ngoodman@
stanford.edu

Learning and the language of thought

Noah D. Goodman
Stanford University

ICCV SIG-11
November 12, 2011

Motivation

What are thoughts made of?
(Concepts... but what are they?)

Motivation

What are thoughts made of?
(Concepts... but what are they?)

How are concepts learned?

Statistics and composition

Statistics and composition

Thought is useful
in an uncertain
world

Statistics and composition

Thought is useful
in an uncertain
world

Statistics and composition

Thought is useful
in an uncertain
world

Why did he yell at me?

Statistics and composition

Thought is useful
in an uncertain
world

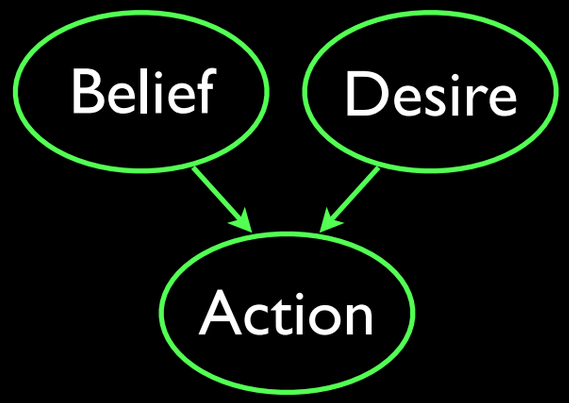
Why did he yell at me?

He wanted to hurt me.
He thought I was a telemarketer.

Statistics and composition

Thought is useful
in an uncertain
world

Why did he yell at me?

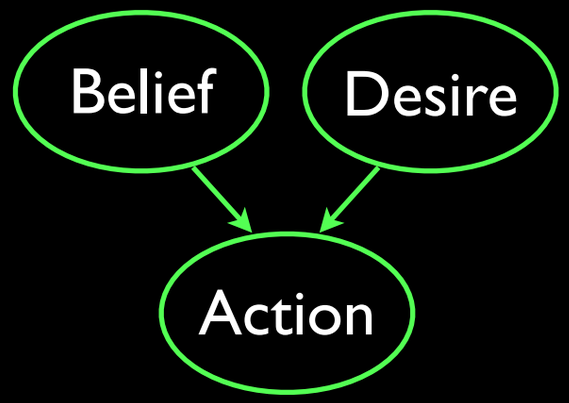


He wanted to hurt me.
He thought I was a telemarketer.

Statistics and composition

Thought is useful
in an uncertain
world

Why did he yell at me?

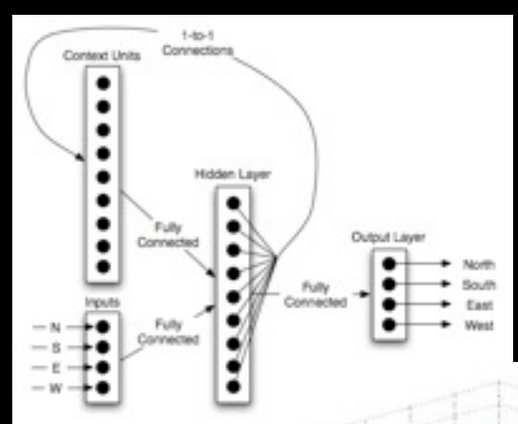


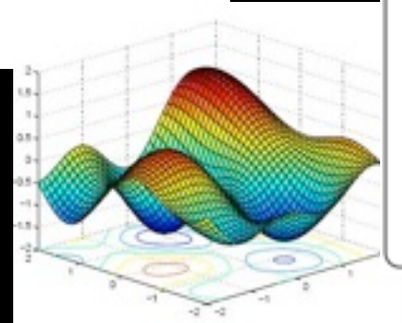
He wanted to hurt me.
He thought I was a telemarketer.

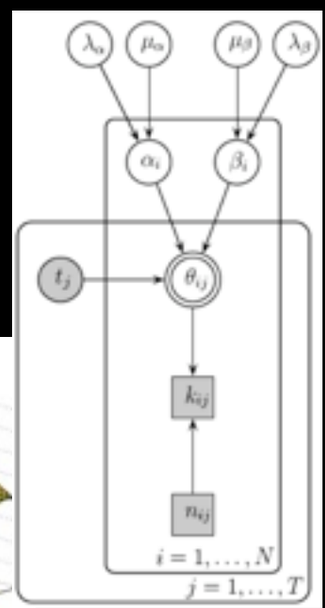
Probabilistic
inference

Statistics and composition

Thought is useful in an uncertain world





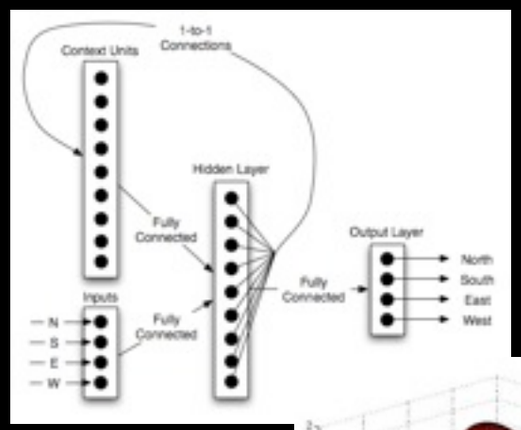


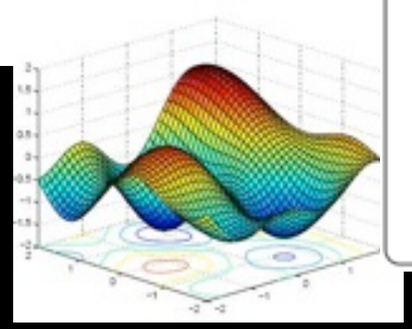
Probabilistic inference

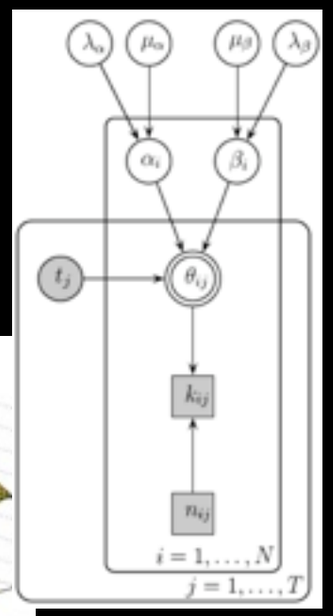
Statistics and composition

Thought is useful in an uncertain world

Thought is productive: "the infinite use of finite means"





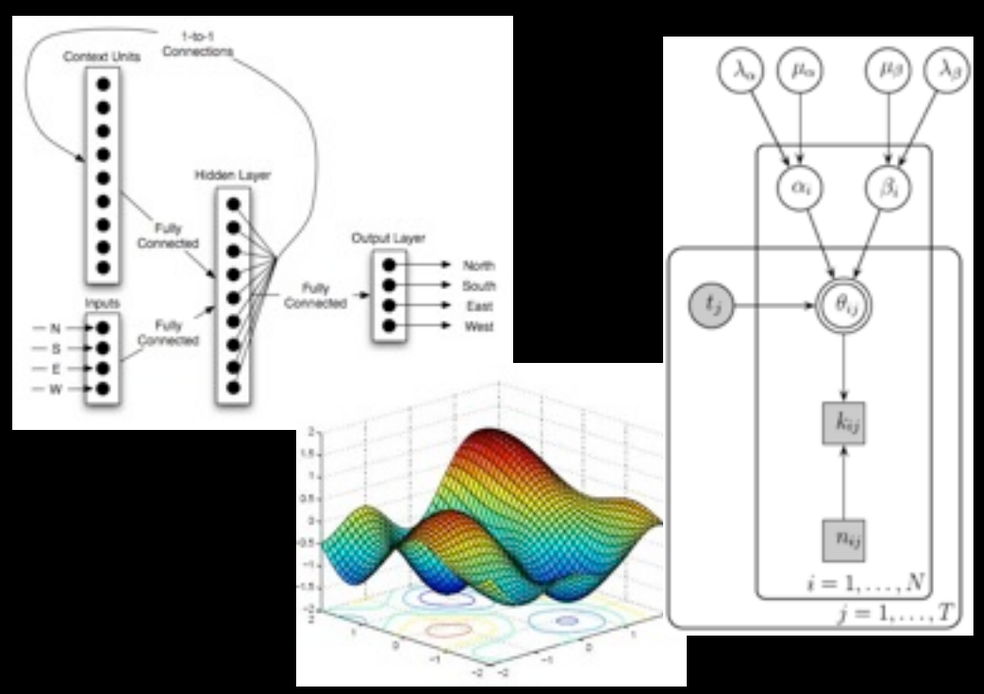


Probabilistic inference

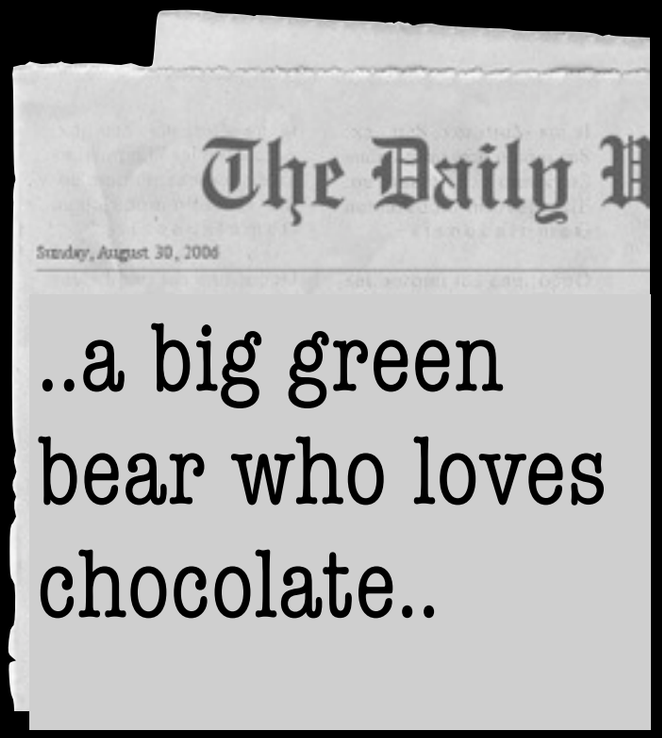
Statistics and composition

Thought is useful in an uncertain world

Thought is productive: "the infinite use of finite means"

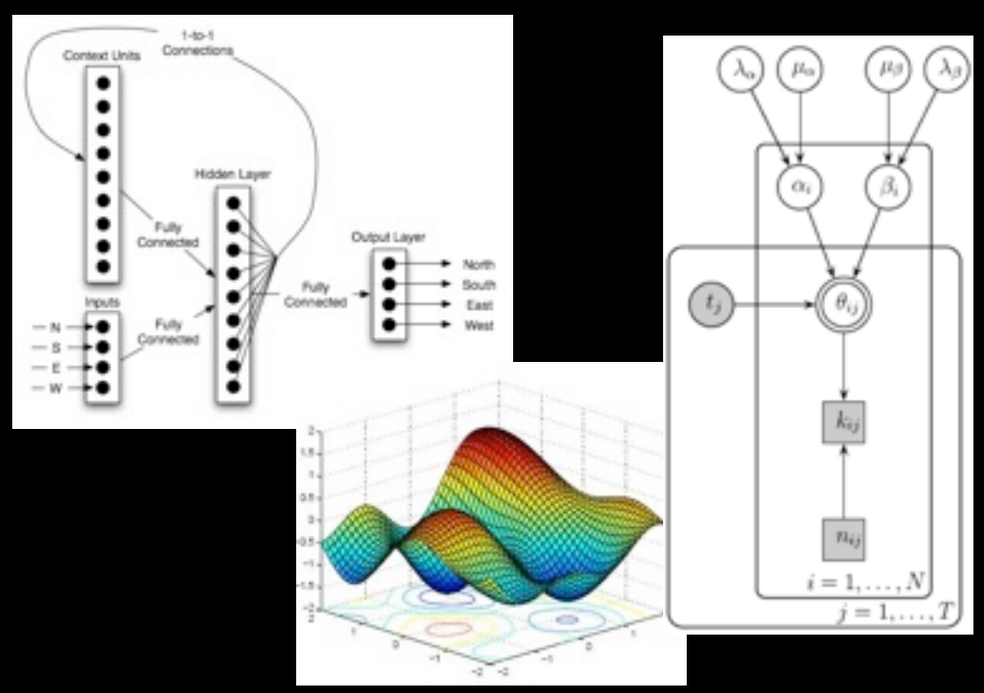


Probabilistic inference



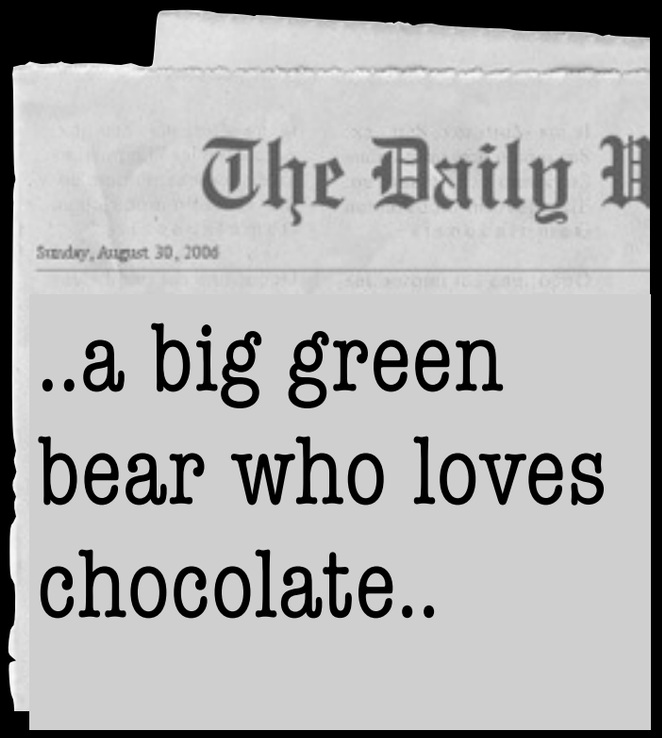
Statistics and composition

Thought is useful in an uncertain world



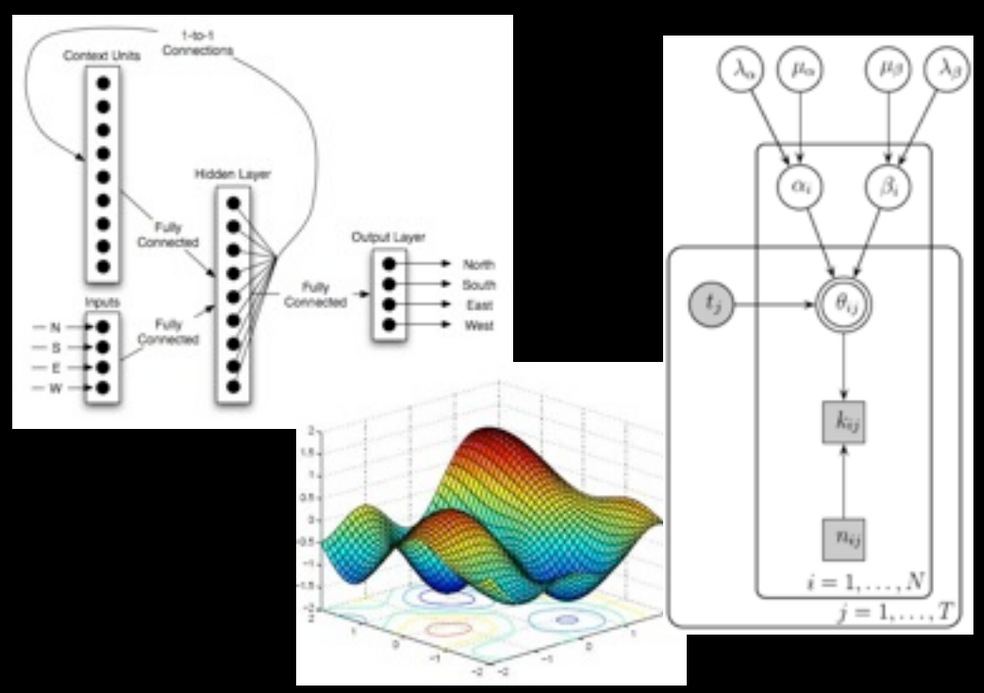
Probabilistic inference

Thought is productive: "the infinite use of finite means"



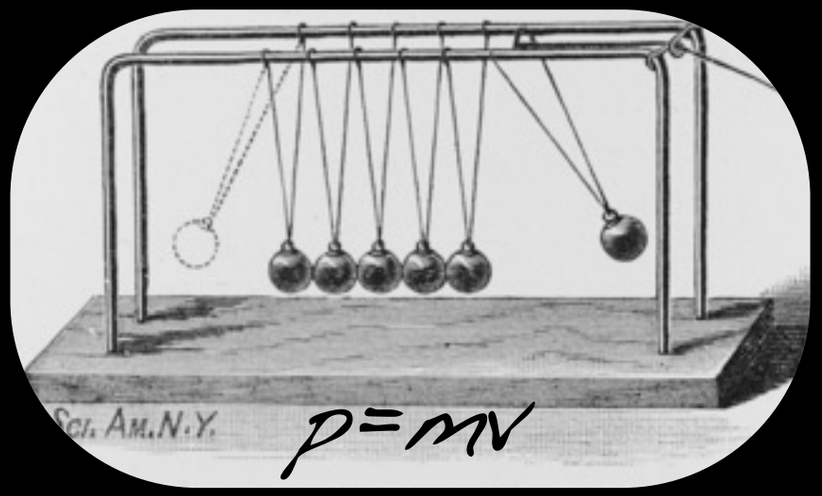
Statistics and composition

Thought is useful in an uncertain world



Probabilistic inference

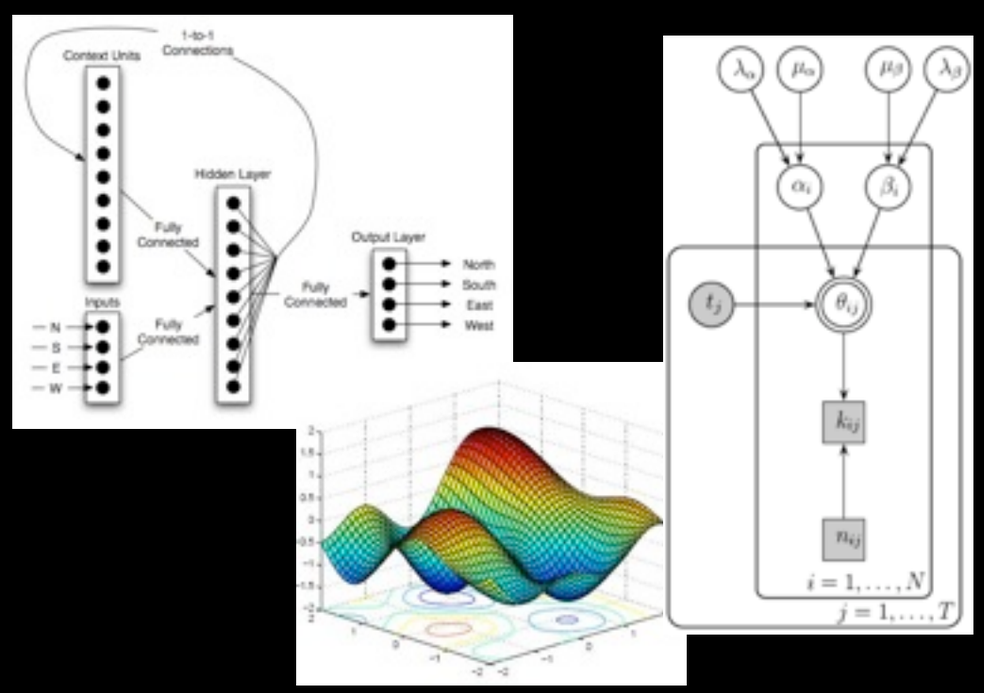
Thought is productive: "the infinite use of finite means"



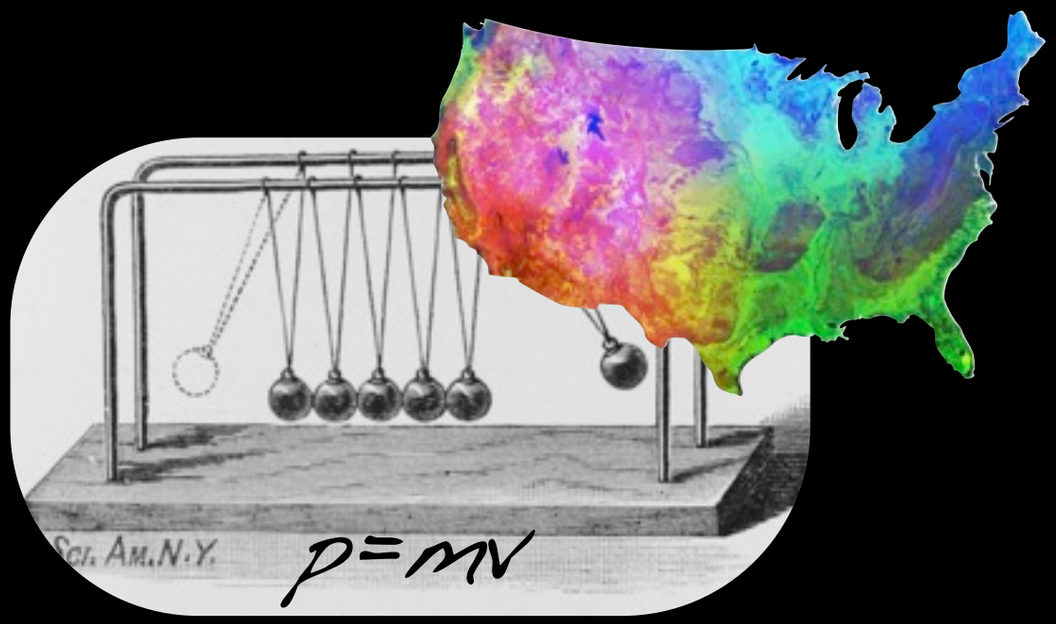
Statistics and composition

Thought is useful in an uncertain world

Thought is productive: "the infinite use of finite means"



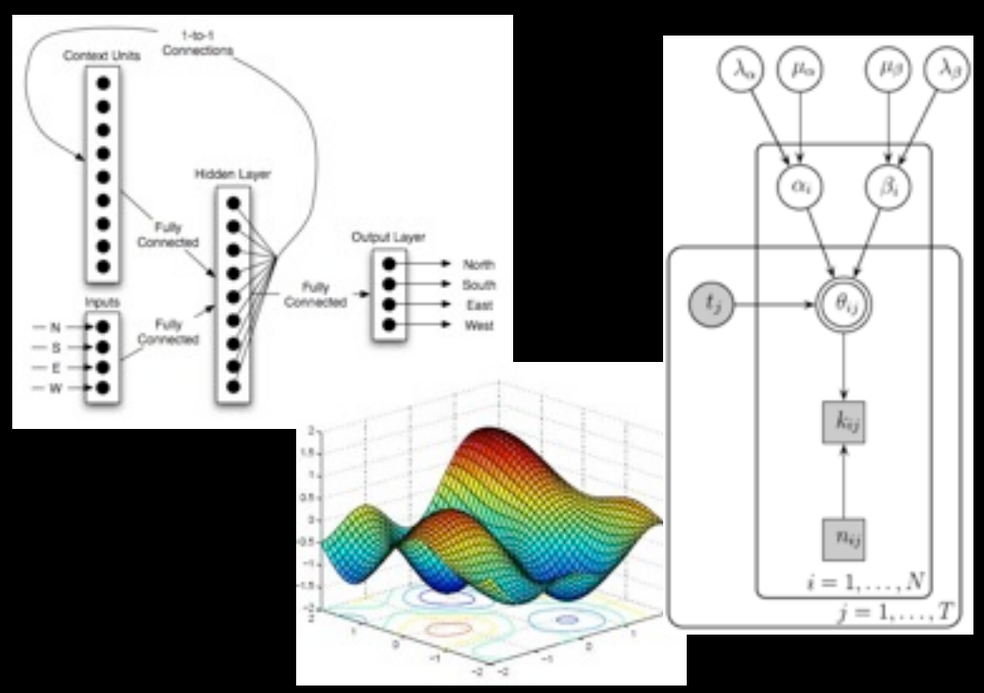
Probabilistic inference

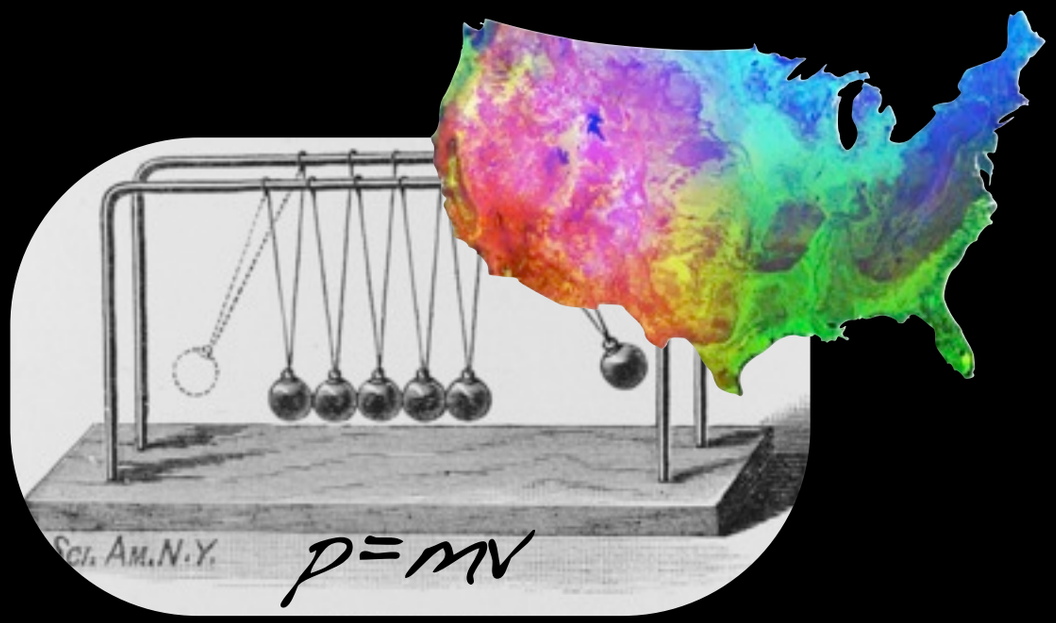


Statistics and composition

Thought is useful in an uncertain world

Thought is productive: "the infinite use of finite means"





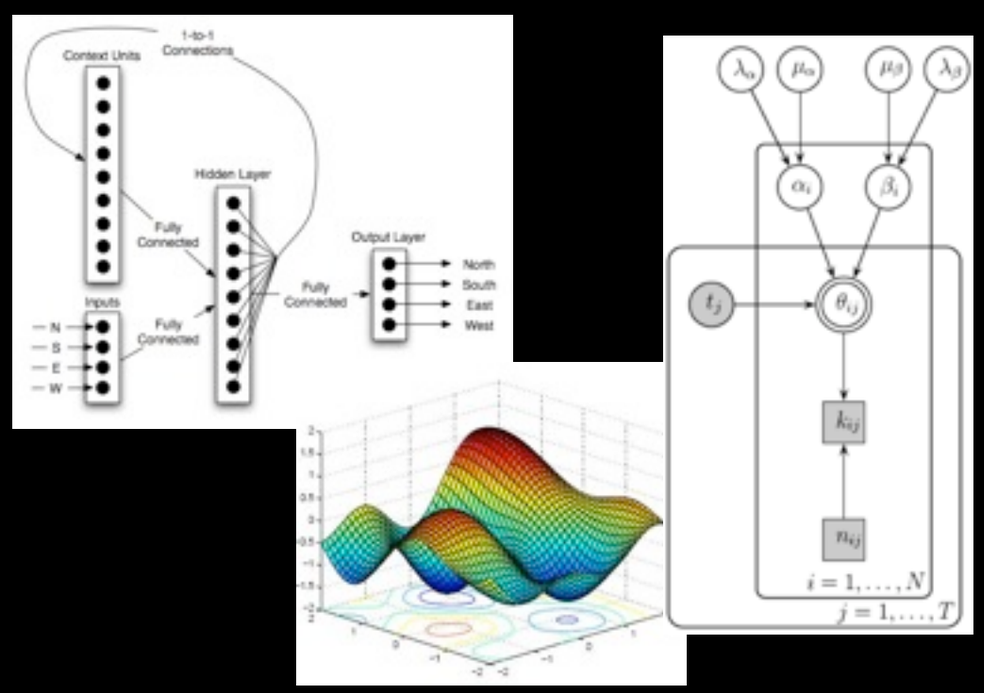
Probabilistic inference

Compositional representations

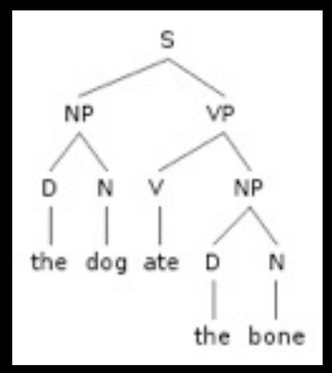
Statistics and composition

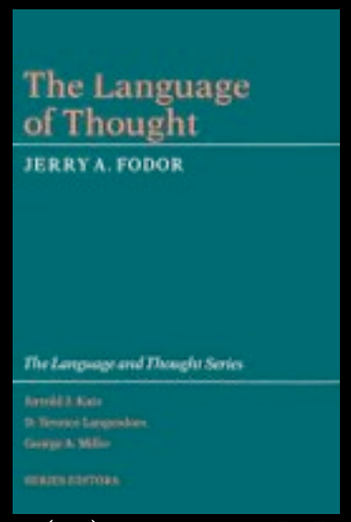
Thought is useful in an uncertain world

Thought is productive: "the infinite use of finite means"



