NEURAL PROGRAM SYNTHESIS WITH QUERY

Anonymous authors
Paper under double-blind review

ABSTRACT

Aiming to find a program satisfying the user intent given input-output examples, program synthesis has attracted increasing interest in the area of machine learning. Despite the promising performance of existing methods, most of their success comes from the privileged information of well-designed input-output examples. However, providing such input-output examples is unrealistic because it requires the users have the ability to describe the underlying program with several input-output examples under the actual distribution. In this work, we propose a query-based framework that trains a query neural network to generate informative input-output examples automatically and interactively. To optimize the framework, we propose the *functional space (F-space)* to represent input-output examples and the programs and model the similarity between them in a continuous and differentiable way. We evaluate the effectiveness and generalization of the proposed query-based framework on the Karel task and the list processing task. Experimental results show that the query-based framework can generate informative input-output examples which achieve and even outperform well-designed input-output examples of state-of-the-art methods.

1 INTRODUCTION

Program synthesis is the task of automatically finding a program that satisfies the user intent expressed in the form of some specifications like input-output examples (Gulwani et al., 2017). Recently, there has been an increasing interest in tackling it using neural networks in various domains, including string manipulation (Gulwani, 2011; Gulwani et al., 2012; Devlin et al., 2017b), list processing (Balog et al., 2017; Zohar & Wolf, 2018) and graphic applications (Ellis et al., 2018; 2019).

Despite their promising performance, most of their success relies on the well-designed input-output examples, without which the performance of the program synthesis model drops heavily. Traditionally, the input-output examples are selected elaborately. For example, in Karel (Devlin et al., 2017b; Bunel et al., 2018), the given input-output examples are required to have a high branch coverage ratio on the test program; In list processing, the given input-output examples are guided by constraint propagation (Balog et al., 2017) to ensure their effectiveness on the unknown program. However, providing such input-output examples is unrealistic because it requires the users to be experienced programmers to describe the underlying program with several input-output examples. Worse, the users must be familiar with the distribution of the training dataset to avoid input-output examples that are out-of-distribution (Shin et al., 2019). Some researchers have tried to solve this problem by search or random selection to find counter examples iteratively (Jha et al., 2010; Laich et al., 2020; Hajipour et al., 2020). However, they do not model the similarity between the input-output examples and programs accurately, and thus the quality of the examples cannot be guaranteed. In summary, how to generate informative input-output examples without expert experience is still an important problem that remains a challenge.

In this paper, we propose a query-based framework that can tackle the above-mentioned problems efficiently. Inspired by the interactive program synthesis, the query-based framework trains a query neural network to generate informative input-output examples by querying the underlying program iteratively. Specifically, we firstly train a query network with programs and then use it to generate informative input-output examples. After that, we use these generated examples to train the program synthesis network. This framework has three advantages: (1) There is no need for well-designed input-output examples in both training and testing, which leads to a good generalization. (2) The query network works in a generative manner, which can essentially reduce the computation cost

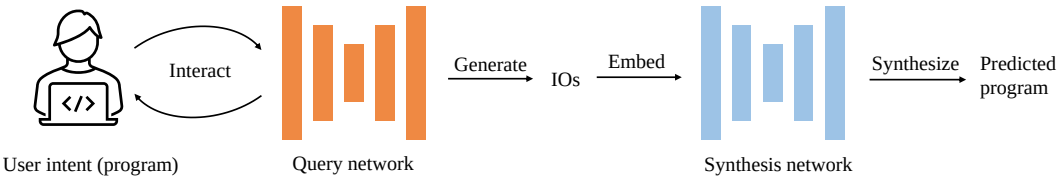


Figure 1: The query-based framework. The query network generates informative input-output examples by interacting with the user, and then the generated examples are sent to the synthesis network to synthesize the underlying program.

while facing problems with large input-output space. (3) The query network serves as a plug-and-play module with high scalability, for it is separate from the program synthesis process entirely.

To train the query network, we model the similarity between input-output examples and programs by introducing the *functional space* (F-space) and defining the distance between programs in it. Each program can be projected into a vector in F-space. The distance between two vectors indicates the functional difference between these two programs, so if two programs are projected into the same vector in the F-space, they are supposed to be functionally equivalent. Each input-output example corresponds to a set of programs that satisfy it, and thus it can be represented by a set of vectors in F-space. Furthermore, we project this set of vectors to a distribution (Sun & Nielsen, 2019) to make the representation continuous and differentiable, and thus can be optimized by the back-propagation algorithm. With the concept of information theory, we train the query network to generate queried input-output examples that maximize the probability of the corresponding program and minimize the probability of the others, resulting in the maximization of the mutual information between the queries and the corresponding programs. In this way, the query network can generate informative queried input-output examples efficiently.

We evaluate our method on the Karel task and the list processing task. Without utilizing the well-designed input-output examples both in training and testing, we achieve and even outperform the state-of-the-art performance on the Karel task and list processing task, which shows the effectiveness and generalization of our query-based framework.

2 Problem Formulation

The process of query is shown in Figure 1: The query network generates an input example each time, and then gets a corresponding output example from the oracle as the response. This process repeats until the limit of the number of queries is reached. Then, the generated input-output examples are sent to the synthesis network to finish the task of program synthesis. To lay a theoretical foundation of our query method, we give out the problem formulation as follows.

First, we define the functional equivalence, which is consistent with the definition of equivalence of functions in mathematics.

Definition 2.1 (Functional equivalence)**.** *Let $\mathbb{I}$ be the input example domain containing all valid input examples, $\mathbb{O}$ be the output example domain containing all possible output examples, and $\mathbb{P}$ be the program domain containing all valid programs under the domain specific language (DSL), each program $p \in \mathbb{P}$ can be seen as a function $p : \mathbb{I} \to \mathbb{O}$. For two programs $p_i \in \mathbb{P}$ and $p_j \in \mathbb{P}$, p_i and p_j are functional equivalent if and only if $\forall x \in \mathbb{I}, p_i(x) = p_j(x)$.*

Using the concept of functional equivalence, we can formulate the program synthesis task:

Definition 2.2 (Program synthesis)**.** *Suppose there is an underlying program $p \in \mathbb{P}$ and K input-output examples $[\![e]\!] = \{(x_k, y_k)|(x_k, y_k) \in \mathbb{I} \times \mathbb{O}\}^K$ generated by it, the program synthesis task aims to find a program $\hat{p}$ which is functional equivalent to p using $[\![e]\!]$.*

Following the definitions above, we can define the task of query.

