

A hand holding a Rubik's cube, with the text overlaid on the image.

CS 957, System-2 AI Neuro-Symbolic AI

Mohammad Hossein Rohban

Mar 2025

Sharif University of Technology

NEURAL-SYMBOLIC RECURSIVE MACHINE FOR SYSTEMATIC GENERALIZATION

Qing Li¹, Yixin Zhu³, Yitao Liang^{1,3}, Ying Nian Wu², Song-Chun Zhu^{1,3}, Siyuan Huang¹

¹National Key Laboratory of General Artificial Intelligence, BIGAI

²Department of Statistics, UCLA

³Institute for Artificial Intelligence, Peking University

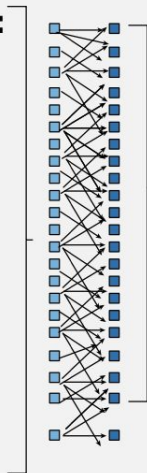
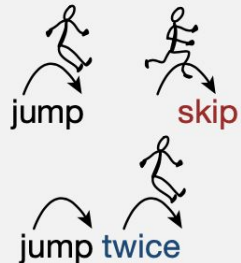
<https://liqing-ustc.github.io/NSR>

Systematic Compositionality

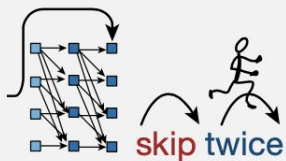
a

Query instruction:
skip twice

Study examples:



Query output:

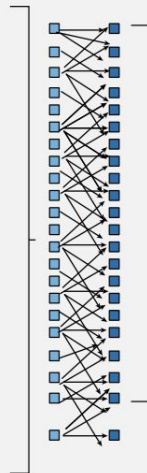
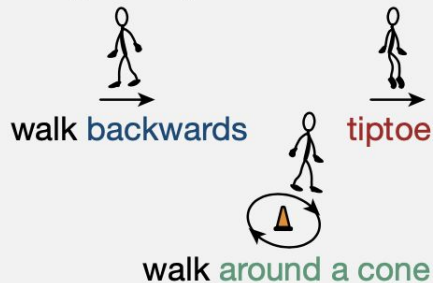


Compare with
behavioural target

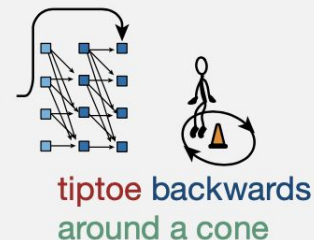
b

Query instruction:
tiptoe backwards
around a cone

Study examples:



Query output:



Compare with
behavioural target

Systematic Compositionality

infinite use of finite means

essential for extrapolating from limited data to novel combinations

e.g. SCAN benchmark

jump	⇒	JUMP
jump left	⇒	LTURN JUMP
jump around right	⇒	RTURN JUMP RTURN JUMP RTURN JUMP RTURN JUMP
turn left twice	⇒	LTURN LTURN
jump thrice	⇒	JUMP JUMP JUMP
jump opposite left and walk thrice	⇒	LTURN LTURN JUMP WALK WALK WALK
jump opposite left after walk around left	⇒	LTURN WALK LTURN WALK LTURN WALK LTURN WALK LTURN LTURN JUMP

Figure 1. Examples of SCAN commands (left) and the corresponding action sequences (right).

