Appendix

Table of Contents

List of Figures **20**

List of Tables **20**

A Program Embedding Space Visualizations **21**

B Cross Entropy Method Trajectory Visualization **22**

C Program Embedding Space Interpolations **25**

D Program Evolution **26**

E Interpretability: Human Debugging of LEAPS Programs **27**

F Optimal and Synthesized Programs **31**
F.1 Program Behavior Reconstruction 34
F.2 Karel Environment Tasks 35

G Additional Generalization Experiments **35**
G.1 Generalization on FOURCORNER, TOPOFF, and HARVESTER 35
G.2 Generalization to Unseen Configurations 36

H Additional Analysis on Experimental Results **37**
H.1 DRL vs. DRL-abs 37
H.2 VIPER generalization 37

I Detailed Descriptions and Illustrations of Ablations and Baselines **37**
I.1 Ablations 38
I.2 Baselines 38

J Program Dataset Generation Details **41**

K Karel Task Details **41**
K.1 STAIRCLIMBER 43
K.2 FOURCORNER 44
K.3 TOPOFF 44
K.4 MAZE 44
K.5 CLEANHOUSE 44
K.6 HARVESTER 44

L Hyperparameters and Training Details **44**
L.1 DRL and DRL-abs 44
L.2 DRL-abs-t 45
L.3 HRL 45
L.4 Naïve 46
L.5 VIPER 47
L.6 Program Embedding Space VAE Model 48
L.7 Cross-Entropy Method (CEM) 49
L.8 Random Search LEAPS Ablation 50

M Computational Resources **51**

List of Figures

4 Visualizations of Learned Program Embedding Space 22
5 StairClimber CEM Trajectory Visualization 23
6 FourCorner CEM Trajectory Visualization 24
7 Human Debugging Experiment User Interface 27
8 Human Debugging Experiment Example Programs 29
9 Ground-Truth Test Programs and Karel Programs 31
10 Program Reconstruction Task Synthesized Programs 33
11 LEAPS Karel Tasks Synthesized Programs 34
12 LEAPS Ablations Illustrations 39
13 Baseline Methods Illustrations 40
14 Program Length Histograms 41
15 Karel Task Start/End State Depictions 43
16 Karel Rollout Visualizations 53

List of Tables

5 LEAPS Close Latent Program Interpolation 25
6 LEAPS Far Latent Program Interpolation 25
7 Program Evolution Over CEM Search 26
8 Human Debugging Experiment Results 27
10 Unseen Configurations Performance 36
11 Program Token Generation Probabilities 41
12 LEAPS Length 100 Synthesized Karel Programs 42

A Program Embedding Space Visualizations

In this section, we present and analyze visualizations providing insights on the program embedding spaces learned by LEAPS and its variations. To investigate the learned program embedding space, we perform dimensionality reduction with PCA [100] to embed the following data to a 2D space for visualizations shown in Figure 4:

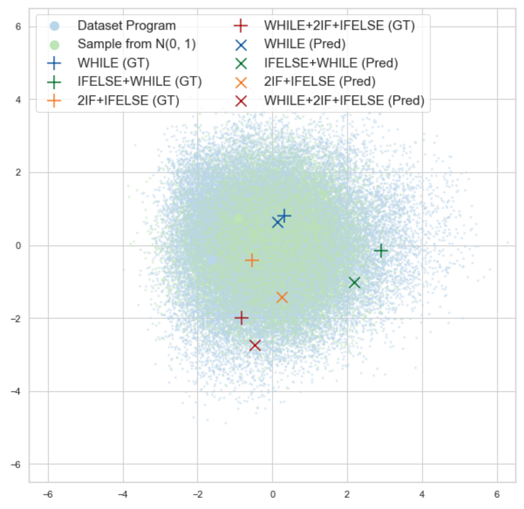
- Latent programs from the training dataset encoded by a learned encoder q_ϕ, visualized as blue scatters. There are 35k training programs.
- Samples drawn from a normal distribution $\mathcal{N}(0, 1)$, visualized as green scatters. This is to show how a distribution would look like if the embedding space is learned by using a highly weighted KL-divergence penalty (*i.e.* a large β value the VAE loss). We compared this against the latent program distribution learned by our method to justify the effectiveness of the proposed objectives: the program behavior reconstruction loss ($\mathcal{L}^{\mathrm{R}}$) and the latent behavior reconstruction loss ($\mathcal{L}^{\mathrm{L}}$).
- Ground-truth (GT) test programs from the testing dataset, encoded by a learned decoder q_ϕ, visualized as plus signs ($+$) with different colors. We selected 4 test programs.
- Reconstructed programs which are predicted (Pred) by each method given visualized as crosses ($\times$) with different colors. Since there are 4 test programs selected, 4 reconstructed programs are visualized. Each pair of test program and predicted program is visualized with the same color. These predicted (*i.e.* synthesized) programs are also shown in Figure 10.

Embedding Space Coverage. Even though the testing programs are not in the training program dataset, and therefore are unseen to models, their embedding vectors still lie in the distribution learned by all the models. This indicates that the learned embedding spaces cover a wide distribution of programs.

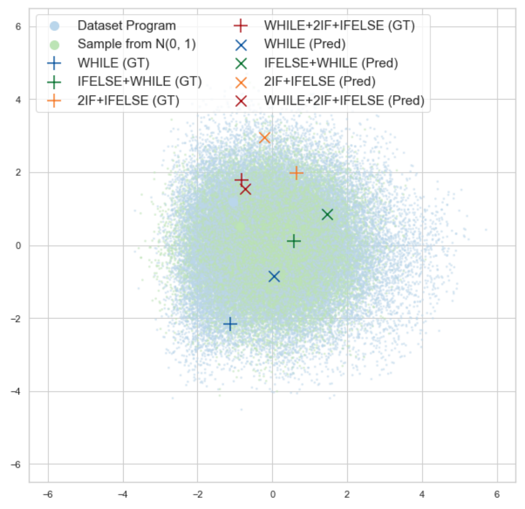
Latent Program Distribution vs. Normal Distribution. We now compare two distributions: the latent program distribution formed by encoding all the training programs to the program embedding space and a normal distribution $\mathcal{N}(0, 1)$. One can view the normal distribution as the distribution obtained by heavily enforcing the weight of the KL-divergence term when training a VAE model. We discuss the shape of the latent program distribution in the learned program embedding space as follows:

- LEAPS-P: since LEAPS+P simply optimizes the β-VAE loss (the program reconstruction loss $\mathcal{L}^{\mathrm{P}}$), which puts a lot of emphasis on the KL-divergence term, the shape of the latent program distribution is very similar to a normal distribution as shown in Figure 4 (a).
- LEAPS-P+R: while LEAPS+P+R additionally optimizes the program behavior reconstruction loss $\mathcal{L}^{\mathrm{R}}$, the shape of the latent program distribution is still similar to a normal distribution, as shown in Figure 4 (b). We hypothesize that it is because the program behavior reconstruction loss alone might not be strong or explicit enough to introduce a change.
- LEAPS-P+L: the shape of the latent program distribution in the program embedding space learned by LEAPS+P+L is significantly different from a normal distribution, as shown in Figure 4 (c). This suggest that employing the latent behavior reconstruction loss $\mathcal{L}^{\mathrm{L}}$ dramatically contributes to the learning. We believe it is because the latent behavior reconstruction loss is optimized with direct gradients and therefore provides a stronger learning signal especially compared to the program behavior reconstruction loss $\mathcal{L}^{\mathrm{R}}$, which is optimized using REINFORCE [64].
- LEAPS (LEAPS-P+R+L): LEAPS optimizes the full objective that includes all three proposed objectives and form a similar distribution shape as the one learned by LEAPS+P+L. However, the distance between each pair of the ground-truth testing program and the predicted program is much closer in the program embedding space learned by LEAPS compared to the space learned by LEAPS+P+L. This justifies the effectiveness of the proposed program behavior reconstruction loss $\mathcal{L}^{\mathrm{R}}$, which can bring the programs with similar behaviors closer in the embedding space.

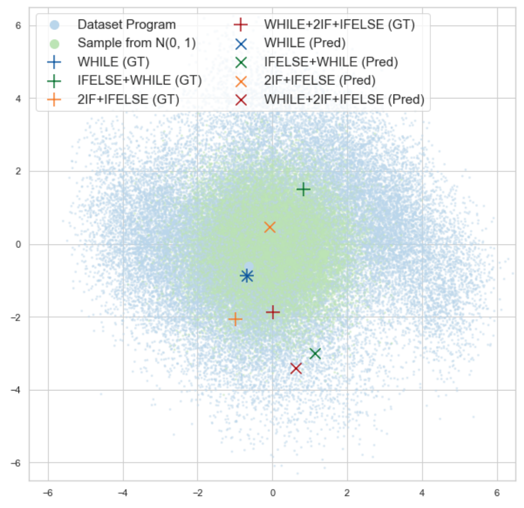
Summary. The visualizations of the program embedding spaces learned by LEAPS and its ablations qualitatively justify the effectiveness of the proposed learning objectives, as complementary to the quantitative results presented in the main paper.



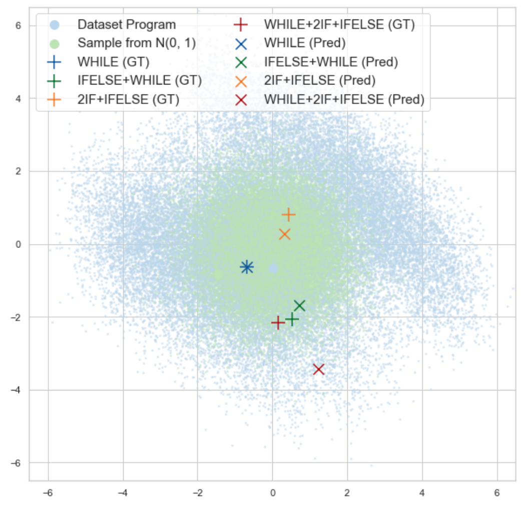
(a) LEAPS-P



(b) LEAPS-P+R



(c) LEAPS-P+L



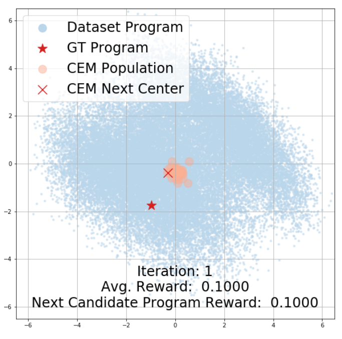
(d) LEAPS

Figure 4: **Visualizations of learned program embedding space.** We perform dimensionality reduction with PCA to embed encoded programs from the training dataset, samples drawn from a normal distribution, programs from the testing dataset, and programs reconstructed by models to a 2D space. The shape of the latent training programs in the program embedding spaces learned by LEAPS-P and LEAPS-P+R are similar to a normal distribution, while in the program embedding spaces learned by LEAPS and LEAPS-P+L, the shape is more twisted, suggesting the effectiveness of the proposed latent behavior reconstruction objective. Moreover, the distances between pairs of ground-truth programs and their reconstructions are smaller in the program embedding space learned by LEAPS, highlighting the advantage of employing both of the two proposed behavior reconstruction objectives.

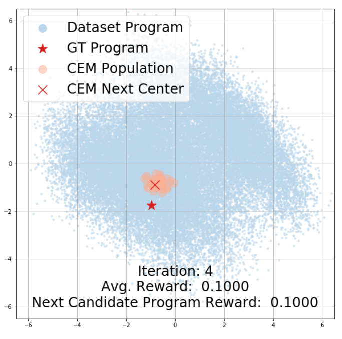
B Cross Entropy Method Trajectory Visualization

As described in the main paper, once the program embedding space is learned by LEAPS, our goal becomes searching for a latent program that maximizes the reward described by a given task MDP. To this end, we adapt the Cross Entropy Method (CEM) [65], a gradient-free continuous search algorithm, to iteratively search over the program embedding space. Specifically, we iteratively perform the following steps:

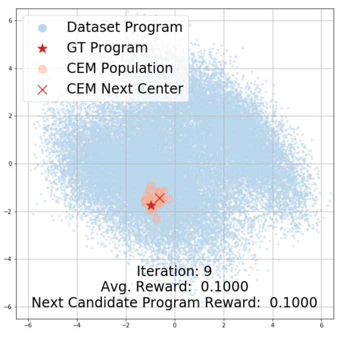
1. Sample a distribution of candidate latent programs.



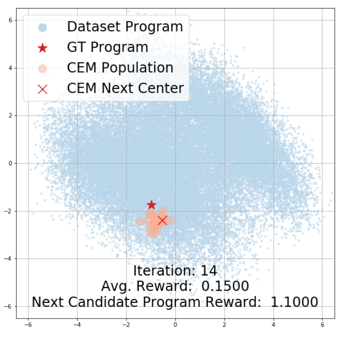
(a) Iteration 1



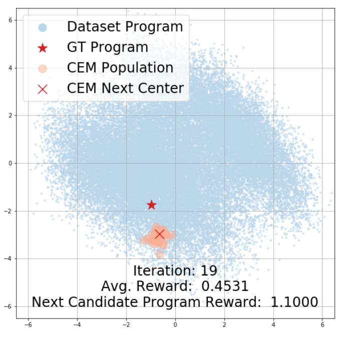
(b) Iteration 4



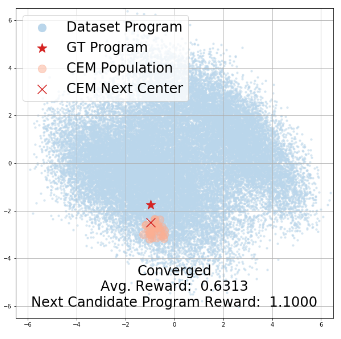
(c) Iteration 9



(d) Iteration 14



(e) Iteration 19



(f) Iteration 23

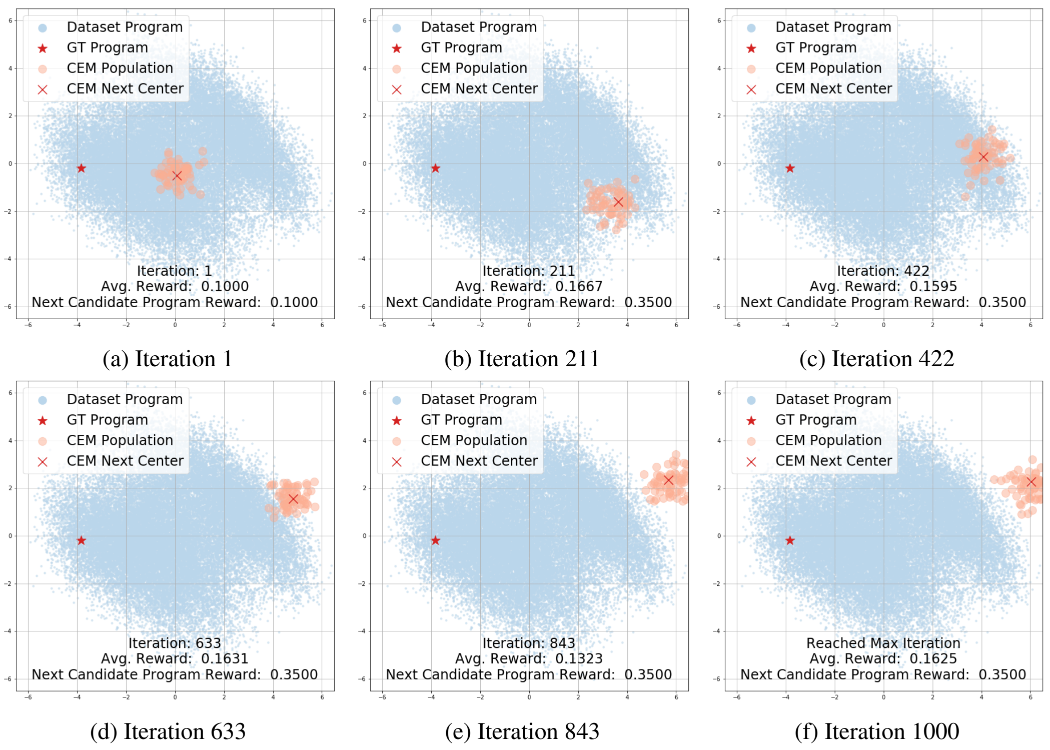
Figure 5: **STAIRCLIMBER CEM Trajectory Visualization.** Latent training programs from the training dataset, a ground-truth program for STAIRCLIMBER task, CEM populations, and CEM next candidate programs are embedded to a 2D space using PCA. Both the average reward of the entire population and the reward of the next candidate program (CEM Next Center) consistently increase as the number of iterations increase. Also, the CEM population gradually moves toward where the ground-truth program is located.

