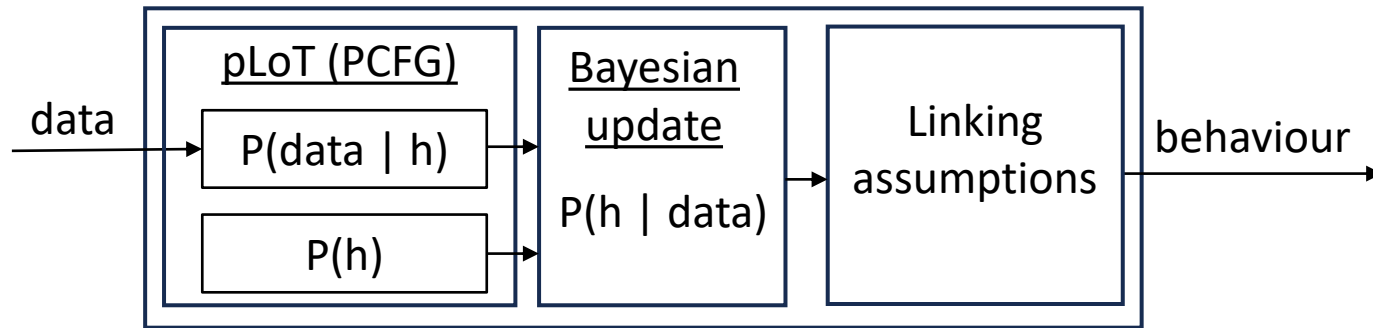


Part V

Conclusions & Recent Developments

Fausto Carcassi

Intro & structure



- Kinship
- Numbers
- Geometry
- Sequences
- Error rates
- Learning speed
- Language acquisition data
- Typological distributions

Part I	Introduction: On the very idea of an LoT
Part II	Technical background
Part III	Bayesian program induction (LOTlib3)
Part IV	Case studies
Part V	Summary & Future prospects

Various directions of development

- Re-emergence in philosophy
- New conceptual domains
- Technical tool (Program induction)
- Neurosymbolic architectures
- More realistic cognitive extensions

The Best Game in Town:
The Re-Emergence of the Language of Thought Hypothesis
Across the Cognitive Sciences¹

Jake Quilty-Dunn, Washington University in St. Louis, USA, quiltydunn@gmail.com,
sites.google.com/site/jakequiltydunn/

Nicolas Porot, Africa Institute for Research in Economics and Social Sciences, Mohammed VI
Polytechnic University, Morocco, nicolasporot@gmail.com, nicolasporot.com

Eric Mandelbaum, The Graduate Center & Baruch College, CUNY, USA,
eric.mandelbaum@gmail.com, ericmandelbaum.com

Short Abstract: This paper provides a survey of evidence from computational cognitive psychology, perceptual psychology, developmental psychology, comparative psychology, and social psychology, in favor of the language of thought hypothesis (LoTH). We outline six core properties of LoTs and argue that these properties cluster together throughout cognitive science. Instead of regarding LoT as a relic of the previous century, researchers in cognitive science and philosophy of mind should take seriously the explanatory breadth of LoT-based architectures as computational/representational approaches to the mind continue to advance.

¹ All authors contributed equally; authorship is in reverse alphabetical order.

Stitch

Library learning with Stitch

Bowers et al (**2023**) – Top-Down Synthesis for Library Learning

$\lambda x. +\ 3\ (*\ (+\ 2\ 4)\ 2)$

$\lambda xs. \text{map}\ (\lambda x. +\ 3\ (*\ 4\ (+\ 3\ x)))\ xs$

$\lambda x. *\ 2\ (+\ 3\ (*\ x\ (+\ 2\ 1)))$

$\text{fn0} = \lambda\alpha. \lambda\beta. (+\ 3\ (*\ \alpha\ \beta))$

$\lambda x. \text{fn0}\ (+\ 2\ 4)\ 2$

$\lambda xs. \text{map}\ (\lambda x. \text{fn0}\ 4\ (+\ 3\ x))\ xs$

$\lambda x. *\ 2\ (\text{fn0}\ x\ (+\ 2\ 1))$

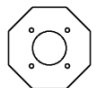
Library learning

A.

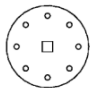
Initial DSL

```
connect | transform | matrix |
circle | line | 0 | 1 | 2 ..
```


Corpus of programs in the initial DSL



```
(connect (connect
(transform (repeat
(transform line (matrix
1 0 -0.5 (/ 0.5 (tan (/
pi 8)))))) ...
```



```
(connect (connect
(transform (transform
circle (matrix 2 0 0
0))(transform ...)
```

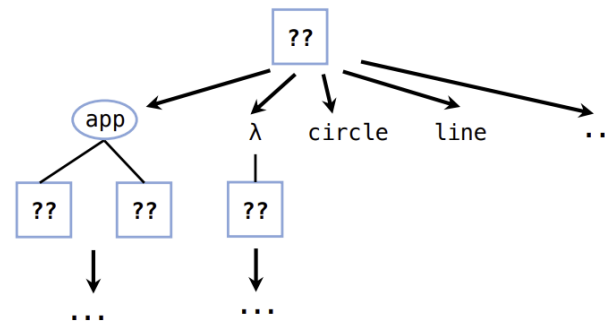


```
(connect (connect
(transform (repeat
(transform line (matrix
1 0 -0.5 (/ 0.5 (tan (/
pi 6)))))) ...
```

...

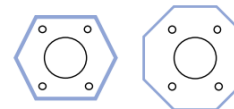
B.

STITCH uses **corpus-driven top-down synthesis** to find new abstractions that can compress the corpus of program trees



learned_fn_0 =
 $(\lambda x. \lambda y. (transform$
 $(repeat (transform line$
 $(matrix 1 0 -0.5 (/ 0.5$
 $(tan (/ pi x)))) x$
 $(matrix 1 (/ (* 2 pi)$
 $x) 0 0)) (matrix y 0 0$
 $0))))$

*Draws polygons parameterized by
number of sides and side length*




C.