Another example

Hint Benchmark

Train		$2 \times 5 \div 9$ 2	$(9 - 9) \times (3 - 4) - 1 \times (0 + 3 - (6 - (2 - 2 \div 2)))$ 0
		$5 \times 5 + (9 - 0 - 2)$ 32	$4 \times (3 + 9) - 7 - (0 - 5)$ 41
	I	$1 \div 4$ 1	$1 \times (2 \div 5) \times (8 - 8 - 6)$ 0
	SS	$1 + 3 \div 4$ 2	$3 \times (7 \times 2) + (8 + 4) + 4 \times 3$ 66
	LS	$3 \times (8 \times (8 \times 1) + 0 \div 9)$ 192	$5 \times (3 + 1 \times 9) + (2 - 5) \times (7 \times (6 + 5))$ 135
	SL	$2 \times (3 \times (3 \div 6 + 6 \times (3 \times 4 \times 6 \div (1 \times 6))) + 0 \div 3)$ 438	$(6 \times 5 - 0) \div ((4 + 9 + 5) \div 9) + (3 - ((2 - (2 + (9 \times 7 - 8 \div 9))) / 4 - 9))$ 18
Test		$6 - 3 \div (9 \times (9 \div (4 - (4 - 7)))) + (1 + 1 / (5 - 2)) \div (7 \times 2 + 6 \div 8)$ 6	$(7 + 3) / (6 - 6 \times (0 \times (6 \div 1))) - (3 \times 1 - 6 - 4 / (4 - 3)) \times (9 \times 3)$ 2
	LL	$(6 + 7) \times 1 + 2 \div 4 + (1 + 4 - 0 \div 3) \times 8 - (1 + 3 \times 8) \times (0 + (2 \times 9 - 0) / 3) \div (8 \div 9)$ 174	$(3 + (8 + (4 - 7) \times (7 + 8))) \times (8 \div 4 - (4 - (6 + 5) + 6)) \div (7 + 5 \times 1 \times 0) \div 5$ 1
		$7 \times (8 \div (1 \times (7 \div 7)) + (1 + 2)) \times 10 + 9 - 5 \div (8 + 4 \div (9 \times 6)) + (8 - (9 - 8 + 3))$ 620	

Another example

PCFG Benchmark

repeat A B C	→	A B C A B C
echo remove_first D K , E F	→	E F F
append swap F G H , repeat I J	→	H G F I J I J

Representation: Grounded Symbol System (GSS)

Neural-Symbolic Recursive Machine (NSR): a principled framework designed for the joint learning of **perception**, **syntax**, and **semantics**.

GSS is at the heart of this:

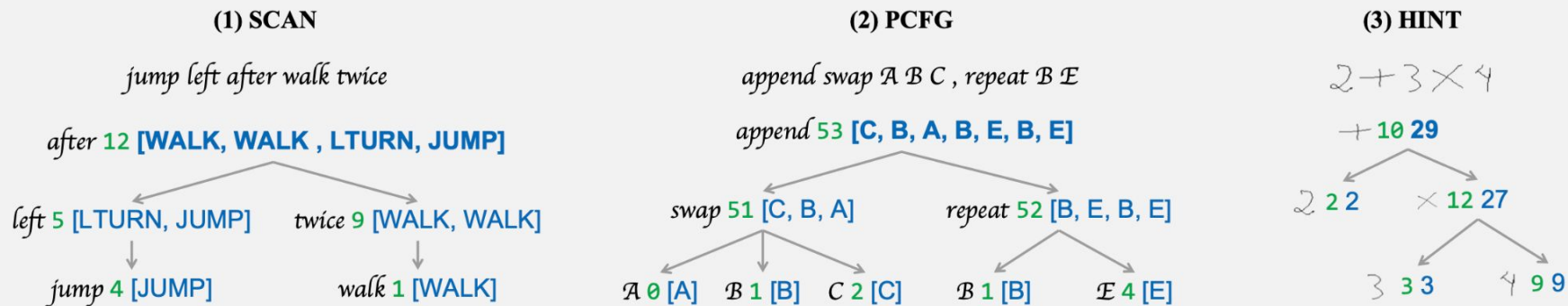


Figure 1: **Illustrative examples of GSSs demonstrating human-like reasoning processes.** (a) SCAN: each node encapsulates a triplet (*word*, **symbol**, **action sequence**). (b) PCFG: nodes consist of triplets (*word*, **symbol**, **letter list**). (c) HINT: nodes contain triplets (*image*, **symbol**, **value**). Symbols are denoted by their indices.

GSS Definition

Formally, a GSS is a directed tree $T = \langle (x, s, v), e \rangle$ with each node being a triplet consisting of grounded input x , an abstract symbol s , and its semantic meaning v .

The edges e represent semantic dependencies, with an edge $i \rightarrow j$ indicating node i 's meaning depends on node j .

How to obtain GSS for a given input?

Hand-designing is not an option.

Should be scalable!

How can we do this?

