A MODEL DETAILS

Dependency Parsing Fig. A1 illustrate the process of parsing an arithmetic expression via the dependency parser. Formally, a *state* $c = (\alpha, \beta, A)$ in the dependency parser consists of a *stack* α, a *buffer* β, and a set of *dependency arcs* A. The initial state for a sequence $s = w_0 w_1 ... w_n$ is $\alpha = [\texttt{Root}], \beta = [w_0 w_1 ... w_n], A = \varnothing$. A state is regarded as terminal if the buffer is empty and the stack only contains the node `Root`. The parse tree can be derived from the dependency arcs A. Let α_i denote the i-th top element on the stack, and β_i the i-th element on the buffer. The parser defines three types of transitions between states:

- LEFT-ARC: add an arc $\alpha_1 \rightarrow \alpha_2$ to A and remove α_2 from the stack α. Precondition: $|\alpha| \geqslant 2$.
- RIGHT-ARC: add an arc $\alpha_2 \rightarrow \alpha_1$ to A and remove α_1 from the stack α. Precondition: $|\alpha| \geqslant 2$.
- SHIFT: move β_1 from the buffer β to the stack α. Precondition: $|\beta| \geqslant 1$.

The goal of the parser is to predict a transition sequence from an initial state to a terminal state. The parser predicts one transition from $\mathcal{T} = \{\text{LEFT-ARC}, \text{RIGHT-ARC}, \text{SHIFT}\}$ at a time, based on the current state $c = (\alpha, \beta, A)$. The state representation is constructed from a local window and contains following three elements: (i) The top three words on the stack and buffer: $\alpha_i, \beta_i, i = 1, 2, 3$; (ii) The first and second leftmost/rightmost children of the top two words on the stack: $lc_1(\alpha_i), rc_1(\alpha_i), lc_2(\alpha_i), rc_2(\alpha_i), i = 1, 2$; (iii) The leftmost of leftmost/rightmost of rightmost children of the top two words on the stack: $lc_1(lc_1(\alpha_i)), rc_1(rc_1(\alpha_i)), i = 1, 2$. We use a special `Null` token for non-existent elements. Each element in the state representation is embedded to a d-dimensional vector $e \in R^d$, and the full embedding matrix is denoted as $E \in R^{|\Sigma| \times d}$, where Σ is the concept space. The embedding vectors for all elements in the state are concatenated as its representation: $c = [e_1\ e_2 ... e_n] \in R^{nd}$. Given the state representation, we adopt a two-layer feed-forward neural network to predict the transition.

Program Induction Program induction, *i.e.*, synthesizing programs from input-output examples, was one of the oldest theoretical frameworks for concept learning within artificial intelligence (Solomonoff, 1964). Recent advances in program induction focus on training neural networks to guide the program search (Kulkarni et al., 2015; Lake et al., 2015; Balog et al., 2017; Devlin et al., 2017; Ellis et al., 2018a;b). For example, Balog et al. (2017) train a neural network to predict properties of the program that generated the outputs from the given inputs and then use the neural network's predictions to augment search techniques from the programming languages community. Ellis et al. (2021) released a neural-guided program induction system, *DreamCoder*, which can efficiently discover interpretable, reusable, and generalizable programs across a wide range of domains, including both classic inductive programming tasks and creative tasks such as drawing pictures and building scenes. DreamCoder adopts a "wake-sleep" Bayesian learning algorithm to extend program space with new symbolic abstractions and train the neural network on imagined and replayed problems.

To learn the semantics of a symbol c from a set of examples D_c is to find a program ρ_c composed from a set of primitives L, which maximizes the following objective:

$$\max_{\rho} p(\rho | D_c, L) \propto p(D_c | \rho)\ p(\rho | L), \tag{A1}$$

where $p(D_c|\rho)$ is the likelihood of the program ρ matching D_c, and $p(\rho|L)$ is the prior of ρ under the program space defined by the primitives L. Since finding a globally optimal program is usually intractable, the maximization in Eq. (A1) is approximated by a stochastic search process guided by a neural network, which is trained to approximate the posterior distribution $p(\rho|D_c, L)$. We refer the readers to DreamCoder (Ellis et al., 2021)[1] for more technical details.

