From Language to Programs: Bridging Reinforcement Learning and Maximum Marginal Likelihood

Kelvin Guu
Statistics
Stanford University
kguu@stanford.edu

Panupong Pasupat
Computer Science
Stanford University
ppasupat@stanford.edu

Evan Zheran Liu
Computer Science
Stanford University
evanliu@stanford.edu

Percy Liang
Computer Science
Stanford University
pliang@cs.stanford.edu

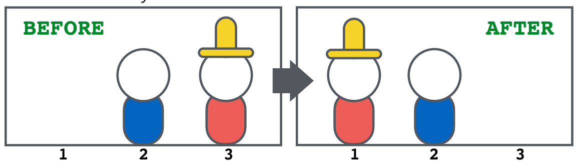
Abstract

Our goal is to learn a semantic parser that maps natural language utterances into executable programs when only indirect supervision is available: examples are labeled with the correct execution result, but not the program itself. Consequently, we must search the space of programs for those that output the correct result, while not being misled by *spurious programs*: incorrect programs that coincidentally output the correct result. We connect two common learning paradigms, reinforcement learning (RL) and maximum marginal likelihood (MML), and then present a new learning algorithm that combines the strengths of both. The new algorithm guards against spurious programs by combining the systematic search traditionally employed in MML with the randomized exploration of RL, and by updating parameters such that probability is spread more evenly across consistent programs. We apply our learning algorithm to a new neural semantic parser and show significant gains over existing state-of-the-art results on a recent context-dependent semantic parsing task.

1 Introduction

We are interested in learning a semantic parser that maps natural language utterances into executable programs (e.g., logical forms). For example, in Figure 1, a program corresponding to the utterance transforms an initial world state into a new world state. We would like to learn from indirect supervision, where each training example is only labeled with the correct output (e.g. a target world state), but not the program that produced that output (Clarke et al., 2010; Liang et al., 2011; Krishnamurthy and Mitchell, 2012; Artzi and Zettlemoyer, 2013; Liang et al., 2017).

"The man in the yellow hat moves to the left of the woman in blue."



Spurious: `move(hasShirt(red), 1)`
Correct: `move(hasHat(yellow), leftOf(hasShirt(blue)))`

Figure 1: The task is to map natural language utterances to a program that manipulates the world state. The correct program captures the true meaning of the utterances, while spurious programs arrive at the correct output for the wrong reasons. We develop methods to prevent the model from being drawn to spurious programs.

The process of constructing a program can be formulated as a sequential decision-making process, where feedback is only received at the end of the sequence when the completed program is executed. In the natural language processing literature, there are two common approaches for handling this situation: 1) reinforcement learning (RL), particularly the REINFORCE algorithm (Williams, 1992; Sutton et al., 1999), which maximizes the expected reward of a sequence of actions; and 2) maximum marginal likelihood (MML), which treats the sequence of actions as a latent variable, and then maximizes the marginal likelihood of observing the correct program output (Dempster et al., 1977).

While the two approaches have enjoyed success on many tasks, we found them to work poorly out of the box for our task. This is because in addition to the sparsity of correct programs, our task also requires weeding out *spurious programs* (Pasupat and Liang, 2016): incorrect interpretations of the utterances that accidentally produce the correct output, as illustrated in Figure 1.

We show that MML and RL optimize closely related objectives. Furthermore, both MML and RL methods have a mechanism for exploring program space in search of programs that generate the correct output. We explain why this exploration tends to quickly concentrate around short spurious programs, causing the model to sometimes overlook the correct program. To address this problem, we propose RANDOMER, a new learning algorithm with two parts:

First, we propose *randomized beam search*, an exploration strategy which combines the systematic beam search traditionally employed in MML with the randomized off-policy exploration of RL. This increases the chance of finding correct programs even when the beam size is small or the parameters are not pre-trained.

Second, we observe that even with good exploration, the gradients of both the RL and MML objectives may still upweight entrenched spurious programs more strongly than correct programs with low probability under the current model. We propose a *meritocratic parameter update rule*, a modification to the MML gradient update, which more equally upweights all programs that produce the correct output. This makes the model less likely to overfit spurious programs.

We apply RANDOMER to train a new neural semantic parser, which outputs programs in a stack-based programming language. We evaluate our resulting system on SCONE, the context-dependent semantic parsing dataset of Long et al. (2016). Our approach outperforms standard RL and MML methods in a direct comparison, and achieves new state-of-the-art results, improving over Long et al. (2016) in all three domains of SCONE, and by over 30% accuracy on the most challenging one.

2 Task

We consider the semantic parsing task in the SCONE dataset[1] (Long et al., 2016). As illustrated in Figure 1, each example consists of a *world* containing several objects (e.g., people), each with certain properties (e.g., shirt color and hat color). Given the initial world state w_0 and a sequence of M natural language utterances $\mathbf{u} = (u_1, \ldots, u_M)$, the task is to generate a program that manipulates the world state according to the utterances. Each utterance u_m describes a single action that transforms the world state w_{m-1} into a new world state w_m. For training, the system receives weakly supervised examples with input $x = (\mathbf{u}, w_0)$ and the target final world state $y = w_M$.

The dataset includes 3 domains: ALCHEMY, TANGRAMS, and SCENE. The description of each domain can be found in Appendix B. The domains highlight different linguistic phenomena: ALCHEMY features ellipsis (e.g., “throw the rest out”, “mix”); TANGRAMS features anaphora on actions (e.g., “repeat step 3”, “bring it back”); and SCENE features anaphora on entities (e.g., “he moves back”, “. . . to his left”). Each domain contains roughly 3,700 training and 900 test examples. Each example contains 5 utterances and is labeled with the target world state after each utterance, but not the target program.

Spurious programs. Given a training example $(\mathbf{u}, w_0, w_M)$, our goal is to find the true underlying program $\mathbf{z}^*$ which reflects the meaning of $\mathbf{u}$. The constraint that $\mathbf{z}^*$ must transform w_0 into w_M, i.e. $\mathbf{z}(w_0) = w_M$, is not enough to uniquely identify the true $\mathbf{z}^*$, as there are often many $\mathbf{z}$ satisfying $\mathbf{z}(w_0) = w_M$: in our experiments, we found at least 1600 on average for each example. Almost all do not capture the meaning of $\mathbf{u}$ (see Figure 1). We refer to these incorrect $\mathbf{z}$'s as *spurious programs*. Such programs encourage the model to learn an incorrect mapping from language to program operations: e.g., the spurious program in Figure 1 would cause the model to learn that “man in the yellow hat” maps to `hasShirt(red)`.

