

21 compilers and 3 orders of magnitude in 60 minutes

a wander through a weird landscape to the heart of compilation

Spring 2019

Hello!

- I am someone who has worked (for pay!) on some compilers: rustc, swiftc, gcc, clang, llvm, tracemonkey, etc.
- Ron asked if I could talk about compiler stuff I know, give perspective on the field a bit.
- This is for students who already know roughly how to write compilers, not going to cover that!



the speaker, in 1979

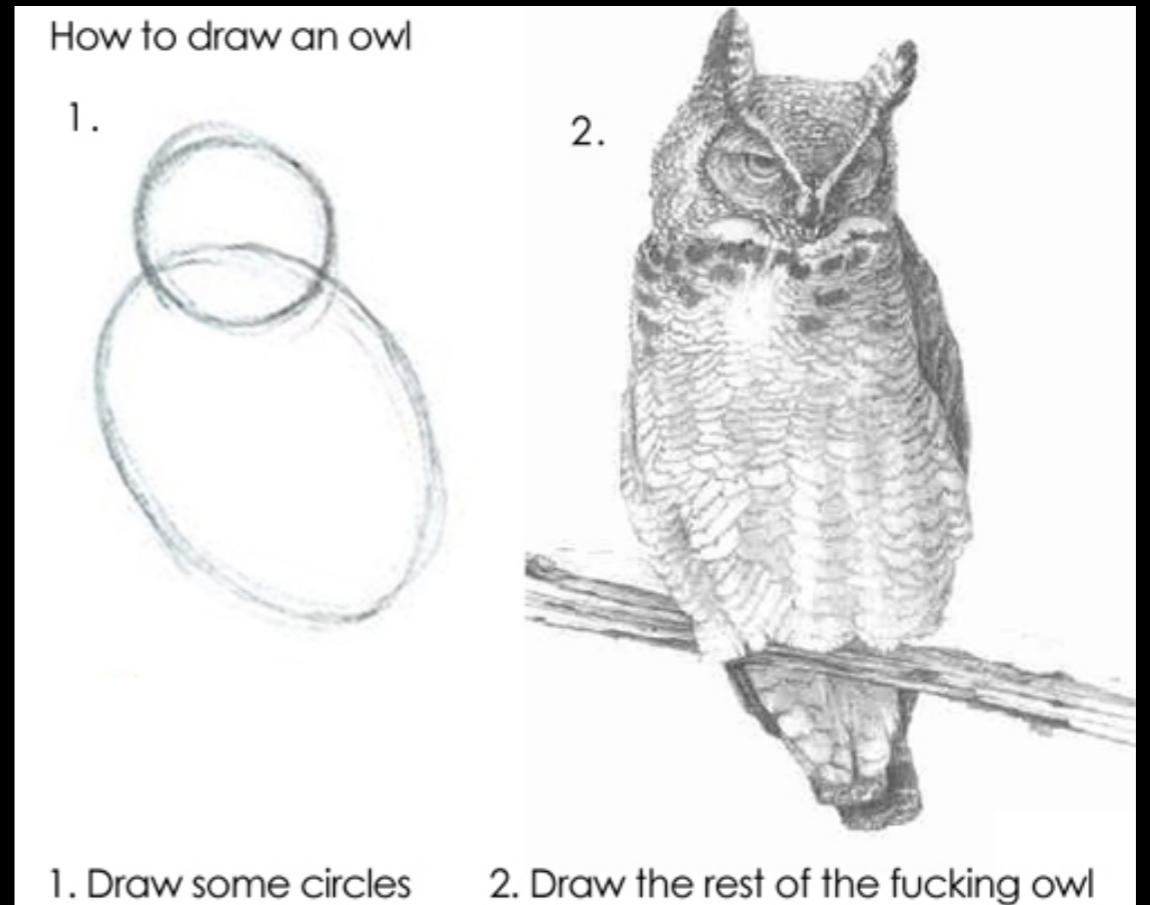
I like compilers!

- Relationship akin to "child with many toy dinosaurs".
- Some are bigger and scarier. We will look at them first.
- Some are weird and wonderful. We will visit them along the way.
- Some are really tiny!



Goal for talk

- I expect gap between class projects and industrial compilers is overwhelming.
- Want to explore space between, demystify and make more design choices clear.
- Reduce terror, spark curiosity, encourage trying it as career!
- If I can compiler, so can you!



Plan of talk

- Describe a few of the giants.
- Talk a bit about what makes them so huge & complex.
- Wander through the wilderness (including history) looking for ways compilers can vary, and examining specimens.
- Also just point out stuff I think is cool / underappreciated.

Caveats

- I'm not a teacher or very good at giving talks.
- Lots of material, not ideal to stop for questions unless you're absolutely lost. Gotta keep pace!
- But: time at end for questions and/or email followup. Happy to return to things you're curious about. Slides are numbered! Jot down any you want to ask about.
- Apologies: not as much industry-talk as I promised. Will try for some. But too many dinosaurs for show and tell!

Part 1: some giants

Specimen #1

Clang

- ~2m lines of C++: 800k lines clang plus 1.2m LLVM. Self hosting, bootstrapped from GCC.
- C-language family (C, C++, ObjC), multi-target (23).
- Single AST + LLVM IR.
- 2007-now, large multi-org team.
- Good diagnostics, fast code.
- Originally Apple, more permissively licensed than GCC.

```
LValue CodeGenFunction::EmitLValue(const Expr *E) {
    ApplyDebugLocation DL(*this, E);
    switch (E->getStmtClass()) {
        default: return EmitUnsupportedLValue(E, "l-value expression");

        case Expr::ObjCPropertyRefExprClass:
            llvm_unreachable("cannot emit a property reference directly");

        case Expr::ObjCSelectorExprClass:
            return EmitObjCSelectorLValue(cast<ObjCSelectorExpr>(E));
        case Expr::ObjCIsaExprClass:
            return EmitObjCIsaExpr(cast<ObjCIsaExpr>(E));
        case Expr::BinaryOperatorClass:
            return EmitBinaryOperatorLValue(cast<BinaryOperator>(E));
        case Expr::CompoundAssignOperatorClass:
            QualType Ty = E->getType();
            if (const AtomicType *AT = Ty->getAs<AtomicType>())
                Ty = AT->getValueType();
            if (!Ty->isAnyComplexType())
                return EmitCompoundAssignmentLValue(cast<CompoundAssignOperator>(E));
            return EmitComplexCompoundAssignmentLValue(cast<CompoundAssignOperator>(E));
        }
        case Expr::CallExprClass:
        case Expr::CXXMemberCallExprClass:
        case Expr::CXXOperatorCallExprClass:
        case Expr::UserDefinedLiteralClass:
            return EmitCallExprLValue(cast<CallExpr>(E));
    }
}
```



Specimen #2

Swiftc

- ~530k lines of C++ plus 2m lines clang and LLVM. Many same authors. Not self-hosting.
- Newer app-dev language.
- Tightly integrated with clang, interop with C/ObjC libraries.
- Extra SIL IR for optimizations.
- Multi-target, via LLVM.
- 2014-now, mostly Apple.

```
RValue RValueEmitter::visitIfExpr(IfExpr *E, SGFContext C) {
    auto &lowering = SGF.getTypeLowering(E->getType());

    if (lowering.isLoadable() || !SGF.silConv.useLoweredAddresses()) {
        // If the result is loadable, emit each branch and forward its result
        // into the destination block argument.
        Condition cond = SGF.emitCondition(E->getCondExpr(),
                                             /*invertCondition*/ false,
                                             SGF.getLoweredType(E->getType()),
                                             NumTrueTaken, NumFalseTaken);

        cond.enterTrue(SGF);
        SILValue trueValue;
        {
            auto TE = E->getThenExpr();
            FullExpr trueScope(SGF.Cleanups, CleanupLocation(TE));
            trueValue = visit(TE).forwardAsSingleValue(SGF, TE);
        }
        cond.exitTrue(SGF, trueValue);

        cond.enterFalse(SGF);
        SILValue falseValue;
        {
            auto EE = E->getElseExpr();
            FullExpr falseScope(SGF.Cleanups, CleanupLocation(EE));
            falseValue = visit(EE).forwardAsSingleValue(SGF, EE);
        }
        cond.exitFalse(SGF, falseValue);
    }
```