Definition 2.3 (Query)**.** *The process of query is a map f_q. Given a program $p \in \mathbb{P}$ and a set of history K input-output examples $[\![e]\!] = \{(x_k, y_k)|(x_k, y_k) \in \mathbb{I} \times \mathbb{O}\}^K$, the process of query generates a query $x_{k+1} = f_q([\![e]\!]) \in \mathbb{I}$, and then get a response $y_{k+1} = p(x_{k+1}) \in \mathbb{O}$. The query and response are added to history input-output examples: $[\![e]\!] \leftarrow [\![e]\!] \cup \{(x_{k+1}, y_{k+1})\}$ and this process repeats.*

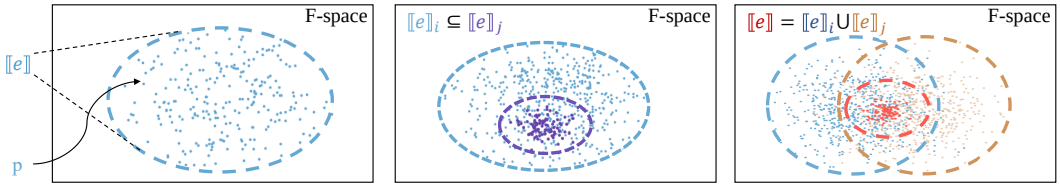


Figure 2: The illustration of F-space. Left: The projection of the input-output examples $[\![e]\!]$ and program p; Middle: The subset relationship between $[\![e]\!]_i$ and $[\![e]\!]_j$. Note that this relation is opposite in F-space; Right: The union operation of $[\![e]\!]_i$ and $[\![e]\!]_j$. This operation is also opposite in F-space where an union operation correspond to an intersection operation in F-space.

The query process aims to distinguish the underlying program with as few queries as possible.

3 METHODS

The query network and the synthesis network are trained separately. The query network is trained to distinguish different programs by querying, and after that, the synthesis network is trained to predict the corresponding program using the queried input-output examples. In the following, first, we introduce the F-space to model the similarity between the input-output examples and programs, and then, we illustrate how to train the query network with it.

3.1 F-SPACE

To model the query process with a neural network, we propose the *functional space (F-space)*.

Definition 3.1 (F-space and functional distance)**.** *F-space is a $|\mathbb{I}|$ dimensional space which consists of all valid programs that can be implemented by program domain $\mathbb{P}$. Each program is represented by $|\mathbb{I}|$ different output examples $\boldsymbol{v} = (y_1, y_2, \ldots, y_{|\mathbb{I}|})$. The distance in F-space can be measured by the number of different output examples: $d(\boldsymbol{v}_i, \boldsymbol{v}_j) = |diff(\boldsymbol{v}_i, \boldsymbol{v}_j)|$.*

Intuitively, F-space $(\mathbb{P}, d)$ measures the functional differences between programs. Each program can be represented by a vector in F-space, and if different programs are represented by the same vector $\boldsymbol{v} = (y_1, y_2, \ldots, y_{|\mathbb{I}|})$ in F-space, it indicates that these two programs get the same result for all inputs, and they are considered functional equivalent which is consistent with Definition 2.1. In practice, a space with dimension $|\mathbb{I}|$ is too large to compute. To solve this problem, we utilize the sparsity of F-space and learn an approximate space with dimension reduction by neural networks.

Regarding input-output examples, representing them by vectors is not appropriate. Consider a set of input-output examples $[\![e]\!] = \{(x_k, y_k)\}^K$, we cannot find a vector that represents it when $K < |\mathbb{I}|$ because there are more than one programs which satisfies these K input-output examples. For example, when there is only one input-output example $[\![e]\!] = \{(2, 4)\}$, we cannot distinguish the underlying program $y = 2x$ from $y = x^2$. Thus, when $K < |\mathbb{I}|$, the input-output examples $[\![e]\!] = \{(x_k, y_k)\}^K$ should be modeled as a set of vectors in F-space, projected by the set of programs that satisfy them.

We summarize the properties of the representation of input-output examples in F-space as follows:

- Each set of input-output examples $[\![e]\!] = \{(x_k, y_k)\}^K$ should be represented by a set of F-space vectors $[\![r]\!] = \{\boldsymbol{v}_n\}^N$.
- For two sets of input-output examples $[\![e]\!]_i$ and $[\![e]\!]_j$, if $[\![e]\!]_i \subseteq [\![e]\!]_j$, then their F-space representations $[\![r]\!]_i$ and $[\![r]\!]_j$ have the relationship $[\![r]\!]_i \supseteq [\![r]\!]_j$.
- For two sets of input-output examples $[\![e]\!]_i$ and $[\![e]\!]_j$, suppose $[\![e]\!]' = [\![e]\!]_i \cap [\![e]\!]_j$, then in F-space, their corresponding representations have the relationship $[\![r]\!]' = [\![r]\!]_i \cup [\![r]\!]_j$
- Following the property above, if $[\![e]\!]' = [\![e]\!]_i \cup [\![e]\!]_j$, then in F-space, their corresponding representations have the relationship $[\![r]\!]' = [\![r]\!]_i \cap [\![r]\!]_j$

Representing input-output examples by sets satisfies all these properties already. However, to optimize the representation using neural networks, we need to make it continuous and differentiable.

Thus, we model the representation of $[\![e]\!]$ in F-space as a Normal distribution (Ren & Leskovec, 2020; Sun & Nielsen, 2019), and the probability of the distribution indicates the possibility that it is the underlying program to be synthesized given input-output examples $[\![e]\!]$. We illustrate the projection, the subset relationship, and the union/intersection operation of distributions in Figure 2.

Under the circumstance of F-space, the query process becomes the process of reducing the uncertainty of the distribution. As described in Section 2, the query process aims to distinguish the underlying program with as few queries as possible. In F-space, the equivalent description is, the training of the query network aims to find F-space using as few queries as possible. To train the query network, we define several neural operators for the neural network training as follows.

Program projection. To project a program p to a vector $\boldsymbol{v}$, we traditionally use a sequence model as our encoder:

$$\boldsymbol{v} = Encoder_p(p). \tag{1}$$

Note that the program synthesis model differs largely on different datasets, so we choose to vary our program encoder according to the specific program synthesis models on different datasets. Mostly, our program encoder is the reverse of the program synthesis decoder.