Probabilistic inference





$$\forall x \; King(x) \implies Man(x)$$

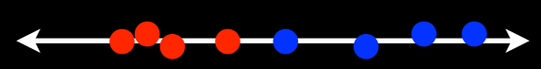
$$\forall y \; Man(y) \iff \neg Woman(y)$$

Compositional representations

Statistics and composition

Thought is useful in an uncertain world

Thought is productive: "the infinite use of finite means"



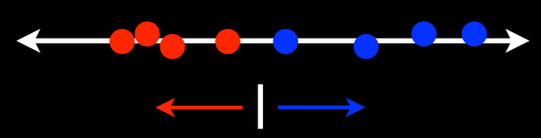
Probabilistic inference

Compositional representations

Statistics and composition

Thought is useful in an uncertain world

Thought is productive: "the infinite use of finite means"



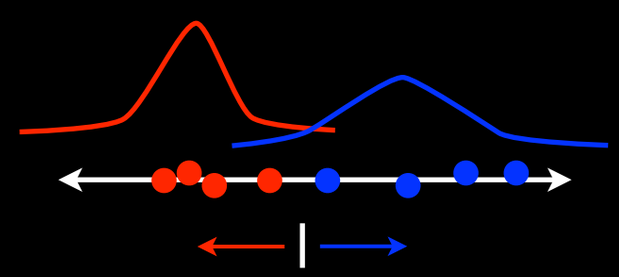
Probabilistic inference

Compositional representations

Statistics and composition

Thought is useful in an uncertain world

Thought is productive: "the infinite use of finite means"



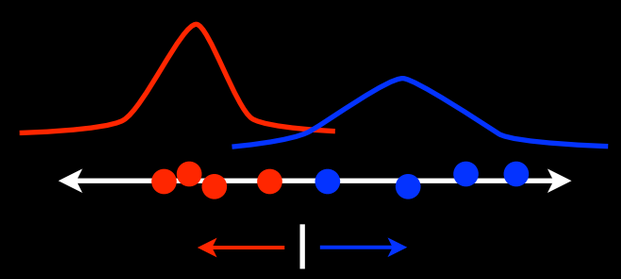
Probabilistic inference

Compositional representations

Statistics and composition

Thought is useful in an uncertain world

Thought is productive: "the infinite use of finite means"



Probabilistic inference

Generative models

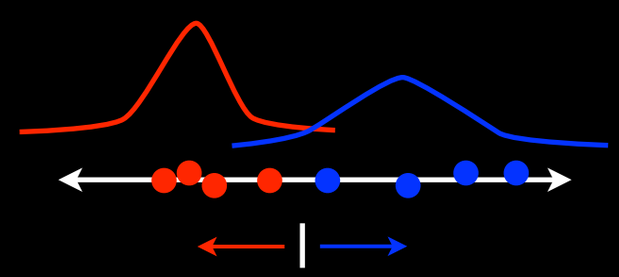
Compositional representations

Statistics and composition

Probabilistic language of thought hypothesis

Thought is useful in an uncertain world

Thought is productive: "the infinite use of finite means"



Probabilistic inference

Generative models

Compositional representations

Probabilistic LoT

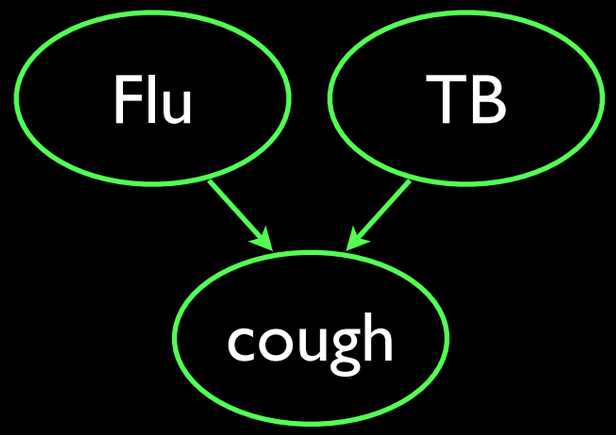
- The **probabilistic** language of thought hypothesis:
  - Mental representations are compositional,
  - Their meaning is probabilistic,
  - They encode generative knowledge,
- Hence, they support thinking and learning by probabilistic inference.

Probabilistic LoT

- The **probabilistic** language of thought hypothesis:
  - Mental representations are compositional,
  - Their meaning is probabilistic,
  - They encode generative knowledge,
- Hence, they support thinking and learning by probabilistic inference.

Can this hypothesis be formalized?

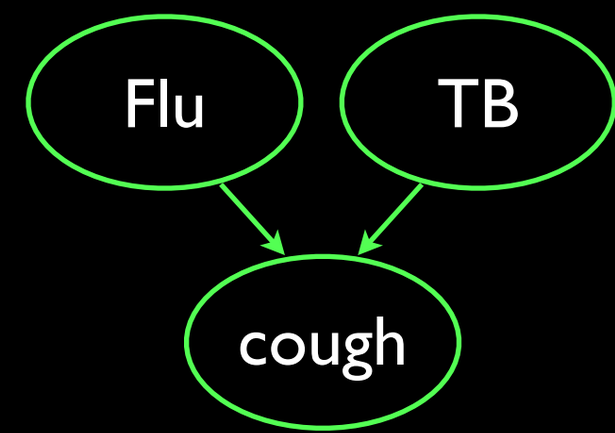
Probabilistic generative models



- Bayes nets:
  - Mental model of the causal process that gives rise to observations.
  - What's "hidden inside" the nodes and arrows?

Probabilistic generative models

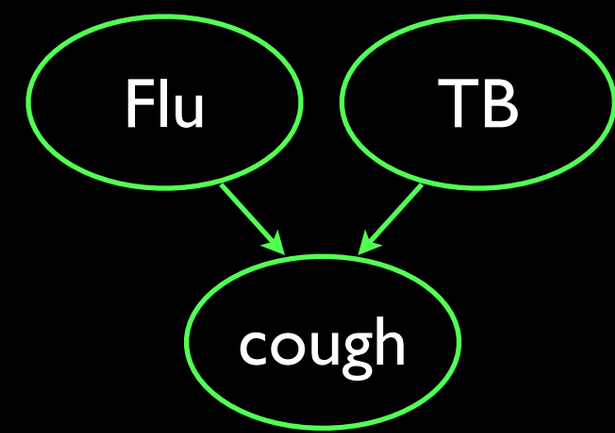
- Bayes nets:
  - Mental model of the causal process that gives rise to observations.
  - What's "hidden inside" the nodes and arrows?



cough	TB	no TB
flu	0.8	0.7
no flu	0.7	0.1

Probabilistic generative models

- Bayes nets:
  - Mental model of the causal process that gives rise to observations.

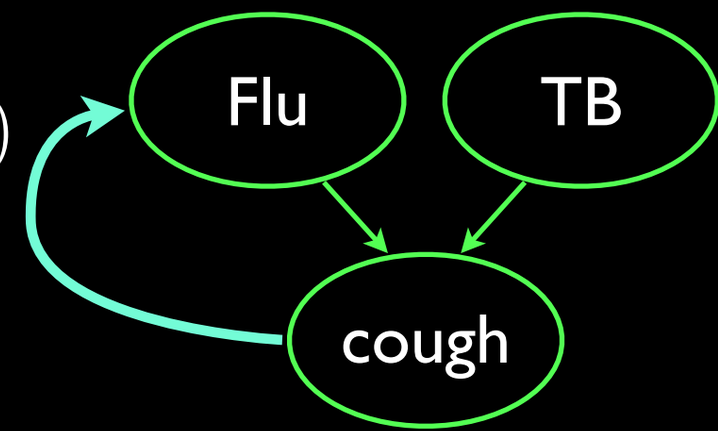


cough	TB	no TB
flu	0.8	0.7
no flu	0.7	0.1

Probabilistic generative models

$$P(H|d) \propto P(d|H)P(H)$$

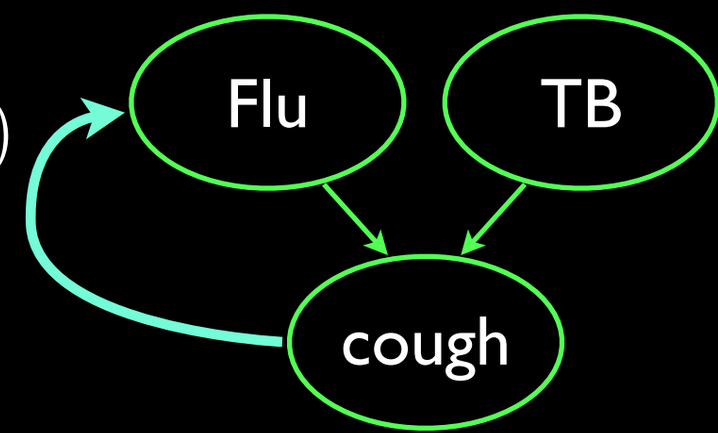
- Bayes nets:
  - Mental model of the causal process that gives rise to observations.



cough	TB	no TB
flu	0.8	0.7
no flu	0.7	0.1

Probabilistic generative models

$$P(H|d) \propto P(d|H)P(H)$$

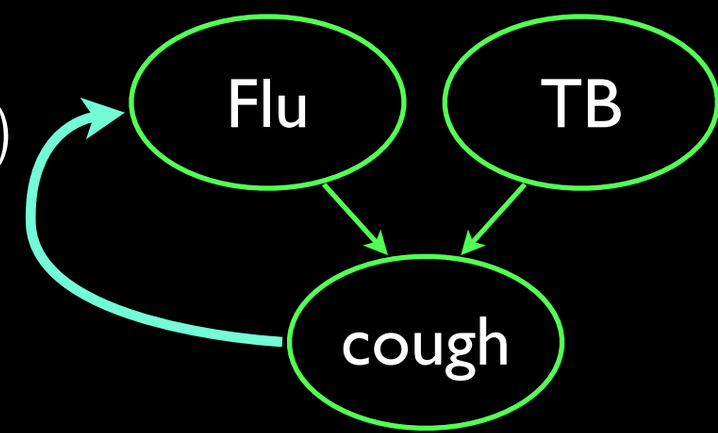


cough	TB	no TB
flu	0.8	0.7
no flu	0.7	0.1

- Bayes nets:
  - Mental model of the causal process that gives rise to observations.
  - What's "hidden inside" the nodes and arrows?

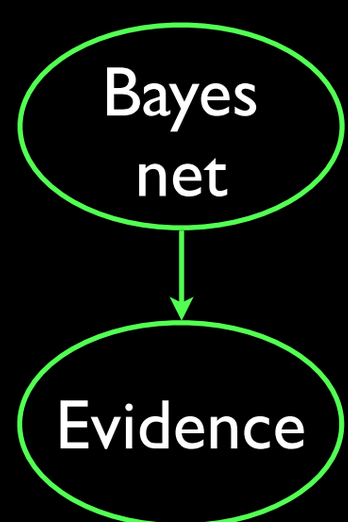
Probabilistic generative models

$$P(H|d) \propto P(d|H)P(H)$$



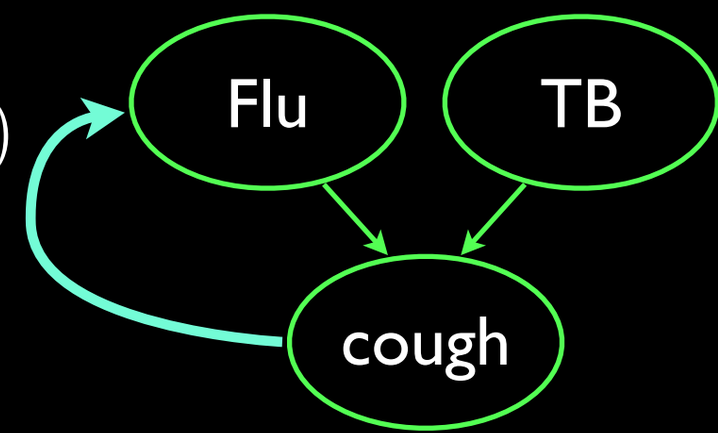
cough	TB	no TB
flu	0.8	0.7
no flu	0.7	0.1

- Bayes nets:
  - Mental model of the causal process that gives rise to observations.
  - What's "hidden inside" the nodes and arrows?



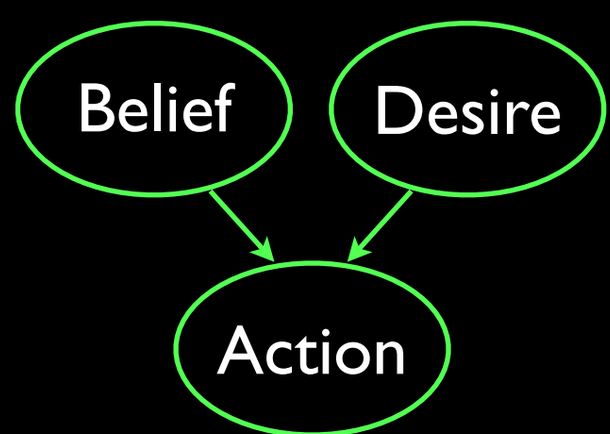
Probabilistic generative models

$$P(H|d) \propto P(d|H)P(H)$$



cough	TB	no TB
flu	0.8	0.7
no flu	0.7	0.1

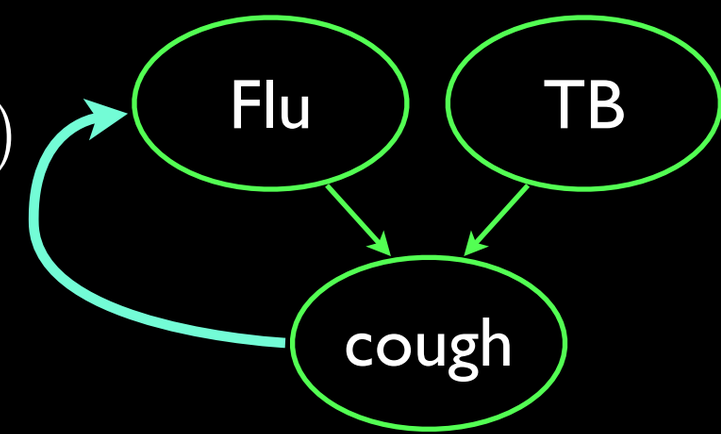
- Bayes nets:
  - Mental model of the causal process that gives rise to observations.
  - What's "hidden inside" the nodes and arrows?



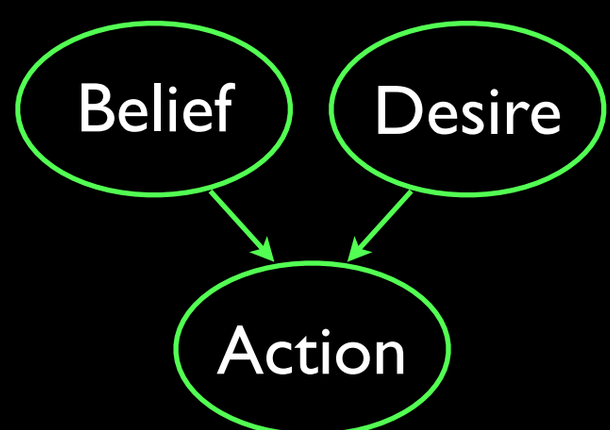
Probabilistic generative models

$$P(H|d) \propto P(d|H)P(H)$$

- Bayes nets:
  - Mental model of the causal process that gives rise to observations.
  - What's "hidden inside" the nodes and arrows?



cough	TB	no TB
flu	0.8	0.7
no flu	0.7	0.1



write talk	want fame	want sleep	..
going to workshop	0.7	0.05	
it's raining	0.002	0.004	
:			

λ calculus

λ calculus

- Notation:
  - Function have parentheses on the wrong side: `(sin x)`
  - Operators always go at the beginning: `(+ x y)`

λ calculus

- Notation:
  - Function have parentheses on the wrong side: `(sin x)`
  - Operators always go at the beginning: `(+ x y)`
- λ makes functions, define binds values to symbols:

```
(define double
  (λ (x) (+ x x)))
```

λ calculus

- Notation:
  - Function have parentheses on the wrong side: `(sin x)`
  - Operators always go at the beginning: `(+ x y)`
- λ makes functions, define binds values to symbols:

```
(define double
  (λ (x) (+ x x)))
```

```
(double 3)   => 6
```

λ calculus

- Notation:
  - Function have parentheses on the wrong side: `(sin x)`
  - Operators always go at the beginning: `(+ x y)`
- λ makes functions, define binds values to symbols:

```
(define double
  (λ (x) (+ x x)))
```

```
(double 3)   => 6
```

```
(define repeat
  (λ (f) (λ (x) (f (f x)))))
```

λ calculus

- Notation:
  - Function have parentheses on the wrong side: `(sin x)`
  - Operators always go at the beginning: `(+ x y)`
- λ makes functions, define binds values to symbols:

```
(define double
  (λ (x) (+ x x)))
```

```
(double 3)   => 6
```

```
(define repeat
  (λ (f) (λ (x) (f (f x)))))
```

```
((repeat double) 3)   => 12
```

λ calculus

- Notation:
  - Function have parentheses on the wrong side: `(sin x)`
  - Operators always go at the beginning: `(+ x y)`
- λ makes functions, define binds values to symbols:

```
(define double
  (λ (x) (+ x x)))
```

```
(double 3)  => 6
```

```
(define repeat
  (λ (f) (λ (x) (f (f x)))))
```

```
((repeat double) 3)  => 12
```

```
(define 2nd-derivative (repeat derivative))
```

Church

Random primitives:

```
(define a (flip 0.3))
(define b (flip 0.3))
(define c (flip 0.3))
(+ a b c)
```

Goodman, Mansinghka, Roy, Bonawitz, Tenenabum (2008)

Church

Random primitives:

```
(define a (flip 0.3)) => 1
(define b (flip 0.3))
(define c (flip 0.3))
(+ a b c)
```

Goodman, Mansinghka, Roy, Bonawitz, Tenenabum (2008)

Church

Random primitives:

```
(define a (flip 0.3)) => 1
(define b (flip 0.3)) => 0
(define c (flip 0.3))
(+ a b c)
```

Goodman, Mansinghka, Roy, Bonawitz, Tenenabum (2008)

Church

Random primitives:

```
(define a (flip 0.3)) => 1
(define b (flip 0.3)) => 0
(define c (flip 0.3)) => 1
(+ a b c)
```

Goodman, Mansinghka, Roy, Bonawitz, Tenenabum (2008)

Church

Random primitives:

```
(define a (flip 0.3)) => 1
(define b (flip 0.3)) => 0
(define c (flip 0.3)) => 1
(+ a b c)             => 2
```

Goodman, Mansinghka, Roy, Bonawitz, Tenenabum (2008)

Church

Random primitives:

```
(define a (flip 0.3)) => 1 0
(define b (flip 0.3)) => 0 0
(define c (flip 0.3)) => 1 0
(+ a b c)             => 2 0
```

Goodman, Mansinghka, Roy, Bonawitz, Tenenabum (2008)

Church

Random primitives:

```
(define a (flip 0.3)) => 1 0 0
(define b (flip 0.3)) => 0 0 0
(define c (flip 0.3)) => 1 0 1
(+ a b c)             => 2 0 1
```

Goodman, Mansinghka, Roy, Bonawitz, Tenenabum (2008)

Church

Random primitives:

```
(define a (flip 0.3)) => 1 0 0
(define b (flip 0.3)) => 0 0 0
(define c (flip 0.3)) => 1 0 1
(+ a b c)             => 2 0 1..
```

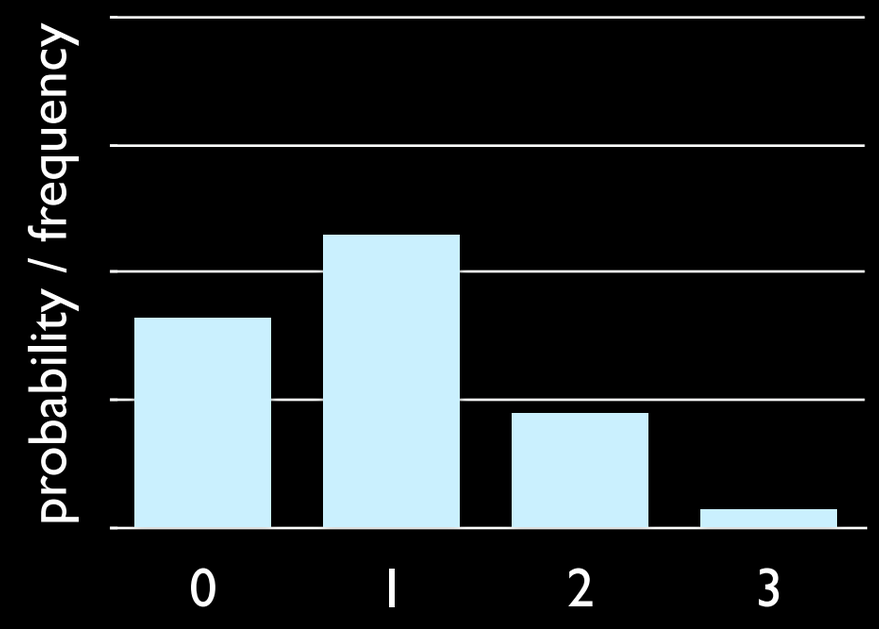
Goodman, Mansinghka, Roy,
Bonawitz, Tenenabum (2008)

Church

Random primitives:

```
(define a (flip 0.3))
(define b (flip 0.3))
(define c (flip 0.3))
(+ a b c)
```

```
=> 1 0 0
=> 0 0 0
=> 1 0 1
=> 2 0 1..
```



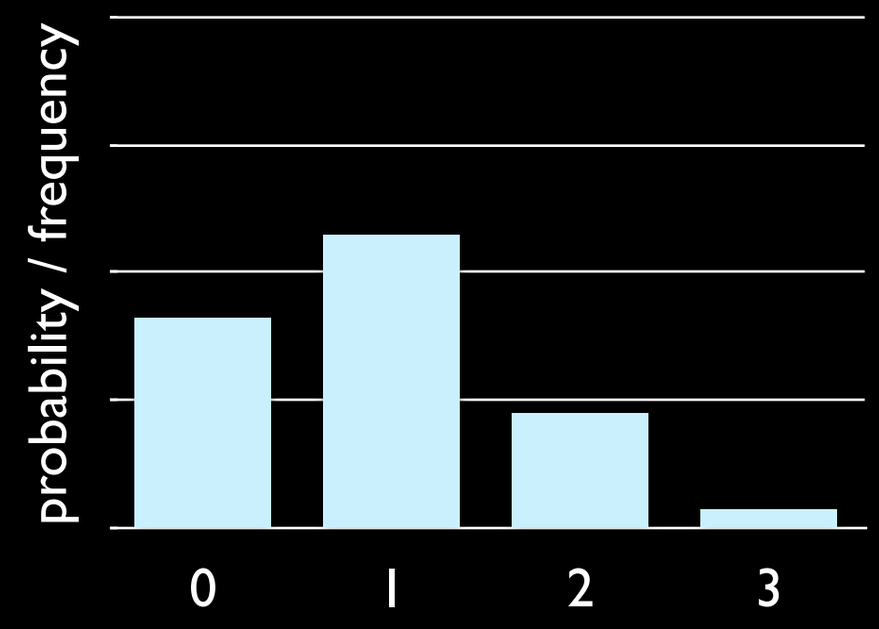
Goodman, Mansinghka, Roy, Bonawitz, Tenenabum (2008)

Church

Random primitives:

```
(define a (flip 0.3))
(define b (flip 0.3))
(define c (flip 0.3))
(+ a b c)
```

```
=> 1 0 0
=> 0 0 0
=> 1 0 1
=> 2 0 1..
```



Theorem: Any computable distribution can be represented by a Church expression.