Specimen #3

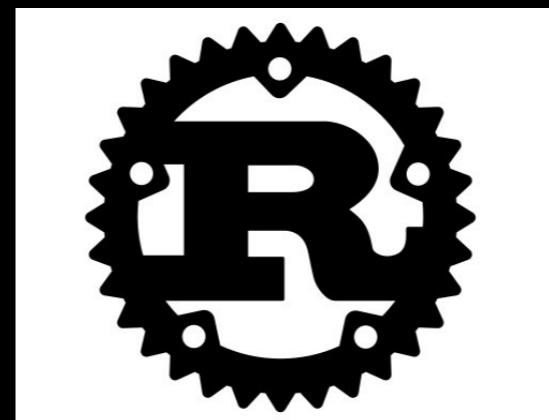
Rustc

- ~360k lines of Rust, plus 1.2m lines LLVM. Self-hosting, bootstrapped from OCaml.
- Newer systems language.
- Two extra IRs (HIR, MIR).
- Multi-target, via LLVM.
- 2010-now, large multi-org team.
- Originally mostly Mozilla. And yes I did a lot of the initial bring-up so my name is attached to it forever; glad it worked out!

```
fn expr_as_rvalue(
    &mut self,
    mut block: BasicBlock,
    scope: Option<region::Scope>,
    expr: Expr<'tcx>,
) -> BlockAnd<Rvalue<'tcx>> {
    debug!(
        "expr_as_rvalue(block={:?}", scope={:?}", expr={:?})",
        block, scope, expr
    );

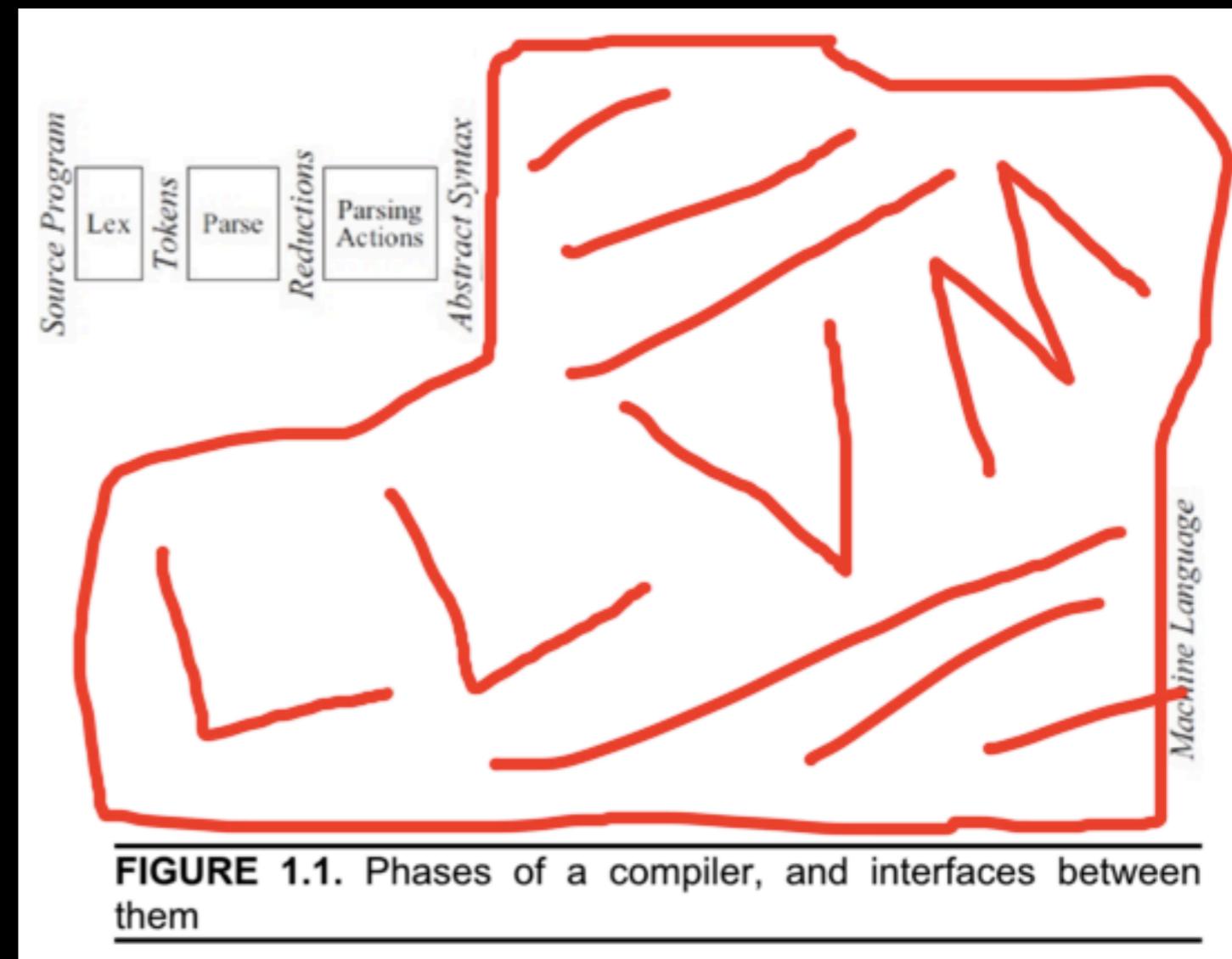
    let this = self;
    let expr_span = expr.span;
    let source_info = this.source_info(expr_span);

    match expr.kind {
        ExprKind::Scope {
            region_scope,
            lint_level,
            value,
        } => {
            let region_scope = (region_scope, source_info);
            this.in_scope(region_scope, lint_level, block, |this| {
                this.as_rvalue(block, scope, value)
            })
        }
    }
}
```



Aside: what is this "LLVM"?

- Notice the last 3 languages all end in LLVM. "Low Level Virtual Machine"
<https://github.com/llvm/llvm-project>
- Strongly typed IR, serialization format, library of optimizations, lowerings to many target architectures.
- "One-stop-shop" for compiler backends.
- 2003-now, UIUC at first, many industrial contributors now.
- Longstanding dream of compiler engineering world, possibly most successful attempt at it yet.
- Here is a funny diagram of modern compilers from Andi McClure (<https://runhello.com/>)



Specimen #4

GNU Compiler Collection (GCC)

- ~2.2m lines of mostly C, C++. 600k lines Ada. Self-hosting, bootstrapped from other C compilers.
- Multi-language (C, C++, ObjC, Ada, D, Go, Fortran), multi-target (21).
- Language & target-agnostic TREE AST and RTL IR. Challenging to work on.
- 1987-present, large multi-org team.
- Generates quite fast code.
- Originally political project to free software from proprietary vendors. Licensed somewhat protectively.

```
static int
find_reusable_reload (rtx *p_in, rtx out, enum reg_class rclass,
                      enum reload_type type, int opnum, int dont_share)
{
  rtx in = *p_in;
  int i;

  if (earlyclobber_operand_p (out))
    return n_reloads;

  for (i = 0; i < n_reloads; i++)
    if ((reg_class_subset_p (rclass, rld[i].rclass)
         || reg_class_subset_p (rld[i].rclass, rclass))
        /* If the existing reload has a register, it must fit our class. */
        && (rld[i].reg_rtx == 0
             || TEST_HARD_REG_BIT (reg_class_contents[(int) rclass],
                                   true_regnum (rld[i].reg_rtx)))
        && ((in != 0 && MATCHES (rld[i].in, in) && ! dont_share
              && (out == 0 || rld[i].out == 0 || MATCHES (rld[i].out, out))
              || (out != 0 && MATCHES (rld[i].out, out)
                  && (in == 0 || rld[i].in == 0 || MATCHES (rld[i].in, in))))
        && (rld[i].out == 0 || ! earlyclobber_operand_p (rld[i].out))
        && (small_register_class_p (rclass)
             || targetm.small_register_classes_for_mode_p (VOIDmode))
        && MERGABLE_RELOADS (type, rld[i].when_needed, opnum, rld[i].opnum))
    return i;
```



Part 2: why so big?

Size and economics

- Compilers get big because the development costs are seen as justified by the benefits, at least to the people paying the bills.
 - Developer productivity: highly expressive languages, extensive diagnostics, IDE integration, legacy interop.
 - Every drop of runtime performance: shipping on billions of devices or gigantic multi-warehouse fleets.
 - Covering & exploiting all the hardware: someone makes a new chip, they pay for an industrial compiler to make use of it.
 - Writing compilers in verbose languages: for all the usual reasons (compatibility, performance, familiarity).

Tradeoffs and balance

- This is ok!
- The costs and benefits are context dependent.
- Different contexts, weightings: different compilers.
- Remainder of talk will be exploring those differences.
- Always remember: balancing cost tradeoffs by context.
- Totally biased subset of systems: stuff I think is interesting and worth knowing, might give hope / inspire curiosity.

Part 3: variations (this part is much longer)

Variation #1

Fewer optimizations

- In some contexts, "all the optimizations" is too much.
- Too slow to compile, too much memory, too much development / maintenance effort, too inflexible.
- Common in JITs, or languages with lots of indirection anyways (dynamic dispatch, pointer chasing): optimizer can't do too well anyways.

Proebsting's law

- "Compiler Advances Double Computing Power Every 18 Years"
- Sarcastic joke / real comparison to Moore's law: hardware doubles power every 18 months. Swamps compilers.
- Empirical observation though! Optimizations seem to only win ~3-5x, after 60+ years of work.
- Less-true as language gains more abstractions to eliminate (i.e. specialize / de-virtualize). More true if lower-level.

4. Discussion

The results of our experiment suggest that Proebsting's Law is probably true. The reality is somewhat grimmer than Proebsting initially supposed. Research in optimizing compilers has been ongoing since 1955. The compiler technology developed over this 45-year period is able to improve the performance of integer intensive programs by a factor of 3.3. This corresponds to uniform performance improvements of about 2.8% per year. Even if we assume that the beginning of useful compiler optimization research began in the mid 1960's [5], the uniform performance improvement on integer intensive codes due to compiler optimization is still only 3.6% per year. This lies in stark contrast to the 60% per year performance improvements we can expect from hardware due to Moore's Law.

