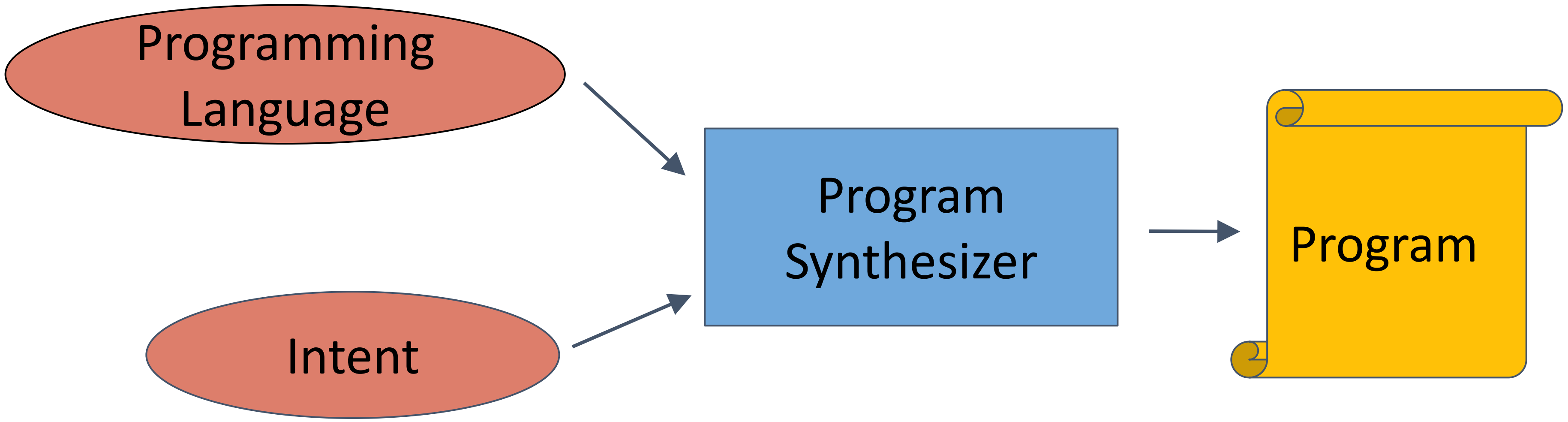
Deep Learning for Program Synthesis: Lessons and Challenges

Dawn Song

UC Berkeley

Program Synthesis

Can we teach computers to write code?



Example Applications:

- End-user programming
- Performance optimization of code
- Virtual assistant

Programming Synthesis via Learning

- How to specify programming intent
  - natural language description
  - input-output examples
  - translation
- How to represent generated program
  - neural networks (fully differentiable)
  - discrete code (non-differentiable)
  - hybrid (combining differentiable & non-differentiable components)

Programming Synthesis as a Perfect Playground for Intelligence

- Ultimate challenge for AGI
  - Building robots without being limited by Physics
  - Need ability to
    - Model the world
    - Define goals & decompose goals
    - Abstract & Reason
    - Plan & Search

Synthesis via Learning: a Powerful Lens

- Target of synthesis
  - Programs:
    - Program synthesis
    - Learning-based program optimization
  - Models:
    - model synthesis; autoML
  - Proofs:
    - proof synthesis; automatic theorem proving
  - Action plan:
    - Robot action plan synthesis/agent synthesis
  - Games, creations
    - Creative synthesis

Example Program Synthesis

- Natural language description translating to code---end-user programming
  - IFTTT programs [NIPS 2017]
  - SQL queries

- Generalization and proof of guarantee for neural program synthesis [ICLR 2017]

- Other examples:
  - Hybrid neural program synthesis: Learning a neural program operating a non-differentiable machine
    - Learning program parser using I/O examples [ICLR 2018]
  - Hierarchical options for neural programming
    - Parameterized hierarchical procedures for neural programming [ICLR 2018]
  - Automating theorem proving using deep learning
    - Coq proof dataset/tool available
    - GamePad: A Learning Environment for Theorem Proving
      Daniel Huang, Prafulla Dhariwal, Dawn Song, Ilya Sutskever

Lessons & Challenges in Program Synthesis via Learning

- Generalization
- Evaluation
  - Be careful with your test set
- Scalability
  - Combining discrete & differentiable approaches
  - Learning abstractions
- Adapt to new tasks
  - Accumulate knowledge from past experience
- What should be a good benchmark suite for program synthesis?

