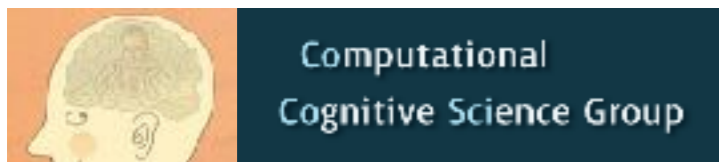


Learning list functions through program induction

Joshua Rule

Learning as Program Induction Workshop
CogSci 2018, 25 July 2018



This talk

- ▶ learning as programming
- ▶ bootstrapping the LOT with term rewriting
- ▶ toward a model of conceptual change

This talk

- ▶ learning as programming
- ▶ bootstrapping the LOT with term rewriting
- ▶ toward a model of conceptual change



Build causal theories from sparse evidence

Navigate complex environments

Recognize objects, reason cross-modally

Tie shoes, make bed, set table

Introspect on beliefs and desires

whisper, shout, sing, joke

Build towers, sandcastles, & Lego cars

Use light switches, door knobs, & smartphones

Talk about dinosaurs, trucks, and fairy tales

Play with others, share, determine ownership

Walk, run, skip, dance, somersault

Use natural language



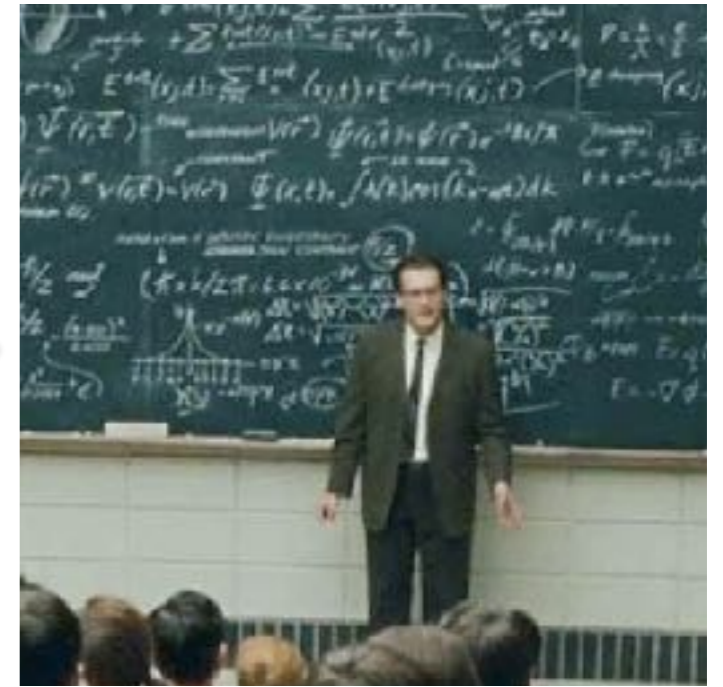
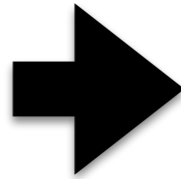
The image shows a physics lecture hall with a professor standing in front of a chalkboard filled with complex mathematical derivations and diagrams. The board is covered in handwritten equations, including:

- Top Left:** Diagrams of magnetic fields and calculations involving μ_0 , I , and L .
- Top Center:** A diagram of a particle's path with labels Y^+ , Y^0 , Y^- , Y^+ , Y^0 , Y^- .
- Top Right:** A graph of $V(x)$ vs x showing a periodic potential, and a diagram of a hydrogen atom with labels $N(\theta)$, N_i , and K .
- Middle Left:** A diagram of a particle's path with labels Y^+ , Y^0 , Y^- , Y^+ , Y^0 , Y^- .
- Middle Center:** A diagram of a particle's path with labels Y^+ , Y^0 , Y^- , Y^+ , Y^0 , Y^- .
- Middle Right:** A diagram of a particle's path with labels Y^+ , Y^0 , Y^- , Y^+ , Y^0 , Y^- .
- Bottom Left:** A diagram of a particle's path with labels Y^+ , Y^0 , Y^- , Y^+ , Y^0 , Y^- .
- Bottom Center:** A diagram of a particle's path with labels Y^+ , Y^0 , Y^- , Y^+ , Y^0 , Y^- .
- Bottom Right:** A diagram of a particle's path with labels Y^+ , Y^0 , Y^- , Y^+ , Y^0 , Y^- .

The professor, a man in a suit, is standing in the center of the lecture hall, facing the students. The students are seated in rows of desks, facing the professor and the chalkboard.



The Puzzle of Learning & Cognitive Development

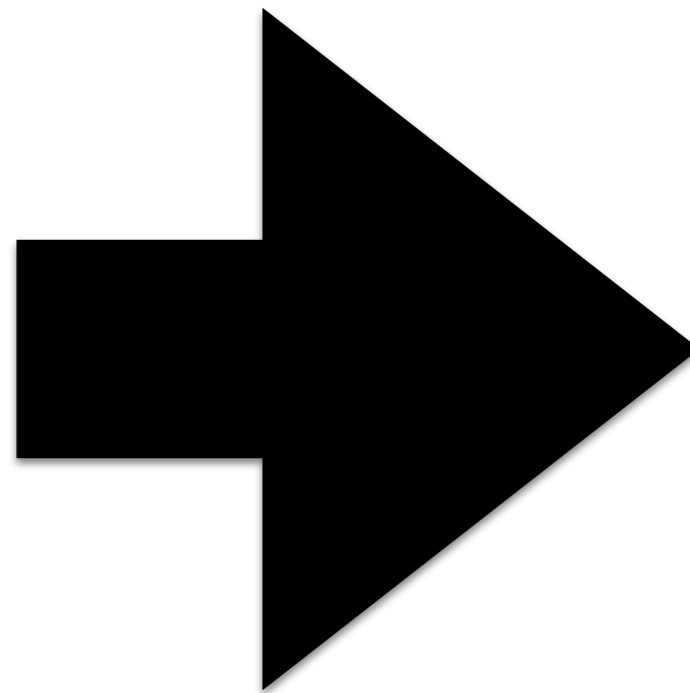


(see Tenenbaum, Kemp, Griffiths, Goodman, 2011; Carey, 2009)

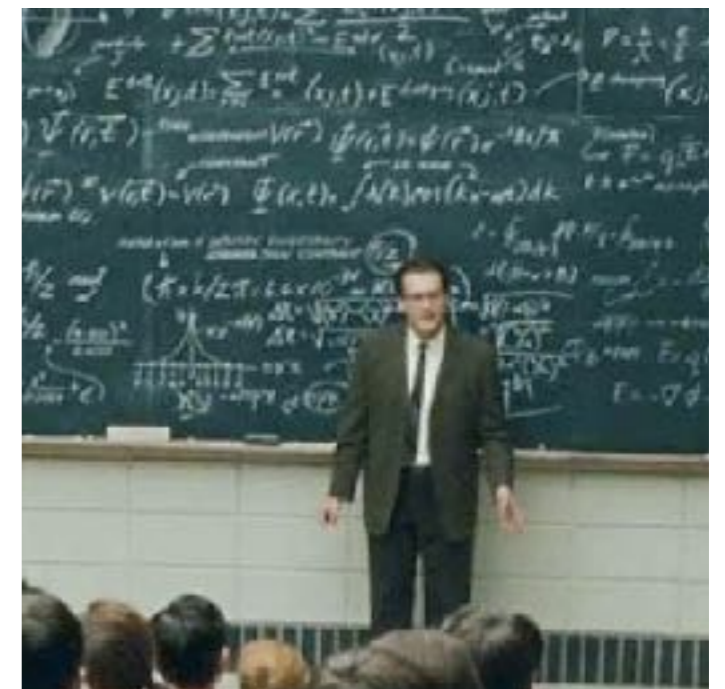
The Puzzle of Learning & Cognitive Development



Initial State



Learning Mechanism

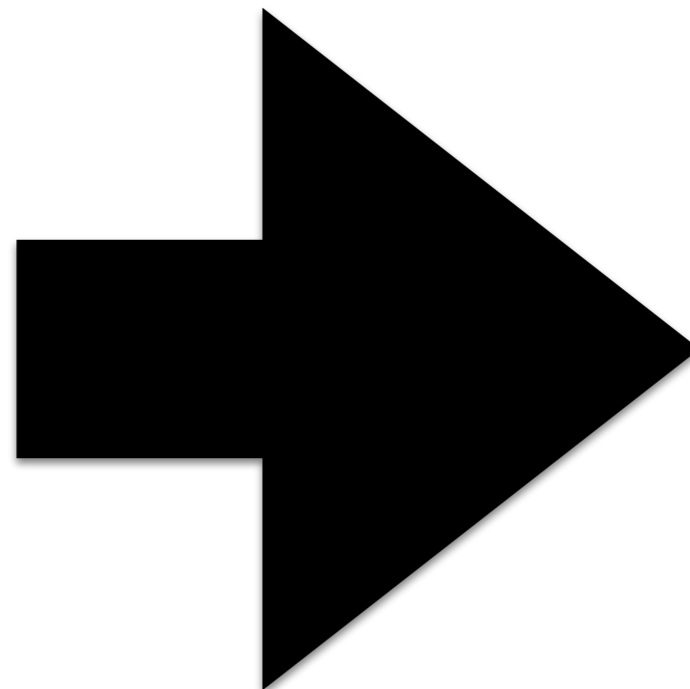


Final State

The Puzzle of Learning & Cognitive Development



Initial State



Learning Mechanism



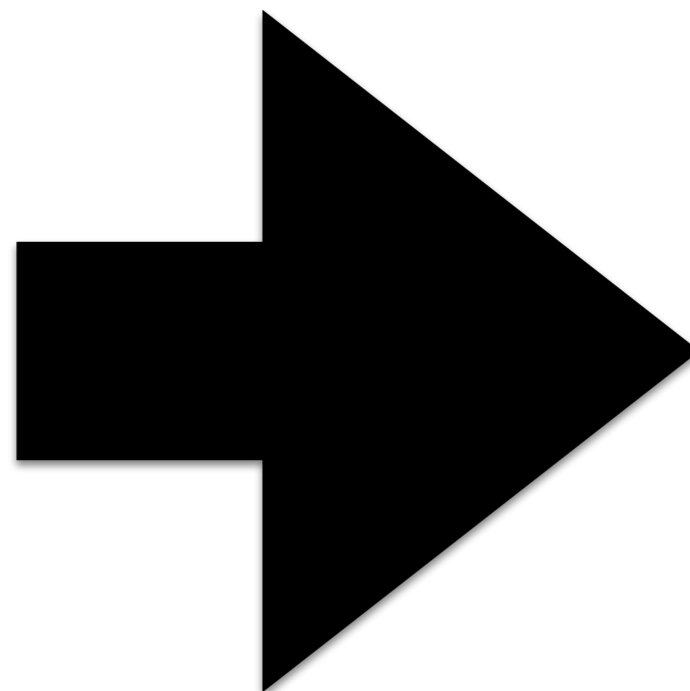
Final State

The Puzzle of Learning & Cognitive Development



Initial State

LOT?



Learning Mechanism

***inductive
bootstrapping?***



Final State

LOT'?

(Carey, 1985, 2009; Carey, Spelke, 1994)
(Fodor, 1975; Turing, 1936; Fodor & Pylyshyn, 1988; Goodman, Tenenbaum, & Gerstenberg, 2015)

Three questions about learning in the LOT

1. How are concepts represented?

2. How are changes proposed?

3. How are proposals assessed?

Three questions about learning in the LOT

1. How are concepts represented?

- **programs in some language**

2. How are changes proposed?

3. How are proposals assessed?

Three questions about learning in the LOT

1. How are concepts represented?

- programs in some language

2. How are changes proposed?

- **small, random syntactic changes to a concept definition**

3. How are proposals assessed?

Three questions about learning in the LOT

1. How are concepts represented?

- programs in some language

2. How are changes proposed?

- small, random syntactic changes to a concept definition

3. How are proposals assessed?

- **accuracy & description length (& sometimes efficiency)**

Three questions about learning in the LOT

1. How are concepts represented?

- programs in some language

2. How are changes proposed?

- small, random syntactic changes to a concept definition

3. How are proposals assessed?

- accuracy & description length (& sometimes efficiency)

This talk

- ▶ learning as programming
- ▶ bootstrapping the LOT with term rewriting
- ▶ toward a model of conceptual change

A nagging problem

$$\mathbf{LOT} + \mathbf{bootstrapping} = \mathbf{LOT'}$$

A nagging problem

LOT + bootstrapping = LOT'

modeled as

***Prog. Lang.
&
Library + stochastic
search = Prog. Lang.
&
Library'***

Three questions about learning in the LOT

1. How are concepts represented?

- **programs in some language**

2. How are changes proposed?

- small, random syntactic changes to a concept definition

3. How are proposals assessed?

- accuracy & description length (& sometimes efficiency)