Spurious programs in SCONE. In this dataset, utterances often reference objects in different ways (e.g. a person can be referenced by shirt color, hat color, or position). Hence, any target programming language must also support these different reference strategies. As a result, even a single action such as moving a person to a target destination can be achieved by many different programs, each selecting the person and destination in a different way. Across multiple actions, the number of programs grows combinatorially.[2] Only a few programs actually implement the correct reference strategy as defined by the utterance. This problem would be more severe in any more general-purpose language (e.g. Python).

[1] https://nlp.stanford.edu/projects/scone

[2] The number of well-formed programs in SCENE exceeds 10^{15}

3 Model

We formulate program generation as a sequence prediction problem. We represent a program as a sequence of program tokens in postfix notation; for example, `move(hasHat(yellow), leftOf(hasShirt(blue)))` is linearized as `yellow hasHat blue hasShirt leftOf move`. This representation also allows us to incrementally execute programs from left to right using a stack: constants (e.g., `yellow`) are pushed onto the stack, while functions (e.g., `hasHat`) pop appropriate arguments from the stack and push back the computed result (e.g., the list of people with yellow hats). Appendix B lists the full set of program tokens, $\mathcal{Z}$, and how they are executed. Note that each action always ends with an *action token* (e.g., `move`).

Given an input $x = (\mathbf{u}, w_0)$, the model generates program tokens $z_1, z_2, \ldots$ from left to right using a neural encoder-decoder model with attention (Bahdanau et al., 2015). Throughout the generation process, the model maintains an utterance pointer, m, initialized to 1. To generate z_t, the model's encoder first encodes the utterance u_m into a vector e_m. Then, based on e_m and previously generated tokens $z_{1:t-1}$, the model's decoder defines a distribution $p(z_t \mid x, z_{1:t-1})$ over the possible values of $z_t \in \mathcal{Z}$. The next token z_t is sampled from this distribution. If an action token (e.g., `move`) is generated, the model increments the utterance pointer m. The process terminates when all M utterances are processed. The final probability of generating a particular program $\mathbf{z} = (z_1, \ldots, z_T)$ is $p(\mathbf{z} \mid x) = \prod_{t=1}^{T} p(z_t \mid x, z_{1:t-1})$.

Encoder. The utterance u_m under the pointer is encoded using a bidirectional LSTM:

$$\begin{aligned} h_i^{\mathrm{F}} &= \mathrm{LSTM}(h_{i-1}^{\mathrm{F}}, \Phi_{\mathrm{u}}(u_{m,i})) \\ h_i^{\mathrm{B}} &= \mathrm{LSTM}(h_{i+1}^{\mathrm{B}}, \Phi_{\mathrm{u}}(u_{m,i})) \\ h_i &= [h_i^{\mathrm{F}}; h_i^{\mathrm{B}}], \end{aligned}$$

where $\Phi_{\mathrm{u}}(u_{m,i})$ is the fixed GloVe word embedding (Pennington et al., 2014) of the ith word in u_m. The final utterance embedding is the concatenation $e_m = [h_{|u_m|}^{\mathrm{F}}; h_1^{\mathrm{B}}]$.

Decoder. Unlike Bahdanau et al. (2015), which used a recurrent network for the decoder, we opt for a feed-forward network for simplicity. We use e_m and an embedding $f(z_{1:t-1})$ of the previous execution history (described later) as inputs to compute an attention vector c_t:

$$\begin{aligned} q_t &= \mathrm{ReLU}(W_{\mathrm{q}}[e_m; f(z_{1:t-1})]) \\ \alpha_i &\propto \exp(q_t^\top W_{\mathrm{a}} h_i) \qquad (i = 1, \ldots, |u_m|) \\ c_t &= \sum_i \alpha_i h_i. \end{aligned}$$

Finally, after concatenating q_t with c_t, the distribution over the set $\mathcal{Z}$ of possible program tokens is computed via a softmax:

$$p(z_t \mid x, z_{1:t-1}) \propto \exp(\Phi_{\mathrm{z}}(z_t)^\top W_{\mathrm{s}}[q_t; c_t]),$$

where $\Phi_{\mathrm{z}}(z_t)$ is the embedding for token z_t.

Execution history embedding. We compare two options for $f(z_{1:t-1})$, our embedding of the execution history. A standard approach is to simply take the k most recent tokens $z_{t-k:t-1}$ and concatenate their embeddings. We will refer to this as TOKENS and use $k = 4$ in our experiments.

We also consider a new approach which leverages our ability to incrementally execute programs using a stack. We summarize the execution history by embedding the state of the stack at time $t-1$, achieved by concatenating the embeddings of all values on the stack. (We limit the maximum stack size to 3.) We refer to this as STACK.

4 Reinforcement learning versus maximum marginal likelihood

Having formulated our task as a sequence prediction problem, we must still choose a learning algorithm. We first compare two standard paradigms: reinforcement learning (RL) and maximum marginal likelihood (MML). In the next section, we propose a better alternative.

4.1 Comparing objective functions

Reinforcement learning. From an RL perspective, given a training example (x, y), a policy makes a sequence of decisions $\mathbf{z} = (z_1, \ldots, z_T)$, and then receives a reward at the end of the episode: $R(\mathbf{z}) = 1$ if $\mathbf{z}$ executes to y and 0 otherwise (dependence on x and y has been omitted from the notation).