Neural Program Synthesis

Training data	123 234	452 345	612 367	**Input**
	357	797	979	**Output**

Neural Program Synthesis

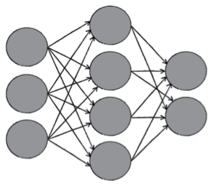
Training data

123 234	452 345	612 367	**Input**
357	797	979	**Output**

Neural Program Architecture

50
70

Test input

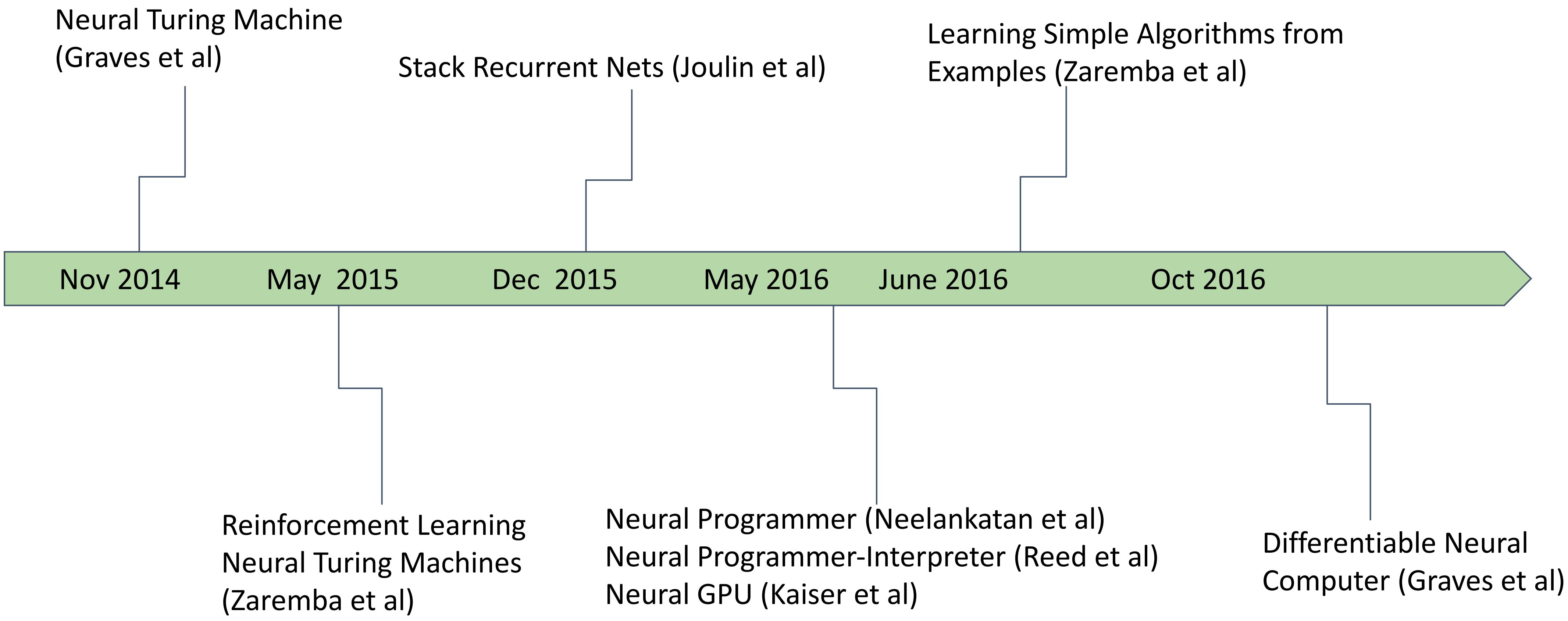


Learned neural program

120

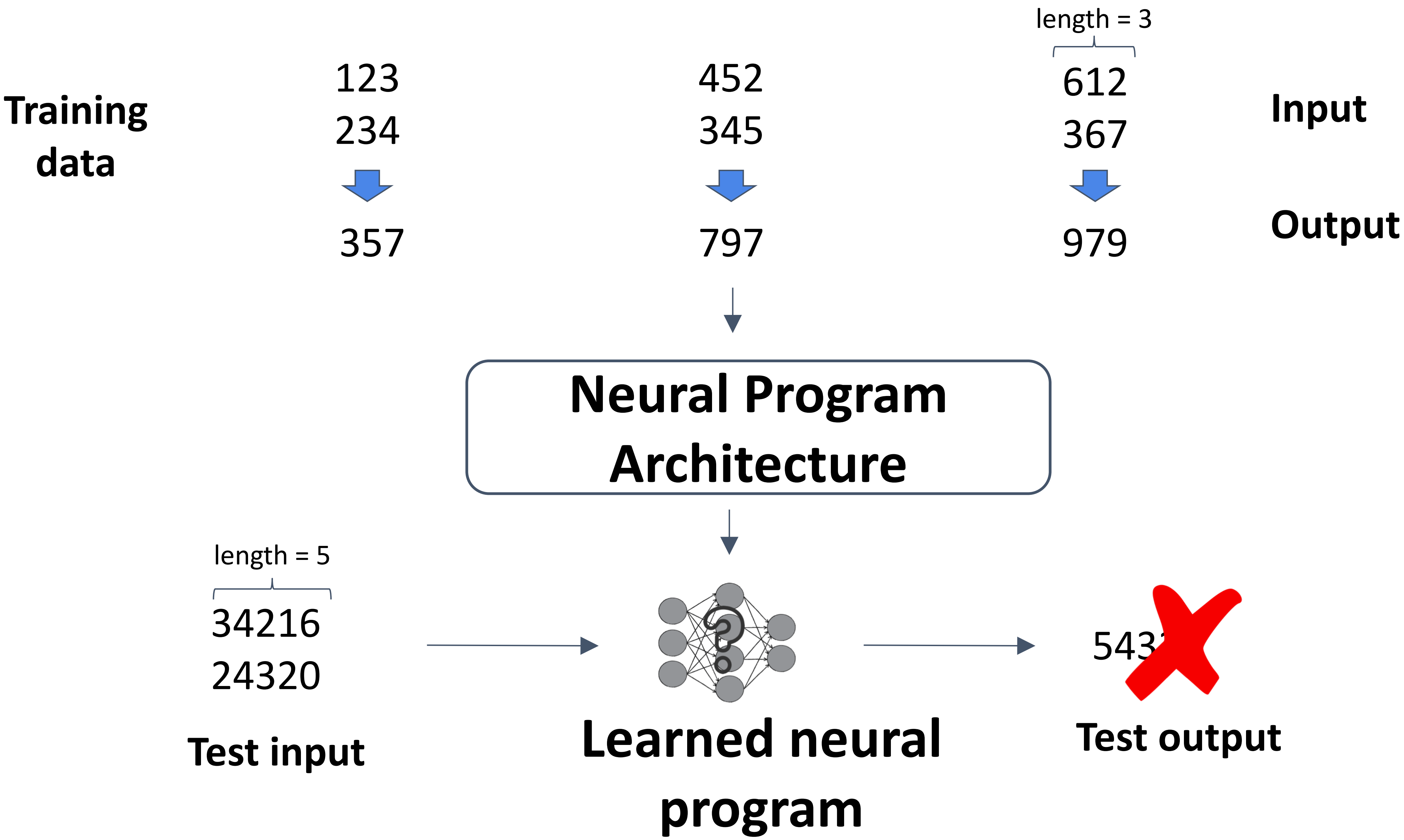
Test output

Neural Program Architectures

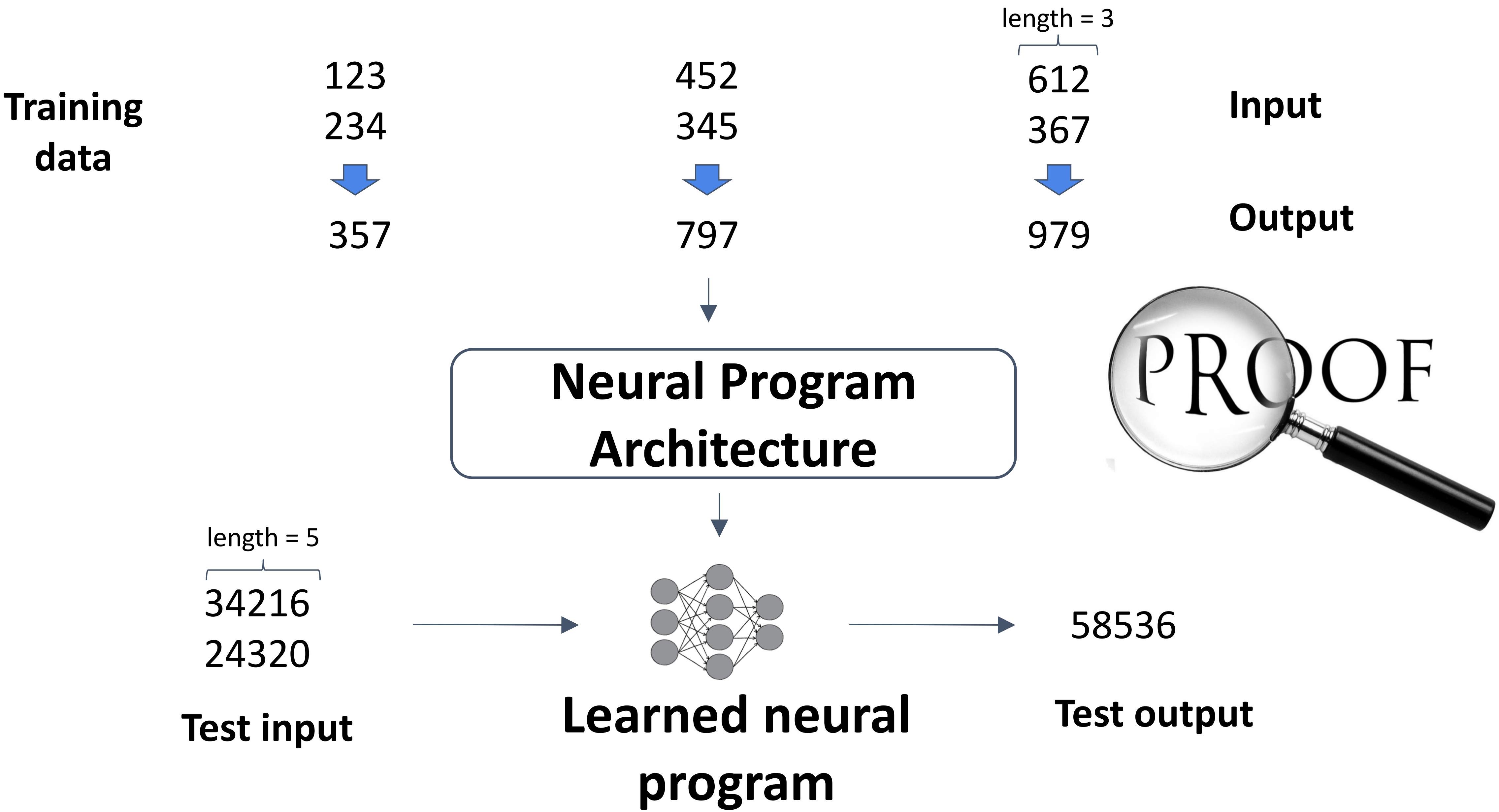


Neural Program Synthesis Tasks: Copy, Grade-school addition, Sorting, Shortest Path

Challenge 1: Generalization



Challenge 2: No Proof of Generalization



Our Approach: Introduce Recursion

Learn recursive neural programs

Jonathon Cai, Richard Shin, Dawn Song: Making Neural Programming Architectures Generalize via Recursion
[ICLR 2017, **Best Paper Award**]

Recursion

- Fundamental concept in Computer Science and Math
- Solve whole problem by reducing it to smaller subproblems (*reduction rules*)
- *Base cases* (smallest subproblems) are easier to reason about

Quicksort

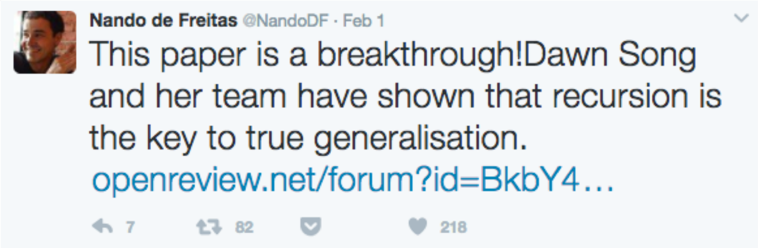
Our Approach: Making Neural Programming Architectures Generalize via Recursion

- **Proof of Generalization**:
  - Recursion enables provable guarantees about neural programs
  - Prove perfect generalization of a learned recursive program via a verification procedure
    - Explicitly testing on all possible base cases and reduction rules (Verification set)

- Learn & generalize faster as well
  - Trained on same data, non-recursive programs do not generalize well

Accuracy on Random Inputs for Quicksort

Length of Array	Non-Recursive	Recursive
3	100%	100%
5	100%	100%
7	100%	100%
11	73.3%	100%
15	60%	100%
20	30%	100%
22	20%	100%
25	3.33%	100%
30	3.33%	100%
70	0%	100%



Jonathon Cai, Richard Shin, Dawn Song: Making Neural Programming Architectures Generalize via Recursion [ICLR 2017, **Best Paper Award**]

Lessons

- Program architecture impacts generalization & provability
- Recursive, modular neural architectures are easier to reason, prove, generalize
- Explore new architectures and approaches enabling strong generalization & security properties for broader tasks

Lessons & Challenges in Program Synthesis via Learning

- Generalization
- Evaluation
  - Be careful with your test set
- Scalability
  - Combining discrete & differentiable approaches
  - Learning abstractions
- Adapt to new tasks
  - Accumulate knowledge from past experience
- What should be a good benchmark suite for program synthesis?

Background

Some past work in neural program synthesis from input-output examples:

- String transformations: Neuro-Symbolic Program Synthesis (Parisotto et al 2017), RobustFill (Devlin et al 2017)
- Array manipulation: DeepCoder (Balog et al 2017)
- Karel: Bunel et al 2018

These methods learn to search over possible programs, using **supervised learning** with a large **synthetic** training dataset.

Hypothesis of training on synthetic data:
Given a large enough random training set, the neural program synthesis model will work well on arbitrary tasks.

