What lies beneath the Language of Thought

Steve Piantadosi
UC Berkeley, Psychology
July, 2018

Humans learn many formal systems

- **Basic logic** (e.g., and, or, not, iff)
- **Natural language logic** (e.g. “and”, “or”)
- **First-order logic quantifiers** (e.g. $\forall,\exists$)
- **Second-order quantification** (e.g. there exists a property P …)
- **Generalized quantifiers** (e.g. natural language “most”)
- **Grammars** (e.g. context-free grammars)
- **Programming languages** (e.g. python, haskell, prolog)
- **Tree structures and relations** (e.g. kinship systems)
- **Dominance hierarchies/relations** (e.g. Putin > Trump)
- **Physics** (e.g. block stacking)
- **Arbitrary graphs** (e.g. subway map)
- **Games** (e.g. tic-tac-toe, nim, battleship)
- **Simulations** (e.g. hypotheticals)
- **Mathematics** (e.g. calculus, algebra)
- **Reasoning** (e.g. knights and naves)

Where does all of this come from?

and, or not, if, while,
for, pair, define,
plus, times, empty,
equals, etc.

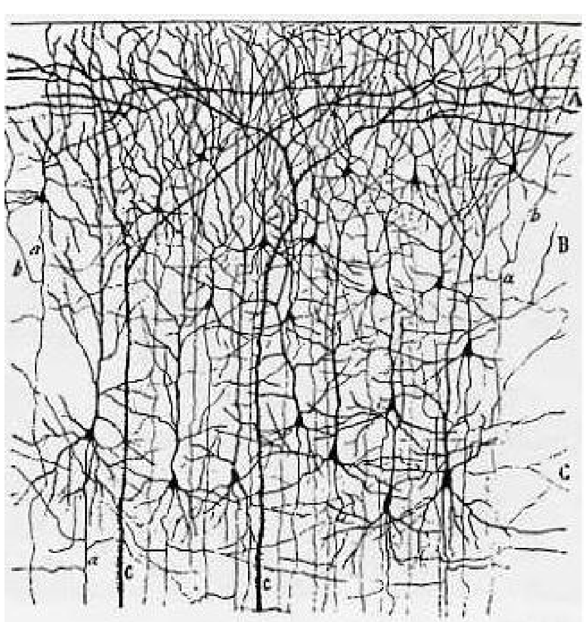
```
if(not empty(s)) {
    w = “one”
    for(s in S) {
        w = wordAfter(w)
    }
}
```

Bayesian program
induction

Two central problems

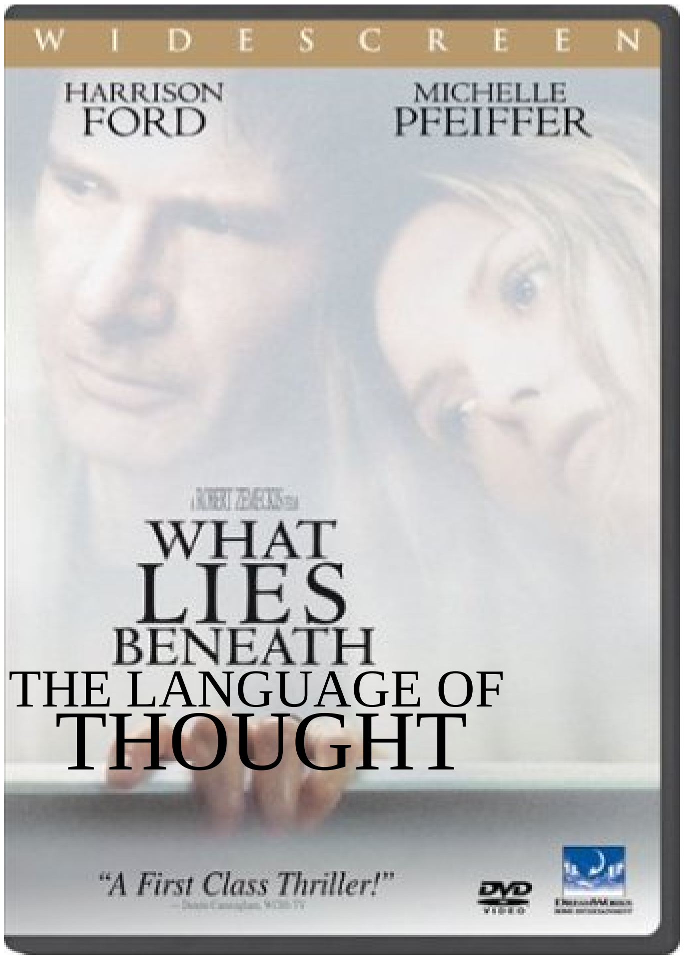
- **What can we learn without presupposing the necessary concepts?**
  - For instance: do we need to assume logic? What if newborns don't know logic? If they don't, what could it possibly be built from? Are we committed to newborns having the full power of a compiler?