NSR Overview

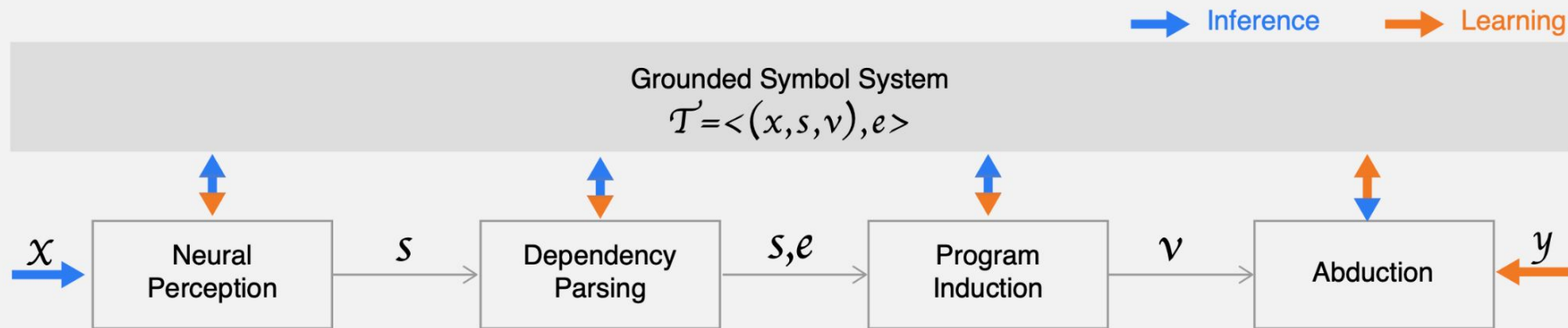


Figure 2: **Illustration of the inference and learning pipeline in NSR.**

Neural Perception

Helps with **perceptual variability** of raw input signals

$$p(s|x; \theta_p) = \prod_i p(w_i|x_i; \theta_p) = \prod_i \text{softmax}(\phi(w_i, x_i; \theta_p)),$$

$\phi(w_i, x_i; \theta_p)$ is the **scoring function** parameterized by a neural network.

w_i are the deduced symbols

Dependency Parsing

transition-based neural dependency parser

the parser identifies possible **transitions** to transform the input sequence into a **dependency tree**

$$p(e|s; \theta_s) = p(\mathcal{T}|s; \theta_s) = \prod_{t_i \in \mathcal{T}} p(t_i|c_i; \theta_s), \quad (2)$$

where θ_s are the parameters of the parser, $\mathcal{T} = \{t_1, t_2, \dots, t_l\} \models e$ is the sequence of transitions that generates the dependency tree e , and c_i is the state representation at step i .

Dependency Parsing (cont.)

ID	Stack Buffer	Transition	Dependency
0	3 + 4 × 2	Shift	
1	3 + 4 × 2	Shift	
2	3 + 4 × 2	Left-Arc	3 ← +
3	+ 4 × 2	Shift	
4	+ 4 × 2	Shift	
5	+ 4 × 2	Left-Arc	4 ← ×
6	+ × 2	Shift	
7	+ × 2	Right-Arc	× → 2
8	+ ×	Right-Arc	+ → ×

Start: $\sigma = [\text{ROOT}]$, $\beta = w_1, \dots, w_n$, $A = \emptyset$

1. Shift $\sigma, w_i | \beta, A \rightarrow \sigma | w_i, \beta, A$
2. Left-Arc_r $\sigma | w_i | w_j, \beta, A \rightarrow \sigma | w_j, \beta, A \cup \{r(w_j, w_i)\}$
3. Right-Arc_r $\sigma | w_i | w_j, \beta, A \rightarrow \sigma | w_i, \beta, A \cup \{r(w_i, w_j)\}$

Finish: $\sigma = [w]$, $\beta = \emptyset$

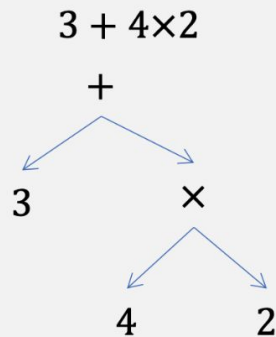


Figure A1: **Applying the transition-based dependency parser to an example of HINT.** It is similar for SCAN and PCFG.

Program Induction

Utilize **functional programs** to express the **semantics** of symbols

Formally, given input symbols s and their dependencies e , the relationship is defined as:

$$p(v|e, s; \theta_l) = \prod_i p(v_i|s_i, \text{children}(s_i); \theta_l), \quad (3)$$

where θ_l denotes the set of induced programs for each symbol.

Identifying a program that **aligns with given examples**, employing **pre-defined** primitives.