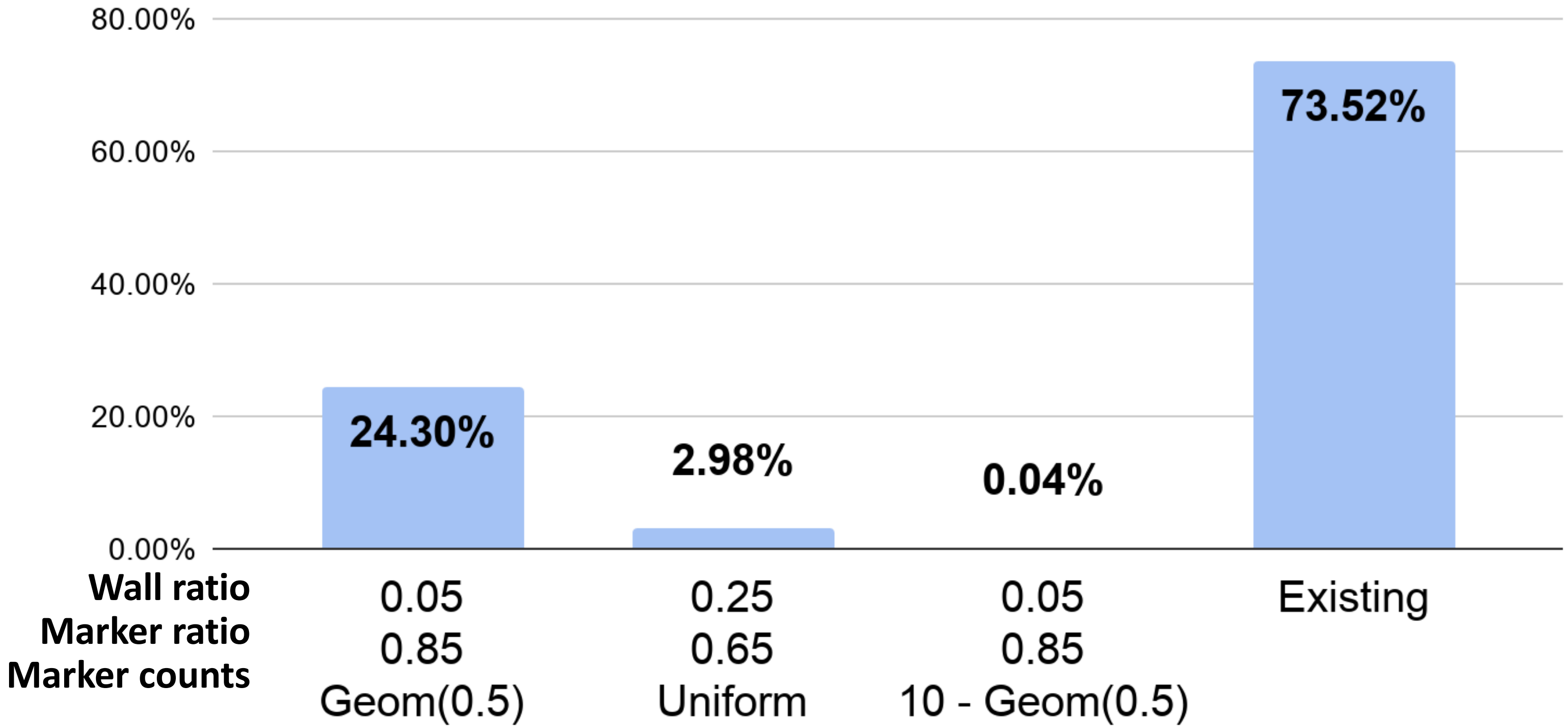
Our findings:
These models can be **highly sensitive** to how the random data was generated for more complex domains such as Karel.

Choosing different I/O examples or programs can decrease accuracy to **0%.**

New data generation methodology needed for training similar models.

Testing with new distributions over I/O examples

By changing the distribution over I/O examples, we can lower performance down to **0.04%**.



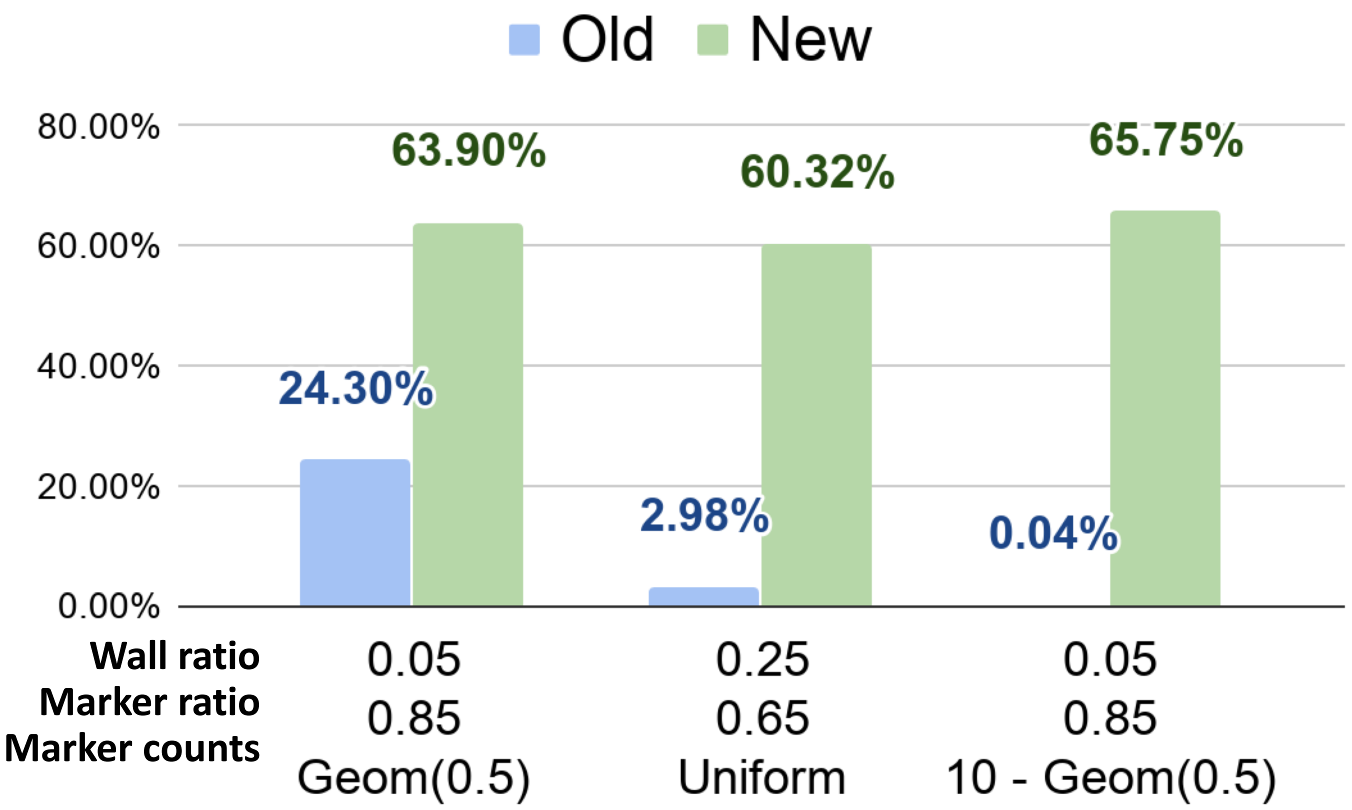
Best (of 12)

Worst (of 12)

Augmenting the training data

If we retrain a **single** model on a more *uniform* distribution over possible I/O, we significantly recover performance on the specialized distributions.

Note: the training distribution is **not** simply a union of the test distributions.



Lessons

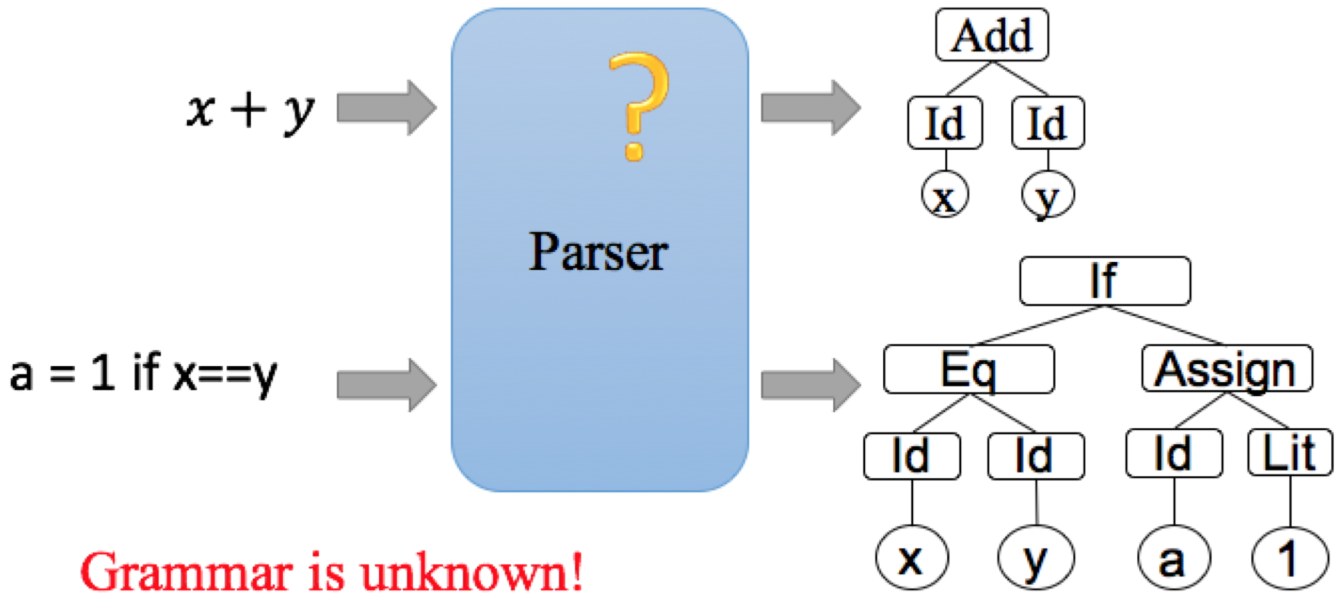
- Randomly generated datasets can be unexpectedly biased in various ways
- Simple methods for random sampling may be insufficient
- Important to consider distributions over inputs, as well as programs
- New methodology for synthetic training data:
  - Define various **salient random variables** that capture desired features of the input space and the program space
  - Ensure **uniformity** the random variables as much as possible
- Training with our new methodology leads to significant performance improvement on various test sets

Lessons & Challenges in Program Synthesis via Learning

- Generalization
- Evaluation
  - Be careful with your test set
- Scalability
  - Combining discrete & differentiable approaches
  - Learning abstractions
- Adapt to new tasks
  - Accumulate knowledge from past experience
- What should be a good benchmark suite for program synthesis?