[1] https://github.com/ellisk42/ec

B LEARNING

Derivation of Eq. (5) Take the derivative of L w.r.t. θ_p,

$$\begin{aligned}\nabla_{\theta_p} L(x,y) &= \nabla_{\theta_p} \log p(y|x) = \frac{1}{p(y|x)} \nabla_{\theta_p} p(y|x) \\ &= \sum_T \frac{p(T,y|x;\Theta)}{\sum_{T'} p(T',y|x;\Theta)} \nabla_{\theta_p} \log p(s|x;\theta_p) \\ &= \mathbb{E}_{T\sim p(T|x,y)}[\nabla_{\theta_p} \log p(s|x;\theta_p)].\end{aligned} \tag{A2}$$

Similarly, for θ_s, θ_l, we have

$$\begin{aligned}\nabla_{\theta_s} L(x,y) &= \mathbb{E}_{T\sim p(T|x,y)}[\nabla_{\theta_s} \log p(e|s;\theta_s)], \\ \nabla_{\theta_l} L(x,y) &= \mathbb{E}_{T\sim p(T|x,y)}[\nabla_{\theta_l} \log p(v|s,e;\theta_l)],\end{aligned} \tag{A3}$$

Deduction-Abduction Alg. A1 describes the procedure for learning NSR by the proposed deduction-abduction algorithm. Fig. 3 illustrates the one-step abduction over perception, syntax, and semantics in HINT and Fig. A2 visualizes a concrete example to illustrate the deduction-abduction process. It is similar for SCAN and PCFG.

C EXPRESSIVENESS AND GENERALIZATION OF NSR

Expressiveness

Lemma C.1. Given a finite unique set of $\{x^i : i = 0, ..., N\}$, there exists a sufficiently capable neural network f_p such that: $\forall x^i, f_p(x^i) = i$.

This lemma asserts the existence of a neural network capable of mapping every element in a finite set to a unique index, *i.e.*, $x^i \to i$, as supported by (Hornik et al., 1989; Lu et al., 2017). The parsing process in this scenario is straightforward, given that every input is mapped to a singular token.

Lemma C.2. Any index space can be constructed from the primitives $\{0, \texttt{inc}\}$.

This lemma is grounded in the fact that all indices are natural numbers, which can be recursively defined by $\{0, \texttt{inc}\}$, allowing the creation of indices for both inputs and outputs.

Generalization Equivariance and compositionality are formalized utilizing group theory, following the approaches of Gordon et al. (2019) and Zhang et al. (2022). A discrete group $\mathcal{G}$ comprises elements $\{g_1, ..., g_{|\mathcal{G}|}\}$ and a binary group operation "$\cdot$", adhering to group axioms (closure, associativity, identity, and invertibility). Equivariance is associated with a *permutation* group $\mathcal{P}$, representing permutations of a set $\mathcal{X}$. For compositionality, a *composition* operation $\mathcal{C}$ is considered, defining $T_c : (\mathcal{X}, \mathcal{X}) \to \mathcal{X}$.

The three modules of NSR—neural perception (Eq. (1)), dependency parsing (Eq. (2)), and program induction (Eq. (3))—exhibit equivariance and compositionality, functioning as pointwise transformations based on their formulations. Eqs. (1) to (3) demonstrate that in all three modules of the NSR system, the joint distribution is factorized into a product of several independent terms. This factorization process makes the modules naturally adhere to the principles of equivariance and recursiveness, as outlined in Definitions 3.1 and 3.2.

D EXPERIMENTS

D.1 EXPERIMENTAL SETUP

For tasks taking symbols as input (*i.e.*, SCAN and PCFG), the perception module is not required in NSR; For the task taking images as input, we adopt ResNet-18 as the perception module, which is pre-trained unsupervisedly (Van Gansbeke et al., 2020) on handwritten images from the training set. In the dependency parser, the token embeddings have a dimension of 50, the hidden dimension of the transition classifier is 200, and we use a dropout of 0.5. For the program induction, we adopt the

default setting in DreamCoder (Ellis et al., 2021). For learning NSR, both the ResNet-18 and the dependency parser are trained by the Adam optimizer (Kingma and Ba, 2015) with a learning rate of 10^{-4}. NSR are trained for 100 epochs for all datasets.