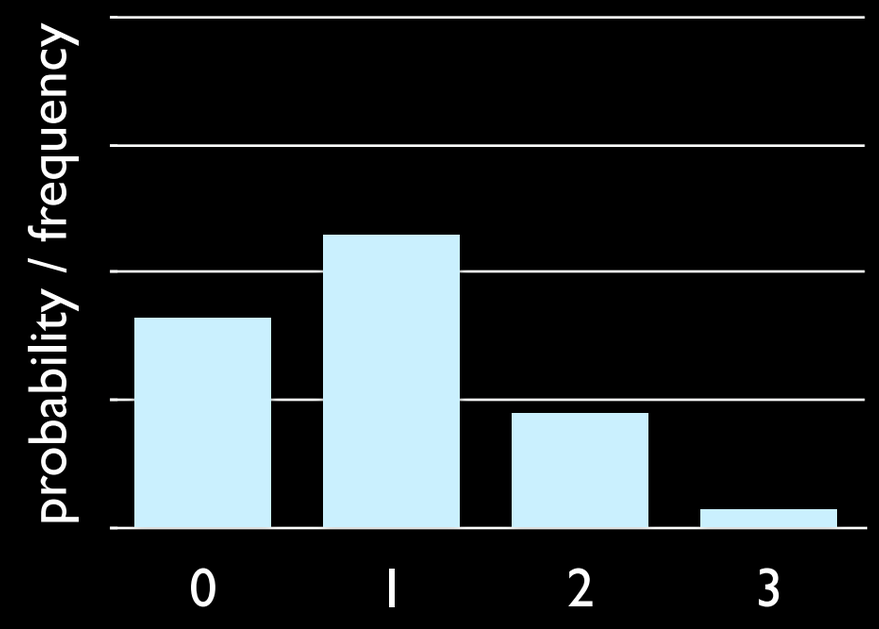
Goodman, Mansinghka, Roy, Bonawitz, Tenenabum (2008)

Church

Random primitives:

```
(define a (flip 0.3))
(define b (flip 0.3))
(define c (flip 0.3))
(+ a b c)
```

```
=> 1 0 0
=> 0 0 0
=> 1 0 1
=> 2 0 1..
```

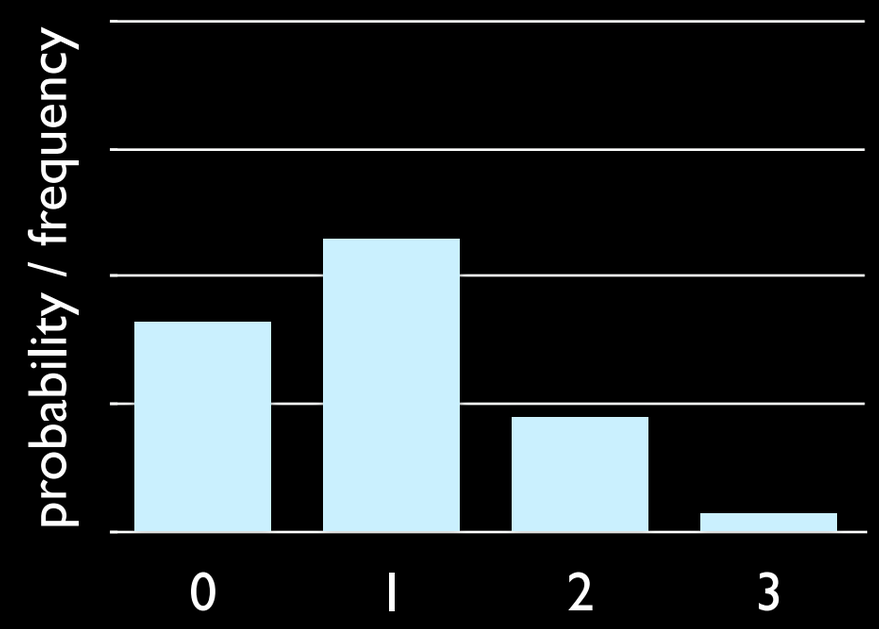


Goodman, Mansinghka, Roy, Bonawitz, Tenenabum (2008)

Church

Random primitives:

```
(define a (flip 0.3)) => 1 0 0
(define b (flip 0.3)) => 0 0 0
(define c (flip 0.3)) => 1 0 1
(+ a b c)             => 2 0 1..
```



Conditioning (inference):

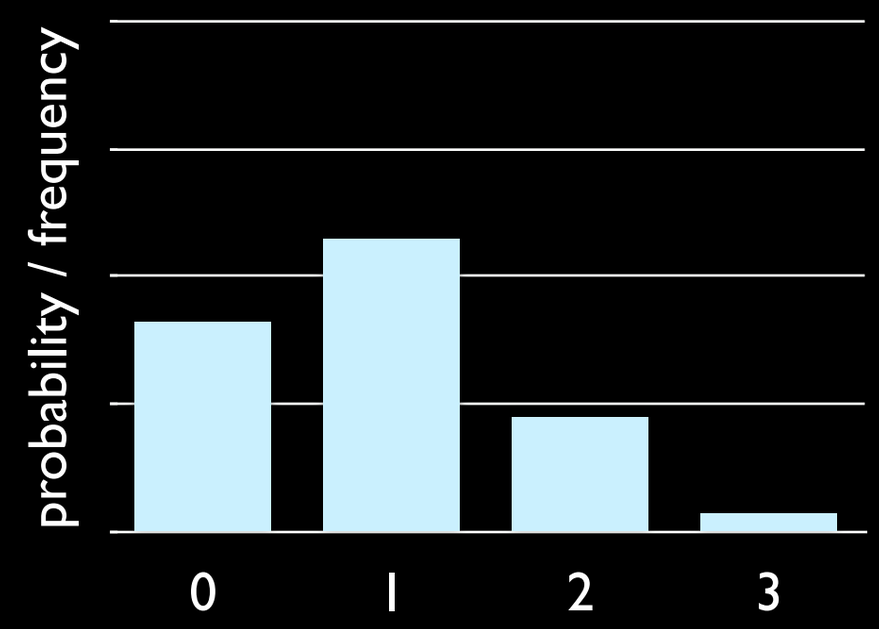
```
(query
  (define a (flip 0.3))
  (define b (flip 0.3))
  (define c (flip 0.3))
  (+ a b c)
  (= (+ a b) 1))
```

Goodman, Mansinghka, Roy, Bonawitz, Tenenabum (2008)

Church

Random primitives:

```
(define a (flip 0.3)) => 1 0 0
(define b (flip 0.3)) => 0 0 0
(define c (flip 0.3)) => 1 0 1
(+ a b c)             => 2 0 1..
```



Conditioning (inference):

```
(query
  (define a (flip 0.3))
  (define b (flip 0.3))
  (define c (flip 0.3))
  (+ a b c)
  (= (+ a b) 1))
```

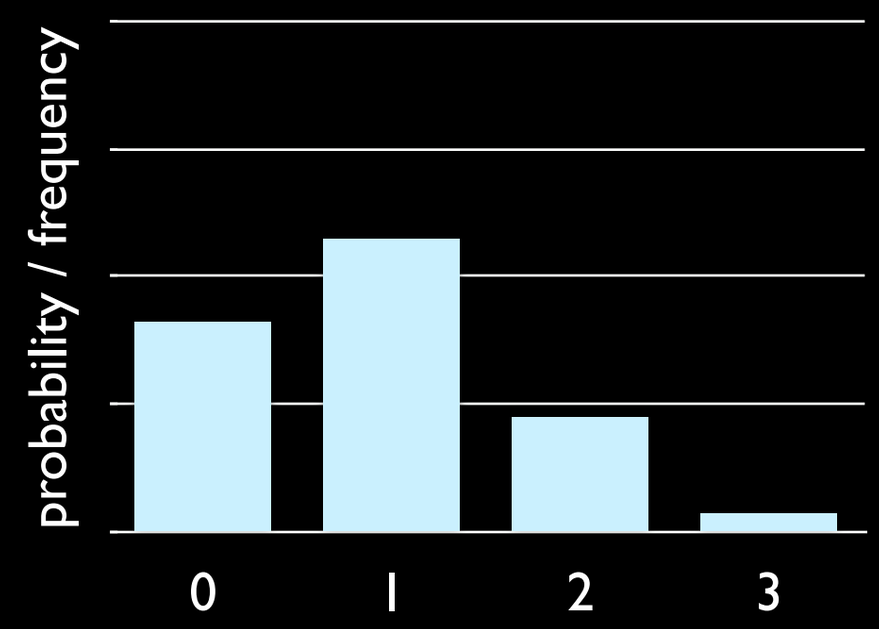
Query

Goodman, Mansinghka, Roy, Bonawitz, Tenenabum (2008)

Church

Random primitives:

```
(define a (flip 0.3)) => 1 0 0
(define b (flip 0.3)) => 0 0 0
(define c (flip 0.3)) => 1 0 1
(+ a b c)             => 2 0 1..
```



Conditioning (inference):

```
(query
  (define a (flip 0.3))
  (define b (flip 0.3))
  (define c (flip 0.3))
  (+ a b c)
  (= (+ a b) 1))
```

Query

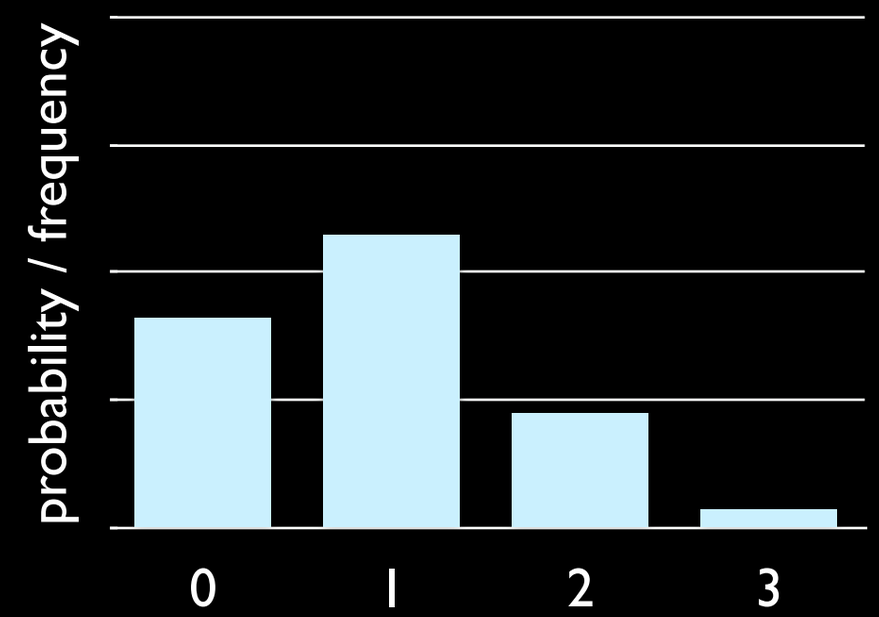
Condition, must be true

Goodman, Mansinghka, Roy, Bonawitz, Tenenabum (2008)

Church

Random primitives:

```
(define a (flip 0.3)) => 1 0 0
(define b (flip 0.3)) => 0 0 0
(define c (flip 0.3)) => 1 0 1
(+ a b c)             => 2 0 1..
```



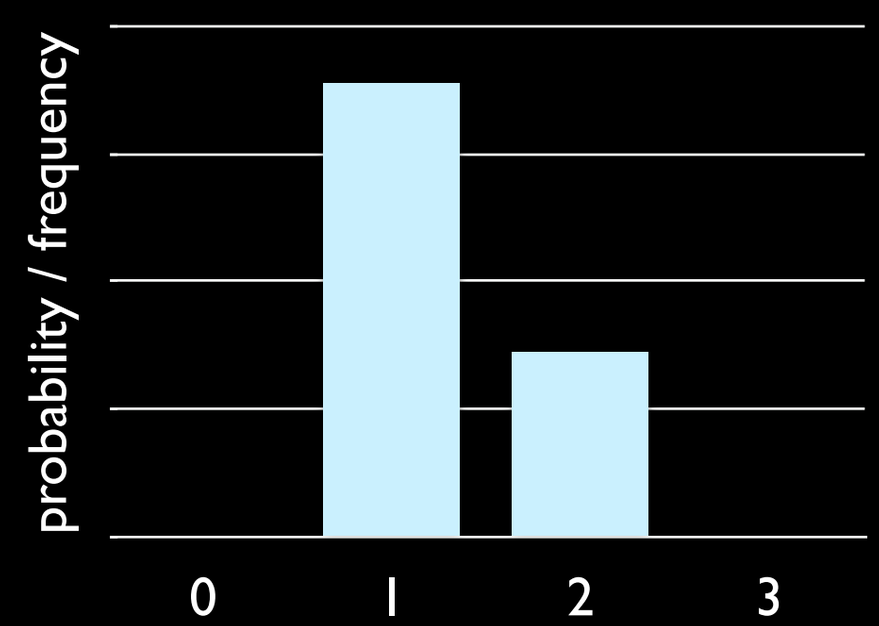
Conditioning (inference):

```
(query
  (define a (flip 0.3))
  (define b (flip 0.3))
  (define c (flip 0.3))
  (+ a b c)
  (= (+ a b) 1))
```

=>

Query

Condition, must be true



Goodman, Mansinghka, Roy, Bonawitz, Tenenabum (2008)

Example: PCFG

```
(define S (λ () (if (flip)
                    (list (S) (S))
                    ‘s)))
```

Example: PCFG

```
(define S (λ () (if (flip)
                    (list (S) (S))
                    ‘s)))
```

```
(define (tree-unfold symbol)
  (if (terminal? symbol)
      symbol
      (map tree-unfold (transition symbol))))
```

PLoT

- Formalizing the PLoT:
  Mental representations (concepts) are functions in a **stochastic** lambda calculus (e.g. Church).
  - Separate the process of inference from representations and the inferences they license (Cf. Marr's levels).
  - (However see our work on inference....)

http://projects.csail.mit.edu/church

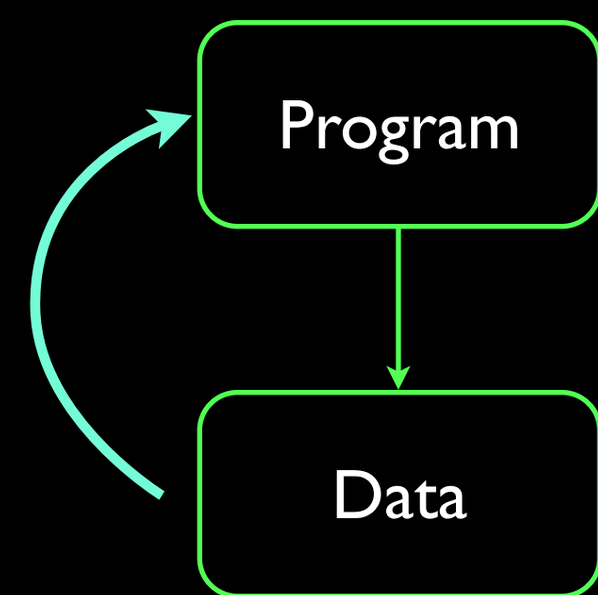
Outline

If concepts are probabilistic programs, then concept learning is *probabilistic program induction*.

Outline

If concepts are probabilistic programs, then concept learning is *probabilistic program induction*.

```
(query
 (define concept (sample-PLoT-expression))
 concept
 (and (= (noisy (sample concept)) obs1)
      (= (noisy (sample concept)) obs2)
      ...))
```

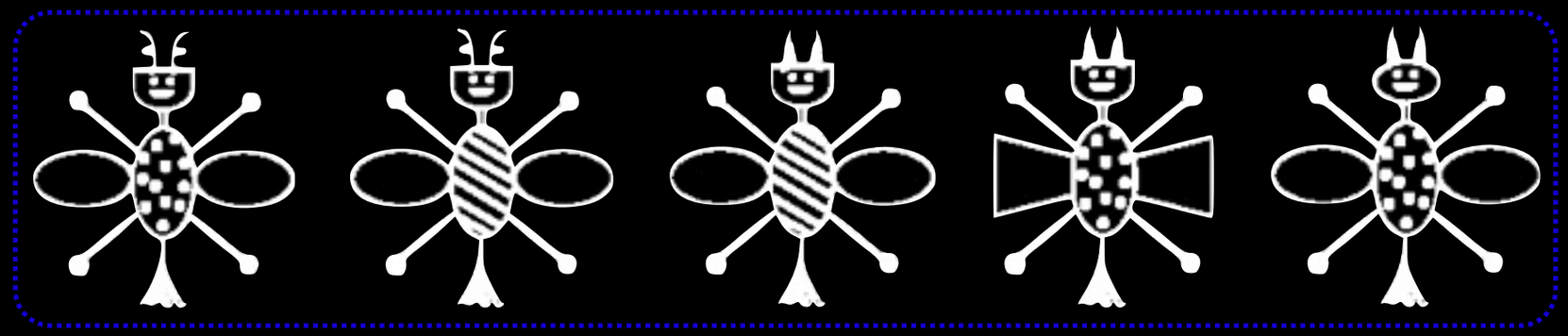


Categorization

Medin & Schaffer (1978):

Categorization

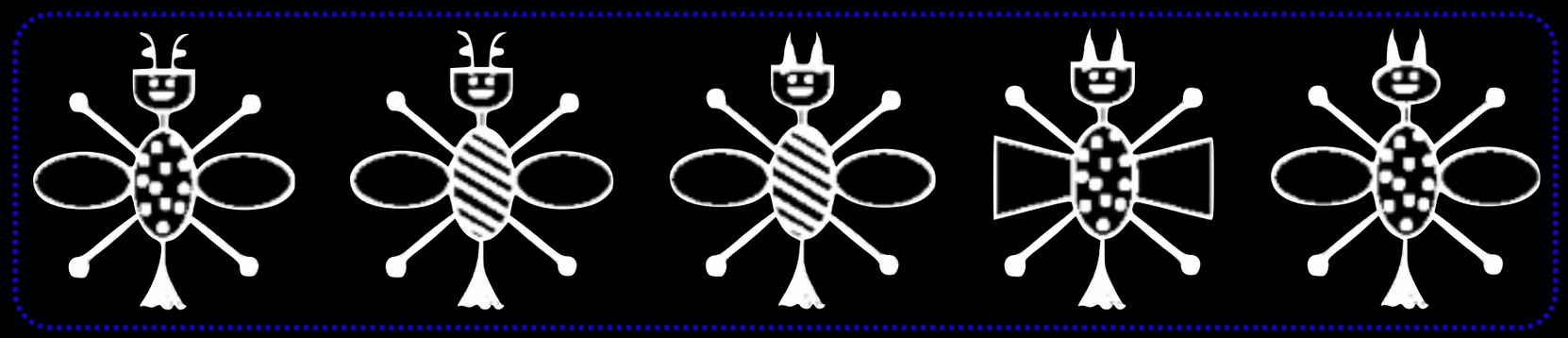
Medin & Schaffer (1978):



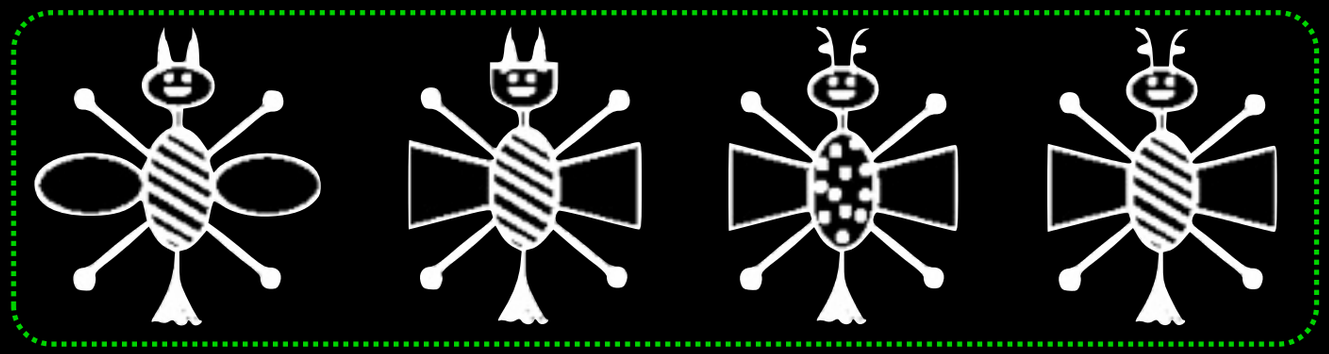
"These are **Feps**"

Categorization

Medin & Schaffer (1978):



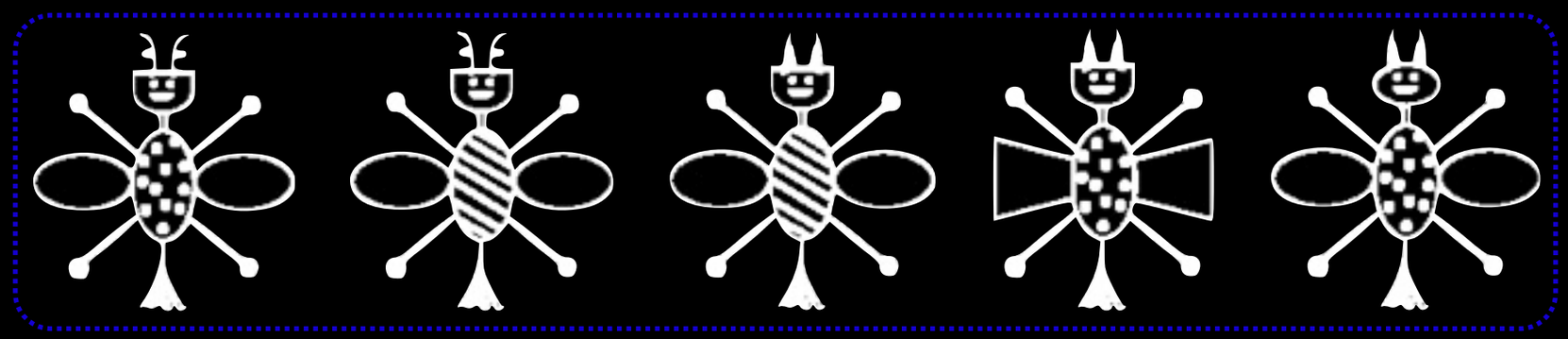
"These are **Feps**"



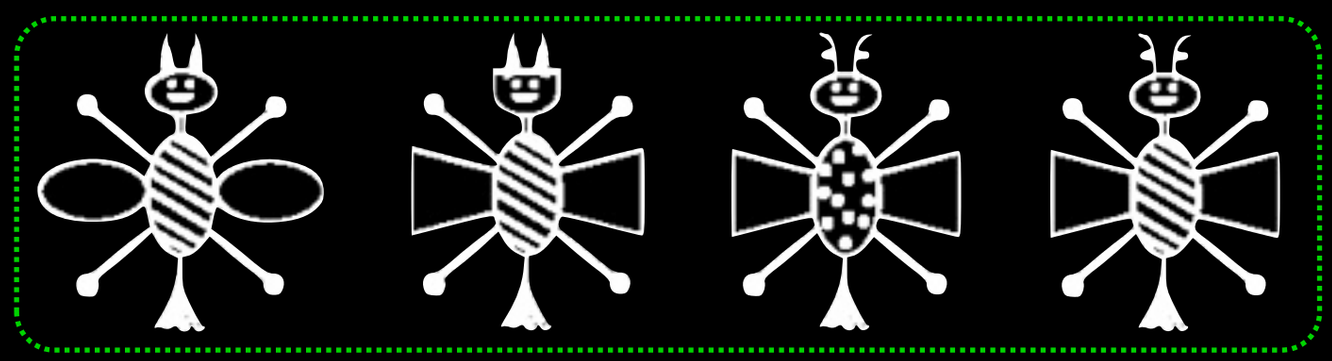
"These are **not Feps**"

Categorization

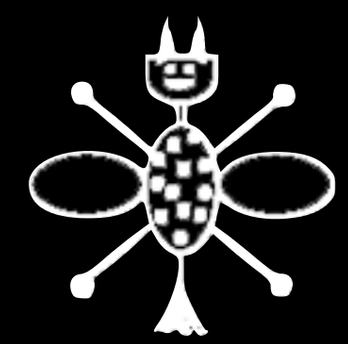
Medin & Schaffer (1978):



"These are **Feps**"



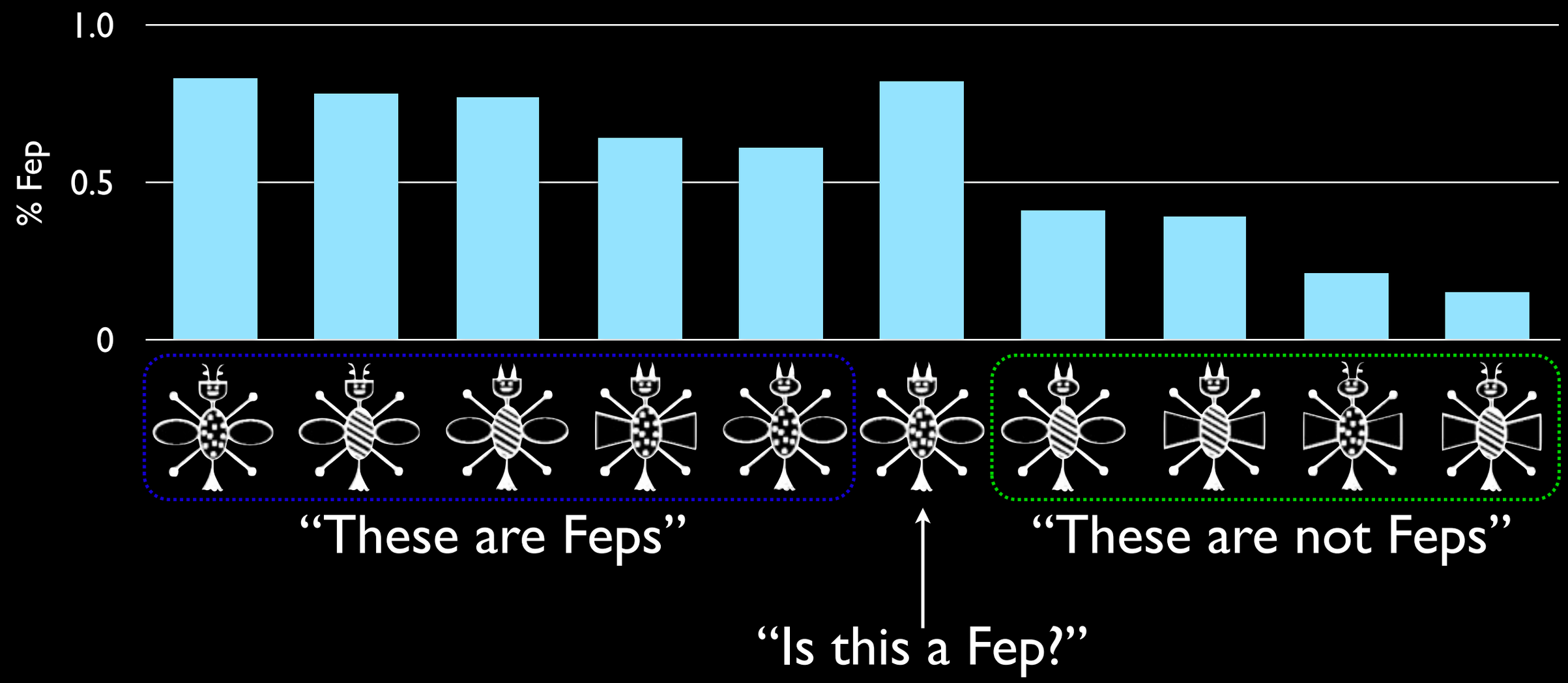
"These are **not Feps**"



"Is this a Fep?"

