Why Programming is a Good Medium for Expressing Poorly Understood and Sloppily Formulated Ideas

title originally by
Marvin Minsky, 1967
in *Design and Planning*

title reused by
Gerald Jay Sussman

My Essential Points

- Programming is a linguistic phenomenon, like mathematics or English.
- Programming provides novel means for us to express ourselves, as individuals.
- Programming helps clarify ideas: we must be precise and unambiguous.
- In programming there is both

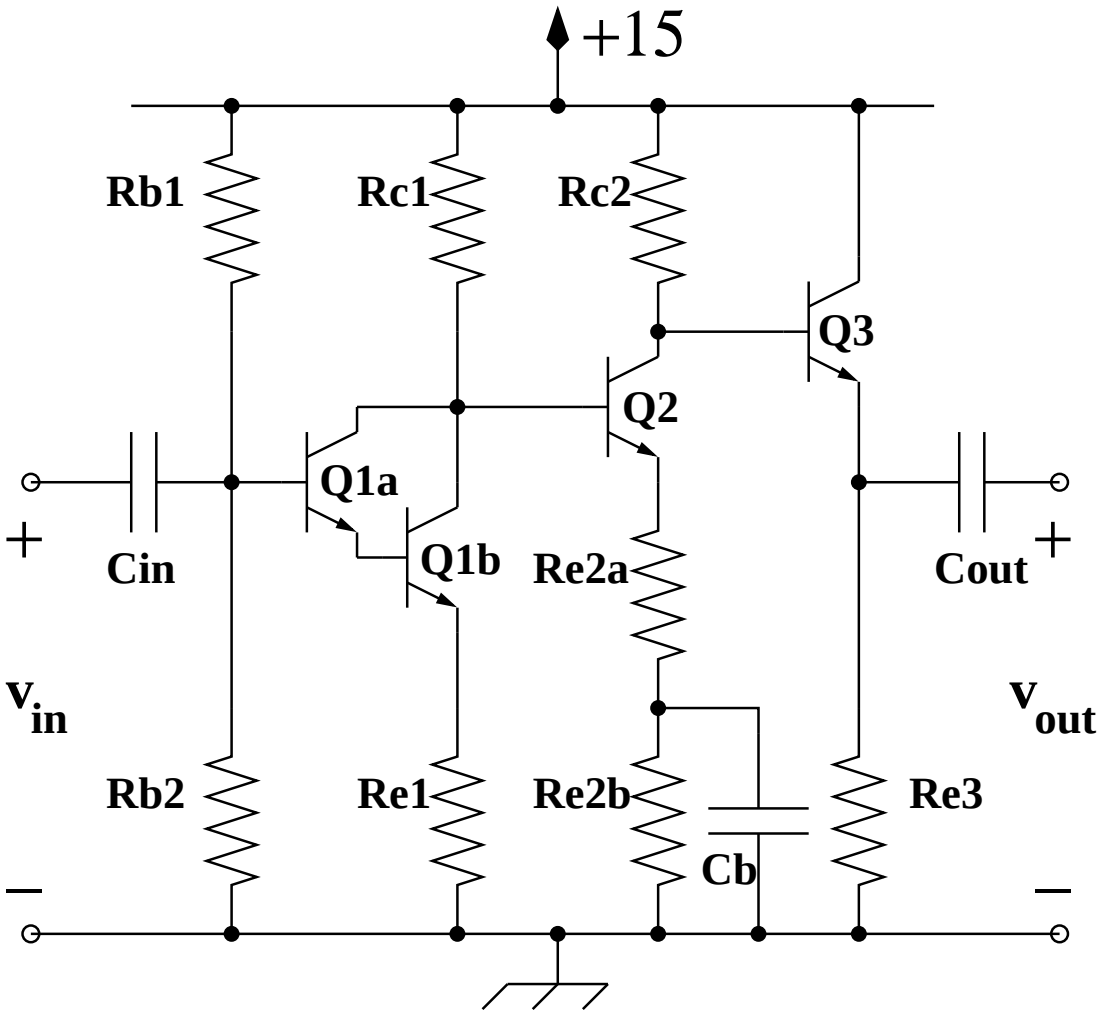
Poetry and Prose

In programs we can express

- knowledge of the world
- models of possible worlds
- structures of beauty
- emotional content

In this talk I want to illustrate this range of expression with fragments of programs

- that I have written,
- for practical use,
- for scientific research,
- for the instruction of students at MIT.