Compute All training can be done using a single NVIDIA GeForce RTX 3090Ti under 24 hours.

D.2 Ablation Study

To explore how well the individual modules of NSR are learned, we perform an ablation study on HINT to analyze the performance of each module of NSR. Specifically, along with the final results, the HINT dataset also provides the symbolic sequences and parse trees for evaluation. For Neural Perception, we report the accuracy of classifying each symbol. For Dependency parsing, we report the accuracy of attaching each symbol to its correct parent, given the ground-truth symbol sequence as the input. For Program Induction, we report the accuracy of the final results, given the ground-truth symbol sequence and parse tree.

Overall, each module achieves high accuracy, as shown in Tab. A1. For Neural Perception, most errors come from the two parentheses, "(" and ")", because they are visually similar. For Dependency Parsing, we analyze the parsing accuracies for different concept groups: digits (100%), operators (95.85%), and parentheses (64.28%). The parsing accuracy of parentheses is much lower than those of digits and operators. We think this is because, as long as digits and operators are correctly parsed in the parsing tree, where to attach the parentheses does not influence the final results because parentheses have no semantic meaning. For Program Induction, we can manually verify that the induced programs (Fig. 4) have correct semantics. The errors are caused by exceeding the recursion limit when calling the program for multiplication. The above analysis is also verified by the qualitative examples in Fig. A3.

D.3 Qualitative Examples

Figs. A3 and A4 show several examples of the NSR predictions on SCAN and HINT.

Fig. A5 illustrates the evolution of semantics along the training of NSR in HINT. This pattern is highly in accordance with how children learn arithmetic in developmental psychology (Carpenter et al., 1999): The model first masters the semantics of digits as *counting*, then learns $+$ and $-$ as recursive counting, and finally figures out how to define $\times$ and $\div$ based on $+$ and $-$. Crucially, $\times$ and $\div$ are impossible to be correctly learned before mastering $+$ and $-$. The model is endowed with such an incremental learning capability since the program induction module allows the semantics of concepts to be built compositionally from those learned earlier (Ellis et al., 2021).

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: σ = [ROOT], β = w_1, ..., w_n, A = ∅
1. Shift σ, w_i|β, A ➔ σ|w_i, β, A
2. Left-Arc$_r$ σ|w_i|w_j, β, A ➔ σ|w_j, β, AU{$r(w_j,w_i)$}
3. Right-Arc$_r$ σ|w_i|w_j, β, A ➔ σ|w_i, β, AU{$r(w_i,w_j)$}
Finish: σ = [w], β = ∅

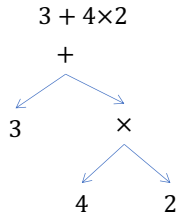


Figure A1: **Applying the transition-based dependency parser to an example of HINT.** It is similar for SCAN and PCFG.

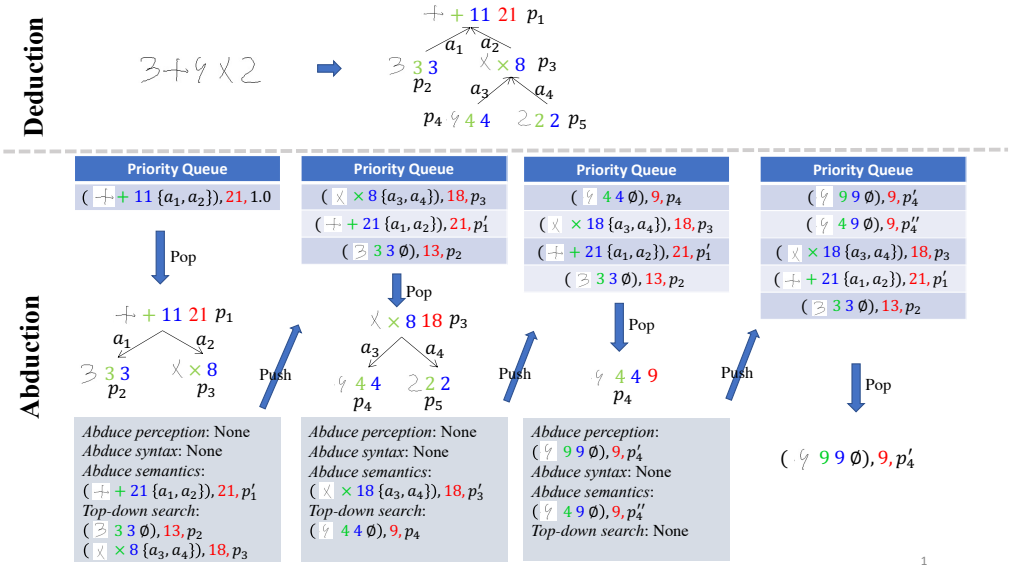
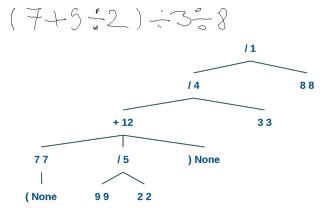