Three questions about learning in the LOT

1. How are concepts represented?

- **programs in some *fixed* language**

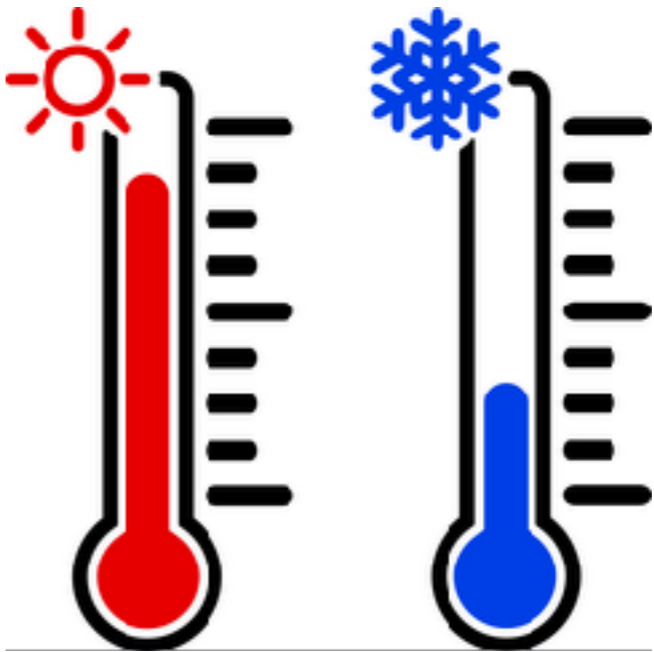
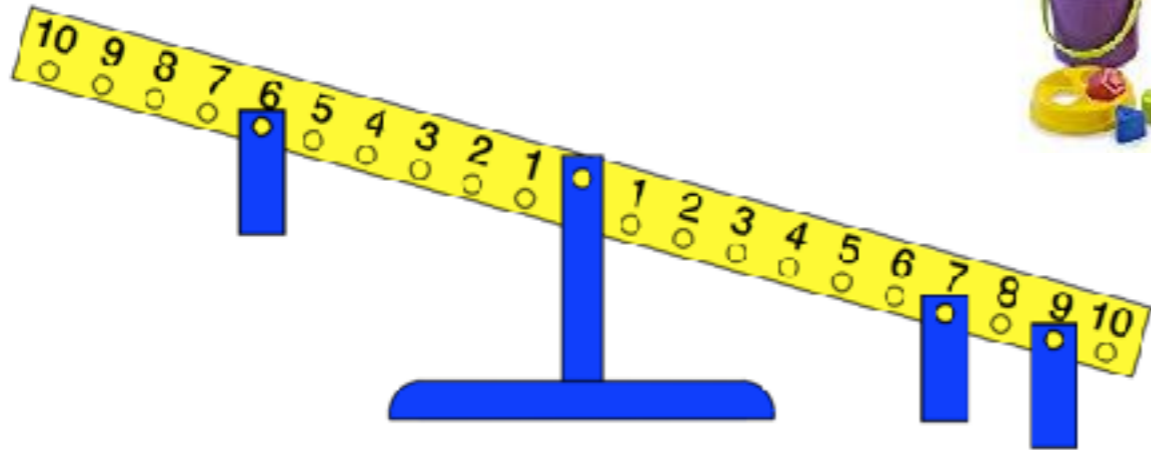
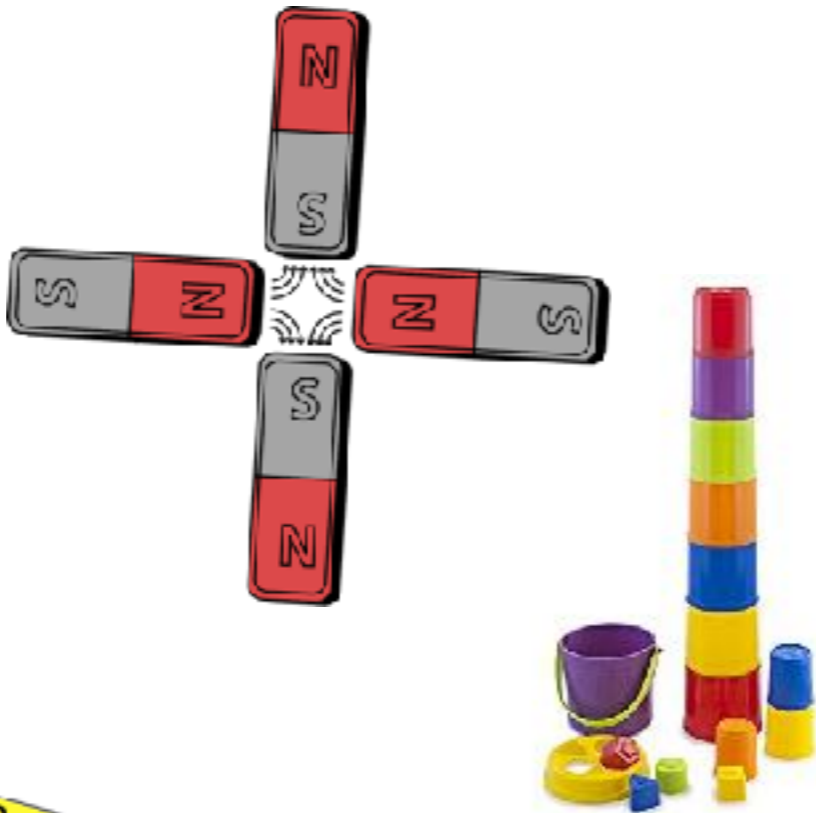
2. How are changes proposed?

- small, random syntactic changes to a concept definition

3. How are proposals assessed?

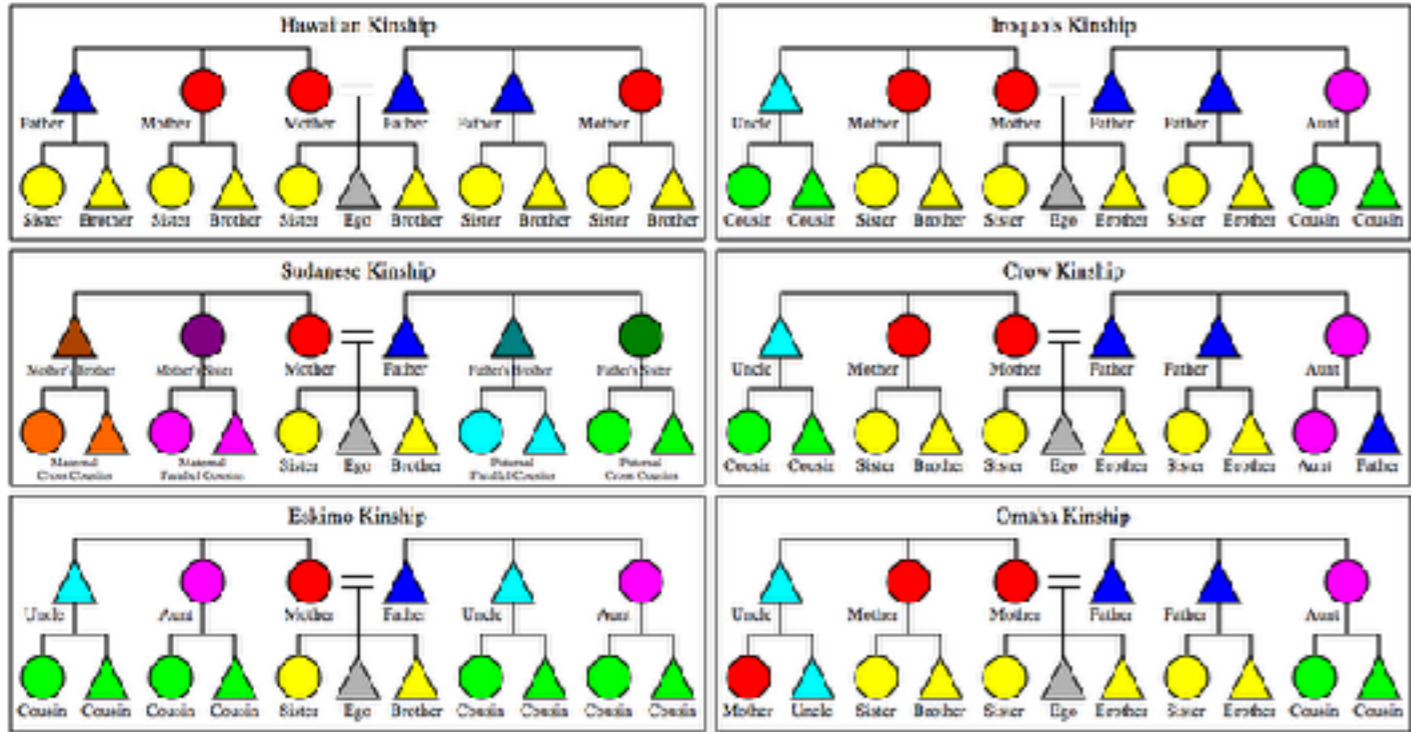
- accuracy & description length (& sometimes efficiency)

		pollen ♂	
		B	b
pistil ♀	B	BB	Bb
	b	Bb	bb



AABABA
ABAAAB
AABAAB
ABAAAB
ABAABA

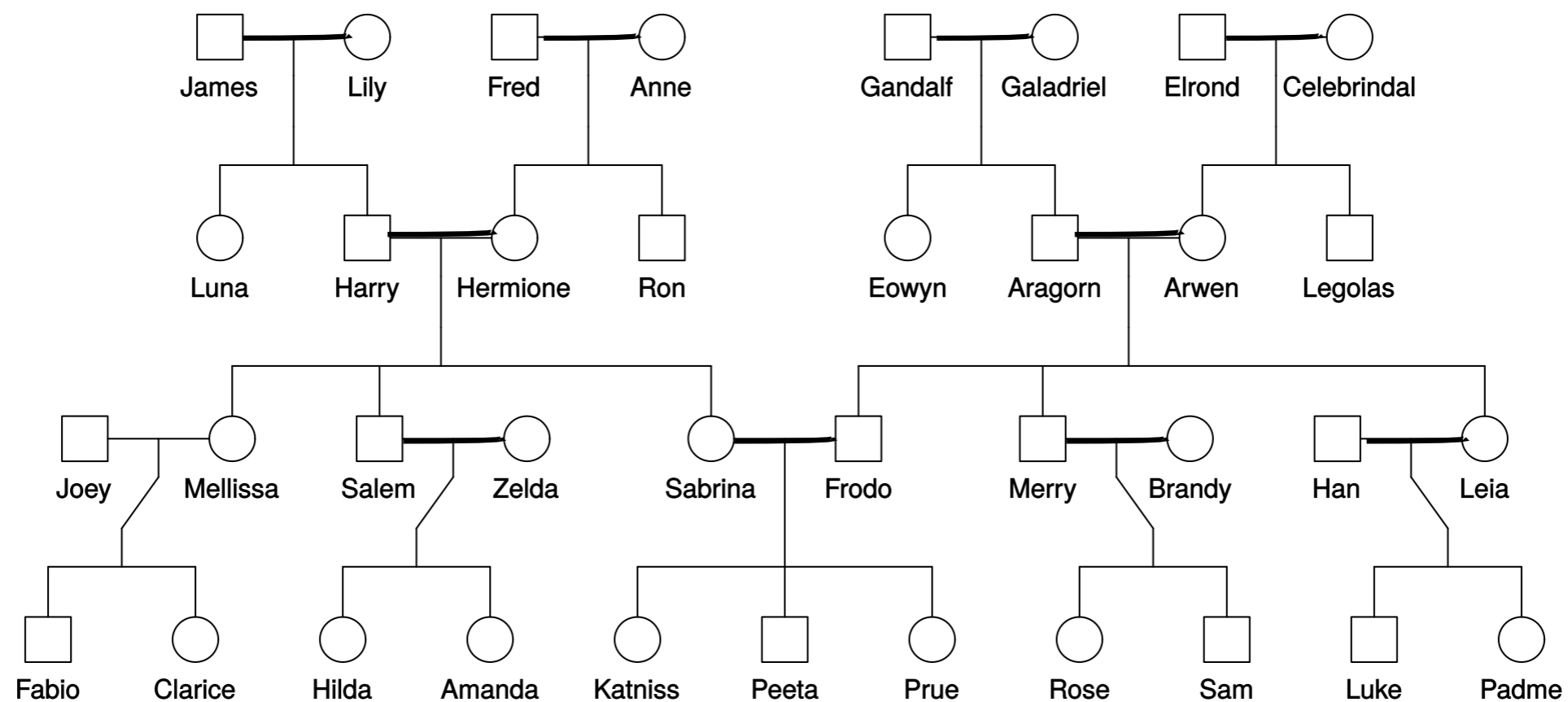
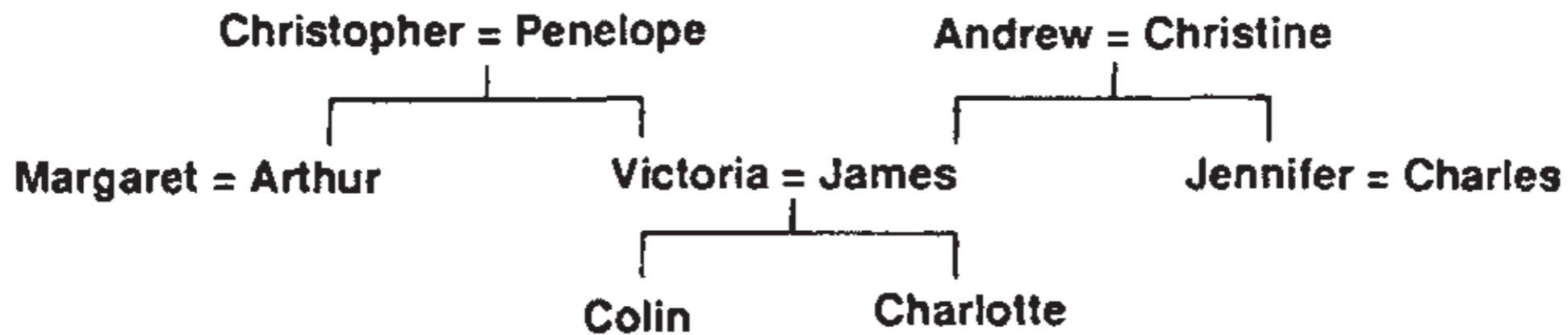
six-hundred-forty-seven-
thousand-nine-hundred-
sixteen



Kinship is a great space for studying conceptual change

Definite Gender	boy, girl, man, woman
Generic Gender	male, female
Definite Nuclear	brother, sister, mother, father, husband, wife, son, daughter
Generic Nuclear	sibling, spouse, parent, child
Definite Extended	aunt, uncle, nephew, niece, grandmother, grandfather, granddaughter, grandson, grandnephew, grandniece
Generic Extended	grandparent, grandchild, cousin
Structurally Recursive	great-aunt, great-uncle, great-grandfather, great-grandmother, great-grandparent, great-granddaughter, great-grandson, great-grandchild, great-great-, great-great-great, ...
Linearly Recursive	ancestor, descendant
Nonlinearly Recursive	relative, blood relative, in-law, m^{th} cousin n^{th} removed, step-relations

typical kinship data



potential kinship data

```
⋮  
true → husband(Christopher, Penelope)  
true → cousin(Rose, Luke)  
true → uncle(Arthur, Colin)  
true → brother(Arthur, Victoria)  
true → man(Arthur)  
true → girl(Charlotte)  
true → dad(Joey, Clarice)  
true → brother(Sam, Rose)  
true → great-uncle(Ron, Katniss)  
true → sister(Katniss, Prue)  
true → sister(Prue, Katniss)  
true → husband(James, Victoria)  
true → sister(Rose, Sam)  
false → sister(Sam, Rose)  
⋮
```

potential kinship grammar

potential kinship grammar

male

female

spouse

parent

potential kinship grammar

male(Aragorn)

female(Arwen)

spouse(Aragorn, Arwen)

parent(Elrond, Arwen)

potential kinship grammar

male(Aragorn)

female(Arwen)

spouse(Aragorn, Arwen)

parent(Elrond, Arwen)

