EE380, Stanford, 2007 – 02 – 14

Building your own dynamic language is fun and easy!

first steps toward reinventing computing

Ian Piumarta

Viewpoints Research Institute

ian@squeakland.org

preamble

talk and slides

- algorithms and structures for offline contemplation
- motivation for experiments
  - make to *know*, not just to have!
- pointers to useful information
- pertinent artefacts from ancient (computing) history

CA101

vpri.org

20,000 LOC to describe a complete, practical personal computer system

stepping stone to a qualitative reinvention of programming

the system is the curriculum

- ideas and ideals
- comprehensive, clear, high-level, understandable
- practical working system that is its own model
- system to learn about systems
- an *exploratorium of itself*

approach

contrast with current computing systems

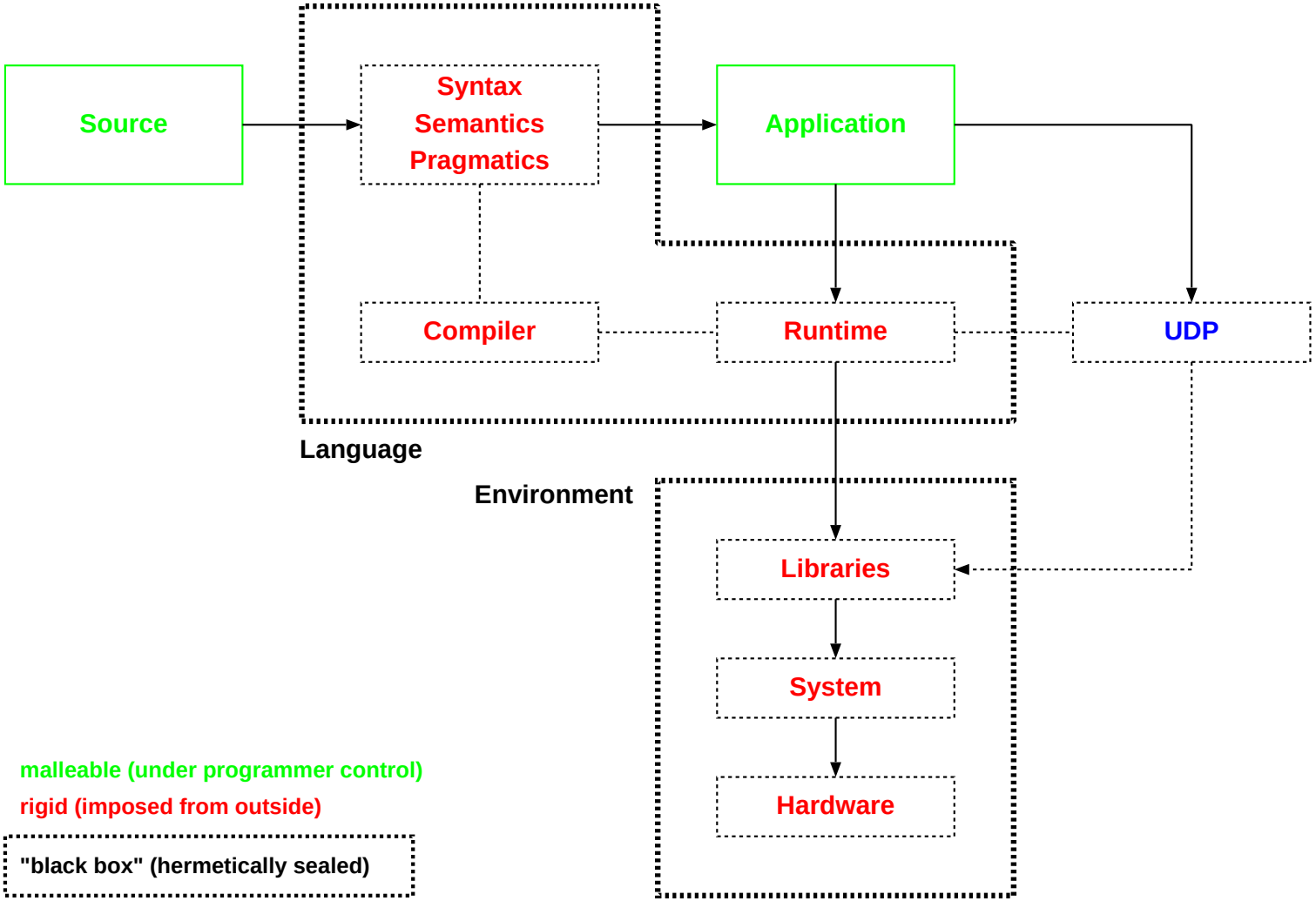
- artefacts often functional (rather than understandable or good models)
- large, complex, costly, buggy, insecure, segregated and inexpressive

models are most useful if

- powerful enough to capture the phenomena
- small enough to be comprehensible

conventional programming languages?

conventional programming



arcane (for the moment) example

incremental syntax and semantics

(typescript of demo at the end of these slides)

math wins!

algebra

- abstraction of operations and types
  - subsumes many concrete kinds of things and relations
- entire system emphasizes similarities, diminishes differences
- enormous contraction of the number of meanings that have to be specified

LISP 1.5 — self-describing function

```
evalquote[fn; x] = apply[fn; x; NIL]

apply[fn; x; a] =
  [ atom[fn]            -> [ eq[fn; CAR]  -> caar[x];
                             eq[fn; CDR]  -> cdar[x];
                             eq[fn; CONS] -> cons[car[x]; cadr[x]];
                             eq[fn; ATOM] -> atom[car[x]];
                             eq[fn; EQ]   -> eq[car[x]; cadr[x]];
                             T            -> apply[eval[fn; a]; x; a] ];
    eq[car[fn]; LAMBDA] -> eval[caddr[fn]; pairlis[cadr[fn]; x; a]];
    eq[car[fn]; LABEL]  -> apply[caddr[fn]; x;
                                 cons[cons[cadr[fn]; caddr[fn]]; a]] ]

eval[e; a] =
  [ atom[e]      -> cdr[assoc[e; a]];
    atom[car[e]] -> [ eq[car[e], QUOTE] -> cadr[e];
                      eq[car[e]; COND]  -> evcon[cdr[e]; a];
                      T                 -> apply[car[e]; evlis[cdr[e]; a];
                                                 a] ];
    T            -> apply[car[e]; evlis[cdr[e]; a]; a] ]

evcon[c; a] =
  [ eval[caar[c]; a] -> eval[cadar[c]; a];
    T                -> evcon[cdr[c]; a] ]

evlis[m; a] =
  [ null[m] -> NIL;
    T       -> cons[eval[car[m]; a]; evlis[cdr[m]; a]] ]
```