- **How can we talk about *programs* (if statements, logic, sets, etc.) when real brains look like *this* crazy stuff?**



```
if(not empty(s)) {
    w = "one"
    for(s in S) {
        w = wordAfter(w)
    }
}
```

THE LANGUAGE OF
THOUGHT



The key idea

- **View the LOT as a first and foremost a system for <u>encoding</u>**
  - a general model learner
  - grounded in simple underlying dynamics – not a normal programming language
    (e.g. not a C compiler – or even a scheme compiler)
  - informed by cognitive psychology about what is natural

    (composition, structure, recursion)

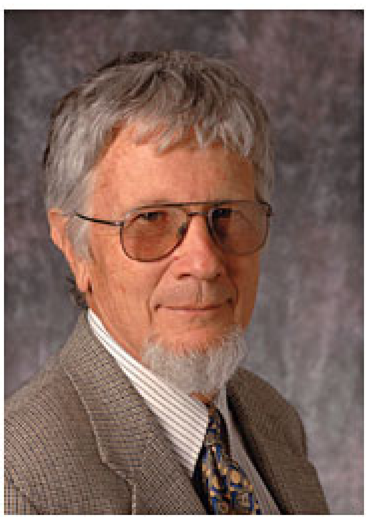
- **Learning is creating a representation that is <u>isomorphic</u> to some thing in the world.**
  - LOT expressions must "act like" stuff in the world
  - The primitives must let it "act like" anything we can comprehend

Human learn many formal systems

- **Basic logic** (e.g., and, or, not, iff)
- **Natural language logic** (e.g. “and”, “or”)
- **First-order logic quantifiers** (e.g. $\forall,\exists$)
- **Second-order quantification** (e.g. there exists a property P …)
- **Generalized quantifiers** (e.g. natural language “most”)
- **Grammars** (e.g. context-free grammars)
- **Programming languages** (e.g. Python, Haskell, Prolog)
- **Tree structures and relations** (e.g. kinship systems)
- **Dominance hierarchies/relations** (e.g. Putin > Trump)
- **Physics** (e.g. block stacking)
- **Arbitrary graphs** (e.g. Boston subway map)
- **Games** (e.g. tic-tac-toe, nim, battleship)
- **Simulations** (e.g. hypotheticals)

Isomorphism as the heart of representation

A mental representation is a functioning isomorphism between a set of processes in the brain and a behaviorally important aspect of the world. This way of defining a representation is taken directly from the mathematical definition of a representation. To establish a representation in mathematics is to establish an isomorphism (formal correspondence) between two systems of mathematical investigation (for example, between geometry and algebra) that permits one to use one system to establish truths about the other (as in analytic geometry, where algebraic methods are used to prove geometric theorems). [Gallistel]



Int. J. Systems Sci., 1970, vol. 1, No. 2, 89-97

EVERY GOOD REGULATOR OF A SYSTEM MUST BE A MODEL OF THAT SYSTEM[1]

Roger C. Conant

Department of Information Engineering, University of Illinois, Box 4348, Chicago, Illinois, 60680, U.S.A.

and W. Ross Ashby

Biological Computers Laboratory, University of Illinois, Urbana, Illinois 61801, U.S.A.[2]

[Received 3 June 1970]

The design of a complex regulator often includes the making of a model of the system to be regulated. The making of such a model has hitherto been regarded as optional, as merely one of many possible ways.

m this paper a theorem is presented which shows, under very broad conditions, that any regulator that is maximally both successful and simple *must* be isomorphic with the system being regulated. (The exact assumptions are given.) Making a model is thus necessary.

The theorem has the interesting corollary that the living brain, so far as it is to be successful and efficient as a regulator for survival, *must* proceed, in learning, by the formation of a model (or models) of its environment.

