Recursion Theorem

Droste®
HAARLEM - HOLLAND
cacao
Netto 250 g e

Scheme Mystery

```
((lambda (x)
   (list x (list 'quote x)))
 (quote (lambda (x)
          (list x (list 'quote x)))))
```

Py

- _='_=%r;print(_%%_)';print(_%_)

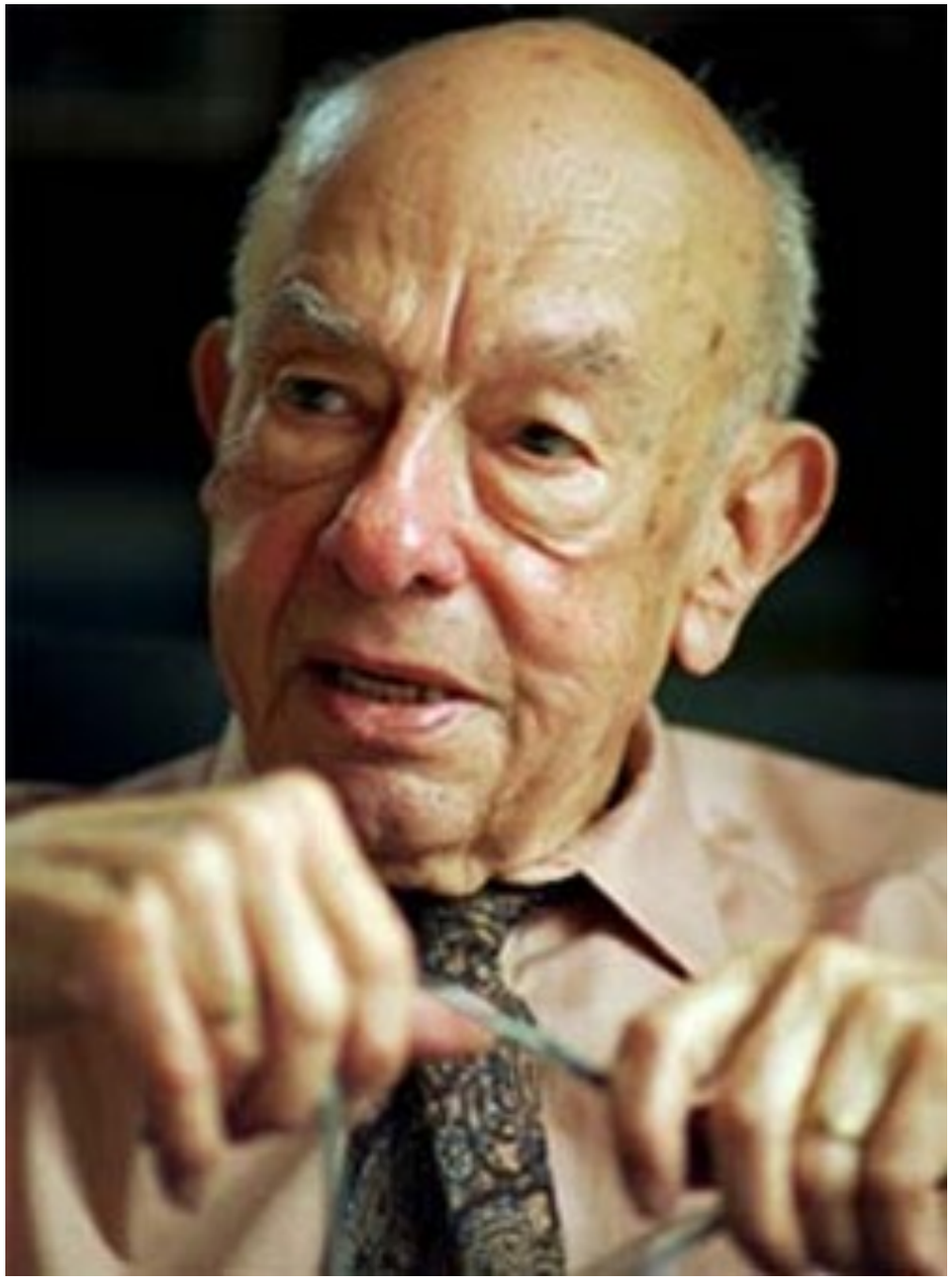
C Quine

```
main(){char *c="main(){char *c=%c%s%c;printf(c,34,c,34);}";printf(c,34,c,34);}
```