**Learned library with new
abstractions**

```
connect | transform | matrix |
circle | line | 0 | 1 | 2
learned_fn_0 | learned_fn_1 |
learned_fn_2...
```

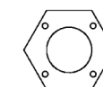
**Compressed corpus of programs
rewritten with the library**



```
(connect (connect
(learned_fn_0 8 1) ..)
```



```
(connect (connect
(transform (transform
circle (matrix 2 0 0
0))(transform ...)
```



```
(connect (connect
(learned_fn_0 6 1) ..)
```

Stitch

What could we use library learning for?

DreamCoder

Some problems with LOTlib3-style pLoT

Problem I: pLoT is not *scalable* compared to neural network systems!

- Why do you think that is?

Problem II: pLoT needs to start with a *domain specific* language.

- If we start with domain general primitives, programs would be too long!

Problem III: Combinatorial /discrete nature of the search space

- The space of programs isn't continuous, so we can't use our best algorithms (e.g., gradient descent or Hamiltonian Monte Carlo)

DreamCoder: The general idea

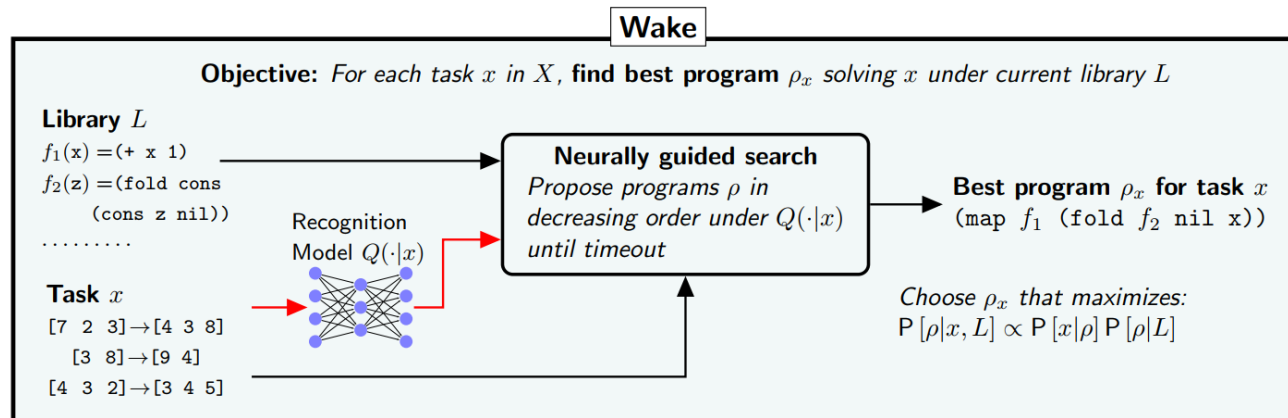
Grow new concepts

- Inferred programs become shorter!
- Domain adaptability
- (Older version of) Stitch

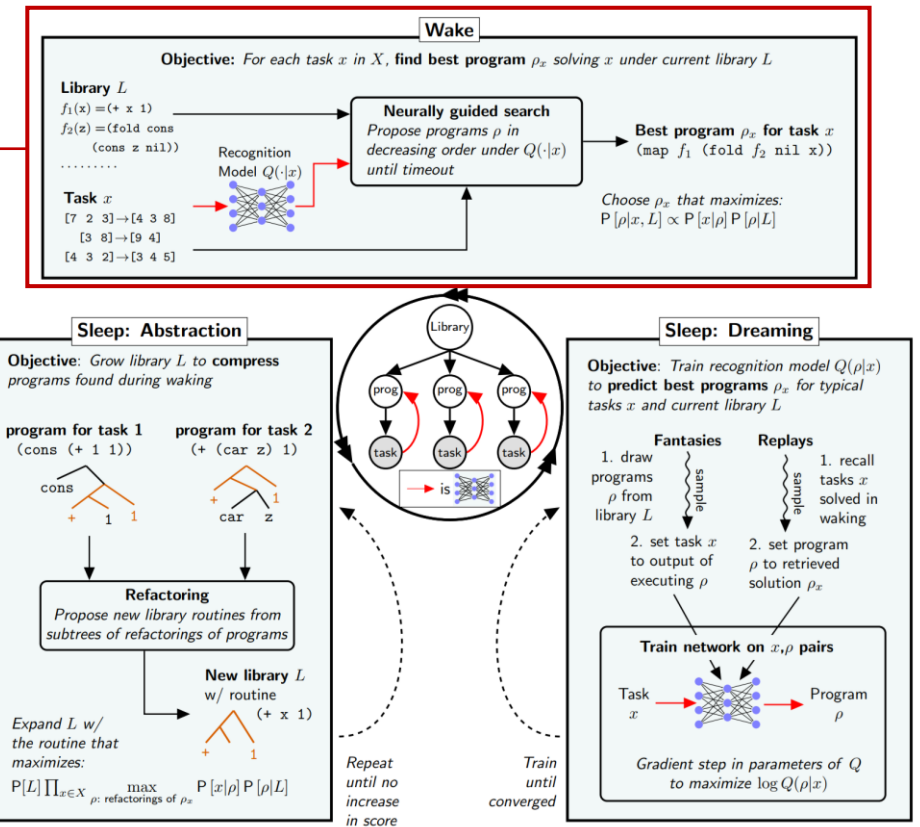
Learn *implicit procedural knowledge*

- We search the space based on observed object
- We need *intuition*
- Train a neural networks to propose programs from observations!