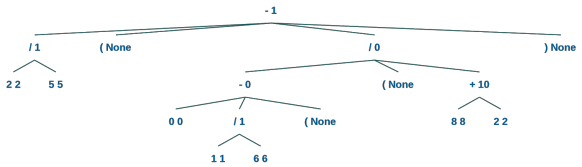
Figure A2: **An illustration of the deduction-abduction process for an example of HINT.** Given a handwritten expression, the system performs a greedy deduction to propose an initial solution, generating a wrong result. In abduction, the root node, paired with the ground-truth result, is first pushed 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.

Test subset I

GT: (7+9/2)/3/8 = 1 PD: (7+9/2)/3/8 = 1

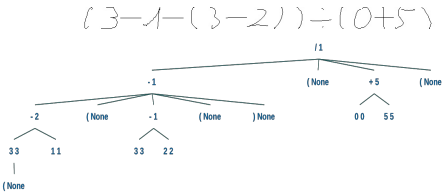


GT: 2/5-(0-1/6)/(8+2) = 1 PD: 2/5-(0-1/6(/(8+2) = 1

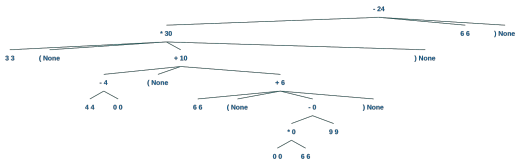


Test subset SS

GT: (3-1-(3-2))/(0+5) = 1 PD: (3-1-(3-2()/(0+5(= 1

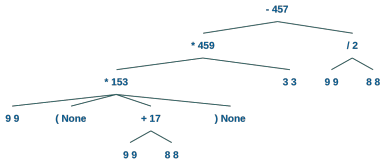


GT: 3*(4-0+(6+(0*6-9))-6) = 12 PD: 3*(4-0+(6+(0*6-9))-6) = 24

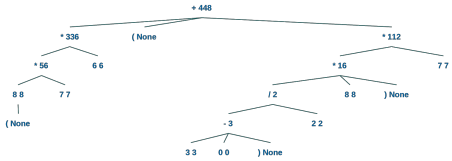


Test subset SL

GT: 9*(9+8)*3-9/8 = 457 PD: 9*(9+8)*3-9/8 = 457

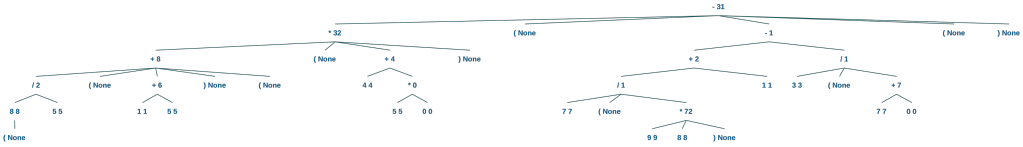


GT: (8*7*6+(3-0)/2*8)*7 = 2464 PD: (8*7*6+(3-0)/2*8)*7 = 448



Test subset LS

GT: (8/5+(1+5))*(4+5*0)-(7/(9*8)+1-3/(7+0)) = 31 PD: (8/5+(1+5)(*(4+5*0)-(7/(9*8)+1-3/(7+0() = 31



Test subset LL

GT: (8*7-5/5)*(3-(2-1)+1)/(9*1*(8+1)+(9+3)-0) = 2 PD: (8*7-5/5)*(3-(2-1)+1)/(9*1*(8+1)/(9+3)-0) = 24

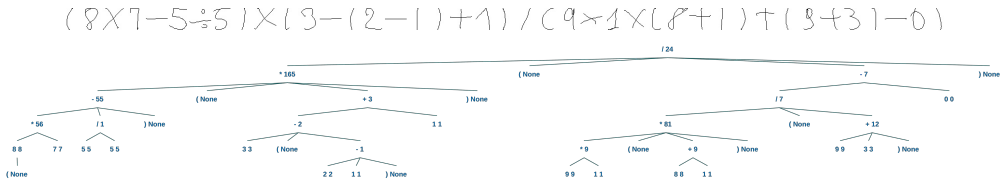


Figure A3: **Examples of NSR predictions on the test set of HINT.** "GT" and "PD" denote "ground-truth" and "prediction," respectively. Each node in the tree is a tuple of (symbol, value).

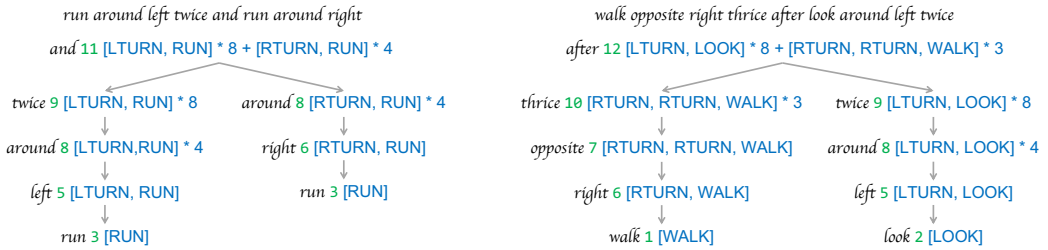


Figure A4: **Examples of NSR predictions on the test set of the SCAN LENGTH split.** We use * (repeating the list) and + (concatenating two lists) to shorten the outputs for easier interpretation.