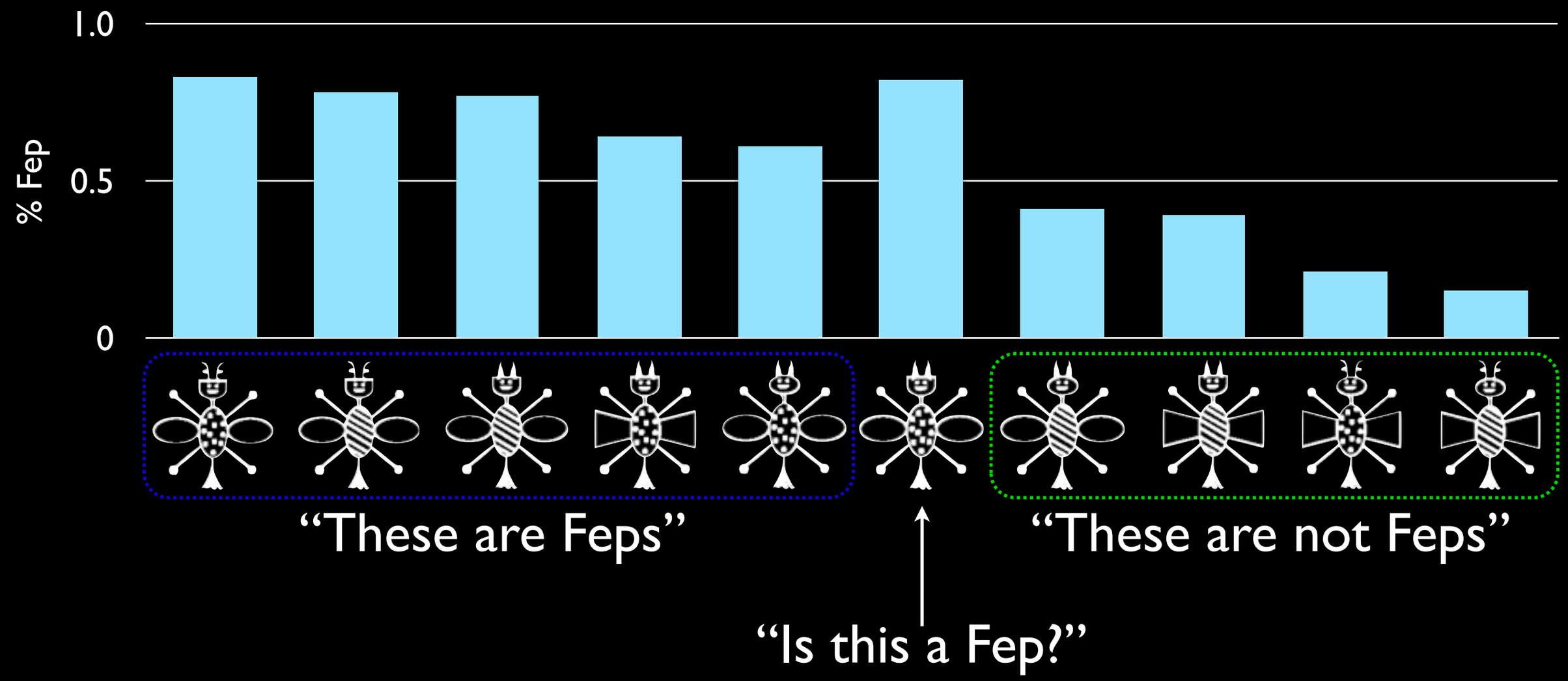
Categorization

Medin & Schaffer, 1978 (data from Nosofsky, et al.,1994):



Categorization

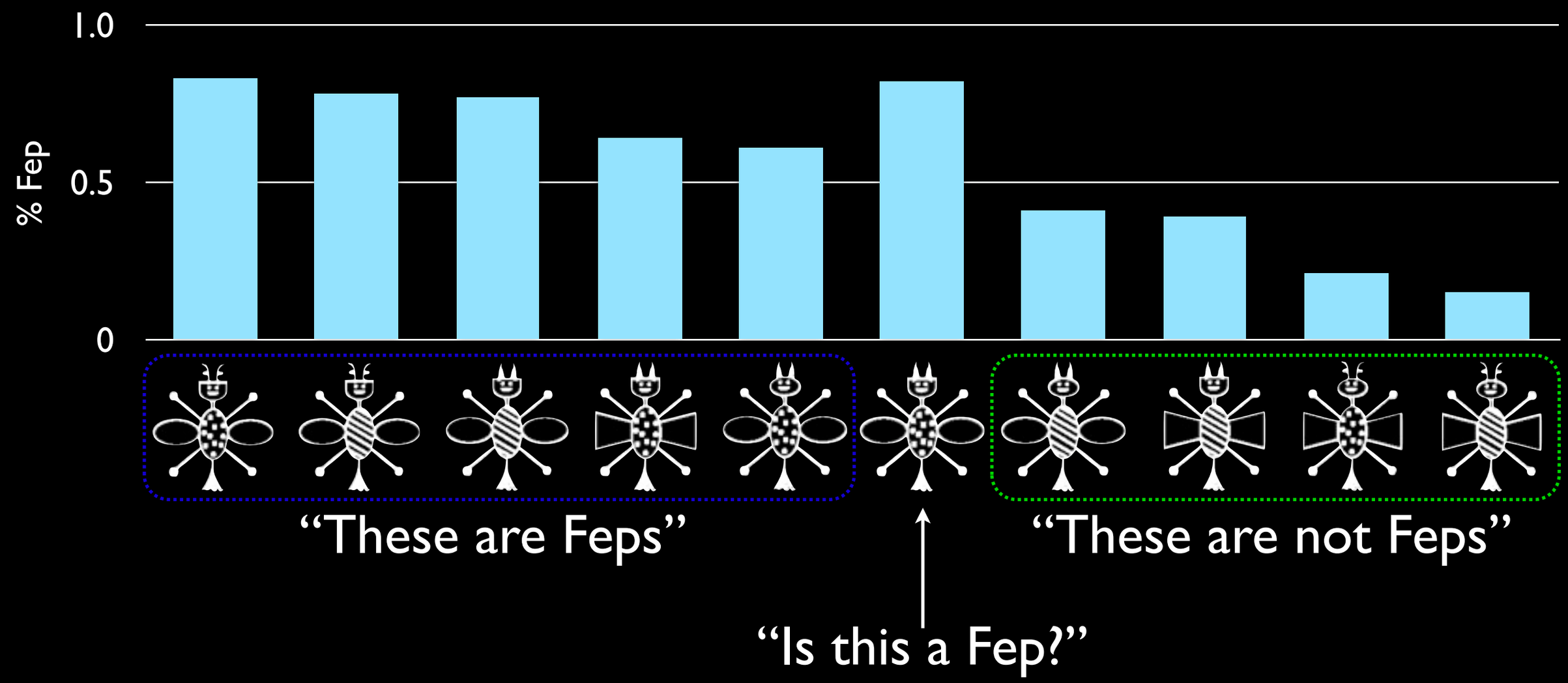
Medin & Schaffer, 1978 (data from Nosofsky, et al.,1994):



- Graded judgements

Categorization

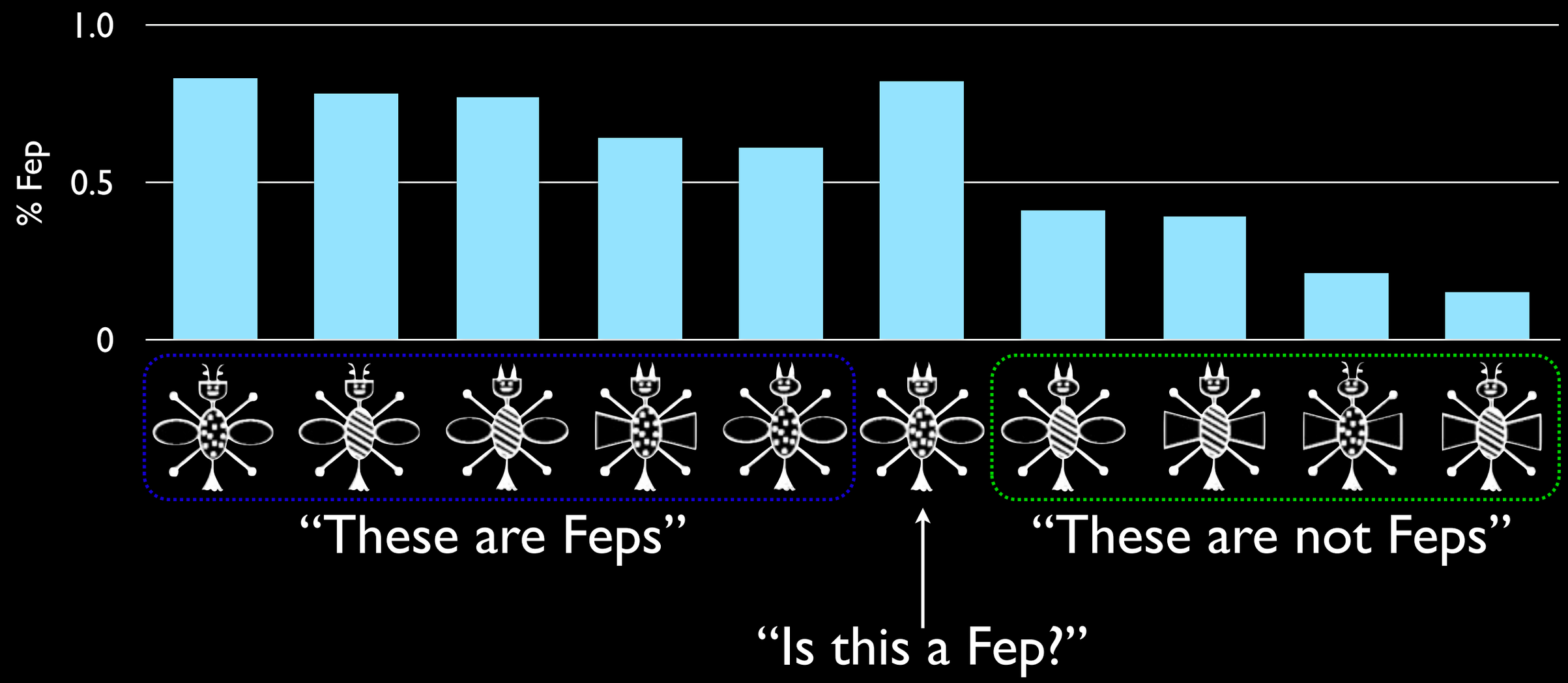
Medin & Schaffer, 1978 (data from Nosofsky, et al.,1994):



- Graded judgements
- Typicality

Categorization

Medin & Schaffer, 1978 (data from Nosofsky, et al.,1994):

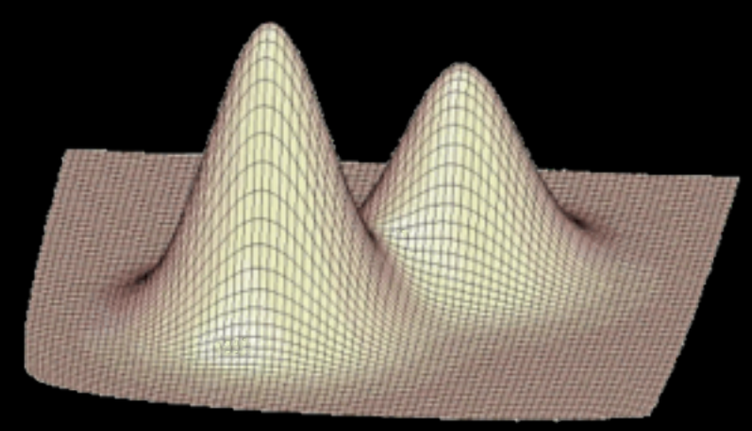


- Graded judgements
- Typicality
- Prototype enhancement

Categorization

- Rule-based category learning:
  - Infinitely many concepts formed compositionally.

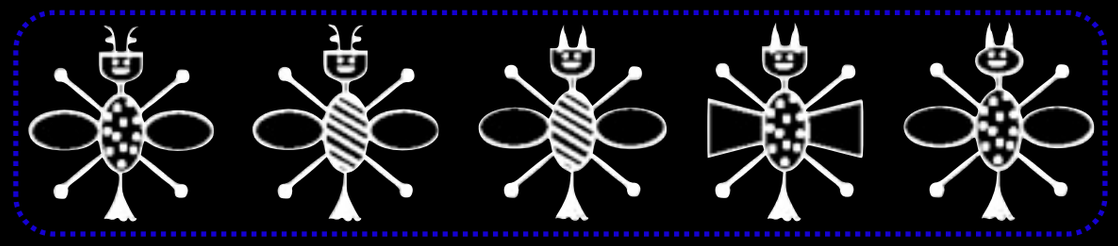
- Statistical category learning:
  - Graded inferences from sparse, noisy evidence.



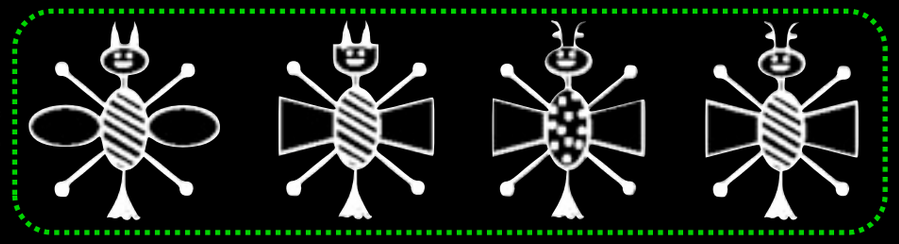
Generating rules

"It's a Fep if it has flat head and round wings"

Feps:



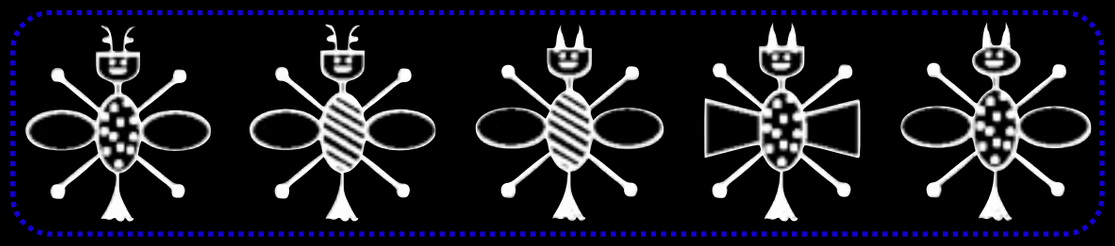
non-Feps:



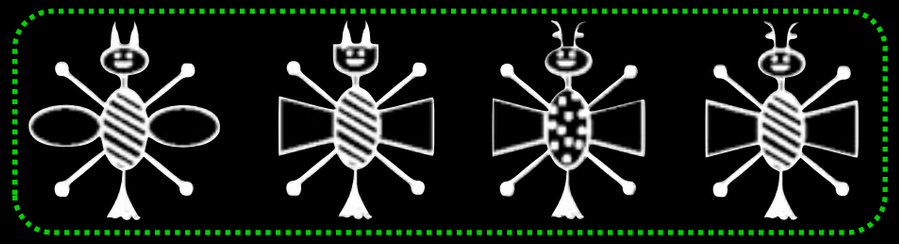
Generating rules

```
(define fep?
 (λ (x)
  (and (flat-head x)
       (round-wings x))))
```

Feps:



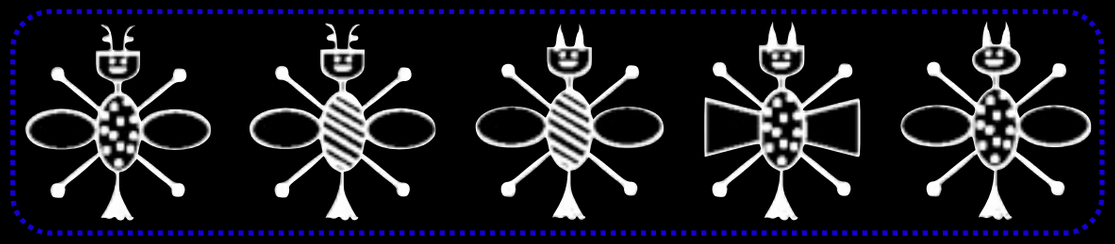
non-Feps:



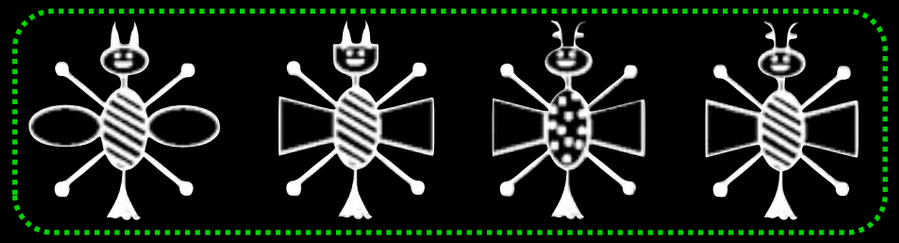
Generating rules

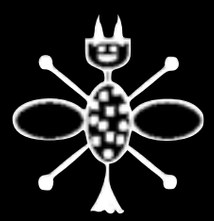
```
(define fep?
 (λ (x)
  (and (flat-head x)
       (round-wings x))))
```

Feps:



non-Feps:

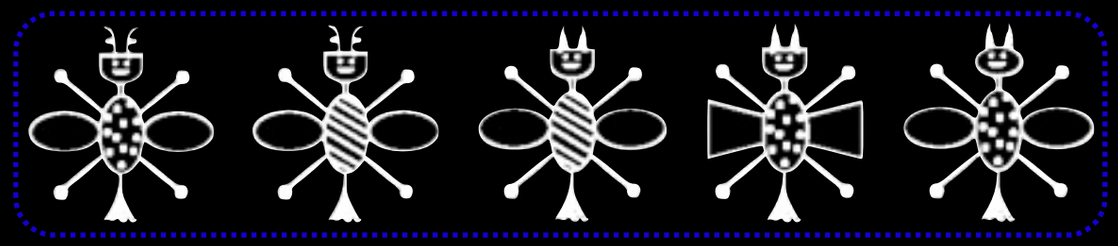


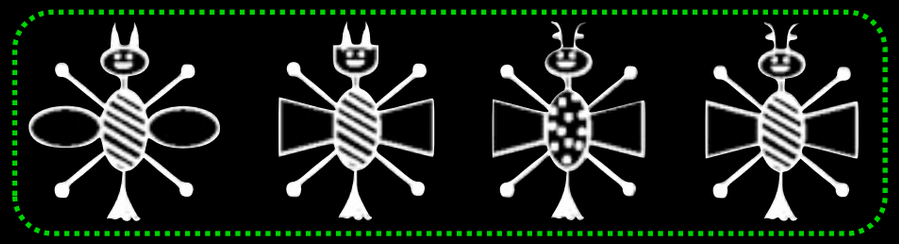
(fep? )

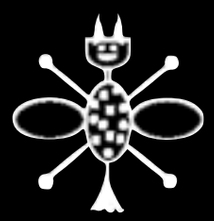
=> true

Generating rules

```
(define fep?
 (λ (x)
  (and (flat-head x)
       (round-wings x))))
```

Feps: 

non-Feps: 

(fep? )

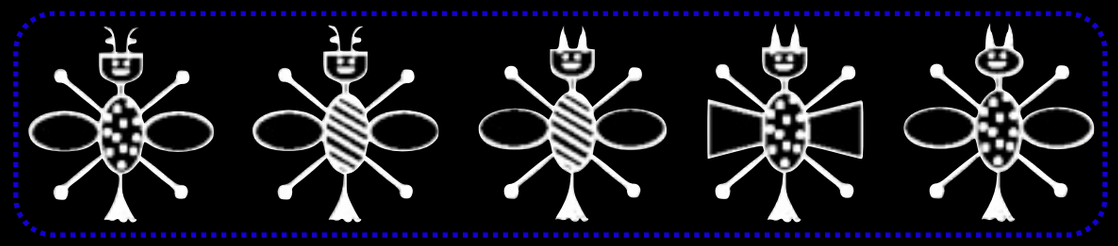
=> true

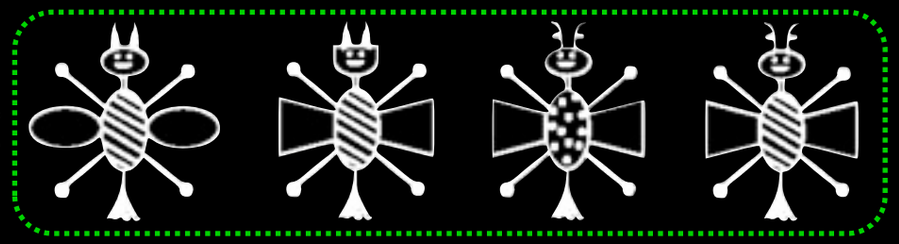
```
(define rule-generator
 (λ ()
  (if (flip 0.3)
      (sample-feature)
      (combine-rules (sample-feature)
                     (rule-generator)))))
```

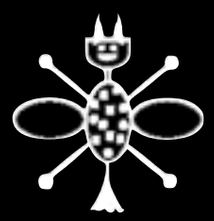
```
(define combine-rules
 (λ (r1 r2)
  (λ (x) (and (r1 x) (r2 x)))))
```

Generating rules

```
(define fep?
 (λ (x)
  (and (flat-head x)
       (round-wings x))))
```

Feps: 

non-Feps: 

(fep? )

=> true

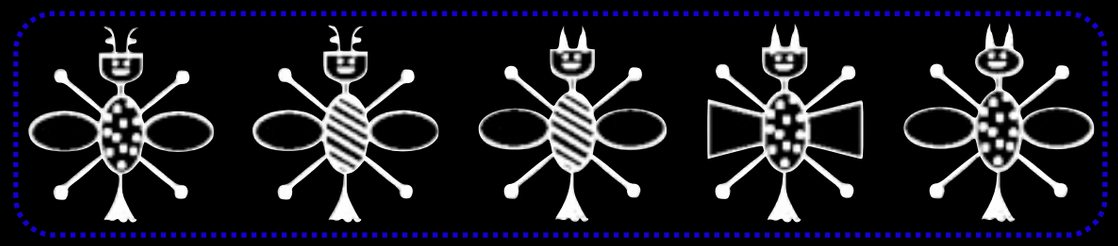
```
(define rule-generator
 (λ ()
  (if (flip 0.3)
      (sample-feature)
      (combine-rules (sample-feature)
                     (rule-generator)))))
```

```
(define combine-rules
 (λ (r1 r2)
  (λ (x) (and (r1 x) (r2 x)))))
```

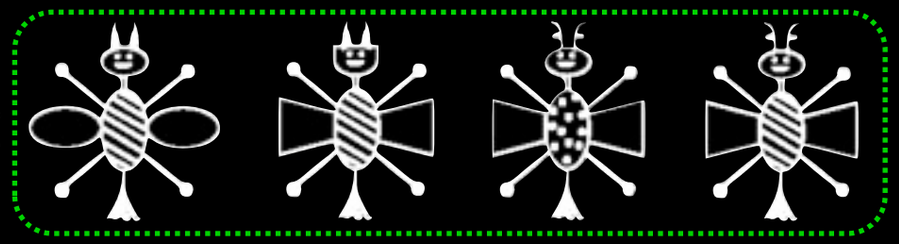
Generating rules

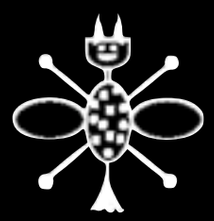
```
(define fep?
 (λ (x)
  (and (flat-head x)
       (round-wings x))))
```

Feps:



non-Feps:



(fep? )

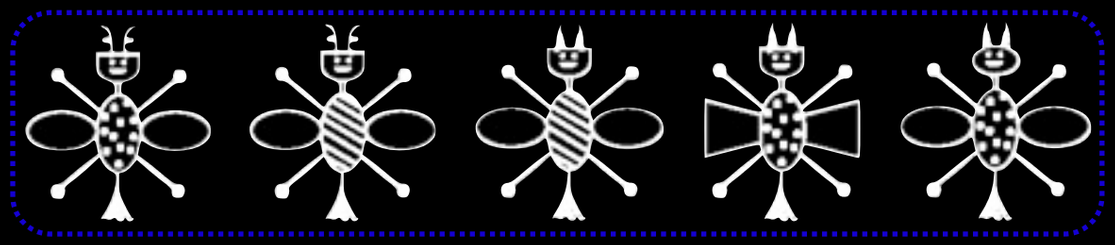
=> true

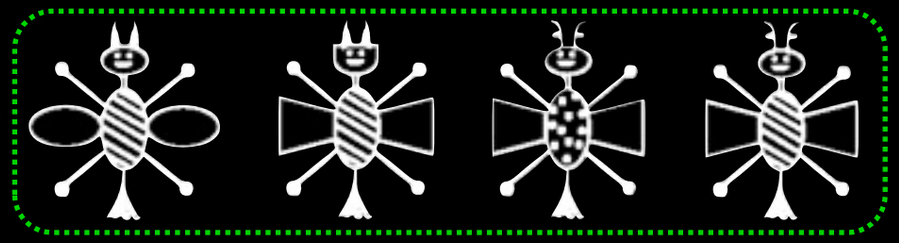
```
(define rule-generator
 (λ ()
  (if (flip 0.3)
      (sample-feature)
      (combine-rules (sample-feature)
                     (rule-generator)))))
```

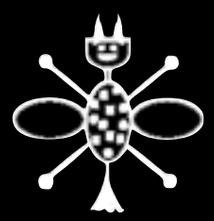
```
(define combine-rules
 (λ (r1 r2)
  (λ (x) (and (r1 x) (r2 x)))))
```

Generating rules

```
(define fep?
 (λ (x)
  (and (flat-head x)
       (round-wings x))))
```

Feps: 

non-Feps: 

(fep? )

=> true

```
(define rule-generator
 (λ ()
  (if (flip 0.3)
      (sample-feature)
      (combine-rules (sample-feature)
                     (rule-generator)))))
```

```
(define combine-rules
 (λ (r1 r2)
  (λ (x) (and (r1 x) (r2 x)))))
```

- Longer rules have lower probability (Occam's razor).

Generating rules

Put uncertainty over rule probabilities:

```
(define rule-prob (uniform 0 1))
(define rule-generator
 (λ ()
  (if (flip rule-prob)
      ...
```

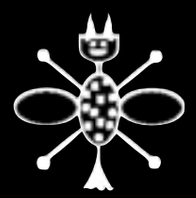
Generate disjunctive normal form (DNF) rules:

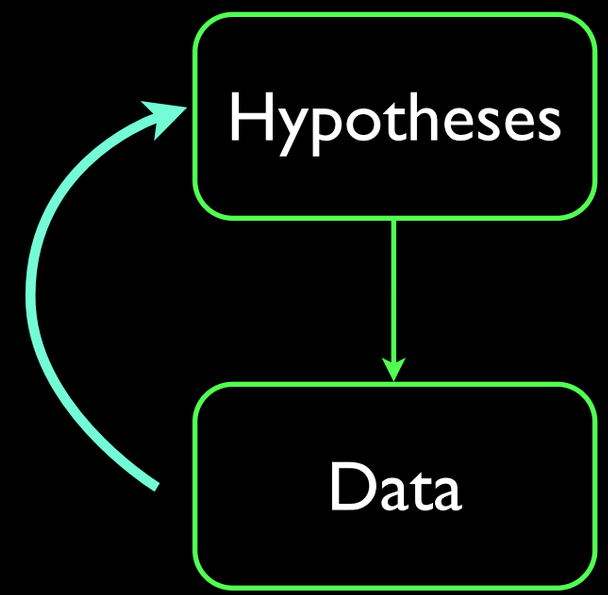
```
(define fep?
 (λ (x)
  (and (or (flat-head x) ...)
       (or (round-wings x) ...)
       ...)))
```

The general idea:
grammar-based induction.

Inducing rules

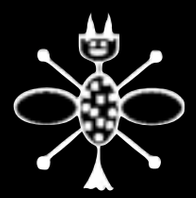
Inference:

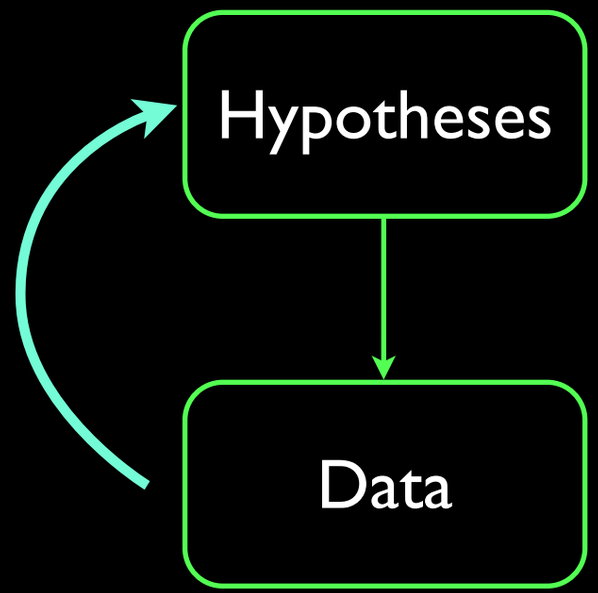
```
(query
 (define rule (rule-generator))
 (rule  )
 (and (= (rule  ) true)
      (= (rule  ) false)
      ...))
```



Inducing rules

Inference:

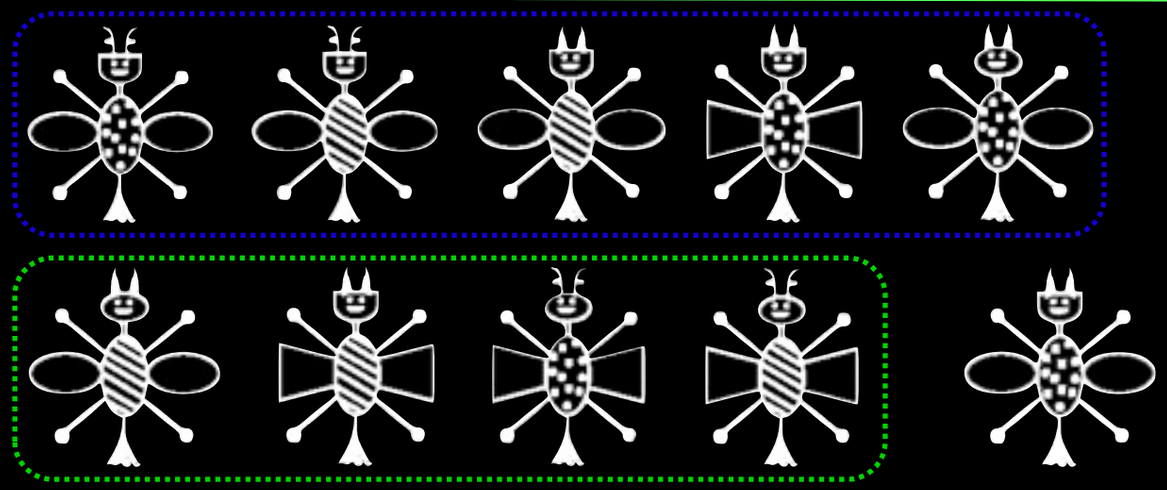
```
(query
 (define rule (rule-generator))
 (rule  )
 (and (= (noisy (rule  )) true)
      (= (noisy (rule  )) false)
      ...))
```