Selected a universal set of primitives, such as **O**, **inc**, **dec**, **==**, and **if**, proven to be **sufficient** for any **symbolic function** representation


Program Induction (cont.)

Given a set of **labeled examples**, find a **set of functions for each symbol** such that the **root** is evaluated equal to the **given label**.

Include a minimal subset of **Lisp primitives** and the **recursion primitive**

DreamCoder (Ellis et al., 2021) is adapted for program induction.

Model Inference

Begins with the **neural perception** module translating input x , e.g.  into a sequence of symbols, $2 + 3 \times 9$.

Dependency parsing module then structures this sequence into a **dependency tree**, for instance, $+ \rightarrow 2 \times$ and $\times \rightarrow 3 9$.

Program induction module employs **programs associated with each symbol** to compute node values in a **bottom-up** fashion, yielding calculations like $3 \times 9 \Rightarrow 27$, followed by $2 + 27 \Rightarrow 29$.

Learning

Backpropagation?

REINFORCE?

$$p(y|x; \Theta) = \sum_T p(T, y|x; \Theta) = \sum_{s, e, v} p(s|x; \theta_p) p(e|s; \theta_s) p(v|s, e; \theta_l) p(y|v),$$

Gradient Updates

$$\nabla_{\theta_p} L(x, y) = \mathbb{E}_{T \sim p(T|x, y)} [\nabla_{\theta_p} \log p(s|x; \theta_p)],$$

$$\nabla_{\theta_s} L(x, y) = \mathbb{E}_{T \sim p(T|x, y)} [\nabla_{\theta_s} \log p(e|s; \theta_s)],$$

$$\nabla_{\theta_l} L(x, y) = \mathbb{E}_{T \sim p(T|x, y)} [\nabla_{\theta_l} \log p(v|s, e; \theta_l)],$$

$$p(T|x, y) = \frac{p(T, y|x; \Theta)}{\sum_{T'} p(T', y|x; \Theta)} = \begin{cases} 0, & \text{if } T \notin Q \\ \frac{p(T|x; \Theta)}{\sum_{T' \in Q} p(T'|x; \Theta)}, & \text{if } T \in Q \end{cases}$$

with Q as the set of T congruent with y .

Gradient Updates (cont.)

Due to the computational challenge of taking expectations with respect to this posterior distribution, **Monte Carlo** sampling is utilized for approximation.

Deduction-Abduction Alg. :

- For an instance (x, y) , we initially perform a **greedy deduction** from x to obtain an initial GSS T .
- To align T^* with correct label y , apply a **top-down abduction search**
- Adjust **perception**, **syntax**, and **semantics**.
- Theoretically, this method acts as a Metropolis-Hastings sampler for $p(T \mid x, y)$

Deduction vs Abduction?

Deductive reasoning is the process of drawing **valid inferences**.

Modus ponens (also known as "affirming the antecedent" or "the law of detachment") is the primary deductive **rule of inference**. It applies to arguments that have as first premise a **conditional statement** ($P \rightarrow Q$) and as second premise the antecedent (P) of the conditional statement. It obtains the consequent (Q) of the conditional statement as its conclusion. The argument form is listed below:

1. $P \rightarrow Q$ (First premise is a conditional statement)
2. P (Second premise is the antecedent)
3. Q (Conclusion deduced is the consequent)

Deduction vs Abduction? (cont.)

Deduction is **backward**!

In **logic**, **explanation** is accomplished through the use of a **logical theory** T representing a **domain** and a set of observations O . Abduction is the process of deriving a set of explanations of O according to T and picking out one of those explanations. For E to be an explanation of O according to T , it should satisfy two conditions:

- O follows from E and T ;
- E is **consistent** with T .

In formal logic, O and E are assumed to be sets of **literals**. The two conditions for E being an explanation of O according to theory T are formalized as:

$$T \cup E \models O;$$

$T \cup E$ is consistent.

Metropolis-Hastings Sampling

Metropolis algorithm (symmetric proposal distribution)

Let $f(x)$ be a function that is proportional to the desired probability density function $P(x)$ (a.k.a. a target distribution)^[a].