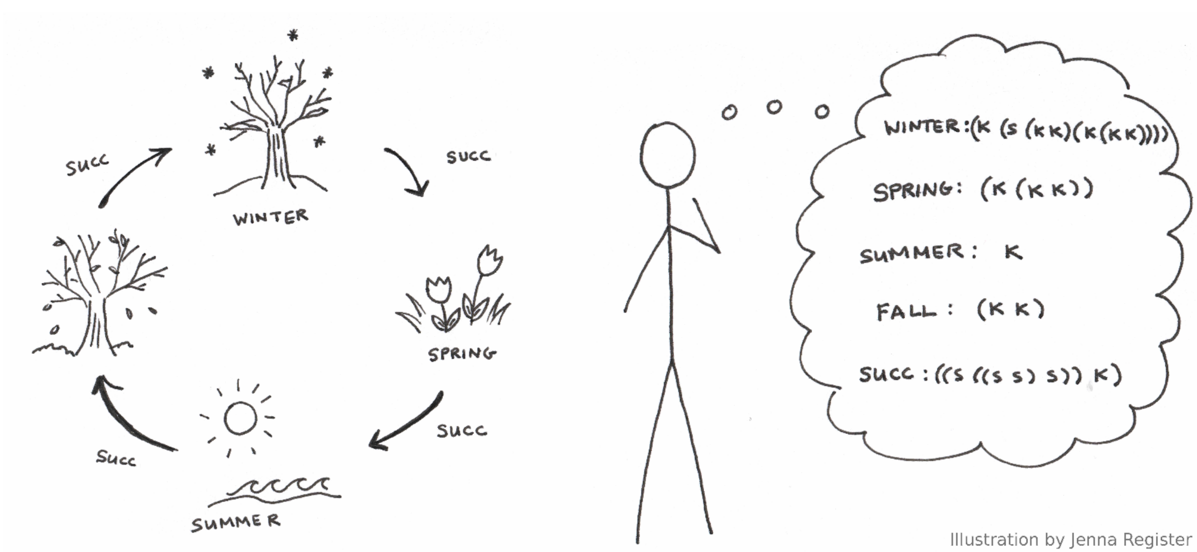
1. INTRODUCTION

Today, as a step towards the control of complex dynamic systems, models are being used ubiquitously. Being modelled, for instance, are the air traffic flow around New york, the endocrine balances of the pregnant sheep, and the flows of money among the banking centres.

Combinatory logic (or something like it)

- **A solution to several problems:**
  - CL allows encoding of arbitrary logical systems (is Turing-complete)
  - CL is based in very simple dynamics
    (which themselves specify only how CL terms interact)

    (no *cognitive* primitives)



Combinatory logic

Moses Schönfinkel

John von Neumann

Haskell Curry

The LOT does not need (explicit) variables

- **Variable binding is a problem in neuroscience / cognitive science**
  In fact, central in debates about representation (Marcus 2003)

$$f(x) = \underbrace{x+1} \qquad\qquad f(x,y) = \underbrace{x+y+1}$$

The LOT's syntax of changes depending on the context! What a nightmare!

- **Not a problem in CL:** variables emerge *only implicitly* through how primitives treat their arguments
  - f(x)=x+1 can be written as (S + (K 1)), no need for x.

Combinatory logic is super simple

Rule 1: (**K** x y) $\rightarrow$ x

or in other notation K(x,y) $\rightarrow$ x

Rule 2: (**S** x y z) $\rightarrow$ ((x z) (y z))

or in other notation, S(x,y,z) $\rightarrow$ x(z,y(z))

Currying – a function can take the next arguments in line

e.g. ((K x) y) $\rightarrow$ (K x y) $\rightarrow$ x

For example f(x)=x+1

- **Let f := (S + (K 1))**
- **Then,**

```
(f 7) :=  ((S + (K 1)) 7)      ; Definition of f
→ (S + (K 1) 7)                ; Currying
→ ((+ 7) ((K 1) 7))            ; Definition of S
→ (+ 7 ((K 1) 7))              ; Currying
→ (+ 7 (K 1 7))                ; Currying
→ (+ 7 1)                      ; Definition of K
```

But we can do better

- But what about terms like “+” and “1”? These are still “standard” LOT operations whose meaning must be determined elsewhere.

- **<u>Pure combinatory logic</u>** permits computation <u>without</u> any primitives other than **S, K.**

Motivating question

- **Is there any learning system that can acquire the most primitive computational concepts?**
  - True/false
  - If/then
  - Quantification
  - Iteration/Recursion
  - Data structures (e.g. lists/trees)
  - Identity function

An example

```
true   := (K K)
false  := K
and    := ((S (S (S S))) (K (K K)))
or     := ((S S) (K (K K)))
not    := ((S ((S K) S)) (K K))
```

```
(or true false) = (((S S) (K (K K)))  (K K)  K)
```

An example

```
true   := (K K)
false  := K
and    := ((S (S (S S))) (K (K K)))
or     := ((S S) (K (K K)))
not    := ((S ((S K) S)) (K K))
```

```
(or true false) = (((S S) (K (K K)))  (K K)  K)
→ (((S S) (K (K K)) (K K))  K)      ; Currying rule
```

An example

```
true   := (K K)
false  := K
and    := ((S (S (S S))) (K (K K)))
or     := ((S S) (K (K K)))
not    := ((S ((S K) S)) (K K))
```

```
(or true false) = (((S S) (K (K K)))  (K K)  K)
→ (((S S) (K (K K)) (K K))  K)        ; Currying rule
→ ((S S (K (K K)) (K K)) K)           ; Currying rule twice
```

An example

```
true   := (K K)
false  := K
and    := ((S (S (S S))) (K (K K)))
or     := ((S S) (K (K K)))
not    := ((S ((S K) S)) (K K))
```

```
(or true false) = (((S S) (K (K K)))  (K K)  K)
→ (((S S) (K (K K)) (K K))  K)          ; Currying rule
→ ((S S (K (K K)) (K K)) K)             ; Currying rule twice
→ ((S (K K) ((K (K K)) (K K))) K)       ; Definition of S
```