We focus on *policy gradient* methods, in which a stochastic policy function is trained to maximize the expected reward. In our setup, $p_\theta(\mathbf{z} \mid x)$ is the policy (with parameters θ), and its expected reward on a given example (x, y) is

$$G(x, y) = \sum_{\mathbf{z}} R(\mathbf{z})\, p_\theta(\mathbf{z} \mid x), \qquad (1)$$

where the sum is over all possible programs. The overall RL objective, J_{RL}, is the expected reward across examples:

$$J_{\text{RL}} = \sum_{(x,y)} G(x, y). \tag{2}$$

Maximum marginal likelihood. The MML perspective assumes that y is generated by a partially-observed random process: conditioned on x, a latent program $\mathbf{z}$ is generated, and conditioned on $\mathbf{z}$, the observation y is generated. This implies the marginal likelihood:

$$p_\theta(y \mid x) = \sum_{\mathbf{z}} p(y \mid \mathbf{z})\, p_\theta(\mathbf{z} \mid x). \tag{3}$$

Note that since the execution of $\mathbf{z}$ is deterministic, $p_\theta(y \mid \mathbf{z}) = 1$ if $\mathbf{z}$ executes to y and 0 otherwise. The log marginal likelihood of the data is then

$$J_{\text{MML}} = \log \mathcal{L}_{\text{MML}}, \tag{4}$$

$$\text{where} \quad \mathcal{L}_{\text{MML}} = \prod_{(x,y)} p_\theta(y \mid x). \tag{5}$$

To estimate our model parameters θ, we maximize J_{MML} with respect to θ.

With our choice of reward, the RL expected reward (1) is equal to the MML marginal probability (3). Hence the only difference between the two formulations is that in RL we optimize the *sum* of expected rewards (2), whereas in MML we optimize the *product* (5).[3]

4.2 Comparing gradients

In both policy gradient and MML, the objectives are typically optimized via (stochastic) gradient ascent. The gradients of J_{RL} and J_{MML} are closely related. They both have the form:

$$\begin{aligned}\nabla_\theta J &= \sum_{(x,y)} \mathbb{E}_{\mathbf{z}\sim q}\left[R(\mathbf{z}) \nabla \log p_\theta(\mathbf{z} \mid x)\right] \qquad (6)\\ &= \sum_{(x,y)} \sum_{\mathbf{z}} q(\mathbf{z}) R(\mathbf{z}) \nabla \log p_\theta(\mathbf{z} \mid x),\end{aligned}$$

where $q(\mathbf{z})$ equals

$$q_{\text{RL}}(\mathbf{z}) = p_\theta(\mathbf{z} \mid x) \quad \text{for } J_{\text{RL}}, \tag{7}$$

$$\begin{aligned} q_{\text{MML}}(\mathbf{z}) &= \frac{R(\mathbf{z}) p_\theta(\mathbf{z} \mid x)}{\sum_{\tilde{\mathbf{z}}} R(\tilde{\mathbf{z}}) p_\theta(\tilde{\mathbf{z}} \mid x)} \qquad (8)\\ &= p_\theta(\mathbf{z} \mid x, R(\mathbf{z}) \neq 0) \quad \text{for } J_{\text{MML}}.\end{aligned}$$

[3] Note that the log of the product in (5) does *not* equal the sum in (2).

Taking a step in the direction of $\nabla \log p_\theta(\mathbf{z} \mid x)$ upweights the probability of $\mathbf{z}$, so we can heuristically think of the gradient as attempting to upweight each reward-earning program $\mathbf{z}$ by a *gradient weight* $q(\mathbf{z})$. In Subsection 5.2, we argue why q_{MML} is better at guarding against spurious programs, and propose an even better alternative.

4.3 Comparing gradient approximation strategies

It is often intractable to compute the gradient (6) because it involves taking an expectation over all possible programs. So in practice, the expectation is approximated.

In the policy gradient literature, *Monte Carlo integration* (MC) is the typical approximation strategy. For example, the popular REINFORCE algorithm (Williams, 1992) uses Monte Carlo sampling to compute an unbiased estimate of the gradient:

$$\Delta_{\text{MC}} = \frac{1}{B} \sum_{\mathbf{z}\in\mathcal{S}} [R(\mathbf{z}) - c] \nabla \log p_\theta(\mathbf{z} \mid x), \tag{9}$$

where $\mathcal{S}$ is a collection of B samples $\mathbf{z}^{(b)} \sim q(\mathbf{z})$, and c is a *baseline* (Williams, 1992) used to reduce the variance of the estimate without altering its expectation.

In the MML literature for latent sequences, the expectation is typically approximated via *numerical integration* (NUM) instead:

$$\Delta_{\text{NUM}} = \sum_{\mathbf{z}\in\mathcal{S}} q(\mathbf{z}) R(\mathbf{z}) \nabla \log p_\theta(\mathbf{z} \mid x). \tag{10}$$

where the programs in $\mathcal{S}$ come from beam search.

Beam search. Beam search generates a set of programs via the following process. At step t of beam search, we maintain a beam $\mathcal{B}_t$ of at most B *search states*. Each state $s \in \mathcal{B}_t$ represents a partially constructed program, $s = (z_1, \ldots, z_t)$ (the first t tokens of the program). For each state s in the beam, we generate all possible *continuations*,

$$\begin{aligned}\texttt{cont}(s) &= \texttt{cont}((z_1, \ldots, z_t))\\ &= \{(z_1, \ldots, z_t, z_{t+1}) \mid z_{t+1} \in \mathcal{Z}\}.\end{aligned}$$

We then take the union of these continuations, $\texttt{cont}(\mathcal{B}_t) = \bigcup_{s\in\mathcal{B}_t} \texttt{cont}(s)$. The new beam $\mathcal{B}_{t+1}$ is simply the highest scoring B continuations in $\texttt{cont}(\mathcal{B}_t)$, as scored by the policy, $p_\theta(s \mid x)$. Search is halted after a fixed number of iterations or when there are no continuations possible. $\mathcal{S}$ is then the set of all complete programs discovered during beam search. We will refer to this as *beam search MML* (BS-MML).

In both policy gradient and MML, we think of the procedure used to produce the set of programs $\mathcal{S}$ as an *exploration strategy* which searches for programs that produce reward. One advantage of numerical integration is that it allows us to decouple the *exploration strategy* from the *gradient weights* assigned to each program.

5 Tackling spurious programs

In this section, we illustrate why spurious programs are problematic for the most commonly used methods in RL (REINFORCE) and MML (beam search MML). We describe two key problems and propose a solution to each, based on insights gained from our comparison of RL and MML in Section 4.

5.1 Spurious programs bias exploration

As mentioned in Section 4, REINFORCE and BS-MML both employ an *exploration strategy* to approximate their respective gradients. In both methods, exploration is guided by the current model policy, whereby programs with high probability under the current policy are more likely to be explored. A troubling implication is that programs with low probability under the current policy are likely to be overlooked by exploration.