The performance difference between optimized and unoptimized programs is larger for the floating-point intensive codes in SPECfp95. This indicates that compiler research has had a larger effect on improving the performance of scientific codes than on improving the performance of ordinary, integer intensive applications. Again, if we assume compiler research has been ongoing since 1955, we get a doubling of performance every 16 years. This corresponds to uniform performance improvements of about 4.9% per year over this 45 year period. This is only slightly better than the results for integer intensive programs.

Scott, Kevin. On Proebsting's Law. 2001

Frances Allen Got All The Good Ones

- 1971: "A Catalogue of Optimizing Transformations".
- The ~8 passes to write if you're going to bother.
- Inline, Unroll (& Vectorize), CSE, DCE, Code Motion, Constant Fold, Peephole.
- That's it. You're welcome.
- Many compilers just do those, get ~80% best-case perf.



https://commons.wikimedia.org/wiki/File:Allen_mag_2528-3750K-b.jpg - CC BY-SA 2.0

Specimen #5

V8

- 660k lines C++ including backends. Not self-hosting.
- JavaScript compiler in Chrome, Node.
- Multi-target (7), multi-tier JIT.
Optimizations mix of classical stuff and dynamic language stuff from Smalltalk.
- Multiple generations of optimization and IRs. Always adjusting for sweet spot of runtime perf vs. compile time, memory, maintenance cost, etc.
- Recently added slower (non-JIT) interpreter tier, removed others.
- 2008-present, mostly Google, open source.

```
// Shared routine for word comparison against zero.
void InstructionSelector::VisitWordCompareZero(Node* user, Node* value,
                                               FlagsContinuation* cont) {
    // Try to combine with comparisons against 0 by simply inverting the branch.
    while (value->opcode() == IrOpcode::kWord32Equal && CanCover(user, value)) {
        Int32BinopMatcher m(value);
        if (!m.right().Is(0)) break;

        user = value;
        value = m.left().node();
        cont->Negate();
    }

    if (CanCover(user, value)) {
        switch (value->opcode()) {
            case IrOpcode::kWord32Equal:
                cont->OverwriteAndNegateIfEqual(kEqual);
                return VisitWordCompare(this, value, kX64Cmp32, cont);
            case IrOpcode::kInt32LessThan:
                cont->OverwriteAndNegateIfEqual(kSignedLessThan);
                return VisitWordCompare(this, value, kX64Cmp32, cont);
            case IrOpcode::kInt32LessThanOrEqual:
                cont->OverwriteAndNegateIfEqual(kSignedLessThanOrEqual);
                return VisitWordCompare(this, value, kX64Cmp32, cont);
            case IrOpcode::kUInt32LessThan:
                cont->OverwriteAndNegateIfEqual(kUnsignedLessThan);
                return VisitWordCompare(this, value, kX64Cmp32, cont);
            case IrOpcode::kUInt32LessThanOrEqual:
                cont->OverwriteAndNegateIfEqual(kUnsignedLessThanOrEqual);
                return VisitWordCompare(this, value, kX64Cmp32, cont);
        }
    }
}
```

Compiler-friendly implementation (and input) languages

- Note: your textbook has 3 implementation flavours. Java, C, ML. No coincidence.
- ML designed as implementation language for symbolic logic (expression-tree wrangling) system: LCF (1972).
- LCF written in Lisp. Lisp also designed as implementation language for symbolic logic system: Advice Taker (1959).
- Various family members: Haskell, OCaml, Scheme, Racket.
- All really good at defining and manipulating trees. ASTs, types, IRs, etc. Usually make for much smaller/simpler compilers.

Specimen #6

Glasgow Haskell Compiler (GHC)

- 180k lines Haskell, self-hosting, bootstrapped from Chalmers Lazy ML compiler.
- Pure-functional language, very advanced type-system.
- Several tidy IRs after AST: Core, STG, CMM. Custom backends.
- 1991-now, initially academic researchers, lately Microsoft after they were hired there.

```
stmtToInstrs :: CmmNode e x -> NatM InstrBlock
stmtToInstrs stmt = do
  dflags <- getDynFlags
  is32Bit <- is32BitPlatform
  case stmt of
    CmmComment s  -> return (unitOL (COMMENT s))
    CmmTick {}     -> return nilOL

  CmmUnwind regs -> do
    let to_unwind_entry :: (GlobalReg, Maybe CmmExpr) -> UnwindTable
        to_unwind_entry (reg, expr) = M.singleton reg (fmap toUnwindExpr expr)
    case foldMap to_unwind_entry regs of
      tbl | M.null tbl -> return nilOL
      | otherwise -> do
        lbl <- mkAsmTempLabel <$> getUniqueM
        return $ unitOL $ UNWIND lbl tbl

  CmmAssign reg src
  | isFloatType ty      -> assignReg_FltCode format reg src
  | is32Bit && isWord64 ty -> assignReg_I64Code reg src
  | otherwise            -> assignReg_IntCode format reg src
    where ty = cmmRegType dflags reg
          format = cmmTypeFormat ty

  CmmStore addr src
  | isFloatType ty      -> assignMem_FltCode format addr src
  | is32Bit && isWord64 ty -> assignMem_I64Code addr src
  | otherwise            -> assignMem_IntCode format addr src
    where ty = cmmExprType dflags src
          format = cmmTypeFormat ty
```

Specimen #7

Chez Scheme

- 87k lines Scheme (a Lisp), self-hosting, bootstrapped from C-Scheme.
- 4 targets, good performance, incremental compilation.
- Written on "nanopass framework" for compilers with many similar IRs. Chez has 27 different IRs!
- 1984-now, academic-industrial, mostly single developer. Getting down to the size-range where a compiler is small enough to be that.

```
(define asm-size
  (lambda (x)
    (case (car x)
      [(asm) 0]
      [(byte) 1]
      [(word) 2]
      [else 4])))

(define asm-move
  (lambda (code* dest src)
    (Trivit (dest src)
      (record-case src
        [(imm) (n)
          (if (and (eqv? n 0) (record-case dest [(reg) r #t] [else #f]))
              (emit xor dest dest code*)
              (emit movi src dest code*))]
        [(literal) stuff (emit movi src dest code*)]
        [else (emit mov src dest code*)]))))

(define-who asm-move/extend
  (lambda (op)
    (lambda (code* dest src)
      (Trivit (dest src)
        (case op
          [(sext8) (emit movsb src dest code*)]
          [(sext16) (emit movsw src dest code*)]
          [(zext8) (emit movzb src dest code*)]
          [(zext16) (emit movzw src dest code*)]
          [else (sorry! who "unexpected op ~s" op)))))))
```

Specimen #8

Poly/ML

- 44k lines SML, self-hosting.
- Single machine target (plus byte-code), AST + IR, classical optimizations. Textbook style.
- Standard platform for symbolic logic packages
Isabelle and HOL.
- 1986-now, academic, mostly single developer.

```
| cgOp(PushToStack(RegisterArg reg)) =
|   let
|     val (rc, rx) = getReg reg
|   in
|     opCodeBytes(PUSH_R rc, if rx then SOME{w=false, b = true,
|                                               x=false, r = false }
|                           else NONE)
|   end
|
| cgOp(PushToStack(MemoryArg{base, offset, index})) =
|   opAddressPlus2(Group5, LargeInt.toInt offset, base, index, 0w6)
|
| cgOp(PushToStack(NonAddressConstArg constnt)) =
|   if is8BitL constnt
|   then opCodeBytes(PUSH_8, NONE) @ [Word8.fromLargeInt constnt]
|   else if is32bit constnt
|   then opCodeBytes(PUSH_32, NONE) @ int32Signed constnt
|   else
|     let
|       val opb = opCodeBytes(Group5, NONE)
|       val mdrm = modrm(Based0, 0w6 (* push *), 0w5 (* PC rel *))
|     in
|       opb @ [mdrm] @ int32Signed(tag 0)
|     end
|
| cgOp(PushToStack(AddressConstArg _)) =
|   (
|     case targetArch of
|       Native64Bit => (* Put it in the constant area. *)
|         let
|           val opb = opCodeBytes(Group5, NONE)
|           val mdrm = modrm(Based0, 0w6 (* push *), 0w5 (* PC rel *));
|         in
|           opb @ [mdrm] @ int32Signed(tag 0)
|         end
|     | Native32Bit => opCodeBytes(PUSH_32, NONE) @ int32Signed(tag 0)
|     | ObjectId32Bit =>
```