function is meaning derived from form

```
evalquote[fn; x] = apply[fn; x; NIL]

apply[fn; x; a] =
  [ atom[fn]            -> [ eq[fn; CAR]  -> caar[x];
                             eq[fn; CDR]  -> cdar[x];
                             eq[fn; CONS] -> cons[car[x]; cadr[x]];
                             eq[fn; ATOM] -> atom[car[x]];
                             eq[fn; EQ]   -> eq[car[x]; cadr[x]];
                             T            -> apply[eval[fn; a]; x; a] ];
    eq[car[fn]; LAMBDA] -> eval[caddr[fn]; pairlis[cadr[fn]; x; a]];
    eq[car[fn]; LABEL]  -> apply[caddr[fn]; x;
                                 cons[cons[cadr[fn]; caddr[fn]]; a]] ]

eval[e; a] =
  [ atom[e]      -> cdr[assoc[e; a]];
    atom[car[e]] -> [ eq[car[e], QUOTE] -> cadr[e];
                      eq[car[e]; COND]  -> evcon[cdr[e]; a];
                      T                 -> apply[car[e]; evlis[cdr[e]; a];
                                                 a] ];
    T            -> apply[car[e]; evlis[cdr[e]; a]; a] ]

evcon[c; a] =
  [ eval[caar[c]; a] -> eval[cadar[c]; a];
    T                -> evcon[cdr[c]; a] ]

evlis[m; a] =
  [ null[m] -> NIL;
    T       -> cons[eval[car[m]; a]; evlis[cdr[m]; a]] ]
```

five *elementary functions* (and one *elementary form*)

interdependence of function and form

elementary functions create and manipulate of elements of structure

- structural details are largely irrelevant
- (or at least reduced to simplest essentials)

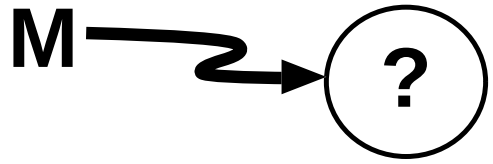
interaction with (manipulation of) structure requires function

- can we make functional details be largely irrelevant?
- (or at least reduced to simplest essentials)

form as encapsulated ‘something’: objects

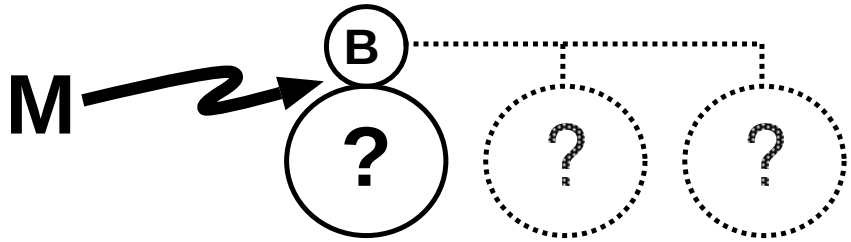
minimal object:

- encapsulates behaviour
- (subsumes state)



no assumptions about object contents

- decouple implementation from representation
- representation arbitrary
- behaviour replaceable (and shareable)
- implementation of behaviour replaceable (and shareable)



self-describing data

behaviour is associated with, and accessed uniquely through, data elements

> [...] there is a very important class of properties of elements which has to do not so much with the physical attributes of the element as with the function which is to be performed on that element by some procedure
>
> [...] the item stored in the component of the element whose name is currently in index J would be an actual TRA instruction to transfer control to the appropriate point in the flow diagram.
>
> Douglas T. Ross
> *A Generalized Technique for Symbol Manipulation and Numerical Calculation*
> ACM Conference on Symbol Manipulation, May 20–21, 1960, Philadelphia, PA

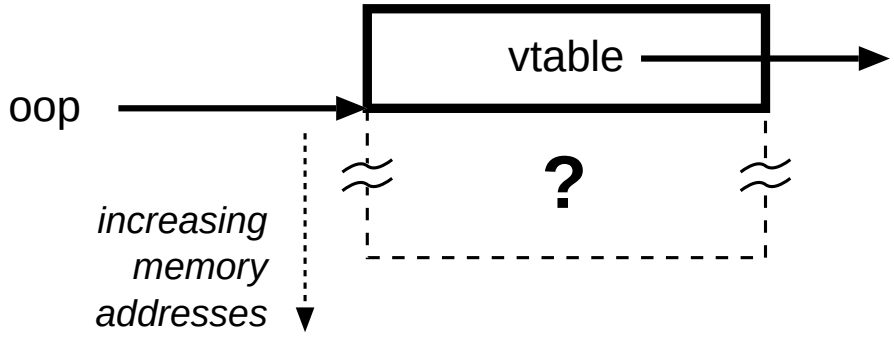
from an oo perspective

objects

- manage complexity
- encapsulate behaviour
- encapsulate (even ‘foreign’) structure
  - non-object members
- avoid namespace pollution
- are flexible building blocks for arbitrary data structures
- exhibit dualities with functions, e.g:

`apply[fn; args; alist]`	$\Leftrightarrow$	`alist.fn[alist; args]`
`alist` = environment	$\Leftrightarrow$	`alist` = ‘receiver’
namespace	$\Leftrightarrow$	method dictionary

the minimal object




```
send(message, object, ...) =
  method := object[-1].lookup(message)
  method(object, ...)
```

dynamic binding

```
bind(object, message) =
        vtbl.lookup(vtbl, message)
```

method cache

```
bind(object, message) =
      vtbl ⟵ object[-1] ;
      cache[vtbl, message] ? cache[vtbl, message]
                           : cache[vtbl, message] ⟵ vtbl.lookup(vtbl, message)
```

immediate types

```
bind(object, message) =
    vtbl ⟵ object == 0 ? vtbl_nil
                       : object & 1 ? vtbl_fixint
                                    : object[-1] ;
    cache[vtbl, message] ? cache[vtbl, message]
                         : cache[vtbl, message] ⟵ vtbl.lookup(vtbl, message)
```

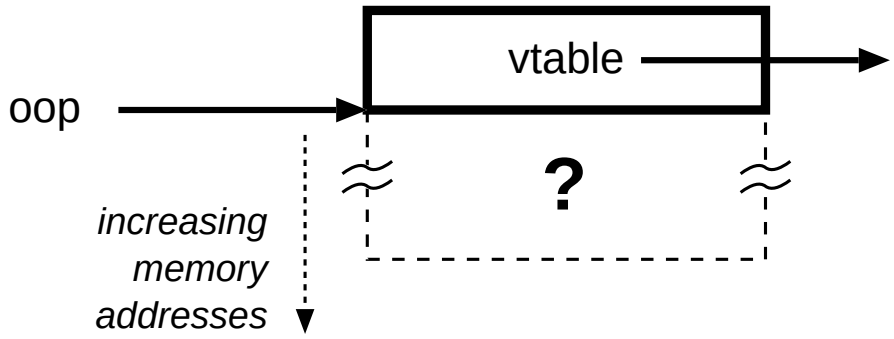
circular semantics

```
bind(object, message) =
      vtbl ⟵ object == 0 ? vtbl_nil
                         : object & 1 ? vtbl_fixint
                                      : object[-1] ;
      cache[vtbl, message] ? cache[vtbl, message]
                           : cache[vtbl, message] ⟵ vtbl.lookup(vtbl, message)

vtbl.lookup(vtbl, message) =
      (bind(vtbl, #lookup))(vtbl, message)
```