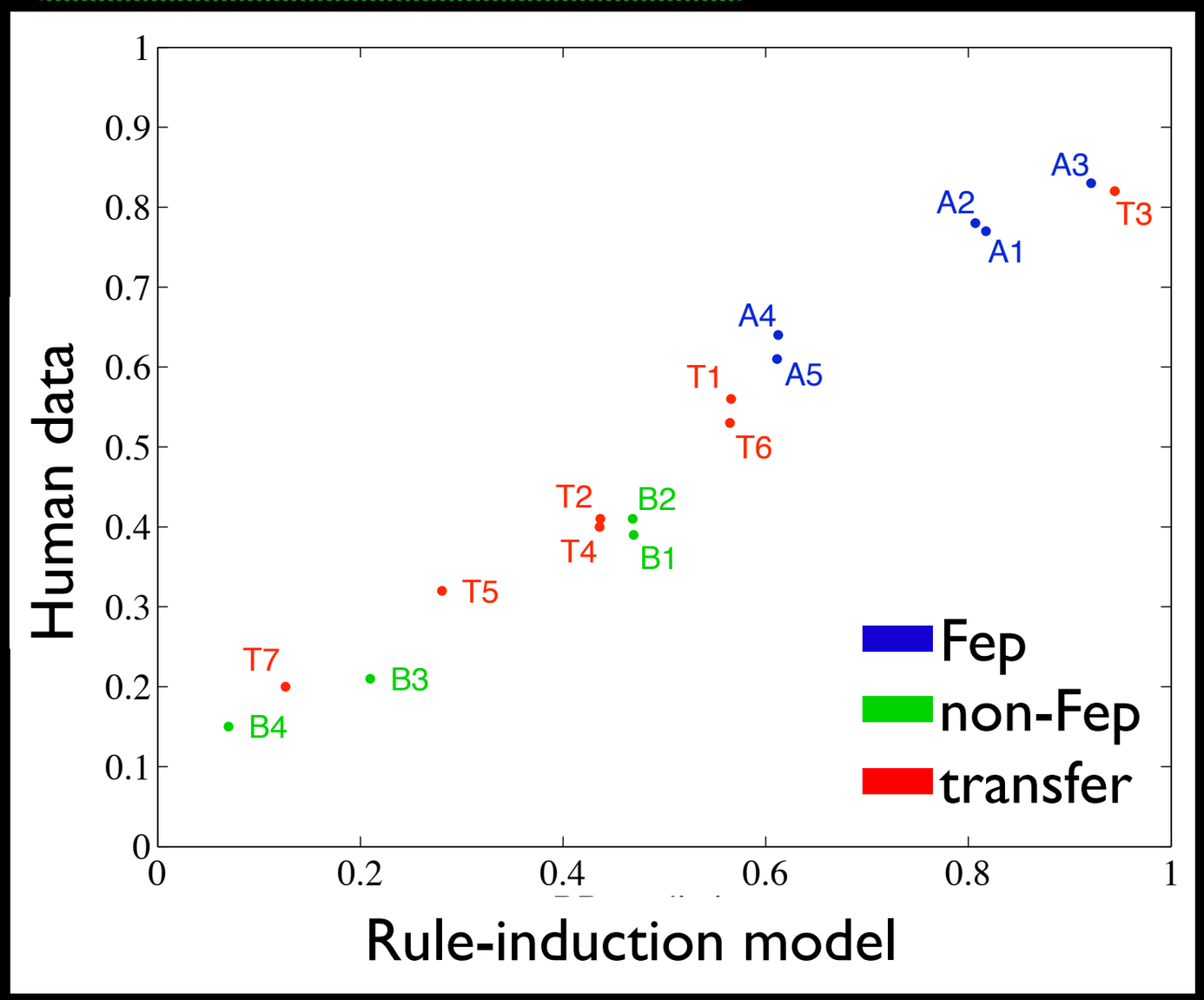


Observation noise:

```
(define noisy
 (λ (bit) (if (flip b) bit (not bit))))
```

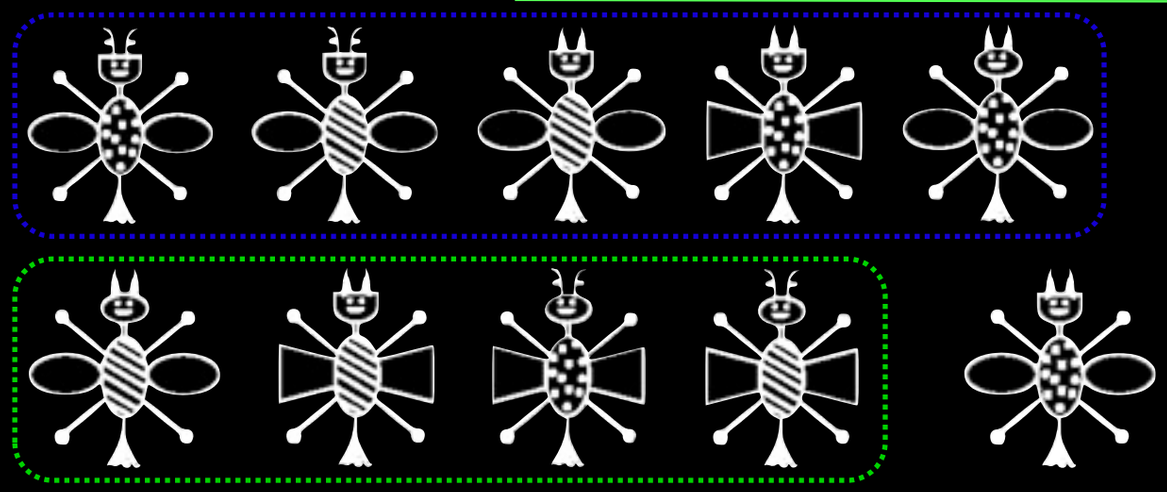
Categorization

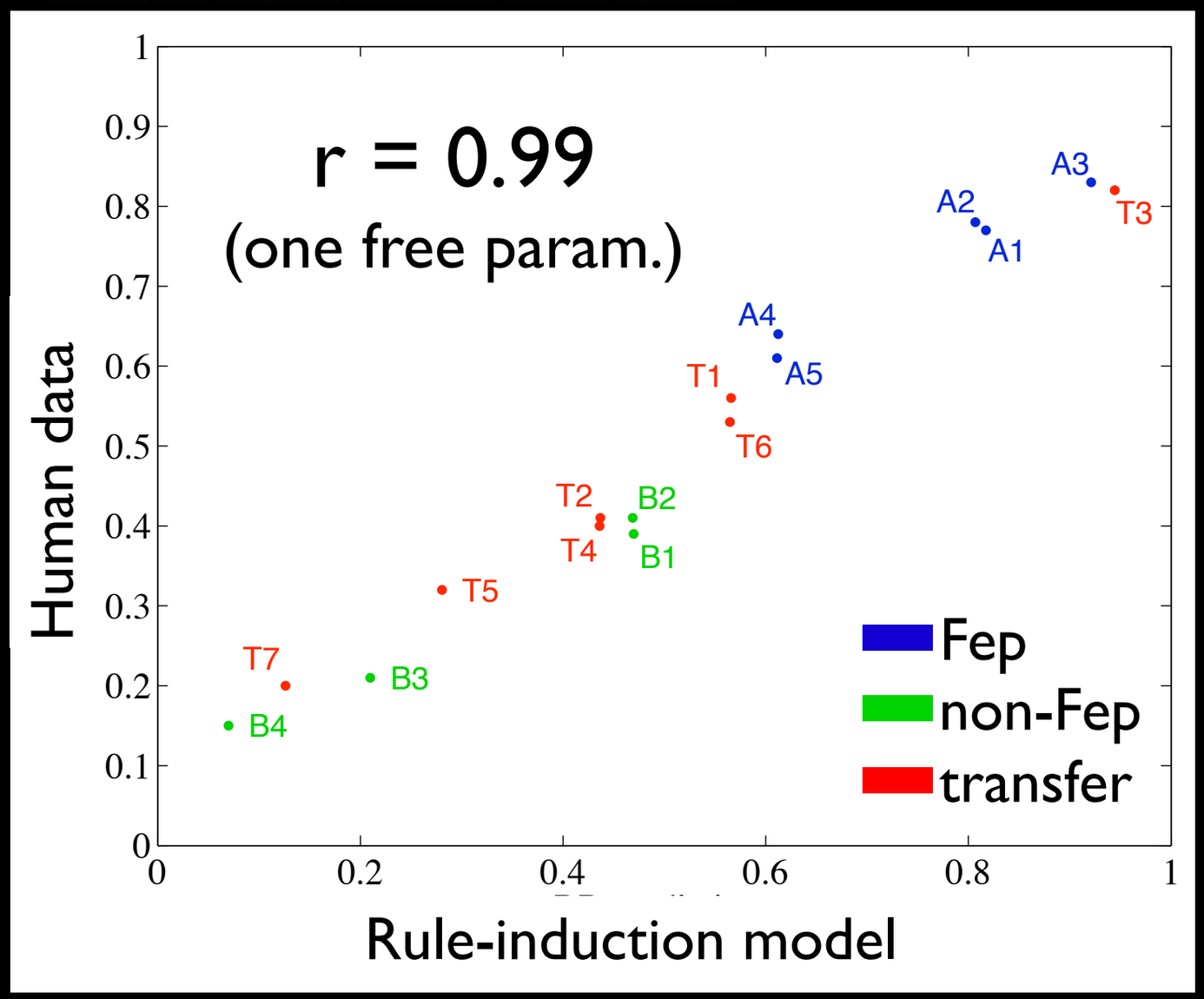




Goodman, et al.
(2007, 2008a, 2008b)

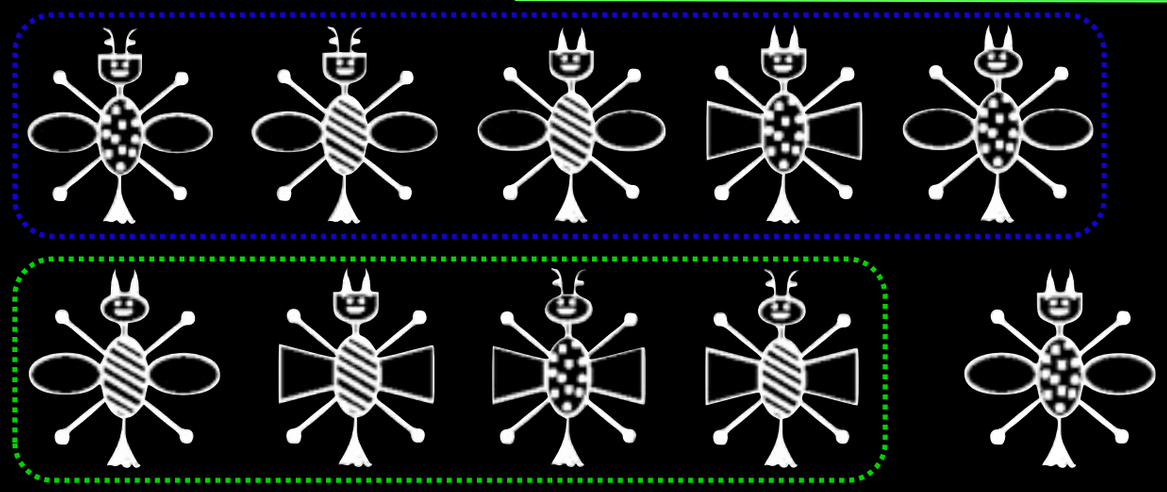
Categorization

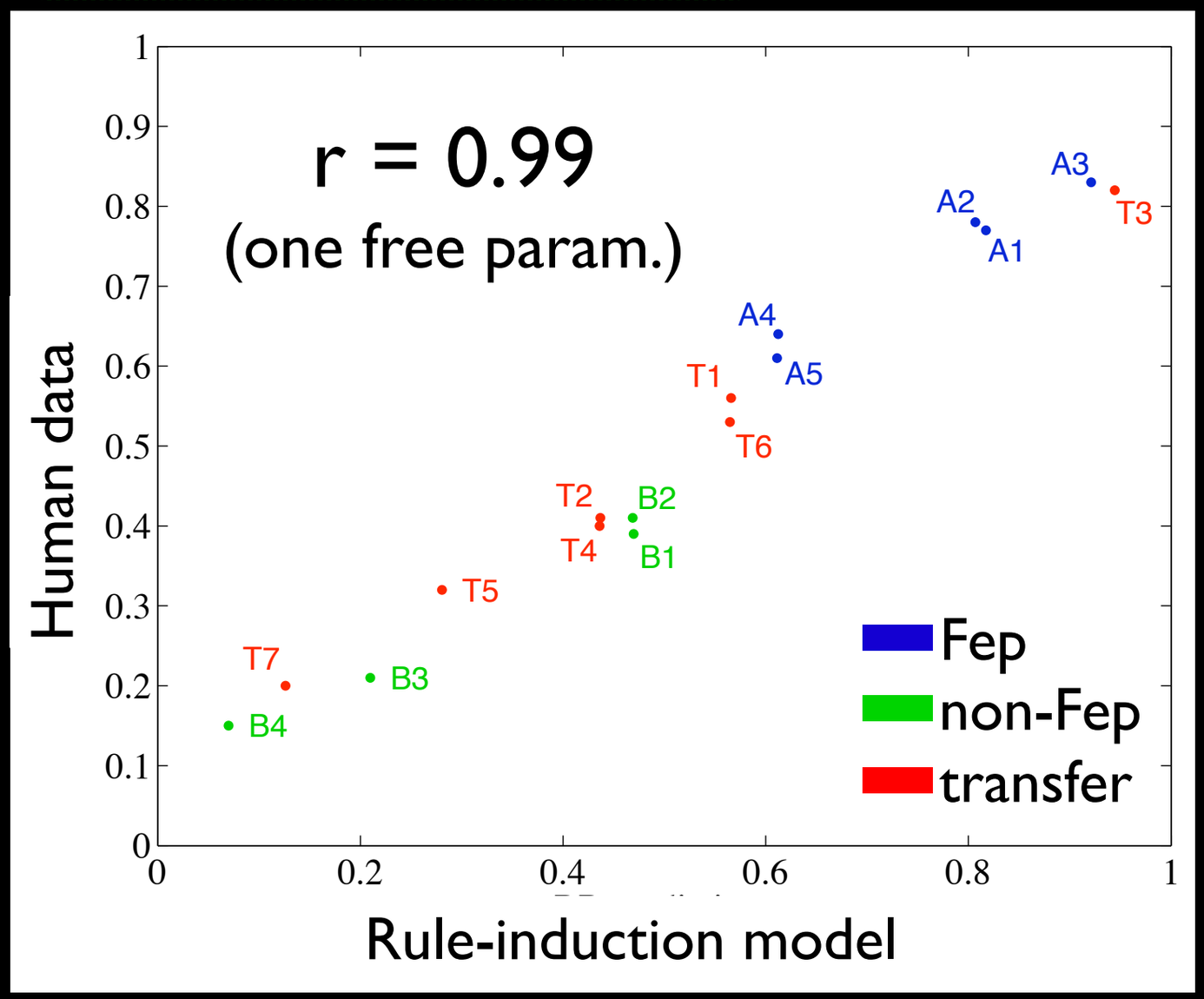




Goodman, et al.
(2007, 2008a, 2008b)

Categorization

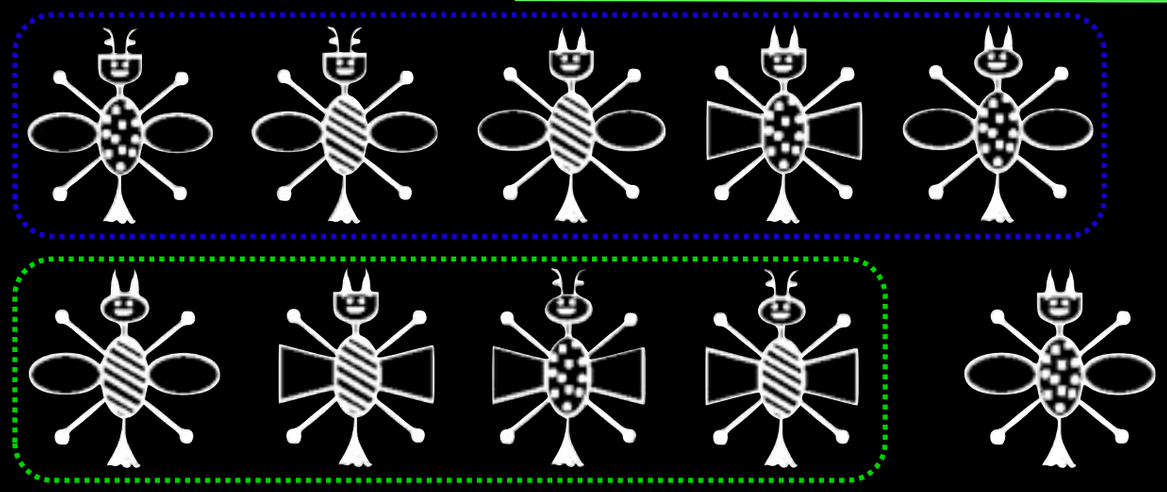


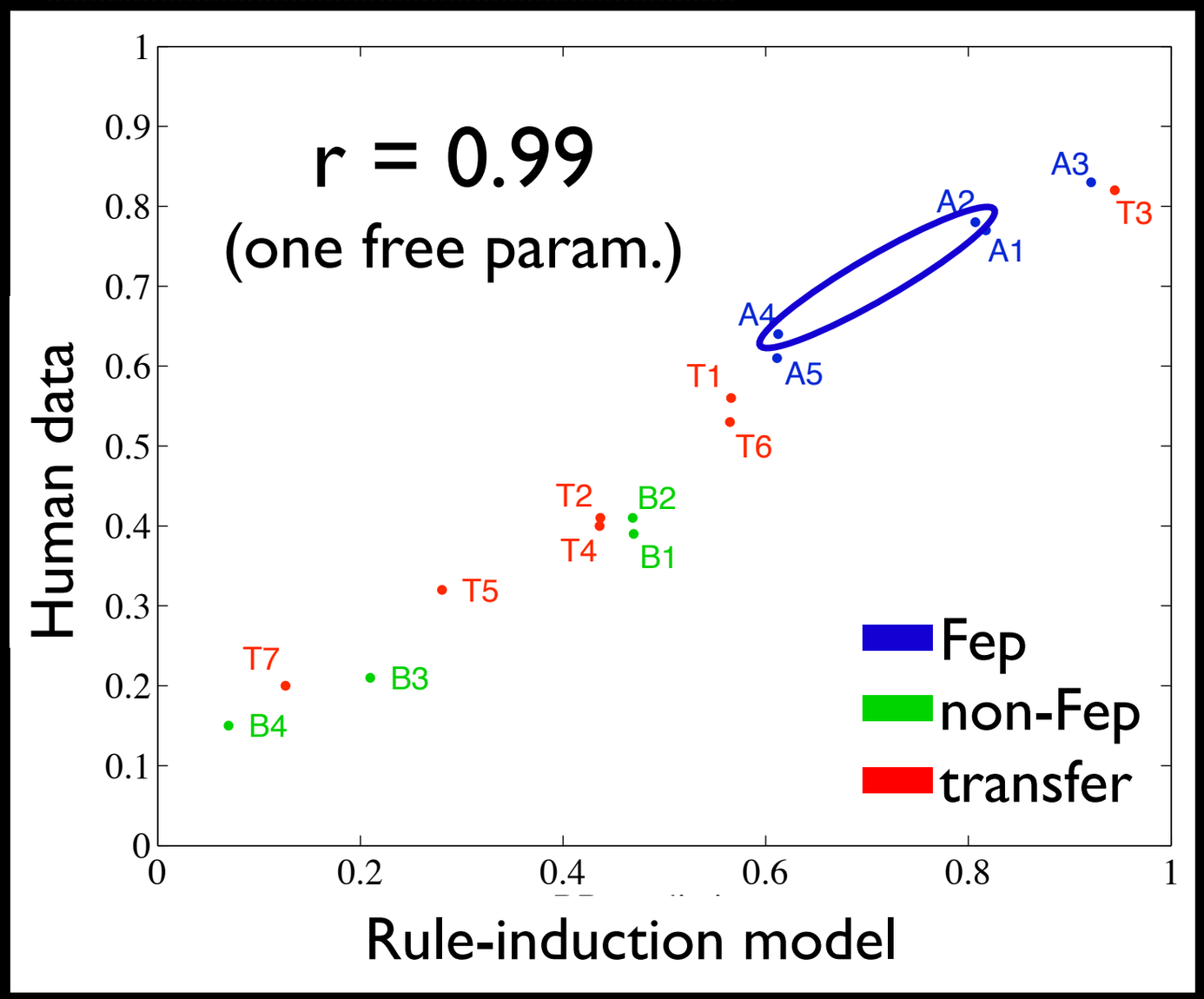


- Graded judgments

Goodman, et al.
(2007, 2008a, 2008b)

Categorization

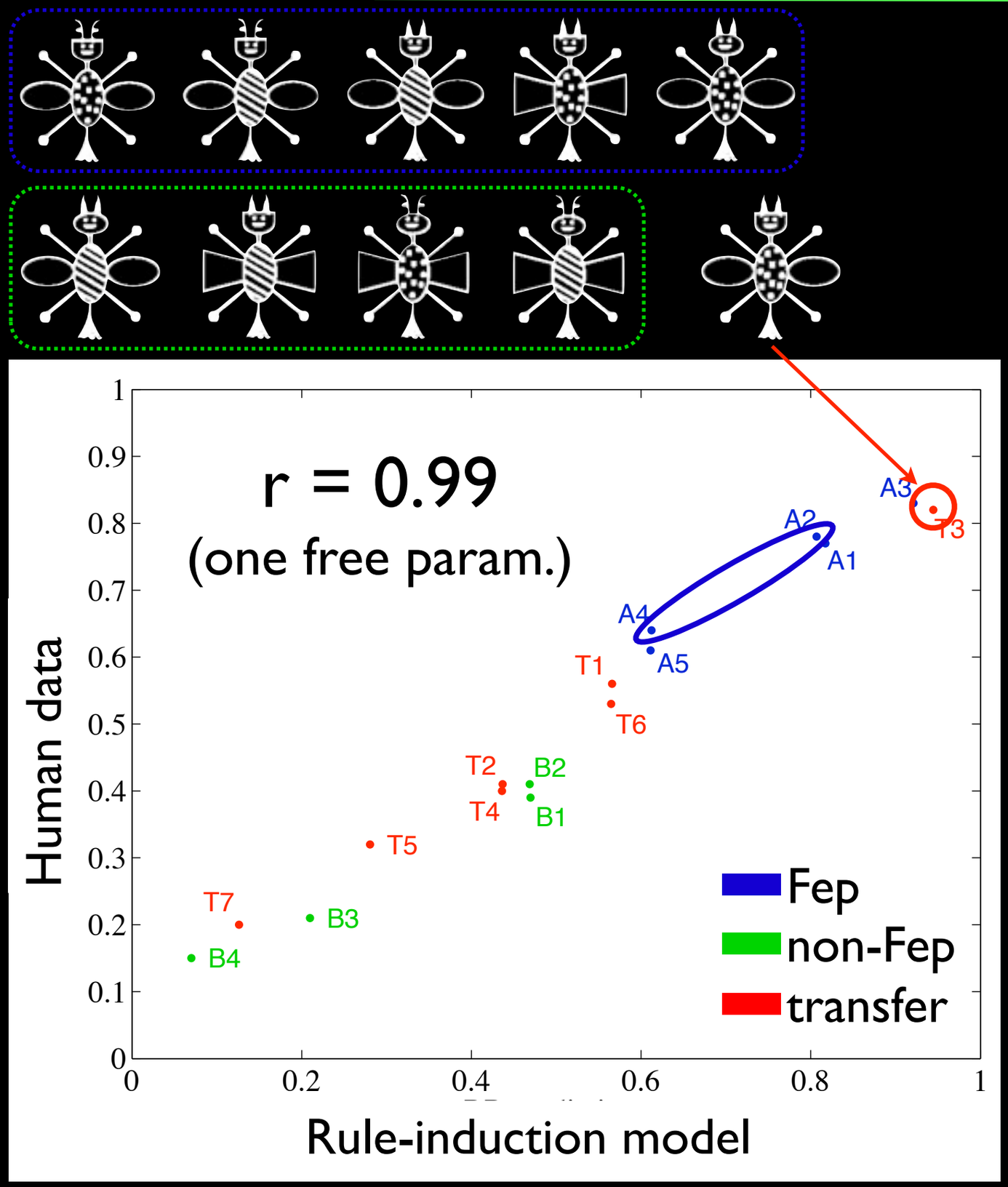




- Graded judgments
- Typicality

Goodman, et al.
(2007, 2008a, 2008b)

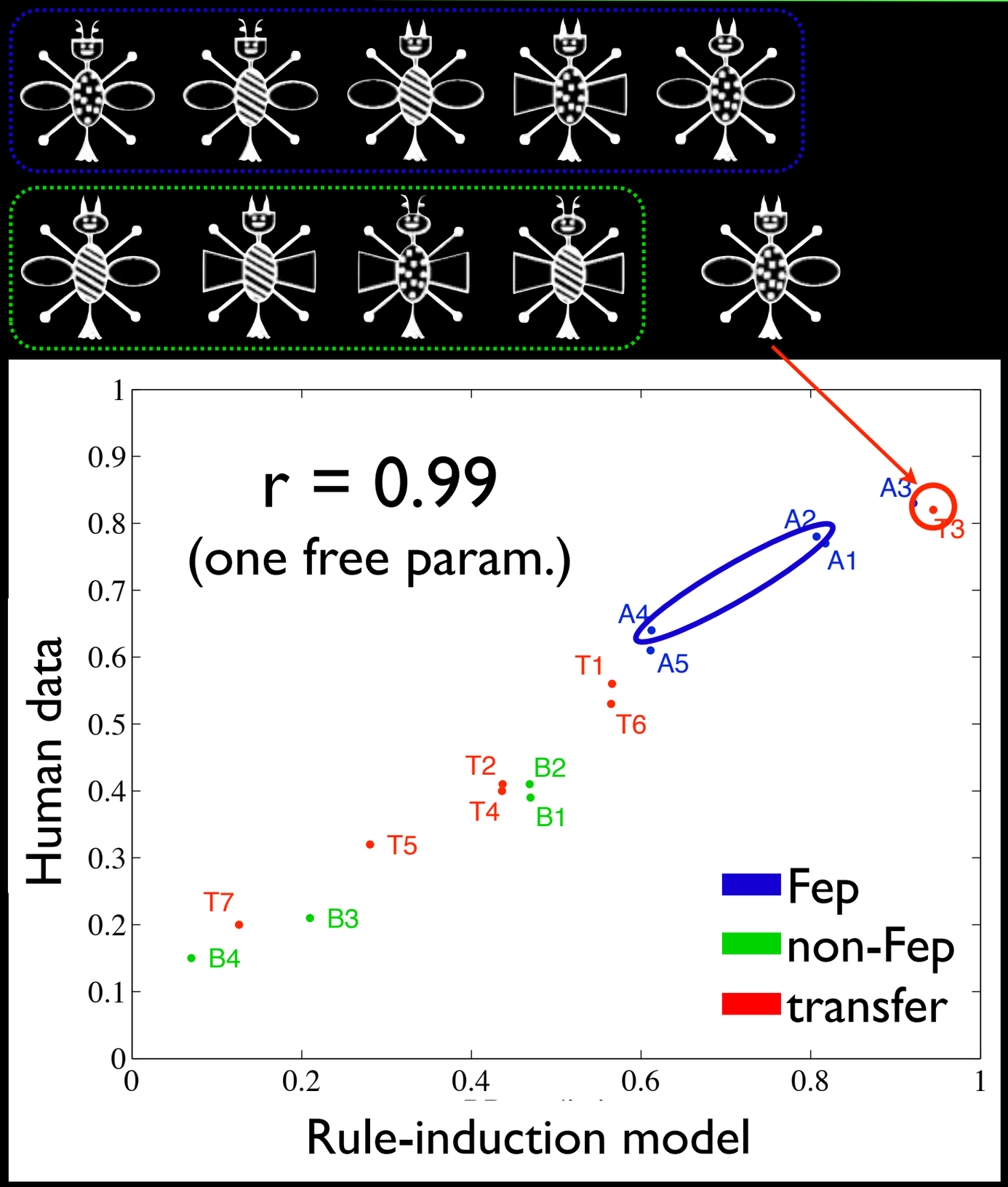
Categorization



- Graded judgments
- Typicality
- Prototype enhancement

Goodman, et al. (2007, 2008a, 2008b)