:

and(male(x), spouse(x, y)) → husband(x, y)

and(female(x), spouse(x, y)) → wife(y, y)

and(female(x), sibling(x, y)) → sister(x, y)

and(male(x), sibling(x, y)) → brother(x, y)

and(male(x), parent(x, y)) → father(x, y)

and(female(x), parent(x, y)) → mother(x, y)

and(male(x), parent(y, x)) → son(x, y)

and(female(x), parent(y, x)) → daughter(x, y)

and(parent(z,y), parent(z,x)) → sibling(x, y)

:

parent(x, y) → ancestor(x, y)

and(parent(x, y), ancestor(y, z)) → ancestor(x, y)

:

and(ancestor(x, y), ancestor(x, z)) → blood_relative(y, z)

Three questions about learning in the LOT

1. How are concepts represented?

- **programs in some *fixed* language**

2. How are changes proposed?

- small, random syntactic changes to a concept definition

3. How are proposals assessed?

- accuracy & description length (& sometimes efficiency)

Three questions about learning in the LOT

1. How are concepts represented?

- **programs in some *adaptive* language**

2. How are changes proposed?

- small, random syntactic changes to a concept definition

3. How are proposals assessed?

- accuracy & description length (& sometimes efficiency)

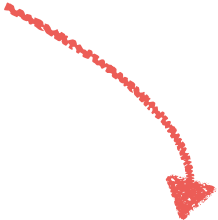
Term Rewriting Systems

$$TRS = (\Sigma, R)$$

Term Rewriting Systems

Signature:

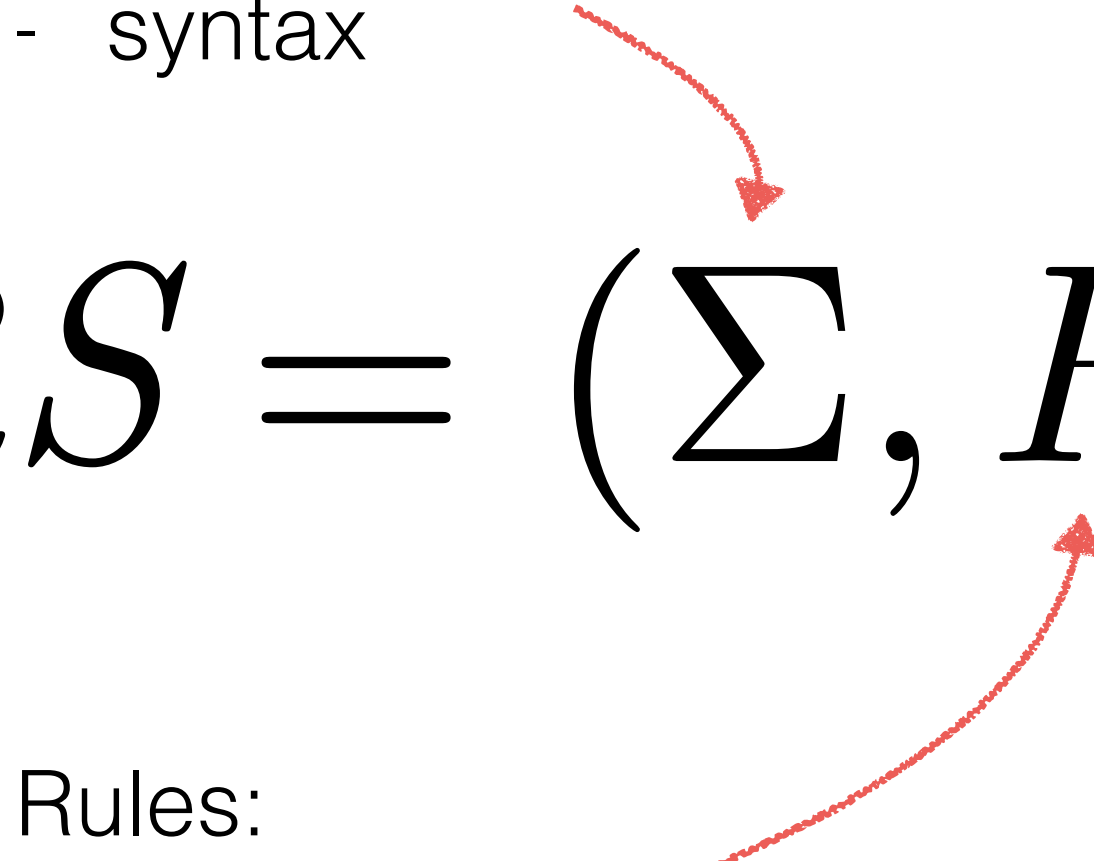
- a set of primitives
- what things exist
- syntax


$$TRS = (\Sigma, R)$$

Term Rewriting Systems

Signature:

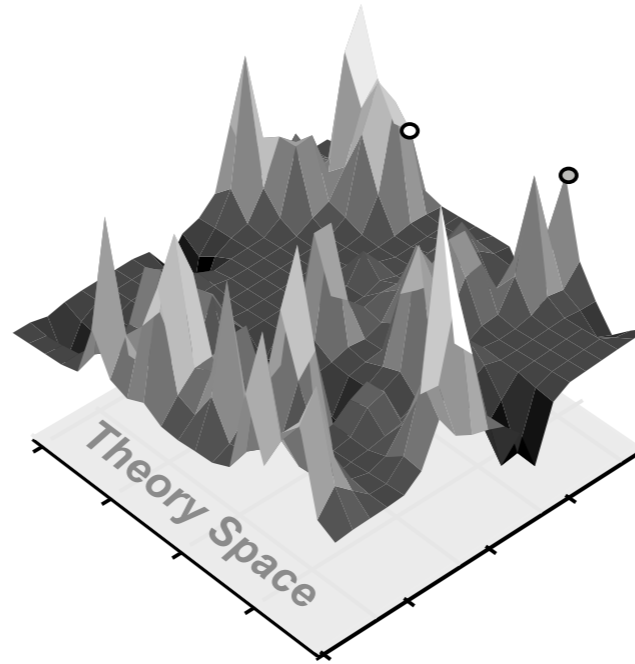
- a set of primitives
- what things exist
- syntax

$$TRS = (\Sigma, R)$$
A diagram consisting of two red arrows. The first arrow starts at the word 'syntax' in the list under 'Signature' and points to the symbol Σ in the TRS equation. The second arrow starts at the word 'rules' in the list under 'Rules' and points to the symbol R in the TRS equation.

Rules:

- a list of rewrite rules
- how things behave
- semantics

Stochastic search over TRSs



- remove a symbol s from Σ_{i-1} and all rules involving s from R_{i-1}
- add a symbol s to Σ_i
- generate a new rule r and add it to R_i
- remove a rule r from R_{i-1}

One solution: models LOTs as Term Rewriting Systems (TRSs)

$$\mathbf{LOT} \quad + \quad \mathbf{bootstrapping} \quad = \quad \mathbf{LOT'}$$

modeled as

$$\mathbf{TRS} \quad + \quad \mathbf{stochastic} \\ \mathbf{search} \quad = \quad \mathbf{TRS'}$$

This talk

- ▶ learning as programming
- ▶ bootstrapping the LOT with term rewriting
- ▶ toward a model of conceptual change

Learning list concepts through program induction

Joshua Rule,^{1*} Eric Schulz,^{2*} Steven T. Piantadosi,³ & Joshua B. Tenenbaum¹

¹Department of Brain and Cognitive Sciences, Massachusetts Institute of Technology

²Department of Psychology, Harvard University

³Department of Brain and Cognitive Sciences, University of Rochester

*Contributed equally

Abstract

Humans create complex systems of interrelated concepts like mathematics and natural language. Previous work suggests

a fixed semantics, often based on combinatory logic (CL; Dechter et al., 2013; Piantadosi, 2017), λ -calculus (LC; Piantadosi, Tenenbaum, & Goodman, 2012), or first-order logic



Josh Tenenbaum

Steve Piantadosi

Eric Schulz



Existing models of concept learning as program induction have had less success at explaining larger-scale aspects of human learning and conceptual development. These approaches typically learn by stochastically searching through a (possibly infinite) space of possible programs to find good candidates. To help constrain this search, they usually make two limiting assumptions. First, they focus on learning one concept

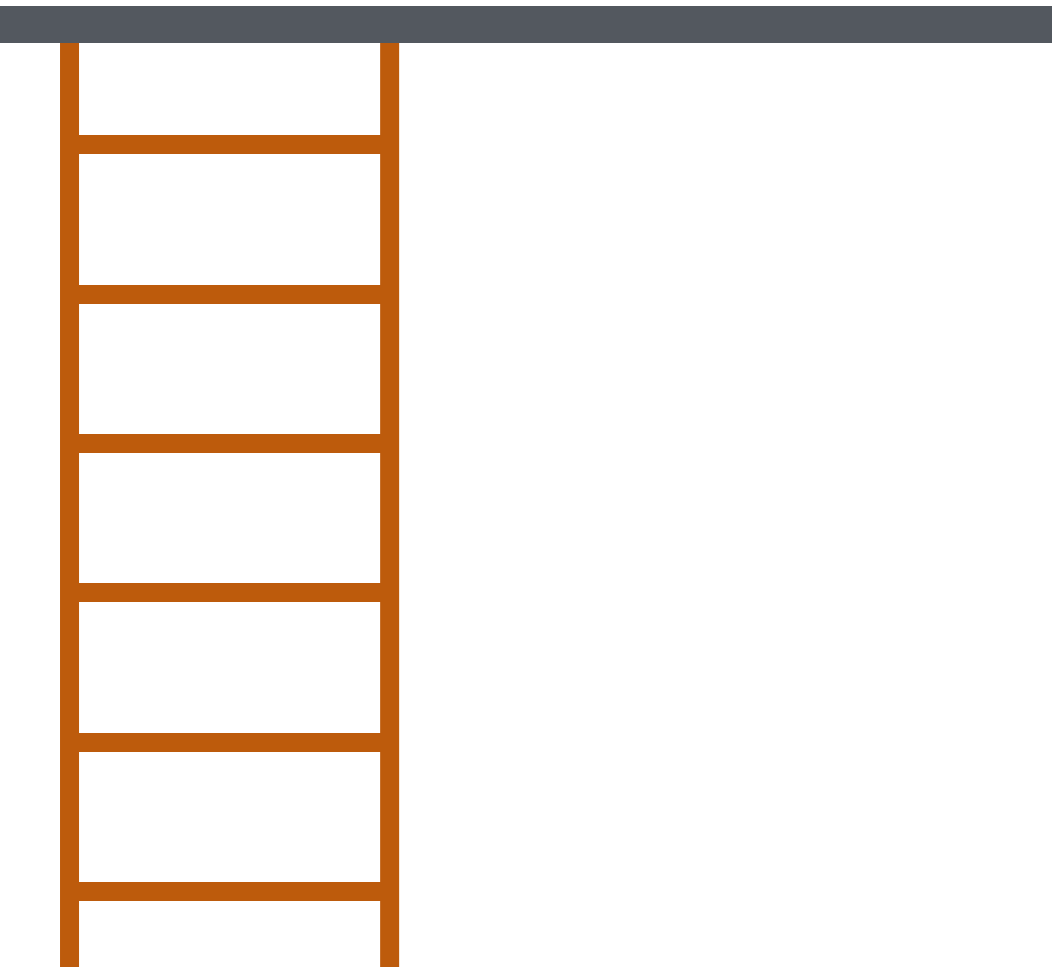
these limitations. The first contribution is to introduce and Concept learning paradigms have often focused on individual concepts rather than conceptual systems. This game that lends itself well to learning concepts and which participants predict. Presenting a concept, transforming it into another. Using this find these concepts are learned more easily and that hard concepts are learned more easily when preceded by a bootstrapping compositional curriculum.

The second contribution is to introduce Term Rewriting Systems (TRSs) as a model for conceptual representations. TRSs, like CL and LC, were originally developed as an abstract model of computation. Two features of TRSs make them particularly suitable for concept learning: 1) unlike CL or LC, the set of primitives can be easily revised; and 2) the meaning of concepts is entirely determined by a set of revisable rewrite rules describing how terms execute over time.