An example

```
true   := (K K)
false  := K
and    := ((S (S (S S))) (K (K K)))
or     := ((S S) (K (K K)))
not    := ((S ((S K) S)) (K K))
```

```
(or true false) = (((S S) (K (K K)))  (K K)  K)
→ (((S S) (K (K K)) (K K))  K)          ; Currying rule
→ ((S S (K (K K)) (K K)) K)             ; Currying rule twice
→ ((S (K K) ((K (K K)) (K K))) K)       ; Definition of S
→ ((S (K K) (K (K K) (K K))) K)         ; Currying
→ ((S (K K) (K K)) K)                   ; Definition of K
→ (S (K K) (K K) K)                     ; Currying
→ ((K K) K ((K K) K))                   ; Definition of S
→ ((K K K) ((K K) K))                   ; Currying
→ (K ((K K) K))                         ; Definition of K
→ (K (K K K))                           ; Currying
→ (K K)                                 ; Definition of K
```

Church encoding

- **This technique is known as <u>church encoding</u>.**

... suppose we have a program that does some complicated calculation with numbers to yield a boolean result. If we replace all the numbers and arithmetic operations with [combinator]-terms representing them and evaluate the program, we will get the same result. Thus, in terms of their effects on the overall result of programs, there is no observable difference between the real numbers and their Church-[encoded]numeral representations. (Pierce 2002)


HOT HEISS
7753

Churiso

- **My lab has been working on a library to infer church encodings from simple relational information.**

	Observations	Mental representation (theory)
Seasons	(succ winter) → spring (succ spring) → summer (succ summer) → fall (succ fall) → winter	
1, 2, Many	(succ one) → two (succ two) → many (succ many) → many	many := (**S** **K** **K**) two := (**K** (**S** **K** **K**)) one := **K** succ := ((**S** (**S** **K** **K**)) (**K** (**S** **K** **K**)))

- **How we infer:** use ideas from the inductive LOT – prefer encodings with <u>short running time</u> + <u>simple structure</u>.

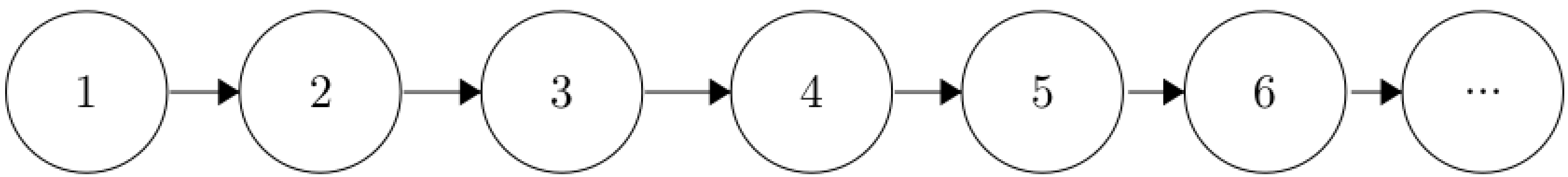
Combinator generalization

- The <u>key</u> feature is that the best CL encoding of some relations will extend to novel, unseen relations.

Number
($\mathbb{Z}$)

```
(succ one) → two
(succ two) → three
(succ three) → four
```

```
succ := K
one := S
two := (K S)
three := (K (K S))
four := (K (K (K S)))
```