Categorization



- Graded judgments
- Typicality
- Prototype enhancement
- Selective attention

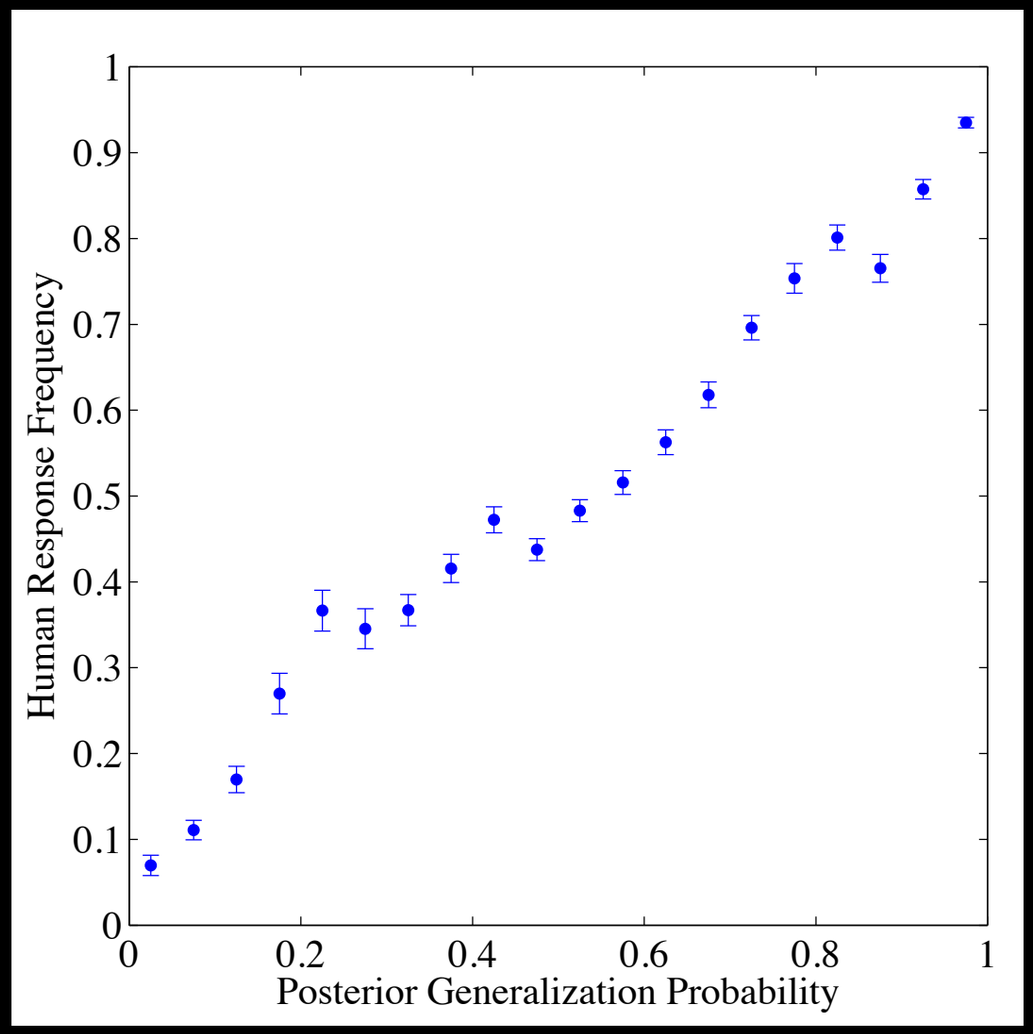
Goodman, et al. (2007, 2008a, 2008b)

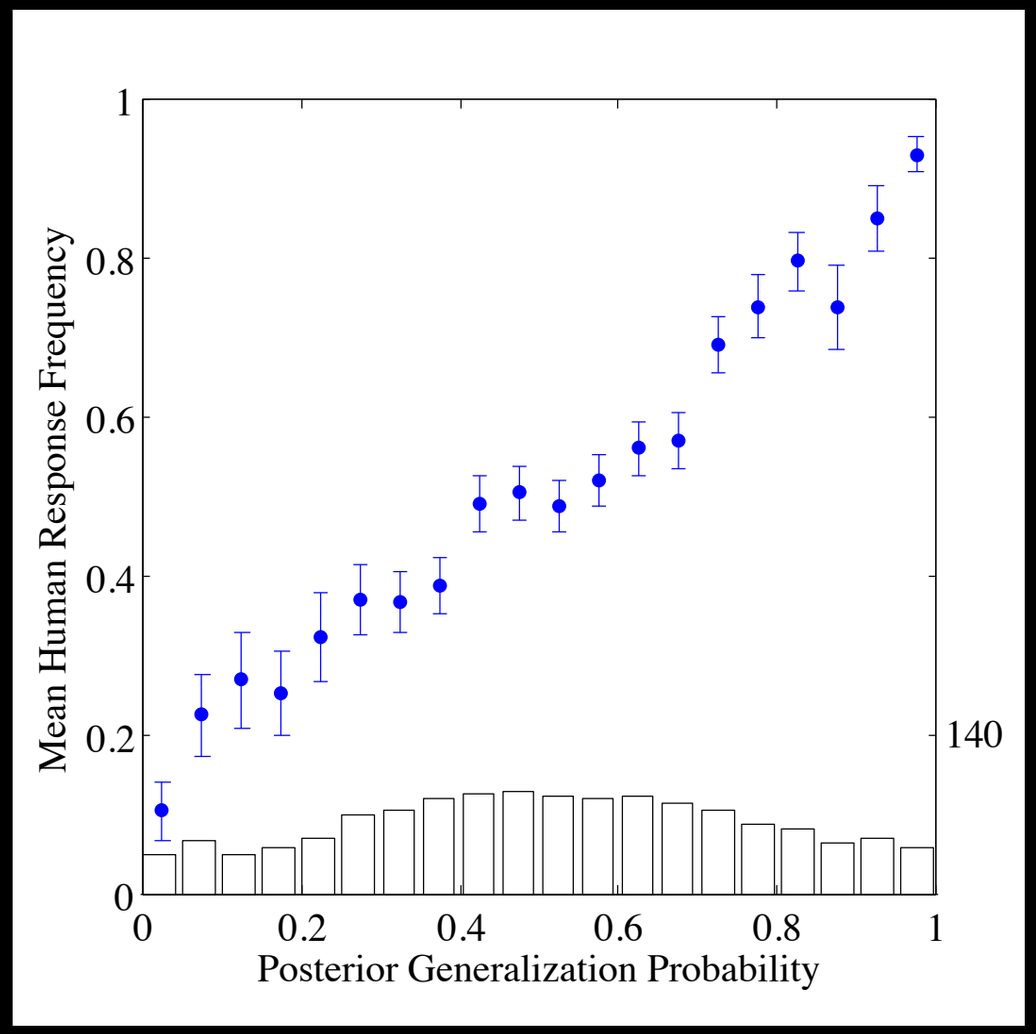
Evaluating languages

- Induction to the language generated by the DNF grammar explains important phenomena (and fits relevant data).
- But is this the *right* LoT?
  - Test on wider data set?
  - Compare to other propositional languages?

Broader test

- 7 Boolean features.
- 43 randomly generated concepts (3-6 pos. + 2 neg. exs)
- 128 judgements (~122 transfer questions)





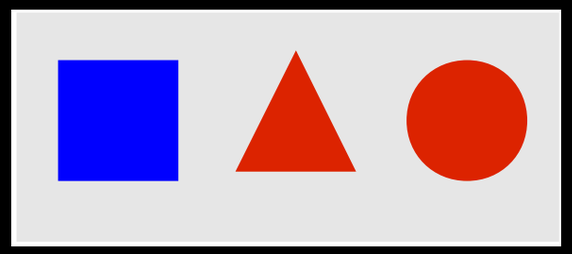
Goodman, et al (2008)

Evaluating languages

- High-throughput MTurk experiment.
  - 108 concepts,
    - Boolean (*circle or red*)
    - Context-dependent ("Determiners")
      (*unique largest , exists another with same shape*)
  - 2 orders per concept,
  - 1596 participants.

Piantadosi, Goodman, Tenenbaum (in prep)

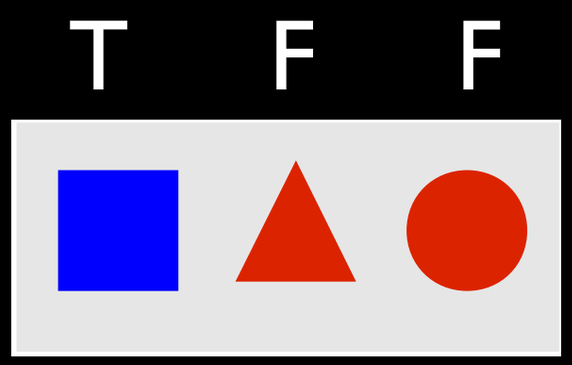
Evaluating languages



- High-throughput MTurk experiment.
  - 108 concepts,
    - Boolean (*circle or red*)
    - Context-dependent ("Determiners")
      (*unique largest , exists another with same shape*)
  - 2 orders per concept,
  - 1596 participants.

Piantadosi, Goodman, Tenenbaum (in prep)

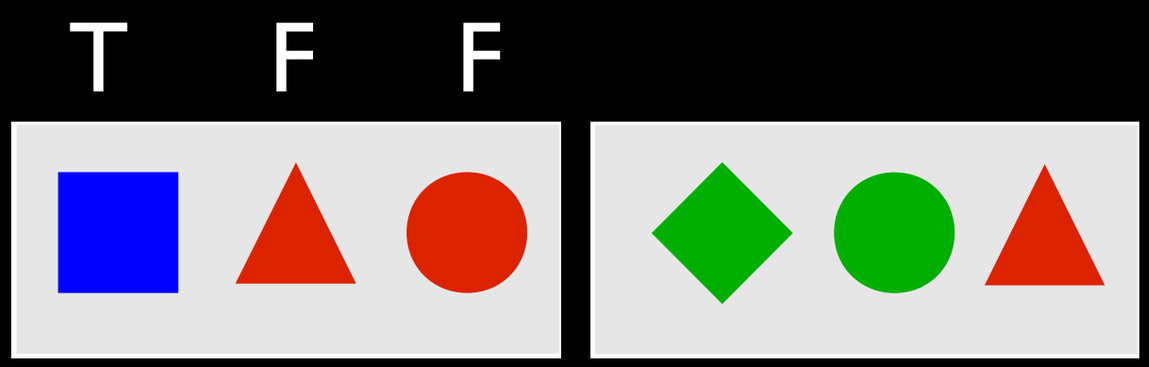
Evaluating languages



- High-throughput MTurk experiment.
  - 108 concepts,
    - Boolean (*circle or red*)
    - Context-dependent ("Determiners") (*unique largest , exists another with same shape*)
  - 2 orders per concept,
  - 1596 participants.

Piantadosi, Goodman, Tenenbaum (in prep)

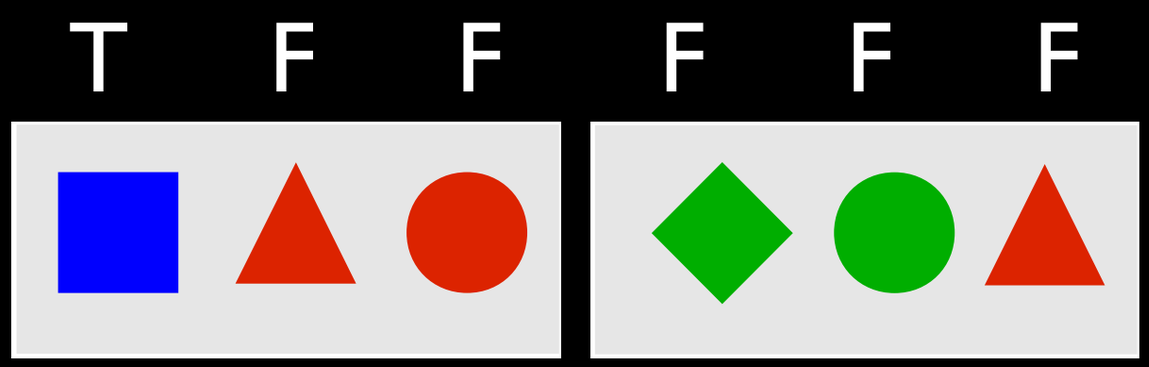
Evaluating languages



- High-throughput MTurk experiment.
  - 108 concepts,
    - Boolean (*circle or red*)
    - Context-dependent ("Determiners")
      (*unique largest , exists another with same shape*)
  - 2 orders per concept,
  - 1596 participants.

Piantadosi, Goodman, Tenenbaum (in prep)

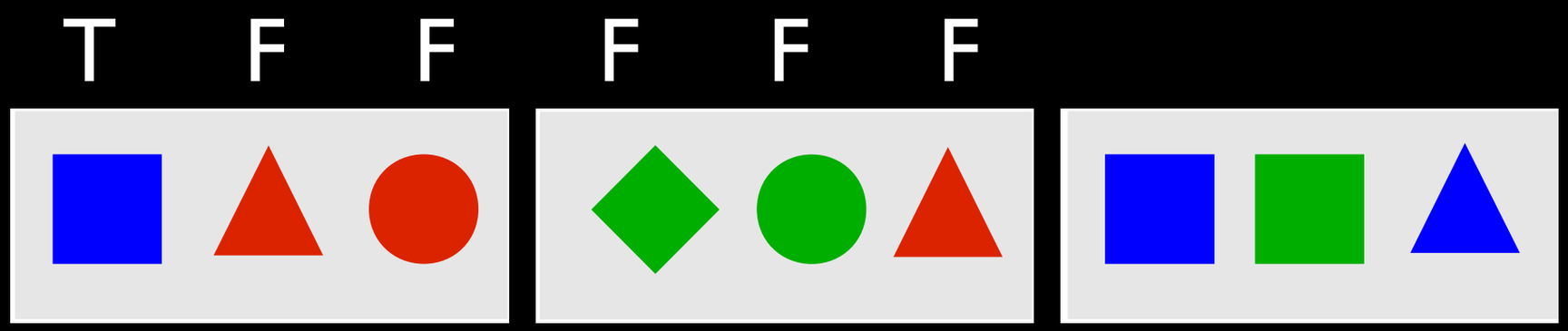
Evaluating languages



- High-throughput MTurk experiment.
  - 108 concepts,
    - Boolean (*circle or red*)
    - Context-dependent ("Determiners") (*unique largest , exists another with same shape*)
  - 2 orders per concept,
  - 1596 participants.

Piantadosi, Goodman, Tenenbaum (in prep)

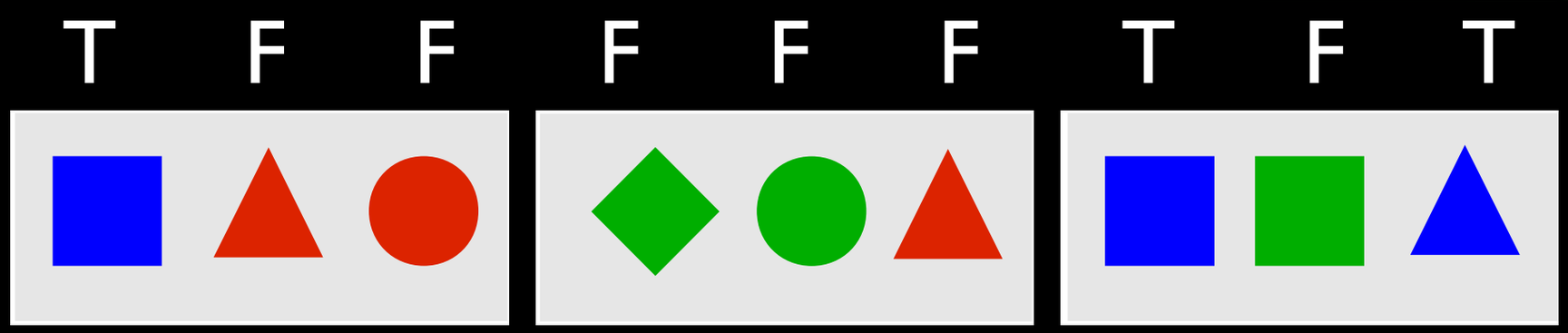
Evaluating languages



- High-throughput MTurk experiment.
  - 108 concepts,
    - Boolean (*circle or red*)
    - Context-dependent ("Determiners")
      (*unique largest , exists another with same shape*)
  - 2 orders per concept,
  - 1596 participants.

Piantadosi, Goodman, Tenenbaum (in prep)

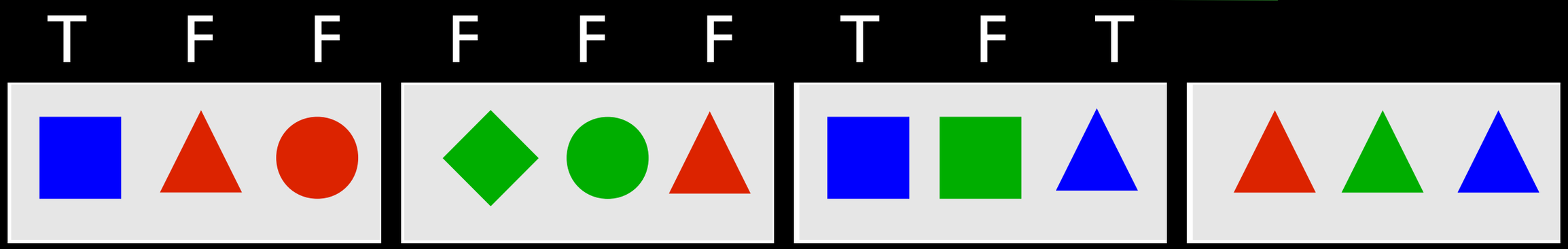
Evaluating languages



- High-throughput MTurk experiment.
  - 108 concepts,
    - Boolean (*circle or red*)
    - Context-dependent ("Determiners")
      (*unique largest , exists another with same shape*)
  - 2 orders per concept,
  - 1596 participants.

Piantadosi, Goodman, Tenenbaum (in prep)

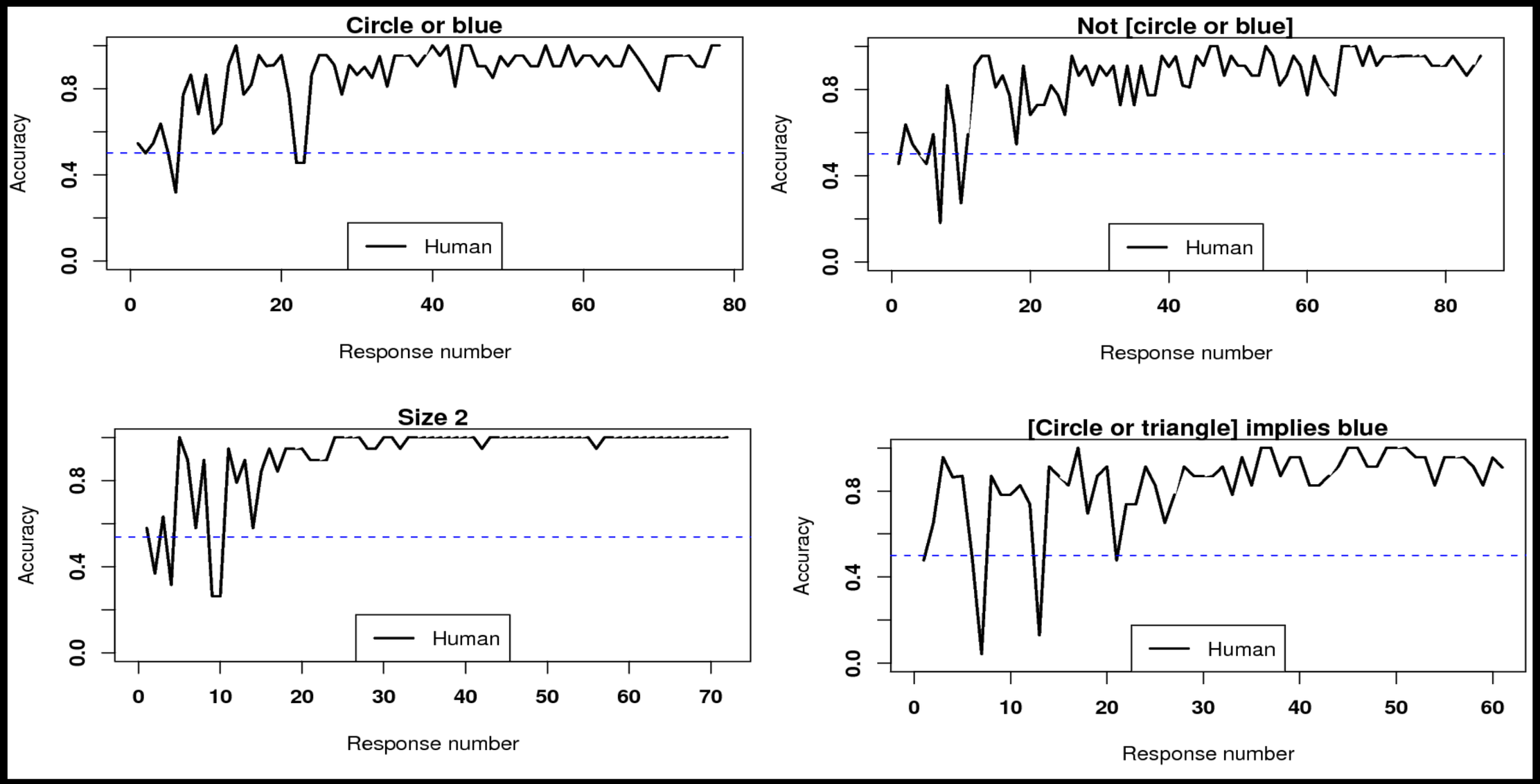
Evaluating languages



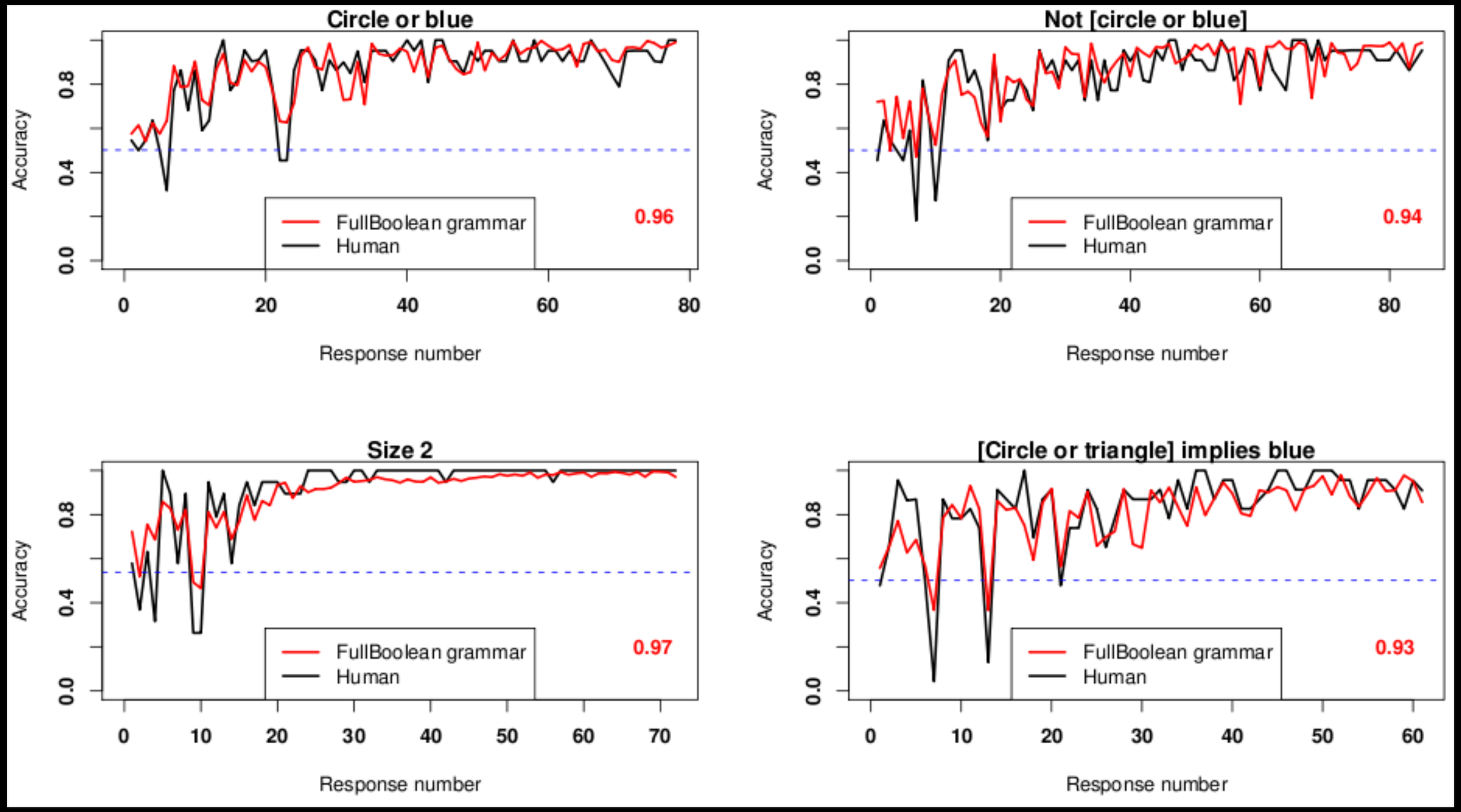
- High-throughput MTurk experiment.
  - 108 concepts,
    - Boolean (*circle or red*)
    - Context-dependent ("Determiners") (*unique largest , exists another with same shape*)
  - 2 orders per concept,
  - 1596 participants.