The third contribution is to introduce the idea of using a meta-language to guide learning. We propose a computational model of concept learning in which hypotheses represent not merely different definitions of a concept, but

(Rule, Schulz, Piantadosi, Tenenbaum, 2018)

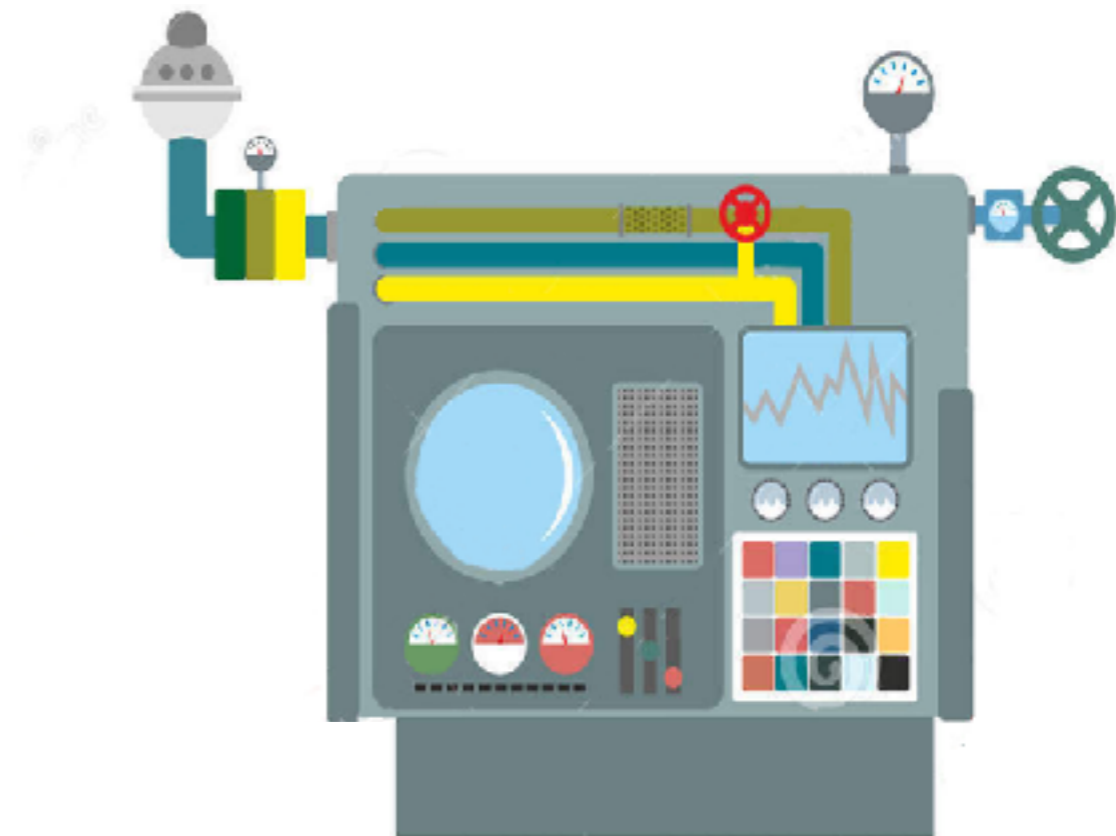
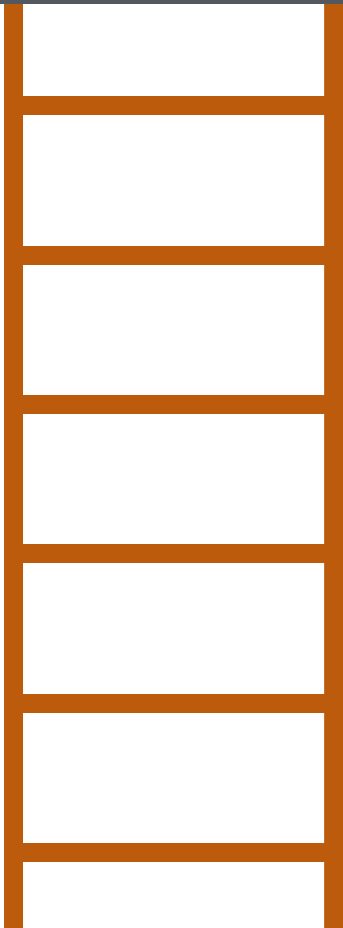
Martha's Magical Machines



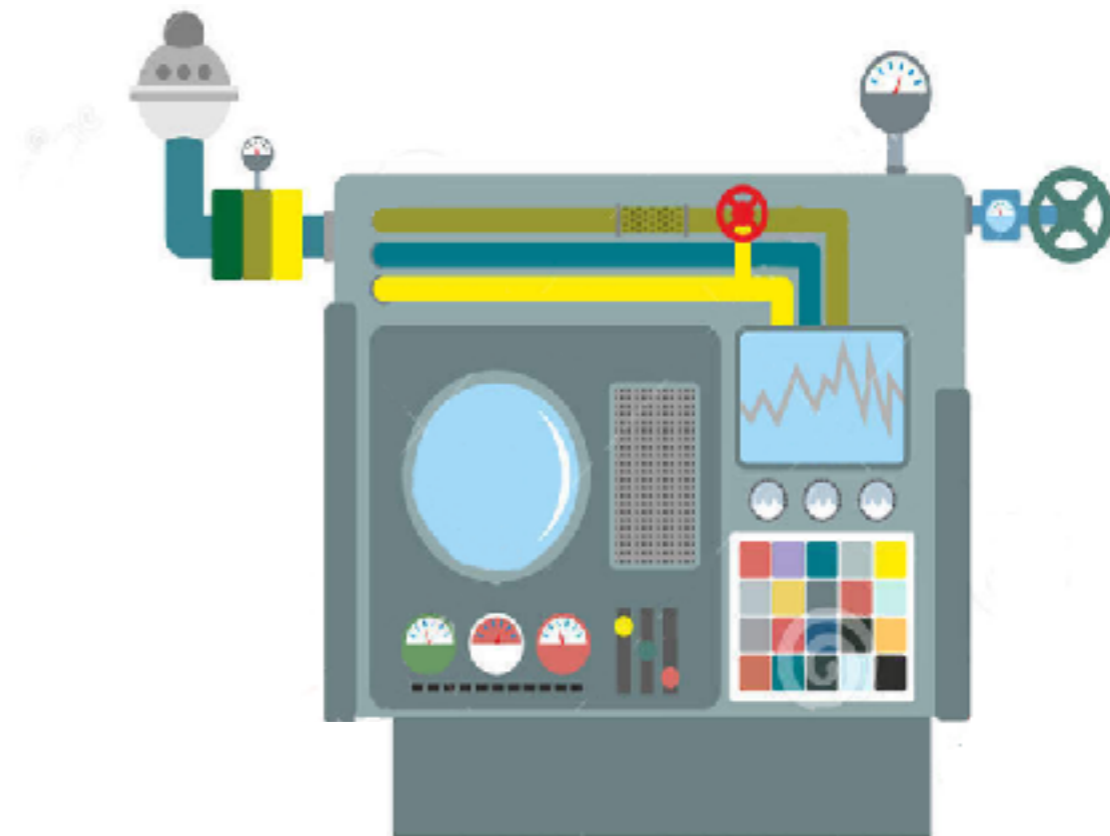
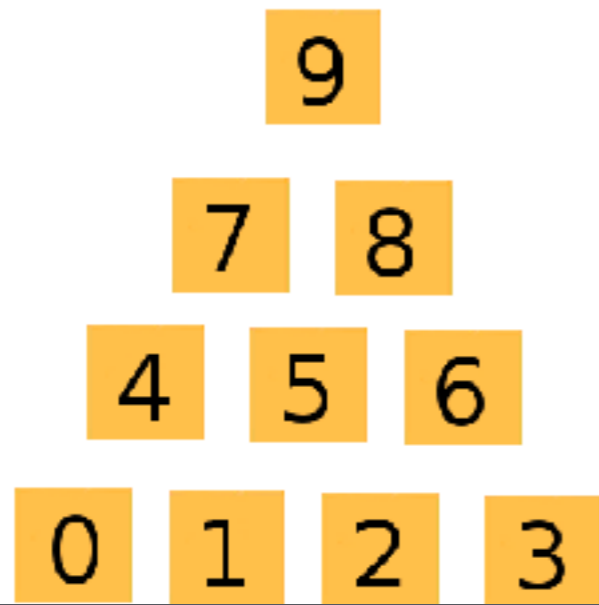
Martha's Magical Machines



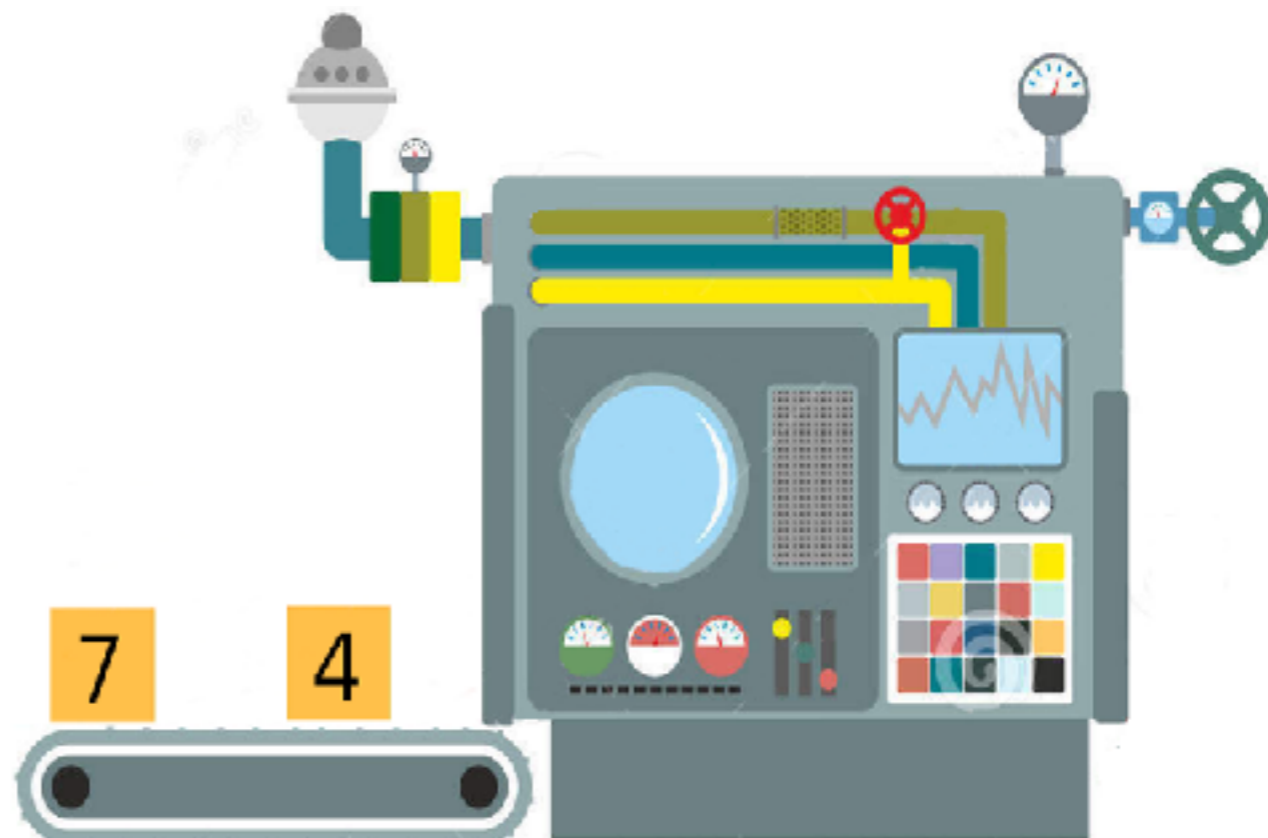
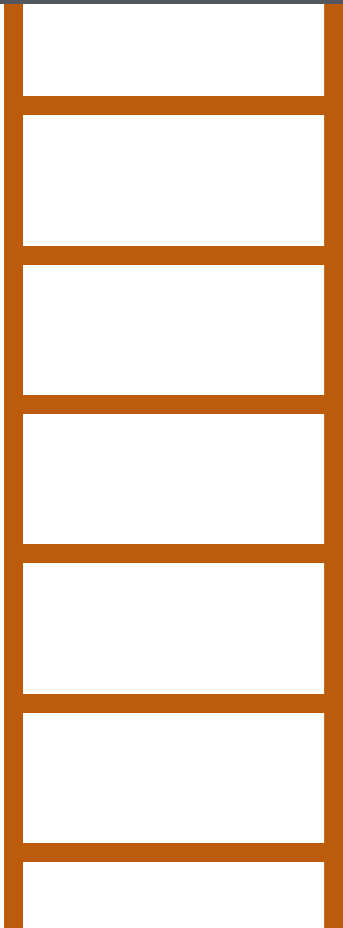
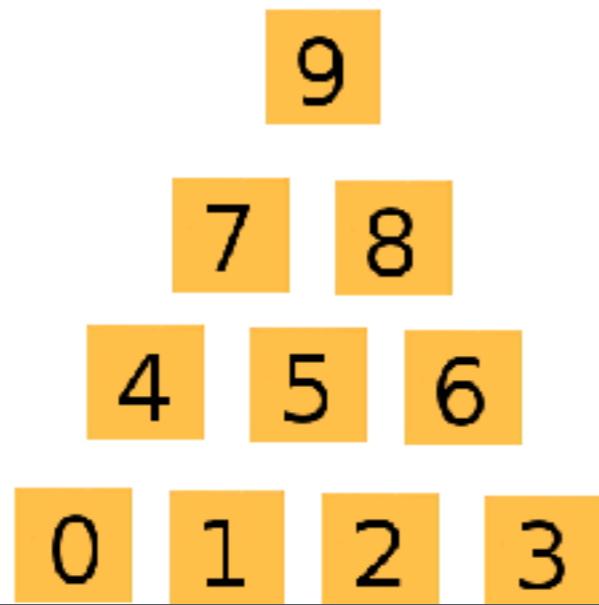
Martha's Magical Machines