Specimen #9

CakeML

- 58k lines SML, 5 targets, self-hosting.
- 9 IRs, many simplifying passes.
- 160k lines HOL proofs: verified!
- ***Language semantics proven to be preserved through compilation!!!***
- Cannot emphasize enough. This was science fiction when I was young.
- CompCert first serious one, now several.
- 2012-now, deeply academic.

```
val WordOp64_on_32_def = Define `  
WordOp64_on_32 (opw:opw) =  
  dtcase opw of  
  | Andw => list_Seq [Assign 29 (Const 0w);  
                      Assign 27 (Const 0w);  
                      Assign 33 (Op And [Var 13; Var 23]);  
                      Assign 31 (Op And [Var 11; Var 21])]  
  | Orw =>  list_Seq [Assign 29 (Const 0w);  
                      Assign 27 (Const 0w);  
                      Assign 33 (Op Or [Var 13; Var 23]);  
                      Assign 31 (Op Or [Var 11; Var 21])]  
  | Xor =>  list_Seq [Assign 29 (Const 0w);  
                      Assign 27 (Const 0w);  
                      Assign 33 (Op Xor [Var 13; Var 23]);  
                      Assign 31 (Op Xor [Var 11; Var 21])]  
  | Add =>  list_Seq [Assign 29 (Const 0w);  
                      Assign 27 (Const 0w);  
                      Inst (Arith (AddCarry 33 13 23 29));  
                      Inst (Arith (AddCarry 31 11 21 29))]  
  | Sub =>  list_Seq [Assign 29 (Const 1w);  
                      Assign 27 (Op Xor [Const (-1w); Var 23]);  
                      Inst (Arith (AddCarry 33 13 27 29));  
                      Assign 27 (Op Xor [Const (-1w); Var 21]);  
                      Inst (Arith (AddCarry 31 11 27 29))]`  
  
val WordShift64_on_32_def = Define `  
WordShift64_on_32 sh n = list_Seq  
  (* inputs in 11 and 13, writes results in 31 and 33 *)  
  (if sh = Ror then  
    (let n = n MOD 64 in  
      (if n < 32 then  
          [Assign 33 (Op Or [ShiftVar Lsl 11 (32 - n);  
                           ShiftVar Lsr 13 n]);  
           Assign 31 (Op Or [ShiftVar Lsl 13 (32 - n);  
                           ShiftVar Lsr 11 n])]  
        else  
          [Assign 33 (Op Or [ShiftVar Lsl 13 (64 - n);  
                           ShiftVar Lsr 11 (n - 32))];  
           Assign 31 (Op Or [ShiftVar Lsl 11 (64 - n);  
                           ShiftVar Lsr 13 n)])])`
```

Variation #3

Meta-languages

- Notice Lisp / ML code looks a bit like grammar productions: recursive branching tree-shaped type definitions, pattern matching.
- There's a language lineage that took that idea ("programs as grammars") to its logical conclusion: metacompilers (a.k.a. "compiler-compilers"). Ultimate in "compiler-friendly" implementation languages.
- More or less: parser glued to an "un-parser".
- Many times half a metacompiler lurks in more-normal compilers:
 - YACCs ("yet another compiler-compiler"): parser-generators
 - BURGs ("bottom-up rewrite generators"): code-emitter-generators
- See also: GCC ".md" files, LLVM TableGen. Common pattern!

Aside: SRI-ARC

- Stanford Research Institute - Augmentation Research Lab. US Air Force R&D project. Very famous for its NLS ("oNLine System").
- History of that project too big to tell here. Highly influential in forms of computer-human interaction, hypertext, collaboration, visualization.
- Less well-known is their language tech: TREE-META and MPS/MPL.

SRI-ARC 18-AUG-72 10:02 10575
SRI-ARC 8 JUNE 1972 10575
Summary
Highlights of 1971

At present the primary language systems developed and
in use at ARC are the Tree-Meta Compiler-compiler
System and the L10 Programming language system which
was written in Tree-Meta. 3cle5a

Work is currently progressing on a Modular Programming
System (MPS) in collaboration with a group at the
Xerox Palo Alto Research Center. 3cle5b

Specimen #10

TREE-META

- 184 lines of TREE-META. Bootstrapped from META-II.
- In the Schorre metacompiler family (META, META-II)
- SRI-ARC, 1967. Made to support language tools in the NLS project.
- "Syntax-directed translation": parse input to trees, un-parse to machine code. Only guided by grammars.
- Hard to provide diagnostics, type-checking, optimization, really anything other than straight translations.
- But: extremely small, simple compilers. Couple pages. Ideal for bootstrap phase.

```
.META PROGM

OUTPT[-,-] => % *1 ':' % '%PUSHJ;' % *2 '%POPJ;' % ;
AC[-,-,-] => *1 *3 ACX [*2,#1] #1 ':' % ;
ACX[AC[-,-,-],#1] => #1 ')'; % *1:*1 *1:*3 ACX[*1:*2,#1]
[-,#1] => #1 ')'; % *1 ;

T/ => '%BT;DATA(@' ;
F/ => '%BF;DATA(@' ;

BALTER[-] => '%SAV;' % *1 '%RSTR;' % ;
OER[-,-] => *1 '%OER;' % *2 ;
OUTAB[-] => <TN <- CONV[*1]; OUT[TN] > ')'; % ;
ERCODE [-,NUM] => *1 '%ERCHK;DATA(' OUTAB[*2]
[---] => *1 '%ERSTR;DATA(' OPSTR[*2] ;
ERR[-] => *1 '%ERCHK;DATA(0);' % ;

D00[-,-] => *1 *2 ;

NDMK[-,-] => '%MKND;DATA(@' *1 ',' *2 ');' % ;
NDLB => '%NDLBL;DATA(@' *1 ');' % ;
MKNODE [-] => '%NDMK;DATA(' *1 ');' % ;

G00/ => '%OUTRE;' % ;
SET/ => '%SET;' % ;

PRIM[.ID] => '%' *1 ';' %
[.SR] => '%SRP;DATA(' OPSTR[*1] ;

CALL[-] => '%CALL;DATA(@' *1 ');' % ;
STST[-] => '%TST;DATA(' OPSTR[*1] ;
SCODE[-] => '%CHRCK;DATA(' OUTAB[*1] ;
ARB[-] => #1 ':' % *1 '%BT;DATA(@' #1 ');' % '%SET;' % ;
BEGINN[---] => <B <- 0 > *2 'ENTRY 0;%INIT;%CALL;DATA(@' *1 ');' %
'%FIN;' % ;
```

In the past contract period Tree Meta was useful in bootstrapping from the old XDS-940 to the new PDP-10. Currently it is being used to create the first MPL compiler.

Specimen #11 (Segue)

Mesa

- 42k lines of Mesa (bootstrapped from MPL, itself from TREE-META).
- One of my favourite languages!
- Strongly typed, modules with separate compilation and type checked linking. Highly influential (Modula, Java).
- Co-designed language, OS, and byte-code VM implemented in CPU microcode, adapted to compiler.
- Xerox PARC, 1976-1981, small team left SRI-ARC, took MPL.

```
sCcall: PROCEDURE [node: TreeIndex] RETURNS [nrets: CARDINAL] =
BEGIN -- generates code for procedure call statement
OPEN FOpCodes;
ptsei: CSEIndex ← operandtype[(tb+node).son1];
portcall: BOOLEAN ← SymTabDefs.XferMode[ptsei] = port;
computedtarget: BOOLEAN;
inlineTree: TreeLink;
nparms: CARDINAL;
savestacksize: INTEGER;
sei: ISEIndex;
bti: CBTIndex;
a: BitAddress;
inlineCall: BOOLEAN;

WITH (tb+node).son1 SELECT FROM
symbol =>
BEGIN
sei ← index;
inlineCall ← (seb+sei).constant AND (seb+sei).extended;
computedtarget ← (ctxb+(seb+sei).ctxnum).ctxlevel # 1G;
END;
ENDCASE =>
BEGIN
inlineCall ← FALSE;
computedtarget ← TRUE;
END;
IF ~inlineCall THEN dumpstack[];
```