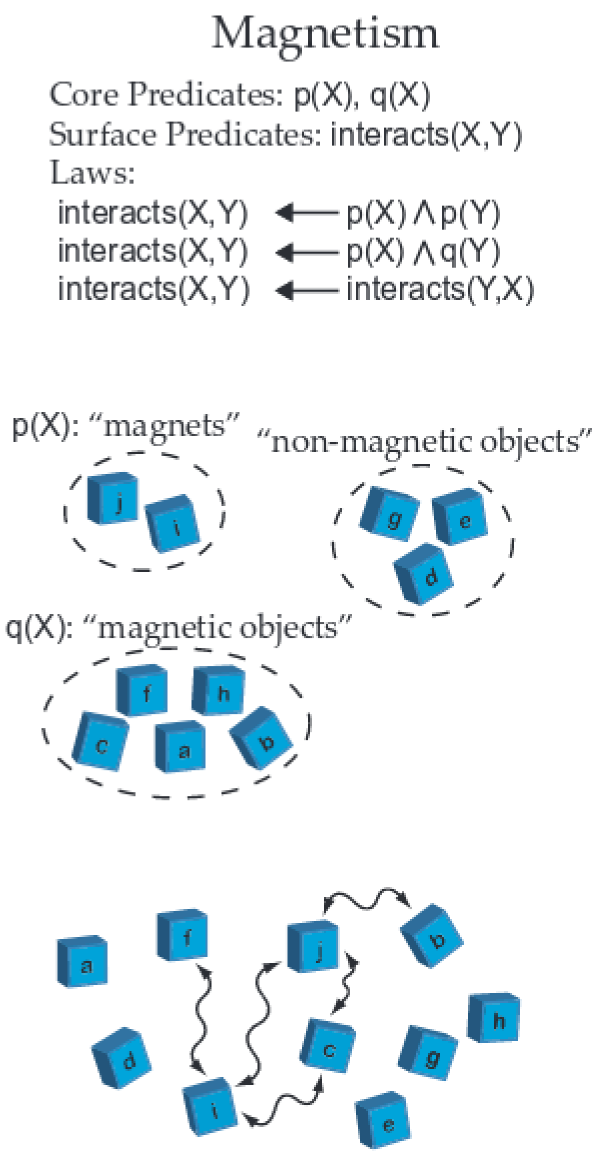


Even-Odd
($\mathbb{Z}_2$)

```
(succ one) → two
(succ two) → three
(succ three) → four

(even one) ↛ True
(even two) → True
(even three) ↛ True
(even four) → True

(odd one) → True
(odd two) ↛ True
(odd three) → True
(odd four) ↛ True
```

```
odd := ((S ((S (K S) K) K)) (S K K
    ↪))
even := ((S (S K K)) (K K))

one := (((S (K (S (K (S S (K K)))
    ↪K)) S) (S K K)) (K K))
succ := (((S (K (S (K (S S (K K)))
    ↪ K)) S) (S K K)) K)
```

Magnetism



Ullman, Goodman, & Tenenbaum (2012)

Magnetism

```
(attract p1 p2) ↛ True
(attract p2 p1) ↛ True
(attract p1 n1) → True
(attract p1 n2) → True
(attract p2 n1) → True
(attract p2 n2) → True
(attract n1 n2) ↛ True
(attract n2 n1) ↛ True
(attract n1 p1) → True
(attract n1 p2) → True
(attract n2 p1) → True
(attract n2 p2) → True
```

```
attract := ((S S) (K I))
n1      := K
n2      := K
p2      := (K K)
p1      := (K K)
```

Magnetism

```
(attract p1 p2) ↛ True
(attract p2 p1) ↛ True
(attract p1 n1) → True
(attract p1 n2) → True
(attract p2 n1) → True
(attract p2 n2) → True
(attract n1 n2) ↛ True
(attract n2 n1) ↛ True
(attract n1 p1) → True
(attract n1 p2) → True
(attract n2 p1) → True
(attract n2 p2) → True
; and one single example
(attract n1 x) → True
```

```
attract := ((S S) (K I))
n1      := K
n2      := K
p2      := (K K)
p1      := (K K)
```

Magnetism

```
(attract p1 p2) ↛ True
(attract p2 p1) ↛ True
(attract p1 n1) → True
(attract p1 n2) → True
(attract p2 n1) → True
(attract p2 n2) → True
(attract n1 n2) ↛ True
(attract n2 n1) ↛ True
(attract n1 p1) → True
(attract n1 p2) → True
(attract n2 p1) → True
(attract n2 p2) → True
; and one single example
(attract n1 x) → True
```

```
attract := ((S S) (K I))
n1      := K
n2      := K
p2      := (K K)
p1      := (K K)
x       := (K K)
```

Dominance

A
|
B
|
C
|
D

Dominance

$(a \succ b \succ c \succ d)$

```
True := K
(dom a b) → True
(dom a c) → True
; No information a,d relation
(dom b c) → True
(dom b d) → True
(dom c d) → True
(dom b a) ↛ True
(dom c a) ↛ True
(dom c b) ↛ True
(dom d b) ↛ True
(dom d c) ↛ True
(dom b a) ↛ True
(dom c b) ↛ True
(dom d c) ↛ True
```

```
a := (K (K K))
b := (S (S K))
c := (S K K)
d := (K K)
dom := (((S (K (S (K (S S (K K))) K)) S
    ↪) (S (K (S (K (S S (K K))) K)) S)
    ↪) K)
```

Formal languages

Language	Facts	Representation

Regular
$((ab)^n)$
	`(a start) → state_a`
	`(b state_a) → accept`
	
	`(a accept) → state_a`
	`(b accept) → invalid`
	
	`(a invalid) → invalid`
	`(b invalid) → invalid`
		`start := (K (K (S K K)))`
		`a := ((S (S K K)) K)`
		`b := ((S (S K K)) ((S (K (S (K (S S (K K))) K)) S) (S K K)))`
		
		`invalid := (((S (K (S (K (S S (K K))) K)) S) (S K K)) (S K K))`
		`accept := (S K K)`

Context-free
$(a^n b^n)$
	`(a start) → got_a`
	`(b got_a) → accept`
	
	`(a got_a) → got_aa`
	`(b got_aa) → want_b`
	`(b want_b) → accept`
	
	`(a got_aa) → got_aaa`
	`(b got_aaa) → want_bb`
	`(b want_bb) → want_b`
	
	`(a got_aaa) → got_aaaa`
	`(b got_aaaa) → want_bbb`
	`(b want_bbb) → want_bb`
		`start := S`
		`a := (S (K K))`
		`b := (((S (K (S (K (S S (K K))) K)) S) (S K K)) S)`
		`accept := (K (S S))`