English

Print out two copies of the following, the second one in quotes:

"Print out two copies of the following, the second one in quotes:"



- To be is to be the value of a variable.
- I have been accused of denying consciousness, but I am not conscious of having done so.

Willard van Orman Quine

Partial Factorial

- $n!_k$ = if $n<k$ then $n!$ else $\perp$
- $n!_{k+1}$ = if $n=0$ then 1 else $n.(n-1)!_k$

- $f(g) = \lambda n.$ if $n=0$ then 1 else $n.g(n-1)$
- $!_{k+1} = f(!_k)$ $\quad$ $!_0 = \perp$
- $! = f(!)$

Recursion Theorem

For every program f: Haskell → Haskell there exists a single-argument function c such that

$$c \equiv f(c).$$

Proof: Let $c = k(k)$ where $k = \lambda w.f(w(w))$

$$c \equiv k(k) \equiv (\lambda w.f(w(w)))(k) \equiv f(k(k)) \equiv f(c)$$

*Haskell is a purely functional language using lazy evaluation, like Baby

Example: Factorial

- f(g) = λn. if n=0 then 1 else n.g(n-1)

- c(f) ⇒ λz.f(c) (z) where c=...

- c(0) ⇒ f(c) (0) ⇒ if 0=0 then 1 else ... ⇒ 1
- c(1) ⇒ f(c) (1) ⇒ if 0=1... else 1·c(0) ⇒ 1·1 ⇒ 1
- c(2) ⇒ f(c) (2) ⇒ if 0=2... else 2·c(1) ⇒ 2·1 ⇒ 2

Recursion

Have $c = k(k)$ where $k = \lambda w.f(w(w))$

Suppose f is the body of a recursive definition:

$$f(g) = \lambda y.\ \text{if } A(y) \text{ then } D(y) \text{ else } E(g,y)$$

Then c (for this f) is the recursive function itself:

$$c(y) \Rightarrow k(k)(y) \Rightarrow (\lambda w.f(w(w)))(k)(y) \Rightarrow f(k(k))(y)$$

$$\Rightarrow (\lambda y.\ \text{if } A(y) \text{ then } D(y) \text{ else } E(k(k),y))(y)$$

$$\Rightarrow (\text{if } A(y) \text{ then } D(y) \text{ else } E(k(k),y) \Rightarrow$$

- $D(y)$
- $E(c,y)$

Why Haskell?

- Haskell uses normal-order evaluation, not applicative order.
- When evaluating f(e), f is evaluated first, not e.
- For applicative order, more complicated:

  Let c := h(k) where

  k(w) := f(h(w))

  h(y) := λz.y(y)(z)

Scheme

```
(let* ((f (lambda (g) (lambda (n)
                  (if (= n 0) 1 (* n (g (- n 1)))))))))
       (h (lambda (y) (lambda (z) ((y y) z))))
          (k (lambda (x) (f (h x))))
          (c (h k)))
    (c 3))
```

==> 6

Viruses

The recursion theorem implies the existence of self-replicating virus programs!

Suppose f(msg)(n) is "boo!; mail msg(n+1) to n"

$c := k(k)$, $k(w) := f(w(w)) \equiv \lambda n.$ [boo!; mail $w(w)$ (n+1) to n]

$c(0) \Rightarrow$ [boo!; mail $k(k)$ (1) to 0]

$\equiv$ [boo!; mail $c(1)$ to 0] $\Rightarrow$ etc.