Input-output example projection. Given one input-output example $[\![e]\!] = \{(x, y)\}$, we can project it into a Normal distribution using the same architecture of the corresponding program synthesis model, except that we add a MLP to output the two parameters of the Normal distribution: $\boldsymbol{\mu}$ and $log(\boldsymbol{\sigma}^2)$.

$$[\boldsymbol{\mu}, log(\boldsymbol{\sigma}^2)] = MLP_e(Encoder_e([\![e]\!])), \tag{2}$$

where MLP means a multi-layer perceptron. Using this projection, we can model $[\![e]\!]$ as $\mathcal{N}(\boldsymbol{\mu}, log(\boldsymbol{\sigma}^2))$ in F-space.

Input-output examples intersection. Given K input-output examples $[\![e]\!] = \{(x_k, y_k)\}^K$, each example $\{(x_k, y_k)\}$ is represented by a Normal distribution $\mathcal{N}(\boldsymbol{\mu}_k, log(\boldsymbol{\sigma}_k^2))$ using the projector above. The purpose of the intersection is to aggregate these Normal distributions into a new one, which represents $[\![e]\!]$ in F-space. Under the assumption of independence, the probability of the intersection distribution should be:

$$Pr_{intersection} = \prod_{i=1}^{K} Pr_i. \tag{3}$$

Fortunately, the product of independent Normal distributions is still a Normal distribution, which means that we can represent it in $[\boldsymbol{\mu}', log(\boldsymbol{\sigma}'^2)]$. In practice, we utilize the attention mechanism with another MLP to let the neural network learn the new Normal distribution (Ren & Leskovec, 2020):

$$[\boldsymbol{\mu}', log(\boldsymbol{\sigma}'^2)] = \sum_{i=1}^{K} w_i [\boldsymbol{\mu}_i, log(\boldsymbol{\sigma}_i^2)], \tag{4}$$

$$w_i = \frac{exp(MLP_{attention}([\boldsymbol{\mu}_i, log(\boldsymbol{\sigma}_i^2)]))}{\sum_{j=1}^{K} exp(MLP_{attention}(\boldsymbol{\mu}_j, log(\boldsymbol{\sigma}_j^2)))}. \tag{5}$$

This formulation not only keeps the form of distributions closed but also approximates the mode of the effective support of Normal distributions, which satisfies our requirement on the intersection of distribution (Sun & Nielsen, 2019).

Inverse projection to query. Finally, we need to generate the query from the representation of input-output examples in F-space. Using the projection and the intersection operation mentioned above, we can project the K input-output examples to the representation $[\boldsymbol{\mu}, log(\boldsymbol{\sigma}^2)]$. To generate query x_{new} from this representation, we introduce a decoder which has a similar architecture to $Encoder_e$ but reversed:

$$x_{new} = Decoder([\boldsymbol{\mu}, log(\boldsymbol{\sigma}^2)]). \tag{6}$$

Next, we will introduce how to train the neural network.

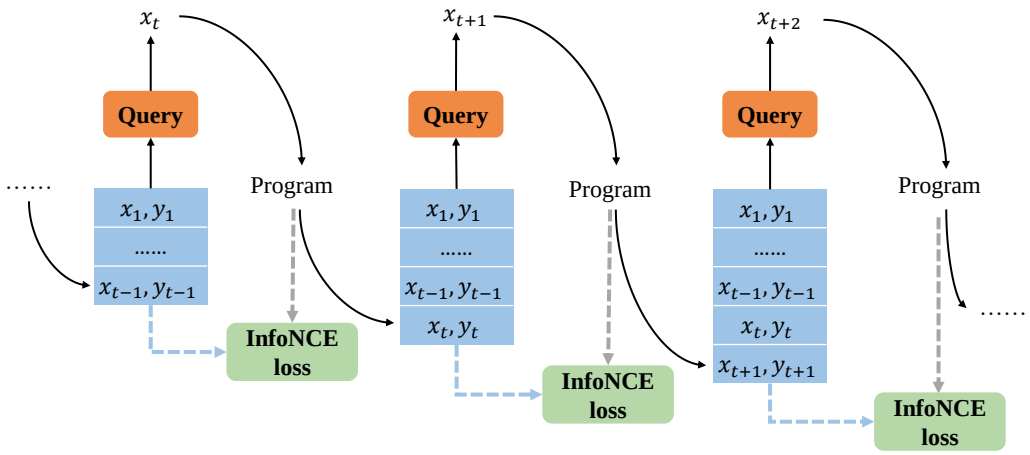


Figure 3: The RNN style training process.

Algorithm 1 Training process

function TRAIN( )
  Initialize max iterations N and max query times T
  for $i \in \{1 \dots N\}$ **do**
    $p \leftarrow Underlying\ programs$
    $(x_0, y_0) \leftarrow < sos >$
    $[\![e]\!] \leftarrow (x_0, y_0)$
    $L \leftarrow 0$
    for $t \in \{1 \dots T\}$ **do**
      $\boldsymbol{v} \leftarrow Encoder_p(p)$ ▷ Encode program, Equation (1)
      $\boldsymbol{\mu}, log(\boldsymbol{\sigma}^2) \leftarrow$ IO-ENCODER$([\![e]\!])$ ▷ Encode input-output examples
      $x_t \leftarrow Decoder(\boldsymbol{\mu}, log(\boldsymbol{\sigma}^2))$ ▷ Query next input
      $y_t \leftarrow p(x_t)$ ▷ Get next output from the oracle
      $[\![e]\!] \leftarrow [\![e]\!] \cup \{x_t, y_t\}$
      $\boldsymbol{\mu}', log(\boldsymbol{\sigma}'^2) \leftarrow$ IO-ENCODER$([\![e]\!])$
      $L \leftarrow Loss(\boldsymbol{v}, \boldsymbol{\mu}', log(\boldsymbol{\sigma}'^2)) + L$ ▷ InfoNCE loss, Equation (7)
    end for
    Update parameters w.r.t. L
  end for
end function

function IO-ENCODER$(\{x_k, y_k\}^K)$
  for $i \in \{1 \dots K\}$ **do**
    $[\boldsymbol{\mu}_i, log(\boldsymbol{\sigma}_i^2)] \leftarrow MLP_e(Encoder_e(\{x_i, y_i\})$ ▷ Encode single example, Equation (2)
  end for
  for $i \in \{1 \dots K\}$ **do**
    $w_i \leftarrow \frac{exp(MLP_{attention}([\boldsymbol{\mu}_i, log(\boldsymbol{\sigma}_i^2)]))}{\sum_{j=1}^{K} exp(MLP_{attention}(\boldsymbol{\mu}_j, log(\boldsymbol{\sigma}_j^2)))}$ ▷ Calculate attentions, Equation (5)
  end for
  $[\boldsymbol{\mu}, log(\boldsymbol{\sigma}^2)] = \sum_{i=1}^{K} w_i [\boldsymbol{\mu}_i, log(\boldsymbol{\sigma}_i^2)]$ ▷ Intersection, Equation (4)
  return $\boldsymbol{\mu}, log(\boldsymbol{\sigma}^2)$
end function

3.2 TRAINING

As mentioned above, we design the query network to fit into an encoder-decoder manner. The encoder learns to encode the input-output examples and the programs into F-space, and then the decoder decodes the representations into a new query. Furthermore, to generate queries successively, a Recurrent Neural Network (RNN) style architecture is used to generate queries progressively. As definition 2.3 says, each time when a query is generated, we feed it into the underlying program and get a response. Then, we add the query and response to the input-output examples and send it to the query network to get the next query. This process are shown in Figure 3 and Algorithm 1.