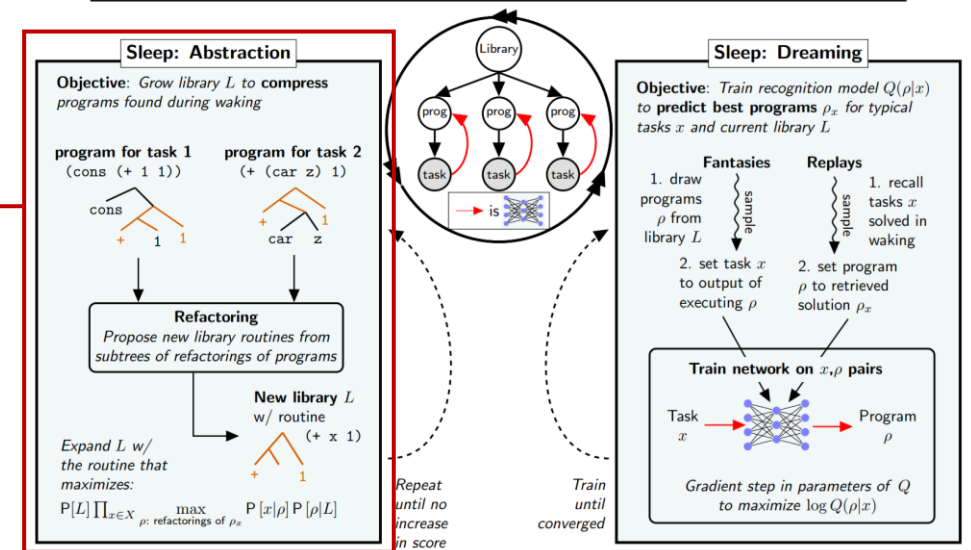
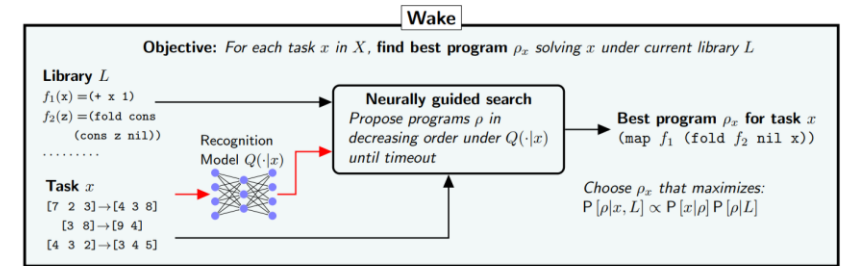
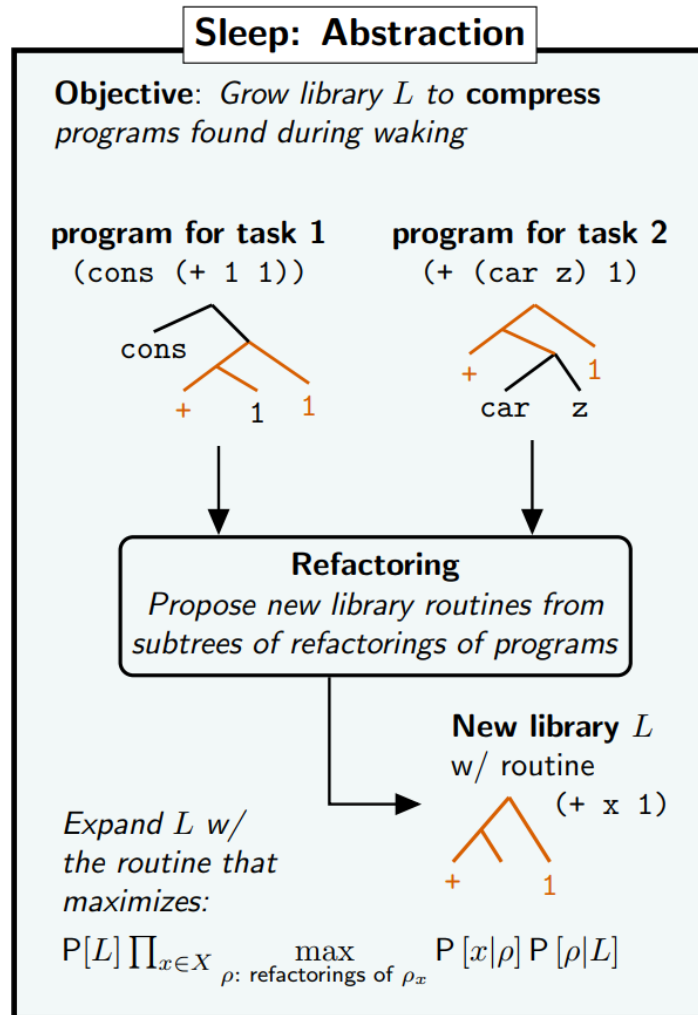
DreamCoder: Learning



- How do we grow new concepts?
- How and when is the ANN trained?
- Solution: Wake/Sleep Program Learning



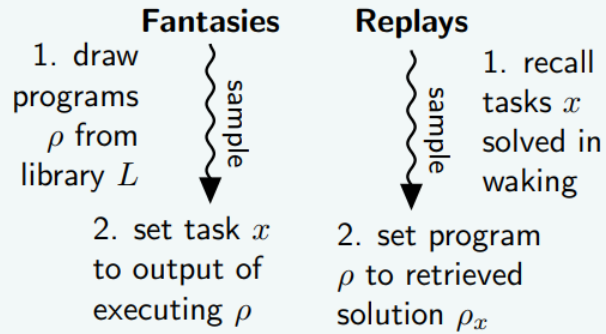
DreamCoder: Learning



DreamCoder: Learning

Sleep: Dreaming

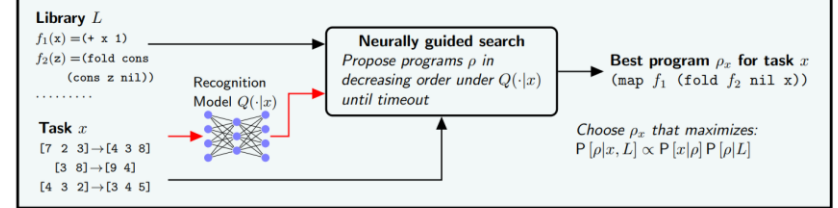
Objective: Train recognition model $Q(\rho|x)$ to predict best programs ρ_x for typical tasks x and current library L