Quantifiers

Existential
$(\exists z \ldots)$

```
(start True) → accept
(start False) → reject

(reject True) → accept
(reject False) → reject

(accept True) → accept
(accept False) → accept
```

```
True := ((S S) K)
False := ((S (S K K)) K)
start := ((S (S K K)) K)
accept := ((S (K (S S K))) (K (K (S S K
    ↪))))
reject := ((S (S K K)) K)
```

Domain	Facts	Representation
Reversal	(reverse x y) → (y x)	reverse := ((S (K (S (S K K)))) K)
If-else	True := (K K) False := K (ifelse True x y) → x (ifelse False x y) → y	ifelse := ((S ((S K) S)) (S K))
Identity	(identity x) → x	identity := (S K K)
Repetition	(repeat f x) → (f (f x))	repeat := ((S (S (K S) K)) (S K K) ↪)
Recursion	(Y f) ⇢ (f (Y f))	Y := (((S (K S) K) ((S ((S (K (S (↪K (S S (K K))) K)) S) (S (S ↪(S K K)))))) S)) (S (K S) K))
Mutual recursion	(f (g (Y⋆ f g))) ⇢ (Y⋆ f ↪ g)	Y⋆ := (((S (K S) K) (((S (K S) K) ↪((S (S (K S) K)) ((S (K (S (↪K (S S (K K))) K)) S) (S (K ↪S) K)))) (S ((S (K (S (K (S ↪S (K K))) K)) S) (S K K))))) ↪ (S (K S) K))
Apply	(apply f x) → (f x)	apply = (S K K)
Tree, List	(first (pair x y)) → x (rest (pair x y)) → y	pair := (((S (K S) K) (S (K (S (K ↪(S S (K K))) K)) S)) ((S (K ↪(S (K (S S (K K))) K)) S) (S ↪ (K (S (K (S S (K K))) K)) S ↪))) first := ((S (S K K)) (K (S K))) rest := ((S (S K K)) (K K))

A wonderful feature of CL

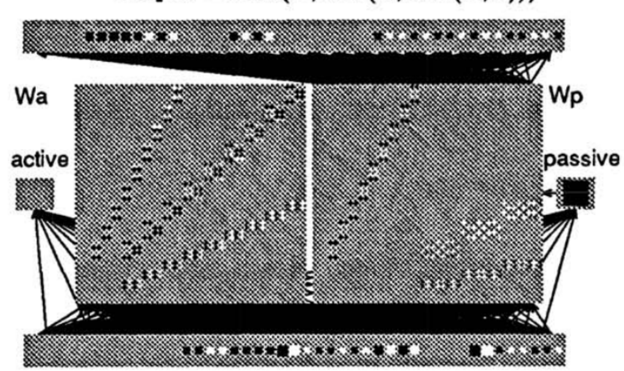
- **Combinatory logic requires only two simple rules:**

  $(\mathbf{K}\ x\ y) \rightarrow x$

  $(\mathbf{S}\ x\ y\ z) \rightarrow ((x\ z)\ (y\ z))$

  - Both are tree transformations
  - Many neural implementations of trees, and simple manipulations (e.g. Smolensky's tensor product coding, Boltzcons, etc.)

Output = cons(V,cons(C,cons(A,B)))

