler writes for fib

```
((fib: Any) ⇒ ((n: Any) ⇒  
  if ({ val x = n; val y = BigInt(1L); asInteger(x) ≤ asInteger(y) })  
  then n  
  else {  
    val x = {  
      val f = Runtime.apply(fib, fib) // [fib fib]  
      Runtime.apply(f, { val a = n; val b = BigInt(1L); asInteger(a) - asInteger(b) })  
    }  
    val y = {  
      val f = Runtime.apply(fib, fib)  
      Runtime.apply(f, { val a = n; val b = BigInt(2L); asInteger(a) - asInteger(b) })  
    }  
    asInteger(x) + asInteger(y)  
  })  
)))
```

THEN THE JVM FINISHES THE JOB

Four stages, two compilers we didn't write



Inlining: small monomorphic lambdas collapse into straight-line code.

Escape analysis: temporaries never reach the heap.

Speculation: type checks become predictable branches.

LIVE DEMO · SCALA → SCALUS → UPLC → OUR ENGINES

```
$ sbt "scalusBridge/run 20"
```

Engine	μs / op	vs CEK
CEK machine	6 387	1×
Closure compiler	389	16.4×
Staged compiler	183	35.0×

fib 20 from Scala via Scalus: 21,891 recursive calls, ~0.9 M CEK steps. Same answer (6765) on every engine.

Fallback numbers from the stage laptop: Scala 3.9.0, Zulu JDK 25, 2 s warm-up + 3 s measurement, Oct 2026.

A DIFFERENT ANSWER IS A CONSENSUS BUG

Fast is easy. Fast *and exactly the same* is the job.

Untyped means Any

Every use checks its type at runtime and fails exactly where CEK fails.

Evaluation order

Evaluate all arguments first, then check them. Matters if one loops forever.

Lazy case, strict calls

Only the chosen branch runs. Every argument of a call always runs.

Budget

Plutus charges every step. Scalus emits the same costs into the code.

— sumTo 100000 —

CEK machine (con integer 5000050000)

Closure compiler **StackOverflowError!**

Staged compiler **StackOverflowError!**

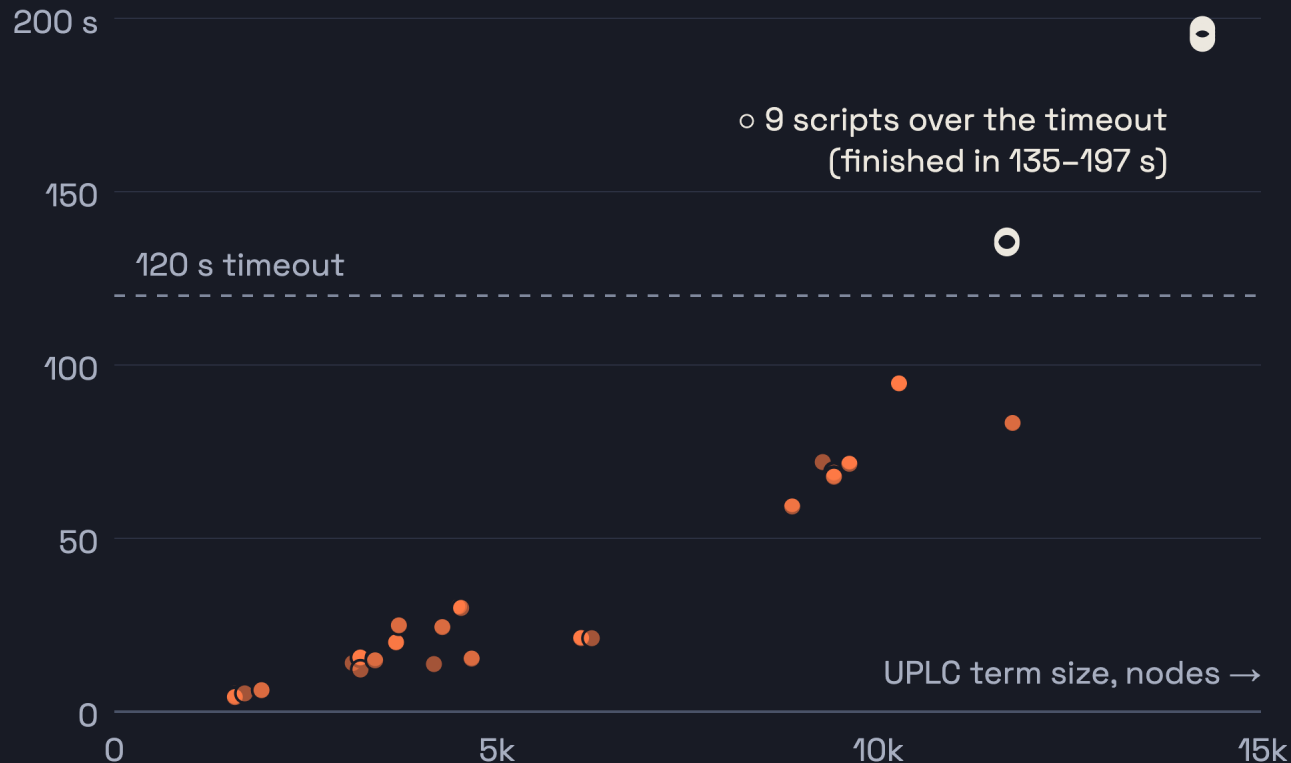
Native calls borrow the JVM stack

Scalus runs native-stack code and falls back to heap continuations when recursion gets deep.