Gradient step in parameters of Q to maximize $\log Q(\rho|x)$

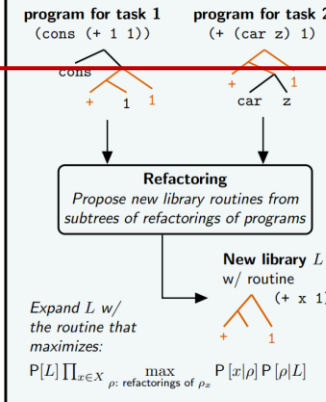
Wake

Objective: For each task x in X , find best program ρ_x solving x under current library L



Sleep: Abstraction

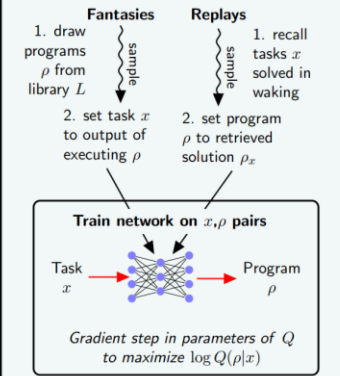
Objective: Grow library L to compress programs found during waking



Repeat until no increase in score

Sleep: Dreaming

Objective: Train recognition model $Q(\rho|x)$ to predict best programs ρ_x for typical tasks x and current library L



Gradient step in parameters of Q to maximize $\log Q(\rho|x)$

Train until converged

DreamCoder

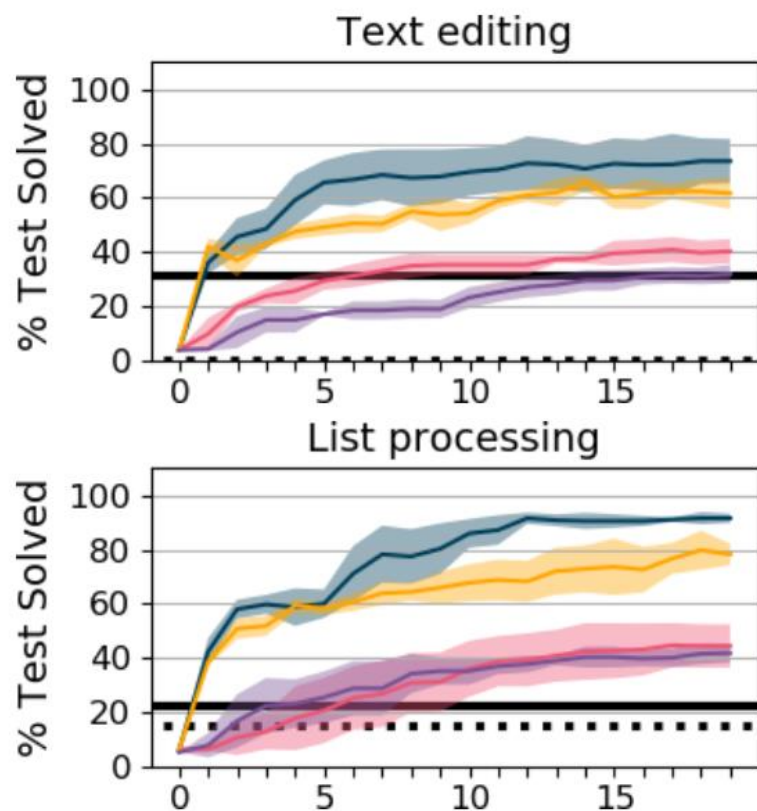
Challenge:

- Solving **list processing** and **text editing** tasks? (218 problems)
- Starting with general-purpose functions

Each round of abstraction builds on concepts from previous sleep cycles

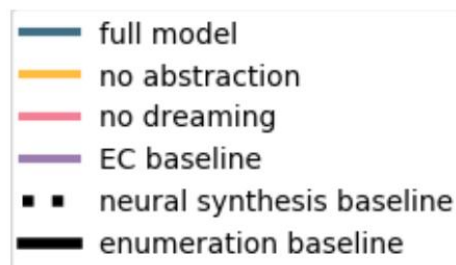
- E.g., first learns **filter**
- then uses it to learn to take the **maximum** element of a list,
- then uses that routine to learn a new library routine for extracting the **nth largest element** of a list
- finally uses to **sort** lists of numbers

DreamCoder



DreamCoder solves 84.3% of the problems with 1 hour & 8 CPUs per problem.

- Strongest baseline (CVC4) solved 82.4% of the problems
- CVC4 had a *different* hand-engineered library of primitives for each problem!



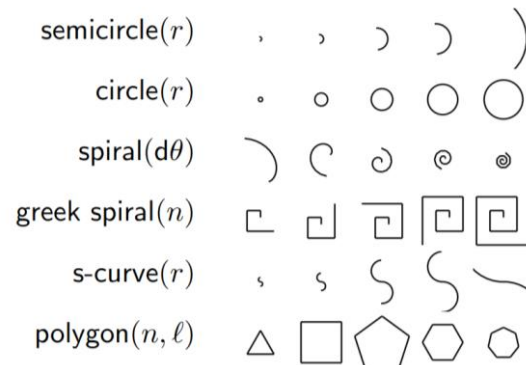
DreamCoder

More creative domains: generating images, plans, and text.

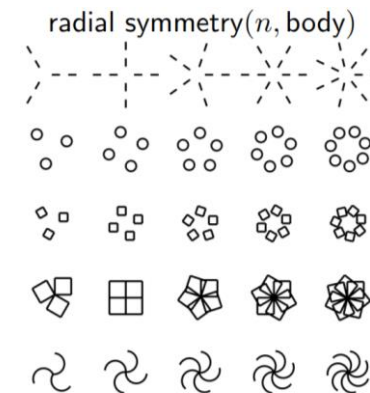
LOGO programs to learn
(30 out of 160)



Some learned
parametric
'shape concepts'