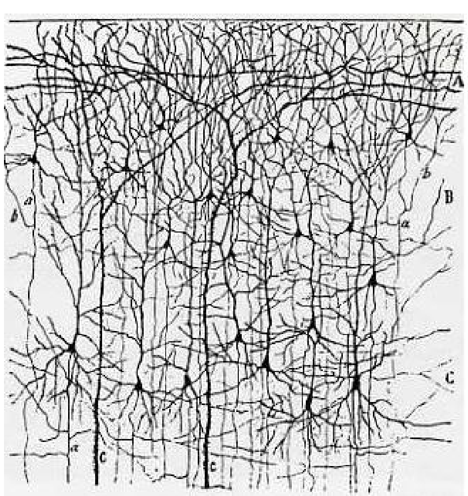


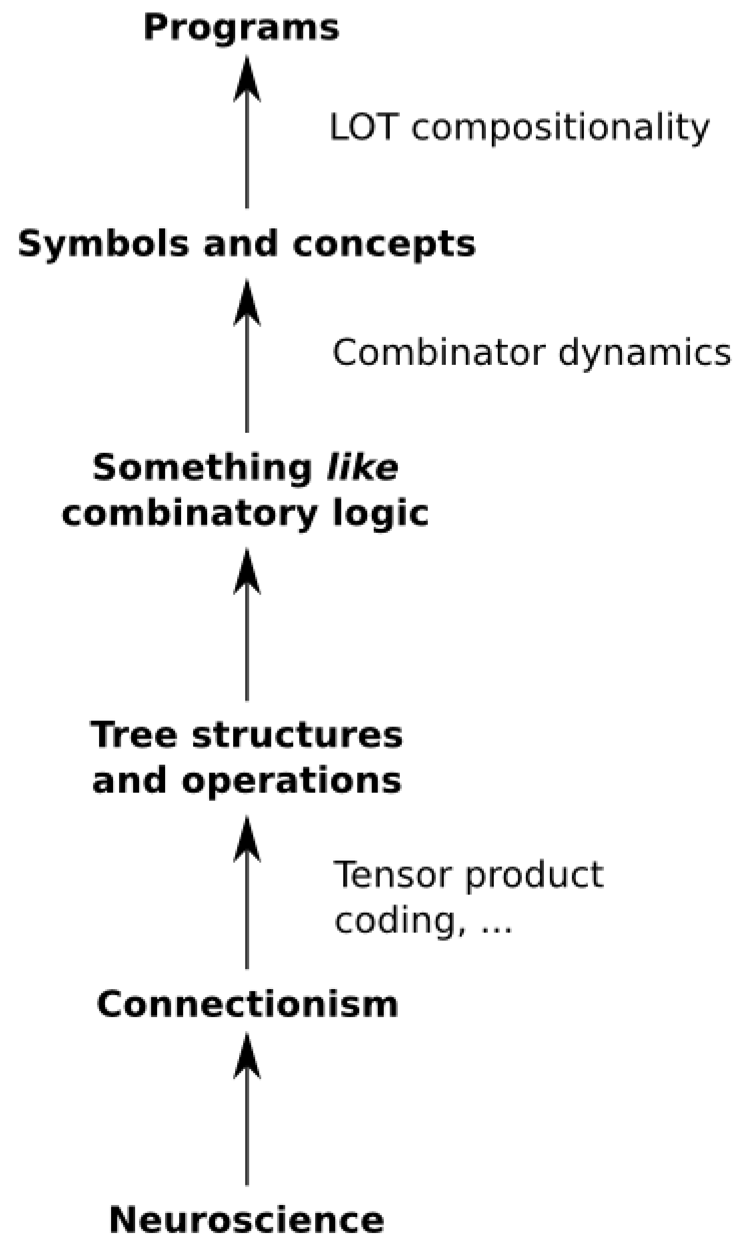
Input = cons(cons(A,B),cons(cons(Aux,V),cons(by,C)))

Figure 1: Recursive tensor product network processing a passive sentence

Legendre, Miyata, Smolensky (1990)

```
if(not empty(s)) {
    w = “one”
    for(s in S) {
        w = wordAfter(w)
    }
}
```





Lessons from CL

- **There is a real sense in which learners can construct almost all logical representations from a primitive, dynamical basis.**

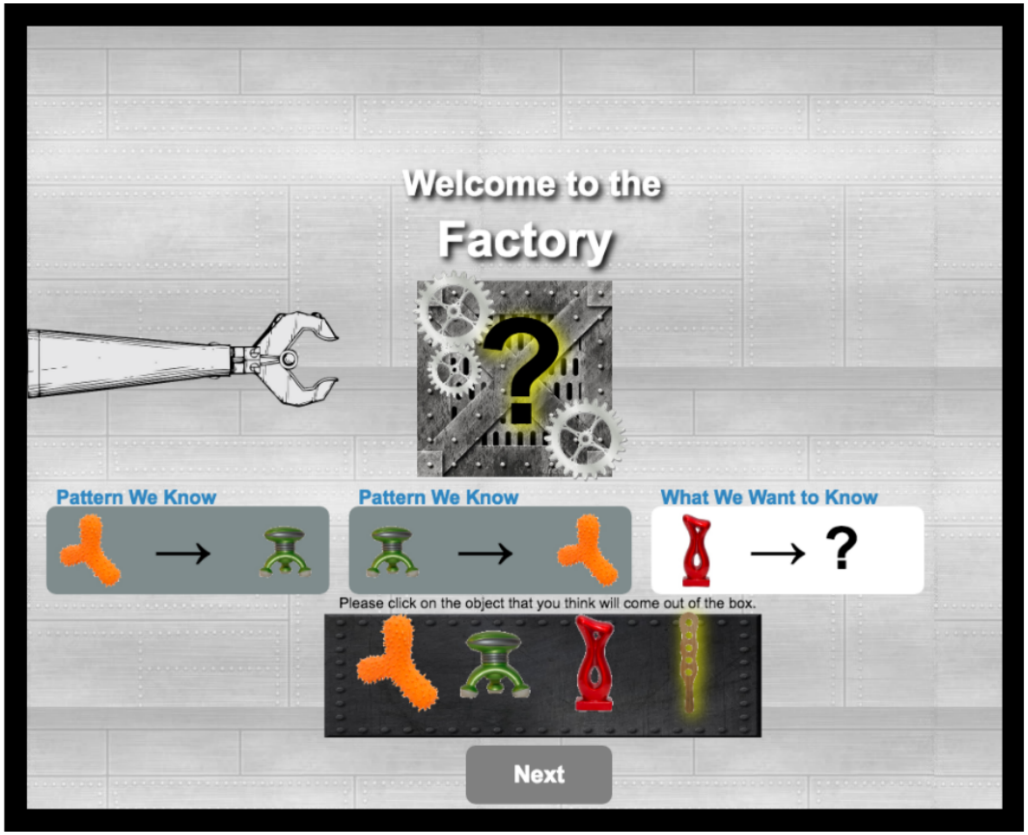
- **Overarching idea:** a language for isomorphisms that is built from pieces with simple, non-cognitive dynamics (no logic, control flow, numbers, etc.)