If the current policy incorrectly assigns low probability to the correct program $\mathbf{z}^*$, it will likely fail to discover $\mathbf{z}^*$ during exploration, and will consequently fail to upweight the probability of $\mathbf{z}^*$. This repeats on every gradient step, keeping the probability of $\mathbf{z}^*$ perpetually low. The same feedback loop can also cause already high-probability spurious programs to gain even more probability. From this, we see that exploration is sensitive to initial conditions: the rich get richer, and the poor get poorer.

Since there are often thousands of spurious programs and only a few correct programs, spurious programs are usually found first. Once spurious programs get a head start, exploration increasingly biases towards them.

As a remedy, one could try initializing parameters such that the model puts a uniform distribution over all possible programs. A seemingly reasonable tactic is to initialize parameters such that the

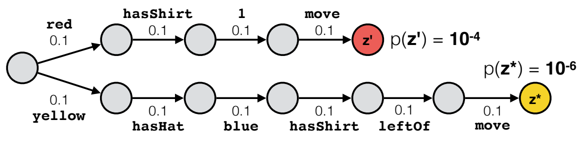


Figure 2: Two possible paths in the tree of all possible programs. One path leads to the spurious program $\mathbf{z}'$ (red) while the longer path leads to the correct program $\mathbf{z}^*$ (gold). Each edge represents a decision and shows the probability of that decision under a uniform policy. The shorter program has two orders of magnitude higher probability.

model policy puts near-uniform probability over the decisions at each time step. However, this causes shorter programs to have orders of magnitude higher probability than longer programs, as illustrated in Figure 2 and as we empirically observe. A more sophisticated approach might involve approximating the total number of programs reachable from each point in the program-generating decision tree. However, we instead propose to reduce sensitivity to the initial distribution over programs.

Solution: randomized beam search

One solution to biased exploration is to simply rely less on the untrustworthy current policy. We can do this by injecting random noise into exploration.

In REINFORCE, a common solution is to sample from an ϵ-greedy variant of the current policy. On the other hand, MML exploration with beam search is deterministic. However, it has a key advantage over REINFORCE-style sampling: even if one program occupies almost all probability under the current policy (a peaky distribution), beam search will still use its remaining beam capacity to explore at least $B - 1$ other programs. In contrast, sampling methods will repeatedly visit the mode of the distribution.

To get the best of both worlds, we propose a simple *ϵ-greedy randomized beam search*. Like regular beam search, at iteration t we compute the set of all continuations $\texttt{cont}(\mathcal{B}_t)$ and sort them by their model probability $p_\theta(s \mid x)$. But instead of selecting the B highest-scoring continuations, we choose B continuations one by one without replacement from $\texttt{cont}(\mathcal{B}_t)$. When choosing a continuation from the remaining pool, we either uniformly sample a random continuation with probability ϵ, or pick the highest-scoring continuation in the pool with probability $1 - \epsilon$. Empirically, we find that this performs much better than both classic beam search and ϵ-greedy sampling (Table 3).

5.2 Spurious programs dominate gradients

In both RL and MML, even if exploration is perfect and the gradient is exactly computed, spurious programs can still be problematic.

Even if perfect exploration visits every program, we see from the gradient weights $q(\mathbf{z})$ in (7) and (8) that programs are weighted proportional to their current policy probability. If a spurious program $\mathbf{z}'$ has 100 times higher probability than $\mathbf{z}^*$ as in Figure 2, the gradient will spend roughly 99% of its magnitude upweighting towards $\mathbf{z}'$ and only 1% towards $\mathbf{z}^*$ even though the two programs get the same reward.

This implies that it would take many updates for $\mathbf{z}^*$ to catch up. In fact, $\mathbf{z}^*$ may never catch up, depending on the gradient updates for other training examples. Simply increasing the learning rate is inadequate, as it would cause the model to take overly large steps towards $\mathbf{z}'$, potentially causing optimization to diverge.

Solution: the meritocratic update rule

To solve this problem, we want the upweighting to be more "meritocratic": any program that obtains reward should be upweighted roughly equally.

We first observe that J_{MML} already improves over J_{RL} in this regard. From (6), we see that the gradient weight $q_{\text{MML}}(\mathbf{z})$ is the policy distribution restricted to and renormalized over only reward-earning programs. This renormalization makes the gradient weight uniform *across examples*: even if all reward-earning programs for a particular example have very low model probability, their combined gradient weight $\sum_{\mathbf{z}} q_{\text{MML}}(\mathbf{z})$ is always 1. In our experiments, J_{MML} performs significantly better than J_{RL} (Table 4).

However, while J_{MML} assigns uniform weight across examples, it is still not uniform over the programs *within each example*. Hence we propose a new update rule which goes one step further in pursuing uniform updates. Extending $q_{\text{MML}}(\mathbf{z})$, we define a β-smoothed version:

$$q_\beta(\mathbf{z}) = \frac{q_{\text{MML}}(\mathbf{z})^\beta}{\sum_{\tilde{\mathbf{z}}} q_{\text{MML}}(\tilde{\mathbf{z}})^\beta}. \tag{11}$$

When $\beta = 0$, our weighting is completely uniform across all reward-earning programs within an example while $\beta = 1$ recovers the original MML weighting. Our new update rule is to simply take a modified gradient step where $q = q_\beta$.[4] We will refer to this as the *β-meritocratic* update rule.

5.3 Summary of the proposed approach

We described two problems[5] and their solutions: we reduce exploration bias using *ϵ-greedy randomized beam search* and perform more balanced optimization using the *β-meritocratic parameter update rule*. We call our resulting approach RANDOMER. Table 1 summarizes how RANDOMER combines desirable qualities from both REINFORCE and BS-MML.

6 Experiments

Evaluation. We evaluate our proposed methods on all three domains of the SCONE dataset. Accuracy is defined as the percentage of test examples where the model produces the correct final world state w_M. All test examples have $M = 5$ (5utts), but we also report accuracy after processing the first 3 utterances (3utts). To control for the effects of randomness, we train 5 instances of each model with different random seeds. We report the median accuracy of the instances unless otherwise noted.