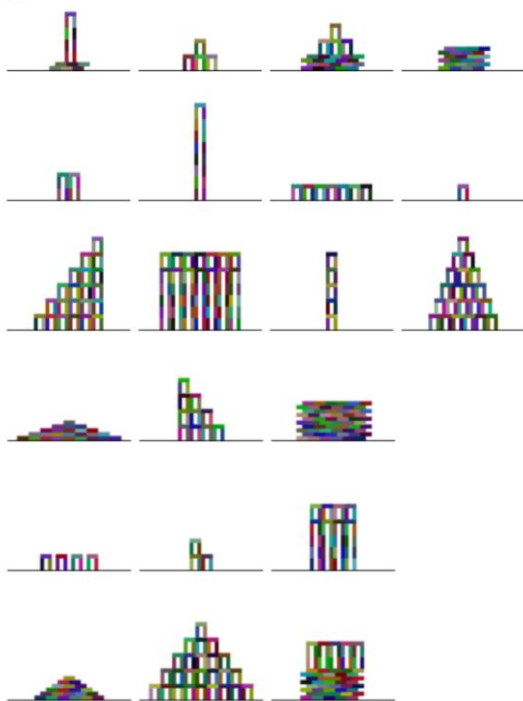


Some learned
higher-order
functions

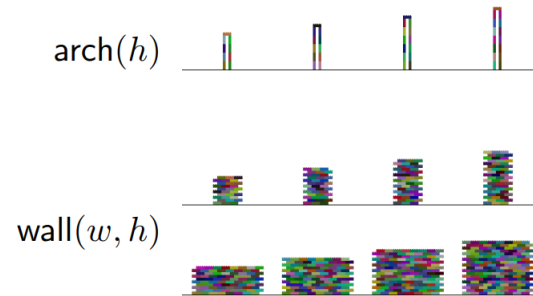


DreamCoder

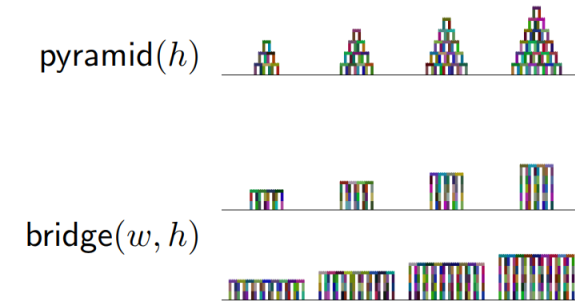
Towers building
programs to learn



Some learned
parametric
'shape concepts'



Some learned
higher-order
functions



DreamCoder

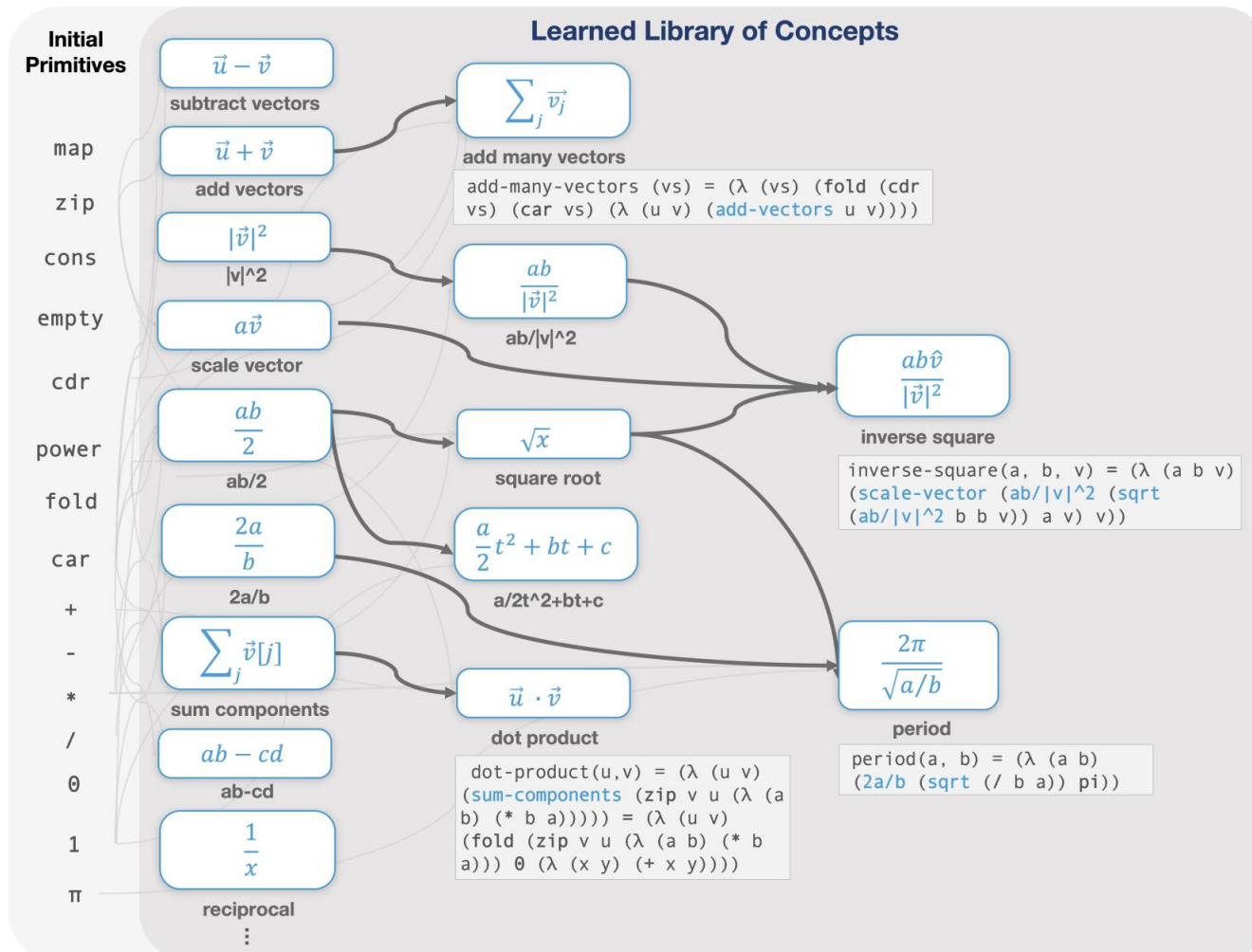
New task:

- Learn a set of 60 physical laws and mathematical identities from quantitative measurements
- E.g., mechanics, electromagnetism