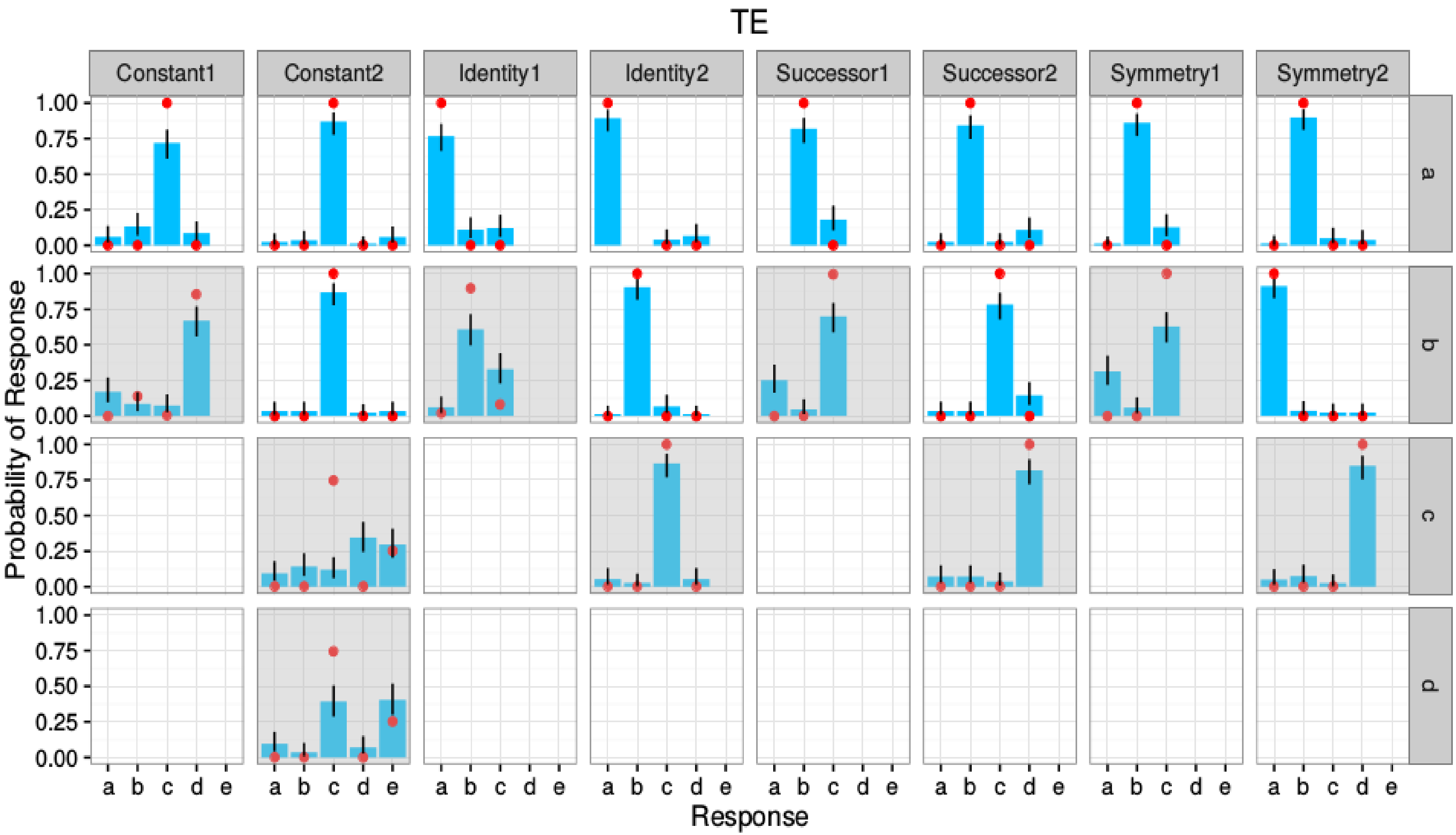
This encoding system is:

dynamical	structured
Turing-complete	compositional
symbolic	variable-free
sub-symbolic	simplicity-driven
deductive	emergent
inductive	parallelizable

Encoding+CL as a psychological theory

$$f(a) = b$$
$$f(b) = c$$
$$f(c) = ?$$





Register & Piantadosi, in prep