To optimize the query network, we utilize the technique in information theory to maximize the mutual information between the input-output examples $[\![e]\!]$ and the corresponding program p. Specifically, for a batch of data $\{([\![e]\!]_n, p_n), p_n\}^N$, we construct positive pairs $\{([\![e]\!]_i, p_i)\}$ and negative pairs $\{([\![e]\!]_i, p_j) | i \neq j\}$. Then, we use the InfoNCE loss (van den Oord et al., 2018) to maximize the lower bound of the mutual information between the input-output examples $[\![e]\!]$ and the corresponding program.

$$L = -log(\frac{exp(f([\![e]\!]_i, p_i))}{exp(f([\![e]\!]_i, p_i) + \sum_{i \neq j} exp(f[\![e]\!]_i, p_j)}), \tag{7}$$

where $f(\cdot, \cdot)$ denotes a relevance function. Intuitively, InfoNCE maximizes the relevance between positive pairs and minimizes the relevance between negative pairs. In F-space, we choose $f([\![e]\!]_i, p_j)$ to be the possibility of p_j in the $[\![e]\!]_i$'s distribution. That is, suppose the neural network project $[\![e]\!]_i$ to $\mathcal{N}(\boldsymbol{\mu}, log(\boldsymbol{\sigma}^2))$ and p_j to $\boldsymbol{v}$, then $f([\![e]\!]_i, p_j) = \mathcal{N}(\boldsymbol{v}; \boldsymbol{\mu}, log(\boldsymbol{\sigma}^2))$, which denotes the probability of $\boldsymbol{v}$ under the Normal distribution with parameters $\boldsymbol{\mu}$ and $log(\boldsymbol{\sigma}^2)$.

Additionally, like the task of sequence generation, we need an input-output example to act as the start signal $< sos >$. Datasets differ largely in program synthesis, so we design specific start signals for each dataset. We make our signal as simple as possible. For example, in Karel, the signal is designed to be an empty map with the robot at the center of the map; In list processing, the signal is just three lists full of *NULL*. More training details can be seen in Appendix A.

4 EXPERIMENT

We studied three problems in our experiment. (1) The metric that is reasonable in our query-based framework. (2) The performance of the query-based framework. (3) The generalization ability of the query-based framework. To do this, first, we demonstrate a comparison among several program synthesis metrics. Then, we present our results of the query on the Karel dataset and the list processing dataset to show our methods' performance and generalization ability.

4.1 METRICS

There are three metrics in neural program synthesis.

- **Semantics**: Given 5 input-output examples, if the predicted program satisfies all these examples, then it is semantically correct.
- **Generalization**: Given 5 input-output examples and a held-out input-output example, if the predicted program satisfies all 6 examples, then it is generally correct.
- **Exact match**: If the predicted program is the same as the ground-truth program, it matches the ground truth exactly.

Among them, exact match is the most strict metric, and generalization is more strict than semantics which means that if a program is generally correct, then it is also semantically correct. Exact match has its drawback in that the predicted program may be different from the ground-truth program, but they are functionally equivalent. Thus, the performance is measured by generalization on Karel and semantics on list processing. However, generalization is not appropriate here for two reasons: (1) It can only measure the performance of the program synthesis process instead of the query process. In the query process, the network chooses input-output examples independently, and judging them by the held-out example is meaningless. (2) Even for the program synthesis process, generalization is not strict enough. The program that satisfies a small set of input-output examples may fail on a larger set of input-output examples.

Thus, the best choice is to use *functional equivalence* as our metric. Unfortunately, the judgment of functional equivalent is a notoriously difficult problem which makes it hard to be applied in evaluation. Thus, we generate 95 held-out input-output examples randomly to achieve a higher branch coverage than 1 held-out example, and make the generalization on these examples a proxy of the functional equivalence:

- **Functional equivalence (proxy)**: Given 5 input-output examples and 95 held-out input-output examples, if the predicted program satisfies all 100 examples, then it is functional equivalent to the ground-truth program.

Table 2: The performance of the program synthesis model trained on input-output examples generated by different methods on the Karel task.

Metric	Bunel et al. (2018)			Chen et al. (2019)		
	Random	Well-designed	Query	Random	Well-designed	Query
Exact match	29.44%	41.16%	41.12%	26.08%	37.36%	38.68%
Functional equivalence	33.08%	48.52%	46.64%	32.28%	47.60%	48.48%

To illustrate the difference between functional equivalence and generalization further, we measured the average branch coverage of these two metrics on Karel's validation set. Given a set of input-output examples, branch coverage evaluates what is the percentage of program branches that are covered by the examples.

The result is shown in Table 1. Functional equivalence outperforms generalization by nearly 10%, which indicates that functional equivalence can represent the correctness of the predicted program much better than generalization.

	Semantics	Generalization	Functional
Branch coverage	86.57%	87.99%	97.58%

Table 1: The branch coverage of semantics, functional equivalence, and generalization.

4.2 KAREL TASK

Karel is an educational programming language used to control a robot living in a 2D grid world (Pattis et al., 1981). The domain specific language (DSL) and other details are included in Appendix B.1. Specifically, the DSL of Karel includes control flow constructs, which make this task more challenging than others. More details can be seen in Appendix B.1.

Settings. Following Section 3, we split the training process into the training of the query network and the training of the synthesis network. For the query network, we set the $Encoder_e$ the same as the one of Bunel et al. (2018) except for the output layer, and the $Encoder_p$ a two-layer LSTM. For the synthesis network, all settings stay unchanged as in Bunel et al. (2018). For all results, we use greedy decoding instead of beam search. As in Section 4.1, exact match and functional equivalence are more appropriate than other metrics in the query task, so we save the model based on them instead of the generalization metric, and this may cause differences in our baseline reproduction.