After 8 cycles of wake/sleep, DreamCoder learns 93% of the rules

- First learn concepts like inner product or vector sum
- Then learn complex like the inverse square law
- Finally use these to formulate laws like Newton's laws of gravitation and Coulomb's law of electrostatic force

DreamCoder



Discovered Physics Equations

Newton's Second Law

$$\vec{a} = \frac{1}{m} \sum_i \vec{F}_i$$

```
(scale-vector (reciprocal m) (reciprocal (sum-components (add-many-vectors Fs)) (map (λ(r) (reciprocal r)) Rs)))
```

Parallel Resistors

$$R_{total} = \left(\sum_i \frac{1}{R_i} \right)^{-1}$$

Work

$$U = \vec{F} \cdot \vec{d}$$

```
(dot-product F d)
```

Force in a Magnetic Field

$$|\vec{F}| = q|\vec{v} \times \vec{B}|$$

```
(* q (ab-cd v_x b_y v_y b_x))
```

Kinetic Energy

$$KE = \frac{1}{2} m |\vec{v}|^2$$

```
(ab/2 m (|v|^2 v))
```

Coulomb's Law

$$\vec{F} \propto \frac{q_1 q_2}{|\vec{r}_1 - \vec{r}_2|^2} \widehat{r_1 - r_2}$$

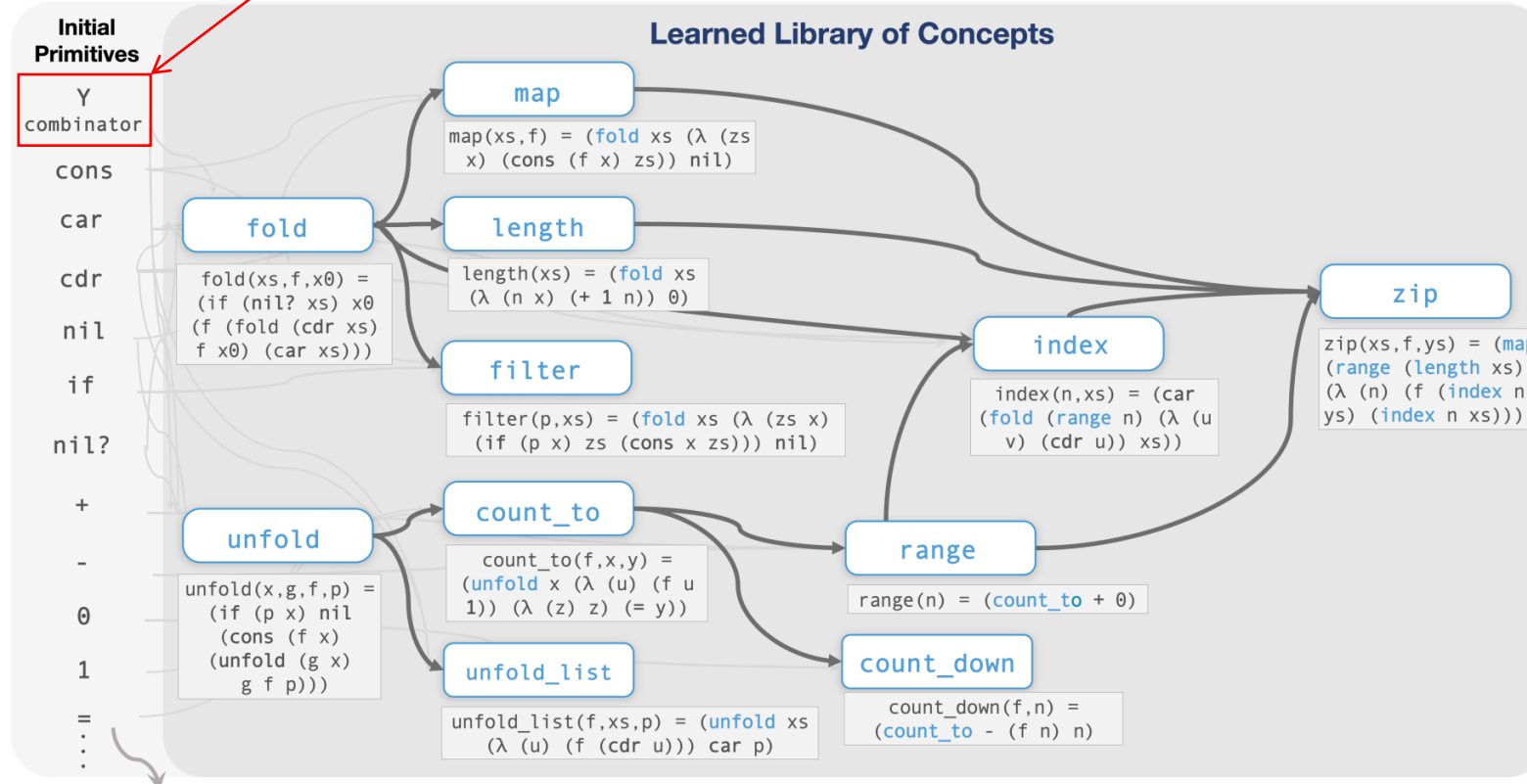
```
(inverse-square q_1 q_2 (subtract-vectors r_1 r_2))
```

```
(λ (x y z u) (map (λ (v) (* (/ (* (power (/ (* x x) (fold (zip z u (λ (w a) (- w a))) 0 (λ (b c) (+ (* b b) c)))) (/ (* 1 1) (+ 1 1))) y) (fold (zip z u (λ (d e) (- d e))) 0 (λ (f g) (+ (* f f) g)))) v)) (zip z u (λ (h i) (- h i)))))
```

Solution to Coulomb's Law if expressed in initial primitives

DreamCoder: Recursive algorithms

Can do recursion



Discovered Recursive Programming Algorithms

Stutter

`[■ ■] → [■ ■ ■ ■]`
`[■ ■ ■] → [■ ■ ■ ■ ■ ■]`
`(fold A (λ (u v) (cons v (cons v u))) nil)`