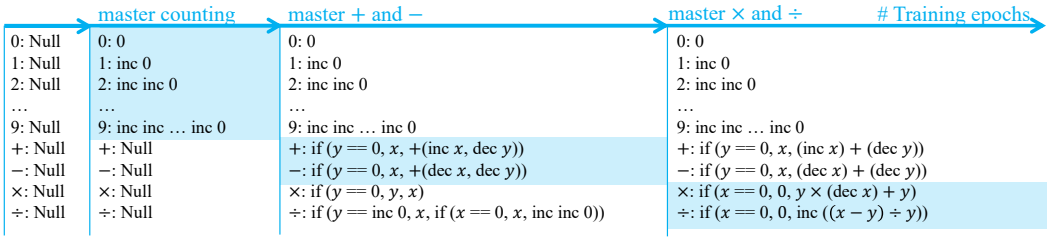


Figure A5: **The evolution of learned programs in NSR for HINT.** The recursive programs in DreamCoder are represented by lambda calculus (a.k.a. λ-calculus) with Y-combinator. Here, we translate the induced programs into pseudo code for easier interpretation. Note that there might be different yet functionally-equivalent programs to represent the semantics of a symbol; we only visualize a plausible one here.

Table A1: **Accuracy of the individual modules of NSR on the HINT dataset.**

Module	Neural Perception	Dependency Parsing	Program Induction
Accuracy	93.51	88.10	98.47

Table A2: **The test accuracy on different splits of SCAN and PCFG.** The results of NeSS on PCFG are reported by adapting the source code from Chen et al. (2020) on PCFG. Reported accuracy (%) is the average of 5 runs with standard deviation if available.

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±0.0	69.2±8.2	56.7±10.2	0.0±0.0	85	56	31
Transformer (Csordás et al., 2021)	-	-	-	20.0	-	96±1	85±1
Transformer (Ontanón et al., 2022)	-	0.0	-	19.6	-	83	63
equivariant Seq2seq (Gordon et al., 2019)	100.0	99.1±0.04	92.0±0.24	15.9±3.2	-	-	-
NeSS (Chen et al., 2020)	100.0	100.0	100.0	100.0	≈0	≈0	≈0
NSR (ours)	**100.0±0.0**	**100.0±0.0**	**100.0±0.0**	**100.0±0.0**	**100±0**	**100±0**	**100±0**

Table A3: **The test accuracy on HINT.** We directly cite the results of GRU, LSTM, and Transformer from Li et al. (2023b). The results of NeSS are reported by adapting its source code on HINT. Reported accuracy (%) is the median and standard deviation of 5 runs.