- message send implemented by sending messages
- override $\Rightarrow$ new semantics

the minimal object

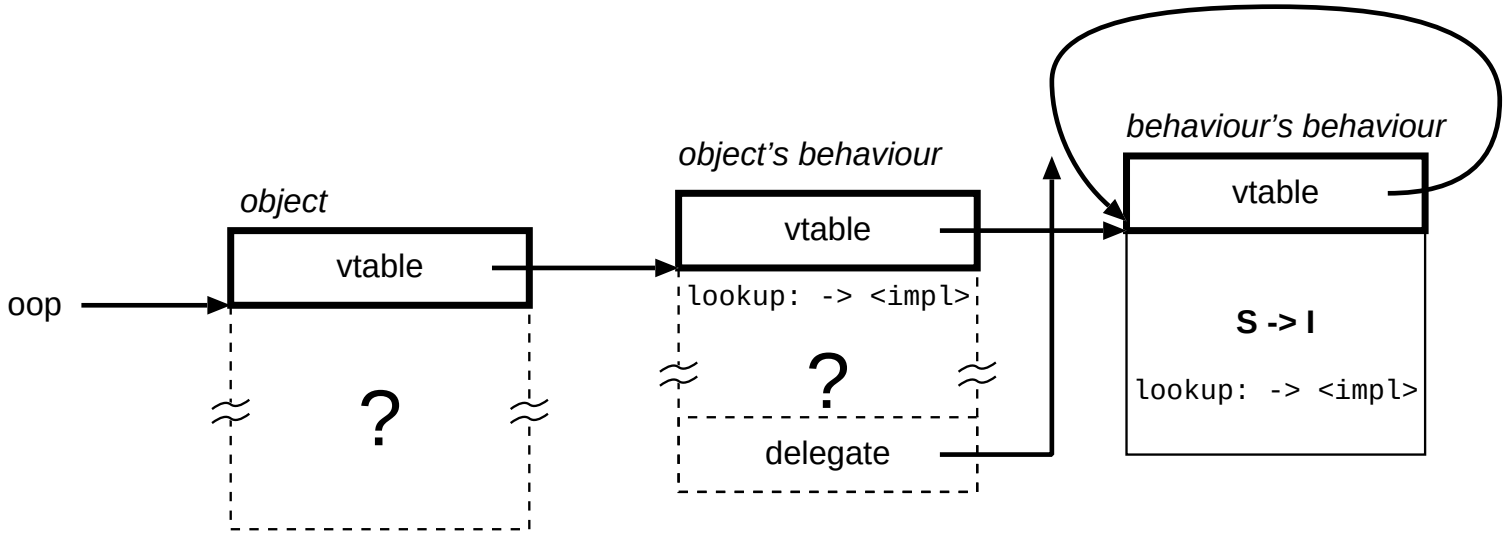



```
send(message, object, ...) =
  method := object[-1].lookup(message)
  method(object, ...)
```

vtable protocol

```
  vtable.lookup(aSelector)
  vtable.methodAtPut(aSelector, aMethodImplementation)
  vtable.intern(aString)
  vtable.allocate(objectSize)
```

everything is an object



objects *are*; methods *do*

consider VM-based language

- represent ‘does’ (computation) with some ‘is’ (CompiledMethod)
- wave magic wand (apply VM to image, class file, ...)
- *representation* of ‘does’ *indirectly* moves messages around between the ‘is’

- methods have no dynamic effect without a VM
- the VM is not an object
- the actions of methods cannot be described purely in object terms
  - bind, apply, sequence

method objects imply how objects might be animated; the animation itself comes from ‘outside’

cf., LISP 1.5: structure implies how functions might be interpreted; the implementation of structure (elementary functions and form) and its sequencing comes from ‘outside’