Take every other

`[■ ■ ■ ■] → [■ ■]`
`[■ ■ ■ ■ ■ ■] → [■ ■ ■]`
`(unfold_list cdr A nil?)`

List lengths

`[[■ ■ ■], [■]] → [3 1]`
`[[■ ■], [], [■]] → [2 0 1]`
`(map A length)`

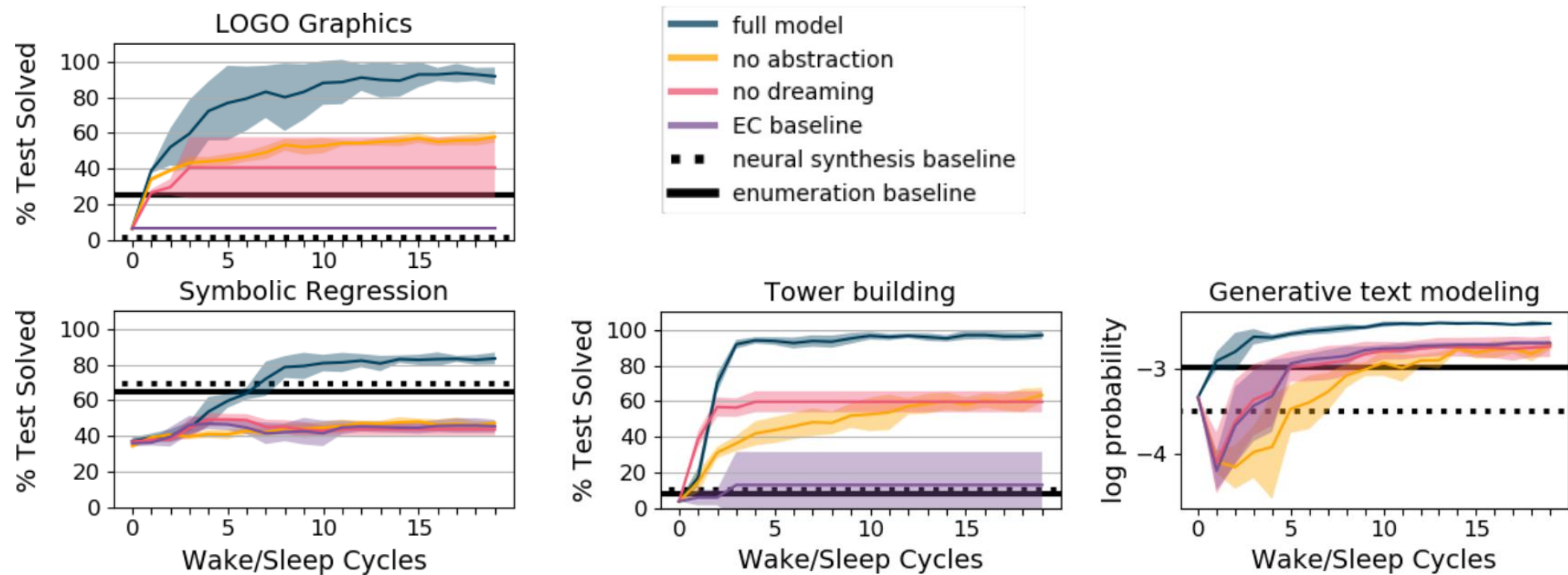
List differences

`[1 8 2], [0 5 1] → [1 3 1]`
`[2 3 6], [1 2 4] → [1 1 2]`
`(zip A - B)`

Solution to list differences if expressed in initial primitives

```
(λ (A B) (Y (Y 0 (λ (z u) (if (= u (Y A (λ (v w) (if (nil? w) 0 (+ 1 (v (cdr w)))))) nil (cons u (z (+ u 1)))))) (λ (a b) (if (nil? b) nil (cons (- (car (Y (Y 0 (λ (c d) (if (= d (car b)) nil (cons d (c (+ d 1)))))) (λ (e f) (if (nil? f) A (cdr (e (cdr f)))))) (car (Y (Y 0 (λ (g h) (if (= h (car b)) nil (cons h (g (+ h 1)))))) (λ (i j) (if (nil? j) B (cdr (i (cdr j)))))) (a (cdr b))))))
```

DreamCoder: Results



DreamCoder

DreamCoder is a big step forward engineering-wise

Connection to pLoT/cognition not yet fully elaborated

- Authors are more interested in program induction

(Very) difficult to use

The Child as Hacker

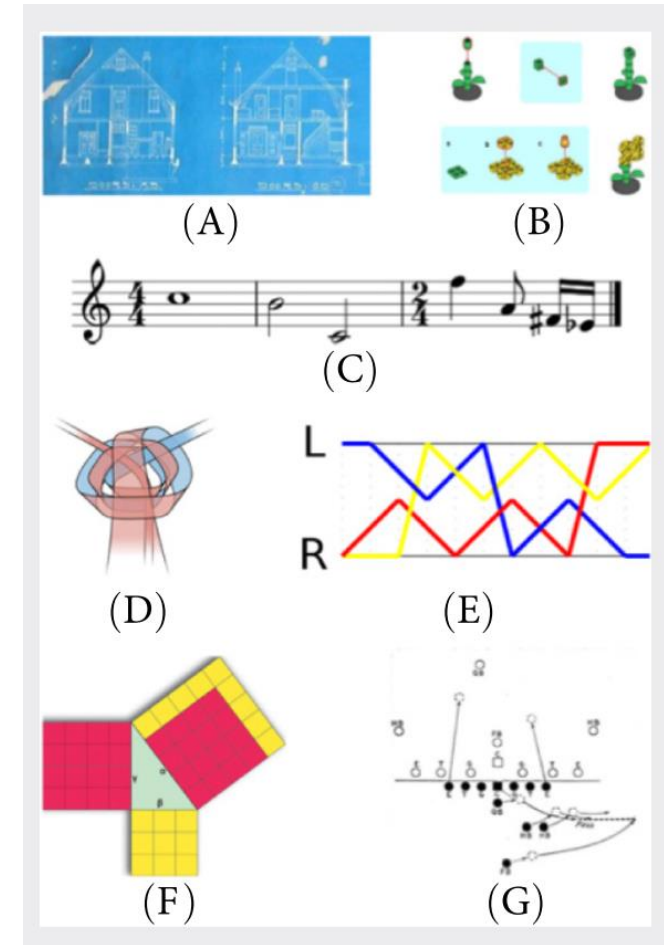
Starting point

pLoT is powerful but limited to:

- **stochastic** search
- for **descriptions** of data
- with a **simplicity** preference

What else can we do with the pLoT?