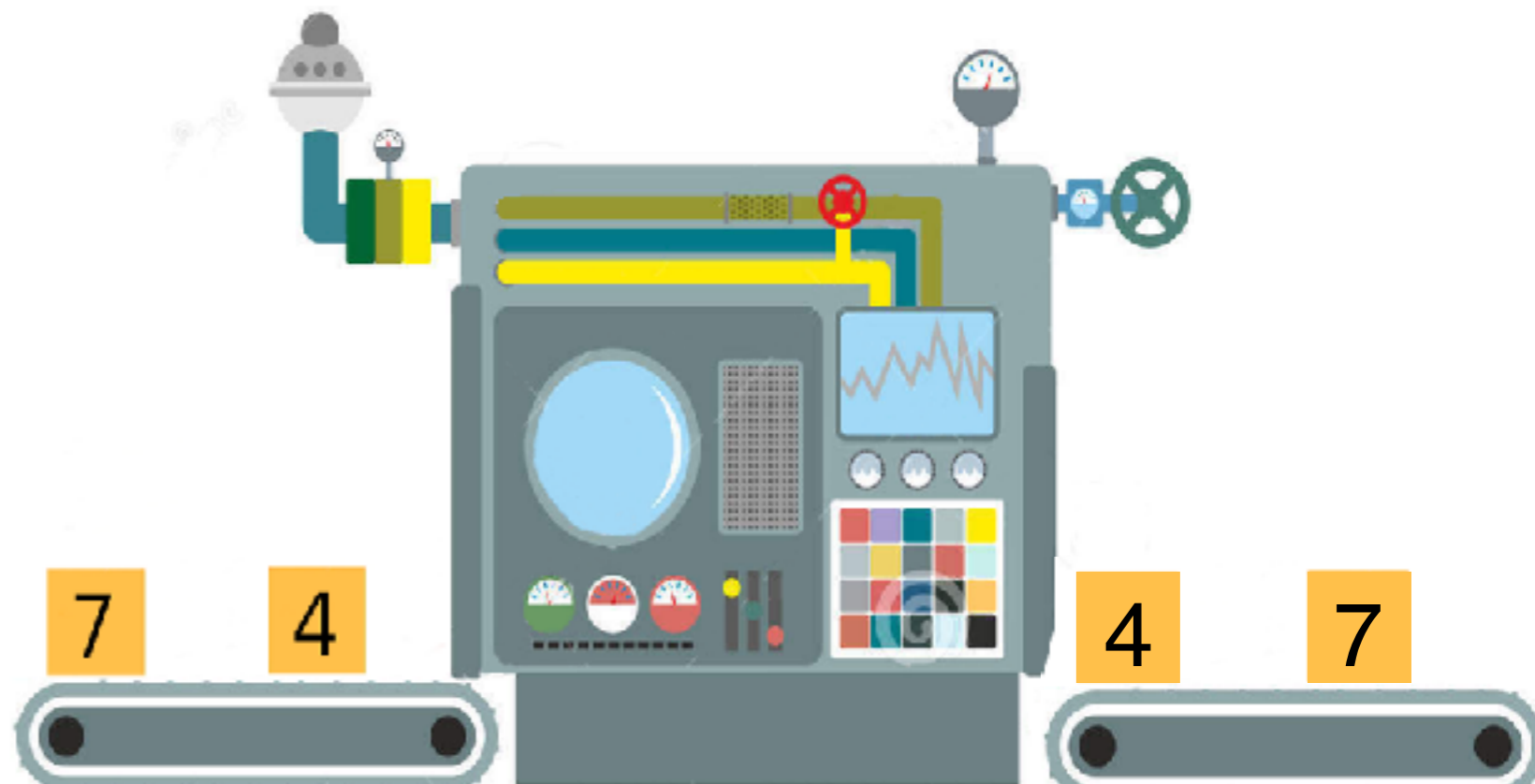
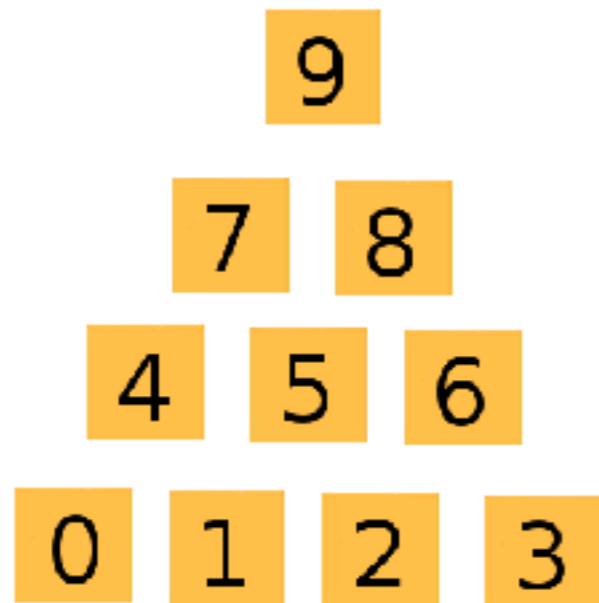
Martha's Magical Machines



Martha's Magical Machines



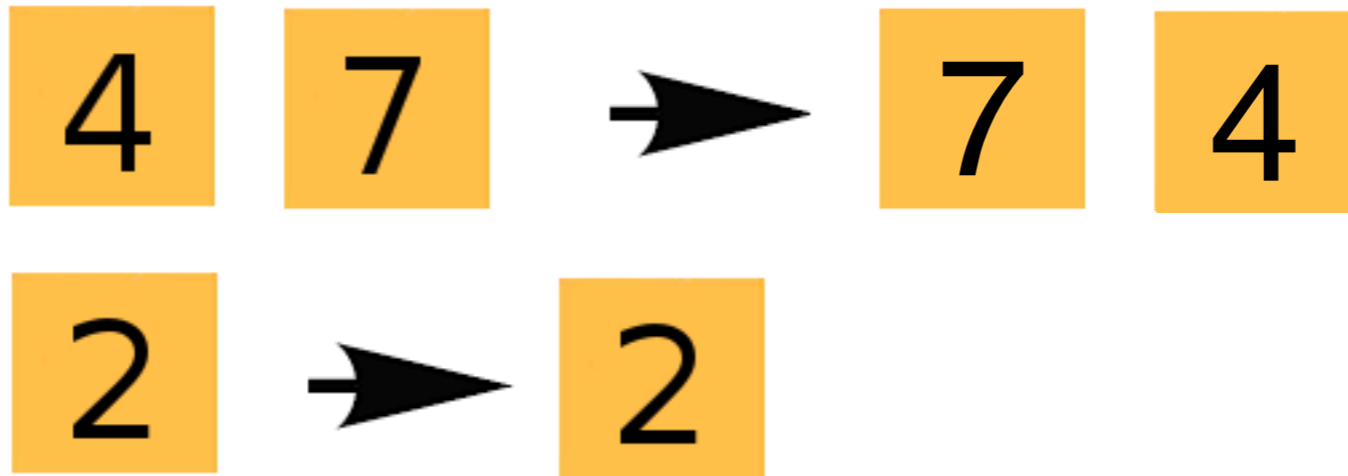
Martha's Magical Machines



Martha's Magical Machines



Martha's Magical Machines



Martha's Magical Machines



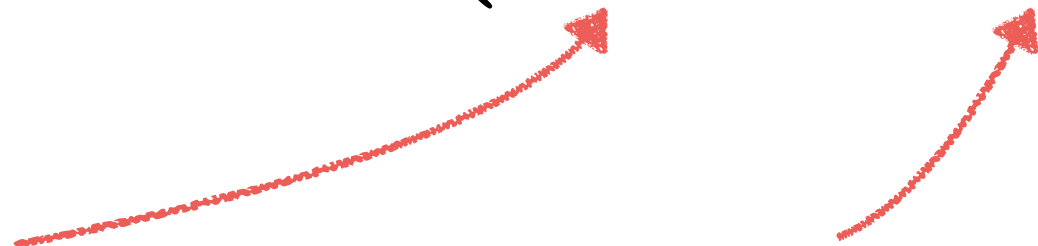
Stochastic search over TRSs

```
def search(data, h0, N=1500, n_top=10, n_steps=50, confidence=2/3):  
    dataset = []  
    h, score = h0, score(h0)  
    hs = heap([(h, score)])  
    for (i, o) in data:  
        for _ in range(N):  
            h_next = propose(h)  
            score_next = score(h_next)  
            h, score = metropolis(h, score, h_next, score_next)  
            hs.insert((h, score))  
        best_hs = hs.take_top(n_top)  
        o_hat = most_likely_output(i, n_steps, best_hs)  
        data.append((i, o))  
        N *= (confidence if o_hat == o else 1/confidence)  
    return hs
```

Model Primitives

Name & Input/Output Pair	Description
$0, 1, 2$	constant natural numbers
$[]$	the empty list
$\text{succ}(0)$	the successor of x
$\text{cons}(1, [2, 3]) = [1, 2, 3]$	prepend x to y
$\text{sum}([1, 2, 3]) = [6]$	sum x
$\text{add}(3, [1, 2, 3]) = [4, 5, 6]$	add x to the elements of y
$\text{insert}(4, [3, 5]) = [3, 4, 5]$	insert x into y in sorted order
$\text{remove}(1, [6, 1, 4]) = [6, 4]$	remove every x in y
$\text{count}(7, [7, 1, 7]) = [2]$	count every x in y
$\text{even}(5) = \text{false}$	true if x is even else false
$\text{greater}(8, 2) = \text{true}$	true if $x > y$ else false
$\text{if}(\text{true}, [7], [2, 5]) = [7]$	if x then y else z
$\text{nth}(3, [9, 5, 8]) = [8]$	the x^{th} element of y

Model Primitives

$$h_0 = (\Sigma^*_0, R_0)$$


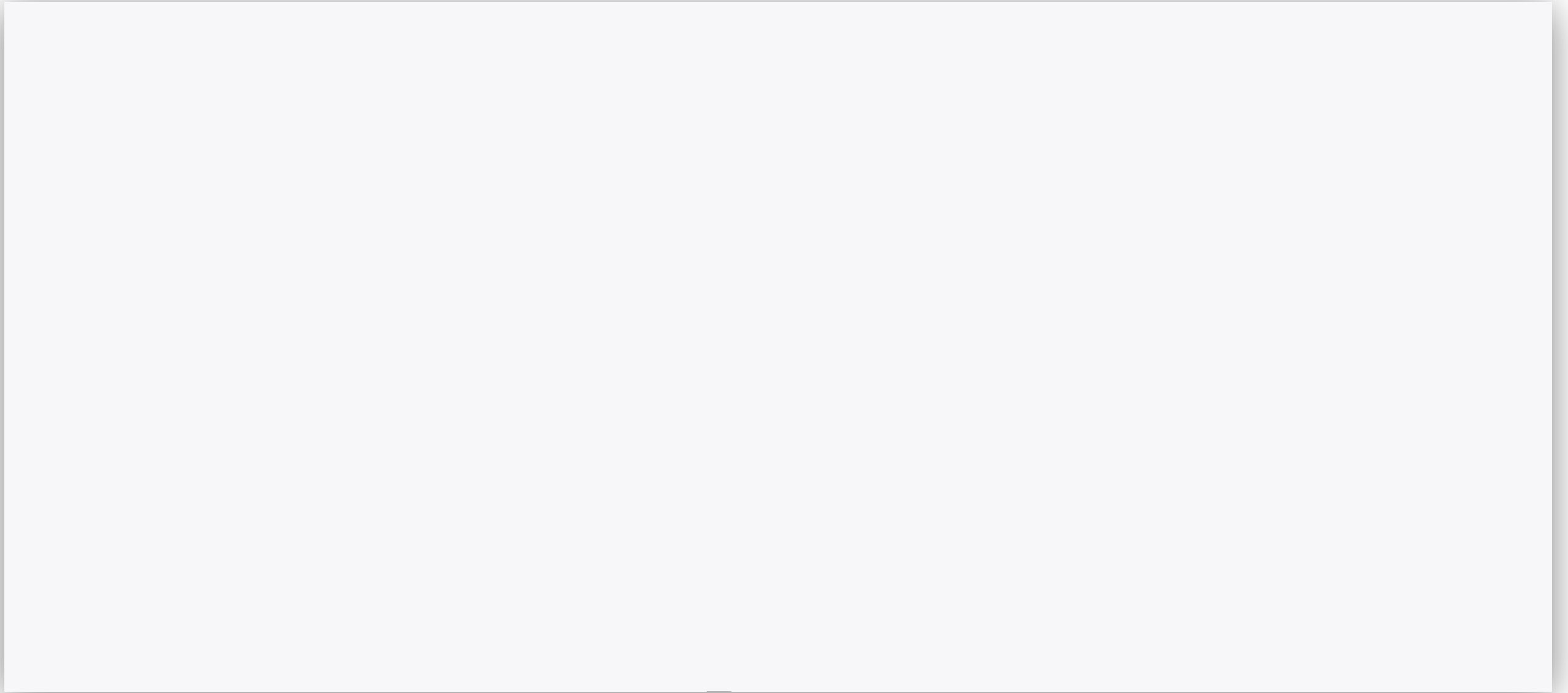
Name & Input/Output Pair	Description
0, 1, 2	constant natural numbers
[]	the empty list
succ(0)	the successor of x
cons(1, [2, 3]) = [1, 2, 3]	prepend x to y
sum([1, 2, 3]) = [6]	sum x
add(3, [1, 2, 3]) = [4, 5, 6]	add x to the elements of y
insert(4, [3, 5]) = [3, 4, 5]	insert x into y in sorted order
remove(1, [6, 1, 4]) = [6, 4]	remove every x in y
count(7, [7, 1, 7]) = [2]	count every x in y
even(5) = false	true if x is even else false
greater(8, 2) = true	true if $x > y$ else false
if(true, [7], [2, 5]) = [7]	if x then y else z
nth(3, [9, 5, 8]) = [8]	the x^{th} element of y

*plus the target concept

Experiment 1

- **149 participants** (61 female, mean age=36.93, SD=12.20)
- **5 concepts/participant (out of 12)**
- **10 trials/concept**

Experiment 1



Experiment 1

```
# const xs: return 3  
# Example: const([1,2,4]) = [3]  
const(x_) = 3;
```

Experiment 1

```
# const xs: return 3  
# Example: const([1,2,4]) = [3]  
const(x_) = 3;
```

```
# index-in-head xs: return the headth element of the xs  
# Example: index_in_head([2,3]) = [3]  
index-in-head(cons(0 y_)) = 0  
index-in-head(cons(succ(x_) y_)) = nth(x_ y_);
```