2. Decode the sampled latent programs into programs using the learned program decoder p_θ.
3. Execute the programs in the task environment and obtain the corresponding rewards.
4. Update the CEM sampling distribution based on the rewards.

This process is repeated until either convergence or the maximum number of sampling steps has been reached.

We perform dimensionality reduction with PCA [100] to embed the following data to a 2D space; the visualizations of CEM trajectories are shown in Figure 5 and Figure 6:

- Latent programs from the training dataset encoded by a learned encoder q_ϕ, visualized as blue scatters. There are 35k training programs. This is to visualize the shape of the program distribution in the learned program embedding space. This is also visualized in Figure 4.
- Ground-truth (GT) programs that exhibit optimal behaviors for solving the Karel tasks, visualized as red stars ($\star$). Ideally, the CEM population should iteratively move toward where the GT programs are located.
- CEM population is a batch of sampled candidate latent programs at each iteration, visualized as red scatters. Each candidate latent program can be decoded as a program that can be executed in the task environment to obtain a reward. By averaging the reward obtained by every candidate latent program, we can calculate the average reward of this population and show it in the figures as Avg. Reward.
- CEM Next Center, visualized as cross signs ($\times$), indicates the center vector around which the next batch of candidate latent programs will be sampled. This vector is calculated based on a set of candidate latent programs that achieve best reward (*i.e.* elite samples) at each iteration. In this case, it is a weighted average based on the reward each candidate gets from its execution.



(a) Iteration 1 (b) Iteration 211 (c) Iteration 422

(d) Iteration 633 (e) Iteration 843 (f) Iteration 1000

Figure 6: **FOURCORNER CEM Trajectory Visualization.** Latent training programs from the training dataset, a ground-truth program for the FOURCORNER task, CEM populations, and CEM next candidate programs are embedded to a 2D space using PCA. The CEM trajectory does not converge. The ground-truth program lies far away from the initial sampled distribution, which might contribute to the difficulty of converging.

From Figure 5, we observe that both the average reward of the entire population and the reward of the next candidate program (CEM Next Center) consistently increase as the number of iterations increases, justifying the effectiveness of CEM. Moreover, we observe that the CEM population gradually moves toward where the ground-truth program is located, which aligns well with the fact that our proposed framework can reliably synthesize task-solving programs.

Yet, the populations might not always exactly converge to where the ground-truth latent program is. We hypothesize this could be attributed to the following reasons:

1. CEM convergence: while the CEM search converges, it can still be suboptimal. Since the search terminates when the next candidate latent program obtains the maximum reward (1.1 as shown in the figure) for 10 iterations, it might not exactly converge to where a ground-truth program is.
2. Dimensionality reduction: we visualized the trajectories and programs by performing dimensionality reduction from 256 to 2 dimensions with PCA, which could cause visual distortions.
3. Suboptimal learned program embedding space: while we aim to learn a program embedding space where all the programs inducing the same behaviors are mapped to the same spot in the embedding space, it is still possible that programs that induce the desired behavior can distribute to more than one location in a learned program embedding space. Therefore, CEM search can converge to somewhere that is different from the ground-truth latent program.

On the other hand, the CEM trajectory shown in Figure 6 does not converge and terminates when reaching the maximum number of iterations. The ground-truth program lies far away from the initial sampled distribution, which might contribute to the difficulty of converging. This aligns with the relatively unsatisfactory performance achieved by LEAPS. Employing a more sophisticated searching

Table 5: Decoded linear interpolations of programs close to each other in the latent space.

Latent Program	Decoded Program
START	`DEF run m( turnRight move WHILE c( frontIsClear c) w( move w) WHILE c( not c( frontIsClear c) c) w( move w) IF c( frontIsClear c) i( move i) m)`
1	`DEF run m( turnRight move WHILE c( frontIsClear c) w( move w) WHILE c( not c( frontIsClear c) c) w( move w) IF c( frontIsClear c) i( move i) m)`
2	`DEF run m( turnRight move WHILE c( frontIsClear c) w( move w) IF c( not c( frontIsClear c) c) i( move i) m)`
3	`DEF run m( turnRight move WHILE c( frontIsClear c) w( move w) IF c( not c( frontIsClear c) c) i( move i) m)`
4	`DEF run m( turnRight move WHILE c( frontIsClear c) w( move w) IF c( not c( frontIsClear c) c) i( move i) m)`
5	`DEF run m( turnRight move WHILE c( frontIsClear c) w( move w) IF c( not c( frontIsClear c) c) i( move i) m)`
6	`DEF run m( turnRight move WHILE c( frontIsClear c) w( move w) IF c( not c( frontIsClear c) c) i( move i) m)`
7	`DEF run m( turnRight move turnLeft WHILE c( frontIsClear c) w( move w) IF c( not c( frontIsClear c) c) i( putMarker i) m)`
8	`DEF run m( turnRight move turnLeft WHILE c( frontIsClear c) w( move w) IF c( not c( frontIsClear c) c) i( putMarker i) m)`
END	`DEF run m( turnRight move turnLeft WHILE c( frontIsClear c) w( move w) IF c( not c( frontIsClear c) c) i( putMarker i) m)`

Table 6: Decoded linear interpolations of programs far from each other in the latent space.

Latent Program	Decoded Program
START	`DEF run m( turnRight turnLeft turnLeft move turnRight putMarker move m)`
1	`DEF run m( turnRight turnLeft turnLeft move turnRight putMarker move m)`
2	`DEF run m( turnRight turnLeft turnLeft move WHILE c( frontIsClear c) w( putMarker w) turnRight move m)`
3	`DEF run m( turnRight turnLeft move turnLeft WHILE c( frontIsClear c) w( putMarker w) move m)`
4	`DEF run m( turnRight turnLeft move WHILE c( frontIsClear c) w( turnLeft w) IF c( not c( frontIsClear c) c) i( move i) m)`
5	`DEF run m( turnRight move turnLeft WHILE c( frontIsClear c) w( move w) IF c( not c( frontIsClear c) c) i( putMarker i) m)`
6	`DEF run m( move turnRight turnLeft move WHILE c( frontIsClear c) w( IF c( not c( rightIsClear c) c) i( putMarker i) w) m)`
7	`DEF run m( move turnRight turnLeft move WHILE c( frontIsClear c) w( IF c( not c( rightIsClear c) c) i( turnLeft i) w) m)`
8	`DEF run m( move turnRight move WHILE c( frontIsClear c) w( IF c( not c( rightIsClear c) c) i( turnLeft i) w) m)`
END	`DEF run m( move turnRight move WHILE c( frontIsClear c) w( IF c( not c( rightIsClear c) c) i( turnLeft i) w) m)`

algorithm or conducting a more thorough hyperparameter search could potentially improve the performance but it is not the main focus of this work.

C Program Embedding Space Interpolations

To learn a program embedding space that allows for smooth interpolation, we propose three sources of supervision. We aim to verify the effectiveness of it by investigating interpolations in the learned program embedding space. To this end, we follow the procedure described below to produce results shown in Table 5 and Table 6.

1. Sampling a pair of programs from the dataset (`START` program and `END` program).
2. Encoding the two programs into the learned program embedding space.
3. Linearly interpolating between the two latent programs to obtain a number of (eight) interpolated latent programs.
4. Decoding the latent programs to obtain interpolated programs (program `1` to program `8`).

Table 7: How predicted programs evolve throughout the course of CEM search for STAIRCLIMBER. See Figure 5 for the corresponding visualization of this CEM search.

Search Iteration	Best Predicted Program
Iteration: 1	DEF run m(IF c(frontIsClear c) i(pickMarker i) WHILE c(leftIsClear c) w(move w) IFELSE c(frontIsClear c) i(turnRight move i) ELSE e(move e) m)
Iteration: 2	DEF run m(WHILE c(markersPresent c) w(move w) IFELSE c(frontIsClear c) i(turnLeft i) ELSE e(move e) WHILE c(leftIsClear c) w(move w) m)
Iteration: 3	DEF run m(WHILE c(not c(frontIsClear c) c) w(move turnRight w) WHILE c(leftIsClear c) w(turnLeft move w) m)
Iteration: 4	DEF run m(WHILE c(not c(frontIsClear c) c) w(pickMarker move w) WHILE c(leftIsClear c) w(turnLeft move w) m)
Iteration: 5	DEF run m(WHILE c(not c(frontIsClear c) c) w(pickMarker turnRight w) WHILE c(leftIsClear c) w(move turnLeft w) m)
Iteration: 6	DEF run m(WHILE c(not c(frontIsClear c) c) w(pickMarker turnRight w) WHILE c(leftIsClear c) w(move turnLeft w) m)
Iteration: 7	DEF run m(WHILE c(not c(leftIsClear c) c) w(turnRight w) IFELSE c(frontIsClear c) i(move i) ELSE e(turnLeft e) WHILE c(rightIsClear c) w(move w) m)
Iteration: 8	DEF run m(WHILE c(not c(leftIsClear c) c) w(turnRight move w) WHILE c(markersPresent c) w(turnLeft move w) m)
Iteration: 9	DEF run m(WHILE c(not c(noMarkersPresent c) c) w(turnRight move w) WHILE c(not c(frontIsClear c) c) w(turnLeft move w) m)
Iteration: 10	DEF run m(WHILE c(not c(noMarkersPresent c) c) w(turnRight move w) WHILE c(leftIsClear c) w(turnLeft move w) m)
Iteration: 11	DEF run m(WHILE c(not c(leftIsClear c) c) w(turnRight move w) WHILE c(noMarkersPresent c) w(turnLeft move w) m)
Iteration: 12	DEF run m(WHILE c(not c(leftIsClear c) c) w(turnRight move w) WHILE c(noMarkersPresent c) w(turnLeft move w) m)
Iteration: 13	DEF run m(WHILE c(not c(leftIsClear c) c) w(turnRight move w) WHILE c(noMarkersPresent c) w(turnLeft move w) m)
Iteration: 14	DEF run m(WHILE c(not c(markersPresent c) c) w(turnRight move w) WHILE c(rightIsClear c) w(move turnLeft w) m)
Iteration: 15	DEF run m(WHILE c(not c(markersPresent c) c) w(turnRight move w) WHILE c(rightIsClear c) w(move turnLeft w) m)
Iteration: 16	DEF run m(WHILE c(not c(markersPresent c) c) w(turnRight move w) WHILE c(rightIsClear c) w(move turnLeft w) m)
Iteration: 17	DEF run m(WHILE c(not c(markersPresent c) c) w(turnRight move w) WHILE c(rightIsClear c) w(move turnLeft w) m)
Iteration: 18	DEF run m(WHILE c(not c(markersPresent c) c) w(turnRight move w) WHILE c(rightIsClear c) w(move turnLeft w) m)
Iteration: 19	DEF run m(WHILE c(not c(markersPresent c) c) w(turnRight move w) WHILE c(rightIsClear c) w(move turnLeft w) m)
Iteration: 20	DEF run m(WHILE c(not c(markersPresent c) c) w(turnRight move w) WHILE c(rightIsClear c) w(move turnLeft w) m)
Iteration: 21	DEF run m(WHILE c(not c(markersPresent c) c) w(turnRight move w) WHILE c(rightIsClear c) w(move turnLeft w) m)
Iteration: 22	DEF run m(WHILE c(not c(markersPresent c) c) w(turnRight move w) WHILE c(rightIsClear c) w(move turnLeft w) m)
Converged	DEF run m(WHILE c(not c(markersPresent c) c) w(turnRight move w) WHILE c(rightIsClear c) w(turnLeft move w) m)

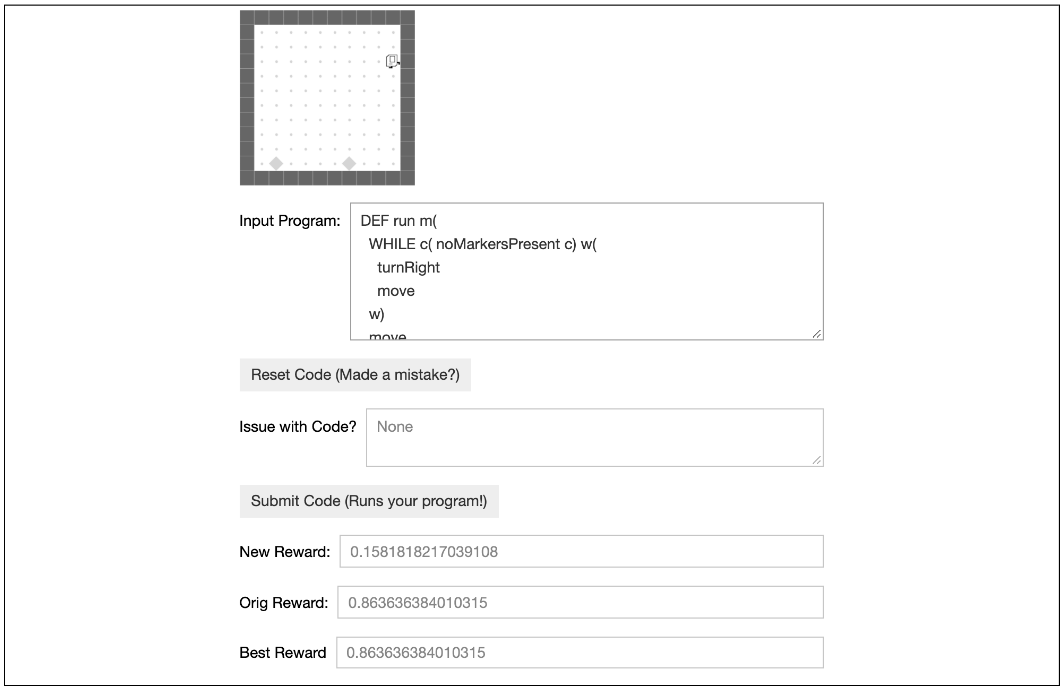
We show two pairs of programs and their interpolations in between below as examples. Specifically, the first pair of programs, shown in Table 5, are closer to each other in the latent space and the second pair of programs, shown in Table 6, are further from each other. We observe that the interpolations between the closer program pair exhibit smoother transitions and the interpolations between the further program pair display more dramatic change.