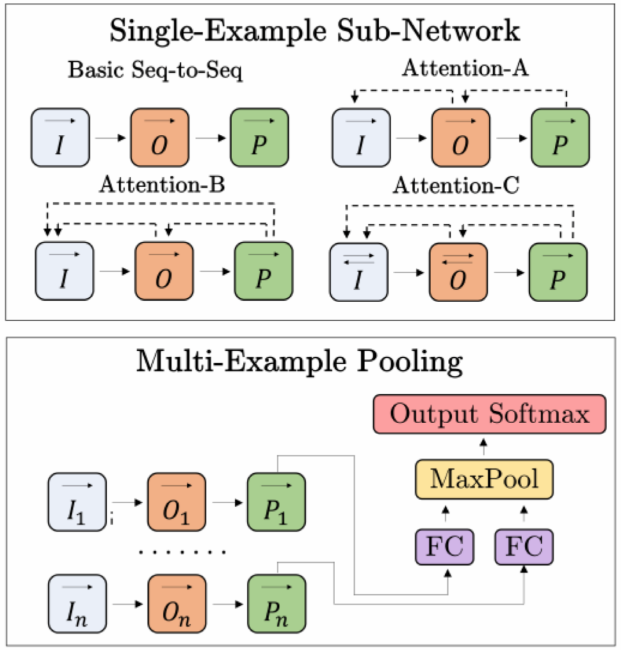
Neural Parser Synthesis

- Task: learning a parser as input-output program synthesis



Existing Approaches

- End-to-end neural networks: sequence-to-sequence based models
  - Do not generalize well.
  - Require a lot of training samples.
  - In our evaluation, we demonstrate that the test accuracies of this type of models are 0% when the test samples are longer than training ones.
- Neural symbolic program synthesis
  - R3NN [4], RobustFill [5].
  - Expressiveness of the program DSL is limited.
  - The lengths of the synthesized programs are up to 20.

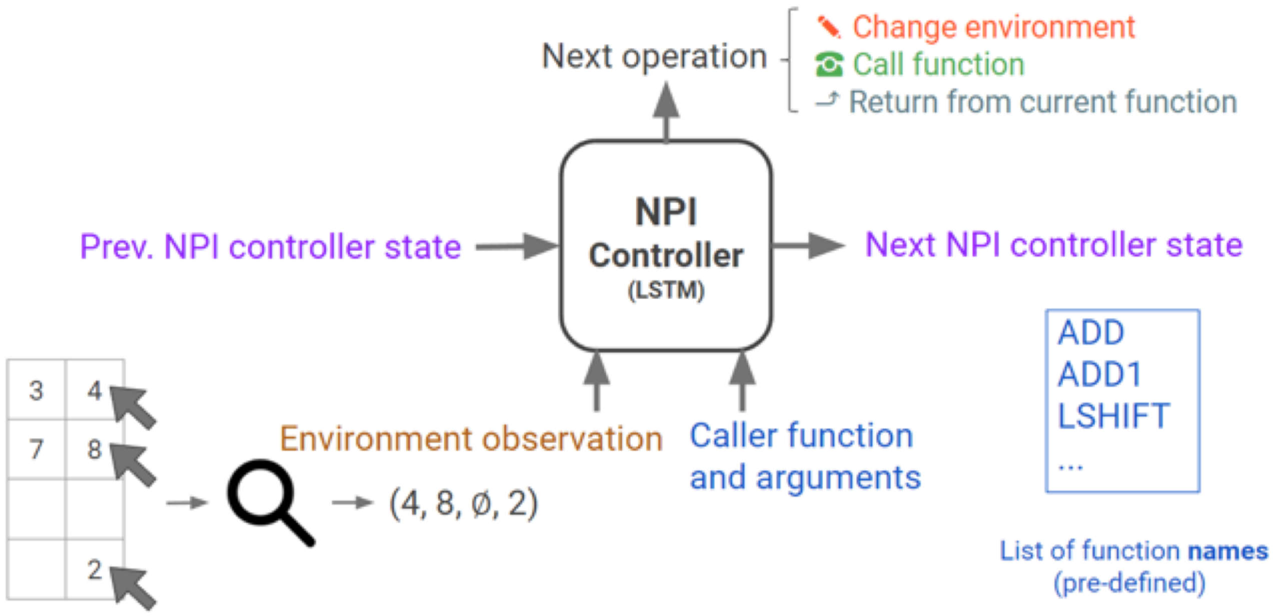


RobustFill model architecture [5]

[4] Parisotto et al. Neural-Symbolic Program Synthesis, ICLR 2017.
[5] Delvlin et al. RobustFill: Neural Program Learning under Noisy I/O, ICML 2017.

Existing Approaches

- NPI-like approaches
  - Training requires supervision on execution traces.
  - The complexity of the learned algorithm is limited.



Neural Programmer-Interpreter [6, 7]

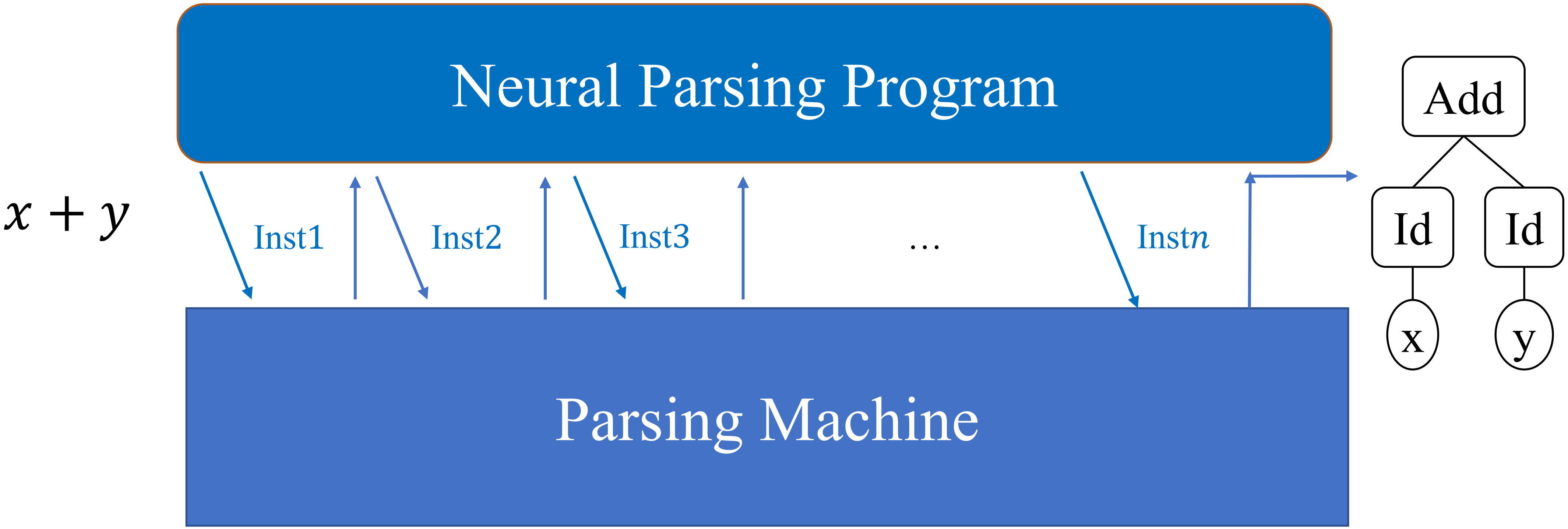
[6] Scott Reed, Nando de Freitas, Neural Programmer-Interpreters, ICLR 2016.
[7] Jonathon Cai, Richard Shin, Dawn Song, Making Neural Programming Architectures Generalize via Recursion, ICLR 2017.

Goals

- Supervision on I/O pairs only
  - No supervision on execution traces
- Full generalization
  - 100% accuracy on arbitrarily long inputs
- Train with a few examples

Our Approach

- **Differential neural programs** operating a **non-differentiable machine**



[Towards Synthesizing Complex Programs from Input-Output Examples](), Xinyun Chen, Chang Liu, Dawn Song. International Conference on Learning Representations (ICLR), 2018.