We generate the dataset with 5 input-output examples using three different methods: randomly selected, baseline models (well-designed), and our method (query). Then, we train the synthesis network on these datasets with two state-of-the-art methods: Bunel et al. (2018) and Chen et al. (2019). Note that there is a slight difference between the program simulators of them, and we choose the one of Bunel et al. (2018) as ours.

Performance. Table 2 presents the performance of the trained synthesis networks, from which we can conclude that (1) The Query dataset performs well on both two training methods, indicating that the query process is totally decoupled with the synthesis process and has a high scalability. (2) Our query method gets comparable results with both baseline methods and largely outperforms the random selection method with more than 10%, which shows its effectiveness.

The number of queries. We also present the relationship between the performance and the number of queries in Figure 4. As the number of queries increases, the performance of the program synthesis improves. Note that the first two queries play essential roles in the whole query process, consistent with the intuition that the queries selected at the beginning contain more information.

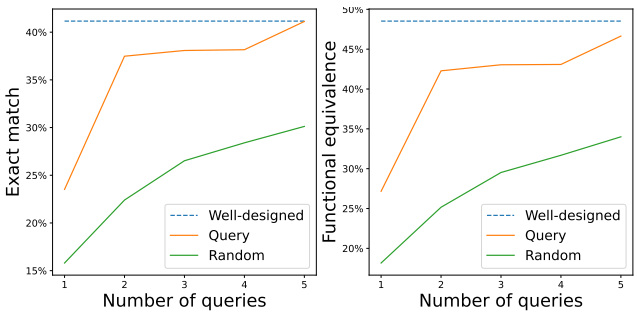


Figure 4: The query process while inference.

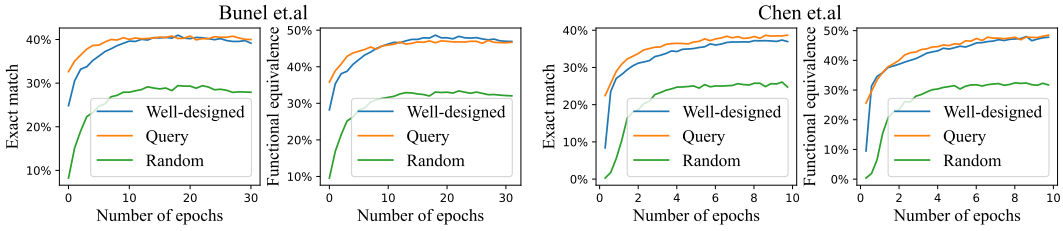


Figure 5: The training curve of the program synthesis model on Karel.

Table 3: The performance of the program synthesis model trained on input-output examples generated by different methods on the list processing task.

Dataset	Metric	Searching for semantics			Searching for exact match		
		Random	Well-designed	Query	Random	Well-designed	Query
D_1	Exact match	20.07%	32.81%	22.26%	50.65%	81.21%	81.56%
	Functional equivalence	52.03%	80.26%	68.20%	50.65%	81.21%	81.56%
D_2	Exact match	9.25%	15.72%	10.44%	22.61%	38.51%	38.63%
	Functional equivalence	24.49%	39.88%	32.94%	22.61%	38.51%	38.63%

Training process. To do a further study, we plot the training curve in Figure 5. It is shown that the Query dataset always has a quick start, which indicates that the query method can extract the features of the distinguishable programs effectively and makes them easier to be synthesized.

4.3 List Processing Task

To show the generalization ability of the query method, we conduct another experiment on the list processing task. The list processing task takes 1-3 lists or integers as input examples, and then produces a list or an integer as the output example. More details can be seen in Appendix B.2. This task is studied by Balog et al. (2017) and Zohar & Wolf (2018) using the search method guided by a neural network, and our experiment is based on their works.

Settings. Following PCCoder (Zohar & Wolf, 2018), we generate two datasets with program length 4 as dataset D_1 and program length up to 12 as dataset D_2. We set the $Encoder_e$ of the query network similar to PCCoder with the output layer changed, and use a single layer LSTM as the $Encoder_p$. The synthesis network and the parameters of complete anytime beam search (CAB) (Zhang, 1998) stay the same as PCCoder, except that the maximum time is set to 5 seconds instead of 5,000 seconds for reality.

Similar to the Karel task, we generate the dataset with 5 input-output examples with three different methods. Furthermore, we use two end conditions of CAB: The searching ends when all input-output examples are satisfied (searching for semantics), and the searching ends when all statements are true (searching for exact match).

Performance. The results are presented in Table 3, from which we can conclude that: (1) Our query method results higher than well-designed input-output examples in searching for exact match, and consistently outperforms the random. (2) When the length of programs increases, the performance decreases largely. This is resulted from the original algorithm of PCCoder that the intermediate variables are difficult to be saved when the ground-truth program is long. However, the query method decreases slower than others, and the gap between the well-designed and the query is closed largely.

5 Related Work

5.1 Program Synthesis

Programming by examples. Synthesizing a program that satisfies the user intent using the provided input-output examples is a challenging problem that has been studied for years (Manna & Waldinger, 1971; Lieberman, 2001; Solar-Lezama et al., 2006; Gulwani, 2011; Gulwani et al., 2012).

Recently, with the development of Deep Learning, more researchers tend to tackle this problem with neural networks in a variety of tasks, including string transformation (Devlin et al., 2017b), list processing (Balog et al., 2017; Zohar & Wolf, 2018), graphic generation (Ellis et al., 2018; Tian et al., 2019), Karel (Devlin et al., 2017a; Bunel et al., 2018), policy abstraction (Sun et al., 2018; Verma et al., 2018) and so on. Additionally, techniques like program debugger (Balog et al., 2020; Gupta et al., 2020), traces (Chen et al., 2019; Shin et al., 2018; Ellis et al., 2019), property signatures (Odena & Sutton, 2020; Odena et al., 2021) are also used to improve the performance of program synthesis. However, their promising performance relies on the well-designed input-output examples which is a high requirement for users. If examples with poor quality are provided, the performance will be affected severely. Worse, if the input-output examples that are out-of-distribution are provided, the program synthesis model cannot finish the task as expected (Shin et al., 2019).