1. Initialization: Choose an arbitrary point x_t to be the first observation in the sample and choose a proposal function $g(x | y)$. In this section, g is assumed to be symmetric; in other words, it must satisfy $g(x | y) = g(y | x)$.
2. For each iteration t :
 - *Propose* a candidate x' for the next sample by picking from the distribution $g(x' | x_t)$.
 - *Calculate* the *acceptance ratio* $\alpha = f(x')/f(x_t)$, which will be used to decide whether to accept or reject the candidate^[b]. Because f is proportional to the density of P , we have that $\alpha = f(x')/f(x_t) = P(x')/P(x_t)$.
 - *Accept or reject*:
 - Generate a uniform random number $u \in [0, 1]$.
 - If $u \leq \alpha$, then *accept* the candidate by setting $x_{t+1} = x'$,
 - If $u > \alpha$, then *reject* the candidate and set $x_{t+1} = x_t$ instead.

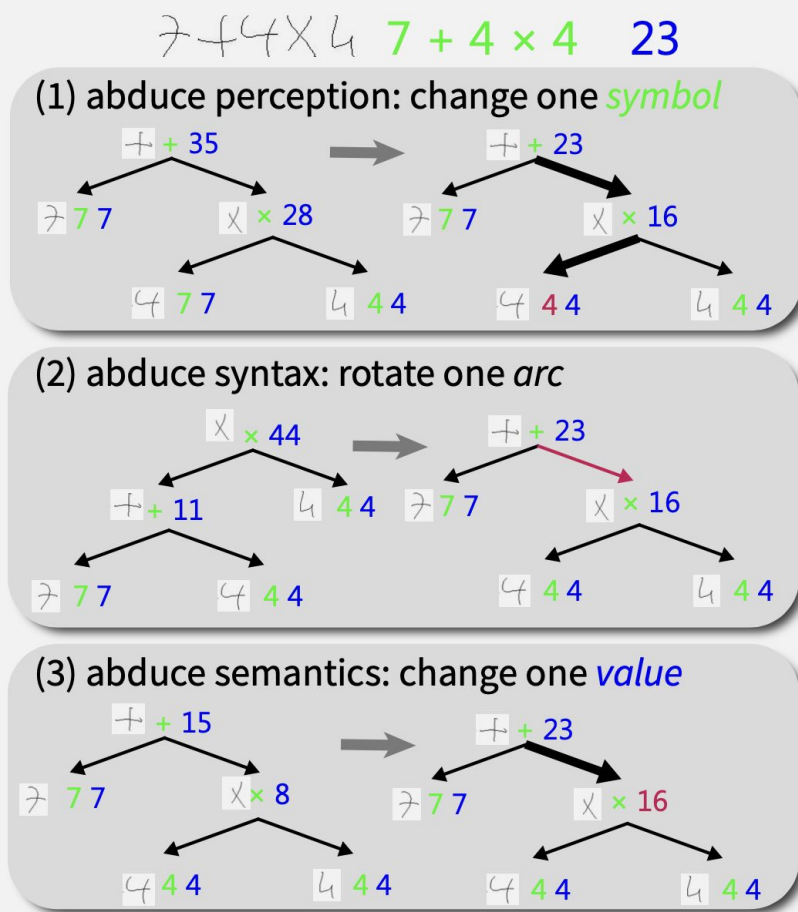


Figure 3: **Illustration of abduction in HINT over perception, syntax, and semantics.** Elements modified during abduction are emphasized in **red**.