Experiment 1

```
# const xs: return 3
# Example: const([1,2,4]) = [3]
const(x_) = 3;

# total xs: sum all the elements of xs
# Example: total([1,2,3]) = [6]
total(x_) = sum(x_);

# increment xs: add 1 to each element of xs
# Example: increment([1,2]) = [2,3]
increment(x_) = add(1 x_);

# head xs: return the first element of xs
# Example: head([2,3,1]) = [2]
head(cons(x_ y_)) = x_;

# length xs: compute the length of xs
# Example: length([2,3,1]) = [3]
length([]) = 0;
length(cons(x_ y_)) = succ(length(y_));

# sort xs: sort xs
# Example: sort([3,1]) = [1,3]
sort([]) = [];
sort(cons(x_ y_)) = insert(x_ sort(y_));

# deduplicate xs: remove all duplicates from xs
# Example: deduplicate([2,1,2,2,1]) = [2,1]
deduplicate([]) = [];
deduplicate(cons(x_ y_)) =
  cons(x_ deduplicate(remove(x_ y_)))
```

```
# cumsum xs: cumulatively sum the elements of xs
# Example: cumsum([2,3,1]) = [2,5,6]
cumsum([]) = [];
cumsum(cons(x_ y_)) = cons(x_ cumsum(add(x_ y_)))

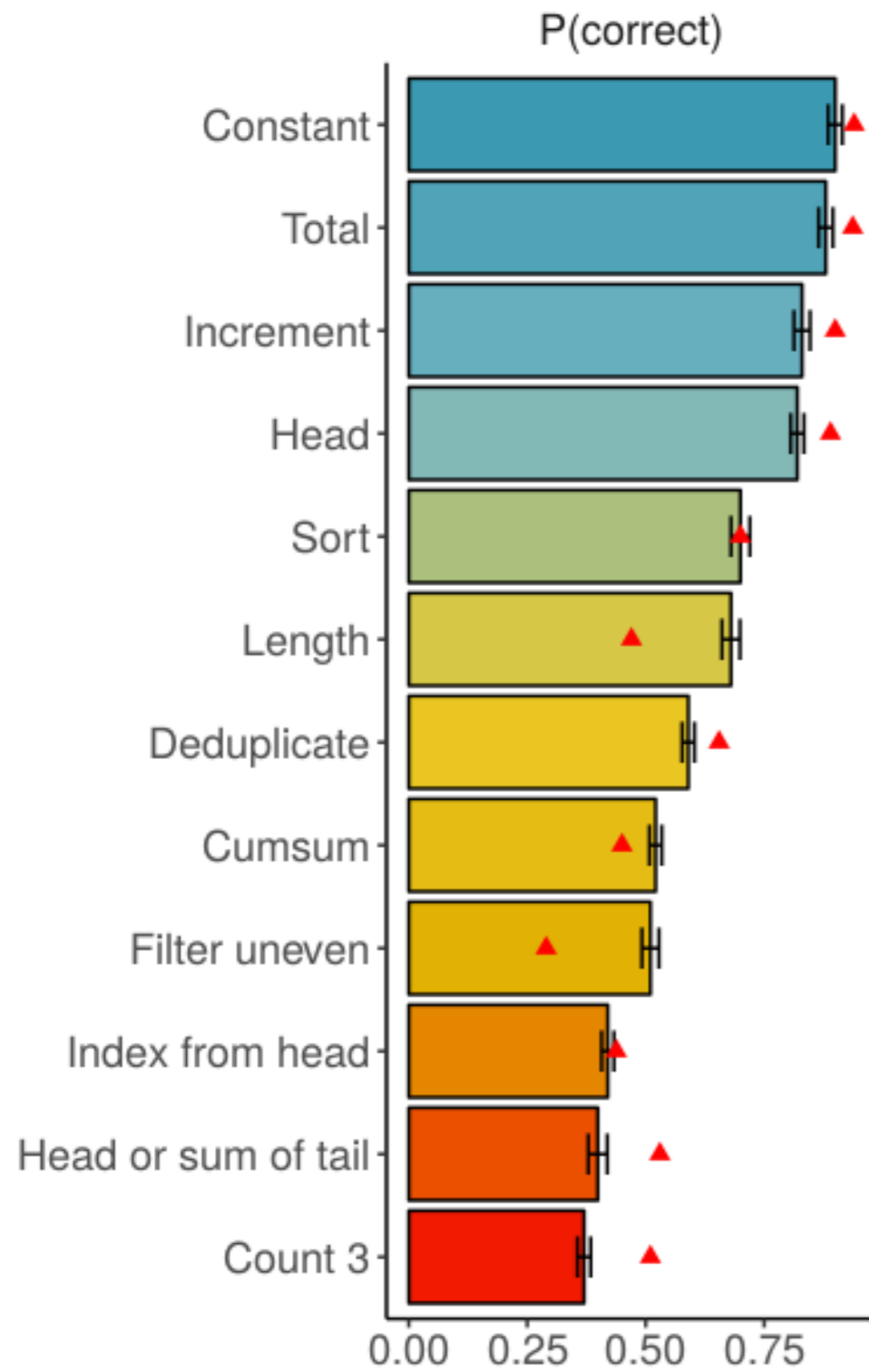
# filter_odd xs: remove the odd numbers from xs
# Example: filter_odd([2,3,1,4]) = [2,4]
filter_odd([]) = [];
filter_odd(cons(x_ y_)) =
  if(even?(x_) cons(x_ filter_odd(y_)) filter_odd(y_));

# index-in-head xs: return the headth element of the xs
# Example: index_in_head([2,3]) = [3]
index-in-head(cons(0 y_)) = 0
index-in-head(cons(succ(x_) y_)) = nth(x_ y_);

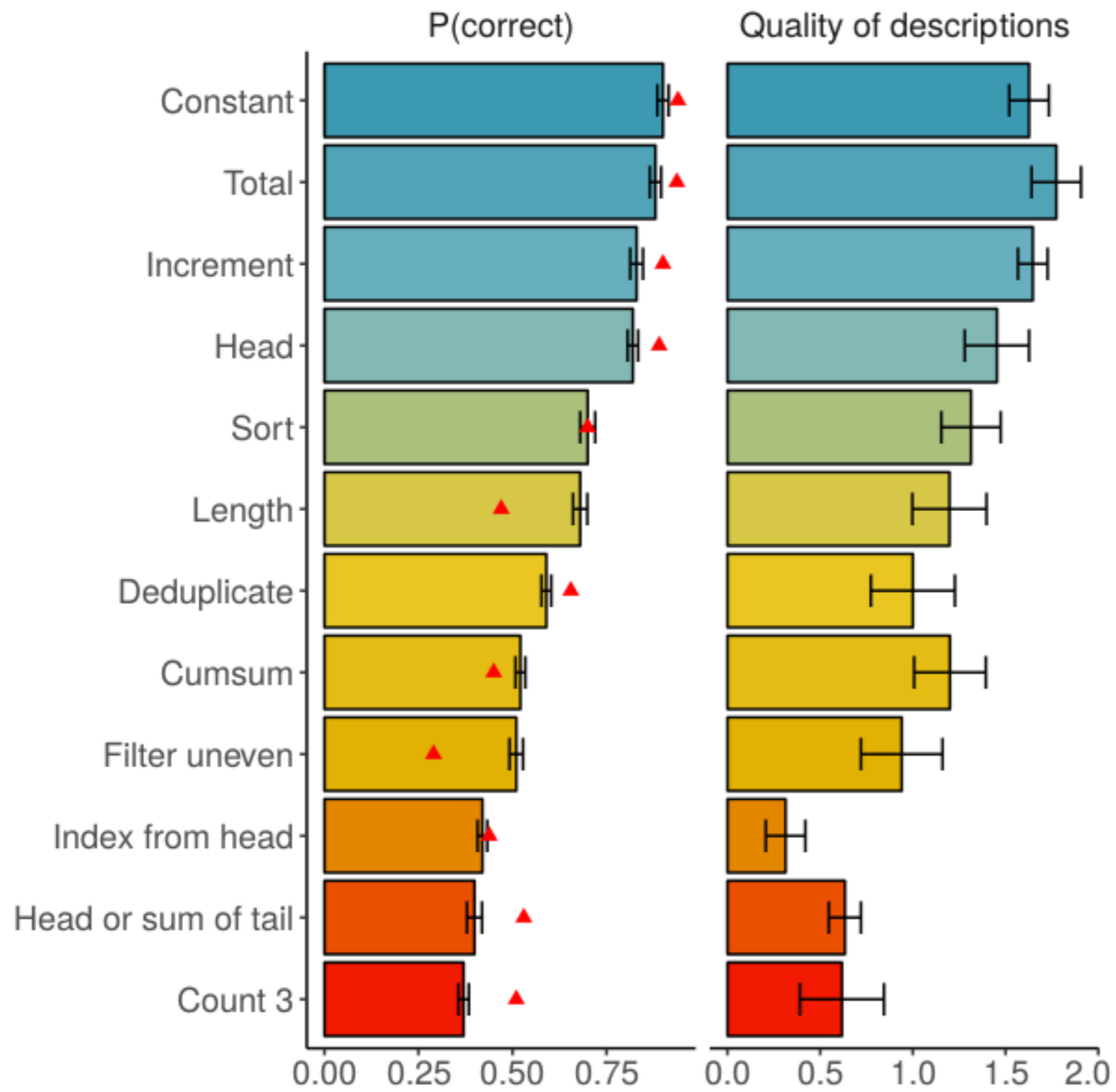
# head-or-tail: return the larger of head or sum-of-tail
# Example: head_or_tail([2,3,1]) = [4]
head-or-tail([]) = 0;
head-or-tail(cons(x_ y_)) =
  if(greater(x_ sum(y_)) x_ sum(y_));

# count3 xs: how often does 3 appear in xs?
# Example: count3([2,3,3]) = [2]
count3(x_) = count(succ(succ(succ(0))) x_);
```

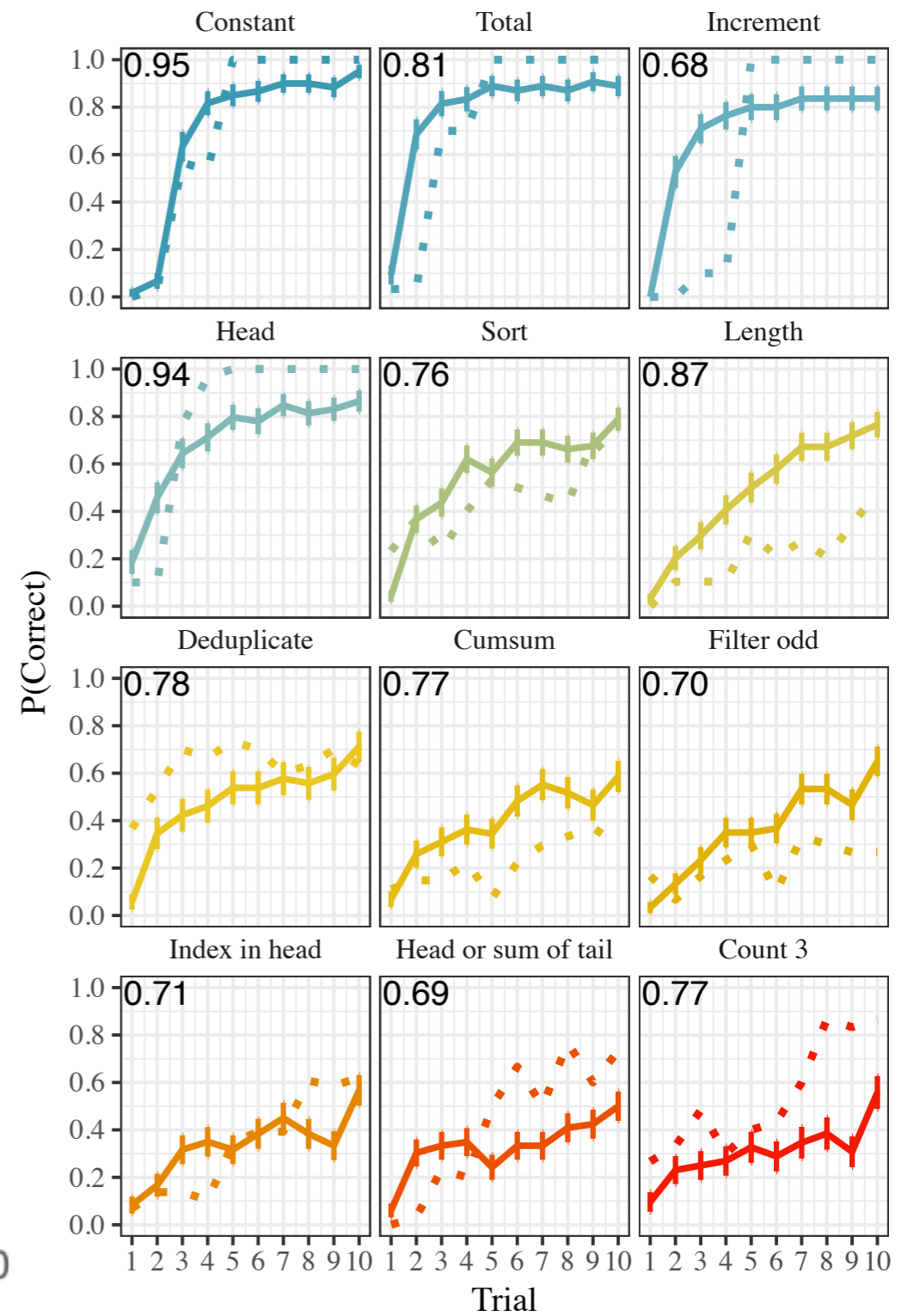
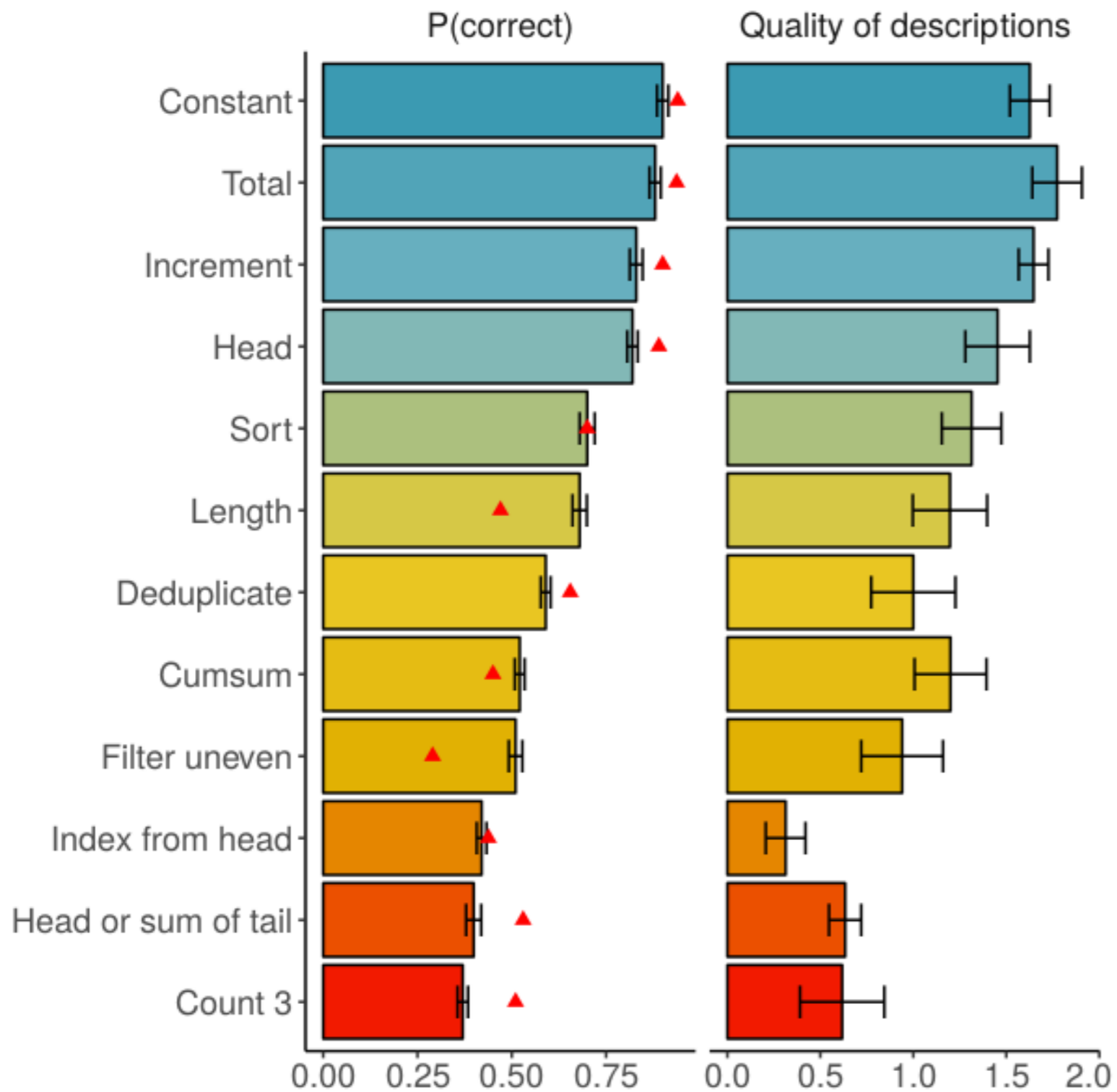
Experiment 1