Piantadosi, Goodman, Tenenbaum (in prep)

Boolean concepts

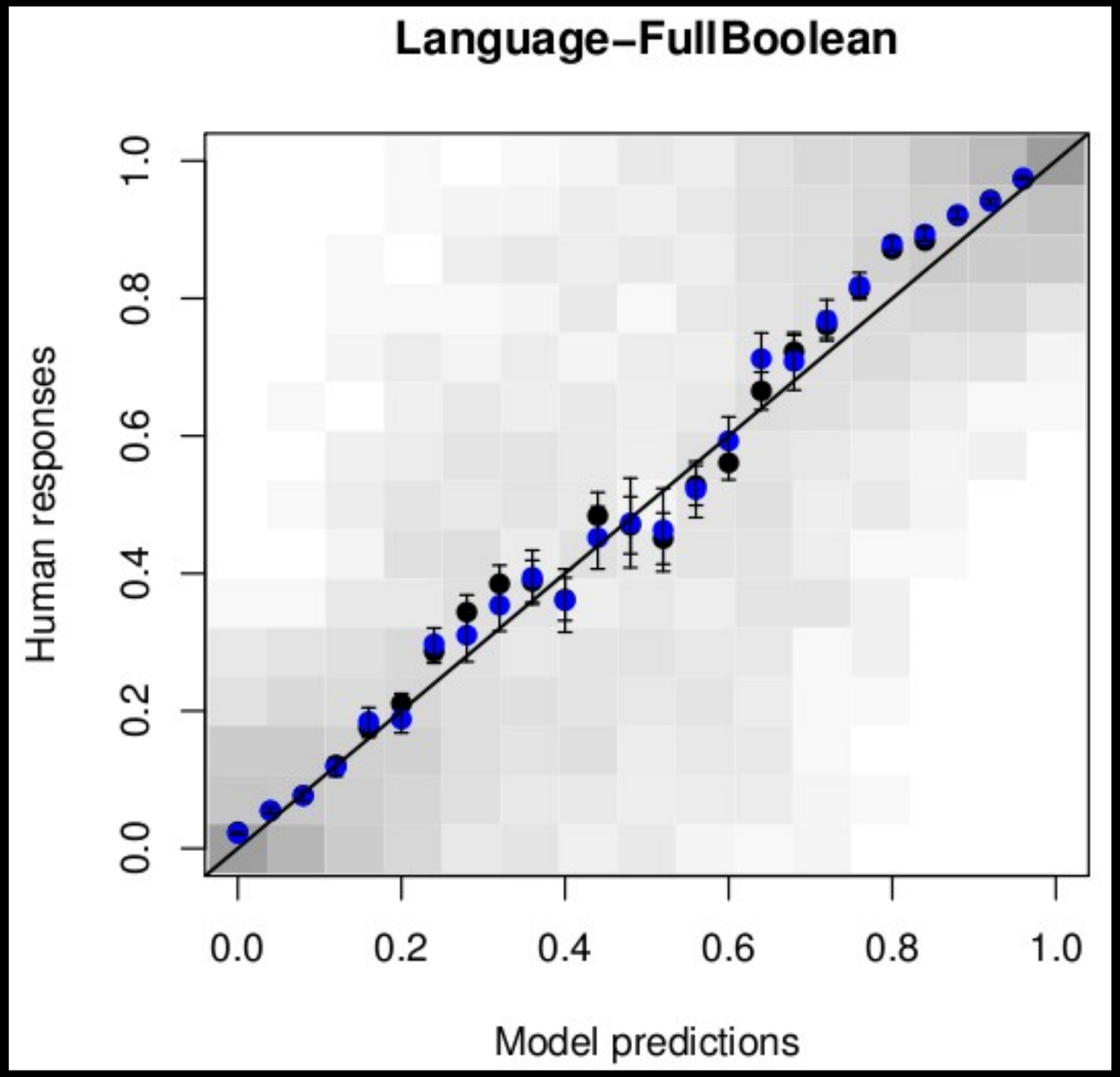


Boolean concepts



Boolean concepts

Best model performance on Boolean concepts:



Comparing languages

- DNF
  disjunctions of conjunctions

```
(λ (x) (or (and (red? x) (circle? x))
(and (red? x) (triangle? x))))
```

- Horn clauses
  conjunctions of implications

```
(λ (x) (and (implies (not (red? x)) false)
(implies (not (triangle? x)) (circle? x))))
```

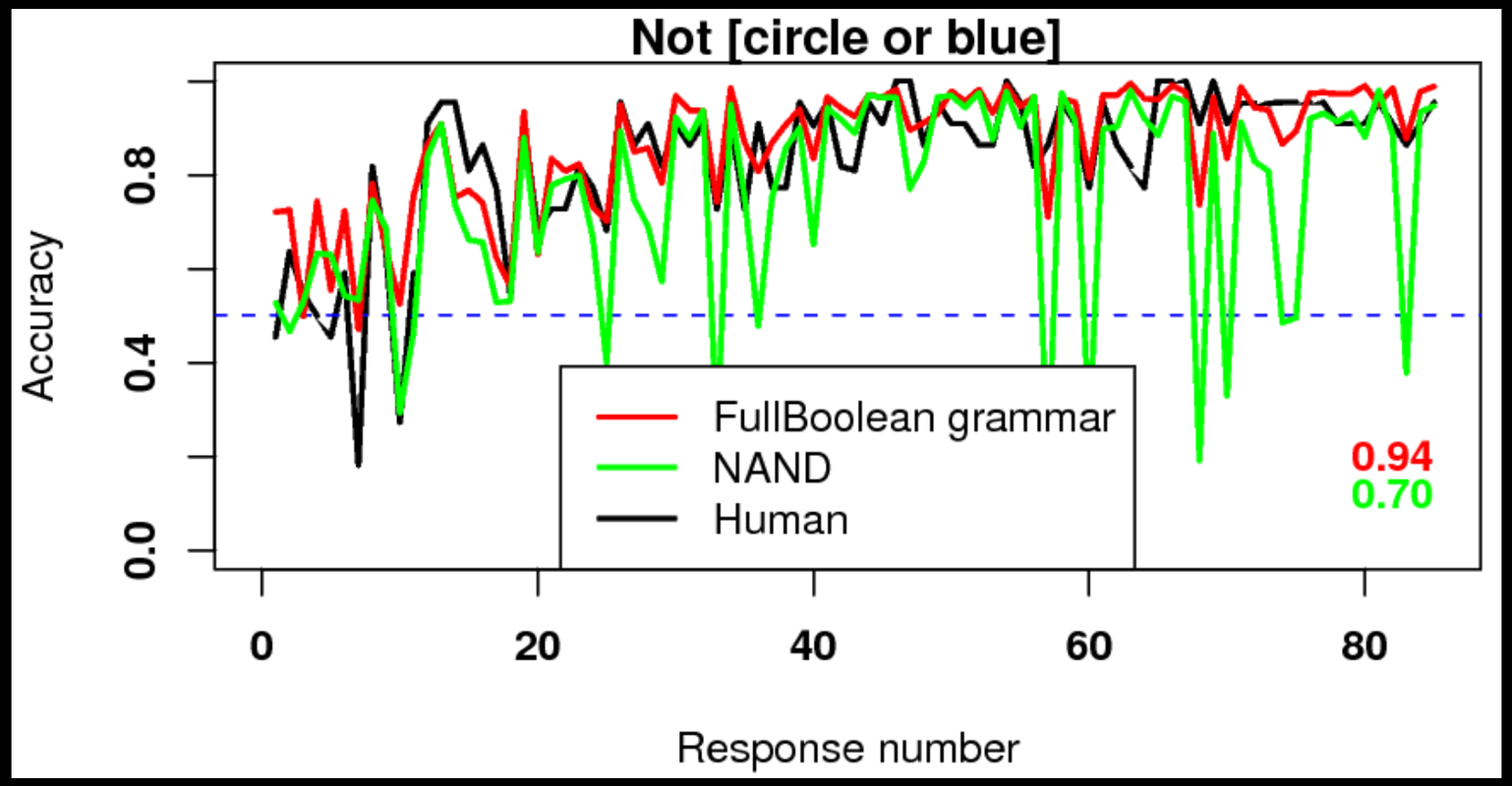
- Full boolean
  any combinations of AND, OR, NOT, IF, IFF

```
(λ (x) (and (red? x) (or (circle? x)
(triangle? x))))
```

- Nand
  combinations of NAND

```
(λ (x) (nand false (nand (red? x)
(nand (nand false (circle? x)) (nand
false (triangle? x)))))
```

Comparing languages

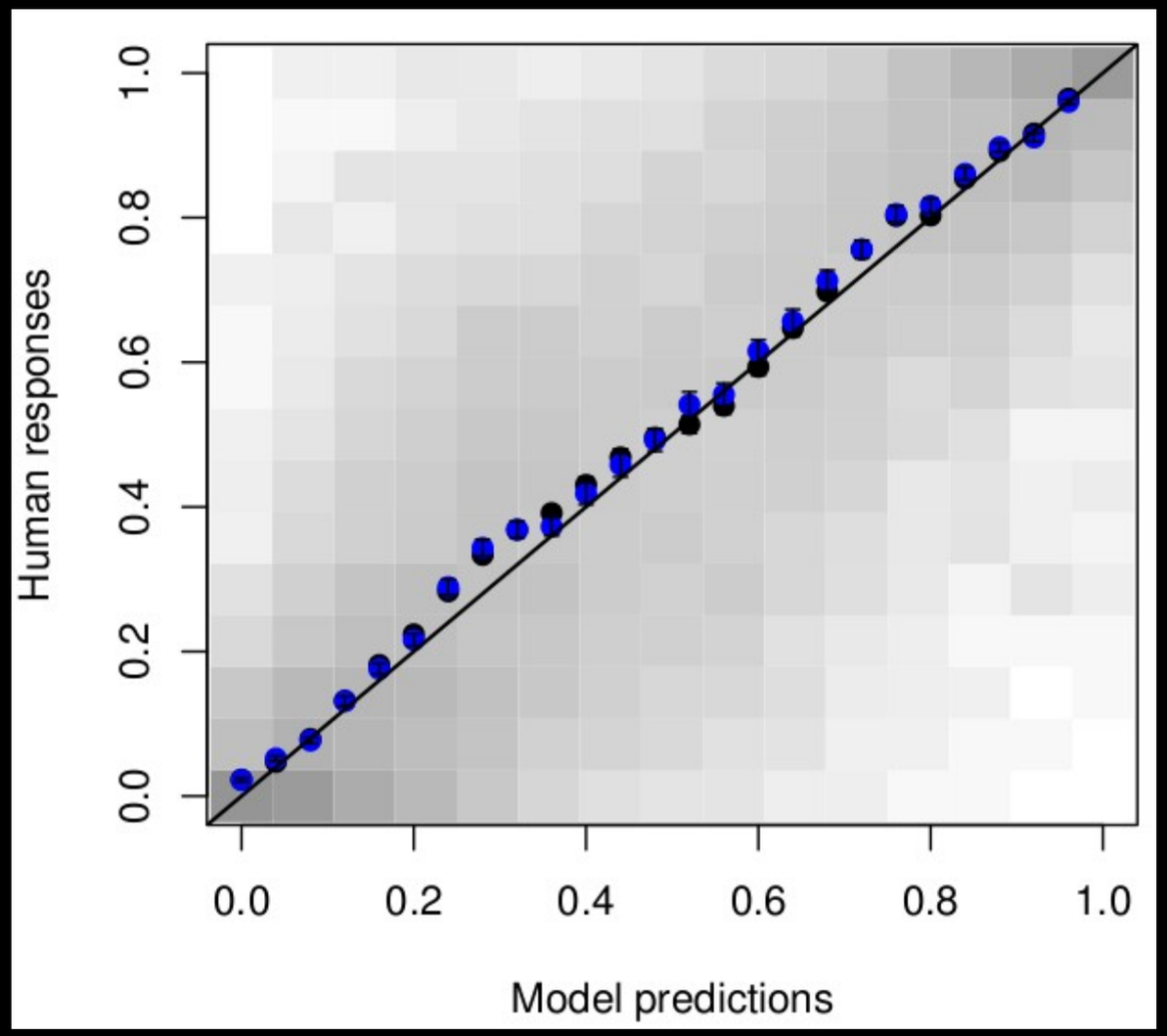


Grammar	H.O. LL
FULLBOOLEAN	-16315.27
CNF	-16333.59
DNF	-16368.31
BICONDITIONAL	-16385.01
IMPLIES	-16442.40
HORNCLAUSE	-16487.25
SIMPLEBOOLEAN	-16490.51
NAND	-16902.68
NOR	-16917.49
UNIFORM	-19482.72
EXEMPLAR	-23645.13
ONLYFEATURES	-31662.08
RESPONSE-BIASED	-37906.77

- Fit hyper-parameters (dirichlet on each NT) for each language.
- Evaluated against held out data.

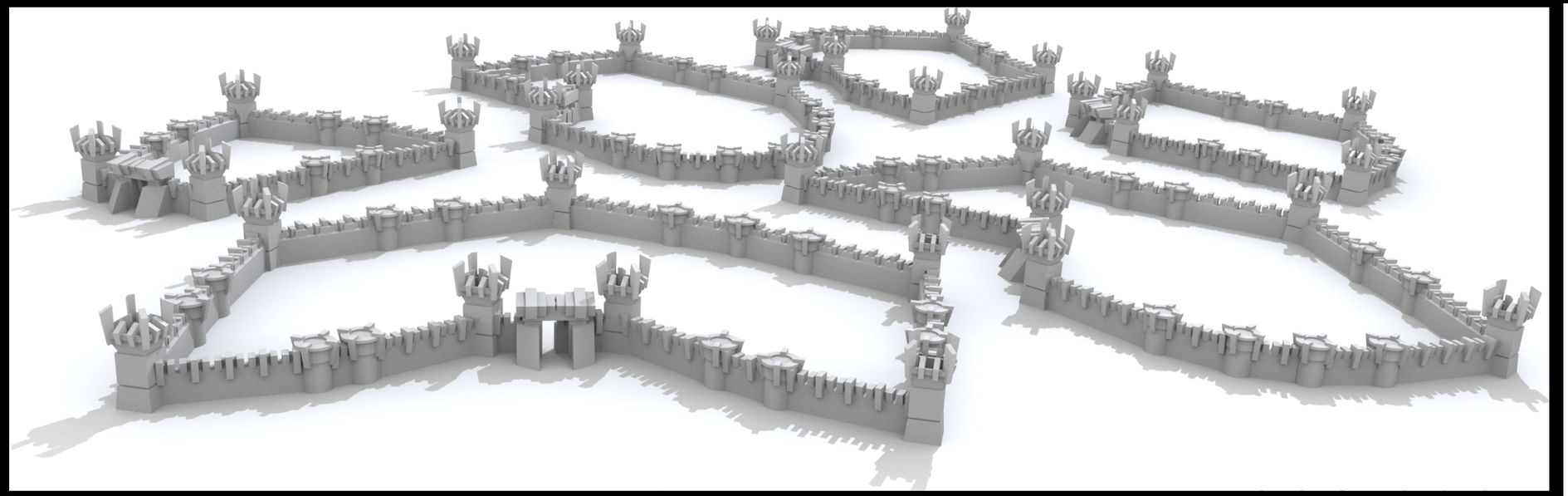
Non-Boolean concepts

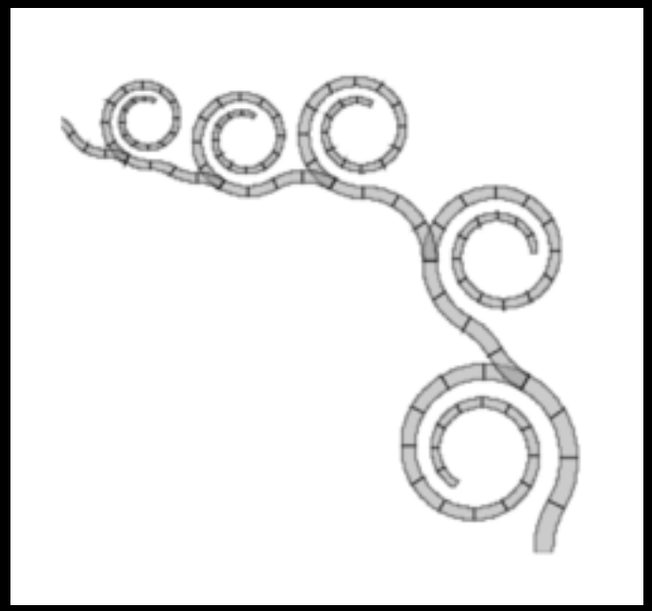
- Best language when context-dependent concepts included is full boolean **plus quantifiers**.

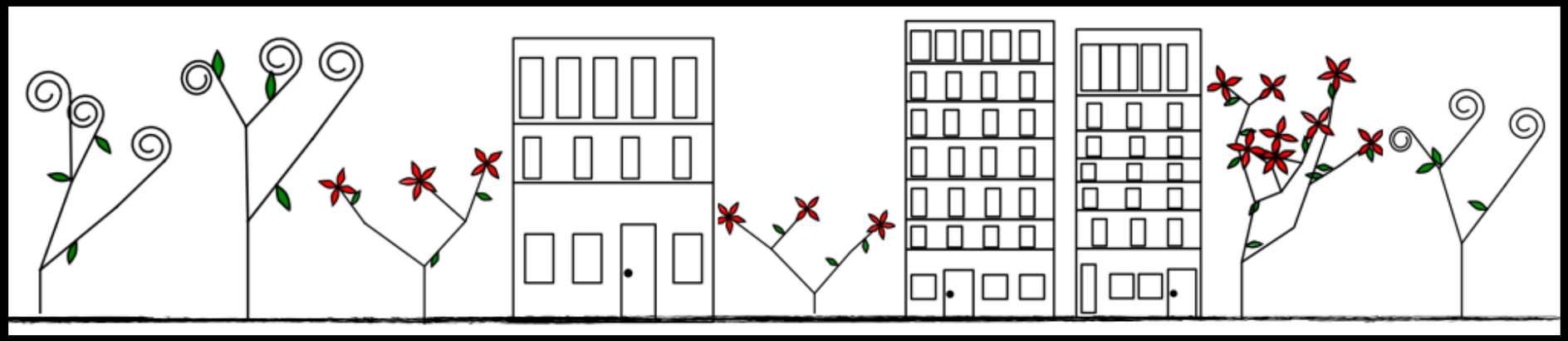


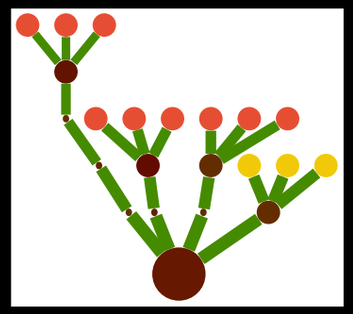
FOL	One-Or-Fewer	Small-Cardinalities	2nd-Ord.-Quan.	H.O. LL
✓	·	·	·	-79279.95
✓	✓	·	·	-79560.90
·	✓	·	·	-79642.46
·	✓	✓	·	-79972.75
✓	✓	·	✓	-80198.75
✓	·	·	✓	-80267.46
✓	·	✓	·	-80285.38
·	✓	·	✓	-80300.00
·	·	✓	·	-80614.35
✓	✓	✓	·	-80942.77
✓	✓	✓	✓	-81138.27
·	✓	✓	✓	-81289.85
✓	·	✓	✓	-81596.68
·	·	✓	✓	-81651.36
FULLBOOLEAN				-81773.43
BICONDITIONAL				-81967.68
SIMPLEBOOLEAN				-82144.71
·	·	·	·	-82219.08
CNF				-82685.21
DNF				-82752.82
·	·	·	✓	-82853.59

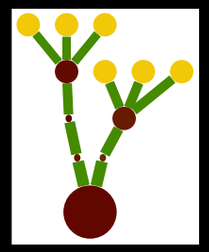
Generative kinds

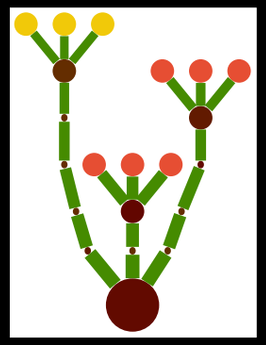


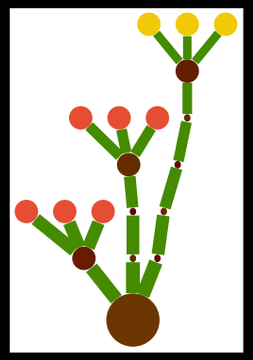


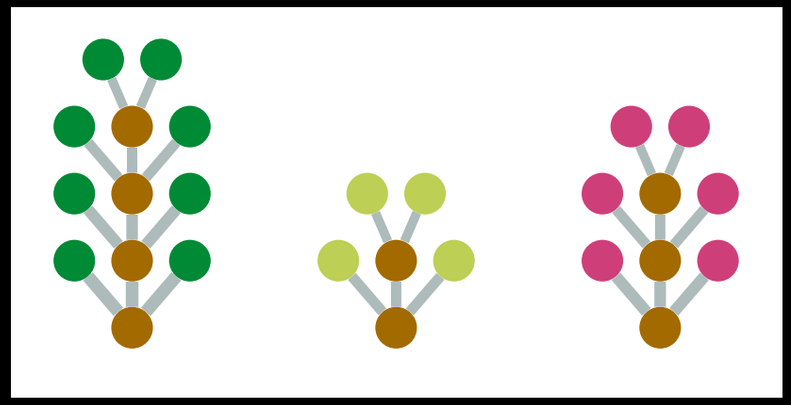








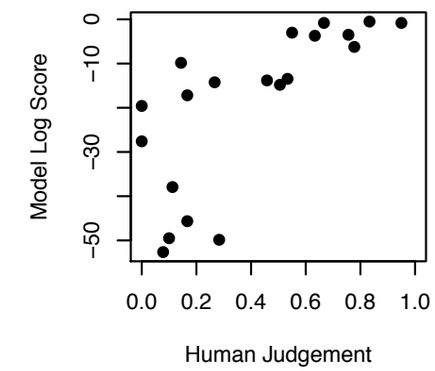




Learning generative kinds

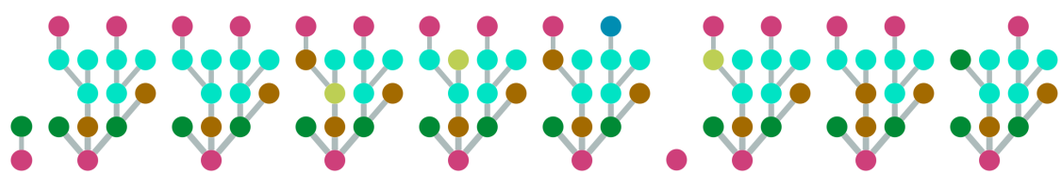
Simple prototype

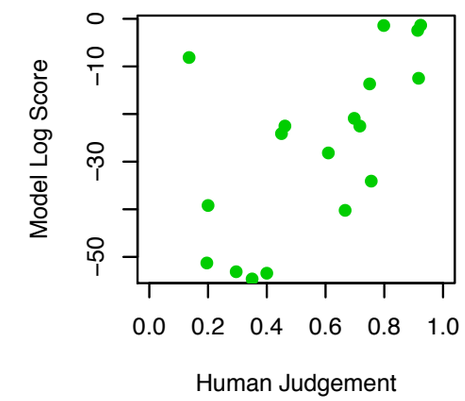
```
(lambda ()
  (node 'a
        (node 'b (node 'c (node 'c)) (node 'b))
        (node 'd (node 'e))))
```



Sub-concepts

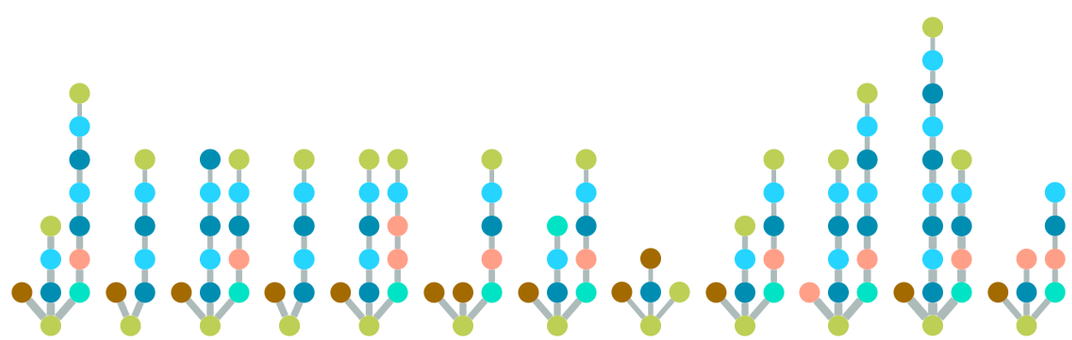
```
(begin
  (define (part) (node 'c (node 'c (node 'a)) (node 'c)))
  (lambda () (node 'a (node 'b) (node 'd (part)) (node 'b (part) (node 'd)))))
```

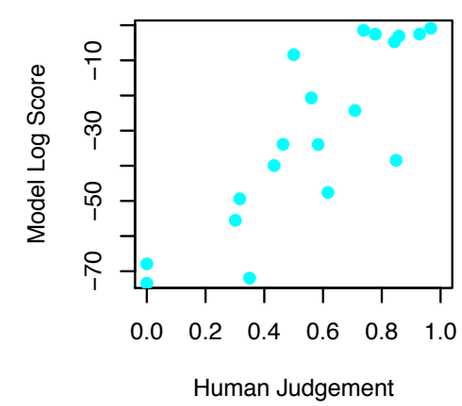




Single recursion

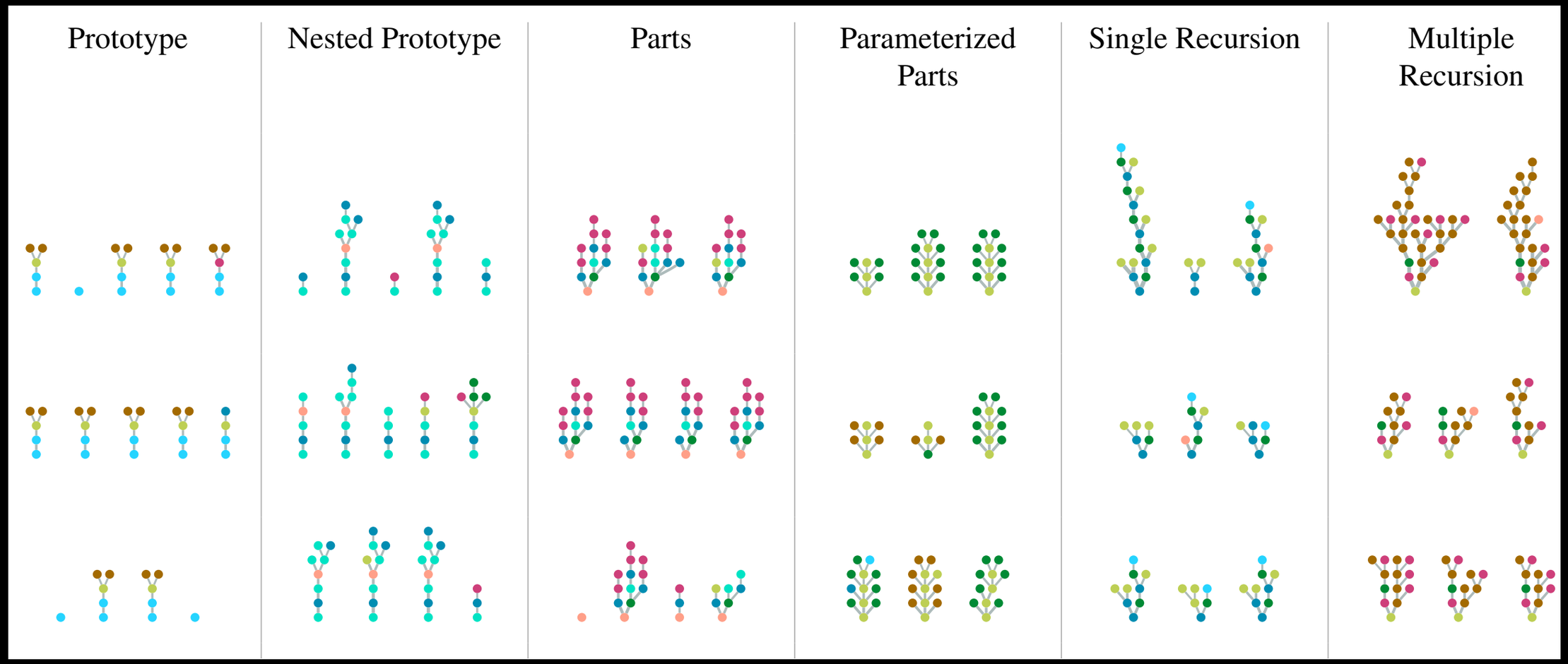
```
(begin
  (define (part) (node 'a (node 'b (if (flip) (node 'c) (part)))))
  (lambda () (node 'c (node 'd) (part) (node 'e (node 'f (part))))))
```





Stuhlmueller, Tenenbaum, Goodman (2010)

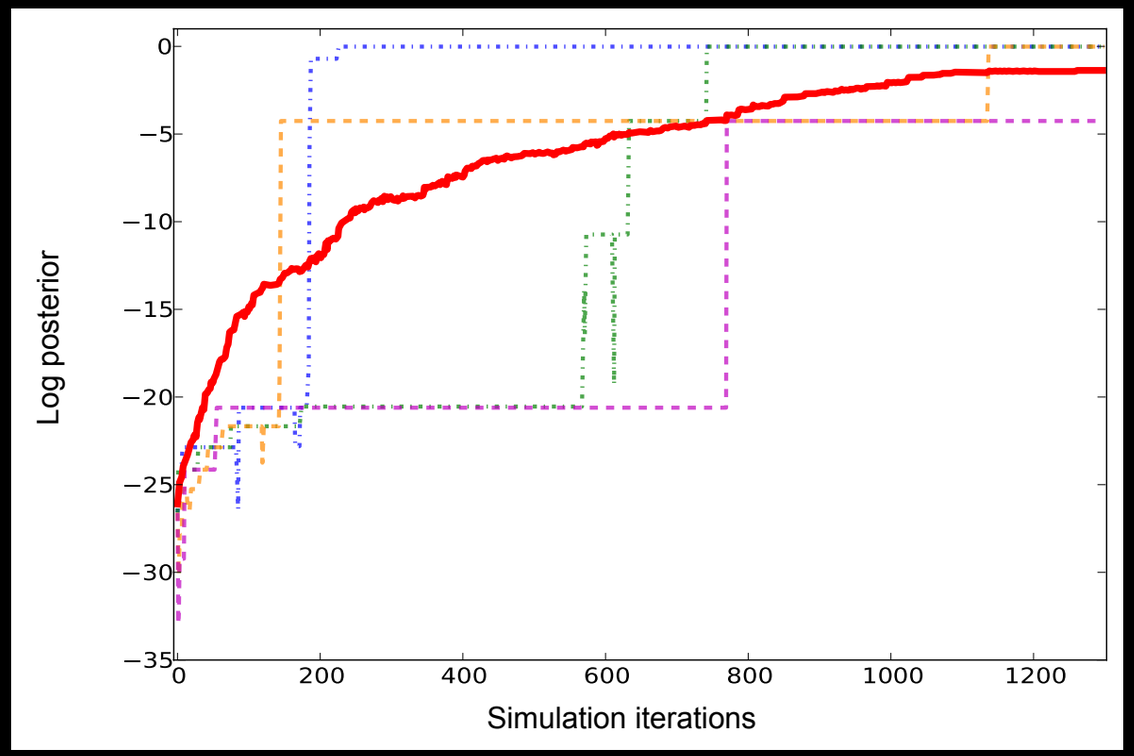
Learning generative kinds



	Set GCM	Transition GCM	Tree GCM	Generative Model
Prototype	0.589	0.751	**0.803**	0.748
Nested Prototype	0.544	0.851	**0.937**	0.904
Parts*	0.320	0.617	0.705	**0.835**
Parameterized Parts	0.298	0.591	0.778	**0.911**
Single Recursion	0.284	0.499	0.637	**0.773**
Multiple Recursion	0.505	0.561	0.451	**0.770**

Algorithms for induction

- What algorithms are capable of learning concepts in a language of thought?
  - All results so far were computed using MCMC based on constituent regeneration (Goodman, et al, 2008).
  - Is this cognitively plausible? Maybe...
  - But this is probably not enough on its own.