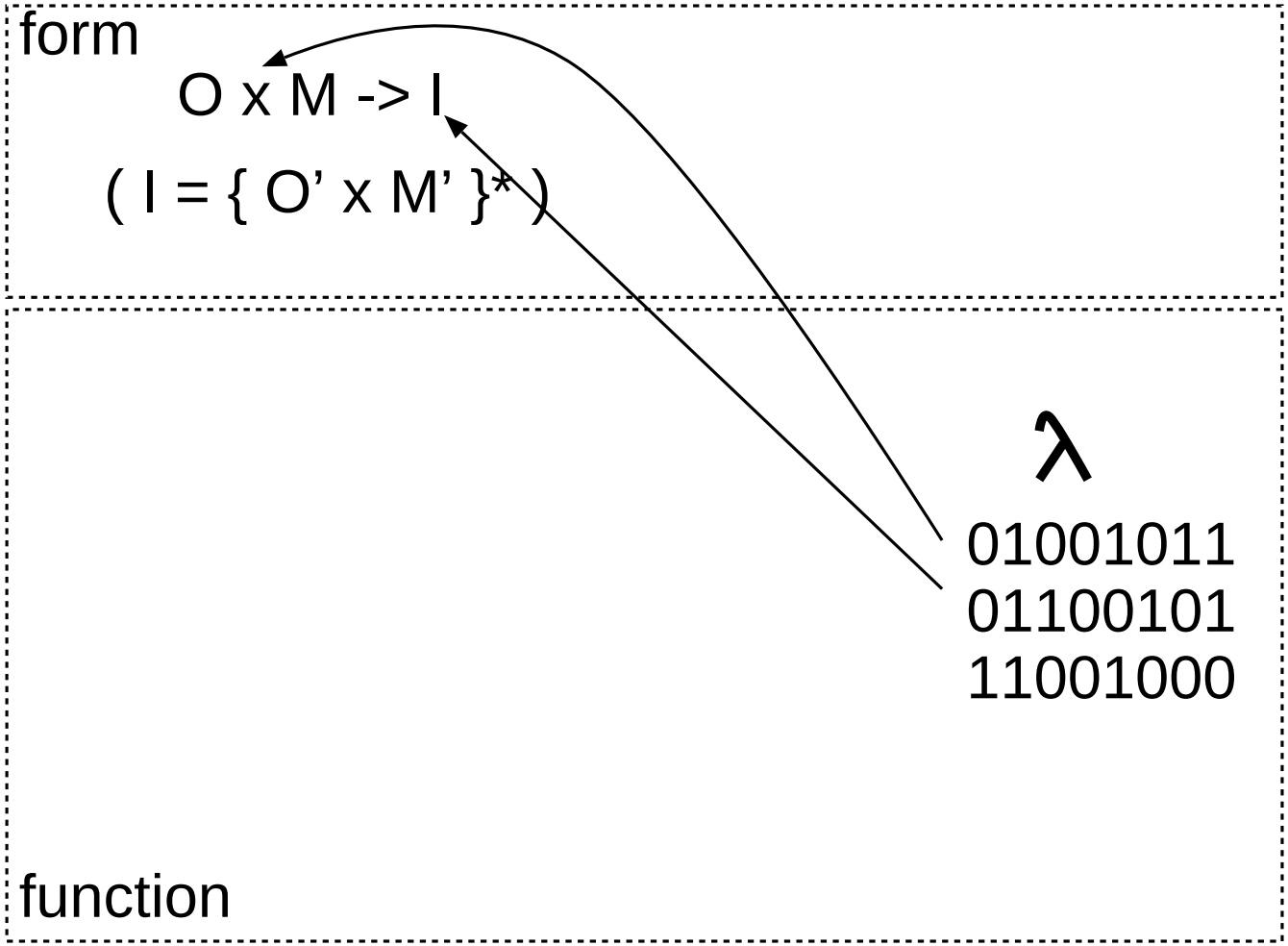
objects are form

form

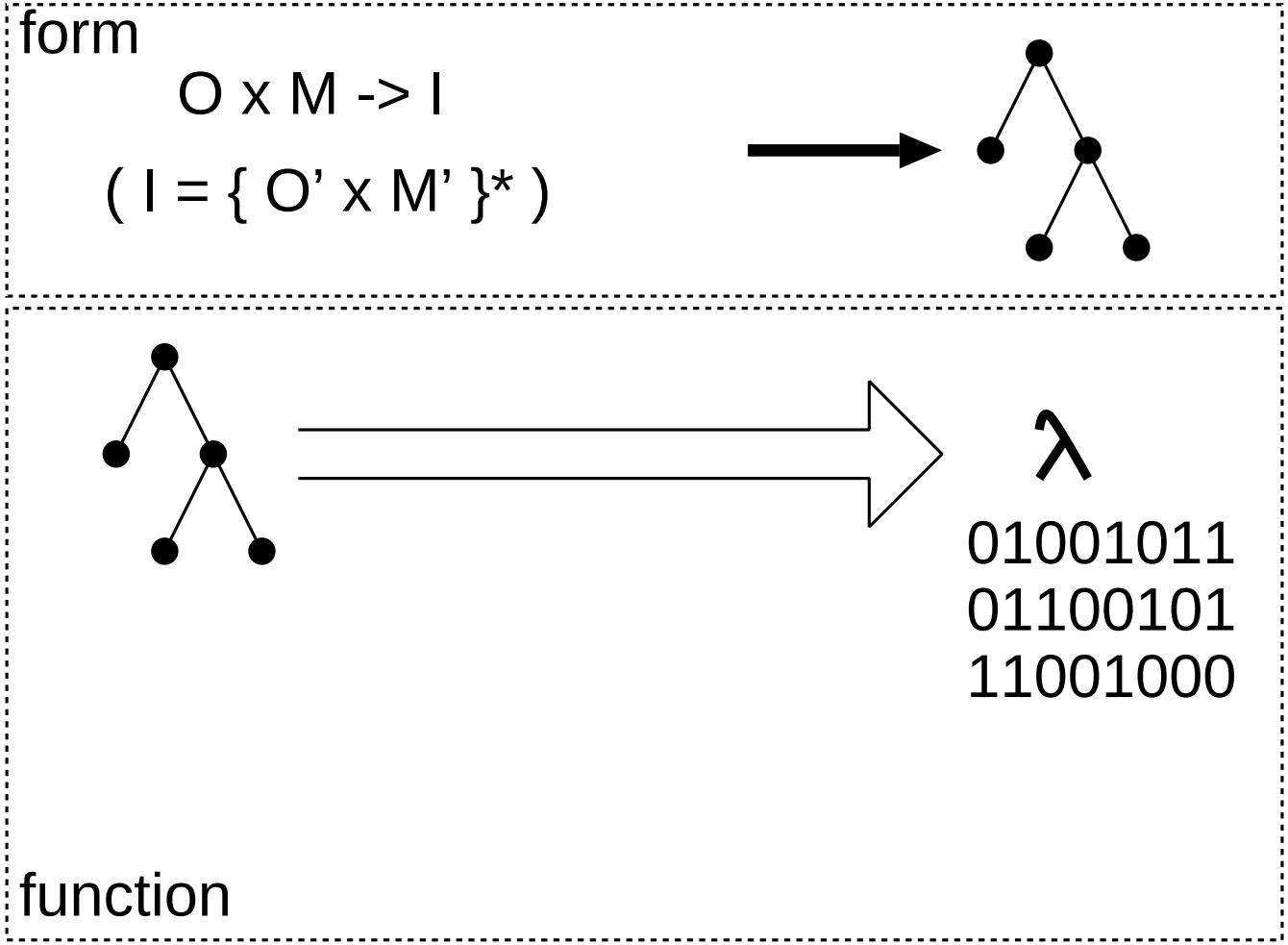
$$O \times M \rightarrow I$$

$$(I = \{ O' \times M' \}^*)$$

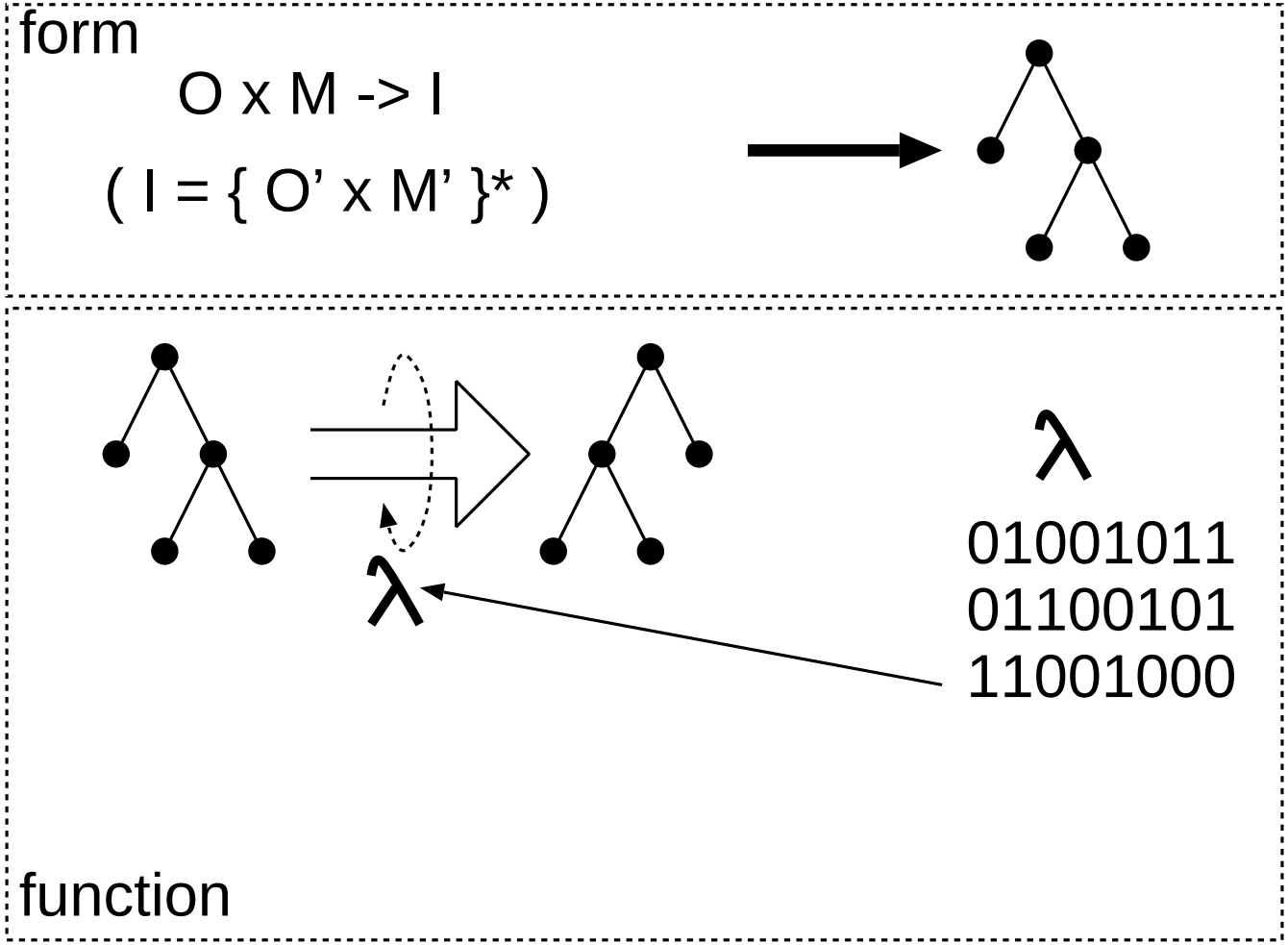
form needs function



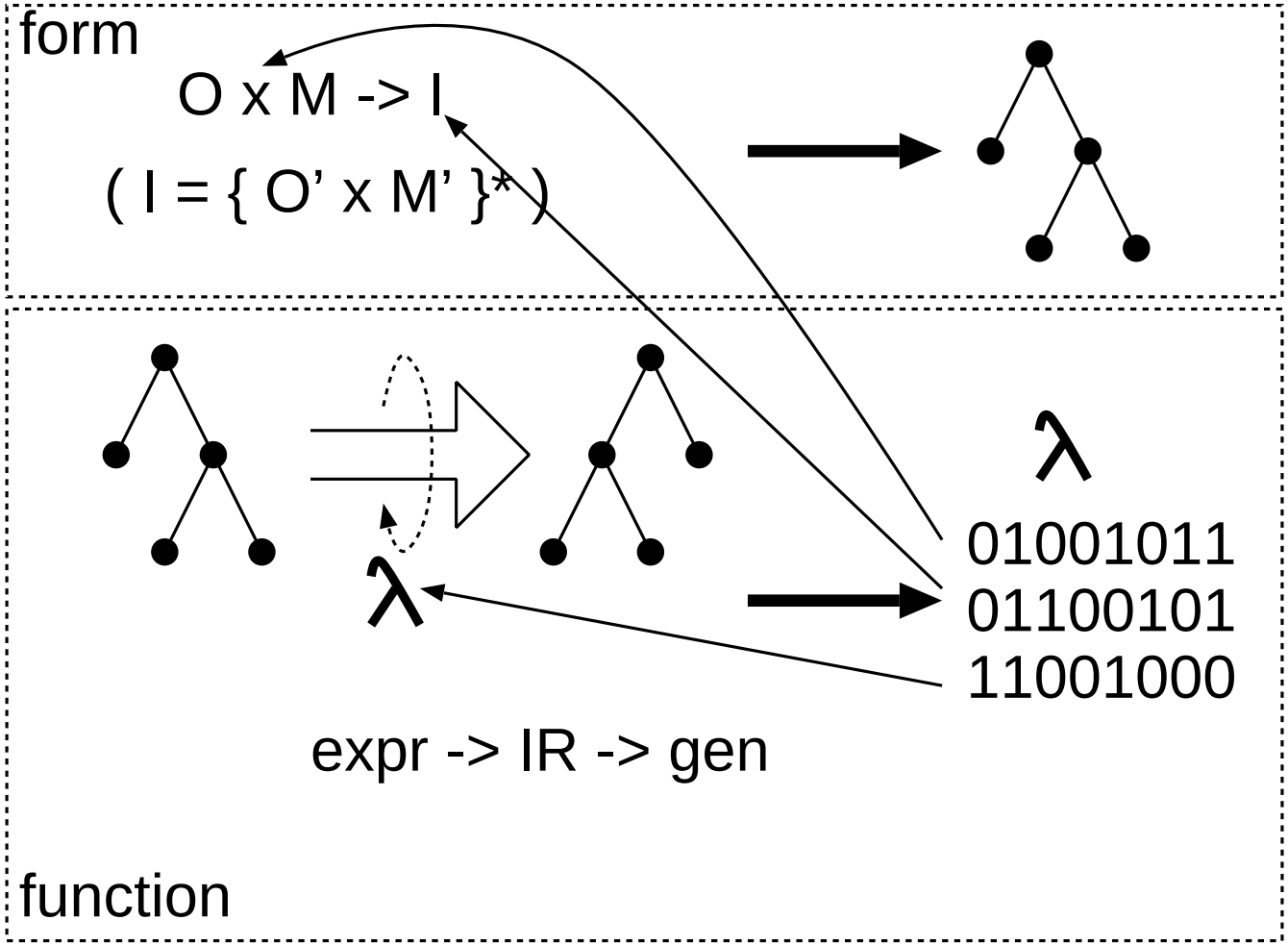
form describes function



functions transform form into function



form describes function implements form



object-function duality

analogy with physical sciences

- ‘particle’ side (objects representing meaning) needs a good model
  - often neglected in functional programming
  - the physical substance of interactions
  - almost subsumed by the generality of 1st-class functions of lists
- ‘field’ side (messaging and events animating objects) needs a good model
  - often neglected in ‘object-oriented’ programming
  - the interstitial side of interactions
  - almost invisible because the concreteness of objects

