EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Conclusions

Or: DreamCoder and conclusive remarks

- *The Language of Thought: computational cognitive science approaches to category learning*
- Who: Fausto Carcassi
- When: Sommer semester 2022

Where are we?

- We have finally reached the last week!
- We have covered quite a lot of stuff:
  - The philosophical background on the Language of Thought (Fodor)
  - Some technical background
    - Formal grammars
    - Compositional semantics with lambda calculus
    - Bayesian inference and MCMC for approximating a posterior
  - The *probabilistic* Language of Thought
  - The LOTlib3 library
  - Some applications to various conceptual domains

Where are we?

- Today let's have a look at two things:
  - What the state of the art is (DreamCoder)
  - Where the people in the field see it going (The Child as a Hacker)
- In the lab this week we will again write together a little inference script for a conceptual domain
  - I was thinking we could do a little model to fit a function with some simple operations
  - So we might get some input-output combos ([0, 0.3], [0.4, -0.1], …) and we have to infer an expression that encodes them (e.g., y = log(x)-2)

DreamCoder: The general idea

- The problem: Program Induction / pLoT is not *scalable* compared to neural network systems!
- Why do you think that is?
- Program Induction systems need to start with a *domain specific* language.
  - This is partially because if we start with domain general primitives the programs we need to solve practical problems become so long that they can't be inferred
- 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 attempts to solve both of these problems!

DreamCoder: The general idea

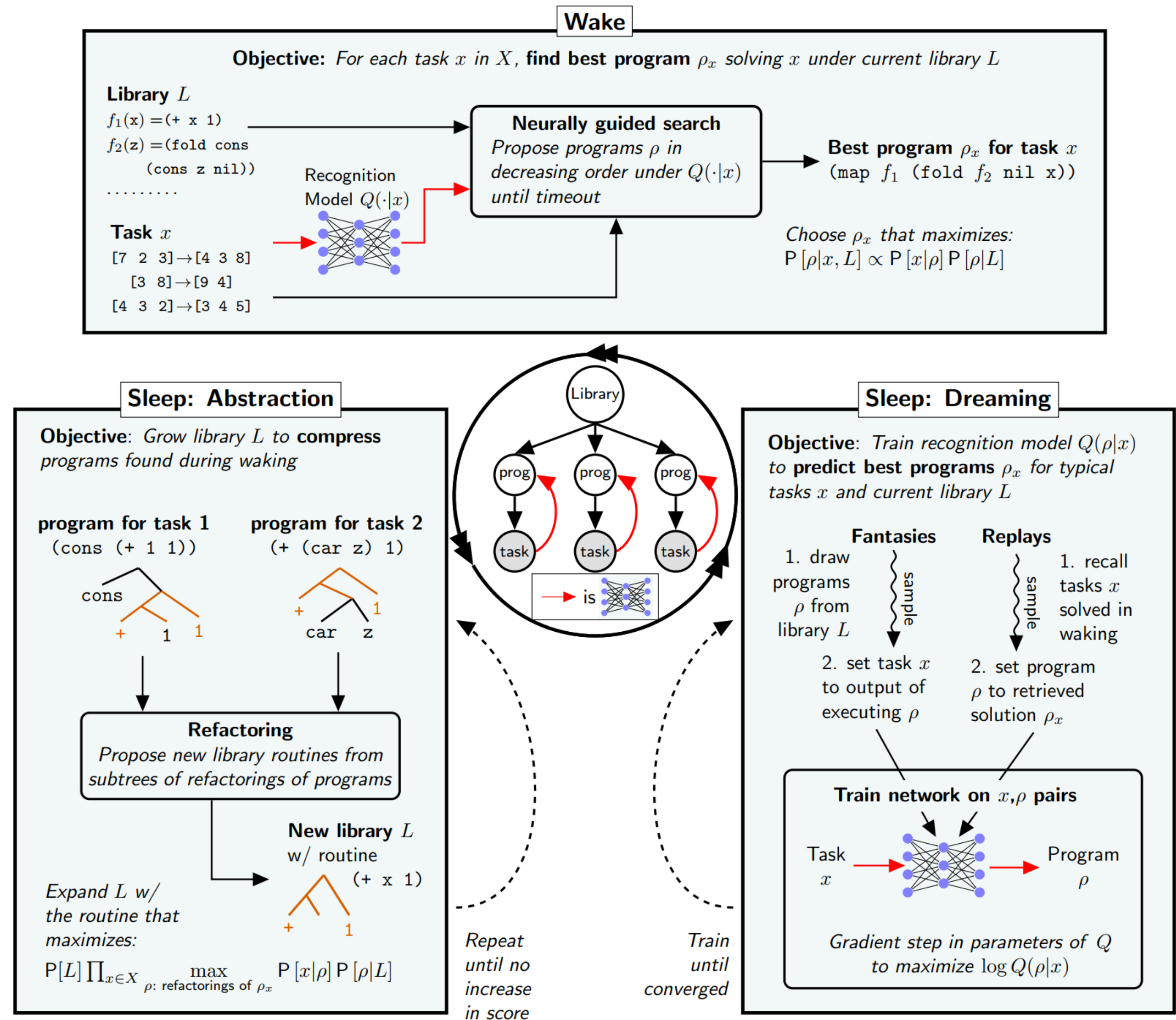
Two related ideas:

- Instead of a fixed list of primitives, *grow* new concepts, so that the inferred programs are shorter!
    - This library of new learned concepts will depend on the specific domain, so that DreamCoder *grows* a domain-specific language.
- Learn *implicit procedural knowledge*
    - This means that observing the object to be inferred gives us some ideas about which concepts we are going to need and how
    - This is what we do when we actual program, and it relies on *intuition*, and so the most natural way is to use a neural networks that goes from the data to the probability of each substitution rule!

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

DreamCoder: Learning

- Various things need to be done:
  - How do we grow new concepts?
  - How and when is the ANN trained?
- Solution: Wake/Sleep Program Learning



DreamCoder: Results

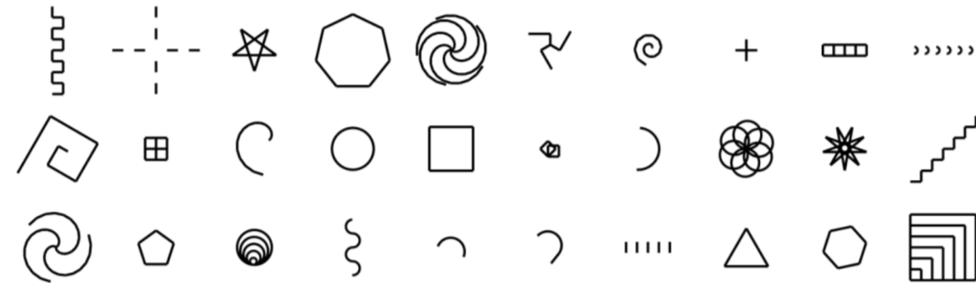
EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- First test. Two classic domains: List processing and text editing (218 problems)
  - In both domains, the program defines a function
  - DreamCoder starts with a domain-general functional basis
- Each round of abstraction built on concepts discovered in earlier 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, which it finally uses to sort lists of numbers
- DreamCoder solves 84.3% of the problems with 1 hour & 8 CPUs per problem.
  - The best-performing synthesizer in this competition (CVC4) solved 82.4% of the problems.
  - CVC4 had a *different* hand-engineered library of primitives for each text editing problem!

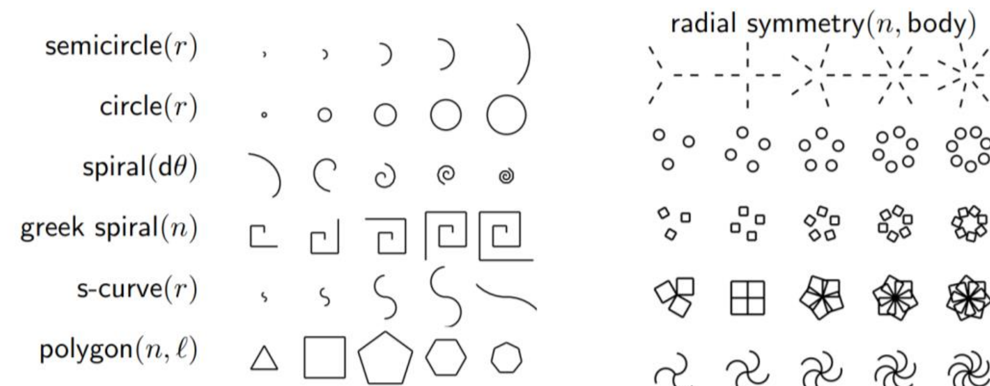
DreamCoder: Results

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- Second test. More creative domains: generating images, plans, and text.
- Programs to learn (30 out of 160) for LOGO shapes task:

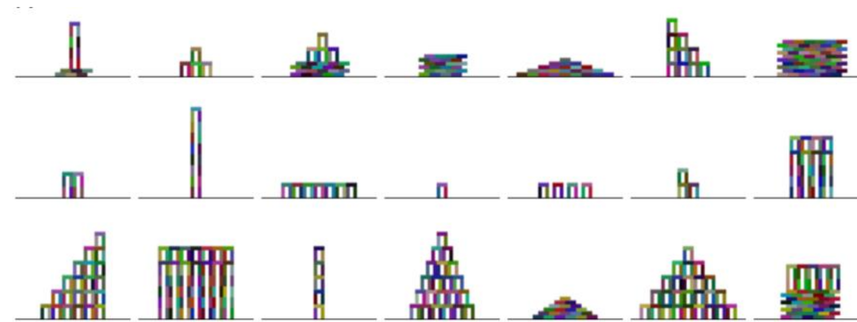


- Some learned parametric ‘shape concepts’ and higher order function:

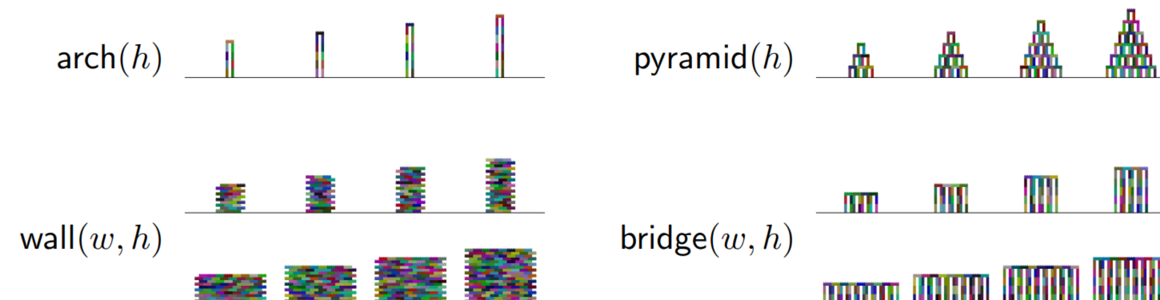


DreamCoder: Results

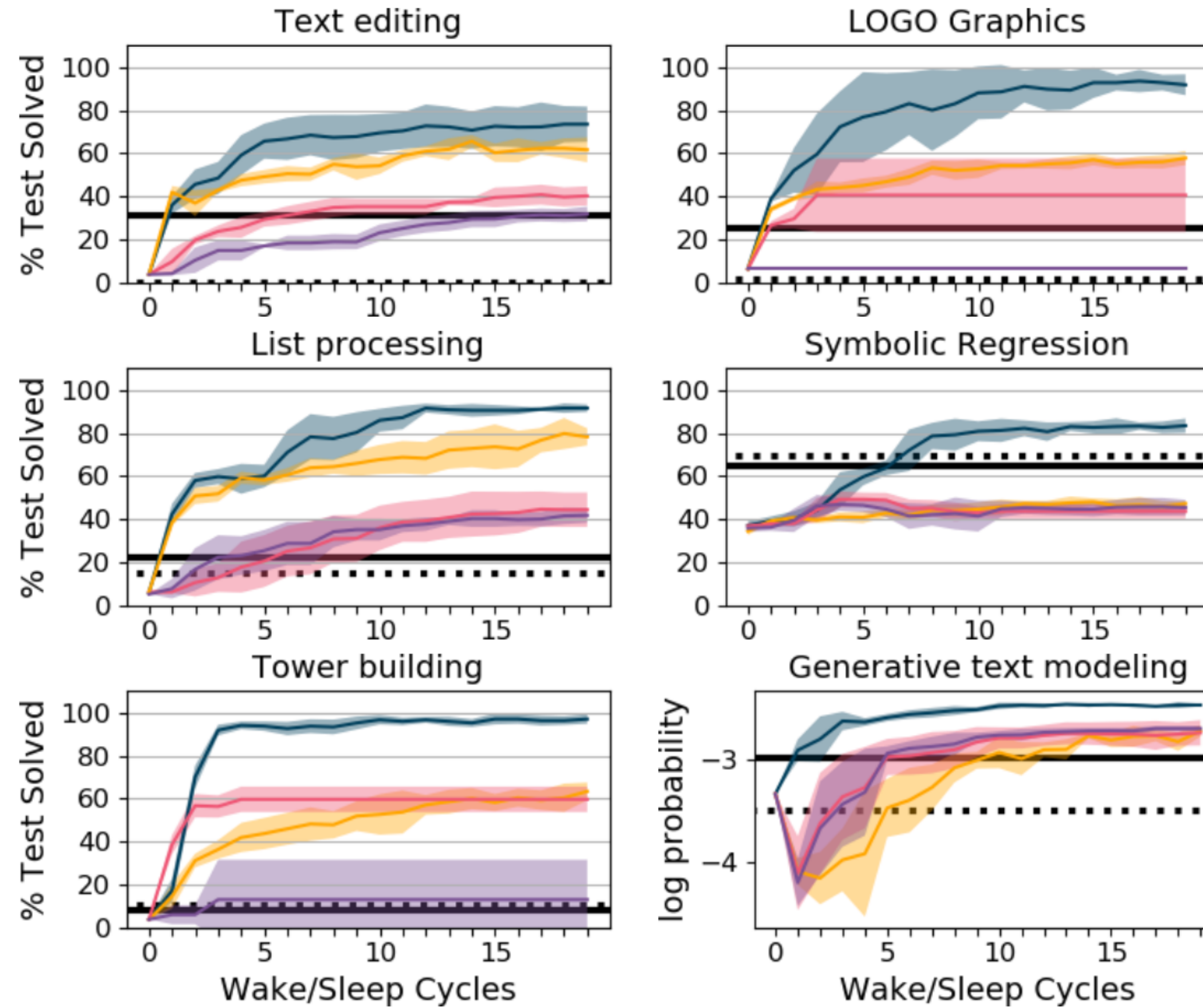
- Second test. More creative domains: generating images, plans, and text.
- Programs to learn for towers building:



- Some learned parametric ‘shape concepts’ and higher order function:



DreamCoder: Results



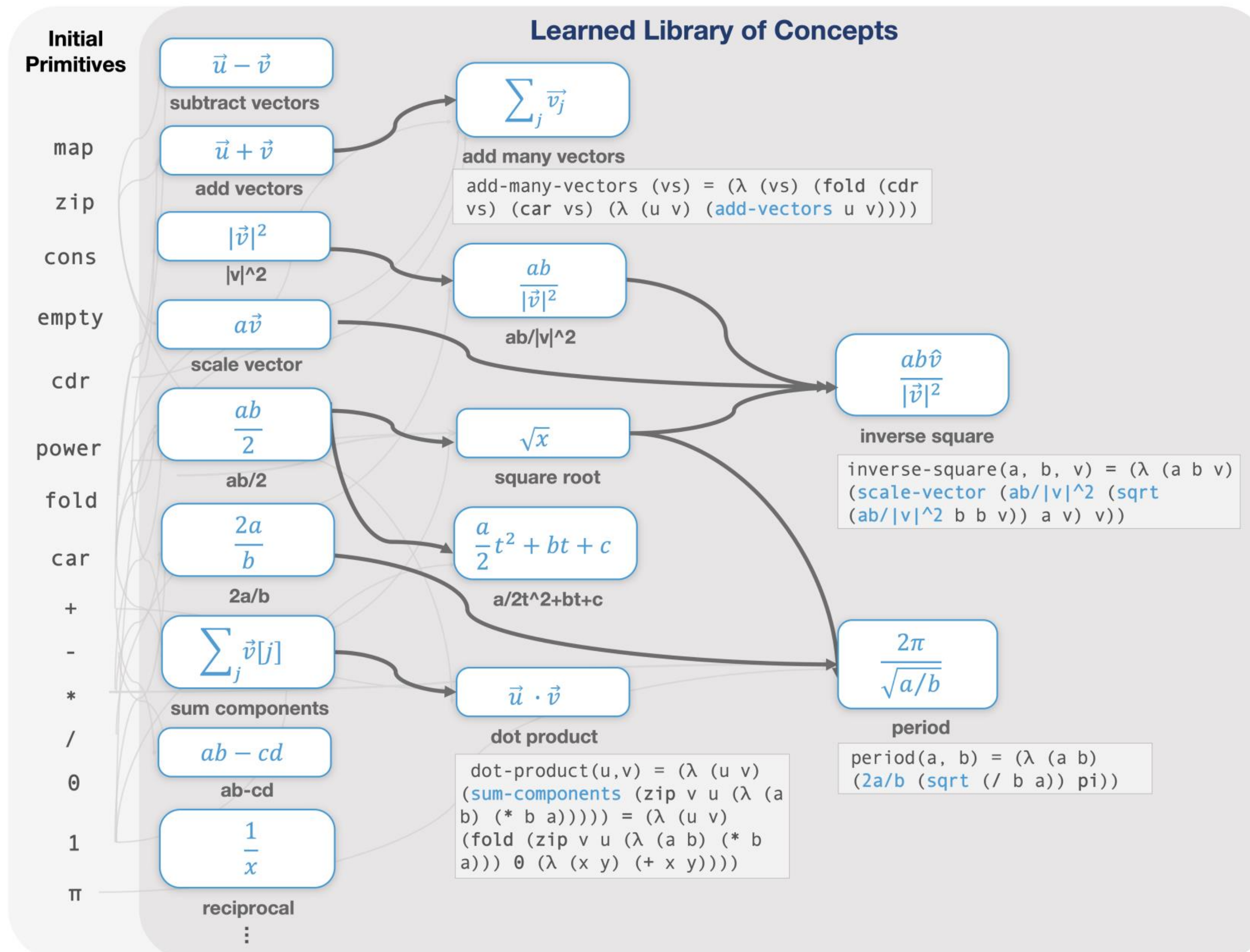
DreamCoder: Results

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- Third set of tasks: Learning whole languages!
- Question: Can DreamCoder solve complex problems starting just with super general operations and progressively building complex concepts?
- 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 learning conceps like inner product, vector sum, and norm
    - Then learns complex like the inverse square law
    - Finally uses these to formulate the actual laws like Newton's laws of gravitation and Coulomb's law of electrostatic force

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

DreamCoder: Results



Discovered Physics Equations

Newton's Second Law

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

```
(scale-vector(reciprocal m)
(add-many-vectors Fs))
```

Parallel Resistors

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

```
(reciprocal (sum-components
(map (λ(r) (reciprocal r))
Rs)))
```