Training. Following Long et al. (2016), we decompose each training example into smaller examples. Given an example with 5 utterances, $\mathbf{u} = [u_1, \dots, u_5]$, we consider all length-1 and length-2 substrings of $\mathbf{u}$: $[u_1], [u_2], \dots, [u_3, u_4], [u_4, u_5]$ (9 total). We form a new training example from each substring, e.g., $(\mathbf{u}', w_0', w_M')$ where $\mathbf{u}' = [u_4, u_5]$, $w_0' = w_3$ and $w_M' = w_5$.

All models are implemented in TensorFlow (Abadi et al., 2015). Model parameters are randomly initialized (Glorot and Bengio, 2010), with no pre-training. We use the Adam optimizer (Kingma and Ba, 2014) (which is applied to the gradient in (6)), a learning rate of 0.001, a mini-batch size of 8 examples (different from the beam size), and train until accuracy on the validation set converges (on average about 13,000 steps). We

[4] Also, note that if exploration were exhaustive, $\beta = 0$ would be equivalent to supervised learning using the set of all reward-earning programs as targets.

[5] These problems concern the gradient w.r.t. a single example. The full gradient averages over multiple examples, which helps separate correct from spurious. E.g., if multiple examples all mention "yellow hat", we will find a correct program parsing this as `hasHat(yellow)` for each example, whereas the spurious programs we find will follow no consistent pattern. Consequently, spurious gradient contributions may cancel out while correct program gradients will all "vote" in the same direction.

Method	Approximation of $E_q[\cdot]$	Exploration strategy	Gradient weight $q(\mathbf{z})$
REINFORCE	Monte Carlo integration	independent sampling	$p_\theta(\mathbf{z} \mid x)$
BS-MML	numerical integration	beam search	$p_\theta(\mathbf{z} \mid x, R(\mathbf{z}) \neq 0)$
RANDOMER	numerical integration	randomized beam search	$q_\beta(\mathbf{z})$

Table 1: RANDOMER combines qualities of both REINFORCE (RL) and BS-MML. For approximating the expectation over q in the gradient, we use numerical integration as in BS-MML. Our exploration strategy is a hybrid of search (MML) and off-policy sampling (RL). Our gradient weighting is equivalent to MML when $\beta = 1$ and more "meritocratic" than both MML and REINFORCE for lower values of β.

use fixed GloVe vectors (Pennington et al., 2014) to embed the words in each utterance.

Hyperparameters. For all models, we performed a grid search over hyperparameters to maximize accuracy on the validation set. Hyperparameters include the learning rate, the baseline in REINFORCE, ϵ-greediness and β-meritocraticness. For REINFORCE, we also experimented with a regression-estimated baseline (Ranzato et al., 2015), but found it to perform worse than a constant baseline.

6.1 Main results

Comparison to prior work. Table 2 compares RANDOMER to results from Long et al. (2016) as well as two baselines, REINFORCE and BS-MML (using the same neural model but different learning algorithms). Our approach achieves new state-of-the-art results by a significant margin, especially on the SCENE domain, which features the most complex program syntax. We report the results for REINFORCE, BS-MML, and RANDOMER on the seed and hyperparameters that achieve the best validation accuracy.

We note that REINFORCE performs very well on TANGRAMS but worse on ALCHEMY and very poorly on SCENE. This might be because the program syntax for TANGRAMS is simpler than the other two: there is no other way to refer to objects except by index.

We also found that REINFORCE required ϵ-greedy exploration to make any progress. Using ϵ-greedy greatly skews the Monte Carlo approximation of ∇J_{RL}, making it more uniformly weighted over programs in a similar spirit to using β-meritocratic gradient weights q_β. However, q_β increases uniformity over reward-earning programs only, rather than over all programs.

Effect of randomized beam search. Table 3 shows that ϵ-greedy randomized beam search consistently outperforms classic beam search. Even when we increase the beam size of classic beam search to 128, it still does not surpass randomized beam search with a beam of 32, and further increases yield no additional improvement.

	ALCHEMY		TANGRAMS		SCENE	
system	3utts	5utts	3utts	5utts	3utts	5utts
LONG+16	56.8	52.3	64.9	27.6	23.2	14.7
REINFORCE	58.3	44.6	**68.5**	**37.3**	47.8	33.9
BS-MML	58.7	47.3	62.6	32.2	53.5	32.5
RANDOMER	**66.9**	**52.9**	65.8	37.1	**64.8**	**46.2**

Table 2: **Comparison to prior work.** LONG+16 results are directly from Long et al. (2016). Hyperparameters are chosen by best performance on validation set (see Appendix A).

		ALCHEMY		TANGRAMS		SCENE	
random	beam	3utts	5utts	3utts	5utts	3utts	5utts
classic beam search							
None	32	30.3	23.2	0.0	0.0	33.4	20.1
None	128	59.0	46.4	60.9	28.6	24.5	13.9
randomized beam search							
$\epsilon = 0.05$	32	58.7	45.5	61.1	32.5	33.4	23.0
$\epsilon = 0.15$	32	**61.3**	48.3	**65.2**	**34.3**	50.8	33.5
$\epsilon = 0.25$	32	60.5	**48.6**	60.0	27.3	**54.1**	**35.7**

Table 3: **Randomized beam search.** All listed models use gradient weight q_{MML} and TOKENS to represent execution history.

Effect of β-meritocratic updates. Table 4 evaluates the impact of β-meritocratic parameter updates (gradient weight q_β). More uniform upweighting across reward-earning programs leads to higher accuracy and fewer spurious programs, especially in SCENE. However, no single value of β performs best over all domains.

Choosing the right value of β in RANDOMER significantly accelerates training. Figure 3 illustrates that while $\beta = 0$ and $\beta = 1$ ultimately achieve similar accuracy on ALCHEMY, $\beta = 0$ reaches good performance in half the time.

Since lowering β reduces trust in the model policy, $\beta < 1$ helps in early training when the current policy is untrustworthy. However, as it grows more trustworthy, $\beta < 1$ begins to pay a price for ignoring it. Hence, it may be worthwhile to anneal β towards 1 over time.

	ALCHEMY		TANGRAMS		SCENE	
$q(\mathbf{z})$	3utts	5utts	3utts	5utts	3utts	5utts
q_{RL}	0.2	0.0	0.9	0.6	0.0	0.0
q_{MML} $(q_{\beta=1})$	61.3	48.3	**65.2**	**34.3**	50.8	33.5
$q_{\beta=0.25}$	**64.4**	**48.9**	60.6	29.0	42.4	29.7
$q_{\beta=0}$	63.6	46.3	54.0	23.5	**61.0**	**42.4**

Table 4: ***β*-meritocratic updates.** All listed models use randomized beam search, $\epsilon = 0.15$ and TOKENS to represent execution history.