objects and functions (messaging system) are duals

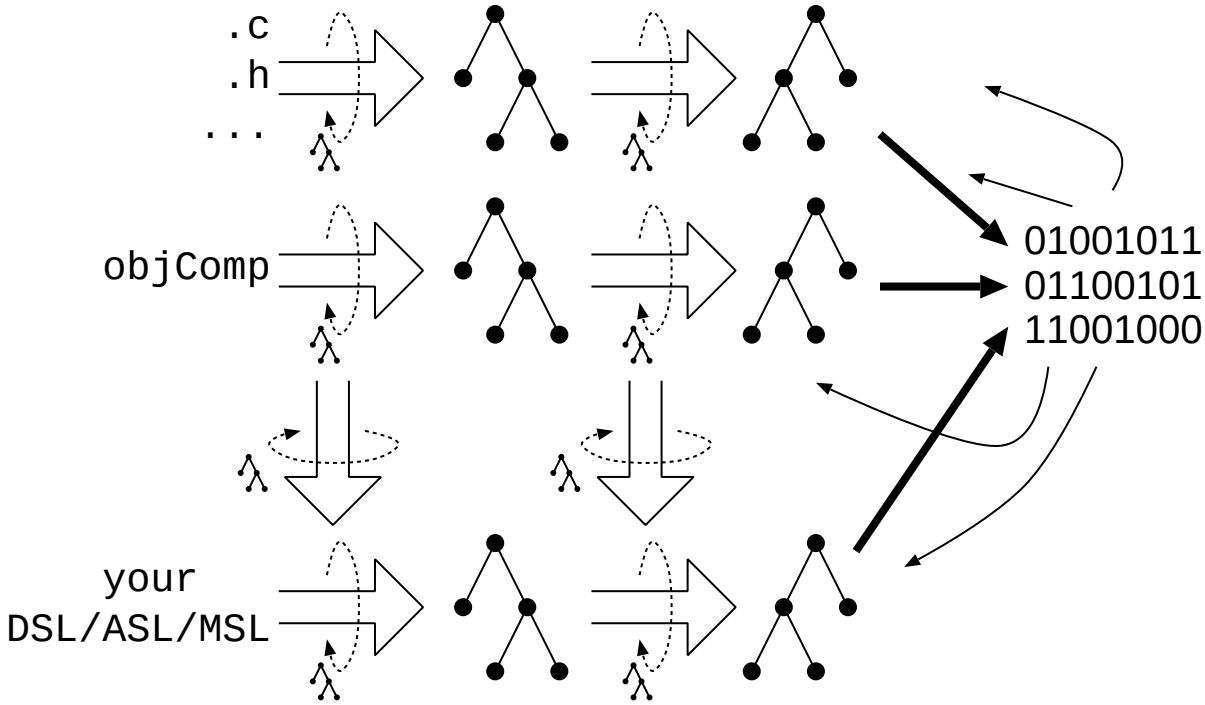
- *must* be kept in balance for the most powerful and compact modeling

complex systems $=$ typical elements $+$ dynamic relationships

- unifies (mathematically) many phenomena previously seen as different cases

everything is self-describing structure

downwards, sideways



& upwards

- lexical, syntactic, semantic, IR analysis as pattern-directed transformation

syntax and grammar

advanced recursive-descent parsing techniques (Birman, 1970)

- easy to write, read, understand
- more general and powerful than ‘traditional’ table-driver parsers
- just as amenable to analytic techniques

synergy (lexical vs. syntactic vs. semantic; delayed vs. immediate meaning & effect):

- TEX (Donald Knuth, 1981)
- input can reason about itself *during its own parsing*

the grandfather of dynamic grammars: Meta-II (Val Schorre, 1962)

- self-describing, self-implementing, self-bootstrapping dynamic syntax

free text to syntactic structure: META-II

```
.SYNTAX PROGRAM

OUT1    = '*1'          .OUT('GN1')
        / '*2'          .OUT('GN2')
        / '*'           .OUT('CI')
        / .STRING       .OUT('CL ' *) ;

OUTPUT  = (   '.OUT' '(' $ OUT1 ')'
            / '.LABEL' .OUT('LB') OUT1 ) .OUT('OUT') ;

EX3     = .ID                   .OUT('CLL' *)
        / .STRING               .OUT('TST' *)
        / '.ID'                 .OUT('ID')
        / '.NUMBER'             .OUT('NUM')
        / '.STRING'             .OUT('SR')
        / '(' EX1 ')'
        / '.EMPTY'              .OUT('SET')
        / '$' .LABEL *1 EX3     .OUT('BT ' *1)
                                .OUT('SET') ;

EX2     =   (EX3 .OUT('BF ' *1) / OUTPUT)
          $ (EX3 .OUT('BE')     / OUTPUT)
          .LABEL *1 ;

EX1     = EX2 $ ('/' .OUT('BT ' *1) EX2) .LABEL *1 ;

ST      = .ID .LABEL * '=' EX1 ';' .OUT('R') ;

PROGRAM = '.SYNTAX' .ID         .OUT('ADR' *)
          $ ST '.END'           .OUT('END') ;
.END
```

alternative approaches: LISP70

Tesler et al, 1973

```
    RULES OF MLISP =
        IF <MLISP>:X THEN <MLISP>:Y ELSE <MLISP>:Z
            -> (COND (:X :Y) (T :Z)) ,

        IF <MLISP>:X THEN <MLISP>:Y
            -> (COND (:X :Y) (T NIL)) ,

        IF <MLISP>:X
            -> <ERROR (MISSING THEN)> ,

        IF
            -> <ERROR (ILLEGAL EXPRESSION AFTER IF)> ,

        :X < :Y
            -> (LESSP :X :Y) ,

        :VAR
            -> :VAR ;
```

```
IF A < B THEN C ELSE D

=>  (COND ((LESSP A B) C)
          (T           D))
```

less interesting for parsing free text

much more interesting for manipulating syntactic structures into ‘canonical’ form

cf. Pre-Scheme (Kelsey, 1997)

canonical meaning to executable form: LISP70 contd.

```
RULES OF COMPILE =
    (COND (T :E)
        -> <COMPILE :E> ,

    (COND (:B :E) ...)
        -> <COMPILE :B>
           (DJUMPF :ELSE)
           <COMPILE :E>
           (JUMP :OUT)
           (LABEL :ELSE)
           <COMPILE (COND ...)>
           (LABEL :OUT) ,

    (LESSP :A :B)
        -> <COMPILE :A>
           <COMPILE :B>
           (FETCH (FUNCTION LESSP)) ,

    :V
        -> (FETCH (VARIABLE :V)) ;
```

machine code the LISP70 way

```
(COND ((LESSP A B) C)
      (T          D))

=>

    (FETCH (VARIABLE A))                (PUSH  P A)
    (FETCH (VARIABLE B))                (PUSH  P B)
    (FETCH (FUNCTION LESSP))            (POP   P VAL)
                                        (CAMG  VAL 0 P)
                                        (SKIPA VAL ZERO)
                                        (MOVEI VAL 1)
                                        (MOVEM VAL 0 P)
    (DJUMPF E0001)                      (POP   P VAL)
                                        (JUMPE VAL E0001)
    (FETCH (VARIABLE C))                (PUSH  P C)
    (JUMP E0002)                        (JUMPA VAL E0002)
    (LABEL E0001)                   E0001
    (FETCH (VARIABLE D))                (PUSH  P D)
    (LABEL E0002)                   E0002
```