LL Machine

State

Input Stream

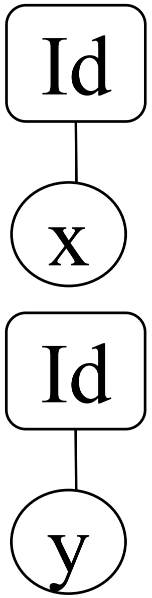
+z

Stack

1	(Id, T2)
0	(Id, T1) (+,+)

T1

T2



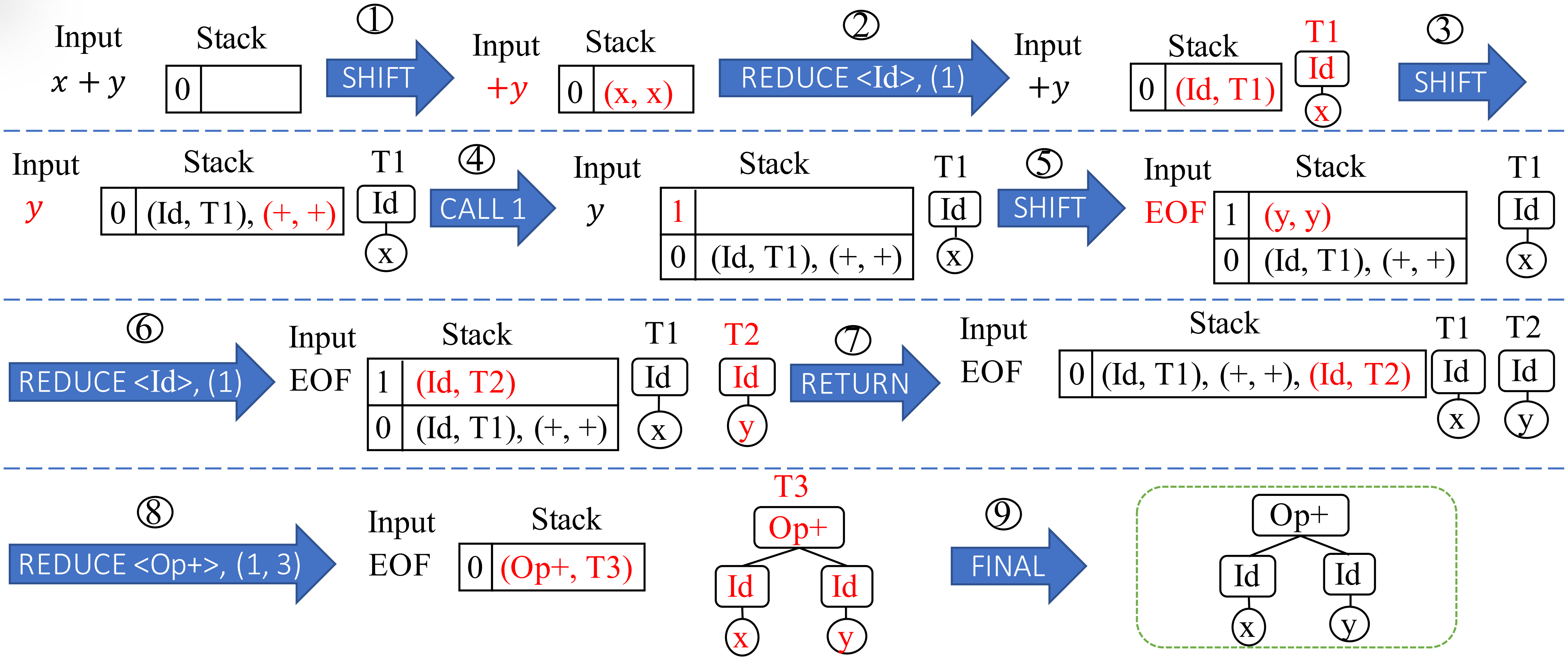
Instruction Set

Parser Functionality
- Shift
- Reduce

Stack Operation
- Call
- Return

Termination
- Final

LL Machine

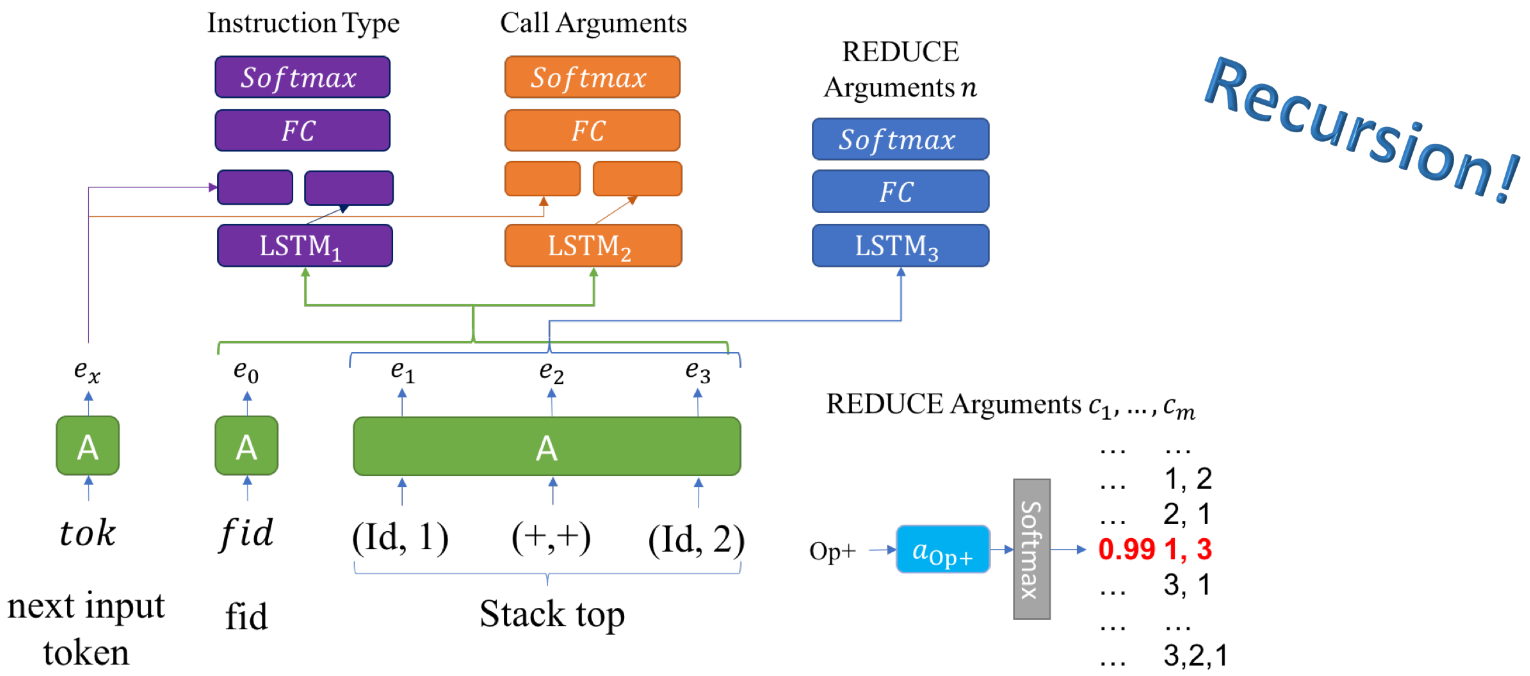


Output prediction → Trace prediction

Provide constraints on the learned parsing programs

Differential Neural Program

- Given the current state of the machine, predict which instruction to be executed next
- Using LSTM to predict instruction types and arguments
- **Prediction is only based on stack top and the next token**

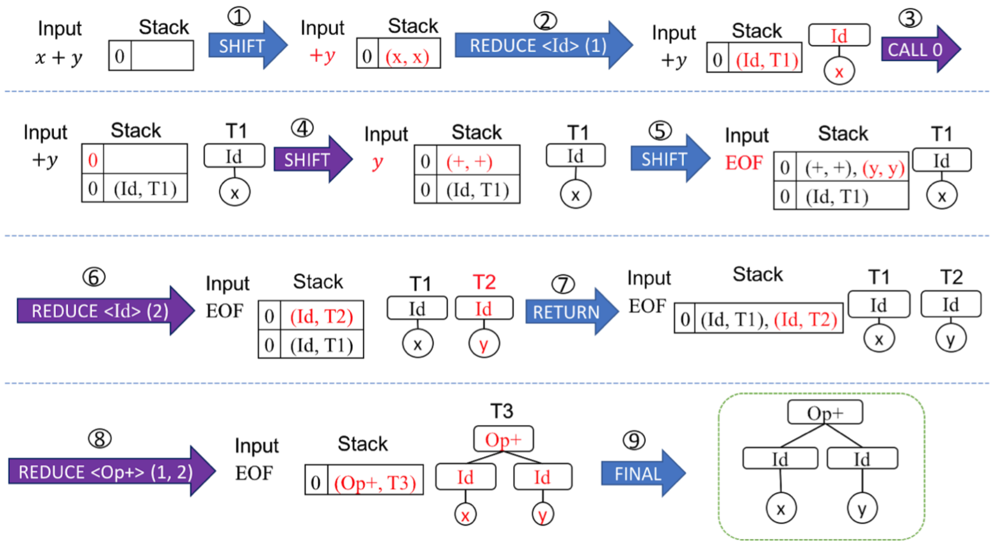


Challenge: execution traces are unknown!

- The space of possible execution traces is very large
    - For a very simple input (e.g., of length 3), it requires 9 instructions to construct the parse tree.
- Policy gradient could easily get stuck at a local optimum.