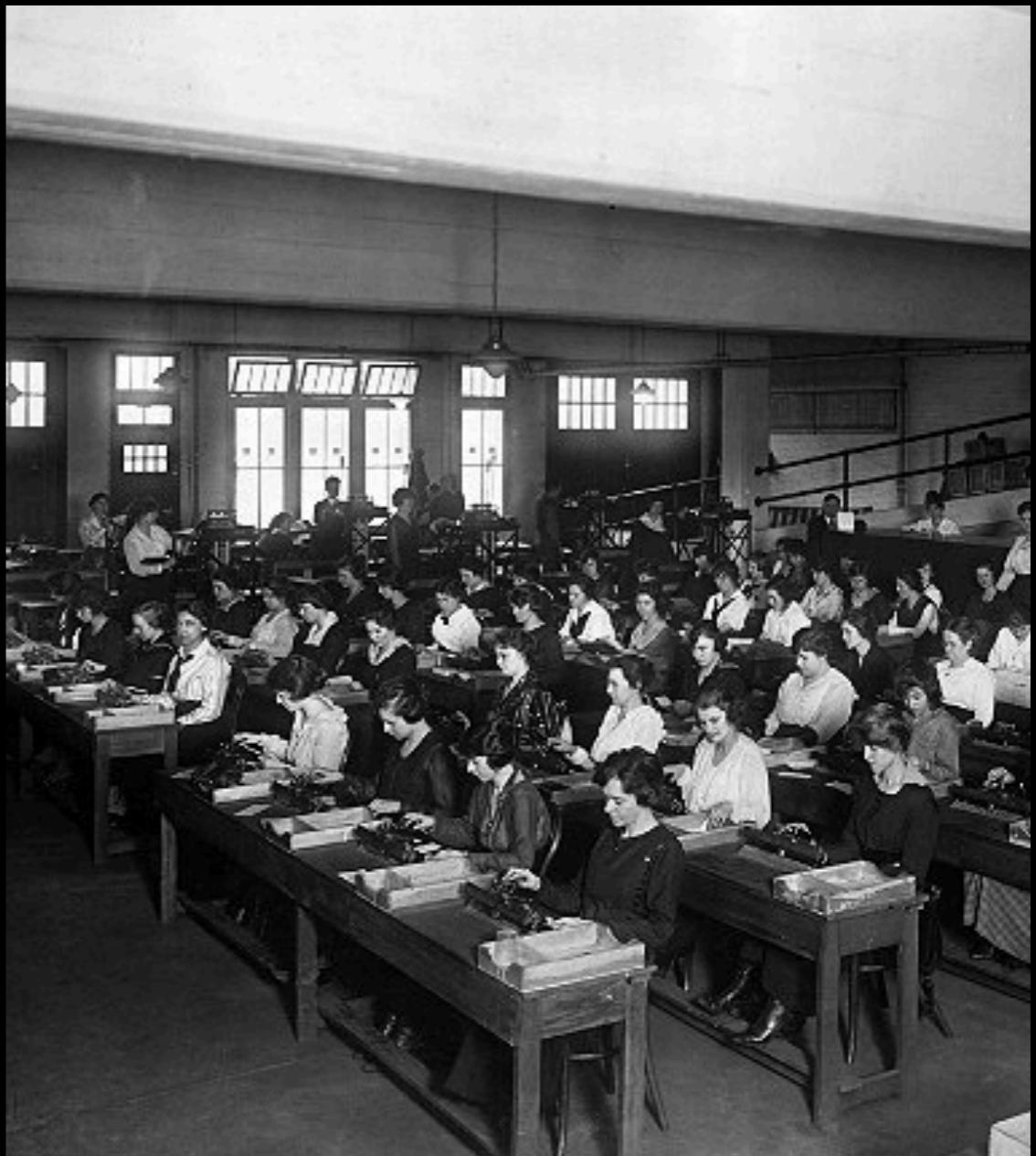


Variations #4, #5, and #6 leverage interpreters

- Mesa and Xerox PARC is a nice segue into next few points: all involve compilers interacting with interpreters.
- Interpreters & compilers actually have a long relationship!
- In fact interpreters predate compilers.
- Let us travel back in time to the beginning, to illustrate!

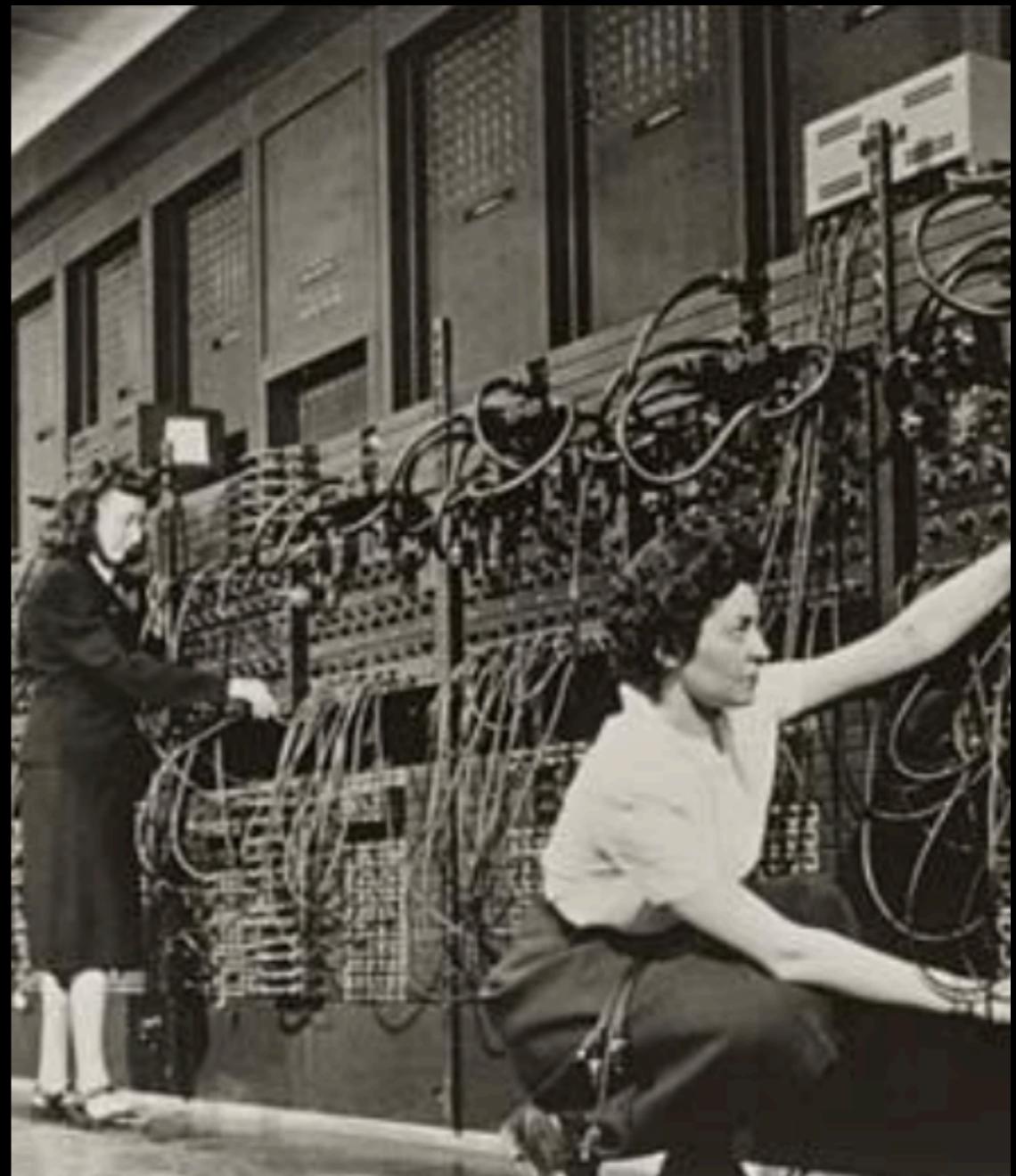
Origins of "computer"

- 1940s: First digital computers.
- Before: fixed-function machines and/or humans (largely women) doing job called "computer".
- Computing power literally measured in "kilo-girls" and "kilo-girl-hours".



ENIAC: general hardware

- 1945: ENIAC built for US Army, Ordnance Corps. Artillery calculations in WWII.
- "Programmers" drawn from "computer" staff, all women.
- "Programming" meant physically rewiring per-task.



Stored Programs

- 1948: Jean Bartik leads team to convert ENIAC to "stored programs", instructions (called "orders") held in memory.
- Interpreted by hardware. Faster to reconfigure than rewiring; but ran slower.
- Subroutine concept developed for factoring stored programs.



First software pseudo codes: interpreters on ENIAC, BINAC, UNIVAC

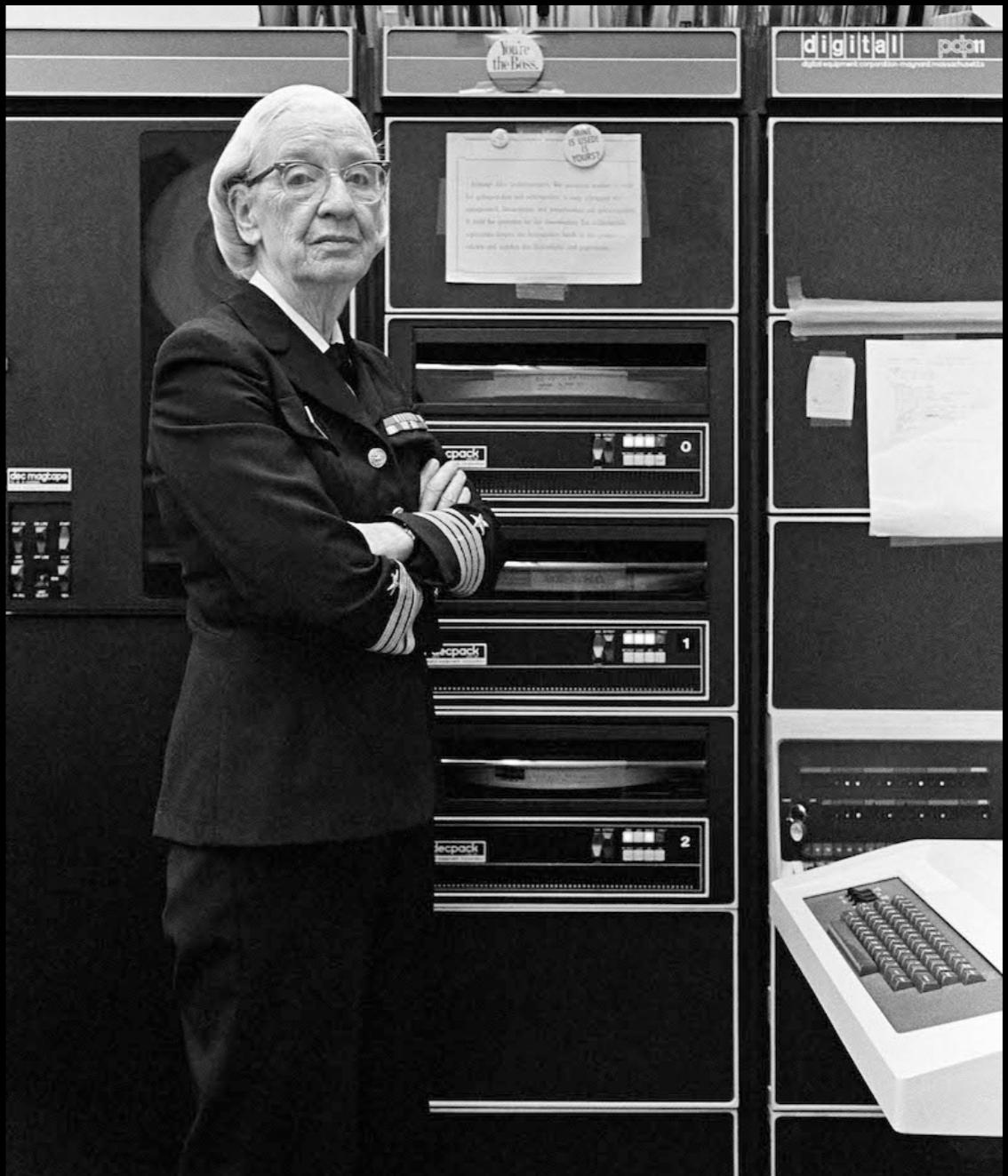
- 1949: "Short Code" software interpreters for higher level "pseudo-code" instructions (non-HW-interpreted) that denote subroutine calls and expressions. ~50x slower than HW-interpreted.

```
X3 = ( X1 + Y1 ) / X1 * Y1    substitute variables
X3 03 09 X1 07 Y1 02 04 X1    substitute operators and parentheses.
                                Note multiplication is represented
                                by juxtaposition.
07Y10204X1Y1
0000X30309X1
                                group into 12-byte words.
```