(Ullman, Goodman, Tenenbaum, 2010)

Program induction

- Program induction is especially hard. How could it be done?
- Idea: anti-unification + argument compression (+ search/MCMC).

```
(/ (+ 1 (* 2 3))
   (+ 1 (* 5 3)))
```

"inverse inlining" →

```
(define (F x) (+ 1 (* x 3)))
(/ (F 2) (F 5))
```

```
(define (F x)
 (+ 1 (* x 3)))
(/ (F 2) (F 5))
```

"de-argument" →

```
(define (F)
 (+ 1 (* (gaussian 3.5 1.0) 3)))
(/ (F) (F))
```

```
(define (F x)
 (+ 1 (* x 3)))
(F (F 1))
```

"de-argument" →

```
(define (F)
 (if (flip) (+ 1 (* (F) 3)) 1))
(F)
```

Program induction

- Program induction is especially hard. How could it be done?
- Idea: anti-unification + argument compression (+ search/MCMC).

```
(/ (+ 1 (* 2 3))
   (+ 1 (* 5 3)))
```

"inverse inlining" →

```
(define (F x) (+ 1 (* x 3)))
(/ (F 2) (F 5))
```

```
(define (F x)
 (+ 1 (* x 3)))
(/ (F 2) (F 5))
```

"de-argument" →

```
(define (F)
 (+ 1 (* (gaussian 3.5 1.0) 3)))
(/ (F) (F))
```

```
(define (F x)
 (+ 1 (* x 3)))
(F (F 1))
```

"de-argument" →

```
(define (F)
 (if (flip) (+ 1 (* (F) 3)) 1))
(F)
```

Program induction

- Program induction is especially hard. How could it be done?
- Idea: anti-unification + argument compression (+ search/MCMC).

```
(/ (+ 1 (* 2 3))
   (+ 1 (* 5 3)))
```

"inverse inlining" →

```
(define (F x) (+ 1 (* x 3)))
(/ (F 2) (F 5))
```

```
(define (F x)
 (+ 1 (* x 3)))
(/ (F 2) (F 5))
```

"de-argument" →

```
(define (F)
 (+ 1 (* (gaussian 3.5 1.0) 3)))
(/ (F) (F))
```

```
(define (F x)
 (+ 1 (* x 3)))
(F (F 1))
```

"de-argument" →

```
(define (F)
 (if (flip) (+ 1 (* (F) 3)) 1))
(F)
```

Program induction

- Program induction is especially hard. How could it be done?
- Idea: anti-unification + argument compression (+ search/MCMC).

```
(/ (+ 1 (* 2 3))
   (+ 1 (* 5 3)))
```

"inverse inlining" →

```
(define (F x) (+ 1 (* x 3)))
(/ (F 2) (F 5))
```

```
(define (F x)
 (+ 1 (* x 3)))
(/ (F 2) (F 5))
```

"de-argument" →

```
(define (F)
 (+ 1 (* (gaussian 3.5 1.0) 3)))
(/ (F) (F))
```

```
(define (F x)
 (+ 1 (* x 3)))
(F (F 1))
```

"de-argument" →

```
(define (F)
 (if (flip) (+ 1 (* (F) 3)) 1))
(F)
```

Program induction

- Program induction is especially hard. How could it be done?
- Idea: anti-unification + argument compression (+ search/MCMC).

```
(/ (+ 1 (* 2 3))
   (+ 1 (* 5 3)))
```

"inverse inlining" →

```
(define (F x) (+ 1 (* x 3)))
(/ (F 2) (F 5))
```

```
(define (F x)
 (+ 1 (* x 3)))
(/ (F 2) (F 5))
```

"de-argument" →

```
(define (F)
 (+ 1 (* (gaussian 3.5 1.0) 3)))
(/ (F) (F))
```

```
(define (F x)
 (+ 1 (* x 3)))
(F (F 1))
```

"de-argument" →

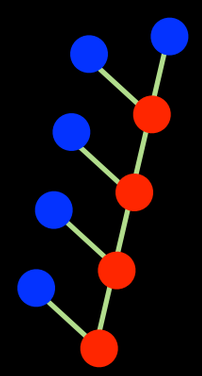
```
(define (F)
 (if (flip) (+ 1 (* (F) 3)) 1))
(F)
```

Program induction

- *Bayesian program merging* algorithm:
  (Cf. Stolcke & Omohundro, 1994)
  - Initial state is exemplar program (mixture of data).
  - Transform programs via anti-unification and argument compression.
  - Beam search* with respect to the posterior score.
    (*Or stochastic search, Monte Carlo, etc.)
    - Likelihood: marginal probability of data given program (computed by lazy particle filter or dynamic progr.).
    - Prior: syntactic complexity.

Hwang, Stuhlmueller, Goodman (2011)

Example



"data incorporation" →

```
(node r (node b)
        (node r (node b)
                (node r (node b)
                        (node r (node b) b)
```

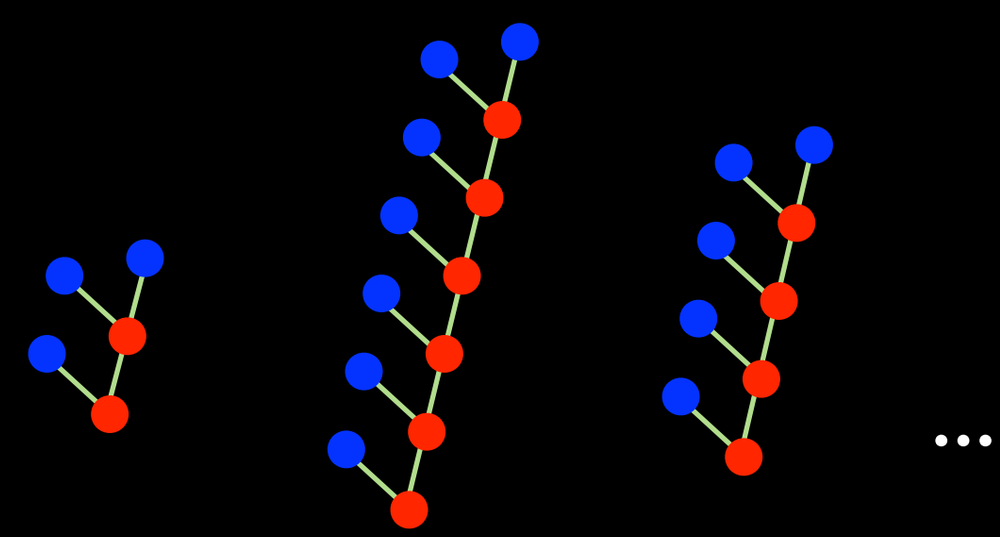
"inverse inline" →

```
(define (F x) (node r (node b) x))
(F (F (F (F b))))
```

"de-argument" →

```
(define (F) (if (flip 0.8)
                (node r (node b) (F))
                b))
(F)
```

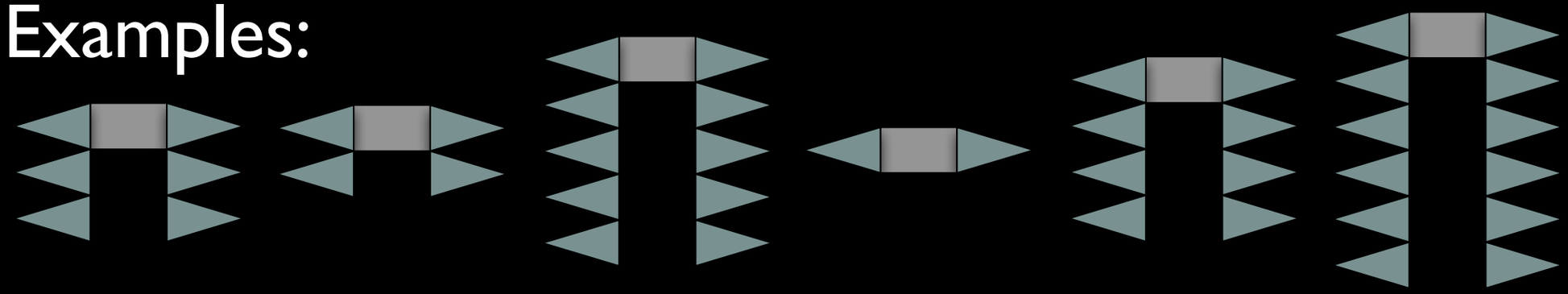
sample ⇢



...

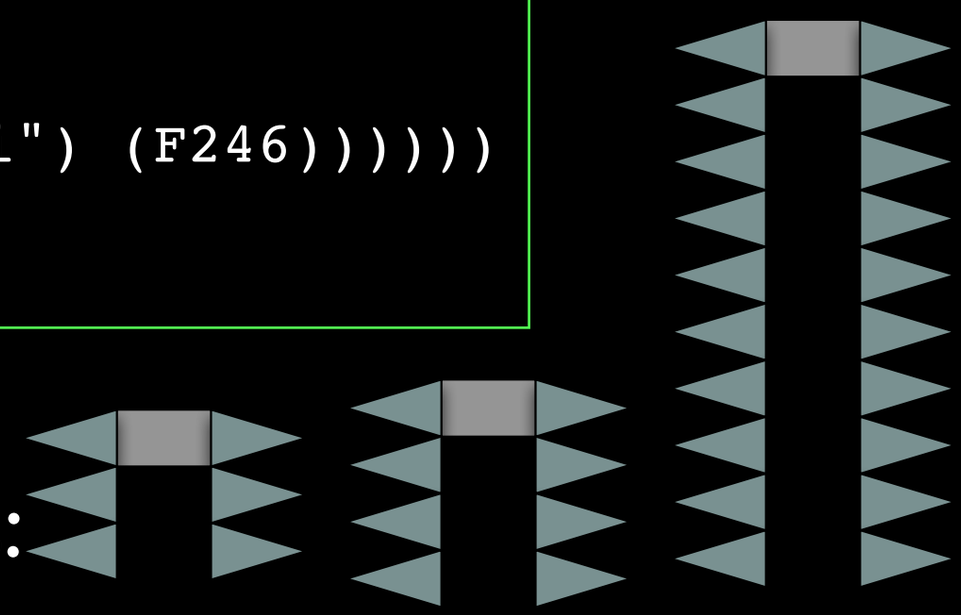
Example

Examples:

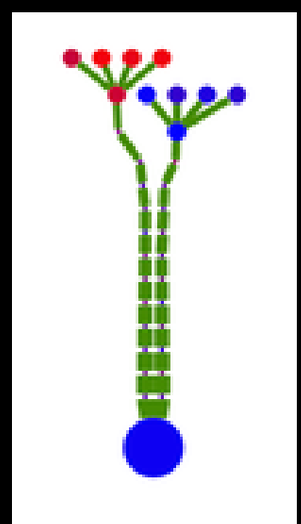


```
(define F248 (lambda () (F247 (F246))))

(define F247 (lambda (V994)
               (elem "rect"
                     (tr "left" (tr "reflect" V994))
                     (tr "right" V994))))

(define F246 (lambda ()
               (elem "tri"
                     (tr "forward"
                         (choose (elem "tri") (F246))))))

(F248)
```

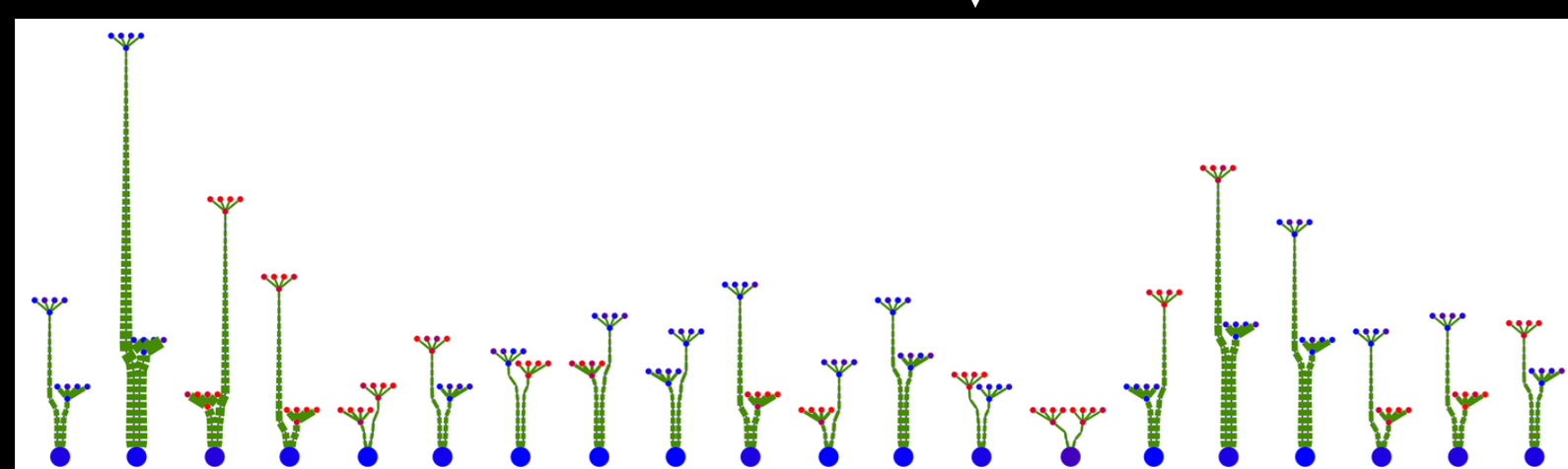
Posterior samples:



Example



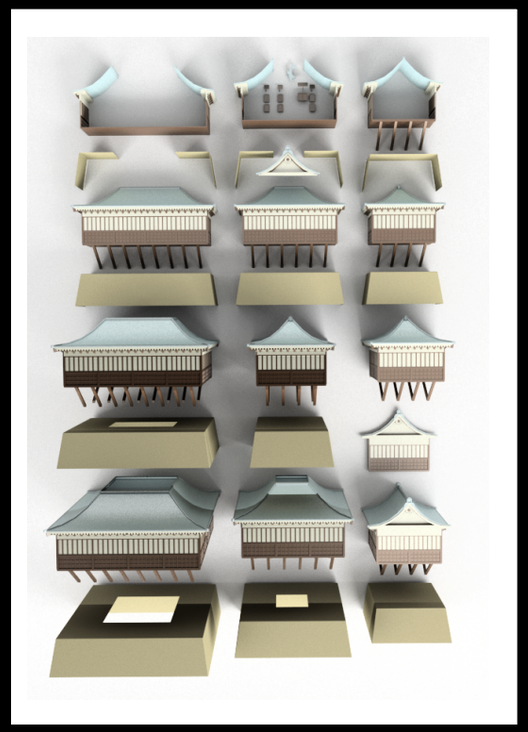
```
(begin (define F4 (lambda (V7 V8) (node (F1 V7 0.1) V8)))
    (define F3 (lambda (V6) (node (F1 V6 0.3))))
    (define F2
      (lambda (V3 V4)
        ((lambda (V5) (F4 V3 (F4 V4 V5)))
          (if (flip 9/11)
              (F2 81.0 85.0)
              (uniform-choice
                     (node (F1 204.0 0.3) (F3 199.0)
                       (F3 243.0) (F3 233.0) (F3 240.0)))
                     (F4 151.0
                       (node (F1 -21.0 0.3) (F3 7.0)
                         (F3 49.0) (F3 3e1) (F3 44.0)))))))
    (define F1
      (lambda (V1 V2)
        (data (color (gaussian V1 25)) (size V2))))
    (lambda ()
      (uniform-choice
             (node (F1 13.0 1) (F2 89.0 111.0)
               (F2 85.0 121.0))))
```



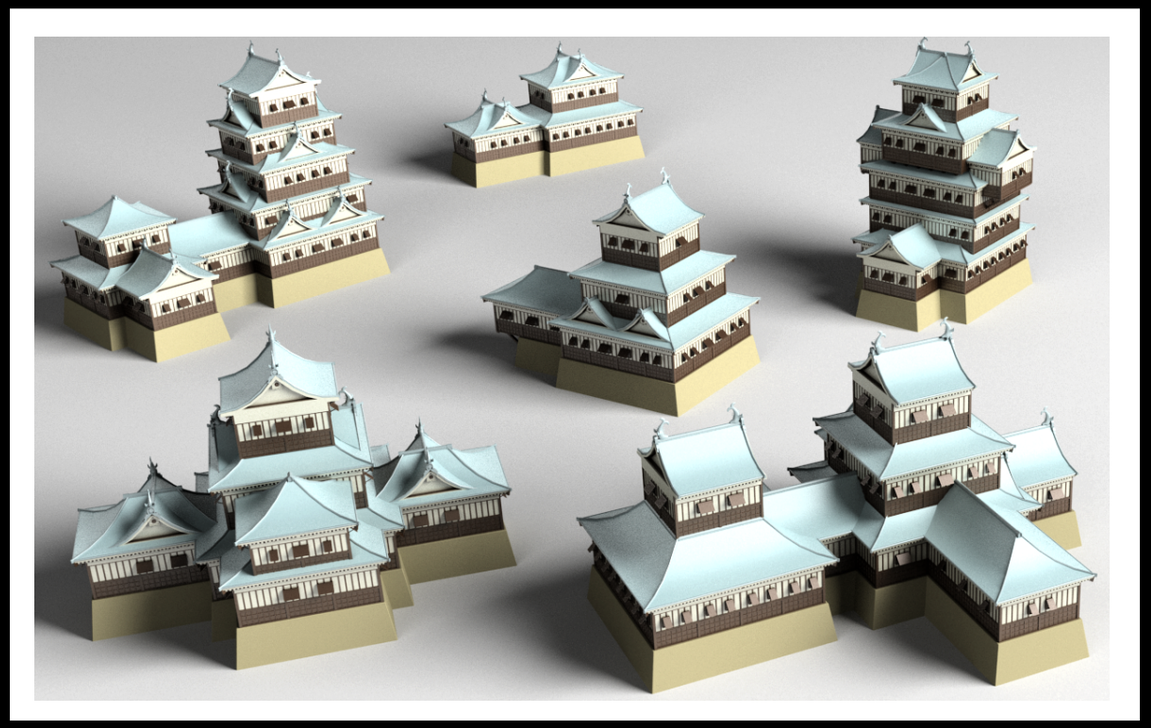
Application

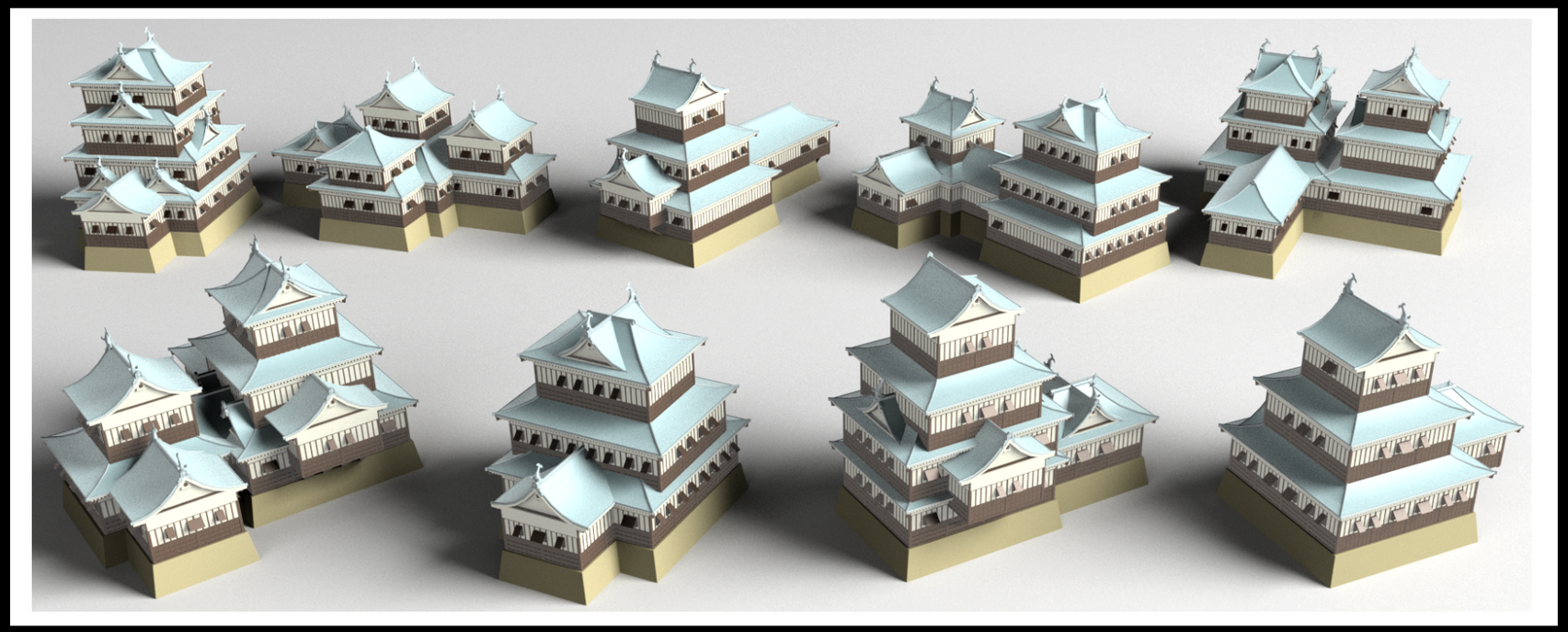
- **Inverse** procedural modeling:
  - Procedural model induced from a small set of artist-created examples.
  - Jerry Talton (Talton, et al, in prep.)

Terminals



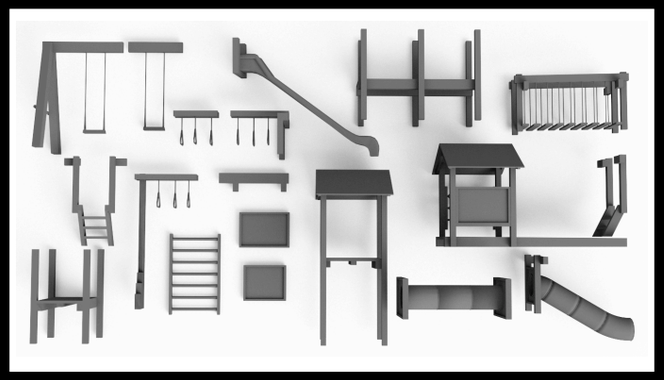
Examples

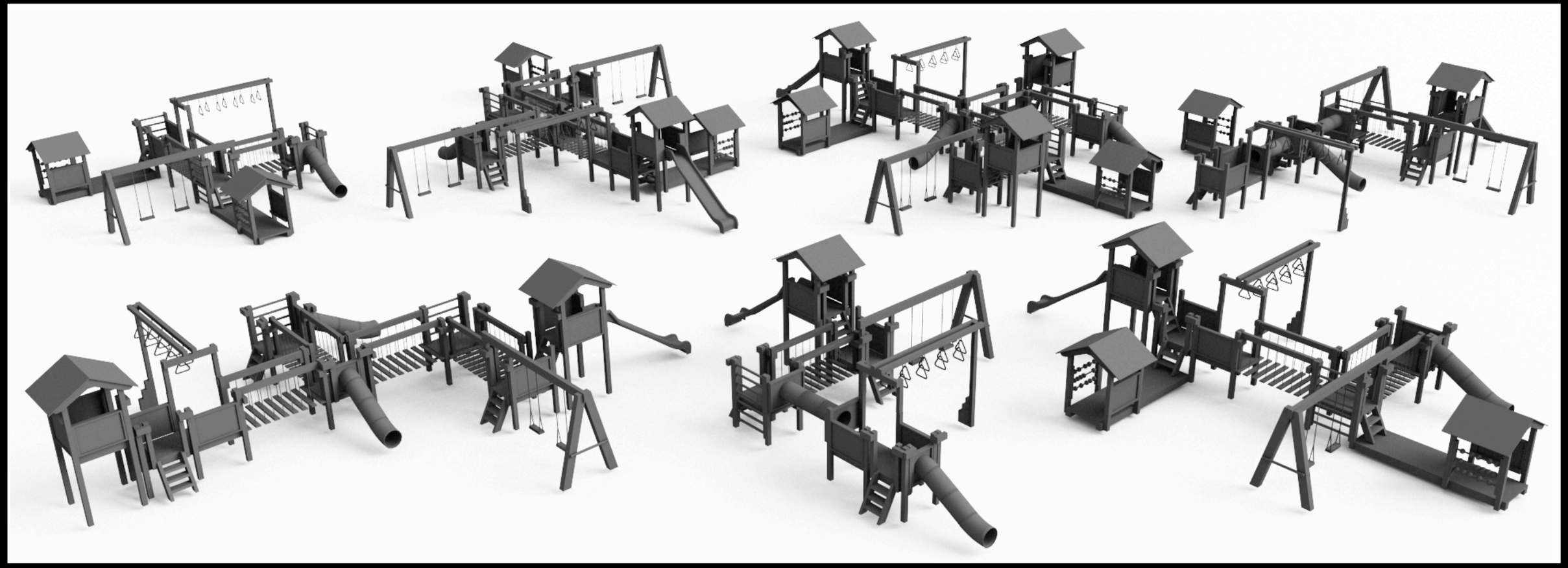




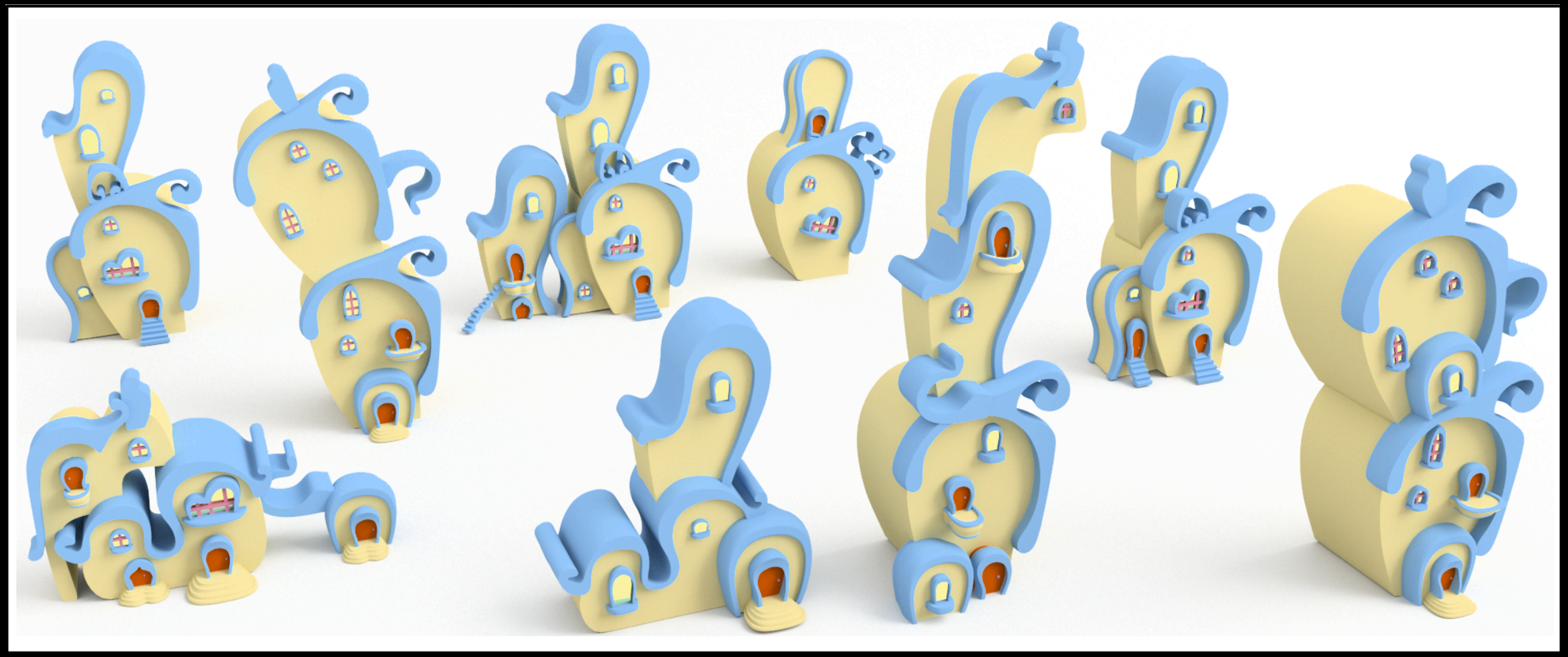
Posterior samples

Application





Application



Conclusion

Probabilistic inference

Generative models

Compositional representations

Probabilistic language of thought