+15
Rb1
Rc1
Rc2
Q3
Q2
Q1a
Q1b
Cin
Re2a
Cout
+
+
v_{in}
v_{out}
Rb2
Re1
Re2b
Re3
Cb
–
–

```
C     A subroutine to advance the positions
C     and velocities of particles using a
C     'Leap-Frog' method.
C
      SUBROUTINE step
      CALL get_forces
      CALL get_time_step
      DO i = 1, nbodies
          vx(i) = vx(i) + time_step*ax(i)
          vy(i) = vy(i) + time_step*ay(i)
          vz(i) = vz(i) + time_step*az(i)
          x(i) = x(i) + time_step*vx(i)
          y(i) = y(i) + time_step*vy(i)
          z(i) = z(i) + time_step*vz(i)
      END DO
      time = time + time_step
      RETURN
      END
```

```
;;; Force law takes two particles p1, and p2.
;;; It returns a pair:
;;;    the acceleration of p1 due to p2
;;;    the acceleration of p2 due to p1

(define (gravitation p1 p2)
  (let* ((dx (- (position p1) (position p2)))
         (rcube (cube (euclidean-norm dx)))
         (am (/ (* G dx) rcube)))
    (cons (* -1 (mass p2) am)
          (* +1 (mass p1) am))))
```

```
;;; For a 2-body force law produces a procedure
;;; that for many bodies returns the acceleration
;;; of each body due to all of the others.

(define (force-law->acc force-law)
  (define (accelerations bodies)
    (let ((p (first bodies)) (r (rest bodies)))
      (if (= (length r) 1) ; A 2-body interaction
          (let ((inc (force-law p (car r))))
            (list (car inc) (cdr inc)))
          (let ((incs
                 (map (lambda (other)
                        (force-law p other))
                      r)))
            (cons (reduce + (map car incs))
              (map + (map cdr incs)
                   (accelerations r))))))))
  accelerations)
```

```
;;; Given a force law produces a procedure that
;;; takes a system state and returns the
;;; derivative of the state.

(define (system-derivative force-law)
  (let ((accelerations
         (force-law->acc force-law)))
    (lambda (system-state)
      (make-system 1              ; dt/dt
        (map (lambda (p a)
               (make-particle (name p)
                              0 ; dm/dt
                              (velocity p)
                              a))
             (particles system-state)
             (accelerations
              (particles system-state)))))))
```

```
;;; Given an integrator to use, an initial state
;;; and a step size h produces a stream of future
;;; states.

(define (integrate integrator initial-state h)
  (let ((step
         (integrator
          (system-derivative gravitation)
          h)))
    (define (next state)
      (cons-stream state (next (step state))))
    (next initial-state)))
```

```
;;;;      A Typical Integrator: RK4
;;; Given a system derivative function f and a
;;; step size h, returns a procedure that given a
;;; system state produces an advanced state.

(define (runge-kutta-4 f h)
  (let ((h* (scale-by h))
        (2* (scale-by 2))
        (1/2* (scale-by 1/2))
        (1/6* (scale-by 1/6)))
    (lambda (y)
      (let* ((k0 (h* (f y)))
             (k1 (h* (f (+ (1/2* k0) y))))
             (k2 (h* (f (+ (1/2* k1) y))))
             (k3 (h* (f (+ k2 y)))))
        (+ (1/6* (+ k0 (2* k1) (2* k2) k3))
           y)))))
```

Euler-Lagrange Equations

Traditional Leibnitz Notation

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{q}^i} - \frac{\partial L}{\partial q^i} = 0.$$

What are we doing here?

Consider the Lagrangian $L = \frac{1}{2}m\dot{x}^2 - V(x)$.

$\frac{\partial L}{\partial \dot{x}} = m\dot{x}$ and $\frac{\partial L}{\partial x} = -\frac{\partial V}{\partial x}$. OK, since x and $\dot{x}$ are independent variables in L.

Now $\dot{x} = dx/dt$, $\ddot{x} = d\dot{x}/dt$ so we get:

$$\frac{d}{dt}(m\dot{x}) = m\ddot{x}.$$

So the equations of motion are:

$$m\ddot{x} = -\frac{\partial V}{\partial x}.$$

What could this mean?

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{q}} - \frac{\partial L}{\partial q} = 0$$ A type error!

What it really means!

$$\frac{d}{dt}\left(\left.\frac{\partial L(t,q,\dot{q})}{\partial \dot{q}}\right|_{\substack{q = w(t) \\ \dot{q} = \frac{dw(t)}{dt}}}\right) - \left.\frac{\partial L(t,q,\dot{q})}{\partial q}\right|_{\substack{q = w(t) \\ \dot{q} = \frac{dw(t)}{dt}}} = 0$$

where w is a path through the configuration space.