Specimen #12

A-0: the first compiler

- Reads interpreter-like pseudo-codes, then emits "compilation" program with all codes resolved to their subroutines.
- Result runs almost as fast as manually coded; but as easy to write-for as interpreter. An interpreter "fast mode".
- Rationale all about balancing time tradeoffs (coding-time, compiler-execution-time, run-time).
- 1951, Grace Hopper, Univac



Balance between
interpretation and compilation
is context dependent too!

Variation #4

Only compile from frontend to IR, interpret residual VM code

- Can stop before real machine code. Emit IR == "virtual machine" code.
- Can further compile or just interpret that VM code.
- Residual VM interpreter has several real advantages:
 - Easier to port to new hardware, or bootstrap compiler. "Just get something running".
 - Fast compilation & program startup, keeps interactive user engaged.
 - Simply easier to write, less labor. Focus your time on frontend semantics.

As a cheap implementation device: bytecode interpreters offer 1/4 of the performance of optimizing native-code compilers, at 1/20 of the implementation cost.

<https://xavierleroy.org/talks/zam-kazam05.pdf>

Specimen #13

Roslyn

- 350k lines C#, 320k lines VB.
Self-hosting, bootstrapped off previous gen.
- Multi-language framework (C#, VB.NET). Rich semantics, good diagnostics, IDE integration.
- Lowers from AST to CIL IR.
Separate CLR project interprets or compiles IR.
- 2011-now, Microsoft, OSS.

```
private void EmitBinaryOperatorInstruction(BoundBinaryOperator expression)
{
    switch (expression.OperatorKind.Operator())
    {
        case BinaryOperatorKind.Multiplication:
            _builder.EmitOpCode(ILOpCode.Mul);
            break;

        case BinaryOperatorKind.Addition:
            _builder.EmitOpCode(ILOpCode.Add);
            break;

        case BinaryOperatorKind.Subtraction:
            _builder.EmitOpCode(ILOpCode.Sub);
            break;

        case BinaryOperatorKind.Division:
            if (IsUnsignedBinaryOperator(expression))
            {
                _builder.EmitOpCode(ILOpCode.Div_un);
            }
            else
            {
                _builder.EmitOpCode(ILOpCode.Div);
            }
            break;
    }
}
```

Specimen #14

Eclipse Compiler for Java (ECJ)

- 146k lines Java, self-hosting, bootstrapped off Javac.
- In Eclipse! Also in many Java products (eg. IntelliJ IDEA). Rich semantics, good diagnostics, IDE integration.
- Lowers from AST to JVM IR. Separate JVM projects interpret or compile IR.
- 2001-now, IBM, OSS.

```
/**  
 * Code generation for the conditional operator ?:  
 *  
 * @param currentScope org.eclipse.jdt.internal.compiler.lookup.BlockScope  
 * @param codeStream org.eclipse.jdt.internal.compiler.codegen.CodeStream  
 * @param valueRequired boolean  
 */  
@Override  
public void generateCode(  
    BlockScope currentScope,  
    CodeStream codeStream,  
    boolean valueRequired) {  
  
    int pc = codeStream.position;  
    BranchLabel endifLabel, falseLabel;  
    if (this.constant != Constant.NotAConstant) {  
        if (valueRequired)  
            codeStream.generateConstant(this.constant, this.implicitConversion);  
        codeStream.recordPositionsFrom(pc, this.sourceStart);  
        return;  
    }  
    Constant cst = this.condition.optimizedBooleanConstant();  
    boolean needTruePart =  
        !(cst != Constant.NotAConstant && cst.booleanValue() == false);  
    boolean needFalsePart =  
        !(cst != Constant.NotAConstant && cst.booleanValue() == true);  
    endifLabel = new BranchLabel(codeStream);
```

Variation #5

Only compile some functions, interpret the rest

- Cost of interpreter only bad at inner loops or fine-grain. Outer loops or coarse-grain (eg. function calls) similar to virtual dispatch.
- Design option: interpret by default, selectively compile hot functions ("fast mode") at coarse grain. Best of both worlds!
 - Keep interpreter-speed immediate feedback to user.
 - Interpreter may be low-effort, portable, can bootstrap.
 - Defer effort on compiler until needed.
 - Anything hard to compile, just call back to interpreter.

Specimen #15

Pharo/Cog

- 54k line VM interpreter and 18k line JIT: C code generated from Smalltalk metaprograms. Bootstrapped from Squeak.
- Smalltalk is what you'll actually hear people mention coming from Xerox PARC.
- Very simple language. "Syntax fits on a postcard". Easy to interpret.
- Complete GUI, IDE, powerful tools.
- Standard Smalltalk style: interpret by default, JIT for "fast mode". Compiler bootstraps-from and calls-into VM whenever convenient.
- Targets ARM, x86, x64, MIPS.
- 2008-now, academic-industrial consortium.

The screenshot shows the Pharo System Browser interface with the title "System Browser: SimpleStackBasedCogit". The left pane displays a class hierarchy with nodes like Printf, Balloon-Engine-P, BytecodeSets-N, VMMaker-Buildin, VMMaker-Interpr, VMMaker-JIT, VMMaker-JITSim, VMMaker-Multith, VMMaker-Plugins, and SimpleStackToR. The "SimpleStackToR" node is currently selected. The right pane shows the implementation of the `genShortJumpIfFalse` method. The code is as follows:

```
distance := self v3: (self generatorAt: byte0)
ShortForward: bytecodePC
Branch: 0
Distance: methodObj.
target := distance + 1 + bytecodePC.
^self genJumpIf: objectMemory falseObject to: target
```

At the bottom of the browser window, there is a footer bar with the text "eem 11/2/2012 10:57 · bytecode generators · 1 implementor · only in char".

Specimen #16

Franz Lisp

- 20k line C interpreter, 7,752 line Lisp compiler.
- Older command-line system, standard Unix Lisp for years.
- Like Smalltalk: very simple language. Actually an AST/IR that escaped from the lab. Easy to interpret.
- Frequent Lisp style: interpret by default; compile for "fast mode". Compiler bootstraps-from and calls-into interpreter whenever convenient.
- Targets m68k and VAX.
- 1978-1988, UC Berkeley.

```
;---- e-move :: move value from one place to another
; this corresponds to d-move except the args are EIADRS
;
(defun e-move (from to)
  (if (and (dtpr from)
            (eq '$ (car from))
            (eq 0 (cadr from)))
      then (e-write2 'clrl to)
      else (e-write3 'movl from to)))

;---- d-move :: emit instructions to move value from one place to another
;
(defun d-move (from to)
  (makecomment `((from ,(e-uncvt from) to ,(e-uncvt to)))
  #+(or for-vax for-tahoe)
  (cond ((eq 'Nil from) (e-move '$ 0) (e-cvt to)))
        (t (e-move (e-cvt from) (e-cvt to)))))

#+for-68k
(let ((froma (e-cvt from))
      (toa (e-cvt to)))
  (if (and (dtpr froma)
            (eq '$ (car froma))
            (and (>& (cadr froma) -1) (<& (cadr froma) 65))
            (atom toa)
            (eq 'd (nthchar toa 1)))
      then ;it's a mov #immed,Dn, where 0 <= immed <= 64
           ; i.e., it's a quick move
      (e-write3 'moveq froma toa)
      else (cond ((eq 'Nil froma) (e-write3 'movl '#.nil-reg toa))
                 (t (e-write3 'movl froma toa))))))
```



Variation #6

Partial Evaluation Tricks

- Consider program in terms of parts that are static (will not change anymore) or dynamic (may change).
- Partial evaluator (a.k.a. "specializer") runs the parts that depend only on static info, emits residual program that only depends on dynamic info.
- Note: interpreter takes two inputs: program to interpret, and program's own input. First is static, but redundantly treated as dynamic.
- So: compiling is like partially evaluating an interpreter, eliminating the redundant dynamic treatment in its first input.