- pLoT production *as* computer programs
- Look at strategies of human programmers!



The Child as Hacker

Logic	First-order, modal, deontic logic
Mathematics	Number systems, geometry, calculus
Natural language	Morphology, syntax, number grammars
Sense data	Audio, images, video, haptics
Computer languages	C, Lisp, Haskell, Prolog, L ^A T _E X
Scientific theories	Relativity, game theory, natural selection
Operating procedures	Robert's rules, bylaws, checklists
Games and sports	Go, football, 8 queens, juggling, Lego
Norms and mores	Class systems, social cliques, taboos
Legal codes	Constitutions, contracts, tax law
Religious systems	Monastic orders, vows, rites and rituals
Kinship	Genealogies, clans/moieties, family trees
Mundane chores	Knotting ties, making beds, mowing lawns
Intuitive theories	Physics, biology, theory of mind
Domain theories	Cooking, lockpicking, architecture
Art	Music, dance, origami, color spaces

Rule et al. 2020, *The Child as Hacker*.

Programs = general-purpose representations for knowledge, inference, planning

Human learning = program induction

Hacking:

- making code better
- along many dimensions
- through an open-ended and internally motivated set of goals and activities.

The Child as Hacker – New goals

Accurate	Demonstrates mastery of the problem; inaccurate solutions hardly count as solutions at all
Concise	Reduces the chance of implementation errors and the cost to discover and store a solution
Easy	Optimizes the effort of producing a solution, enabling the hacker to solve more problems
Fast	Produces results quickly, allowing more problems to be solved per unit time
Efficient	Respects limits in time, computation, storage space, and programmer energy
Novel	Solves a problem unlike previously solved problems, introducing new abilities to the codebase
Useful	Solves a problem of high utility
Modular	Decomposes a system at its semantic joints; parts can be optimized and reused independently
General	Solves many problems with one solution, eliminating the cost of storing distinct solutions
Robust	Degrades gracefully, recovers from errors, and accepts many input formats
Minimal	Reduces available resources to better understand some limit of the problem space
Elegant	Emphasizes symmetry and minimalism common among mature solutions
Portable	Avoids idiosyncrasies of the machine on which it was implemented and can be easily shared
Clear	Reveals code's core structure to suggest further improvements; is easier to learn and explain
Clever	Solves a problem in an unexpected way
Fun	Optimizes for the pleasure of producing a solution

The Child as Hacker – New strategies

Tune parameters	Adjust constants in code to optimize an objective function.
Add functions	Write new procedures for the codebase, increasing its overall abilities by making new computations available for reuse.
Extract functions	Move existing code into its own named procedure to centrally define an already common computation.
Test and debug	Execute code to verify that it behaves as expected and fix problems that arise. Accumulating tests over time increases code's trustworthiness.
Handle errors	Recognize and recover from errors rather than failing before completion, thereby increasing robustness.
Profile	Observe a program's resource use as it runs to identify inefficiencies for further scrutiny.
Refactor	Restructure code without changing the semantics of the computations performed (e.g., remove dead code, reorder statements).
Add types	Add code explicitly describing a program's semantics, so syntax better reflects semantics and supports automated reasoning about behavior.
Write libraries	Create a collection of related representations and procedures that serve as a toolkit for solving an entire family of problems.
Invent languages	Create new languages tuned to particular domains (e.g., HTML, SQL, \LaTeX) or approaches to problem solving (e.g., Prolog, C, Scheme).

The Child as Hacker – New questions

How might traditional accounts of cognitive development be usefully reinterpreted through the lens of hacking? How can core knowledge be mapped to an initial codebase? How can domain-specific knowledge be modeled as code libraries? What chains of revisions develop these libraries? How do libraries interact with each other? **Which hacking techniques are attested in children and when do they appear?** Which values? How can individual learning episodes be interpreted as improving code?

What are children's algorithmic abilities? How do they learn in the absence of new data? What aspects of learning are data-insensitive? How do they extract information from richly structured data? What kinds of nonlocal transformations do we see? Do children ever find more complex theories before finding simpler ones? **How do children move around the immense space of computationally expressive hypotheses?**

The Child as Hacker – New questions

How do humans program? What techniques do they use? What do they value in good code? How do they search the space of programs? Does the use of many techniques make search more effective?

How can the discoveries of computer science best inform models of human cognition? For example, what remains to be learned about human cognition from the study of compilers, type systems, or databases? How can we use the vocabulary of programming and programming languages to more precisely characterize the representational resources supporting human cognition? **Are things like variable binding, symbolic pattern matching, or continuations cognitively primitive?** If so, are they generally available or used only for specific domains? How does the mind integrate symbolic/discrete and statistical/continuous information during learning? What kinds of goals do children have in learning?

MCMC w/ a formal grammar

(If there's time left)

Conclusions

Learning a rule

Robert Feldman → Dr Feldman

Ruth Millican → Dr Millikan

Joanna Newsom → Dj Newsom

“Dr <*last name*>” **or** “D<*first letter of first name*> <*last name*>”?

6 @ 2 = 12

3 @ 4 = 12

10 @ 2 = 12

@ = multiplication **or**

return 12?



Concepts
generation



Classification



Example
generation



Lake et al (2015)



Parsing

Some closing considerations

The pLoT is a *new* framework...
...built on solid foundations.

I tried to convey a Fodorian *picture*.

Much left to do in this program.

Part I	Introduction: On the very idea of an LoT
Part II	Technical background
Part III	Bayesian program induction (LOTlib3)
Part IV	Case studies
Part V	Summary & Future prospects

A concluding note on being one self



If, in short, there is a community of computers living in my head, there had also better be somebody who is in charge; and, by God, it had better be me.

(Fodor 1988)

Thanks everyone!