Algorithm A1: Learning by Deduction-Abduction

Input : Training set $D = (x_i, y_i) : i = 1, 2, \dots, N$

Output : $\theta_p^{(T)}, \theta_s^{(T)}, \theta_l^{(T)}$

```
1 Initial Module: perception  $\theta_p^{(0)}$ , syntax  $\theta_s^{(0)}$ , semantics  $\theta_l^{(0)}$ 
2 for  $t \leftarrow 0$  to  $T$  do
3   Buffer  $\mathcal{B} \leftarrow \emptyset$ 
4   foreach  $(x, y) \in D$  do
5      $T \leftarrow \text{DEDUCE}(x, \theta_p^{(t)}, \theta_s^{(t)}, \theta_l^{(t)})$ 
6      $T^* \leftarrow \text{ABDUCE}(T, y)$ 
7      $\mathcal{B} \leftarrow \mathcal{B} \cup T^*$ 
8    $\theta_p^{(t+1)}, \theta_s^{(t+1)}, \theta_l^{(t+1)} \leftarrow \text{learn}(\mathcal{B}, \theta_p^{(t)}, \theta_s^{(t)}, \theta_l^{(t)})$ 
9 return  $\theta_p^{(T)}, \theta_s^{(T)}, \theta_l^{(T)}$ 
10
11 Function  $\text{DEDUCE}(x, \theta_p, \theta_s, \theta_l)$  :
12   Sample  $\hat{s} \sim p(s|x; \theta_p)$ ,  $\hat{e} \sim p(e|\hat{s}; \theta_s)$ ,  $\hat{v} = f(\hat{s}, \hat{e}; \theta_l)$ 
13   return  $T = \langle (x, \hat{s}, \hat{v}), \hat{e} \rangle$ 
```



```

15 Function ABDUCE ( $T, y$ ) :
16    $Q \leftarrow \text{PriorityQueue}()$ 
17    $Q.\text{push}(\text{root}(T), y, 1.0)$ 
18   while  $Q$  is not empty do
19      $A, y_A, p \leftarrow Q.\text{pop}()$ 
20      $A \leftarrow (x, w, v, \text{arcs})$ 
21     if  $A.v == y_A$  then
22        $\lfloor$  return  $T(A)$ 
23       // Abduce perception
24       foreach  $w' \in \Sigma$  do
25          $A' \leftarrow A(w \rightarrow w')$ 
26         if  $A'.v == y_A$  then
27            $\lfloor Q.\text{push}(A', y_A, p(A'))$ 
28       // Abduce syntax
29       foreach  $\text{arc} \in \text{arcs}$  do
30          $A' \leftarrow \text{rotate}(A, \text{arc})$ 
31         if  $A'.v == y_A$  then
32            $\lfloor Q.\text{push}(A', y_A, p(A'))$ 
33       // Abduce semantics
34        $A' \leftarrow A(v \rightarrow y_A)$ 
35        $Q.\text{push}(A', y_A, p(A'))$ 
36       // Top-down search
37       foreach  $B \in \text{children}(A)$  do
38          $y_B \leftarrow \text{Solve}(B, A, y_A | \theta_l(A.w))$ 
39          $Q.\text{push}(B, y_B, p(B))$ 