D Program Evolution

In this section, we aim to investigate how predicted programs evolve over the course of searching. We visualize converged CEM search trajectories and the reward each program gets on the StairClimber task in Appendix Figure 5. In Table 7, we present the predicted programs corresponding to the CEM search trajectory on the STAIRCLIMBER task in Figure 5. We observe that the sampled programs

Table 8: Mean return (standard deviation) [% increase in performance] after debugging by non-expert humans of LEAPS synthesized programs for 3 statement edits and 5 statement edits. Chosen LEAPS programs are median-reward programs out of 5 LEAPS seeds for each task.

Karel Task	Original Program	3 Edits	5 Edits
TOPOFF	0.86	0.95 (0.07) [10.5%]	1.0 (0.00) [16.3%]
FOURCORNER	0.25	0.75 (0.35) [200%]	0.92 (0.12) (268%)
HARVESTER	0.47	0.85 (0.05) [80.9%]	0.89 (0.00) [89.4%]
Average % Increase	-	97.1%	125%



```
DEF run m(
  WHILE c( noMarkersPresent c) w(
    turnRight
    move
  w)
  move
```

Issue with Code? None

New Reward: 0.1581818217039108

Orig Reward: 0.863636384010315

Best Reward 0.863636384010315

Figure 7: **User Interface for the Human Debugging Interpretability Experiments.** The top contains moving rollout visualizations of the current program in the "Input Program" box, which users are allowed to edit. "Input Program" will first contain the program synthesized by LEAPS. Syntax errors or other issues with code (such as the edit distance being too high) are displayed in the "Issue with Code?" box, the reward of the current inputted program is in the "New Reward" box, and the reward of the original program synthesized by LEAPS is in the "Orig Reward" box. The user's best reward across all inputted programs is kept track of in the "Best Reward" box.

consistently improve as the number of iterations increases, justifying the effectiveness of the learned program embedding and the CEM search.

E Interpretability: Human Debugging of LEAPS Programs

Interpretability in Machine Learning is crucial for several reasons [69, 70]. First, *trust* – interpretable machine learning methods and models may more easily be trusted since humans tend to be reluctant to trust systems that they do not understand. Second, interpretability can improve the *safety* of machine learning systems. A machine learning system that is interpretable allows for diagnosing issues (*e.g*. the distribution shift from training data to testing data) earlier and provides more opportunities to intervene. This is especially important for safety-critical tasks such as medical diagnosis [101–105] and real-world robotics [5–11] tasks. Finally, interpretability can lead to *contestability*, by producing a chain of reasoning, providing insights on how a decision is made and therefore allowing humans to contest unfair or improper decisions.

TopOff

LEAPS (Reward=0.86)

```
DEF run m(
  WHILE c( noMarkersPresent c) w(
    turnRight
    move
  w)
  putMarker
  move
  WHILE c( noMarkersPresent c) w(
    turnRight
    move
  w)
  putMarker
  move
  WHILE c( noMarkersPresent c) w(
    turnRight
    move
  w)
  putMarker
  move
  WHILE c( noMarkersPresent c) w(
    turnRight
    move
  w)
  putMarker
  move
m)
```

3 Edits (Reward=1.0)

```
DEF run m(
  REPEAT R=9 r(
  WHILE c( noMarkersPresent c) w(
    IF c( frontIsClear c) i( move
          i)
  w)
  putMarker
  move
  r)
  WHILE c( noMarkersPresent c) w(
    turnRight
    move
  w)
  putMarker
  move
  WHILE c( noMarkersPresent c) w(
    turnRight
    move
  w)
  putMarker
  move
  WHILE c( noMarkersPresent c) w(
    turnRight
    move
  w)
  putMarker
  move
m)
```

5 Edits (Reward=1.0)

```
DEF run m(
  WHILE c( frontIsClear c) w(
    IF c( markersPresent c) i(
      putMarker
    i)
    move
  w)
  WHILE c( frontIsClear c) w(
  WHILE c( noMarkersPresent c) w(
    turnRight
    move
  w)
  putMarker
  move
  WHILE c( noMarkersPresent c) w(
    turnRight
    move
  w)
  putMarker
  move
  WHILE c( noMarkersPresent c) w(
    turnRight
    move
  w)
  putMarker
  move
  WHILE c( noMarkersPresent c) w(
    turnRight
    move
  w)
  putMarker
  move
  w)
m)
```

FourCorner

LEAPS (Reward=0.25)

```
DEF run m(
  turnRight
  turnRight
  move
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  putMarker
  turnRight
  turnRight
m)
```

3 Edits (Reward=1.0)

```
DEF run m(
  turnRight
  turnRight
  move
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  REPEAT R=4 r(
  REPEAT R=9 r(
  move
  r)
  putMarker
  turnRight
  r)
  turnRight
m)
```

5 Edits (Reward=1.0)

```
DEF run m(
  turnRight
  turnRight
  move
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  turnRight
  putMarker
  REPEAT R=3 r(
  REPEAT R=9 r(
  move
  r)
  putMarker
  turnRight
  r)
m)
```

HARVESTER

LEAPS (REWARD=0.47)

```
DEF run m(
  turnLeft
  turnLeft
  pickMarker
  move
  pickMarker
  move
  pickMarker
  move
  pickMarker
  move
  pickMarker
  move
  turnLeft
  pickMarker
  move
  pickMarker
  move
  pickMarker
  move
  pickMarker
  move
  turnLeft
  pickMarker
  move
  pickMarker
  move
  pickMarker
  move
  turnLeft
  pickMarker
  move
  pickMarker
  move
  pickMarker
  move
  turnLeft
  pickMarker
  move
  pickMarker
  move
  pickMarker
  move
m)
```

3 EDITS (REWARD=0.77)

```
DEF run m(
  turnLeft
  turnLeft
  REPEAT R=4 r(
    pickMarker
    move
    pickMarker
    move
    pickMarker
    move
    pickMarker
    move
    pickMarker
    move
    pickMarker
    move
    turnLeft
  r)
  pickMarker
  move
  pickMarker
  move
  pickMarker
  move
  pickMarker
  move
  pickMarker
  move
  turnLeft
  pickMarker
  move
  pickMarker
  move
  pickMarker
  move
  turnLeft
  pickMarker
  move
  pickMarker
  move
  pickMarker
  move
  turnLeft
  pickMarker
  move
  pickMarker
  move
  pickMarker
  move
m)
```

5 EDITS (REWARD=0.89)

```
DEF run m(
  REPEAT R=3 r(
  pickMarker
  move
  pickMarker
  move
  pickMarker
  move
  pickMarker
  move
  pickMarker
  move
  turnLeft
  r)
  REPEAT R=3 r(
  pickMarker
  move
  pickMarker
  move
  pickMarker
  move
  pickMarker
  move
  turnLeft
  r)
  pickMarker
  move
  pickMarker
  move
  pickMarker
  move
  turnLeft
  pickMarker
  move
  pickMarker
  move
  pickMarker
  move
  turnLeft
  pickMarker
  move
  pickMarker
  move
  pickMarker
  move
m)
```

Figure 8: **Human Debugging Experiment Example Programs.** Example original and human-edited programs for each Karel task for edit distances 3 and 5.

We believe interpretability is especially crucial when it comes to learning a policy that interacts with the environment. In this work, we propose a framework that offers an effective way to acquire an interpretable programmatic policy structured in a program. In the following, we discuss how the proposed framework enjoys interpretability from the three aforementioned aspects. Programs synthesized by the proposed framework can naturally be better *trusted* since one can simply read and understand them. Also, through the program execution trace produced by executing a program, each decision made by the policy (*i.e.* the program) is traceable and therefore satisfies the *contestability* property. Finally, the programs produced by our framework satisfy the *safety* property of interpretability as humans can diagnose and correct for issues by reading and editing the programs.

Our synthesized programs are not only readable to human users but also interactable, allowing *non-expert* users with a basic understanding of programming to diagnose and make edits to improve their performance. To test this hypothesis, we asked people with programming experience who are unfamiliar with our DSL or Karel tasks to edit suboptimal LEAPS programs to improve performance as much as possible on 3 Karel tasks: TOPOFF, FOURCORNER, and HARVESTER through a user interface displayed in Figure 7. Each person was given 1.5 hours (30 minutes per program), including time required to understand what the LEAPS programs were doing, understand the DSL tokens, and fully debug/test their edited programs. For each program, participants were required to modify up to 5 statements, then attempt the task again with up to only 3 modifications as calculated by the Levenshtein distance metric [106]. A single statement modification is defined as any modification/removal/addition of a IF, WHILE, IFELSE, REPEAT, or ELSE statement, or a removal/addition/change of an action statement (*e.g.* move, turnLeft, etc.). Participants were allowed to ask clarification questions, but we would not answer questions regarding how to specifically improve the performance of their program.

We display example edited programs in Figure 8, and the aggregated results of editing in Table 8. We see a significant increase in performance in all three tasks, with an average 97.1% increase in performance with 3 edits and an average 125% increase in performance with 5. These numbers are averaged over 3 people, with standard deviations reported in the table. Thus we see that even slight modifications to suboptimal LEAPS programs can enable much better Karel task performance when edited by non-expert humans.

Our experiments in this section make an interesting connection to works in program/code repair (*i.e.* automatic bug fixing) [107–120], where the aim is to develop algorithms and models that can find bugs or even repair programs without the intervention of a human programmer. While the goal of these works is to fix programs produced by humans, our goal in this section is to allow humans to improve programs synthesized by the proposed framework.

Another important benefit of programmatic policies is *verifiability* - the ability to verify different properties of policies such as correctness, stability, smoothness, robustness, safety, etc. Since programmatic policies are highly structured, they are more amenable to formal verification methods developed for traditional software systems as compared to neural policies. Recent works [12, 14, 15, 121] show that various properties of programmatic policies (programs written using DSLs, decision trees) can be verified using existing verification algorithms, which can also be applied to programs synthesized by the proposed framework.

WHILE:

```
DEF run m(
    WHILE c( frontIsClear c) w(
        turnRight
        move
        pickMarker
        turnRight
        w)
    m)
```

IFELSE+WHILE:

```
DEF run m(
    IFELSE c( markersPresent c) i(
        move turnRight
        i) ELSE e(
            move
        e)
    move
    move
    WHILE c( leftIsClear c) w(
        turnLeft
        w)
    m)
```

2IF+IFELSE:

```
DEF run m(
    IF c( frontIsClear c) i(
        putMarker
    i)
    move
    IF c( rightIsClear c) i(
        move
    i)
    IFELSE c( frontIsClear c) i(
        move
        i) ELSE e(
            move
        e)
    m)
```

WHILE+2IF+IFELSE:

```
DEF run m(
    WHILE c( leftIsClear c) w(
        turnLeft
    w)
    IF c( frontIsClear c) i(
        putMarker move
    i)
    move
    IF c( rightIsClear c) i(
        turnRight
        move
    i)
    IFELSE c( frontIsClear c) i(
        move
        i) ELSE e(
            turnLeft move
        e)
    m)
```

STAIRCLIMBER:

```
DEF run m(
    WHILE c( noMarkersPresent c) w(
        turnLeft
        move
        turnRight
        move
        w)
    m)
```

TOPOFF:

```
DEF run m(
    WHILE c( frontIsClear c) w(
        IF c( markersPresent c) i(
            putMarker
            i)
        move
        w)
    m)
```

CLEANHOUSE:

```
DEF run m(
    WHILE c( noMarkersPresent c) w(
        IF c( leftIsClear c) i(
            turnLeft
            i)
        move
        IF c( markersPresent c) i(
            pickMarker
            i)
        w)
    m)
```

FOURCORNER:

```
DEF run m(
    WHILE c( noMarkersPresent c) w(
        WHILE c( frontIsClear c) w(
            move
            w)
        IF c( noMarkersPresent c) i
            (
            putMarker
            turnLeft
            move
            i)
        w)
    m)
```

MAZE:

```
DEF run m(
    WHILE c( noMarkersPresent c) w(
        IFELSE c( rightIsClear c) i
            (
            turnRight
            i) ELSE e(
                WHILE c( not c(
                    frontIsClear
                    c) c) w(
                    turnLeft
                    w)
            e)
            move
        w)
    m)
```

HARVESTER:

```
DEF run m(
    WHILE c( markersPresent c) w(
        WHILE c( markersPresent c) w(
            pickMarker
            move
            w)
        turnRight
        move
        turnLeft
        WHILE c( markersPresent c) w(
            pickMarker
            move
            w)
        turnLeft
        move
        turnRight
        w)
    m)
```

Figure 9: **Ground-Truth Test and Karel Programs.** Here we display ground-truth test set programs used for reconstruction experiments and example ground-truth programs that we write which can solve the Karel tasks (there are an infinite number of programs that can solve each task). Conditionals are enclosed in `c( c)`, while loops are enclosed in `w( w)`, if statements are enclosed in `i( i)`, and the main program is enclosed in `DEF run m( m)`.