Futamura Projections

- Famous work relating programs P, interpreters I, partial evaluators E, and compilers C. The so-called "Futamura Projections":
 - 1: $E(I, P) \rightarrow$ partially evaluate $I(P) \rightarrow$ emit $C(P)$, a compiled program
 - 2: $E(E, I) \rightarrow$ partially evaluate $\lambda P. I(P) \rightarrow$ emit C, a compiler!
 - 3: $E(E, E) \rightarrow$ partially evaluate $\lambda I. \lambda P. I(P) \rightarrow$ emit a compiler-compiler!
- Futamura, Yoshihiko, 1971. *Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler*. <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.10.2747>
- Formal strategy for building compilers from interpreters and specializers.

Specimen #17

Truffle/Graal

- 240k lines of Java for Graal (VM); 90k lines for Truffle (interpreter-writing framework)
- Actual real system based on first Futamura Projection.
- Seriously competitive! Potential future Oracle JVM.
- Multi-language (JavaScript, Python, Ruby, R, JVM byte code, LLVM bitcode) multi-target (3)
- "Write an interpreter with some machinery to help the partial evaluator, get a compiler for free"
- Originally academic, now Oracle

```
public Variable emitConditional(LogicNode node, Value trueValue, Value falseValue) {
    if (node instanceof IsNullNode) {
        IsNullNode isNullNode = (IsNullNode) node;
        LIRKind kind =
gen.getLIRKind(isNullNode.getValue().stamp(NodeView.DEFAULT));
        Value nullValue = gen.emitConstant(kind, isNullNode.nullConstant());
        return gen.emitConditionalMove(kind.getPlatformKind(),
            operand(isNullNode.getValue()),
            nullValue, Condition.EQ, false,
            trueValue, falseValue);
    } else if (node instanceof CompareNode) {
        CompareNode compare = (CompareNode) node;
        PlatformKind kind =
gen.getLIRKind(compare.getX().stamp(NodeView.DEFAULT))
            .getPlatformKind();
        return gen.emitConditionalMove(kind, operand(compare.getX()),
            operand(compare.getY()),
            compare.condition().asCondition(),
            compare.unorderedIsTrue(),
            trueValue, falseValue);
    } else if (node instanceof LogicConstantNode) {
        return gen.emitMove(((LogicConstantNode) node).getValue() ?
            trueValue : falseValue);
    } else if (node instanceof IntegerTestNode) {
        IntegerTestNode test = (IntegerTestNode) node;
        return gen.emitIntegerTestMove(operand(test.getX()),
            operand(test.getY()),
            trueValue, falseValue);
    } else {
        throw GraalError.unimplemented(node.toString());
    }
}
```

Variation #7

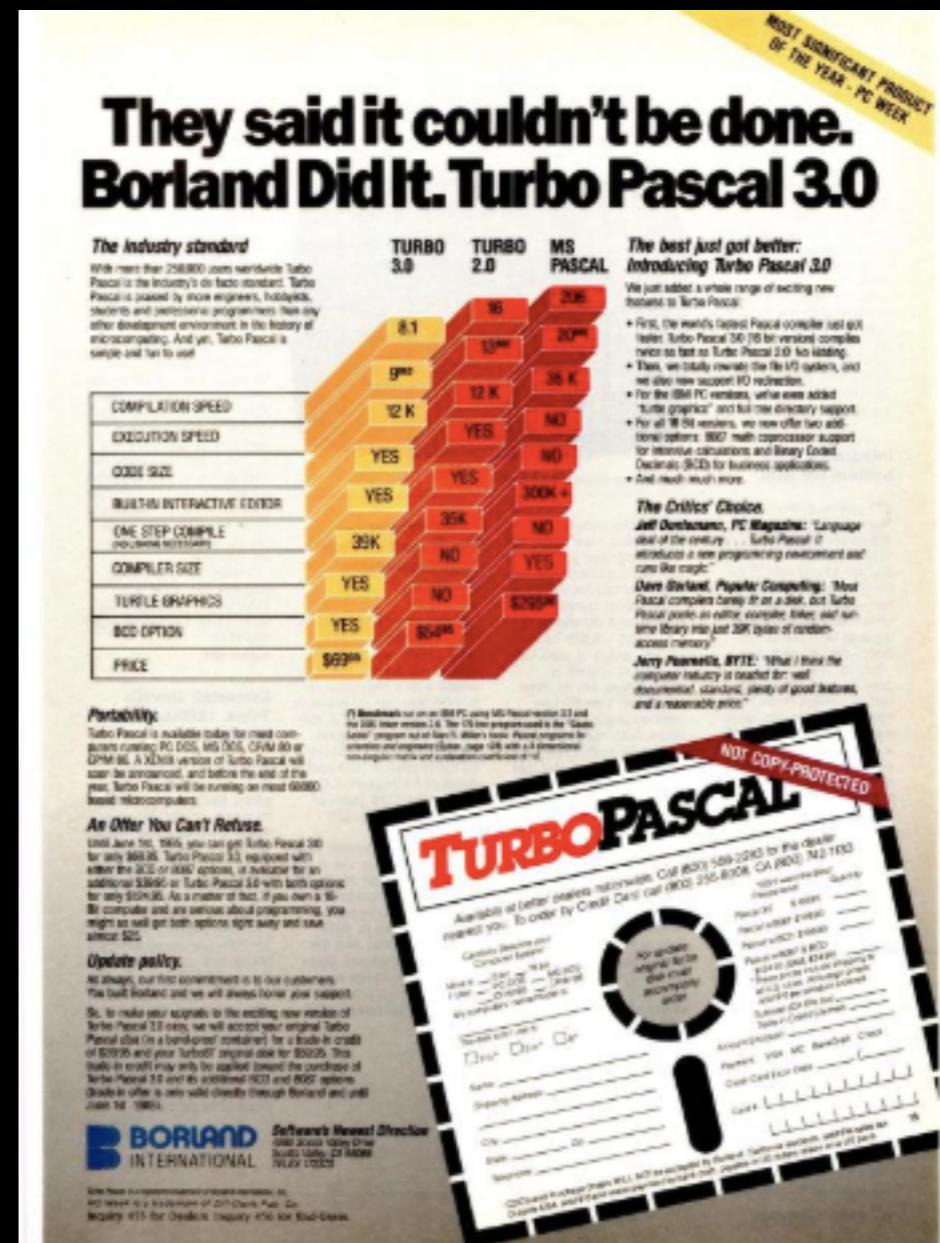
Forget IR and/or AST!

- In some contexts, even building an AST or IR is overkill.
- Small hardware, tight budget, one target, bootstrapping.
- Avoiding AST tricky, languages can be designed to help. So-called "single-pass" compilation, emit code line-at-a-time, while reading.
- Likely means no optimization aside from peephole.

Specimen #18

Turbo Pascal

- 14k instructions including editor. x86 assembly. 39kb on disk.
- Famous early personal-micro compiler. Single-pass, no AST or IR. Single target.
- Proprietary (\$65) so I don't have source. Here's an ad!
- 1983-1992; lineage continues into modern Delphi compiler.



Specimen #19

Manx Aztec C

- 21k instructions, 50kb on disk.
- Contemporary to Turbo Pascal, one of many competitors.
- Unclear if AST or not, no source. Probably no IR.
- Multi-target, Z80 and 8080.
- 1980-1990s, small team.



Specimen #20

Not just the past: 8cc

- 6,740 lines of C, self-hosting, compiles to ~110kb via clang, 220kb via self.
- Don't have to use assembly to get this small! Quite readable and simple. Works.
- Single target, AST but no IR, few diagnostics.
- 2012-2016, mostly one developer.

```
static void emit_binop_int_arith(Node *node) {
    SAVE;
    char *op = NULL;
    switch (node->kind) {
        case '+': op = "add"; break;
        case '-': op = "sub"; break;
        case '*': op = "imul"; break;
        case '^': op = "xor"; break;
        case OP_SAL: op = "sal"; break;
        case OP_SAR: op = "sar"; break;
        case OP_SHR: op = "shr"; break;
        case '/': case '%': break;
        default: error("invalid operator '%d'", node->kind);
    }
    emit_expr(node->left);
    push("rax");
    emit_expr(node->right);
    emit("mov #rax, #rcx");
    pop("rax");
    if (node->kind == '/' || node->kind == '%') {
        if (node->ty->usig) {
            emit("xor #edx, #edx");
            emit("div #rcx");
        } else {
            emit("cqto");
            emit("idiv #rcx");
        }
        if (node->kind == '%')
            emit("mov #edx, #eax");
    } else if (node->kind == OP_SAL || node->kind == OP_SAR ||
               node->kind == OP_SHR) {
        emit("%s #cl, %s", op, get_int_reg(node->left->ty, 'a'));
    } else {
        emit("%s #rcx, #rax", op);
    }
}
```