```

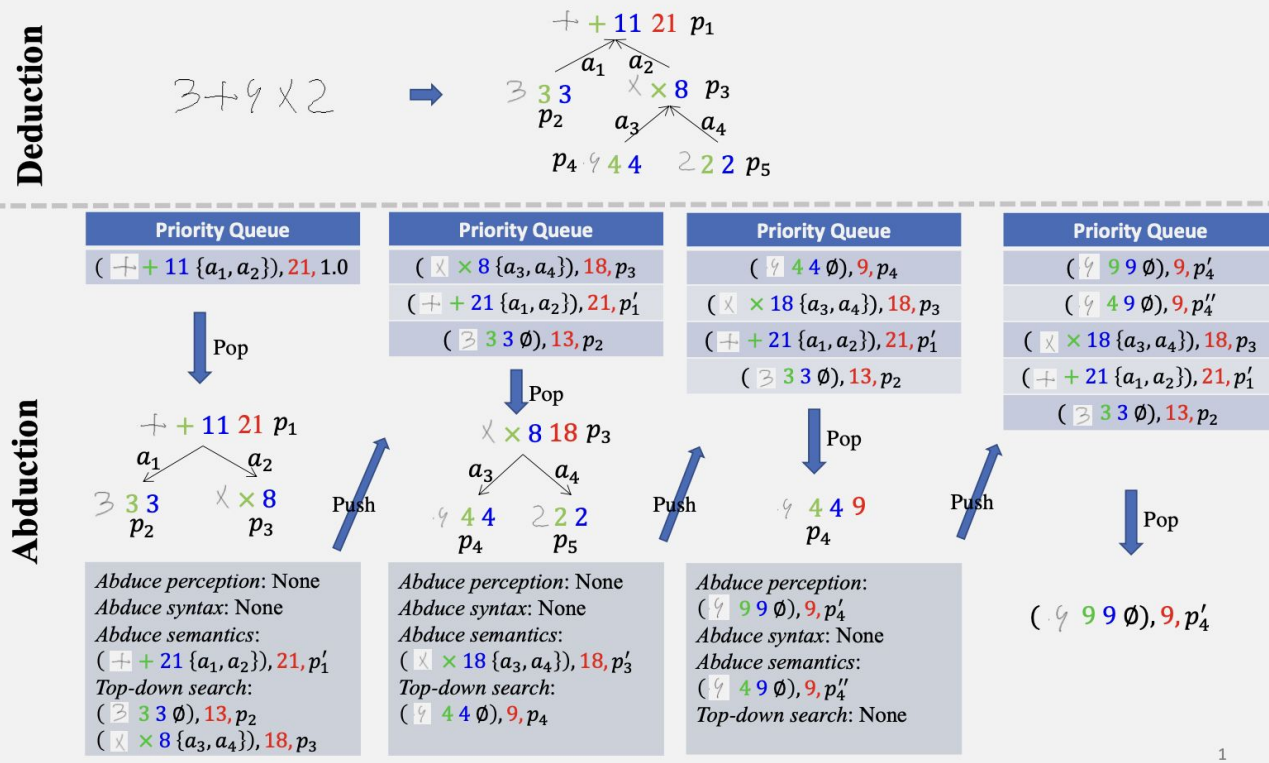


Figure A2: **An illustration of the deduction-abduction process for an example of HINT.** Given a handwritten expression, the system performs a greedy deduction to propose an initial solution, generating a wrong result. In abduction, the root node, paired with the ground-truth result, is first pushed to the priority queue. The abduction over perception, syntax, and semantics is performed on the popped node to generate possible revisions. A top-down search is also applied to propagate the expected value to its children. All possible revisions are then pushed into the priority queue. This process is repeated until we find the most likely revision for the initial solution.

Generalization

For effective **generalization**, it is essential to incorporate certain **inductive biases** that are **minimal** yet **universal** in the program induction.

Advocate for two critical inductive biases: **equivariance** and **compositionality**.

Equivariance enhances the model's systematicity, enabling it to generalize concepts such as “jump twice” from examples like {“run”, “run twice”, “jump”}

Compositionality increases the model's ability to extend these concepts to longer sequences, e.g., “run and jump twice”.

Generalization (cont.)

Definition 3.1 (*Equivariance*). For sets \mathcal{X} and \mathcal{Y} , and a *permutation* group \mathcal{P} with operations $T_p : \mathcal{X} \rightarrow \mathcal{X}$ and $T'_p : \mathcal{Y} \rightarrow \mathcal{Y}$, a mapping $\Phi : \mathcal{X} \rightarrow \mathcal{Y}$ is *equivariant* iff $\forall x \in \mathcal{X}, p \in \mathcal{P} : \Phi(T_p x) = T'_p \Phi(x)$.

Definition 3.2 (*Compositionality*). For sets \mathcal{X} and \mathcal{Y} , with composition operations $T_c : (\mathcal{X}, \mathcal{X}) \rightarrow \mathcal{X}$ and $T'_c : (\mathcal{Y}, \mathcal{Y}) \rightarrow \mathcal{Y}$, a mapping $\Phi : \mathcal{X} \rightarrow \mathcal{Y}$ is *compositional* iff $\exists T_c, T'_c, \forall x_1 \in \mathcal{X}, x_2 \in \mathcal{X} : \Phi(T_c(x_1, x_2)) = T'_c(\Phi(x_1), \Phi(x_2))$.

The three modules of NSR —neural perception, dependency parsing, and program induction— exhibit equivariance and compositionality, functioning as **pointwise** transformations based on their formulations.

Experimental Results on SCAN and PCFG

SCAN with four splits:

- (i) **SIMPLE**, where the dataset is randomly divided into training and test sets
- (ii) **LENGTH**, with training on output sequences **up to 22** actions and testing on sequences from **24 to 48 actions**;
- (iii) **JUMP**, training **excluding the “jump” command mixed with other primitives**, tested on combinations including “jump”;
- (iv) **AROUND RIGHT**, **excluding “around right”** from training but testing on combinations derived from the separately trained “around” and “right.”

Results (cont.)

PCFG with three splits:

- (i) **i.i.d**: where samples are **randomly** allocated for training and testing,
- (ii) **systematicity**: this split is designed to specifically assess the model's capability to interpret **combinations of functions unseen** during training, and
- (iii) **productivity**: this split tests the model's generalization to longer sequences, with training samples containing **up to 8 functions** and test samples having **at least 9 functions**.