F Optimal and Synthesized Programs

In this section, we present the programs from the testing set which are selected for conducting ablation studies in the main paper in Figure 9. Also, we manually write programs that induce optimal behaviors to solve the Karel tasks and present them in Figure 9. Note that while we only show

Naïve

WHILE

```
DEF run m(
    WHILE c(frontIsClear c) w(
        turnRight
        move
        pickMarker
        turnRight
        w)
    m)
```

IFELSE+WHILE

```
DEF run m(
    move
    move
    move
    turnLeft
    turnLeft
    m)
```

2IF+IFELSE

```
DEF run m(
    putMarker
    move
    move
    move
    m)
```

WHILE+2IF+IFELSE

```
DEF run m(
    turnLeft
    putMarker
    move
    move
    WHILE c( markersPresent c) w(
        pickMarker
        pickMarker
        pickMarker
        w)
    m)
```

LEAPS-P

WHILE

```
DEF run m(
    IF c( frontIsClear c) i(
        turnRight
        move
        pickMarker
        turnRight
        i)
    m)
```

IFELSE+WHILE

```
DEF run m(
    IFELSE c( rightIsClear c) i(
        move
        i) ELSE e(
        move
        e)
    move
    move
    IF c( leftIsClear c) i(
        turnLeft
        i)
    m)
```

2IF+IFELSE

```
DEF run m(
    IFELSE c( not c( frontIsClear c) c) i(
        move
        i) ELSE e(
        putMarker
        move
        e)
    move
    move
    m)
```

WHILE+2IF+IFELSE

```
DEF run m(
    WHILE c( leftIsClear c) w(
        turnLeft
        w)
    putMarker
    move
    move
    turnRight
    move
    move
    m)
```

LEAPS-P+R

WHILE

```
DEF run m(
    WHILE c( rightIsClear c) w(
        WHILE c( frontIsClear c) w(
            turnRight
            move
            pickMarker
            turnRight
            w)
        w)
    m)
```

IFELSE+WHILE

```
DEF run m(
    REPEAT R=1 r(
        move
        r)
    REPEAT R=2 r(
        move
        r)
    m)
```

2IF+IFELSE

```
DEF run m(
    IFELSE c( not c( frontIsClear c) c) i(
        move
        i) ELSE e(
        putMarker
        e)
    IFELSE c( rightIsClear c) i(
        move
        i) ELSE e(
        move
        e)
    IF c( rightIsClear c) i(
        move
        i)
    move
    m)
```

WHILE+2IF+IFELSE

```
DEF run m(
    WHILE c( leftIsClear c) w(
        turnLeft
        w)
    putMarker
    move
    move
    turnRight
    move
    move
    m)
```

LEAPS-P+L

WHILE

```
DEF run m(
    WHILE c( frontIsClear c) w(
        turnRight
        move
        pickMarker
        turnRight
        w)
    m)
```

IFELSE+WHILE

```
DEF run m(
    move
    move
    move
    WHILE c( leftIsClear c) w(
        turnLeft
        w)
    m)
```

2IF+IFELSE

```
DEF run m(
    IFELSE c( frontIsClear c) i(
        REPEAT R=0 r(
            turnRight
            r)
        putMarker
        move
        i) ELSE e(
        move
        e)
    move
    move
    m)
```

WHILE+2IF+IFELSE

```
DEF run m(
    WHILE c( leftIsClear c) w(
        turnLeft
        w)
    WHILE c( leftIsClear c) w(
        turnLeft
        w)
    WHILE c( leftIsClear c) w(
        turnLeft
        w)
    WHILE c( leftIsClear c) w(
        turnLeft
        w)
    IF c( frontIsClear c) i(
        putMarker
        move
        i)
    move
    move
    m)
```

LEAPS

WHILE

```
DEF run m(
    WHILE c( frontIsClear c) w(
        turnRight
        move
        pickMarker
        turnRight
        w)
    m)
```

IFELSE+WHILE

```
DEF run m(
    IFELSE c( not c( noMarkersPresent c) c) i(
        move
        turnRight
        i) ELSE e(
        move
        e)
    REPEAT R=2 r(
        move
        r)
    WHILE c( leftIsClear c) w(
        turnLeft
        w)
    m)
```

2IF+IFELSE

```
DEF run m(
    IFELSE c( frontIsClear c) i(
        putMarker
        move
        i) ELSE e(
        move
        e)
    IF c( rightIsClear c) i(
        move
        i)
    move
    m)
```

WHILE+2IF+IFELSE

```
DEF run m(
    WHILE c( leftIsClear c) w(
        turnLeft
        w)
    IF c( frontIsClear c) i(
        putMarker
        move
        i)
    move
    move
    m)
```

Figure 10: **Example program reconstruction task programs generated by all methods.** The programs that achieve the highest reward while being representative of programs generated by most seeds are shown. The naïve program synthesis baseline usually generates the simplest programs, with fewer conditional statements and loops than the LEAPS ablations. Notably, it fails to generate IFELSE statements on these examples, while LEAPS has no problem doing so.

one optimal program for each task, there exist multiple programs that exhibit the desired behaviors for each task. Then, we analyze the program reconstructed by LEAPS, its ablations, and the naïve program synthesis baseline in Section F.1, and discuss the programs synthesized by LEAPS for Karel tasks in Section F.2.

LEAPS Karel Programs

STAIRCLIMBER

```
DEF run m(
    WHILE c( noMarkersPresent c)
         w(
        turnRight
        move
        w)
    WHILE c( rightIsClear c) w(
        turnLeft
        w)
    m)
```

TOPOFF

```
DEF run m(
    WHILE c( noMarkersPresent c)
         w(
        move
        w)
    putMarker
    move
    WHILE c( not c(
         markersPresent c) c) w(
        move w)
    putMarker
    move
    WHILE c( not c(
         markersPresent c) c) w(
        move
        w)
    putMarker
    move
    turnRight
    turnRight
    turnRight
    turnRight
    turnRight
    turnRight
    turnRight
    turnRight
    m)
```

CLEANHOUSE

```
DEF run m(
    WHILE c( noMarkersPresent c)
         w(
        turnRight
        move
        move
        turnLeft
        turnRight
        pickMarker
        w)
    turnLeft
    turnRight
    m)
```

FOURCORNER

```
DEF run m(
    turnRight
    move
    turnRight
    turnRight
    turnRight
    WHILE c( frontIsClear c) w(
        move
        w)
    turnRight
    putMarker
    WHILE c( frontIsClear c) w(
        move
        w)
    turnRight
    putMarker
    WHILE c( frontIsClear c) w(
        move
        w)
    turnRight
    putMarker
    WHILE c( frontIsClear c) w(
        move
        w)
    turnRight
    putMarker
    m)
```

MAZE

```
DEF run m(
    IF c( frontIsClear c) i(
        turnLeft
        i)
    WHILE c( noMarkersPresent c)
         w(
        turnRight
        move
        w)
    m)
```

HARVESTER

```
DEF run m(
    turnLeft
    turnLeft
    pickMarker
    move
    pickMarker
    pickMarker
    move
    pickMarker
    move
    pickMarker
    move
    pickMarker
    move
    turnLeft
    pickMarker
    move
    pickMarker
    move
    pickMarker
    move
    pickMarker
    move
    turnLeft
    pickMarker
    move
    pickMarker
    move
    pickMarker
    move
    pickMarker
    move
    turnLeft
    pickMarker
    move
    pickMarker
    move
    pickMarker
    move
    m)
```

Figure 11: **Example Karel programs generated by LEAPS.** The programs that achieved the best reward out of all seeds are shown.

F.1 Program Behavior Reconstruction

This section serves as a complement to the ablation studies in the main paper, where we aim to justify the effectiveness of the proposed framework and the learning objectives. To this end, we select programs that are unseen to LEAPS and its ablations during the learning program embedding space from the testing set and reconstruct those programs using LEAPS, its ablations and the naïve program

Table 9: Extended reward comparison on original tasks with 8×8 or 12×12 grids and zero-shot generalization to 100×100 grids. LEAPS achieves the best generalization performance on all the tasks except for HARVESTER.

		STAIRCLIMBER	MAZE	FOURCORNER	TOPOFF	HARVESTER
DRL	Original	**1.00** (0.00)	**1.00** (0.00)	0.29 (0.05)	0.32 (0.07)	**0.90** (0.10)
	100x100	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.01 (0.01)	0.00 (0.00)
DRL-abs	Original	0.13 (0.29)	**1.00** (0.00)	0.36 (0.44)	0.63 (0.23)	0.32 (0.18)
	100x100	0.00 (0.00)	0.04 (0.05)	0.37 (0.44)	0.15 (0.12)	0.02 (0.01)
DRL-FCN	Original	**1.00** (0.00)	0.97 (0.03)	0.20 (0.34)	0.28 (0.12)	0.46 (0.16)
	100x100	-0.20 (0.10)	0.01 (0.01)	0.00 (0.00)	0.01 (0.01)	0.02 (0.00)
VIPER	Original	0.02 (0.02)	0.69 (0.05)	0.40 (0.42)	0.30 (0.06)	0.51 (0.07)
	100x100	0.00 (0.00)	0.10 (0.12)	0.40 (0.42)	0.03 (0.00)	**0.04** (0.00)
LEAPS	Original	**1.00** (0.00)	**1.00** (0.00)	**0.45** (0.40)	**0.81** (0.07)	0.45 (0.28)
	100x100	**1.00** (0.00)	**1.00** (0.00)	**0.45** (0.37)	**0.21** (0.03)	0.00 (0.00)

synthesis baseline. Those selected programs are shown in Figure 9 and the reconstructed programs are shown in Figure 10.

The naïve program synthesis baseline fails on the complex WHILE+2IF+IFELSE program, as it rarely synthesizes conditional and loop statements, instead generating long sequences of action tokens that attempt to replicate the desired behavior of those statements. We believe that this is because it is incentivized to initially predict action tokens to gain more immediate reward, making it less likely to synthesize other tokens. LEAPS and its variations perform better and synthesize more complex programs, demonstrating the importance of the proposed two-stage learning scheme in biasing program search. Also, LEAPS synthesizes programs that are more concise and induce behaviors which are more similar to given testing programs, justifying the effectiveness of the proposed learning objectives.

F.2 Karel Environment Tasks

This section is complementary to the main experiments in the main paper, where we compare LEAPS against the baselines on a set of Karel tasks, which is described in detail in Section K. The programs synthesized by LEAPS are presented in Figure 11.

The synthesized programs solve both STAIRCLIMBER and MAZE. For TOPOFF, since the average expected number of markers presented in the last row is 3, LEAPS synthesizes a sub-optimal program that conducts the topoff behavior three times. For CLEANHOUSE, while all the baselines fail on this task, the synthesized program achieves some performance by simply moving around and try to pick up markers. For HARVESTER, LEAPS fails to acquire the desired behavior that required nested loops but produces a sub-optimal program that contains only action tokens.

G Additional Generalization Experiments

Here, we present additional generalization experiments to complement those presented in Section 5.6. In Section G.1, we extend the 100x100 state size zero-shot generalization experiments to 3 additional tasks. In Section G.2, we analyze how well baseline methods and LEAPS can generalize to unseen configurations of a given task.

G.1 Generalization on FOURCORNER, TOPOFF, and HARVESTER

Evaluating zero-shot generalization performance assumes methods to work reasonably well on the original tasks. For this reason (and due to space limitations) we present only STAIRCLIMBER and MAZE for generalization experiments in the main text in Section 5.6 because most methods achieve reasonable performance on these two tasks, with DRL and LEAPS both solving these tasks fully and DRL-abs solving Maze fully.

However, here we also present full results for all tasks except CLEANHOUSE (as no method except LEAPS has a reasonable level of performance on it). The results are summarized in Table 9. We see that LEAPS generalizes well on FOURCORNER and maintains the best performance on TOPOFF. It is outperformed on HARVESTER, although none of the methods do well on HARVESTER as the highest

Table 10: Mean return (standard deviation) [% change in performance] on generalizing to unseen configurations on TOPOFF and HARVESTER task.

TOPOFF	Training configuration %				
	75%	50%	25%	10%	5%
DRL	0.17 (0.05) [-46.8%]	0.12 (0.09) [-62.5%]	0.12 (0.06) [-62.5%]	0.17 (0.13) [-46.8%]	0.13 (0.04) [-59.4%]
DRL-abs	0.23 (0.29) [-63.5%]	0.29 (0.36) [-54.0%]	0.45 (0.45) [-28.6%]	0.24 (0.38) [-61.9%]	0.26 (0.37) [-18.8%]
VIPER	0.27 (0.03) [-10.0%]	0.28 (0.04) [-6.67%]	0.27 (0.06) [-10.0%]	0.27 (0.02) [-10.0%]	0.28 (0.03) [-6.67%]
LEAPS	**0.68** (0.18) [-15.0%]	**0.65** (0.13) [-18.8%]	**0.61** (0.24) [-23.8%]	**0.68** (0.21) [-15.0%]	**0.67** (0.18) [-16.3%]

HARVESTER	Training configuration %				
	75%	50%	25%	10%	5%
DRL	**0.64** (0.24) [-28.9%]	0.71 (0.29) [-21.1%]	0.21 (0.06) [-76.7%]	0.14 (0.09) [-84.4%]	0.04 (0.01) [-95.6%]
DRL-abs	0.14 (0.21) [-56.3%]	0.24 (0.25) [-25.0%]	0.05 (0.06) [-84.4%]	0.13 (0.21) [-59.4%]	0.31 (0.31) [-3.13%]
VIPER	0.54 (0.01) [+5.88%]	**0.54** (0.02) [+5.88%]	**0.55** (0.01) [+7.84%]	**0.54** (0.01) [+5.88%]	**0.44** (0.22) [-13.7%]
LEAPS	0.40 (0.30) [-13.0%]	0.42 (0.27) [-8.69%]	0.50 (0.35) [+08.69%]	0.12 (0.19) [-73.9%]	0.01 (0.03) [-97.6%]

obtained reward by any method is 0.04 (by VIPER). In summary, LEAPS performs the best on 4 out of these 5 tasks, further demonstrating its superior zero-shot generalization performance.

Furthermore, we note that it is possible that a DRL policy employing a fully convolutional network (FCN) as proposed in Long et al. [122] can handle varying observation sizes. FCNs were also demonstrated in Silver et al. [49] to demonstrate better generalization performance than traditional convolutional neural network policies. However, we hypothesize that the generalization performance here will still be poor as there is a large increase in the number of features that the FCN architecture needs to aggregate when transferring from 8x8/12x12 state inputs to 100x100 inputs—a 10x input size increase that FCN is not specifically designed to deal with. We have included both FCN's zero-shot generalization results and its results on the original grid sizes in Table 9. DRL-FCN, where we have replaced the policy and value function networks of PPO with an FCN, does manage to perform zero-shot transfer marginally better than DRL performs when training from scratch (as it DRL's architecture cannot handle varied input sizes) on MAZE and HARVESTER. However, it obtains a negative reward on STAIRCLIMBER as it attempts to navigate away from the stairs when transferring to the 100×100 grid size. Its performance is still far worse than LEAPS and VIPER on most tasks, demonstrating that the programmatic structure of the policy is important for these tasks.