stack machine; no types; simple (difficult to transcend local control, scope, etc.); hard to optimise

lcc

a retargetable C compiler (Fraser and Hanson, 1991)

```
(Block ()
    (JMPF L1
        (LTI4 (INDIRI4 (ADDRLP A))
              (INDIRI4 (ADDRLP B))))
    (INDIRI4 (ADDRGP C))
    (JMP L2)
    (LABEL L1)
    (INDIRI4 (ADDRGP D))
    (LABEL L2))
```

```
VOID  : (JMPF L (LTI4 REGI4 REGI4))     [ cmpl reg(3.2), reg(3.3)
                                          jlt  lbl(2) ]
REGI4 : (INDIR4 (ADDRGP))               [ movl off(2.2), reg(0) ]
REGI4 : (INDIR4 (ADDRLP))               [ movl off(2.2)(%esp), reg(0) ]
VOID  : (JMP)                           [ jmp  lbl(2) ]
```

```
    movl _A, %eax
    movl _B, %ecx
    cmpl %eax, %ecx
    jlt  L1
    movl 4(%esp), %eax
    jmp  L2
L1: movl 8(%esp), %eax
L2:
```

RTL vs. tree

instruction selection as grammar process: BURS

bottom-up rewrite system

```
reduce(tree, startSymbol) =
  foreach rule in startSymbol.startSets
    if match(tree, rule.pattern)
      rule.action()
      return startSymbol
  return false
```

```
match(tree, pattern) =
  if (pattern.isSymbol()) return reduce(tree, pattern)
  if (tree.first != pattern.first) return false
  foreach treeElement, patternElement in tree.tail, pattern.tail
    unless match(treeElement, patternElement)
      return false
  return true
```

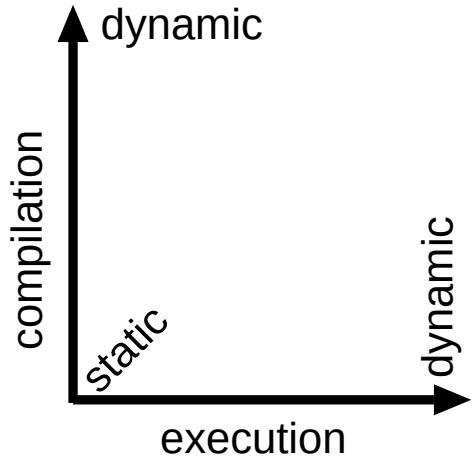
describe code generation (instruction selection) as tree rewriting

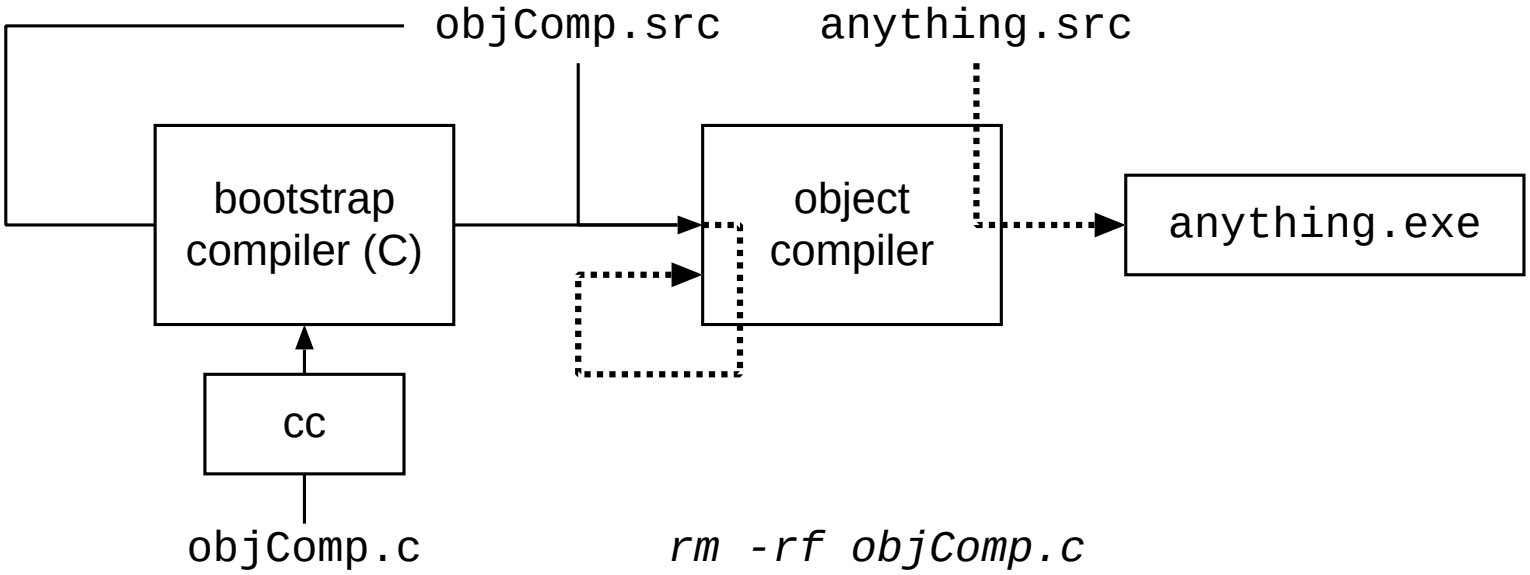
- single, unifed, simple, tiny, pervasive compilation ‘mechanism’

static implementation of a dynamic universe

independent axes

- compilation (static/offline vs. dynamic/incremental)
- execution (static/early-bound vs. dynamic/late-bound)





complexity of implementation

object compiler (in itself): 2200 LOC

transform structure (s-expressions) to canonical form: 620 LOC

code generation:

MI framework:	420 LOC
+	176 LOC (PowerPC)
+	118 LOC (Intel x86)
	714

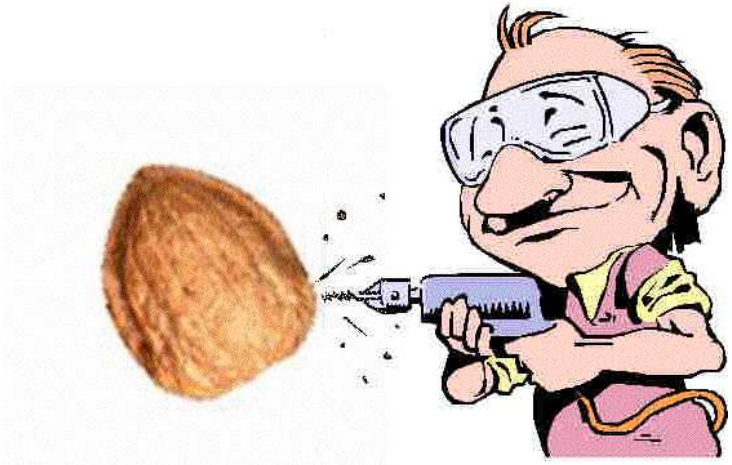
scorecard

- √ extend the program's code during execution
- √ extend objects and definitions during execution
- √ a program analyzing its own structure, code, types or data
- √ executable data structures
- √ offline and online compilation
- √ VM, just-in-time, online & offline compilation
- √ ability to directly modify machine code
- √ generating new objects from a runtime definition
- √ runtime alteration of object or type system
- √ changing the inheritance or type tree
- √ closures, continuations, introspection
- √ new language constructs, optimisations, grammar