Model	**Symbol Input**						**Image Input**					
	I	**SS**	**LS**	**SL**	**LL**	**Avg.**	**I**	**SS**	**LS**	**SL**	**LL**	**Avg.**
GRU	76.2±0.6	69.5±0.6	42.8±1.5	10.5±0.2	15.1±1.2	42.5±0.7	66.7±2.0	58.7±2.2	33.1±2.7	9.4±0.3	12.8±1.0	35.9±1.6
LSTM	92.9±1.4	90.9±1.1	74.9±1.5	12.1±0.2	24.3±0.3	58.9±0.7	83.9±0.9	79.7±0.8	62.0±2.5	11.2±0.1	21.0±0.8	51.5±1.0
Transformer	98.0±0.3	96.8±0.6	78.2±2.9	11.7±0.3	22.4±1.1	61.5±0.9	88.4±1.3	86.0±1.3	62.5±4.1	10.9±0.2	19.0±1.0	53.1±1.6
NeSS	≈0	≈0	≈0	≈0	≈0	≈0	-	-	-	-	-	-
NSR (ours)	**98.0±0.2**	**97.3±0.5**	**83.7±1.2**	**95.9±4.6**	**77.6±3.1**	**90.1±2.7**	**88.5±1.0**	**86.2±0.9**	**67.1±2.4**	**83.2±3.9**	**58.2±3.3**	**76.0±2.6**

Algorithm A1: Learning by Deduction-Abduction

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

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

Initial Module: perception $\theta_p^{(0)}$, syntax $\theta_s^{(0)}$, semantics $\theta_l^{(0)}$
for $t \leftarrow 0$ *to* T **do**
  Buffer $\mathcal{B} \leftarrow \varnothing$
  foreach $(x, y) \in D$ **do**
    $T \leftarrow \textsc{Deduce}(x, \theta_p^{(t)}, \theta_s^{(t)}, \theta_l^{(t)})$
    $T^* \leftarrow \textsc{Abduce}(T, y)$
    $\mathcal{B} \leftarrow \mathcal{B} \cup T^*$
  $\theta_p^{(t+1)}, \theta_s^{(t+1)}, \theta_l^{(t+1)} \leftarrow \text{learn}(\mathcal{B}, \theta_p^{(t)}, \theta_s^{(t)}, \theta_l^{(t)})$
return $\theta_p^{(T)}, \theta_s^{(T)}, \theta_l^{(T)}$

Function $\textsc{Deduce}(x, \theta_p, \theta_s, \theta_l)$ **:**
  Sample $\hat{s} \sim p(s|x; \theta_p), \hat{e} \sim p(e|\hat{s}; \theta_s), \hat{v} = f(\hat{s}, \hat{e}; \theta_l)$
  return $T =< (x, \hat{s}, \hat{v}), \hat{e} >$

Function $\textsc{Abduce}(T, y)$ **:**
  $Q \leftarrow$ PriorityQueue()
  $Q.push(\text{root}(T), y, 1.0)$
  while Q *is not empty* **do**
    $A, y_A, p \leftarrow Q.pop()$
    $A \leftarrow (x, w, v, arcs)$
    if $A.v == y_A$ **then**
      return $T(A)$
    `// Abduce perception`
    foreach $w' \in \Sigma$ **do**
      $A' \leftarrow A(w \rightarrow w')$
      if $A'.v == y_A$ **then**
        $Q.push(A', y_A, p(A'))$
    `// Abduce syntax`
    foreach $arc \in arcs$ **do**
      $A' \leftarrow \text{rotate}(A, arc)$
      if $A'.v == y_A$ **then**
        $Q.push(A', y_A, p(A'))$
    `// Abduce semantics`
    $A' \leftarrow A(v \rightarrow y_A)$
    $Q.push(A', y_A, p(A'))$
    `// Top-down search`
    foreach $B \in$ *children*(A) **do**
      $y_B \leftarrow \text{Solve}(B, A, y_A | \theta_l(A.w))$
      $Q.push(B, y_B, p(B))$