G.2 Generalization to Unseen Configurations

We present a generalization experiment in the main paper to study how well the baselines and the programs synthesized by the proposed framework can generalize to larger state spaces that are unseen during training without further learning on the STAIRCLIMBER and MAZE tasks. In this section, we investigate the ability of generalizing to different configurations, which are defined based on the marker placement related to solve a task, on both the TOPOFF task and HARVESTER task.

Since solving TOPOFF requires an agent to put markers on top of all markers on the last row, the initial configurations are determined by the marker presence on the last row. The grid has a size of 10×10 inside the surrounding wall. We do not spawn a marker at the bottom right corner in the last row, leaving 9 possible locations with marker, allowing 2^9 possible initial configurations. On the other hand, HARVESTER requires an agent to pick up all the markers placed in the grid. The grid has a size of 6×6 inside the surrounding wall, leaving 36 possible locations in grid with a marker, resulting in 2^{36} possible initial configurations.

We aim to test if methods can learn from only a small portion of configurations during training and still generalize to all the possible configurations without further learning. To this end, we experiment using 75%, 50%, 25%, 10%, 5% of the configurations for training DRL, DRL-abs, and VIPER and for the program search stage of LEAPS. Then, we test zero-shot generalization of the learned models and programs on all the possible configurations. We report the performance in Table 10. We compare the performance each method achieves to its own performance learning from all the configurations (reported in the main paper) to investigate how limiting training configurations affects the performance. Note that the results of training and testing on 100% configurations are reported in the main paper, where no generalization is required.

TopOff. LEAPS outperforms all the baselines on the mean return on all the experiments. VIPER and LEAPS enjoy the lowest and the second lowest performance decrease when learning from only a portion of configurations, which demonstrates the strength of programmatic policies. DRL-abs slightly outperforms DRL, with better absolute performance and lower performance decrease. We believe that this is because DRL takes entire Karel grids as input, and therefore held out configurations are completely unseen to it. In contrast, DRL-abs takes abstract states (*i.e.* local perceptions) as input, which can alleviate this issue.

Harvester. VIPER outperforms almost all other methods on absolute performance and performance decrease, while LEAPS achieves second best results, which again justifies the generalization of programmatic policies. Both DRL and DRL-abs are unable to generalize well when learning from a limited set of configurations, except in the case of DRL-abs learning from 5% of configurations, which can be attributed to the high-variance of DRL-abs results.

H Additional Analysis on Experimental Results

Due to the limited space in the main paper, we include additional analysis of the experimental results in this section.

H.1 DRL vs. DRL-abs

We hypothesize that DRL-abs does not always outperform DRL due to imperfect perception (i.e. state abstraction) design. DRL-abs takes abstract states as input (*i.e.* `frontIsClear()`, `leftIsClear()`, `rightIsClear()`, `markerPresent()` in our design), which only describe local perception while omitting the information of the entire map. Therefore, for tasks such as StairClimber, Harvester, and CleanHouse, which would be easier to solve with access to the entire Karel grid, DRL might outperform DRL-abs. In this work, DRL-abs' abstract states are the perceptions from the DSL we synthesize programs with to make the comparisons fair against our method as well as analyzing the effects of abstract states in the DRL domain. However, a more sophisticated design for perception/state abstraction could potentially improve the performance of DRL-abs.

H.2 VIPER generalization

VIPER operates on the abstract state space which is invariant to grid size. However, for the reasons below, it is still unable to transfer the behavior to the larger grid despite its abstract state representation. We hypothesize that VIPER's performance suffers on zero-shot generalization for two main reasons.

1. It is constrained to imitate the DRL teacher policy during training, which is trained on the smaller grid sizes. Thus its learned policy also experiences difficulty in zero-shot generalization to larger grid sizes.
2. Its decision tree policies cannot represent certain looping behaviors as they simply perform a one-to-one mapping from abstract state to action, thus making it difficult to learn optimal behaviors that require a one-to-many mapping between an abstract state and a set of desired actions. Empirically, we observed that training losses for VIPER decision trees were much higher for tasks such as StairClimber which require such behaviors.

I Detailed Descriptions and Illustrations of Ablations and Baselines

This section provides details on the variations of LEAPS used for ablations studies and the baselines which we compare against. The descriptions of the ablations of LEAPS are presented in Section I.1 and the illustrations are shown in Figure 12. The naïve program synthesis baseline is illustrated in Figure 13 (c) for better visualization. Then, the descriptions of the baselines are presented in Section I.2 and the illustrations are shown in Figure 13.

I.1 Ablations

We first ablate various components of our proposed framework in order to (1) justify the necessity of the proposed two-stage learning scheme and (2) identify the effects of the proposed objectives. We consider the following baselines and ablations of our method.

- Naïve: the naïve program synthesis baseline is a policy that learns to directly synthesize a program from scratch by recurrently predicting a sequence of program tokens. The architecture of this baseline is a recurrent neural network which takes an initial starting token as the input at the first time step, and then sequentially outputs a program token at each time step to compose a program until an end token is produced. Note that the observation of this baseline is its own previously outputted program token instead of the state of the task environment (*e.g.* Karel grids). Also, at each time step, this baseline produces a distribution over all the possible program tokens in the given DSL instead of a distribution over agent's action in the task environment (*e.g.* `move()`). This baseline investigates if an end-to-end learning method can solve the problem. This baseline is illustrated in Figure 13 (c).
- LEAPS-P: the simplest ablation of LEAPS, in which the program embedding space is learned by only optimizing the program reconstruction loss $\mathcal{L}^{\text{P}}$. This baseline is illustrated in Figure 12 (a).
- LEAPS-P+R: an ablation of LEAPS which optimizes both the program reconstruction loss $\mathcal{L}^{\text{P}}$ and the program behavior reconstruction loss $\mathcal{L}^{\text{R}}$. This baseline is illustrated in Figure 12 (b).
- LEAPS-P+L: an ablation of LEAPS which optimizes both the program reconstruction loss $\mathcal{L}^{\text{P}}$ and the latent behavior reconstruction loss $\mathcal{L}^{\text{L}}$. This baseline is illustrated in Figure 12 (c).
- LEAPS (LEAPS-P+R+L): LEAPS with all the losses, optimizing our full objective.
- LEAPS-rand-{8/64}: like LEAPS, this ablation also optimizes the full objective for learning the program embedding space. But when searching latent programs, instead of CEM, it simply randomly samples 8/64 candidate latent programs and chooses the best performing one. These baselines justify the effectiveness of using CEM for searching latent programs.

I.2 Baselines

We evaluate LEAPS against the following baselines (illustrated in Figure 13).

- DRL: a neural network policy trained on each task and taking raw states (Karel grids) as input. A Karel grid is represented as a binary tensor with dimension $W \times H \times 16$ (there are 16 possible states for each grid square) instead of an image. This baseline is illustrated in Figure 13 (a).
- DRL-abs: a recurrent neural network policy directly trained on each Karel task but instead of taking raw states (Karel grids) as input it takes *abstract* states as input (*i.e.* it sees the same perceptions as LEAPS). Specifically, all returned values of perceptions including `frontIsClear()==true`, `leftIsClear()==false`, `rightIsClear()==true`, `markersPresent()==false`, and `noMarkersPresent()==true` are concatenated as a binary vector, which is then fed to the DRL-abs policy as its input. This baseline allows for a fair comparison to LEAPS since the program execution process also utilizes abstract state information. This baseline is illustrated in Figure 13 (b).
- DRL-abs-t: a DRL *transfer* learning baseline in which for each task, we train DRL-abs policies on all other tasks, then fine-tune them on the current task. Thus it acquires a prior by learning to first solve other Karel tasks. Rewards are reported for the policies from the task that transferred with highest return. We only transfer DRL-abs policies as some tasks have different state spaces so that transferring a DRL policy trained on a task to another task with a different state space is not possible.

  This baseline is designed to investigate if acquiring task related priors allows DRL policies to perform better on our Karel tasks. Unlike LEAPS, which acquires priors from a dataset consisting of randomly generated programs and the behaviors those program induce in the environment, DRL-abs-t allows for acquiring priors from goal-oriented behaviors (*i.e.* other Karel tasks).
- HRL: a hierarchical RL baseline in which a VAE is first trained on action sequences from program execution traces used by LEAPS. Once trained, the decoder is utilized as a low-level policy for

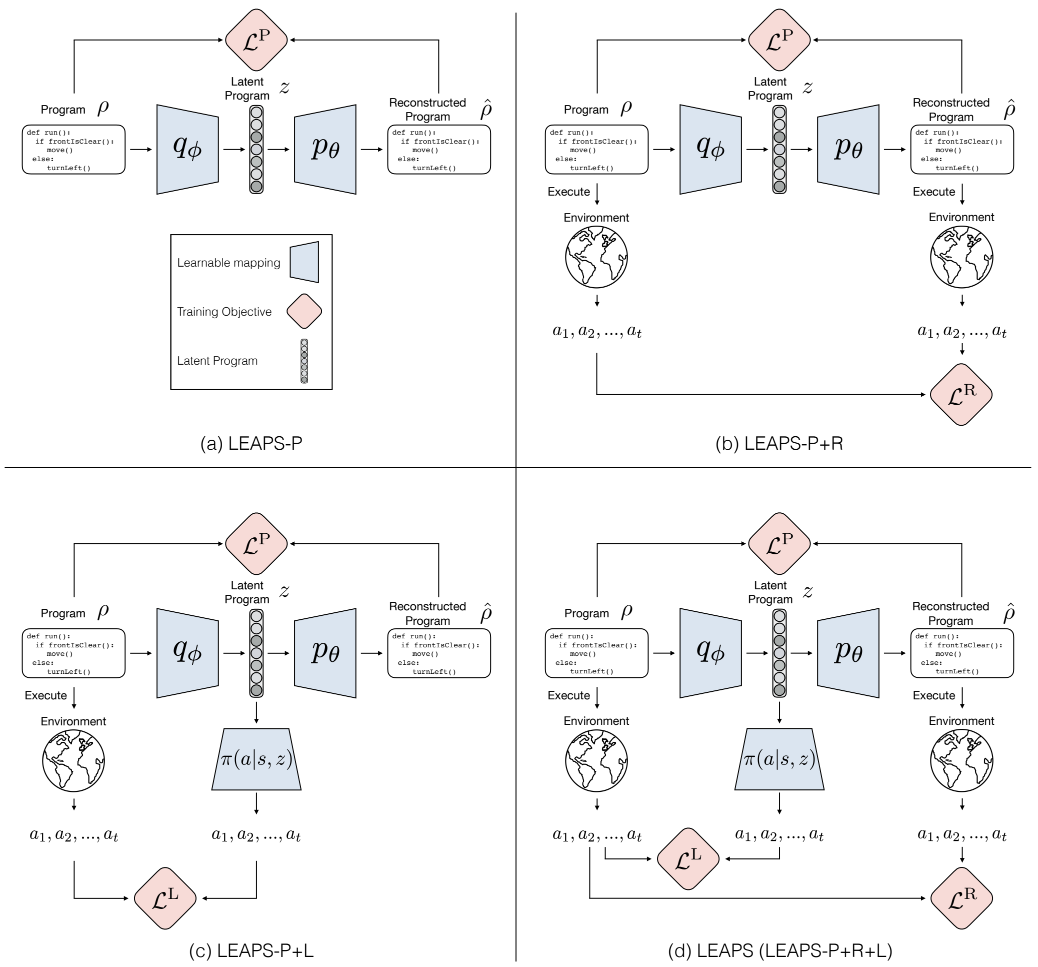


Figure 12: **LEAPS Variations Illustrations.** Blue trapezoids represent the modules whose parameters are being learned in the learning program embedding stage. Red diamonds represent the learning objectives. Gray rounded rectangle represent latent programs (*i.e*. program embeddings), which are vectors. (a) LEAPS-P: the simplest ablation of LEAPS, in which the program embedding space is learned by only optimizing the program reconstruction loss $\mathcal{L}^{\text{P}}$. (b) LEAPS-P+R: an ablation of LEAPS which optimizes both the program reconstruction loss $\mathcal{L}^{\text{P}}$ and the program behavior reconstruction loss $\mathcal{L}^{\text{R}}$. (c) LEAPS-P+L: an ablation of LEAPS which optimizes both the program reconstruction loss $\mathcal{L}^{\text{P}}$ and the latent behavior reconstruction loss $\mathcal{L}^{\text{L}}$. (d) LEAPS (LEAPS-P+R+L): our proposed framework that optimizes all the proposed objectives.

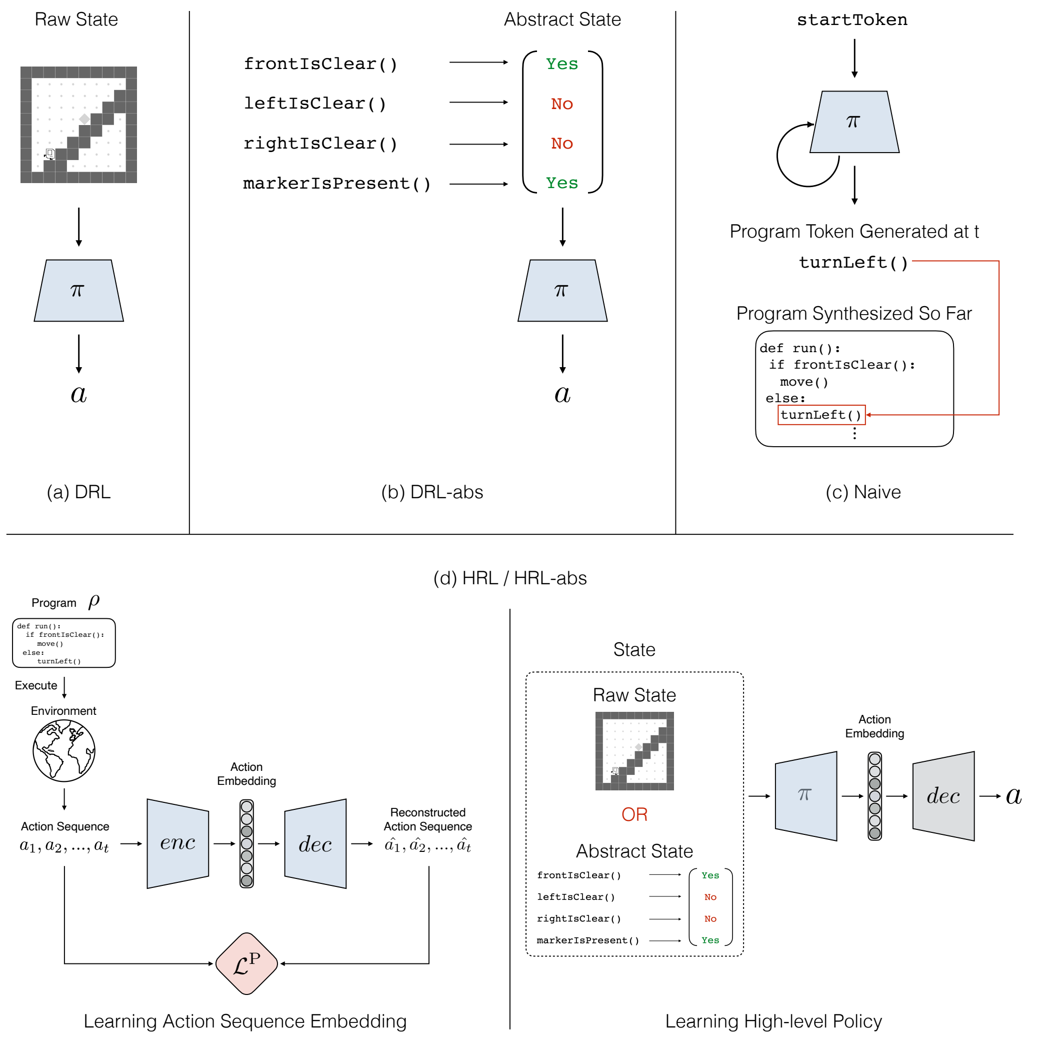
learning a high-level policy to sample actions from. Similar to LEAPS, this baseline utilizes the dataset to produce a prior of the domain. It takes raw states (Karel grids) as input.