‘typical’ programming languages

interesting stuff designed elsewhere

hermetically sealed & inaccessible



to hackers in search of better paradigms for creative expression

(or maybe just *some* way to get *necessary things* done)

‘atypical’ programming language

syntax, semantics, pragmatics built from the same *fluid* stuff

end-user systems are just syntactic, semantic, pragmatic ‘sugar’

if you don’t like the stuff or the sugar, all you need is a spoon...

less is more

> It seems that perfection might be attained not when there is nothing left to add but rather when there is nothing left to take away.
>
> Antoine de Saint-Exupéry, *Terre des Hommes, III : L'Avion*, 1939

extreme late binding

- less mechanism $\Rightarrow$ fewer assumptions to early bind
- fewer assumptions $\Rightarrow$ easier to late-bind everything
- eliminate early-bound assumptions $\Rightarrow$ generality

much of the spirit was there in the early 1960's; lost along the way

conclusion #1

objects provide form (that describes function)

- five ‘essential’ methods
- one semantic primitive (dynamic bind in the method cache)
- extrinsic implementation completely described by function model
- $\approx$ 2200 LOC for self-hosting object compiler (Smalltalk-like syntax)
- infinitely extensible/reusable ‘object framework’
- representation compatible with ‘foreign’ payloads

conclusion #2

functions describe behaviour (that implements objects)

- five (ish) ‘essential’ interpretations of structure
- one primitive form (conditional)
- extrinsic representation completely described by object model
- $\approx$ 1300 LOC for self-hosting dynamic function $\longrightarrow$ native code compiler
- infinitely extensible/reusable ‘behaviour framework’
- output is native code compatible with platform ABI

conclusion #3

take away:

- ‘dynamic’ can apply to *everything* (data, code, types, ...)
- ‘language’ can mean *all of it* (syntax, semantics, implementation, pragmatics, ...)
- it can be made very, very simple
- it can be made very, very general
- it can free you from a multitude of arbitrary, meanlingless pedantries
- it is a *lot* of fun

go home and innovate!

- built your own and share it with the world
- or use ours: releases every month or two

Form follows function — that has been misunderstood. Form and function should be one, joined in a spiritual union.

Frank Lloyd Wright, 1908

It is the grand object of all theory to make these irreducible elements as simple and as few in number as possible, without having to renounce the adequate representation of any empirical content whatever.

Albert Einstein, *Mein Weltbild*, 1934

On the contrary, most of our systems are much more complicated than can be considered healthy, and are too messy and chaotic to be used in comfort and confidence. [...] You see, while we all know that unmastered complexity is at the root of the misery, we do not know what degree of simplicity can be obtained, nor to what extent the intrinsic complexity of the whole design has to show up in the interfaces. We simply do not know yet the limits of disentanglement. We do not know yet whether intrinsic intricacy can be distinguished from accidental intricacy.

Edsger W. Dijkstra, CACM 44(3), 2001

appendix A: putting it all together (parser)

```
Stmt ::=
    "{" Stmt*:s "}"                                    => ‘(begin ,@s)
  | "var" Binding:first ("," Binding)*:rest ";"        => ‘(begin ,first ,@rest)
  | "if" "(" Expr:c ")" Stmt:t ("else" Stmt |
                                Empty => ’0):f         => ‘(if (js-bool ,c) ,t ,f)
  | "while" "(" Expr:c ")" Stmt:s                      => ‘(while (js-bool ,c) ,s)
  | "do" Stmt:stmt "while" "(" Expr:cond ")" ";"       => ‘(while (begin ,stmt ,cond
  | "for" "(" ("var" Binding | Expr):init ";"
              Expr:cond ";" Expr:upd ")" Stmt:s        => ‘(begin ,init
                                                           (while ,cond
                                                             (begin ,s ,upd)))
  | "break" ";"                                        => ’(break)
  | "continue" ";"                                     => ’(continue)
  | "return" (Expr:e => ‘(#return ,e) |
              Empty => ’(#return)):r ";"               => r
  | Expr:e ";"                                         => e
```

(most of) JavaScript parser: 86 LOC

appendix A: putting it all together (semantics)

```
(define js-set
  (lambda (lhs val)
    (match lhs
      ((js-get (js-get :c :n) :p)      `[(js-get ,c ,n) bind: ,p to: ,val])
      ((js-get (js-arr-get :a :i) :p)  `[(js-arr-get ,a ,i) bind: ,p to: ,val])
      ((js-get :c :n)                  `[,c set: ,n to: ,val])
      ((js-arr-get :a :i)              `(js-arr-set ,a ,i ,val))
      (:otherwise                      (error "%o is not assignable" lhs)))))
```

JavaScript semantics: 100 LOC

appendix A: putting it all together (JavaScript)

- semantics: 100
- parser: 86
- library: 102 (minimal Object, String, Date, Number, etc.)
- graphics: 136

just over 400 LOC

with no serious attempt at optimisation, runs a little faster than FireFox

(and a *lot* faster than WebKit aka Safari)

appendix B: syntax demo typescript

```
$ ../main
Welcome to Jolt 0.1 [VPU 5.0 i386 generic]
.(char@ "abc" 0)
 => 97
.(char@ "abc" 1)
 => 98
.(char@ "abc" 2)
 => 99
.(char@ "abc" 3)
 => 0
```

appendix B (contd.)

```
$ ../main boot.k meta-repl.k -
Welcome to Jolt 0.1 [VPU 5.0 i386 generic]
; loading: 'boot.k
; loading: 'quasiquote.k
; loading: 'syntax.k
; loading: 'debug.k
; loading: '../main.sym
; loading: 'object.k
; loading: 'match.k
; loading: 'sugar.k
; loading: 'CheckpointStream.k
; loading: 'meta.k
; loading: 'meta-coke.k
; loading: 'meta-meta.k
> with CokeTokenizer {
  Coke2 ::= Coke:c ("[" Coke2:i "]" => `(char@ ,c ,i) |
            Empty                   =>  c)
}
> (read-eval-print Coke2 StdIn 1 1 1)
> "abc"[0]
parsed: (#char@ 'abc' 0)
 => 97
> "abc"[1]
parsed: (#char@ 'abc' 1)
 => 98
> "abc"[2]
parsed: (#char@ 'abc' 2)
 => 99
> "abc"[3]
parsed: (#char@ 'abc' 3)
 => 0
```