Interactive program synthesis. Considering the high requirement of input-output examples, Le et al. (2017) build an abstract framework in which the program synthesis system can interact with users to guide the process of synthesis. Following this framework, multiple interactive synthesis systems are studied. Among them, Mayer et al. (2015), Wang et al. (2017), and Laich et al. (2020) tend to find counter-examples as queries to the users in a random manner. Although they get rid of the restrictions on the users, the quantity of the input-output examples is not guaranteed, which may get the synthesis process into trouble. Padhi et al. (2018) select more than one query each time and let the users choose which one to answer. This brings an additional burden on the users, and the users are unaware of which query will improve the program synthesized most. Most recently, Ji et al. (2020) utilizes the minimax branch to select the question where the worst answer gives the best reduction of the program domain. Theoretically, the performance of the selection is guaranteed. However, this method is based on search and hardly be applied to tasks with large input-output space. Worse, the query process and the synthesis process are bound, which results in its poor scalability. In contrast, our method split the process of query and the process of synthesis, and thus make their combination more flexible.

5.2 Active Learning

Active learning is a research domain that aims to reduce the cost of labeling by selecting most representative samples iteratively from the unlabeled dataset, and then asking them to an oracle for labeling while training. According to Shui et al. (2020), active learning can be divided into pool-based sampling (Angluin, 1988; King et al., 2004), which judges whether a sample should be selected for query-based on the evaluation of the entire dataset; steam-based sampling (Dagan & Argamon, 1995; Krishnamurthy, 2002), which judges each sample independently compared to pool-based sampling; and membership query synthesis (Lewis & Gale, 1994), which means that the unlabeled sample can be generated by the learner instead of selecting from the dataset only. Although active learning involves querying an oracle iteratively, which is similar to our framework, there are still two main differences between them. (1) For the final purpose, active learning aims to finish the training stage at a low cost, and the inference stage remains the same as other deep learning tasks without the query process, while our framework aims to find the oracle (*i.e.*the underlying program) in a symbolic form, and the query process is retained during inference. (2) For the query process, active learning assumes only one oracle (*i.e.*the same query always gets the same label). In contrast, our framework assumes multiple oracles (*i.e.*the same query will get different responses if the underlying programs are different).

6 Conclusion

In this work, we propose a query-based framework to finish the program synthesis task more realistically. Moreover, to model the relationship between the input-output examples and programs, we establish a theoretical foundation by formulating the query problem and introducing the F-space. In F-space, we represent input-output examples and the programs and model their similarity in a continuous and differentiable way. Using these techniques, we conduct a series of experiments that shows the effectiveness, generalization, and scalability of our query-based framework. We believe that our methods work not only on the program synthesis tasks, but also on any task that aims to simulate an underlying oracle, including reverse engineering, symbolic regression, scientific discovery, and so on.

REFERENCES

D. Angluin. Queries and concept learning. *Machine Learning*, 2:319–342, 1988.

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, S. Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. In *ICLR (Poster)*. OpenReview.net, 2017.

Matej Balog, Rishabh Singh, Petros Maniatis, and Charles Sutton. Neural program synthesis with a differentiable fixer. *ArXiv*, abs/2006.10924, 2020.

Rudy Bunel, M. Hausknecht, J. Devlin, Rishabh Singh, and P. Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. In *ICLR (Poster)*. OpenReview.net, 2018.

Xi Chen, Yan Duan, Rein Houthooft, J. Schulman, Ilya Sutskever, and P. Abbeel. Infogan: Interpretable representation learning by information maximizing generative adversarial nets. In *NIPS*, 2016.

Xinyun Chen, Chang Liu, and D. Song. Execution-guided neural program synthesis. In *ICLR*, 2019.

Ido Dagan and S. Argamon. Committee-based sampling for training probabilistic classifiers. In *ICML*, 1995.

J. Devlin, Rudy Bunel, Rishabh Singh, M. Hausknecht, and P. Kohli. Neural program meta-induction. In *NIPS*, 2017a.

J. Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel rahman Mohamed, and P. Kohli. Robustfill: Neural program learning under noisy i/o. In *ICML*, 2017b.

Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and J. Tenenbaum. Learning to infer graphics programs from hand-drawn images. In *NeurIPS*, 2018.

Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, J. Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a repl. In *NeurIPS*, 2019.

S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL '11*, 2011.

Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55:97–105, 2012.

Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Found. Trends Program. Lang.*, 4:1–119, 2017.

Kavi Gupta, P. E. Christensen, Xinyun Chen, and D. Song. Synthesize, execute and debug: Learning to repair for neural program synthesis. In *NeurIPS*, 2020.

Hossein Hajipour, Mateusz Malinowski, and Mario Fritz. Ireen: Iterative reverse-engineering of black-box functions via neural program synthesis. *ArXiv*, abs/2006.10720, 2020.

Eric Jang, Shixiang Shane Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. In *ICLR (Poster)*. OpenReview.net, 2017.

Susmit Jha, Sumit Gulwani, S. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. *2010 ACM/IEEE 32nd International Conference on Software Engineering*, 1:215–224, 2010.

Ruyi Ji, Jingjing Liang, Yingfei Xiong, Lu Zhang, and Zhenjiang Hu. Question selection for interactive program synthesis. *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020.

R. King, Ken E. Whelan, F. M. Jones, Philip G. K. Reiser, Christopher H. Bryant, S. Muggleton, D. Kell, and S. Oliver. Functional genomic hypothesis generation and experimentation by a robot scientist. *Nature*, 427:247–252, 2004.

V. Krishnamurthy. Algorithms for optimal scheduling and management of hidden markov model sensors. *IEEE Trans. Signal Process.*, 50:1382–1397, 2002.

Larissa Laich, Pavol Bielik, and Martin T. Vechev. Guiding program synthesis by learning to generate examples. In *ICLR*, 2020.

Vu Le, Daniel Perelman, Oleksandr Polozov, Mohammad Raza, A. Udupa, and S. Gulwani. Interactive program synthesis. *ArXiv*, abs/1703.03539, 2017.

D. Lewis and W. Gale. A sequential algorithm for training text classifiers. In *SIGIR '94*, 1994.

H. Lieberman. Your wish is my command: Programming by example. 2001.

Chris J. Maddison, Andriy Mnih, and Yee Whye Teh. The concrete distribution: A continuous relaxation of discrete random variables. In *ICLR (Poster)*. OpenReview.net, 2017.

Z. Manna and R. Waldinger. Toward automatic program synthesis. *Commun. ACM*, 14:151–165, 1971.

Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, B. Zorn, and S. Gulwani. User interaction models for disambiguation in programming by example. *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, 2015.

Augustus Odena and Charles Sutton. Learning to represent programs with property signatures. In *ICLR*. OpenReview.net, 2020.

Augustus Odena, Kensen Shi, David Bieber, Rishabh Singh, and Charles Sutton. Bustle: Bottom-up program-synthesis through learning-guided exploration. In *ICLR*. OpenReview.net, 2021.