This baseline is also designed to investigate if acquiring priors allow DRL policies to perform better. Similar to LEAPS, which acquires priors from a dataset consisting of randomly generated programs and the behaviors those program induce in the environment, HRL is trained to acquire priors by learning to reconstruct the behaviors induced by the programs. One can also view this baseline as a version of the framework proposed in [123] with some simplifications, which also learns an embedding space using a VAE and then trains a high-level policy to utilize this embedding space together with the low-level policy whose parameters are frozen. This baseline is illustrated in Figure 13 (d).

- HRL-abs: the same method as HRL but taking abstract states (*i.e*. local perceptions) as input. This baseline is illustrated in Figure 13 (d).

- VIPER [12]: A decision-tree programmatic policy which imitates the behavior of a deep RL teacher policy via a modified DAgger algorithm [66]. This decision tree policy cannot synthesize loops, allowing us to highlight the performance advantages of more expressive program representation that LEAPS is able to take advantage of.

All the baselines are trained with PPO [67] or SAC [68], including the VIPER teacher policy. More training details can be found in Section L.



```
def run():
  if frontIsClear():
    move()
  else:
    turnLeft()
      ⋮
```

Figure 13: **Baseline Methods Illustrations.** (a) DRL: a DRL policy that takes raw state input (*i.e.* a Karel grid represented as a $W \times H \times 12$ binary tensor as there are 12 possible states for each grid square). (b) DRL-abs: a DRL policy that takes abstract state input, containing a vector of returned values of perceptions, *e.g.* `frontIsClear()==true` and `markersPresent()==false`. (c) Naive: a naïve program synthesis baseline that learns to directly synthesize a program from scratch by recurrently predicting a sequence of program tokens. (d) HRL/HRL-abs: a hierarchical RL baseline in which a VAE, consisting of a encoder enc and a decoder dec, is first trained to reconstruct action sequences from program execution traces used by LEAPS. Once the action embedding space is learned, it employs a high-level policy π that learns from scratch to solve task by predicting a distribution in the learned action embedding space. Note that the parameters of the decoder dec are frozen (represented in gray) when the high-level policy is learning. The HRL policy takes raw state input (same as the DRL baseline) and the HRL-abs policy takes abstract state input (same as the DRL-abs baseline).

J Program Dataset Generation Details

To learn a program embedding space for the proposed framework and its ablations, we randomly generate 50k programs to form a dataset with 35k training programs and 7.5k programs for validation and testing. Simply generating programs by uniformly sampling all the tokens from the DSL would yield programs that mainly only contain action tokens since the chance to synthesize conditional statements with correct grammar is low. Therefore, to produce programs that are longer and deeply nested with conditional statements to induce more complex behaviors, we propose to sample programs using a probabilistic sampler.

To generate each program, we sample program tokens according to the probabilities listed in Table 11 at every step until we sample an ending token or when a maximum program length is reached. When generating programs, we ensure that no program is identical to any other. Each token is generated sequentially, and length is effectively governed by the `STMT_STMT` token detailed in Table 11's caption. There is a maximum depth limit of 4 nested conditional/loop statements, and a maximum statement depth limit of 6 (can't have more than 6 nested `STMT_STMT` tokens). Note that this sampling procedure does not guarantee that the programs generated will terminate, hence when executing them to obtain ground-truth interactions for training the Program Behavior and Latent Behavior Reconstruction losses we limit the max program execution length to 100 environment timesteps. This sampling procedure results in the distribution of program lengths seen in Figure 14.

Intuitively, shorter lengths can bias synthesized programs to compress the same behaviors into fewer tokens through the use of loops, making program search easier. Therefore, in our experiments, we have limited the maximum output program length of LEAPS to 45 tokens (as the maximum in the dataset is 44). As shown in the example programs generated by LEAPS in Figure 11, LEAPS successfully generates loops for our Karel tasks, which can be probably attributed to this bias of program length. We further verify this intuition by rerunning LEAPS with the max program length set to 100 tokens on the Karel tasks. We display generated programs in Table 12, where we see that some of the generated programs are indeed much longer and lack loop statements and structures.

Table 11: The probability of sampling program tokens when generating the program dataset. Tokens are generated sequentially, and STMT_STMT refers to breaking up the current token into two tokens, each of which is selected according to the same probability distribution again. Thus it effectively controls how long programs will be.

	`WHILE`	`REPEAT`	`STMT_STMT`	`ACTION`	`IF`	`IFELSE`
Standard Dataset	0.15	0.03	0.5	0.2	0.08	0.04

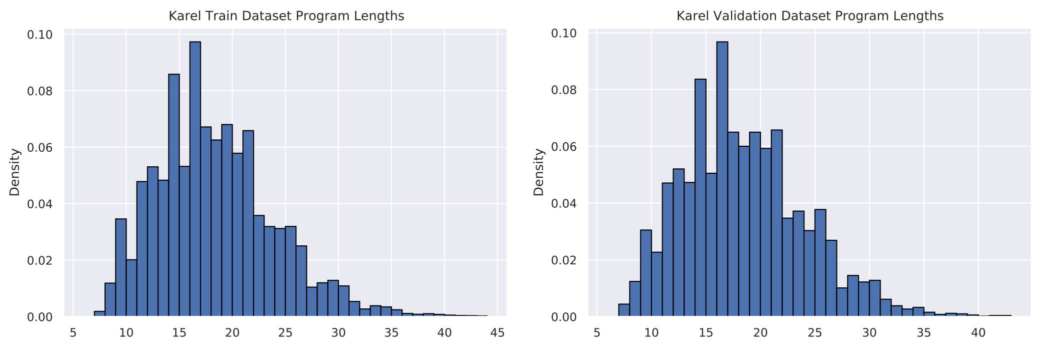


Figure 14: Histograms of the program length (*i.e*. number of program tokens) in the training and validation datasets.

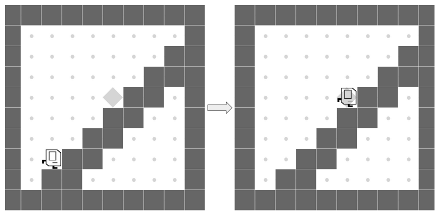
K Karel Task Details

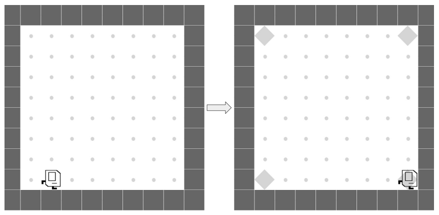
MDP Tasks We utilize environment state based reward functions for the RL tasks STAIRCLIMBER, FOURCORNER, TOPOFF, MAZE, HARVESTER, and CLEANHOUSE. For each task, we average

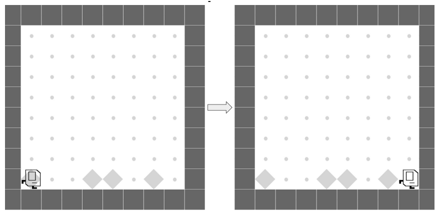
Table 12: **LEAPS Length 100 Synthesized Karel Programs.** Line breaks are not shown here as the programs are very long. The examples picked are ones that represent the programs generated by most seeds for each task. Without the 45 token restriction on program lengths, programs for TOPOFF, FOURCORNER, and HARVESTER are very long and have repetitive movements that can easily be put into `REPEAT` or `WHILE` loops. The CLEANHOUSE program also contains repeated, somewhat redundant `WHILE` loops. MAZE and STAIRCLIMBER programs are mostly unaffected by the change in maximum program length. These programs demonstrate that the bias induced by program length restriction is important for producing more complex programs in the program synthesis phase of LEAPS.

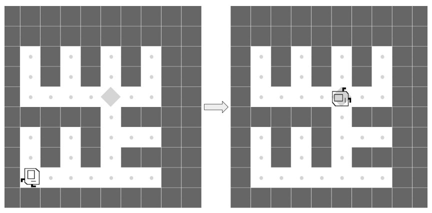
Karel Task	Program
STAIRCLIMBER	`DEF run m( turnLeft turnRight turnLeft turnLeft turnRight WHILE c( noMarkersPresent c) } w( turnLeft move w) m)`
TOPOFF	`DEF run m(WHILE c(noMarkersPresent c) w(move w) turnRight turnRight turnRight turnRight turnRight} turnRight putMarker turnRight turnRight move turnRight move turnRight move turnRight move turnRight move turnRight move turnRight move turnRight move turnRight move turnRight move turnRight move turnRight move turnRight move turnRight move turnRight move turnRight move turnRight move turnRight move m)`
CLEANHOUSE	`DEF run m(turnRight pickMarker turnLeft turnRight turnLeft pickMarker move turnLeft WHILE c(leftIsClear c) w(pickMarker move w) turnRight turnLeft pickMarker move turnLeft WHILE c(leftIsClear c) w(pickMarker move w) turnLeft pickMarker} WHILE c(leftIsClear c) w(pickMarker move turnLeft pickMarker w)} WHILE c(noMarkersPresent c) w(turnLeft move pickMarker w) turnLeft pickMarker turnLeft m)`
FOURCORNER	`DEF run m(turnRight WHILE c(frontIsClear c) w(move w) turnRight WHILE c(frontIsClear c) w(move w) turnRight WHILE c(frontIsClear c) w(move w) turnRight putMarker WHILE c(frontIsClear c) w(move w) turnRight putMarker WHILE c(frontIsClear c) w(move w)} turnRight putMarker WHILE c(frontIsClear c) w(move w) turnRight putMarker m)`
MAZE	`DEF run m( WHILE c( noMarkersPresent c) w( REPEAT R=1 r( turnRight r) move w) turnLeft turnRight m)`
HARVESTER	`DEF run m(turnLeft turnRight pickMarker move pickMarker move turnRight move pickMarker move pickMarker move turnRight move pickMarker move pickMarker move pickMarker move turnRight move pickMarker move pickMarker move pickMarker move turnRight move pickMarker move pickMarker move pickMarker move pickMarker move turnRight move pickMarker move pickMarker move pickMarker move pickMarker move turnRight move pickMarker move pickMarker move pickMarker move pickMarker move pickMarker move turnRight move pickMarker move pickMarker move pickMarker move pickMarker move pickMarker move turnRight move m)`

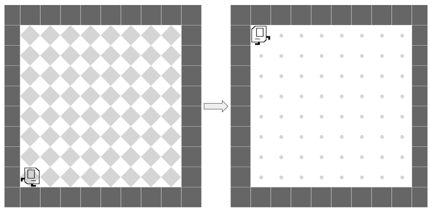
performance of the policies on 10 random environment start configurations. For all tasks with marker placing objectives, the final reward will be 0—regardless of the any other agent actions—if a marker is placed in the wrong location. This is done in order to discourage "spamming" marker placement on every grid location to exploit the reward functions. All rewards described below are then normalized so that the return is between [0, 1.0] for tasks without penalties, and [-1.0, 1.0] for tasks with negative penalties, for easier learning for the DRL methods. We visualize all tasks as well as their start and ideal end states in Figure 15 on a 10×10 grid for consistency in the visualizations (except CLEANHOUSE).


(a) STAIRCLIMBER


(b) FOURCORNER


(c) TOPOFF


(d) MAZE


(e) HARVESTER

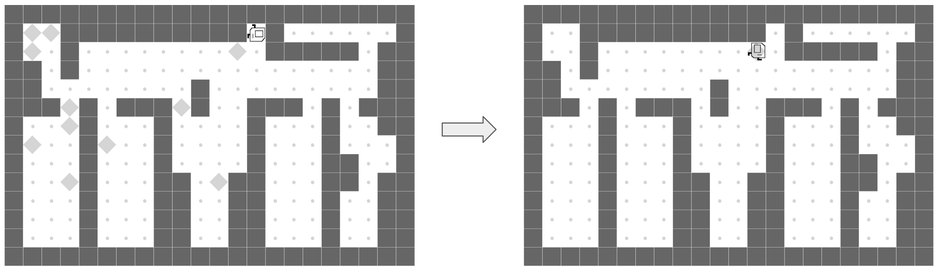

(f) CLEANHOUSE

Figure 15: Example of initial configurations and their ideal end states of the Karel tasks. Note that we show only one example of initial configuration and its ideal end sate pair for each task. However, markers, walls and agent's position are randomized in initial configurations depending upon task. Please see section K for more details.

K.1 STAIRCLIMBER

The goal is to climb the stairs to reach where the marker is located. The reward is defined as a sparse reward: 1 if the agent reaches the goal in the environment rollout, -1 if the agent moves to a position off of the stairs during the rollout, and 0 otherwise. This is on a 12×12 grid, and the marker location and agent's initial location are randomized between rollouts.

K.2 FOURCORNER

The goal is to place a marker at each corner of the Karel environment grid. The reward is defined as sum of corners having a marker divided by four. If the Karel state has a marker placed in wrong location, the reward will be 0. This is on a 12×12 grid.

K.3 TOPOFF

The goal is to place a marker wherever there is already a marker in the last row of the environment, and end up in the rightmost square on the bottom row at the end of the rollout. The reward is defined as the number of consecutive places until the agent either forgets to place a marker where the marker is already present or places a marker at an empty location in last row, with a bonus for ending up on the last square. This is on a 12×12 grid, and the marker locations in the last row are randomized between rollouts.