	ALCHEMY		TANGRAMS		SCENE	
	3utts	5utts	3utts	5utts	3utts	5utts
HISTORY	61.3	48.3	**65.2**	**34.3**	50.8	33.5
STACK	**64.2**	**53.2**	63.0	32.4	**59.5**	**43.1**

Table 5: **TOKENS vs STACK embedding.** Both models use $\epsilon = 0.15$ and gradient weight q_{MML}.

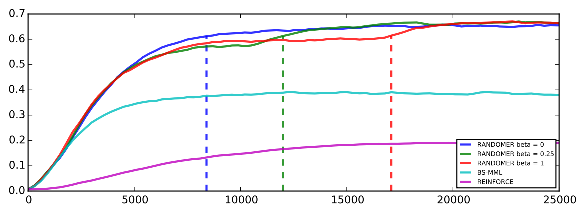


Figure 3: Validation set accuracy (y-axis) across training iterations (x-axis) on ALCHEMY. We compare RANDOMER, BS-MML and REINFORCE. Vertical lines mark the first time each model surpasses 60% accuracy. RANDOMER with $\beta = 0$ reaches this point twice as fast as $\beta = 1$. REINFORCE plateaus for a long time, then begins to climb after 40k iterations (not shown). Training runs are averaged over 5 seeds.

Effect of execution history embedding. Table 5 compares our two proposals for embedding the execution history: TOKENS and STACK. STACK performs better in the two domains where an object can be referenced in multiple ways (SCENE and ALCHEMY). STACK directly embeds objects on the stack, invariant to the way in which they were pushed onto the stack, unlike TOKENS. We hypothesize that this invariance increases robustness to spurious behavior: if a program accidentally pushes the right object onto the stack via spurious means, the model can still learn the remaining steps of the program without conditioning on a spurious history.

Fitting vs overfitting the training data. Table 6 reveals that BS-MML and RANDOMER use different strategies to fit the training data. On the depicted training example, BS-MML actually achieves higher expected reward / marginal probability than RANDOMER, but it does so by putting most of its probability on a spurious program—a form of overfitting. In contrast, RANDOMER spreads probability mass over multiple reward-earning programs, including the correct ones.

As a consequence of overfitting, we observed at test time that BS-MML only references people by positional indices instead of by shirt or hat color, whereas RANDOMER successfully learns to use multiple reference strategies.

7 Related work and discussion

Semantic parsing from indirect supervision. Our work is motivated by the classic problem of learning semantic parsers from indirect supervision (Clarke et al., 2010; Liang et al., 2011; Artzi and Zettlemoyer, 2011, 2013; Reddy et al., 2014; Pasupat and Liang, 2015). We are interested in the initial stages of training from scratch, where getting any training signal is difficult due to the combinatorially large search space. We also highlighted the problem of spurious programs which capture reward but give incorrect generalizations.

Maximum marginal likelihood with beam search (BS-MML) is traditionally used to learn semantic parsers from indirect supervision.

Reinforcement learning. Concurrently, there has been a recent surge of interest in reinforcement learning, along with the wide application of the classic REINFORCE algorithm (Williams, 1992)—to troubleshooting (Branavan et al., 2009), dialog generation (Li et al., 2016), game playing (Narasimhan et al., 2015), coreference resolution (Clark and Manning, 2016), machine translation (Norouzi et al., 2016), and even semantic parsing (Liang et al., 2017). Indeed, the challenge of training semantic parsers from indirect supervision is perhaps better captured by the notion of sparse rewards in reinforcement learning.

The RL answer would be better exploration, which can take many forms including simple action-dithering such as ϵ-greedy, entropy regularization (Williams and Peng, 1991), Monte Carlo tree search (Coulom, 2006), randomized value functions (Osband et al., 2014, 2016), and methods which prioritize learning environment dynamics (Duff, 2002) or under-explored states (Kearns and Singh, 2002; Bellemare et al., 2016; Nachum et al., 2016). The majority of these methods employ Monte Carlo sampling for exploration. In

Utterance: the man in the purple shirt and red hat moves just to the right of the man in the red shirt and yellow hat

	program	prob
	RANDOMER ($\epsilon = 0.15, \beta = 0$)	
⋆	`move(hasHat(red), rightOf(hasHat(red)))`	0.122
⋆	`move(hasShirt(purple), rightOf(hasShirt(red)))`	0.061
o	`move(hasHat(red), rightOf(index(allPeople, 1)))`	0.059
⋆	`move(hasHat(red), rightOf(hasHat(yellow)))`	0.019
o	`move(index(allPeople, 2), rightOf(hasShirt(red)))`	0.018
x	`move(hasHat(red), 8)`	0.018
	BS-MML	
o	`move(index(allPeople, 2), 2)`	0.887
x	`move(index(allPeople, 2), 6)`	0.041
x	`move(index(allPeople, 2), 5)`	0.020
x	`move(index(allPeople, 2), 8)`	0.016
x	`move(index(allPeople, 2), 7)`	0.009
x	`move(index(allPeople, 2), 3)`	0.008

Table 6: Top-scoring predictions for a *training* example from SCENE (* = correct, o = spurious, x = incorrect). RANDOMER distributes probability mass over numerous reward-earning programs (including the correct ones), while classic beam search MML overfits to one spurious program, giving it very high probability.

contrast, we find randomized beam search to be more suitable in our setting, because it explores low-probability states even when the policy distribution is peaky. Our β-meritocratic update also depends on the fact that beam search returns an *entire set* of reward-earning programs rather than one, since it renormalizes over the reward-earning set. While similar to entropy regularization, β-meritocratic update is more targeted as it only increases uniformity of the gradient among reward-earning programs, rather than across all programs.

Our strategy of using randomized beam search and meritocratic updates lies closer to MML than RL, but this does not imply that RL has nothing to offer in our setting. With the simple connection between RL and MML we established, much of the literature on exploration and variance reduction in RL can be directly applied to MML problems. Of special interest are methods which incorporate a value function such as actor-critic.