Grand Finale

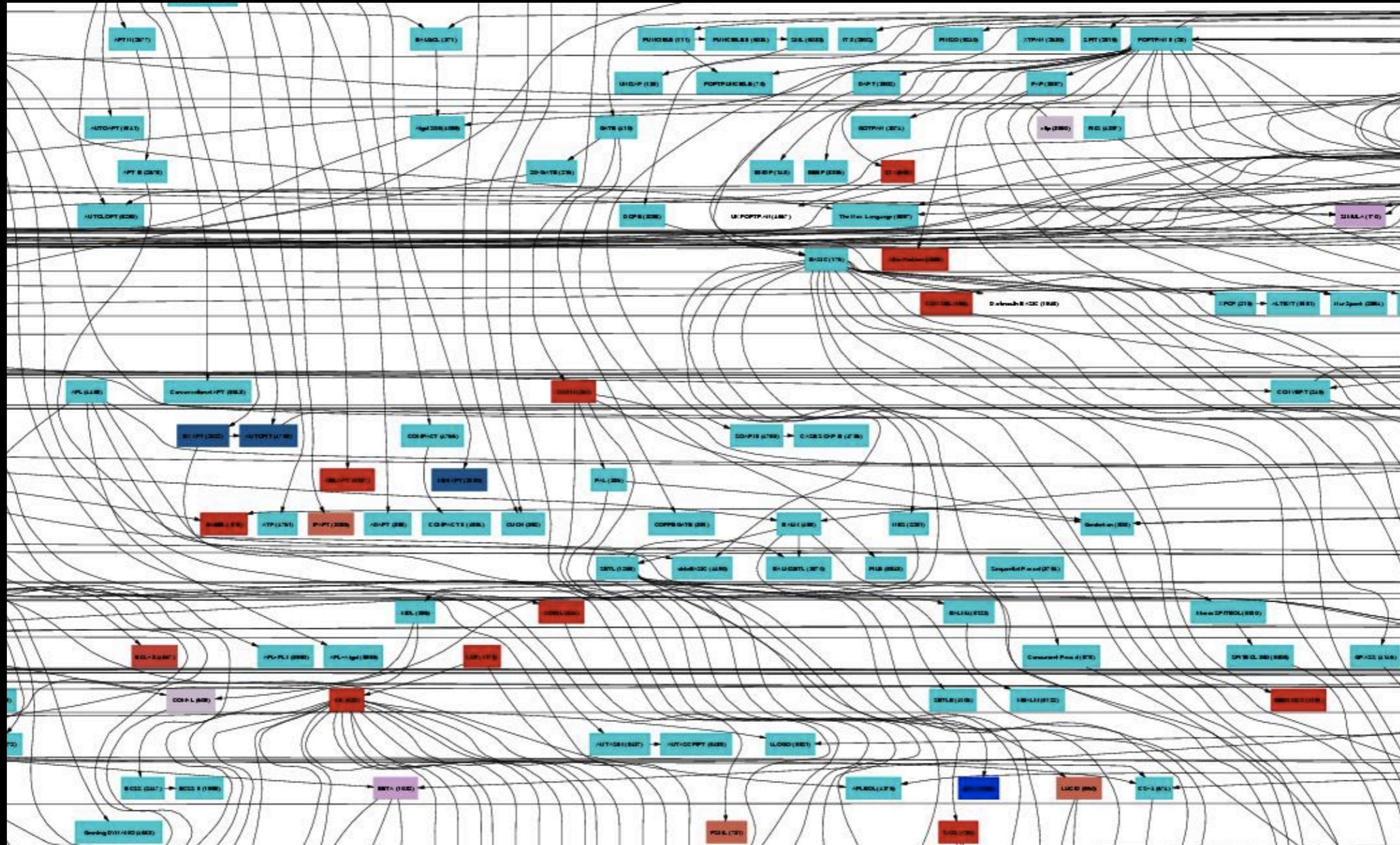
Specimen #21

JonesForth

- 692 instruction VM, 1,490 lines Forth for compiler, REPL, debugger, etc.
- Educational implementation of Forth.
- Forth, like Lisp, is nearly VM code at input (postfix not prefix).
- Minimal partial-compiler turns user "words" into chains of indirect-jumps. Machine-code primitive words.
- Interactive system with quote, eval, control flow, exceptions, debug inspector. Pretty high expressivity!
- 2009, one developer.

```
\ IF is an IMMEDIATE word which compiles 0BRANCH followed by a dummy offset, and places
\ the address of the 0BRANCH on the stack. Later when we see THEN, we pop that address
\ off the stack, calculate the offset, and back-fill the offset.
: IF IMMEDIATE
    ' 0BRANCH ,      \ compile 0BRANCH
    HERE @           \ save location of the offset on the stack
    0 ,              \ compile a dummy offset
;
: THEN IMMEDIATE
    DUP
    HERE @ SWAP -
    SWAP !          \ calculate the offset from the address saved on the stack
;                      \ store the offset in the back-filled location
: ELSE IMMEDIATE
    ' BRANCH ,
    HERE @
    0 ,
    SWAP
    DUP
    HERE @ SWAP -
    SWAP !          \ definite branch to just over the false-part
;                      \ save location of the offset on the stack
;                      \ compile a dummy offset
;                      \ now back-fill the original (IF) offset
;                      \ same as for THEN word above
```

Coda

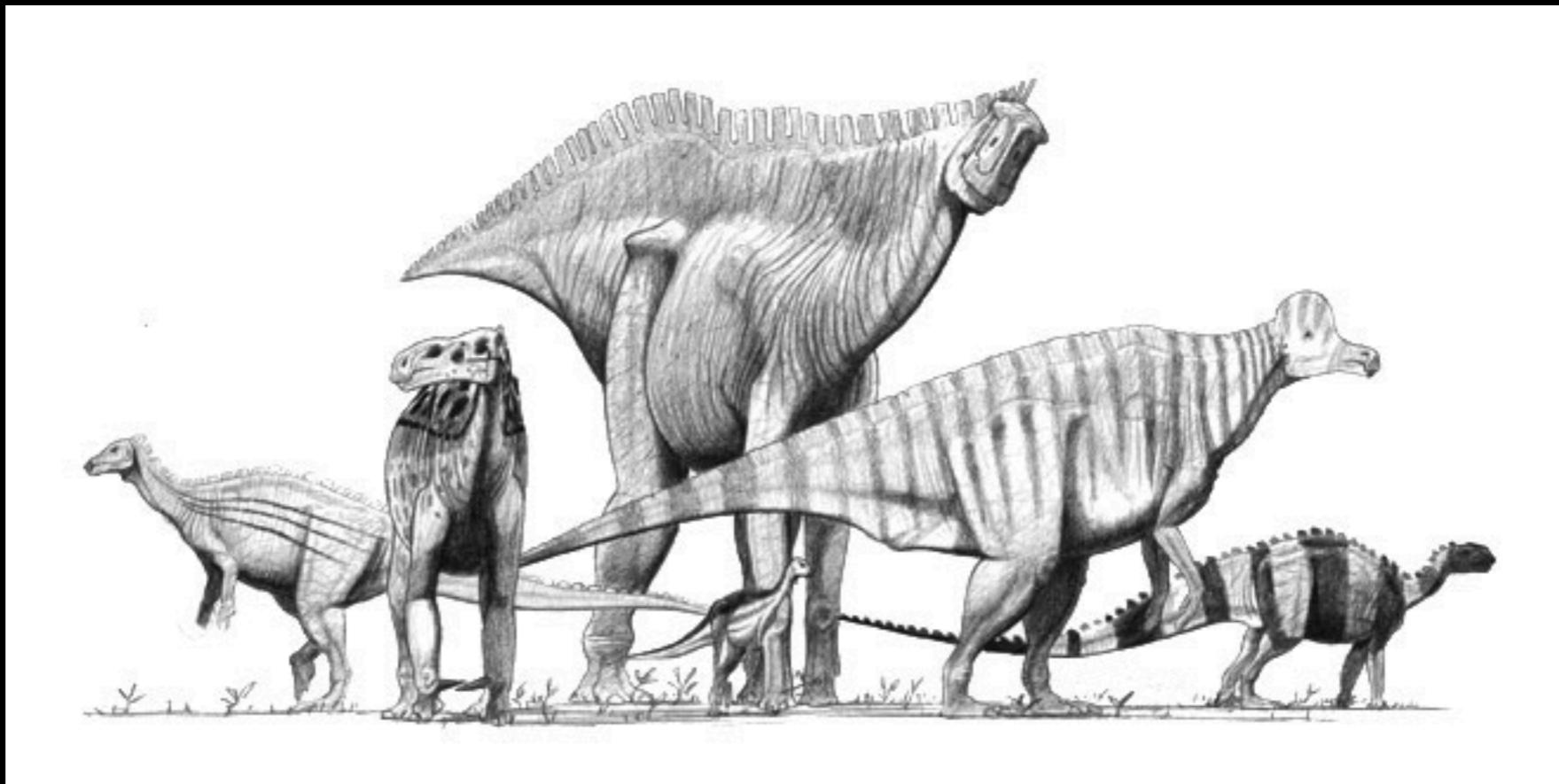


There have been *a lot* of languages

<http://hopl.info> catalogues 8,945 programming languages from the 18th century to the present

Go study them: past and present!
Many compilers possible!
Pick a future you like!

The End!



https://en.wikipedia.org/wiki/Dinosaur#/media/File:Ornithopods_jconway.jpg

(I also probably ought to mention that due to using some CC BY-SA pictures,
this talk is licensed CC BY-SA 4.0 international)