Saswat Padhi, Prateek Jain, Daniel Perelman, Oleksandr Polozov, S. Gulwani, and T. Millstein. Flashprofile: a framework for synthesizing data profiles. *Proceedings of the ACM on Programming Languages*, 2:1 – 28, 2018.

Richard Pattis, J Roberts, and M Stehlik. Karel the robot. *A gentele introduction to the Art of Programming*, 1981.

Hongyu Ren and J. Leskovec. Beta embeddings for multi-hop logical reasoning in knowledge graphs. In *NeurIPS*, 2020.

Richard Shin, Illia Polosukhin, and D. Song. Improving neural program synthesis with inferred execution traces. In *NeurIPS*, 2018.

Richard Shin, Neel Kant, Kavi Gupta, Christopher M. Bender, Brandon Trabucco, Rishabh Singh, and D. Song. Synthetic datasets for neural program synthesis. In *ICLR (Poster)*. OpenReview.net, 2019.

Changjian Shui, Fan Zhou, Christian Gagn'e, and B. Wang. Deep active learning: Unified and principled method for query and training. In *AISTATS*, 2020.

Armando Solar-Lezama, Liviu Tancau, R. Bodík, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS XII*, 2006.

Ke Sun and F. Nielsen. Information-geometric set embeddings (igse): From sets to probability distributions. *ArXiv*, abs/1911.12463, 2019.

Shao-Hua Sun, Hyeonwoo Noh, S. Somasundaram, and Joseph J. Lim. Neural program synthesis from diverse demonstration videos. In *ICML*, 2018.

Yonglong Tian, Andrew Luo, Xingyuan Sun, Kevin Ellis, W. Freeman, J. Tenenbaum, and Jiajun Wu. Learning to infer and execute 3d shape programs. In *ICLR (Poster)*. OpenReview.net, 2019.

Aäron van den Oord, Yazhe Li, and Oriol Vinyals. Representation learning with contrastive predictive coding. *ArXiv*, abs/1807.03748, 2018.

Abhinav Verma, V. Murali, Rishabh Singh, P. Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In *ICML*, volume 80 of *Proceedings of Machine Learning Research*, pp. 5052–5061. PMLR, 2018.

Chenglong Wang, Alvin Cheung, and R. Bodík. Interactive query synthesis from input-output examples. *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017.

Weixiong Zhang. Complete anytime beam search. In *AAAI/IAAI*, 1998.

Amit Zohar and Lior Wolf. Automatic program synthesis of long programs with a learned garbage collector. In *NeurIPS*, 2018.

A TRAINING DETAILS

A.1 MODEL DETAILS AND HYPER PARAMETERS

Karel. The query encoder is the same as the one used by Bunel et al. (2018) composed of a Convolutional Neural Network (CNN) with the residual link. After the query encoder, there is an MLP to project the embedding into $\boldsymbol{\mu}, log(\boldsymbol{\sigma}^2)$. The dimension of $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ is set to 256, and thus the hidden size of MLP is 512. The query decoder is similar to the query encoder except that the number of channels is reversed, and additional batch normalization is added to keep the generation more stable. The program encoder is a two-layer LSTM with a hidden size 256.

For training, the learning rate of the query network is set to 10^{-4} while the learning rate of the synthesis network stays the same with the original methods. The batch size is 128, and the random seed is set to 100.

List processing. The query encoder is the same as the one in PCCoder. The dimension of $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ is set to 256 for dataset D_1 and 128 for dataset D_2 without much tuning. The query decoder is a single-layer linear network with dimension 256 for dataset D_1 and 128 for dataset D_2. The program encoder is a single layer LSTM with hidden size 256 for dataset D_1 and 128 for dataset D_2.

For training, the learning rate of the query network is set to 10^{-4} with a 0.1 decay every 40 epochs. The learning rate of the synthesis network is 10^{-3}, which stays the same with the original methods with 0.1 decay every 4 epochs. The batch size of the query process is 64, the batch size of synthesis is 32 for D_1 and 100 for D_2. The random seed is set to 100.

A.2 OTHER TRAINING TECHNIQUES

Latent code. The subsequent queries are easy to fail in a mode collapse, generating similar queries repetitively. To tackle this, we introduce a latent code like InfoGAN (Chen et al., 2016) indicating the query step as another input and ask the encoder in the next query step to decode it accurately.

Kullback-Leibler (KL) divergence. Similar to the latent code, we add the reciprocal of the KL divergence between different queries on the same program as another loss to make the queries more diverse. However, this loss does not have a significant impact on training most of the time and makes the training process unstable.

Gumbel softmax. Note that the queries are discrete, which makes the query network cannot be optimized by back-propagation. Thus, we take advantage of the Gumbel-Softmax distribution (Maddison et al., 2017; Jang et al., 2017) and make the query process differentiable.

Curriculum learning. The query network is trained progressively. At the beginning of the training, the query network generates one query only. As the training goes on, the number of query times increases until it achieves the query time limit.

B EXPERIMENT DETAILS

B.1 THE KAREL TASK

Karel is an educational programming language, which can be used to control a robot to move in a 2D grid world (Pattis et al., 1981). Figure 6 presents the domain language specification (DSL) of

Karel. Figure 7 shows a Karel program example. Each cell of the grid world is represented as a 16-dimensional vector corresponding to the features described in Table 4 (Bunel et al., 2018).

Note that in the well-designed dataset, the size of the grid world can be changed from 2 to 16. However, for the convenience of training, we set the size of the query world to 16. Moreover, to guarantee that every query can be recognized by the program simulator, we design the output layer of the query network elaborately. Even the query network is initialized randomly at the beginning of training, it can still generate recognizable queries.

To show more details of our queried examples, we present two examples in Figure 10 and Figure 11. Each example consists of the underlying program and the corresponding queries that can cover all its branches.

$$
\begin{array}{rcl}
\text{Prog } p & := & \texttt{def run}() \ : \ s \\
\text{Stmt } s & := & \texttt{while}(b) : s \mid \texttt{repeat}(r) : s \mid s_1; s_2 \mid a \\
 & \mid & \texttt{if}(b) : s \mid \texttt{ifelse}(b) : s_1 \ \texttt{else} : s_2 \\
\text{Cond } b & := & \texttt{frontIsClear}() \mid \texttt{leftIsClear}() \mid \texttt{rightIsClear}() \\
 & \mid & \texttt{markersPresent}() \mid \texttt{noMarkersPresent}() \mid \texttt{not } b \\
\text{Action a} & := & \texttt{move}() \mid \texttt{turnRight}() \mid \texttt{turnLeft}() \\
 & \mid & \texttt{pickMarker}() \mid \texttt{putMarker}() \\
\text{Cste } r & := & 0 \mid 1 \mid \cdots \mid 19
\end{array}
$$

Figure 6: The DSL of Karel

Hero facing North
Hero facing South
Hero facing West
Hero facing East
Obstacle
Grid boundary
1 marker
2 marker
3 marker
4 marker
5 marker
6 marker
7 marker
8 marker
9 marker
10 marker

Table 4: Representation of a cell in grid world

B.2 THE LIST PROCESSING TASK

The list processing task takes 1-3 lists or integers as input examples and produces a list or an integer as the output example. An example is shown in Figure 9. Figure 8 shows the DSL of list processing. Following Zohar & Wolf (2018), we generate two datasets: D_1 with program length 4 and D_2 with program length up to 12.