Maximum likelihood and RL. It is tempting to group our approach with sequence learning methods which interpolate between supervised learning and reinforcement learning (Ranzato et al., 2015; Venkatraman et al., 2015; Ross et al., 2011; Norouzi et al., 2016; Bengio et al., 2015; Levine, 2014). These methods generally seek to make RL training easier by pre-training or "warm-starting" with fully supervised learning. This requires each training example to be labeled with a reasonably correct output sequence. In our setting, this would amount to labeling each example with the correct program, which is not known. Hence, these methods cannot be directly applied.

Without access to correct output sequences, we cannot directly maximize likelihood, and instead resort to maximizing the *marginal* likelihood (MML). Rather than proposing MML as a form of pre-training, we argue that MML is a superior substitute for the standard RL objective, and that the β-meritocratic update is even better.

Simulated annealing. Our β-meritocratic update employs exponential smoothing, which bears resemblance to the simulated annealing strategy of Och (2003); Smith and Eisner (2006); Shen et al. (2015). However, a key difference is that these methods smooth the objective function whereas we smooth an expectation in the gradient. To underscore the difference, we note that fixing $\beta = 0$ in our method (total smoothing) is quite effective, whereas total smoothing in the simulated annealing methods would correspond to a completely flat objective function, and an uninformative gradient of zero everywhere.

Neural semantic parsing. There has been recent interest in using recurrent neural networks for semantic parsing, both for modeling logical forms (Dong and Lapata, 2016; Jia and Liang, 2016; Liang et al., 2017) and for end-to-end execution (Yin et al., 2015; Neelakantan et al., 2016). We develop a neural model for the context-dependent setting, which is made possible by a new stack-based language similar to Riedel et al. (2016).

Acknowledgments. This work was supported by the NSF Graduate Research Fellowship under No. DGE-114747 and the NSF CAREER Award under No. IIS-1552635.

Reproducibility. Our code is made available at https://github.com/kelvinguu/lang2program. Reproducible experiments are available at https://worksheets.codalab.org/worksheets/0x88c914ee1d4b4a4587a07f36f090f3e5/.

References

M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean,

M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. 2015. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* .

Y. Artzi and L. Zettlemoyer. 2011. Bootstrapping semantic parsers from conversations. In *Empirical Methods in Natural Language Processing (EMNLP)*. pages 421–432.

Y. Artzi and L. Zettlemoyer. 2013. Weakly supervised learning of semantic parsers for mapping instructions to actions. *Transactions of the Association for Computational Linguistics (TACL)* 1:49–62.

D. Bahdanau, K. Cho, and Y. Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations (ICLR)*.

M. Bellemare, S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, and R. Munos. 2016. Unifying count-based exploration and intrinsic motivation. In *Advances in Neural Information Processing Systems (NIPS)*. pages 1471–1479.

S. Bengio, O. Vinyals, N. Jaitly, and N. Shazeer. 2015. Scheduled sampling for sequence prediction with recurrent neural networks. In *Advances in Neural Information Processing Systems (NIPS)*. pages 1171–1179.

S. Branavan, H. Chen, L. S. Zettlemoyer, and R. Barzilay. 2009. Reinforcement learning for mapping instructions to actions. In *Association for Computational Linguistics and International Joint Conference on Natural Language Processing (ACL-IJCNLP)*. pages 82–90.

K. Clark and C. D. Manning. 2016. Deep reinforcement learning for mention-ranking coreference models. *arXiv preprint arXiv:1609.08667* .

J. Clarke, D. Goldwasser, M. Chang, and D. Roth. 2010. Driving semantic parsing from the world's response. In *Computational Natural Language Learning (CoNLL)*. pages 18–27.

R. Coulom. 2006. Efficient selectivity and backup operators in Monte-Carlo tree search. In *International Conference on Computers and Games*. pages 72–83.

A. P. Dempster, L. N. M., and R. D. B. 1977. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society: Series B* 39(1):1–38.

L. Dong and M. Lapata. 2016. Language to logical form with neural attention. In *Association for Computational Linguistics (ACL)*.

M. O. Duff. 2002. *Optimal Learning: Computational procedures for Bayes-adaptive Markov decision processes*. Ph.D. thesis, University of Massachusetts Amherst.

X. Glorot and Y. Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *International Conference on Artificial Intelligence and Statistics*.

R. Jia and P. Liang. 2016. Data recombination for neural semantic parsing. In *Association for Computational Linguistics (ACL)*.

M. Kearns and S. Singh. 2002. Near-optimal reinforcement learning in polynomial time. *Machine Learning* 49(2):209–232.

D. Kingma and J. Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* .

J. Krishnamurthy and T. Mitchell. 2012. Weakly supervised training of semantic parsers. In *Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP/CoNLL)*. pages 754–765.

S. Levine. 2014. *Motor Skill Learning with Local Trajectory Methods*. Ph.D. thesis, Stanford University.

J. Li, W. Monroe, A. Ritter, D. Jurafsky, M. Galley, and J. Gao. 2016. Deep reinforcement learning for dialogue generation. In *Empirical Methods in Natural Language Processing (EMNLP)*.

C. Liang, J. Berant, Q. Le, and K. D. F. N. Lao. 2017. Neural symbolic machines: Learning semantic parsers on Freebase with weak supervision. In *Association for Computational Linguistics (ACL)*.

P. Liang, M. I. Jordan, and D. Klein. 2011. Learning dependency-based compositional semantics. In *Association for Computational Linguistics (ACL)*. pages 590–599.

R. Long, P. Pasupat, and P. Liang. 2016. Simpler context-dependent logical forms via model projections. In *Association for Computational Linguistics (ACL)*.

O. Nachum, M. Norouzi, and D. Schuurmans. 2016. Improving policy gradient by exploring under-appreciated rewards. *arXiv preprint arXiv:1611.09321* .

K. Narasimhan, T. Kulkarni, and R. Barzilay. 2015. Language understanding for text-based games using deep reinforcement learning. *arXiv preprint arXiv:1506.08941* .

A. Neelakantan, Q. V. Le, and I. Sutskever. 2016. Neural programmer: Inducing latent programs with gradient descent. In *International Conference on Learning Representations (ICLR)*.

M. Norouzi, S. Bengio, N. Jaitly, M. Schuster, Y. Wu, D. Schuurmans, et al. 2016. Reward augmented maximum likelihood for neural structured prediction. In *Advances In Neural Information Processing Systems*. pages 1723–1731.