Experiment 1



Experiment 1



Experiment 2

- **91 participants** (46 males, mean age=34.51, SD=10.57)
- **randomly assigned condition:**
relevant curriculum or random curriculum
- **4 concepts/condition**
- **10 trials/concept**

Experiment 2

```
# head xs: return the first element of xs
# Example: head([2,3,1]) = [2]
head(cons(x_ y_)) = x_;
```

```
# tail xs: return all but the first element of xs
# Example: tail([2,3,3]) = [3,3]
tail([]) = [];
tail(cons(x_ y_)) = y_;
```

```
# count3 xs: how often does 3 appear in xs?
# Example: count3([3,2,3]) = [2]
count3(x_) = count(succ(succ(succ(0))) x_);
```

```
# count-head-in-tail xs: how often is head in the tail?
# Example: count-head-in-tail([2,3,2]) = [1]
count-head-in-tail([]) = 0;
count-head-in-tail(x_) = count(head(x_) tail(x_));
```

Experiment 2

```
# head xs: return the first element of xs
# Example: head([2,3,1]) = [2]
head(cons(x_ y_)) = x_;

# tail xs: return all but the first element of xs
# Example: tail([2,3,3]) = [3,3]
tail([]) = [];
tail(cons(x_ y_)) = y_;
```

```
# count3 xs: how often does 3 appear in xs?
# Example: count3([3,2,3]) = [2]
count3(x_) = count(succ(succ(succ(0))) x_);

# count-head-in-tail xs: how often is head in the tail?
# Example: count-head-in-tail([2,3,2]) = [1]
count-head-in-tail([]) = 0;
count-head-in-tail(x_) = count(head(x_) tail(x_));
```

```
# const xs: return 3
# Example: const([1,2,4]) = [3]
const(x_) = 3;

# total xs: sum all the elements of xs
# Example: total([1,2,3]) = [6]
total(x_) = sum(x_);

# increment xs: add 1 to each element of xs
# Example: increment([1,2]) = [2,3]
increment(x_) = add(1 x_);

# length xs: compute the length of xs
# Example: length([2,3,1]) = [3]
length([]) = 0;
length(cons(x_ y_)) = succ(length(y_));

# sort xs: sort xs
# Example: sort([3,1]) = [1,3]
sort([]) = [];
sort(cons(x_ y_)) = insert(x_ sort(y_));

# deduplicate xs: remove all duplicates from xs
# Example: deduplicate([2,1,2,2,1]) = [2,1]
deduplicate([]) = [];
deduplicate(cons(x_ y_)) =
  cons(x_ deduplicate(remove(x_ y_)));
```

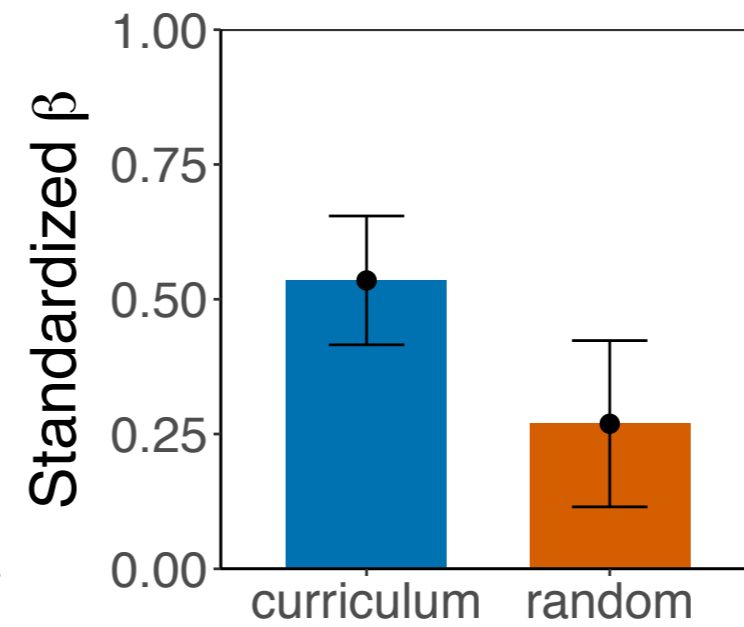
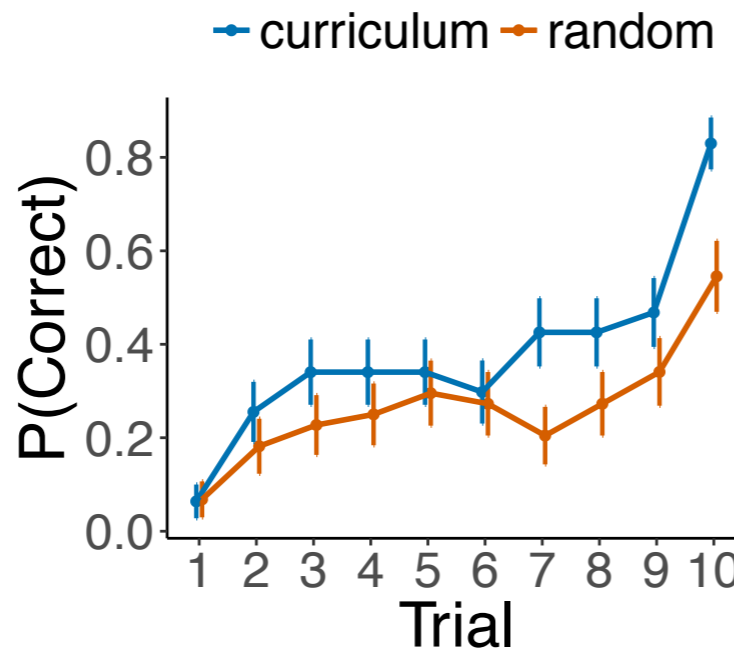
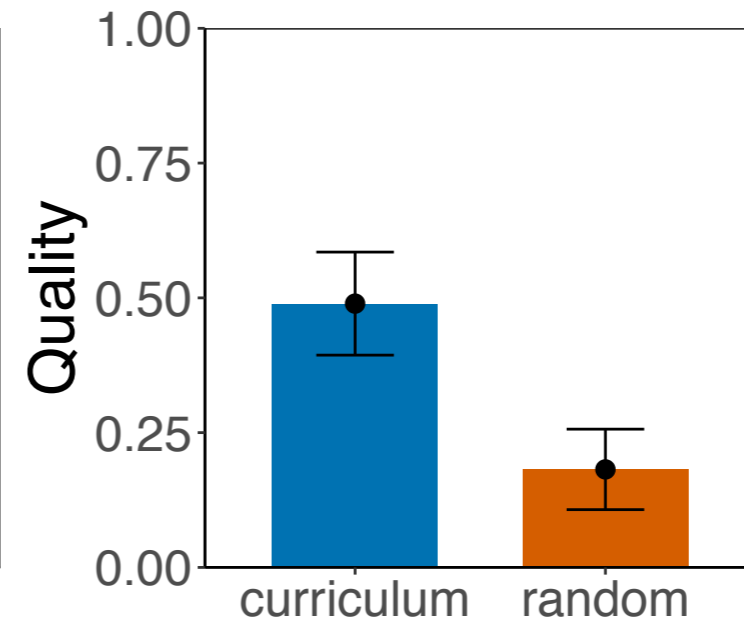
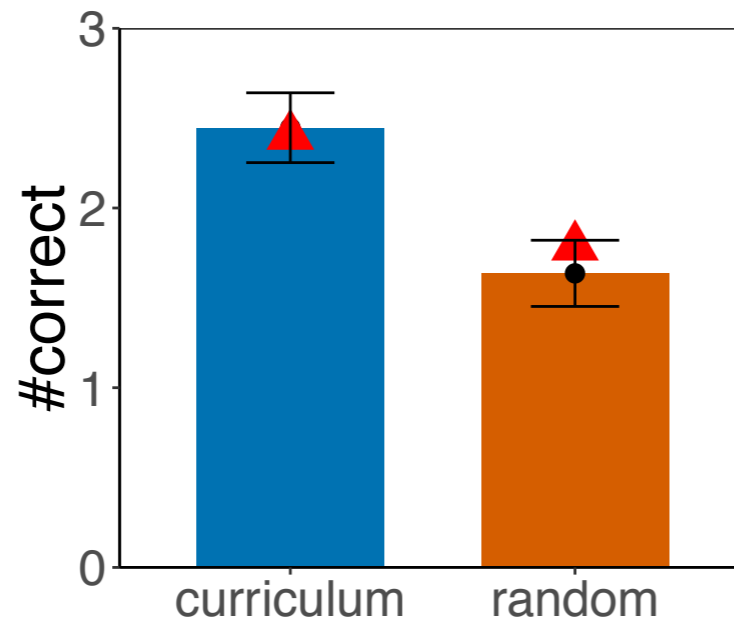
```
# cumsum xs: cumulatively sum the elements of xs
# Example: cumsum([2,3,1]) = [2,5,6]
cumsum([]) = [];
cumsum(cons(x_ y_)) = cons(x_ cumsum(add(x_ y_)));

# filter_odd xs: remove the odd numbers from xs
# Example: filter_odd([2,3,1,4]) = [2,4]
filter_odd([]) = [];
filter_odd(cons(x_ y_)) =
  if(even?(x_) cons(x_ filter_odd(y_)) filter_odd(y_));

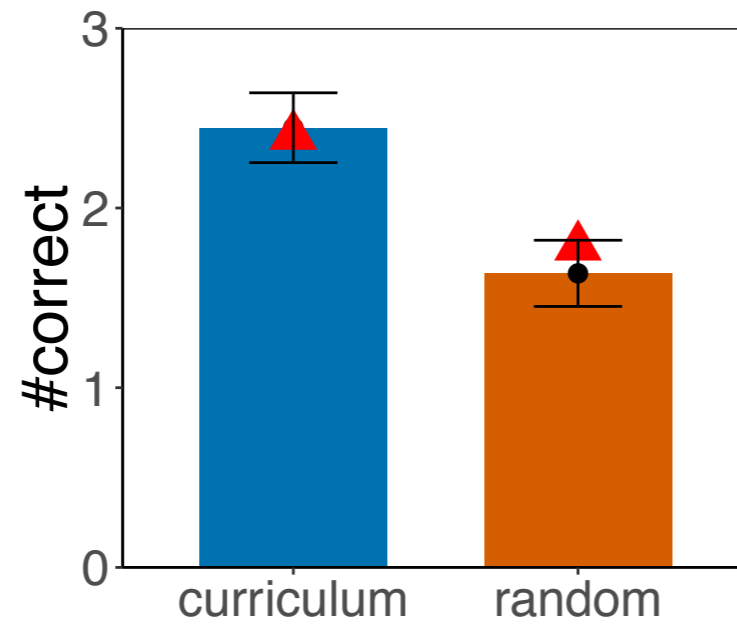
# index-in-head xs: return the headth element of the xs
# Example: index_in_head([2,3]) = [3]
index-in-head(cons(0 y_)) = 0
index-in-head(cons(succ(x_) y_)) = nth(x_ y_);

# head-or-tail: return the larger of head or sum-of-tail
# Example: head_or_tail([2,3,1]) = [4]
head-or-tail([]) = 0;
head-or-tail(cons(x_ y_)) =
  if(greater(x_ sum(y_)) x_ sum(y_));
```