Keep the CEK machine as the oracle: test every engine against it — **including on failures.**

THE TWIST · THE BILL

Real scripts took **minutes** to compile



22 s

median compile, 4 s to 197 s, growing faster than script size

93 %

of it in our own generator, building Exprs. scalac itself: ~2 s

175 000

runs to break even vs the CEK machine (median)

Scalus Hybrid JIT, 89 plutus_use_cases scripts, one JVM, fresh staging compiler per script. Apple M3 Max, Zulu JDK 25 (C2), Scala 3.3.8, Oct 2026.

THE INVESTIGATION

scalac wasn't slow. **We were.**

01 • profile

93 % of the time is in *our* generator, before scalac starts.

02 • shape

Log-log slope 1.45: time grows faster than the script. Extra work at every level.

03 • cause

Every `Expr` has an **owner**. Splice it somewhere else and the compiler copies the whole tree.

```
λ → val → λ → val → ... → leaf    // built bottom-up, outside its final home
```

Each level re-owns everything below it: a node at depth d is copied d times. $O(n^2)$, hidden behind a \$.

THE FIX

Create an **Expr** where it will live

BEFORE · QUADRATIC

```
case Add(a, b) ⇒  
  val ca = gen(a, env)    // built here...  
  val cb = gen(b, env)  
  '{ val x = $ca           // ...re-owned here:  
    val y = $cb           // whole tree copied  
    asInteger(x) + asInteger(y) }
```

AFTER · LINEAR

```
case Add(a, b) ⇒  
  '{ val x = ${ gen(a, env) } // built in place  
    val y = ${ gen(b, env) }  
    asInteger(x) + asInteger(y) }
```

22 s → 2.3 s

median compile · max 197 s → 7.7 s

175 000 → 16 000

runs to break even vs CEK (median)

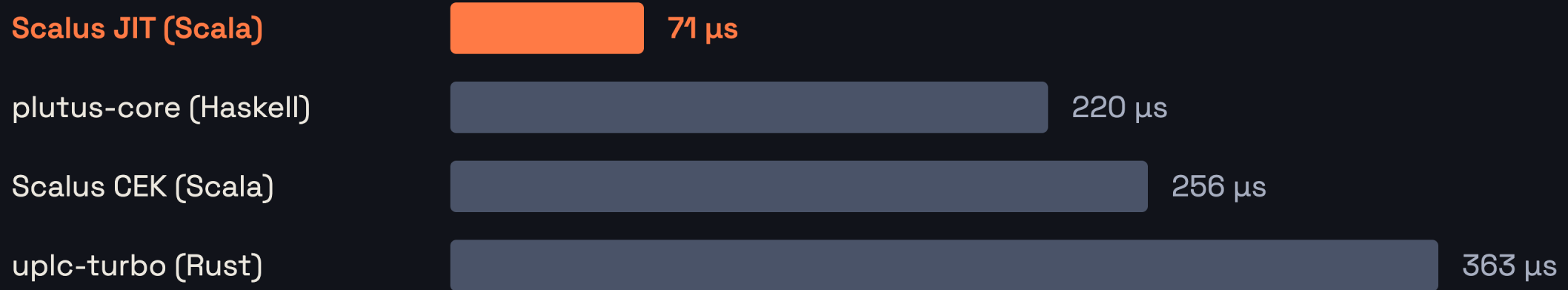
20 s → 0.17 s

our generator (median) · 93 % → 6 %
of compile

Scalus Hybrid JIT, all 89 plutus_use_cases scripts (before: the 80 that finished in 120 s). Steady speed-up unchanged: 3.9× vs CEK.

REAL WORLD · PLUTUS VM BENCHMARK

Scalus JIT vs other Plutus VMs



Per contract vs Haskell: median **4.4×** · 72 of 89 at least 2× · up to 8.9×



Geometric mean, flat-decode + evaluate, all 89 plutus_use_cases scripts (all VMs pass), one machine, 2026-10-11 · SAIB cardano-plutus-vm-benchmark

SO WHERE DOES IT PAY?

A JIT is a cache for hot scripts

top 10 scripts • 61 %

next 90 • ~29 %

100 of 2,224 scripts (4.5 %) do about 90 % of all script runs on Cardano mainnet. The other 2,124 share the rest.

Immutable

Scripts are addressed by hash.
Compile once, key by hash,
never invalidate.

Tiered

The interpreter serves cold scripts at once. Hot ones compile in the background. That makes it a JIT.

Sweet spot: L2

Hydrozoa, a Scala 3 state-channel L2,
runs a few apps' scripts all day.

Cardano mainnet, epoch 543: 117,978 blocks, 311,393 script runs, 2,224 unique scripts. Top 100 = 93–95 % of runs (Plutus V1/V2), 87 % (V3).

TAKE HOME

Pick the tier your workload pays for

Engine	fib 20	Start-up	Use it for
Interpreter (CEK)	1×	none	The spec and test oracle. Cold and one-off scripts.
Closure compiler	16×	milliseconds	Warm scripts. No metaprogramming needed.
Staged compiler	35×	seconds + warm-up	The hot few: one script, many runs.

A staged compiler is an interpreter that returns `{ code }` instead of values.

Generate simple code; let the JVM optimize. And create each `Expr` where it will live.

Speed-ups: this talk's toy engines on fib 20, stage laptop. On real contracts, Scalus JIT vs Scalus CEK: $\approx 3.6\text{--}3.9\times$.

`(case (con bool True) (error) questions)`

Thank you

Alex Nemish x.com/atlanter

SCALUS

scalus.org

TALK CODE

github.com/lantr-io/tiny-uplc-jit

LANTR

lantr.io