F. J. Och. 2003. Minimum error rate training in statistical machine translation. In *Association for Computational Linguistics (ACL)*. pages 160–167.

I. Osband, C. Blundell, A. Pritzel, and B. V. Roy. 2016. Deep exploration via bootstrapped DQN. In *Advances In Neural Information Processing Systems*. pages 4026–4034.

I. Osband, B. V. Roy, and Z. Wen. 2014. Generalization and exploration via randomized value functions. *arXiv preprint arXiv:1402.0635* .

P. Pasupat and P. Liang. 2015. Compositional semantic parsing on semi-structured tables. In *Association for Computational Linguistics (ACL)*.

P. Pasupat and P. Liang. 2016. Inferring logical forms from denotations. In *Association for Computational Linguistics (ACL)*.

J. Pennington, R. Socher, and C. D. Manning. 2014. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*.

M. Ranzato, S. Chopra, M. Auli, and W. Zaremba. 2015. Sequence level training with recurrent neural networks. *arXiv preprint arXiv:1511.06732* .

S. Reddy, M. Lapata, and M. Steedman. 2014. Large-scale semantic parsing without question-answer pairs. *Transactions of the Association for Computational Linguistics (TACL)* 2(10):377–392.

S. Riedel, M. Bosnjak, and T. Rocktäschel. 2016. Programming with a differentiable forth interpreter. *CoRR, abs/1605.06640* .

S. Ross, G. Gordon, and A. Bagnell. 2011. A reduction of imitation learning and structured prediction to no-regret online learning. In *Artificial Intelligence and Statistics (AISTATS)*.

S. Shen, Y. Cheng, Z. He, W. He, H. Wu, M. Sun, and Y. Liu. 2015. Minimum risk training for neural machine translation. *arXiv preprint arXiv:1512.02433* .

D. A. Smith and J. Eisner. 2006. Minimum risk annealing for training log-linear models. In *International Conference on Computational Linguistics and Association for Computational Linguistics (COLING/ACL)*. pages 787–794.

R. Sutton, D. McAllester, S. Singh, and Y. Mansour. 1999. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems (NIPS)*.

A. Venkatraman, M. Hebert, and J. A. Bagnell. 2015. Improving multi-step prediction of learned time series models. In *Association for the Advancement of Artificial Intelligence (AAAI)*. pages 3024–3030.

R. J. Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning* 8(3):229–256.

R. J. Williams and J. Peng. 1991. Function optimization using connectionist reinforcement learning algorithms. *Connection Science* 3(3):241–268.

P. Yin, Z. Lu, H. Li, and B. Kao. 2015. Neural enquirer: Learning to query tables. *arXiv preprint arXiv:1512.00965* .

A Hyperparameters in Table 2

System	ALCHEMY	TANGRAMS	SCENE
REINFORCE	Sample size 32 Baseline 10^{-2} $\epsilon = 0.15$ embed TOKENS	Sample size 32 Baseline 10^{-2} $\epsilon = 0.15$ embed TOKENS	Sample size 32 Baseline 10^{-4} $\epsilon = 0.15$ embed TOKENS
BS-MML	Beam size 128 embed TOKENS	Beam size 128 embed TOKENS	Beam size 128 embed TOKENS
RANDOMER	$\beta = 1$ $\epsilon = 0.05$ embed TOKENS	$\beta = 1$ $\epsilon = 0.15$ embed TOKENS	$\beta = 0$ $\epsilon = 0.15$ embed STACK

B SCONE domains and program tokens

token	**type**	**semantics**
Shared across ALCHEMY, TANGRAMS, SCENE		
`1, 2, 3, ...` `-1, -2, -3, ...`	constant	**push:** number
`red`, `yellow`, `green`, `orange`, `purple`, `brown`	constant	**push:** color
`allObjects`	constant	**push:** the list of all objects
`index`	function	**pop:** a list L and a number i **push:** the object $L[i]$ (the index starts from 1; negative indices are allowed)
`prevArg`j ($j = 1, 2$)	function	**pop:** a number i **push:** the j argument from the ith action
`prevAction`	action	**pop:** a number i **perform:** fetch the ith action and execute it using the arguments on the stack
Additional tokens for the ALCHEMY domain An ALCHEMY world contains 7 beakers. Each beaker may contain up to 4 units of colored chemical.		
`1/1`	constant	**push:** fraction (used in the `drain` action)
`hasColor`	function	**pop:** a color c **push:** list of beakers with chemical color c
`drain`	action	**pop:** a beaker b and a number or fraction a **perform:** remove a units of chemical (or all chemical if $a = $ `1/1`) from b
`pour`	action	**pop:** two beakers b_1 and b_2 **perform:** transfer all chemical from b_1 to b_2
`mix`	action	**pop:** a beaker b **perform:** turn the color of the chemical in b to `brown`
Additional tokens for the TANGRAMS domain A TANGRAMS world contains a row of tangram pieces with different shapes. The shapes are anonymized; a tangram can be referred to by an index or a history reference, but not by shape.		
`swap`	action	**pop:** two tangrams t_1 and t_2 **perform:** exchange the positions of t_1 and t_2
`remove`	action	**pop:** a tangram t **perform:** remove t from the stage
`add`	action	**pop:** a number i and a previously removed tangram t **perform:** insert t to position i
Additional tokens for the SCENE domain A SCENE world is a linear stage with 10 positions. Each position may be occupied by a person with a colored shirt and optionally a colored hat. There are usually 1-5 people on the stage.		
`noHat`	constant	**push:** pseudo-color (indicating that the person is not wearing a hat)
`hasShirt`, `hasHat`	function	**pop:** a color c **push:** the list of all people with shirt or hat color c
`hasShirtHat`	function	**pop:** two colors c_1 and c_2 **push:** the list of all people with shirt color c_1 and hat color c_2
`leftOf`, `rightOf`	function	**pop:** a person p **push:** the location index left or right of p
`create`	action	**pop:** a number i and two colors c_1, c_2 **perform:** add a new person at position i with shirt color c_1 and hat color c_2
`move`	action	**pop:** a person p and a number i **perform:** move p to position i
`swapHats`	action	**pop:** two people p_1 and p_2 **perform:** have p_1 and p_2 exchange their hats
`leave`	action	**pop:** a person p **perform:** remove p from the stage