K.4 MAZE

The goal is to find a marker in randomly generated maze. The reward is defined as a sparse reward: 1 if the agent finds the marker in the environment rollout, 0 otherwise. This is on a 8×8 grid, and the marker location, agent's initial location, and the maze configuration itself are randomized between rollouts.

K.5 CLEANHOUSE

We design a complex 14×22 Karel environment grid that resembles an apartment. The goal is to pick up the garbage (markers) placed at 10 different locations and reach the location where there is a dustbin (2 markers in 1 location). To make the task simpler, we place the markers adjacent to any wall in the environment. The reward is defined as total locations cleaned (markers picked) out of the total number of markers placed in initial Karel environment state (10). The agent's initial location is fixed but the marker locations are randomized between rollouts.

K.6 HARVESTER

The goal is to pickup a marker from each location in the Karel environment. The final reward is defined as the number of markers picked up divided the total markers present in the initial Karel environment state. This is on a 8×8 grid. We run both MAZE and HARVESTER on smaller Karel environment grids to save time and compute resources because these are long horizon tasks.

L Hyperparameters and Training Details

L.1 DRL and DRL-abs

RL training directly on the Karel environment is performed with the PPO algorithm [67] for 2M timesteps using the ALF codebase[4]. We tried a discretized SAC [68] implementation (by replacing Gaussian distributions with Categorical distributions), but it was outperformed by PPO on the Karel tasks on all environments. We also tried tabular Q-learning from raw Karel grids (it wouldn't work well on abstract states as the state is partially observed), however it was also consistently outperformed by PPO. For DRL, the policies and value networks are the same with a shared convolutional encoder that first processes the state (as the Karel state size is $(H \times W \times 16)$ for 16 possible agent direction or marker placement values that each state in the grid can take on at a time. The convolutional encoder consists of two layers: the first with 32 filters, kernel size 2, and stride 1, the second with 32 filters, kernel size 4, and stride 1. For DRL-abs, the policy and value networks are both comprised of an LSTM layer and a 2-layer fully connected network, all with hidden sizes of 100.

For each task, we perform a comprehensive hyperparameter grid search over the following parameters, and report results from the run with the best averaged final reward over 5 seeds.

The hyperparameter grid is listed below, shared parameters are also listed:

[4]https://github.com/HorizonRobotics/alf/

- Importance Ratio Clipping: {0.05, 0.1, 0.2}
- Advantage Normalization: {True, False}
- Entropy Regularization: {0.1, 0.01, 0.001}
- Number of updates per training iteration (This controls the ratio of gradient steps to environment steps): {1, 4, 8, 16}
- Number of environment steps per set of training iterations: 32
- Number of parallel actors: 10
- Optimizer: Adam
- Learning Rate: 0.001
- Batch Size: 128

Hyperparameters that performed best for each task are listed below.

DRL	Import Ratio Clip	Adv Norm	Entropy Reg	Updates per Train Iter
CLEANHOUSE	0.1	True	0.01	4
FOURCORNER	0.2	True	0.01	16
HARVESTER	0.05	True	0.01	8
MAZE:	0.05	True	0.001	8
STAIRCLIMBER	0.1	True	0.1	4
TOPOFF	0.05	True	0.001	4

DRL-abs	Import Ratio Clip	Adv Norm	Entropy Reg	Updates per Train Iter
CLEANHOUSE	0.2	True	0.01	8
FOURCORNER	0.05	True	0.01	4
HARVESTER	0.2	True	0.01	4
MAZE:	0.2	True	0.001	4
STAIRCLIMBER	0.05	True	0.1	16
TOPOFF	0.2	True	0.001	8

L.2 DRL-abs-t

DRL-abs-t is limited to DRL-abs policies as the state spaces are different for some of the Karel tasks. For DRL-abs-t, we use the best hyperparameter configuration for each Karel task to train a policy to 1M timesteps. Then, we attempt direct policy transfer to each other task by training for another 1M timesteps on the new task with the same hyperparameters (excluding transferring to the same task). Numbers reported are from the task transfer that achieved the highest reward. The tasks that we transfer from for each task are listed below:

DRL-abs-t	Transferred from
CLEANHOUSE	HARVESTER
FOURCORNER	TOPOFF
HARVESTER	MAZE
MAZE	STAIRCLIMBER
STAIRCLIMBER	HARVESTER
TOPOFF	HARVESTER

L.3 HRL

Pretraining stage: We first train a VAE to reconstruct action trajectories generated from our program dataset. For each program, we generate 10 rollouts in randomly configured Karel environments to produce the HRL dataset, giving this baseline the same data as LEAPS. These variable-length action sequences are encoded via an LSTM encoder into a 10-dimensional, continuous latent space and decoded by an LSTM decoder into the original action trajectories. We chose 10-dimensional so as to not make downstream RL too difficult. We tune the KL divergence weight (β) of this network such that it's as high as possible while being able to reconstruct the trajectories well. Network/training details below:

- β: 1.0
- Optimizer: Adam (All optimizers)
- Learning Rates: 0.0003
- Hidden layer size: 128
- # LSTM layers (both encoder/decoder): 2
- Latent embedding size: 10
- Nonlinearity: ReLU
- Batch Size: 128

Downstream (Hierarchical) RL On our Karel tasks, we use the VAE's decoder to decode latent vectors (actions for the RL agent) into varied-length action sequences for all Karel tasks. The decoder parameters are frozen and used for all environments. The RL agent is retrained from scratch for each task, in the same manner as the standard RL baselines DRL-abs and DRL. We use Soft-Actor Critic (SAC, Haarnoja et al. [68]) as the RL algorithm as it is state of the art in many continuous action space environments. SAC grid search parameters for all environments follow below:

- Number of updates per training iteration: {1, 8}
- Number of environment steps per set of training iterations: 8 (multiplied by the number of steps taken by the decoder in the environment)
- Polyak Averaging Coefficient: {0.95, 0.9}
- Number of parallel actors: 1
- Batch size: 128
- Replay buffer size: 1M

The best hyperparameters follow:

HRL-abs	Updates per Train Iter	Polyak Coefficient
CLEANHOUSE	1	0.95
FOURCORNER	8	0.9
HARVESTER	8	0.95
MAZE	1	0.95
STAIRCLIMBER	1	0.9
TOPOFF	1	0.9

HRL	Updates per Train Iter	Polyak Coefficient
CLEANHOUSE	1	0.9
FOURCORNER	1	0.95
HARVESTER	1	0.95
MAZE	8	0.9
STAIRCLIMBER	8	0.95
TOPOFF	8	0.95

L.4 Naïve

The naïve program synthesis baseline takes an initial token as input and outputs an entire program at each timestep to learn a recurrent policy guided by the rewards of these programs. We execute these generated programs on 10 random environment start configurations in Karel to get the reward. We run PPO for 2M Karel environment timesteps. The policy network is comprised of one shared GRU layer, followed by two fully connected layers, for both the policy and value networks. For evaluation, we generate 64 programs from the learned policy, and choose the program with the maximum reward on 10 demonstrations. For each task, we perform a hyperparameter grid search over the following parameters, and report results from the run with the best averaged final reward over 5 seeds. We exponentially decay the entropy loss coefficient in PPO from the initial to final entropy coefficient to avoid local minima during the initial training steps.

- Learning Rate: 0.0005
- Batch Size (B): {64, 128, 256}
- initial entropy coefficient (E_i): {1.0, 0.1}
- final entropy coefficient: {0.01}
- Hidden Layer Size: 64

Hyperparameters that performed best for each task are listed below.

Naïve	B	E_i
WHILE	128	0.1
IFELSE+WHILE	256	1.0
2IF+IFELSE	256	0.1
WHILE+2IF+IFELSE	128	0.1

Naïve	B	E_i
CLEANHOUSE	128	0.1
FOURCORNER	128	1.0
HARVESTER	128	1.0
MAZE	256	1.0
STAIRCLIMBER	128	1.0
TOPOFF	128	1.0

L.5 VIPER

VIPER [12] builds a decision tree programmatic policy by imitating a given teacher policy. We use the best DRL policies as teachers instead of the DQN [124] teacher policy used in Bastani et al. [12]. We did this in order to give the teacher the best performance possible for maximum fairness in comparison against VIPER, as we empirically found the PPO policy to perform much better on our tasks than a DQN policy.

We perform a grid search over VIPER hyperparameters, listed below:

- Max depth of decision tree: {6, 12, 15}
- Max number of samples for tree policy: {100k, 200k, 400k}
- Sample reweighting: {True, False}

The best hyperparameters found for each task are listed below:

VIPER	Max Depth	Max Num Samples	Sample Reweighting
CLEANHOUSE	6	100k	False
FOURCORNER	12	100k	False
HARVESTER	12	400k	True
MAZE	12	100k	True
STAIRCLIMBER	12	400k	True
TOPOFF	15	100k	False

L.6 Program Embedding Space VAE Model

Encoder-Decoder Architecture. The encoder and decoder are both recurrent networks. The encoder structure consists of a PyTorch token embedding layer, then a recurrent GRU cell, and two linear layers that produce μ and $\log \sigma$ vectors to sample the program embedding.

The decoder consists of a recurrent GRU cell which takes in the embedding of the previous token generated and then a linear token output layer which models the log probabilities of all discrete tokens. Since we have access to DSL grammar during program synthesis, we utilize a syntax checker based on the Karel DSL grammar from Bunel et al. [17] at the output of the decoder to limit predictions to syntactically valid tokens. We restrict our decoder from predicting syntactically invalid programs by masking out tokens that make a program syntactically invalid at each timestep. This syntax checker is designed as a state machine that keeps track of a set of valid next tokens based on the current token, open code blocks (*e.g.* `while, if, ifelse`) in the given partial program, and the grammar rules of our DSL. Since we generate a program as a sequence of tokens, the syntax checker outputs at each timestep a mask M, where $M \in \{-\infty, 0\}^{\text{number of DSL tokens}}$, and

$$M_j = \begin{cases} -\infty & \text{if the j-th token is not valid in the current context} \\ 0 & \text{otherwise} \end{cases}$$

This mask is added to the output of the last layer of the decoder, just before the Softmax operation that normalizes the output to a probability over the tokens.

π **Architecture.** The program-embedding conditioned policy π consists of a GRU layer that operates on the inputs and three MLP layers that output the log probability of environment actions. Specifically, it takes a latent program vector, current environment state, and previous action as input and outputs the predicted environment action for each timestep.

To evaluate how close the predicted neural execution traces are to the execution traces of the ground-truth programs, we consider the following metrics:

- Action token accuracy: the percentage of matching actions in the predicted execution traces and the ground-truth execution traces.
- Action sequence accuracy: the percentage of matching action sequences in the predicted execution traces and the ground-truth execution traces. It requires that a predicted execution trace entirely matches the ground-truth execution trace.

After convergence, our model achieves an action token accuracy of 96.5% and an action sequence accuracy of 91.3%.

Training. The reinforcement learning algorithm used for the program behavior reconstruction $\mathcal{L}^{\text{R}}$ is REINFORCE [64].

When training LEAPS with all losses, we first train with the Program Reconstruction ($\mathcal{L}^{\text{P}}$) and Latent Behavior Reconstruction ($\mathcal{L}^{\text{L}}$) losses, essentially setting $\lambda_1 = \lambda_3 = 1$ and $\lambda_2 = 0$ of our full objective, reproduced below:

$$\min_{\theta,\phi,\pi} \lambda_1 \mathcal{L}^{\text{P}}_{\theta,\phi}(\rho) + \lambda_2 \mathcal{L}^{\text{R}}_{\theta,\phi}(\rho) + \lambda_3 \mathcal{L}^{\text{L}}_{\pi}(\rho, \pi), \tag{6}$$

Once this model is trained for one epoch, we then train exclusively with the Program Behavior Reconstruction loss ($\mathcal{L}^{\text{R}}$), setting $\lambda_2 = 1$ and $\lambda_1 = \lambda_3 = 0$, with equal number of updates. These two update steps are repeated alternatively till convergence is achieved. This is done to avoid potential issues of updating with supervised and reinforcement learning gradients at the same time. We did not attempt to train these 3 losses jointly.

All other shared hyperparameters and training details are listed below:

- β: 0.1
- Optimizer: Adam (All optimizers)
- Supervised Learning Rate: 0.001
- RL Learning Rate: 0.0005

- Batch Size: 256
- Hidden Layer Size: 256
- Latent Embedding Size: 256
- Nonlinearity: $Tanh()$

L.7 Cross-Entropy Method (CEM)

CEM search works as follows: we sample an initial latent program vector from the initial distribution D_I, and generate population of latent program vectors from a $\mathcal{N}(0, \sigma I_d)$ distribution, where I_d is the identity matrix of dimension d. The samples are added to the initial latent program vector to obtain the population of latent program vectors which are decoded into programs to obtain their rewards. The population is then sorted based on rewards obtained, and a set of 'elites' with the highest reward are reduced using weighted mean to one latent program vector for the next iteration of sampling. This process repeats for all CEM iterations.

We include the following sets of hyperparameters when searching over the program embedding space to maximize R_{mat} to reproduce ground-truth program behavior or to maximize R_{mat} in the Karel task MDP.

- Population Size (S): {8, 16, 32, 64}
- μ: $\{0.0\}$
- σ: $\{0.1, 0.25, 0.5\}$
- $\%$ of population elites (this refers to the percent of the population considered 'elites'): $\{0.05, 0.1, 0.2\}$
- Exponential σ decay[5]: {True, False}
- Initial distribution D_I: $\{\mathcal{N}(1, \mathbf{0}), \mathcal{N}(0, I_d), \mathcal{N}(0, 0.1 I_d)\}$

Since a comprehensive grid search over the hyperparameter space would be too computationally expensive, we choose parameters heuristically. We report results from the run with the best averaged reward over 5 seeds. Hyperparameters that performed best for each task are listed below.

Ground-Truth Program Reconstruction We include the following sets of hyperparameters when searching over the program embedding space to maximize R_{mat} to reproduce ground-truth program behavior. We allow the search to run for 1000 CEM iterations, counting the search as a success when it achieves 10 consecutive CEM iterations with matching the ground-truth program behaviors exactly in the environment across 10 random environment start configurations. We use same hyperparameter set to compare LEAPS-P, LEAPS-P+R, LEAPS-P+L, and LEAPS.