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

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

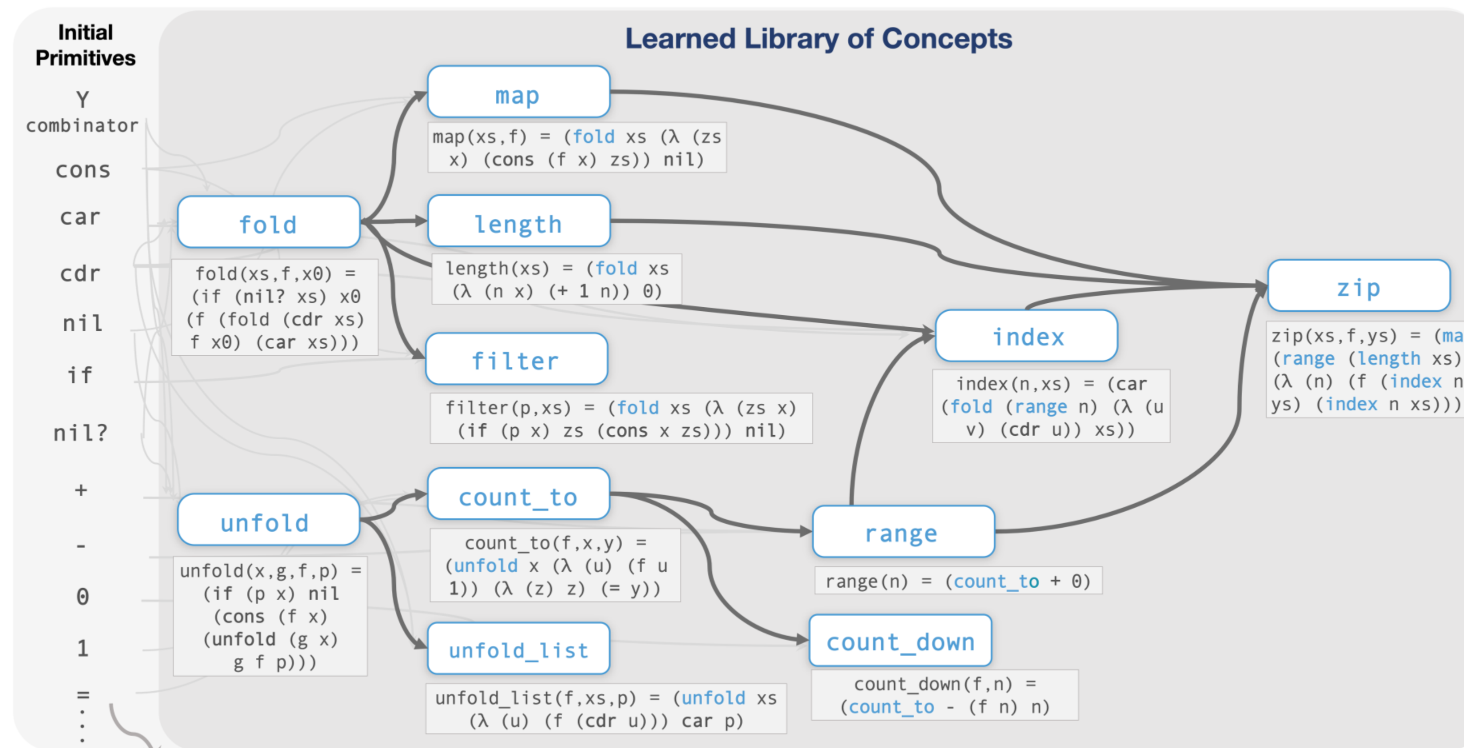
```
(inverse-square q_1 q_2
(subtract-vectors r_1 r_2))
```

```
(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: Results

- They also did it with recursive algorithms:



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 )
```

```
(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)))))))
```

Solution to `list differences` if expressed in initial primitives

DreamCoder: Results

- Overall, I find DreamCoder pretty mindblowing
- All the code is available online and you can get it running on your machine in about half a day
- There is much more to explore in this direction!

Future directions: The Child as a Hacker

- Rule et al. 2020, *The Child as a Hacker*.
- Citing the paper:
  - Programs provide our best general-purpose representations for human knowledge, inference, and planning; human learning is thus increasingly modeled as program induction, learning programs from data.
  - Many formal models of learning as program induction reduce to a stochastic search for concise descriptions of data. Actual human programmers and learners are significantly more complex, using many processes to optimize complex and frequently changing objectives.
  - The goals and activities of hacking, making code better along many dimensions through an open-ended and internally motivated set of goals and activities, are helping to inspire better models of human learning and cognitive development

Future directions: The Child as a Hacker

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- The main point is that we use algorithmic knowledge in all sorts of domains.
- All of these could be modelled in an LoT framework.

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, $\LaTeX$
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

Future directions: The Child as a Hacker

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- The main conceptual move in this paper is that the kind of pLoT theory we have looked at in this course focused on just one dimension
- But actual program development considers many dimensions to optimize!
- Some examples:

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

Future directions: The Child as a Hacker

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- Moreover, there's many strategies to building the program, which weren't captured by our inference algorithm.
- Some examples:

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).

Future directions: The Child as a Hacker

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- **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?**
- **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? What improvements do they inspire? How do they move around the space of goals? What data structures does this movement suggest for goal management?

The end!

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- And that's it for the lectures this course folks!
- Let me know when you want to discuss topics for the final project.
  - I'll be out of Tubingen until the end of August
- I'll still see you on Wednesday
  - We'll implement another model from scratch
- Any questions?
- I was asked to let you write evaluations during class time