Input length	Number of shortest valid execution traces (under-estimation)
3	1572
5	2,771,712
7	7,458,826,752

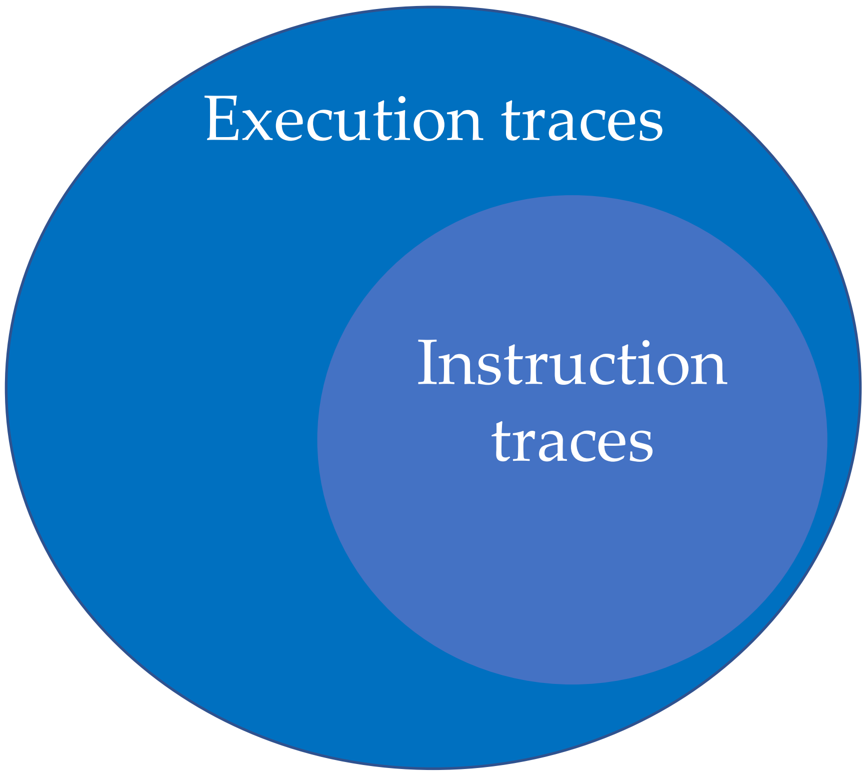
For illustration purposes, here we consider the grammar that includes only addition and multiplication, which is a small subset of the grammars in our evaluation.



An example of the alternative trace that leads to the correct output.

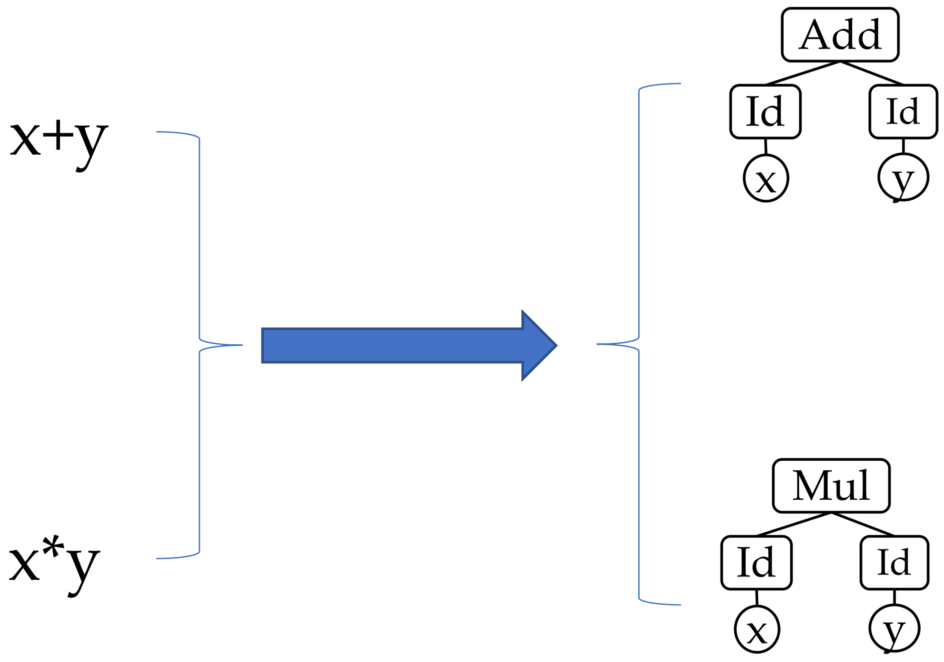
Reinforcement Learning-based Two-phase Search Algorithm

Instruction



Phase I

Argument



Phase II

Reinforcement Learning-based Two-phase Search Algorithm

- Two-phase learning
  - **Input-output pairs only learning**: use policy gradient to sample instruction traces, and rely on weakly supervised learning to verify if the trace is good or not.
  - **Weakly supervised learning**: assuming instruction traces are provided, train the argument prediction networks (policy-gradient with specially designed reward functions).

No supervision on the traces $\xrightarrow[\text{found in Phase I}]{\text{Valid instruction traces}}$ Weak supervision on instruction traces

Evaluation

While-Lang

Train	Test	Ours	Seq2seq	Seq2tree	Stack LSTM	Queue LSTM	DeQue LSTM	Robust-Fill (Projected)
Curriculum	Training	**100%**	81.29%	**100%**	**100%**	**100%**	**100%**	13.67%
	Test-10	**100%**	0%	0.8%	0%	0%	0%	0%
	Test-100	**100%**	0%	0%	0%	0%	0%	0%
	Test-1000	**100%**	0%	0%	0%	0%	0%	0%
	Test-5000	**100%**	0%	0%	0%	0%	0%	0%
Std-10	Training	**100%**	94.67%	**100%**	81.01%	72.98%	82.59%	0.19%
	Test-10	**100%**	20.9%	88.7%	2.2%	0.7%	2.8%	0%
	Test-100	**100%**	0%	0%	0%	0%	0%	0%
	Test-1000	**100%**	0%	0%	0%	0%	0%	0%
Std-50	Training	**100%**	87.03%	**100%**	0%	0%	0%	0.0019%
	Test-50	**100%**	86.6%	99.6%	0%	0%	0%	0%
	Test-500	**100%**	0%	0%	0%	0%	0%	0%
	Test-5000	**100%**	0%	0%	0%	0%	0%	0%

Evaluation

Lambda-Lang

Train	Test	Ours	Seq2seq	Seq2tree	Stack LSTM	Queue LSTM	DeQue LSTM	Robust-Fill (Projected)
Curriculum	Training	**100%**	96.47%	**100%**	**100%**	**100%**	**100%**	29.21%
	Test-10	**100%**	0%	0%	0%	0%	0%	0%
	Test-100	**100%**	0%	0%	0%	0%	0%	0%
	Test-1000	**100%**	0%	0%	0%	0%	0%	0%
	Test-5000	**100%**	0%	0%	0%	0%	0%	0%
Std-10	Training	**100%**	93.53%	**100%**	0%	95.93%	2.23%	0.26%
	Test-10	**100%**	86.7%	99.6%	0%	6.5%	0.1%	0%
	Test-100	**100%**	0%	0%	0%	0%	0%	0%
	Test-1000	**100%**	0%	0%	0%	0%	0%	0%
Std-50	Training	**100%**	66.65%	89.65%	0%	0%	0%	0.0026%
	Test-50	**100%**	66.6%	88.1%	0%	0%	0%	0%
	Test-500	**100%**	0%	0%	0%	0%	0%	0%
	Test-5000	**100%**	0%	0%	0%	0%	0%	0%

Lessons

- **Neural programs operating a non-differentiable machine** can achieve **100% accuracy** on test inputs with length **500×** longer than training inputs, while an end-to-end neural network's accuracy is 0%.

- The **design of the non-differentiable machine** is crucial to **regularize** the programs that can be synthesized, and **leveraging reinforcement learning algorithms** is the key to train a neural network to learn complex programs.

Lessons & Challenges in Program Synthesis via Learning

- Generalization
- Evaluation
  - Be careful with your test set
- Scalability
  - Combining discrete & differentiable approaches
  - Learning abstractions
- Adapt to new tasks
  - Accumulate knowledge from past experience
- What should be a good benchmark suite for program synthesis?

IMAGENET

Object recognition

VQA

SQuAD

Question answering

?