CEM	S	σ	# Elites	Exp Decay	D_I
WHILE	32	0.25	0.1	False	$\mathcal{N}(0, 0.1 I_d)$
IFELSE+WHILE	32	0.25	0.1	True	$\mathcal{N}(0, 0.1 I_d)$
2IF+IFELSE	16	0.25	0.2	True	$\mathcal{N}(0, 0.1 I_d)$
WHILE+2IF+IFELSE	32	0.25	0.2	False	$\mathcal{N}(0, 0.1 I_d)$

MDP Task Performance We include the following sets of hyperparameters when searching over the LEAPS program embedding space to maximize rewards in the MDP. We allow the search to run for 1000 CEM iterations, counting the search as a success when it achieves 10 consecutive CEM iterations of maximizing environment reward (solving the task) across 10 random environment start configurations.

[5]Over the first 500 epochs, we exponentially decay σ to 0.1, and then we keep it at 0.1 for the rest of the epochs if True.

CEM	S	σ	# Elites	Exp Decay	D_I
CLEANHOUSE	32	0.25	0.05	True	$\mathcal{N}(1, \mathbf{0})$
FOURCORNER	64	0.5	0.2	False	$\mathcal{N}(0, 0.1I_d)$
HARVESTER	32	0.5	0.1	True	$\mathcal{N}(0, I_d)$
MAZE	16	0.1	0.1	False	$\mathcal{N}(1, \mathbf{0})$
STAIRCLIMBER	32	0.25	0.05	True	$\mathcal{N}(0, 0.1I_d)$
TOPOFF	64	0.25	0.05	False	$\mathcal{N}(0, 0.1I_d)$

L.8 Random Search LEAPS Ablation

The random search LEAPS ablations (LEAPS-rand-8 and LEAPS-rand-64) replace the CEM search method for latent program synthesis with a simple random search method. Both use the full LEAPS model trained with all learning objectives. We sample an initial vector from an initial distribution D_I and add it to either 8 or 64 latent vector samples from a $\mathcal{N}(0, \sigma I_d)$ distribution. We then decode those vectors into programs and evaluate their rewards, and then report the rewards of the best-performing latent program from that population.

As such, the only parameters that we require are the initial sampling distribution and σ. We perform a grid search over the following for both LEAPS-rand-8 and LEAPS-rand-64.

- σ: $\{0.1, 0.25, 0.5\}$
- Initial distribution D_I: $\{\mathcal{N}(0, I_d), \mathcal{N}(0, 0.1I_d)\}$

Ground-Truth Program Reconstruction We report hyperparameters below for both random search methods on program reconstruction tasks.

LEAPS-rand-8	σ	D_I
WHILE	0.1	$\mathcal{N}(0, 0.1I_d)$
IFELSE+WHILE	0.5	$\mathcal{N}(0, 0.1I_d)$
2IF+IFELSE	0.5	$\mathcal{N}(0, 0.1I_d)$
WHILE+2IF+IFELSE	0.5	$\mathcal{N}(0, 0.1I_d)$

LEAPS-rand-64	σ	D_I
WHILE	0.5	$\mathcal{N}(0, 0.1I_d)$
IFELSE+WHILE	0.5	$\mathcal{N}(0, 0.1I_d)$
2IF+IFELSE	0.5	$\mathcal{N}(0, 0.1I_d)$
WHILE+2IF+IFELSE	0.5	$\mathcal{N}(0, 0.1I_d)$

MDP Task Performance We report hyperparameters below for both random search methods on Karel tasks.

LEAPS-rand-8	σ	D_I
CLEANHOUSE	0.5	$\mathcal{N}(0, 0.1I_d)$
FOURCORNER	0.5	$\mathcal{N}(0, 0.1I_d)$
HARVESTER	0.5	$\mathcal{N}(0, 0.1I_d)$
MAZE	0.25	$\mathcal{N}(0, 0.1I_d)$
STAIRCLIMBER	0.5	$\mathcal{N}(0, I_d)$
TOPOFF	0.25	$\mathcal{N}(0, 0.1I_d)$

LEAPS-rand-64	σ	D_I
CLEANHOUSE	0.5	$\mathcal{N}(0, 0.1I_d)$
FOURCORNER	0.25	$\mathcal{N}(0, 0.1I_d)$
HARVESTER	0.5	$\mathcal{N}(0, 0.1I_d)$
MAZE	0.1	$\mathcal{N}(0, 0.1I_d)$
STAIRCLIMBER	0.25	$\mathcal{N}(0, 0.1I_d)$
TOPOFF	0.5	$\mathcal{N}(0, 0.1I_d)$

M Computational Resources

For our experiments, we used both internal and cloud provider machines. Our internal machines are:

- M1: 40-vCPU Intel Xeon with 4 GTX Titan Xp GPUs
- M2: 72-vCPU Intel Xeon with 4 RTX 2080 Ti GPUs

The cloud instances that we used are either 128-thread AMD Epyc or 96-thread Intel Xeon based cloud instances with 4-8 NVIDIA Tesla T4 GPUs. Experiments were run in parallel across many CPUs whenever possible, thus requiring the high vCPU count machines.

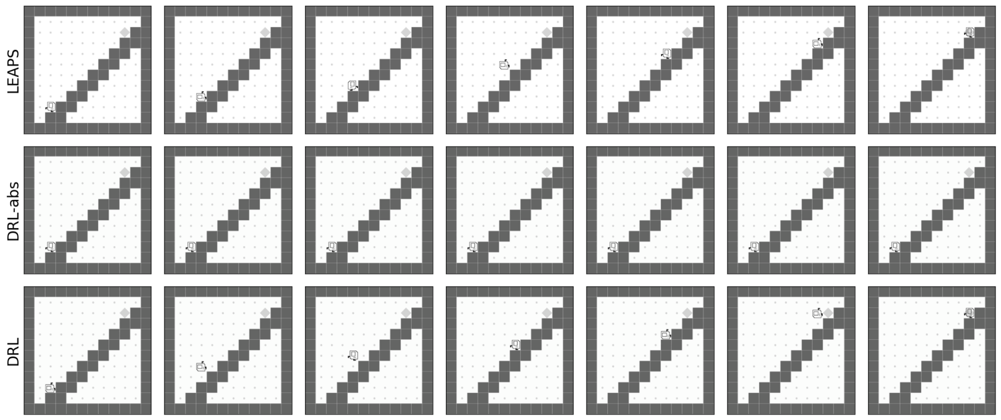
The experiment costs (GPU memory/time) are as follows:

Learning Program Embedding Stage:

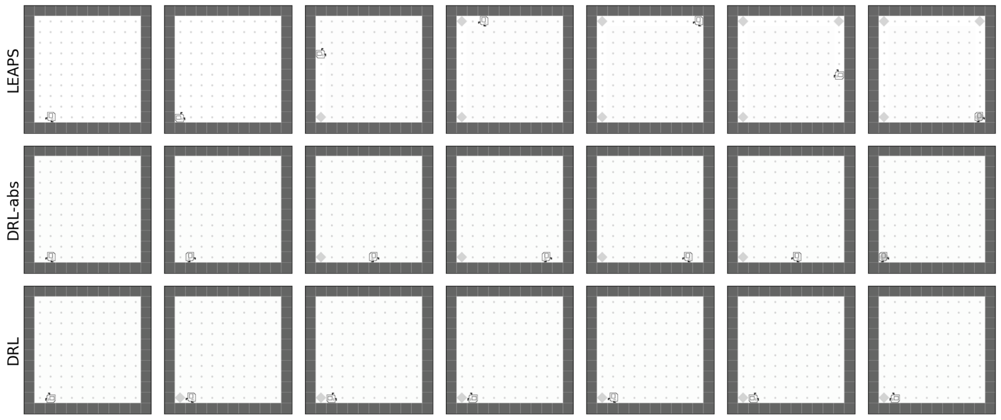
- LEAPS-P: 4.2GB/13hrs on either M1 or M2
- LEAPS-P+R: 4.2GB/44-54hrs on M2
- LEAPS-P+L: 8.7GB/26hrs on either M1 or M2
- LEAPS: 8.8GB/104hrs on M1, 8.8GB/58hrs on M2

Policy Learning Stage:

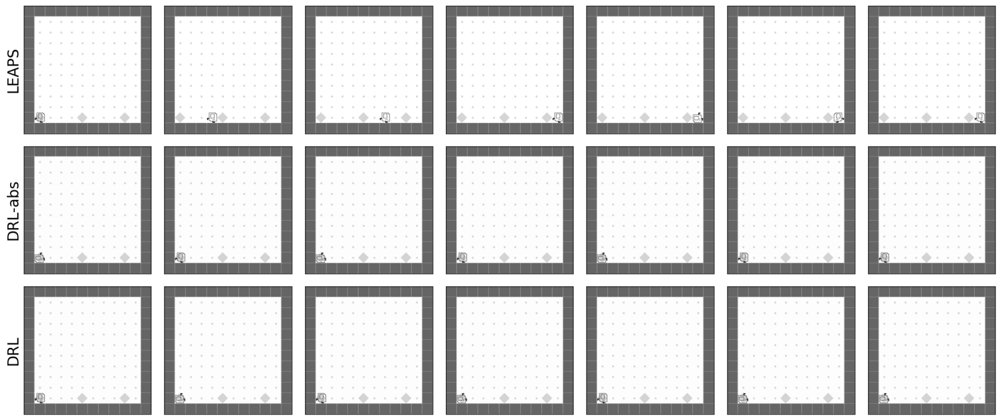
- CEM search: 0.8GB/4-10min (depends on the CEM population size and the number of iterations until convergence)
- DRL/DRL-abs/DRL-abs-t: 0.7-2GB/1hr per run with parallelization across 10 processes
- HRL/HRL-abs: 1-2GB/2.5hrs per run
- VIPER: 0.7GB/20-30 minutes (excluding the time for learning its teacher policy)



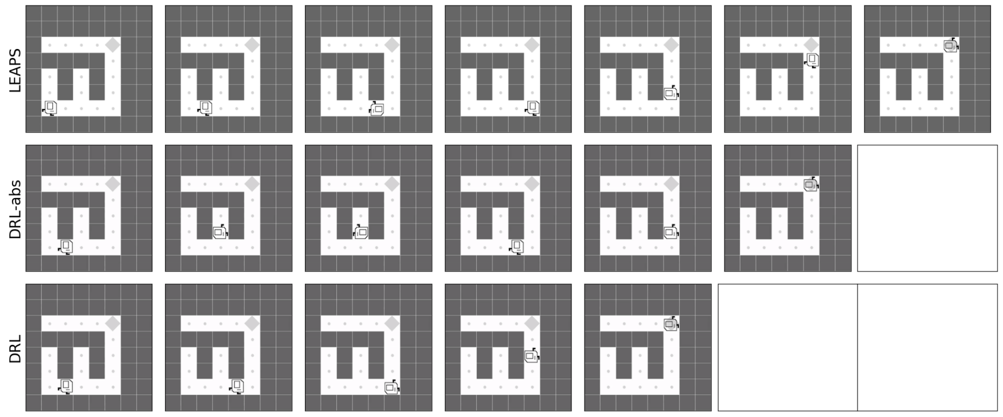
(a) STAIRCLIMBER: LEAPS and DRL are able to climb the stairs, DRL-abs is unable to do so.



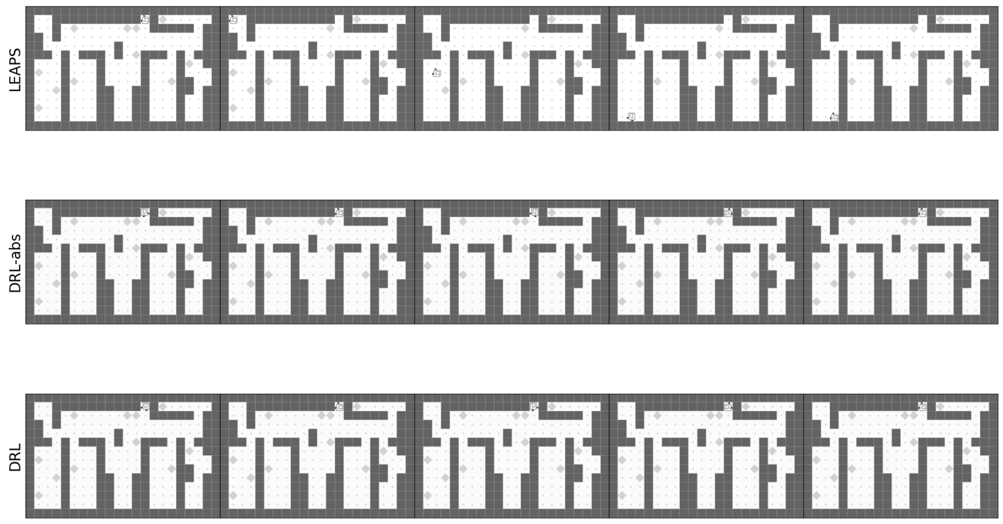
(b) FOURCORNER: In this example, LEAPS generates a program which is able to completely solve the task. Both DRL methods learn to only place one single marker in the left bottom corner.



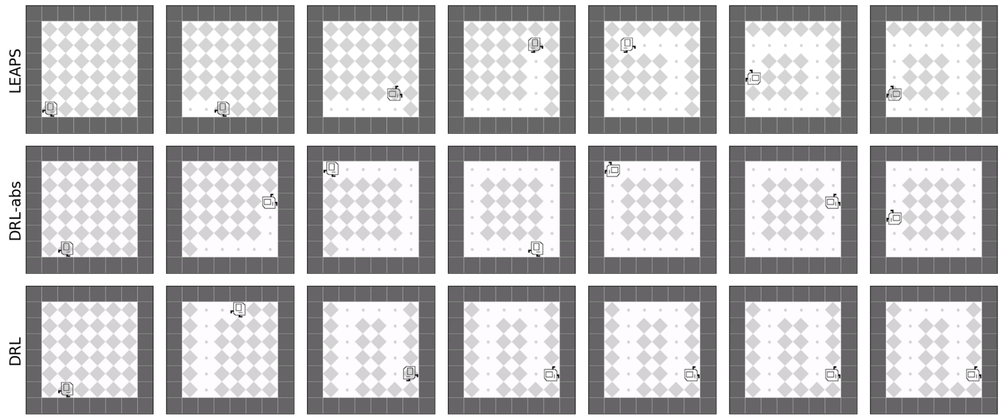
(c) TOPOFF: Here, LEAPS generates a program that solves the task by "topping off" each marker. Both DRL methods only learn to top off the initial marker.



(d) MAZE: All three methods are able to solve the task.



(e) CLEANHOUSE: While both DRL methods learn no meaningful behaviors (generally just spinning around in place), LEAPS generates a program that is able to navigate to and clean the leftmost room.



(f) HARVESTER: All three methods make partial progress on HARVESTER.

Figure 16: **Karel Rollout Visualizations.** Example rollouts for LEAPS, DRL-abs, and DRL for each task.