Expressions to Functions

$$\frac{d}{dt}((\partial_2 L)(t, w(t), \frac{d}{dt}w(t))) - (\partial_1 L)(t, w(t), \frac{d}{dt}w(t)) = 0$$

Let $\Gamma[w](t) = (t, w(t), \frac{d}{dt}w(t))$

$$\frac{d}{dt}((\partial_2 L)(\Gamma[w](t))) - (\partial_1 L)(\Gamma[w](t)) = 0$$

and let $(Df)(t) = \left.\frac{d}{dx}f(x)\right|_{x=t}$

$$D((\partial_2 L) \circ (\Gamma[w])) - (\partial_1 L) \circ (\Gamma[w]) = 0$$

Lagrange Equations as a Program

$$D((\partial_2 L) \circ (\Gamma[w])) - (\partial_1 L) \circ (\Gamma[w]) = 0$$

```
(define ((Lagrange-equations Lagrangian) w)
  (- (D (compose ((partial 2) Lagrangian)
                 (Gamma w)))
     (compose ((partial 1) Lagrangian)
              (Gamma w))))

(define ((Gamma w) t)
  (up t (w t) ((D w) t)))
```

Precise, unambiguous, understandable, and usable: no hidden functions or magic steps!

Maxwell's Equations—Poetry in Physics

$$\text{div } \mathbf{E} = 4\pi\rho. \tag{1.1}$$

$$\text{div } \mathbf{B} = 0. \tag{1.2}$$

$$\text{curl } \mathbf{B} = \frac{1}{c}\frac{\partial \mathbf{E}}{\partial t} + \frac{4\pi}{c}\mathbf{J}. \tag{1.3}$$

$$\text{curl } \mathbf{E} = \frac{-1}{c}\frac{\partial \mathbf{B}}{\partial t}. \tag{1.4}$$

Electrical charges are conserved:

$$\text{div } \mathbf{J} + \frac{\partial \rho}{\partial t} = 0 \tag{1.5}$$

The Wave Equation

The curl of equation (1.4) is

$$\text{curl curl } \mathbf{E} = \frac{-1}{c}\frac{\partial}{\partial t} \text{ curl } \mathbf{B}. \tag{1.6}$$

Expand the left-hand side

$$\text{grad div } \mathbf{E} - \text{Lap } \mathbf{E} = \frac{-1}{c}\frac{\partial}{\partial t} \text{ curl } \mathbf{B}, \tag{1.7}$$

and substitute from equations (1.3) and (1.1).

Get the inhomogeneous wave equation:

$$\text{Lap } \mathbf{E} - \frac{1}{c^2}\frac{\partial^2 \mathbf{E}}{\partial t^2} = 4\pi(\text{grad } \rho + \frac{1}{c^2}\mathbf{J}). \tag{1.8}$$

EVAL/APPLY—Poetry in Programs

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (var-lookup exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((if? exp) (eval-if exp env))
        ((assignment? exp) (assignment exp env))
        ((definition? exp) (definition exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (eval-list (operands exp) env)))
        (else (error "Unknown expression" exp))))
```

```
(define (apply proc args)
  (cond ((primitive-proc? proc)
         (apply-primitive-proc proc args))
        ((compound-proc? proc)
         (eval-sequence (proc-body proc)
           (extend-environment
             (proc-parameters proc)
             args
             (proc-environment proc))))
        (else (error "Unknown proc" proc))))
```

```
(define (eval-list exps env)
  (if (no-operands? exps)
      '()
      (cons (eval (first-operand exps) env)
            (eval-list (rest-operands exps)
                       env))))

(define (eval-if exp env)
  (if (true? (eval (if-predicate exp) env))
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env)))

(define (eval-sequence exps env)
  (cond ((last-exp? exps)
         (eval (first-exp exps) env))
        (else (eval (first-exp exps) env)
              (eval-sequence (rest-exps exps)
                             env))))
```

```
(define (assignment exp env)
  (set-variable-value! (assignment-variable exp)
      (eval (assignment-value exp) env)
      env)
  'ok)

(define (definition exp env)
  (define-variable! (definition-variable exp)
      (eval (definition-value exp) env)
      env)
  'ok)
```

Wiring diagrams can be programs

```
(define (half-adder a b s c)
  (let ((d (make-wire)) (e (make-wire)))
    (or-gate a b d)
    (and-gate a b c)
    (inverter c e)
    (and-gate d e s)))

(define (full-adder a b c-in sum c-out)
  (let ((s (make-wire))
        (c1 (make-wire))
        (c2 (make-wire)))
    (half-adder b c-in s c1)
    (half-adder a s sum c2)
    (or-gate c1 c2 c-out)))
```

This was a revelation!

```
;;; Nondeterminism (implicit search),
;;;  can use lambda-calculus glue.

(define (require p)
  (if (not p) (amb)))

(define (an-element-of list)
  (require (not (null? list)))
  (amb (car list)
       (an-element-of (cdr list))))

(define (prime-sum-pair list1 list2)
  (let ((a (an-element-of list1))
        (b (an-element-of list2)))
    (require (prime? (+ a b)))
    (list a b)))
```

From: Edgar Allen Poe, (1846)
The Philosophy of Composition,

"I select 'The Raven' as most generally known. It is my design to render it manifest that no one point in its composition is referable either to accident or intuition—that the work proceeded step by step, to its completion, with the precision and rigid consequence of a mathematical problem."

From: James Boyk, Concert Pianist

"A work of art is a machine with an aesthetic purpose."