Program synthesis

Common Crawl

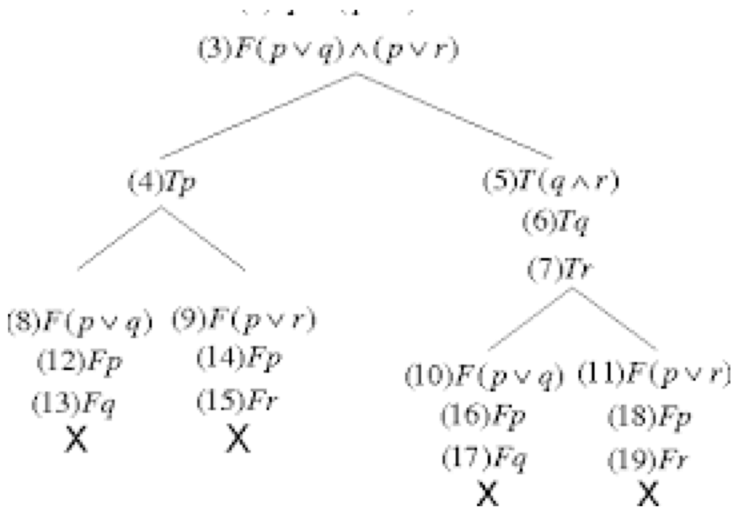
Language modeling

COCO

Common Objects in Context

Image captioning

Theorem proving at multiple levels of abstraction



Automatic (e.g., SMT such as Z3 or tableaux prover)

```
(* This is B & G, Proposition 1.4, for internal actions.                       *)
Proposition coprime_trivg_cent_Fitting gT (A G : {group gT}) :
    A \subset 'N(G) -> coprime #|G| #|A| -> solvable G ->
  'C_A(G) = 1 -> 'C_A('F(G)) = 1.
Proof.
move=> nGA coGA solG regAG; without loss cycA: A nGA coGA regAG / cyclic A.
  move=> IH; apply/trivgP/subsetP=> a; rewrite -!cycle_subG subsetI.
  case/andP=> saA /setIidPl <-.
  rewrite {}IH ?cycle_cyclic ?(coprimegS saA) ?(subset_trans saA) //.
  by apply/trivgP; rewrite -regAG setSI.
pose X := G <*> A; pose F := 'F(X); pose pi := \pi(A); pose Q := 'O_pi(F).
have pi'G: pi^'.-group G by rewrite /pgroup -coprime_pi' //= coprime_sym.
have piA: pi.-group A by apply: pgroup_pi.
have oX: #|X| = (#|G| * #|A|)%N by rewrite [X]norm_joinEr ?coprime_cardMg.
have hallG: pi^'.-Hall(X) G.
  by rewrite /pHall -divgS joing_subl //= pi'G pnatNK oX mulKn.
have nsGX: G <| X by rewrite /normal joing_subl join_subG normG.
have{oX pi'G piA} hallA: pi.-Hall(X) A.
  by rewrite /pHall -divgS joing_subr //= piA oX mulnK.
have nsQX: Q <| X by rewrite !gFnormal_trans.
have{solG cycA} solX: solvable X.
  rewrite (series_sol nsGX) {}solG /= norm_joinEr // quotientMidl //.
  by rewrite morphim_sol // abelian_sol // cyclic_abelian.
have sQA: Q \subset A.
  by apply: normal_sub_max_pgroup (Hall_max hallA) (pcore_pgroup _ _) nsQX.
have pi'F: 'O_pi(F) = 1.
  suff cQG: G \subset 'C(Q) by apply/trivgP; rewrite -regAG subsetI sQA centsC.
  apply/commG1P/trivgP; rewrite -(coprime_TIg coGA) subsetI commg_subl.
  rewrite (subset_trans sQA) // (subset_trans _ sQA) // commg_subr.
  by rewrite (subset_trans _ (normal_norm nsQX)) ?joing_subl.
have sFG: F \subset G.
  have /dprodP[_ defF _ _]: _ = F := nilpotent_pcoreC pi (Fitting_nil _).
  by rewrite (sub_normal_Hall hallG) ?gFsub //= -defF pi'F mul1g pcore_pgroup.
have <-: F = 'F(G).
  apply/eqP; rewrite eqEsubset -{1}(setIidPr sFG) FittingS ?joing_subl //=.
  by rewrite Fitting_max ?Fitting_nil // gFnormal_trans.
apply/trivgP; rewrite /= -(coprime_TIg coGA) subsetI subsetIl andbT.
apply: subset_trans (subset_trans (cent_sub_Fitting solX) sFG).
by rewrite setSI ?joing_subr.
Qed.
```

Interactive (e.g., ITP such as Coq)

6.14 Theorem Let G be a cyclic group with n elements and generated by a. Let $b \in G$ and let $b = a^s$. Then b generates a cyclic subgroup H of G containing n/d elements, where d is the greatest common divisor of n and s. Also, $\langle a^s \rangle = \langle a^t \rangle$ if and only if $\gcd(s, n) = gcd(t, n)$.

Proof That b generates a cyclic subgroup H of G is known from Theorem 5.17. We need show only that H has n/d elements. Following the argument of Case II of Theorem 6.10, we see that H has as many elements as the smallest positive power m of b that gives the identity. Now $b = a^s$, and $b^m = e$ if and only if $(a^s)^m = e$, or if and only if n divides ms. What is the smallest positive integer m such that n divides ms? Let d be the gcd of n and s. Then there exists integers u and v such that

$$d = un + vs.$$

Since d divides both n and s, we may write

$$1 = u(n/d) + v(s/d)$$

where both n/d and s/d are integers. This last equation shows that n/d and s/d are relatively prime, for any integer dividing both of them must also divide 1. We wish to find the smallest positive m such that

$$\frac{ms}{n} = \frac{m(s/d)}{(n/d)} \text{ is an integer.}$$

From the boxed division property (1), we conclude that n/d must divide m, so the smallest such m is n/d. Thus the order of H is n/d.

Taking for the moment $\mathbb{Z}_n$ as a model for a cyclic group of order n, we see that if d is a divisor of n, then the cyclic subgroup $\langle d \rangle$ of $\mathbb{Z}_n$ had n/d elements, and contains all the positive integers m less than n such that $\gcd(m, n) = d$. Thus there is only one subgroup of $\mathbb{Z}_n$ of order n/d. Taken with the preceding paragraph, this shows at once that if a is a generator of the cyclic group G, then $\langle a^s \rangle = \langle a^t \rangle$ if and only if $\gcd(s, n) = \gcd(t, n)$. ◆

Hand-written (rigorous but not formal)

ML + TP at (close to) human-level but still formal?

Leverage ITP (interactive theorem prover)

Traced Coq (TCoq): records proof tree resulting from "execution trace" of a Coq proof

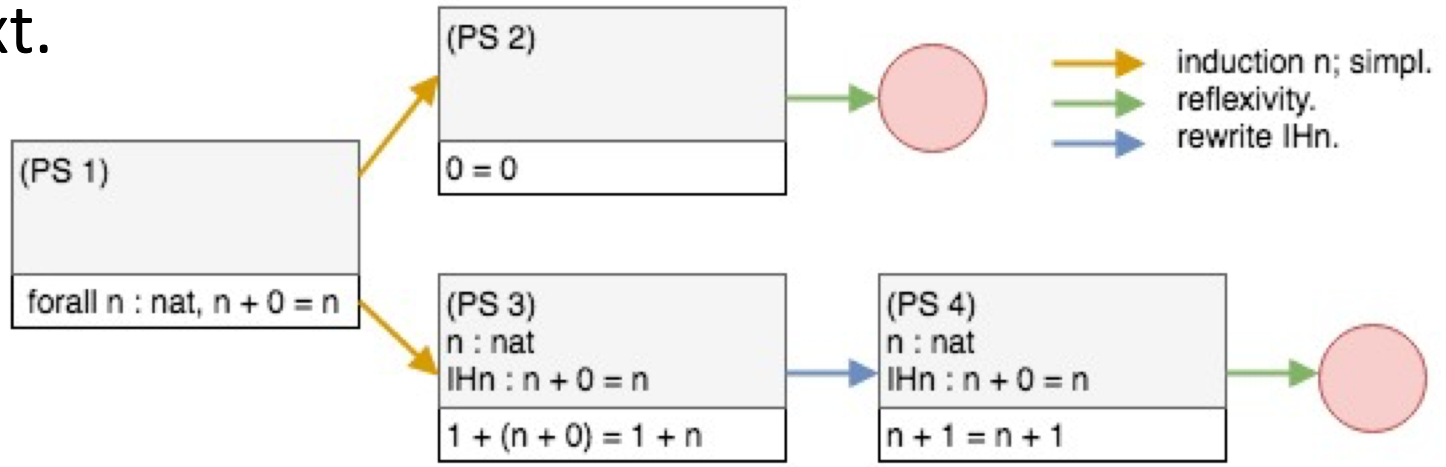
Use data generated by ITP

GamePad: theorem proving as a **Game** and **P**roofs **a**s **d**ata. Provides Python API for TCoq proof trees and lightweight interaction with Coq.

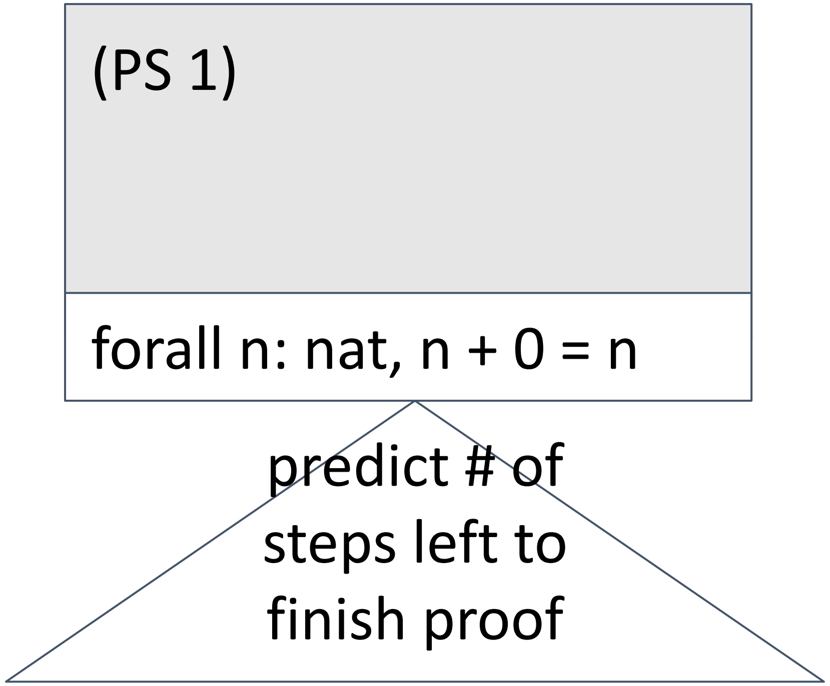
Interactive theorem proving as a game

Objective is to transition all states (double rectangles where top is context and bottom is goal) into terminal states (circles).