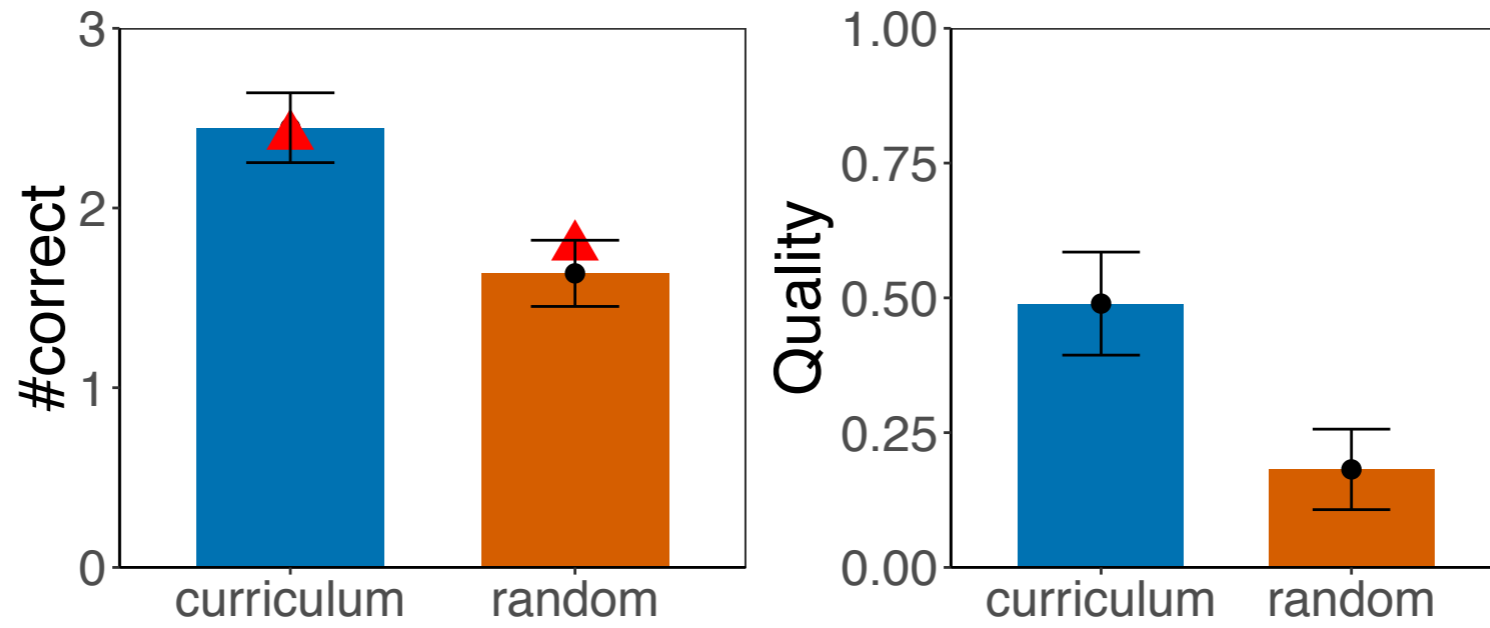
Experiment 2



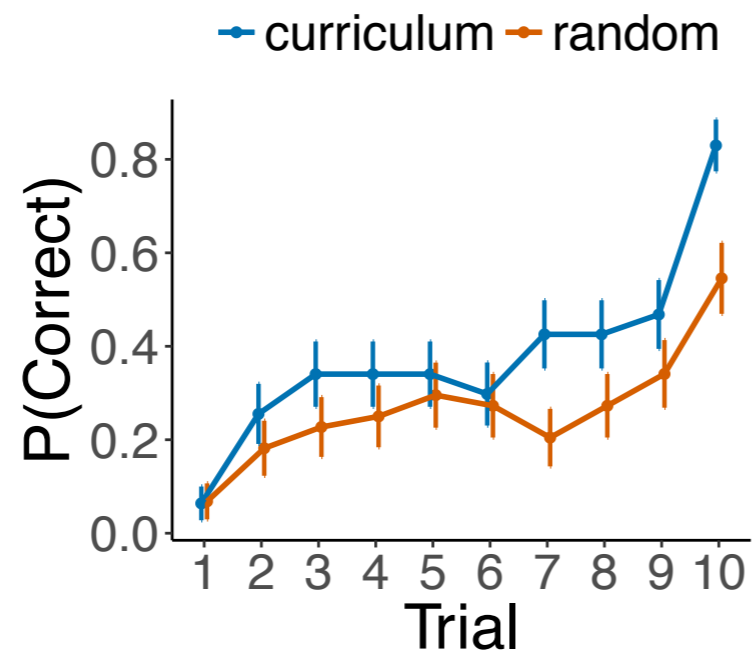
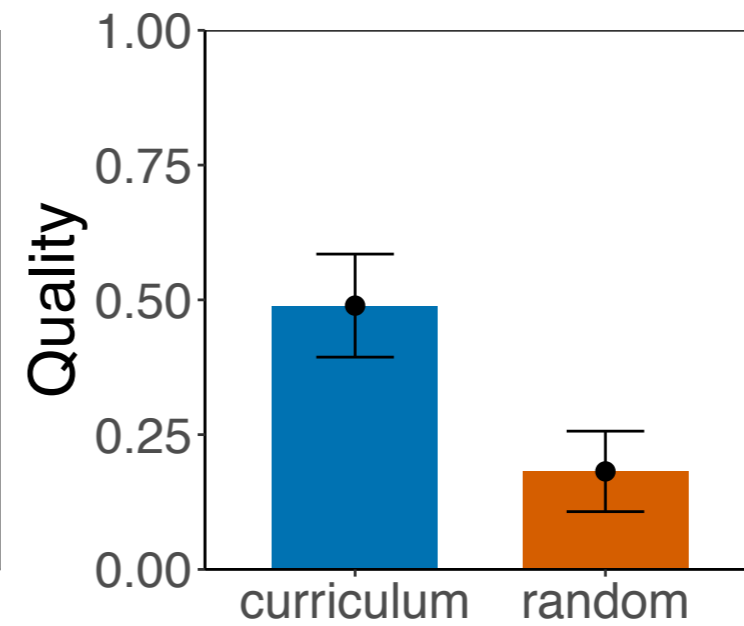
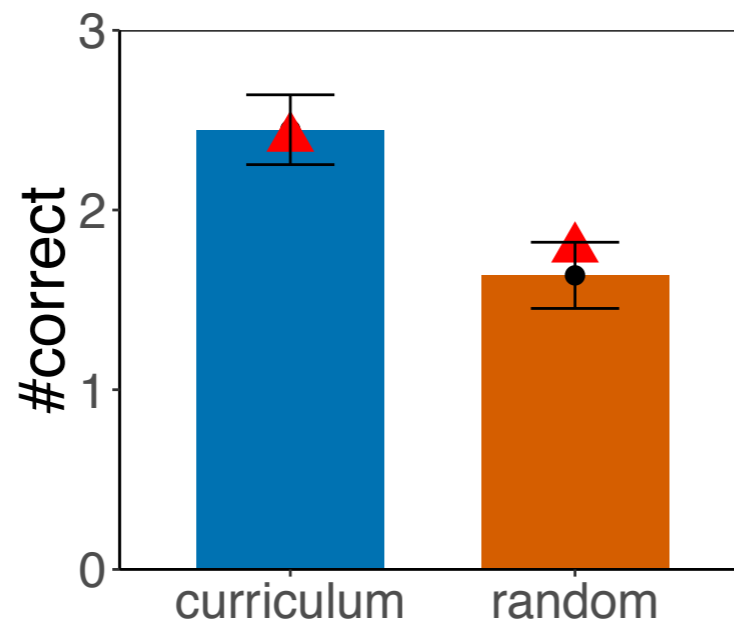
Experiment 2



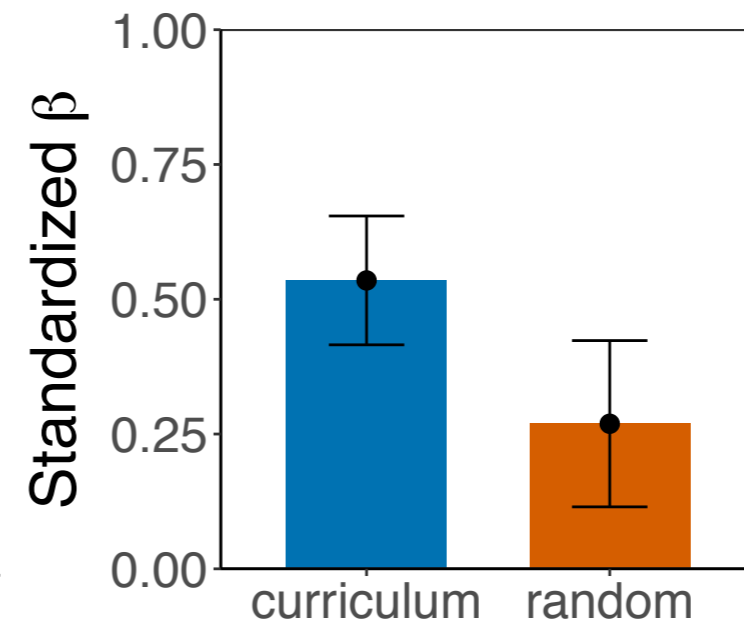
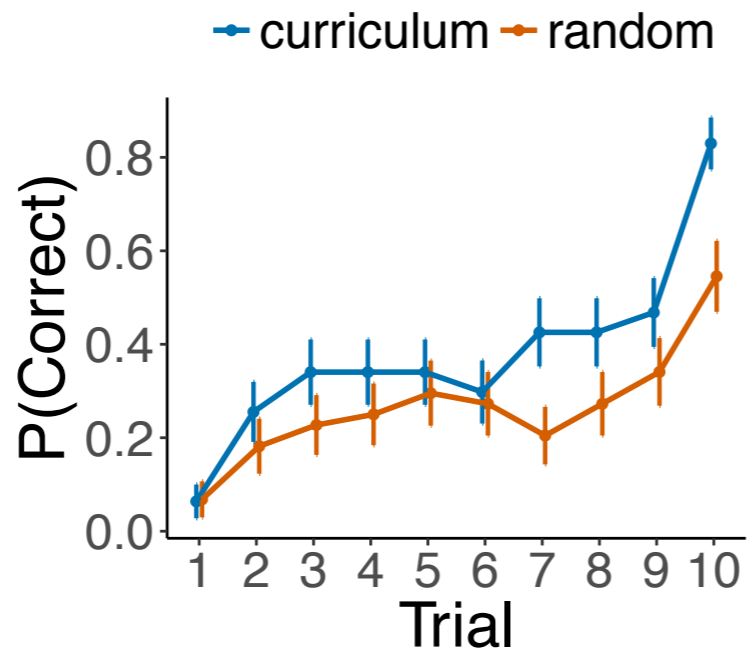
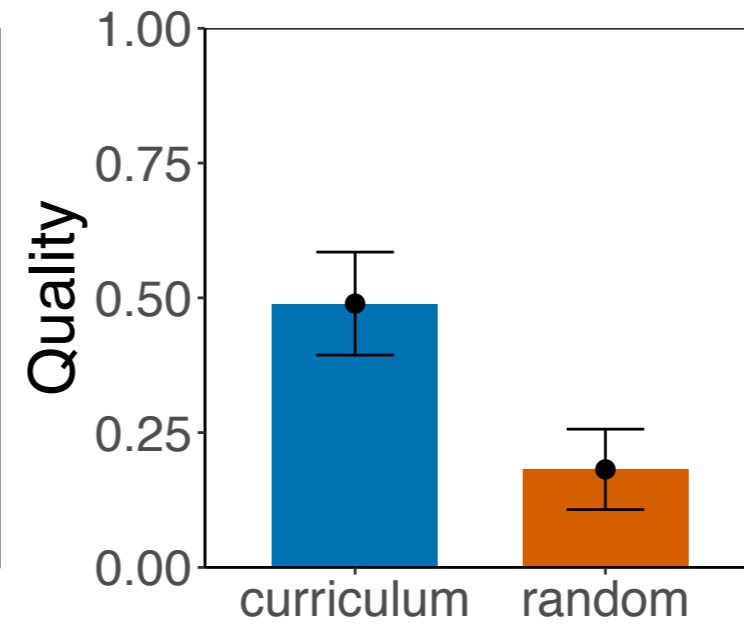
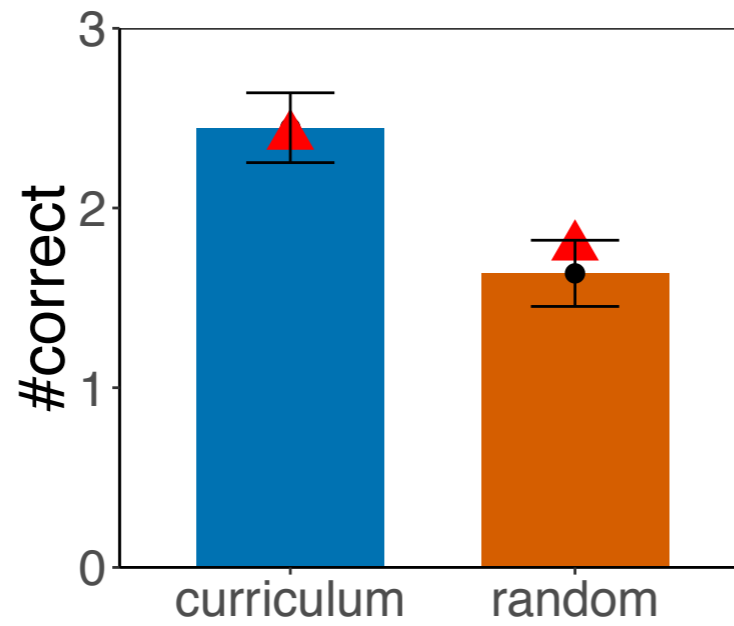
Experiment 2



Experiment 2



Experiment 2



This talk

- ▶ learning as programming
- ▶ bootstrapping the LOT with term rewriting
- ▶ toward a model of conceptual change

This talk

- ▶ learning as programming
- ▶ bootstrapping the LOT with term rewriting
- ▶ toward a model of conceptual change

Thank you!