Table 1: **Test accuracy across various splits of SCAN and PCFG.** The results of NeSS on PCFG are reported by modifying the source code from [Chen et al. \(2020\)](#) for PCFG.

models	SCAN				PCFG		
	SIMPLE	JUMP	AROUND RIGHT	LENGTH	i.i.d.	systematicity	productivity
Seq2Seq (Lake and Baroni, 2018)	99.7	1.2	2.5	13.8	79	53	30
CNN (Dessì and Baroni, 2019)	100.0	69.2	56.7	0.0	85	56	31
Transformer (Csordás et al., 2021)	-	-	-	20.0	-	96	85
Transformer (Ontanón et al., 2022)	-	0.0	-	19.6	-	83	63
equivariant Seq2seq (Gordon et al., 2019)	100.0	99.1	92.0	15.9	-	-	-
NeSS (Chen et al., 2020)	100.0	100.0	100.0	100.0	≈ 0	≈ 0	≈ 0
NSR (ours)	100.0	100.0	100.0	100.0	100	100	100

Induced Programs

$turn : \emptyset \rightarrow []$	$1 \rightarrow \text{WALK}$
$walk : \emptyset \rightarrow [\text{inc } 0]$	$2 \rightarrow \text{LOOK}$
$look : \emptyset \rightarrow [\text{inc inc } 0]$	$3 \rightarrow \text{RUN}$
$run : \emptyset \rightarrow [\text{inc inc inc } 0]$	$4 \rightarrow \text{JUMP}$
$jump : \emptyset \rightarrow [\text{inc inc inc inc } 0]$	$5 \rightarrow \text{LTRUN}$
$left : x \rightarrow \text{cons}(\text{inc inc inc inc inc } 0, x)$	$6 \rightarrow \text{RTURN}$
$right : x \rightarrow \text{cons}(\text{inc inc inc inc inc inc } 0, x)$	
$opposite : x \rightarrow \text{cons}(\text{car}(x), x)$	
$around : x \rightarrow +(+(+(x, x), x), x)$	
$twice : x \rightarrow +(x, x)$	
$thrice : x \rightarrow +(+(x, x), x)$	
$and : x, y \rightarrow +(x, y)$	
$after : x, y \rightarrow +(y, x)$	

(b) Programs induced using NSR.

or vice versa, retains the dependencies as predicted by the dependency parser. (b) **Induced programs for input words using NSR.** Here, x and y represent the inputs, \emptyset signifies empty inputs, `cons` appends an item to the beginning of a list, `car` retrieves the first item of a list, and `+` amalgamates two lists.

Results on HINT

- (i) “I”: expressions seen in training but composed of **unseen handwritten images**.
- (ii) “SS”: **unseen expressions**, but their **lengths** and **values** are **within** the training range.
- (iii) “LS”: expressions are **longer than those in training**, but their **values are within** the same range.
- (iv) “SL”: expressions have **larger values**, and their **lengths are the same** as training.
- (v) “LL”: expressions are **longer**, and their **values are bigger** than those in training.

Results on HINT

Table 2: **Test accuracy on HINT.** Results for GRU, LSTM, and Transformer are directly cited from [Li et al. \(2023b\)](#). NeSS results are obtained by adapting its source code to HINT.

Model	Symbol Input						Image Input					
	I	SS	LS	SL	LL	Avg.	I	SS	LS	SL	LL	Avg.
GRU	76.2	69.5	42.8	10.5	15.1	42.5	66.7	58.7	33.1	9.4	12.8	35.9
LSTM	92.9	90.9	74.9	12.1	24.3	58.9	83.9	79.7	62.0	11.2	21.0	51.5
Transformer	98.0	96.8	78.2	11.7	22.4	61.5	88.4	86.0	62.5	10.9	19.0	53.1
NeSS	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	-	-	-	-	-	-
NSR (ours)	98.0	97.3	83.7	95.9	77.6	90.1	88.5	86.2	67.1	83.2	58.2	76.0

Compositional Machine Translation

The training set comprises **10,000 English-French sentence** pairs, with English sentences primarily initiating with phrases like “I am,” “you are,” and their respective contractions.

Uniquely, the training set includes 1,000 repetitions of the sentence pair (“I am daxy,” “je suis daxiste”), introducing the **pseudoword “daxy.”**

The test set, however, explores **8 different combinations of “daxy” with other phrases**, such as “you are not daxy,” which are absent from the training set.

Results (cont.)

Table 3: Accuracy on compositional machine translation.

Model	Accuracy
Seq2Seq	12.5
Transformer	14.4
NeSS	100.0
NSR (ours)	100.0