1. You can apply a tactic (i.e., take a proof step), which produces other contexts.
2. A state can transition to a terminal state if the goal trivially true given the context.

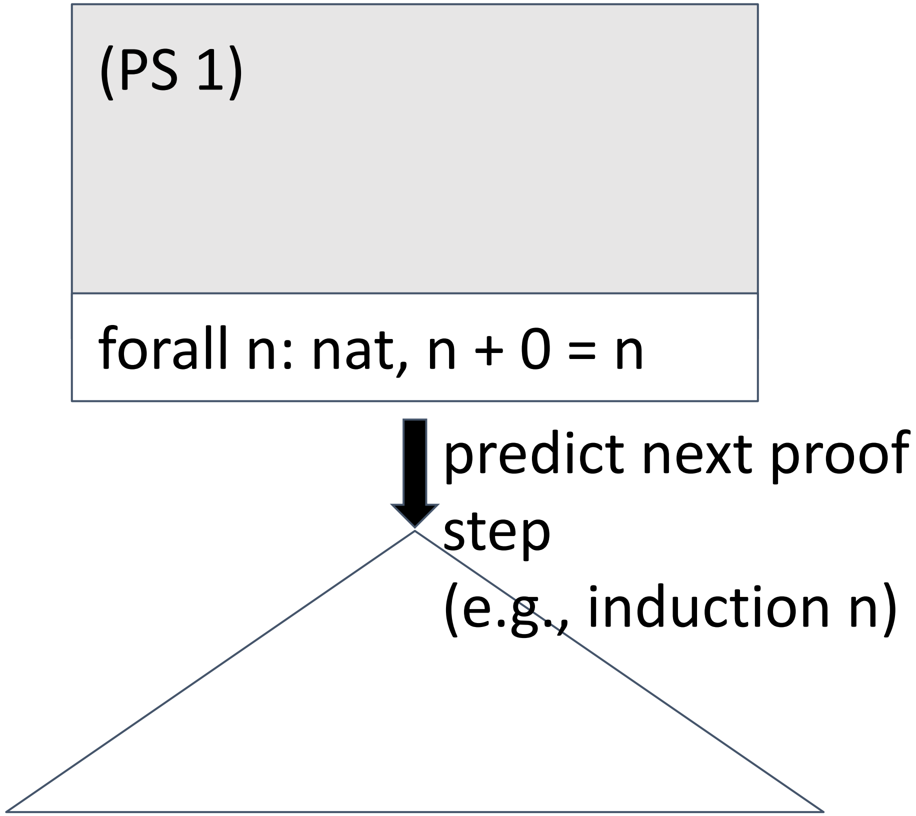


Machine learning tasks



Position Evaluation

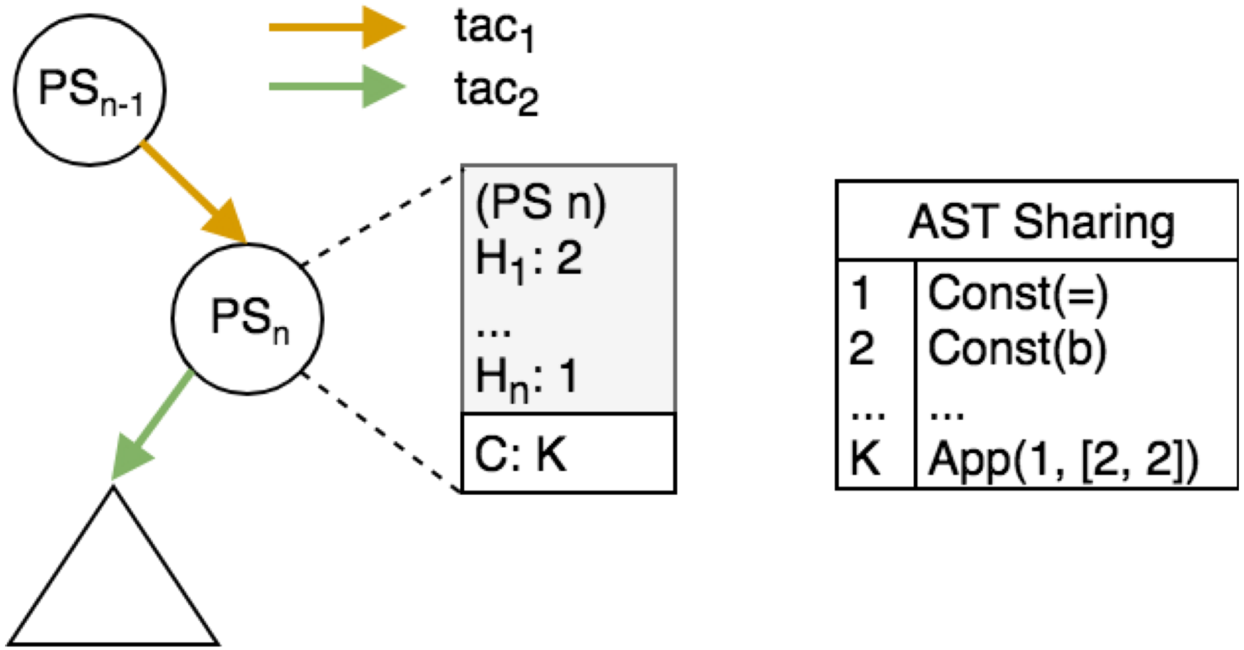
- Think of as value function
- Non-local problem



Tactic Prediction

- Think of as policy
- Local problem
- May require argument synthesis

Representing proof states



Convert proof state PS_n into embedding vector

- Embed context and goal ASTs by embedding each AST.
- Can embed AST using more semantic approach. For example, the "interpreter inspired" embedding of a variable is an environment lookup because the meaning of a variable corresponds to whatever is substituted for it.
- Embedding sharing following AST sharing to save computation (~10x

Use case 1: set up user-defined problem

Step 1:
Set up domain

```
(* The set of the group. *)
Axiom G : Set.

(* The left identity for +. *)
Axiom e : G.

(* The right identity for +. *)
Axiom m : G.

(* + binary operator. *)
Axiom f : G -> G -> G.

(* For readability, we use infix <+> to stand for the binary operator. *)
Infix "<+>" := f (at level 50).

(* [m] is the right-identity for all elements [a] *)
Axiom id_r : forall a, a <+> m = a.

(* [e] is the left-identity for all elements [a] *)
Axiom id_l : forall a, e <+> a = a.
```

Step 2:
Generate proofs (difficulty of domain affects this

```
Lemma rewrite_eq_0: forall b: G, ((e <+> (e <+> m)) <+> ((b <+> ((m <+> m) <+> m)) <+> ((e <+> e) <+> m))) = b.
Proof.
intros.
surgery id_l ((f (f e (f e m)) (f (f b (f (f m m) m)) (f (f e e) m)))) ((f (f e m) (f (f b (f (f m m) m)) (f (f e e) m)))).
surgery id_r ((f (f e m) (f (f b (f (f m m) m)) (f (f e e) m)))) ((f e (f (f b (f (f m m) m)) (f (f e e) m)))).
surgery id_r ((f e (f (f b (f (f m m) m)) (f (f e e) m)))) ((f e (f (f b (f m m)) (f (f e e) m)))).
surgery id_r ((f e (f (f b (f m m)) (f (f e e) m)))) ((f e (f (f b m) (f (f e e) m)))).
surgery id_r ((f e (f (f b m) (f (f e e) m)))) ((f e (f b (f (f e e) m)))).
surgery id_l ((f e (f b (f (f e e) m)))) ((f e (f b (f e m)))).
surgery id_l ((f e (f b (f e m)))) ((f e (f b m))).
surgery id_r ((f e (f b m))) ((f e b)).
surgery id_l ((f e b)) (b).
reflexivity.
Qed.
```

Step 3:
Apply machine learning! We trained a naive tactic predictor (user-defined tactic surgery) and got 14/50 complete proofs.

Use case 2: learn from real-world formalization

Feit-Thompson formalization: one of the largest formal developments (in any system), concerns a deep result in group theory, follows "book proofs" (approx. 255 pages) and good candidate for auto-formalization

Preliminary experiments (accuracies reported)

Model	Pos (Kernel)	Pos (Mid-lvl no implicit)	Tac (Kernel)	Tac (Mid-lvl no implicit)
Constant	53.66	53.66	44.75	44.75
SVM	57.37	57.52	48.94	49.45
GRU	65.30	65.74	58.23	57.50

Also tried predicting identifiers used in tactics, but not synthesizing entire terms

Challenges

1. Leverage human supervision of proofs: TCoq records atomic tactics, compound tactics, and all proof contexts.
2. Using game-like structure of ITP proofs: applied GamePad to user-craft problem and real-world formalization.
3. Difference between syntax and semantics at higher-level of abstraction: implemented interpreter-inspired embeddings for proof contexts.

Future directions

Many interesting directions to explore for theorem proving at close to human-level proofs!

1. Develop more difficult yet tractable user-crafted problems (e.g., infinite sums or integrals)
2. Generative models for tactic arguments (particularly for have tactics)
3. End-to-end training (e.g., reinforcement learning + tree search)

Paper: https://arxiv.org/pdf/1806.00608.pdf

System: https://github.com/ml4tp/gamepad

Lessons & Challenges in Program Synthesis via Learning

- Generalization
- Evaluation
  - Be careful with your test set
- Scalability
  - Combining discrete & differentiable approaches
  - Learning abstractions
- Adapt to new tasks
  - Accumulate knowledge from past experience
- What should be a good benchmark suite for program synthesis?