For the well-designed input-output examples, the length of the list is sampled stochastically and can be changed under 20. However, to make the training simpler, we pad the length of queries to 20. For example, if a query is a list with length 8, the query network should output a list with length 20, and the first 8 elements have their meaning while the last 12 elements are filled with $NULL$.

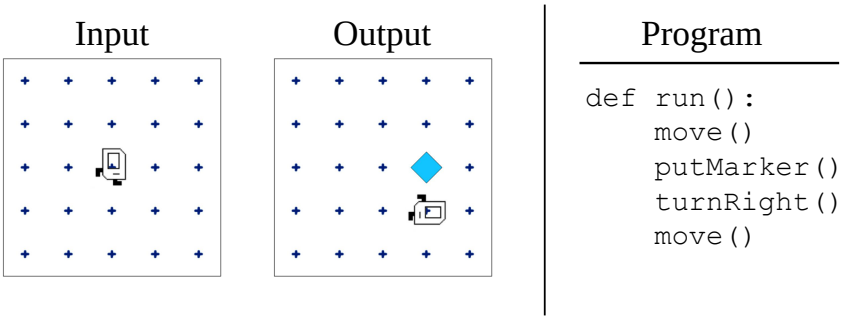


Figure 7: An example of Karel.

Basic Function	:=	$+1 \mid -1 \mid \times 2 \mid \div 2 \mid \times(-1)$
	|	$**2 \mid \times 3 \mid \div 3 \mid \times 4 \mid \div 4$
	|	$>0 \mid <0 \mid \%2 \mid \%2 == 1$
First-order Function	:=	HEAD | LAST | TAKE | DROP | ACCESS
	|	MINIMUM | MAXIMUN | REVERSE | SORT | SUM
Higher-order Function	:=	MAP | FILTER | COUNT | ZIPWITH | SCANL1

Figure 8: The DSL of list processing

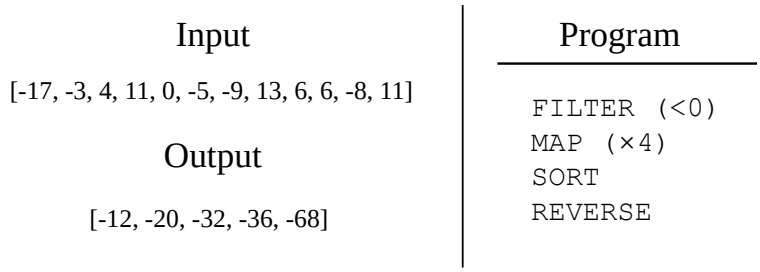
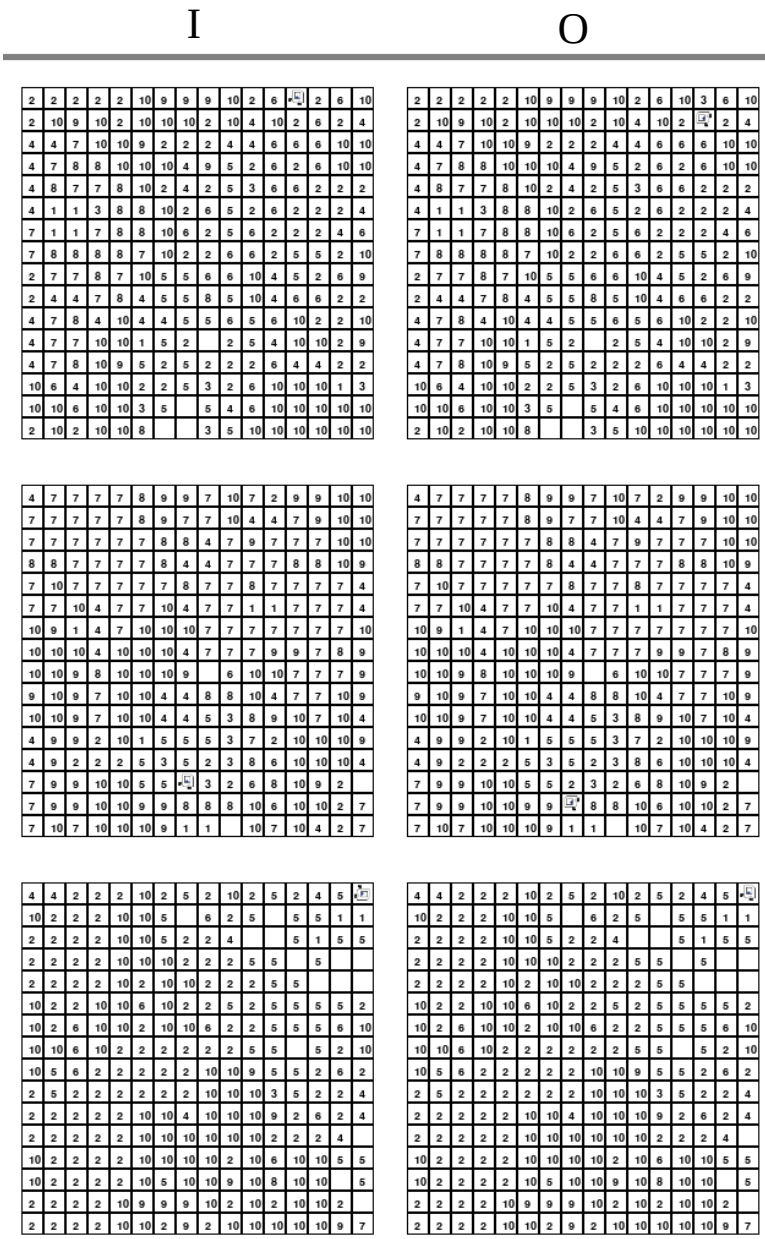


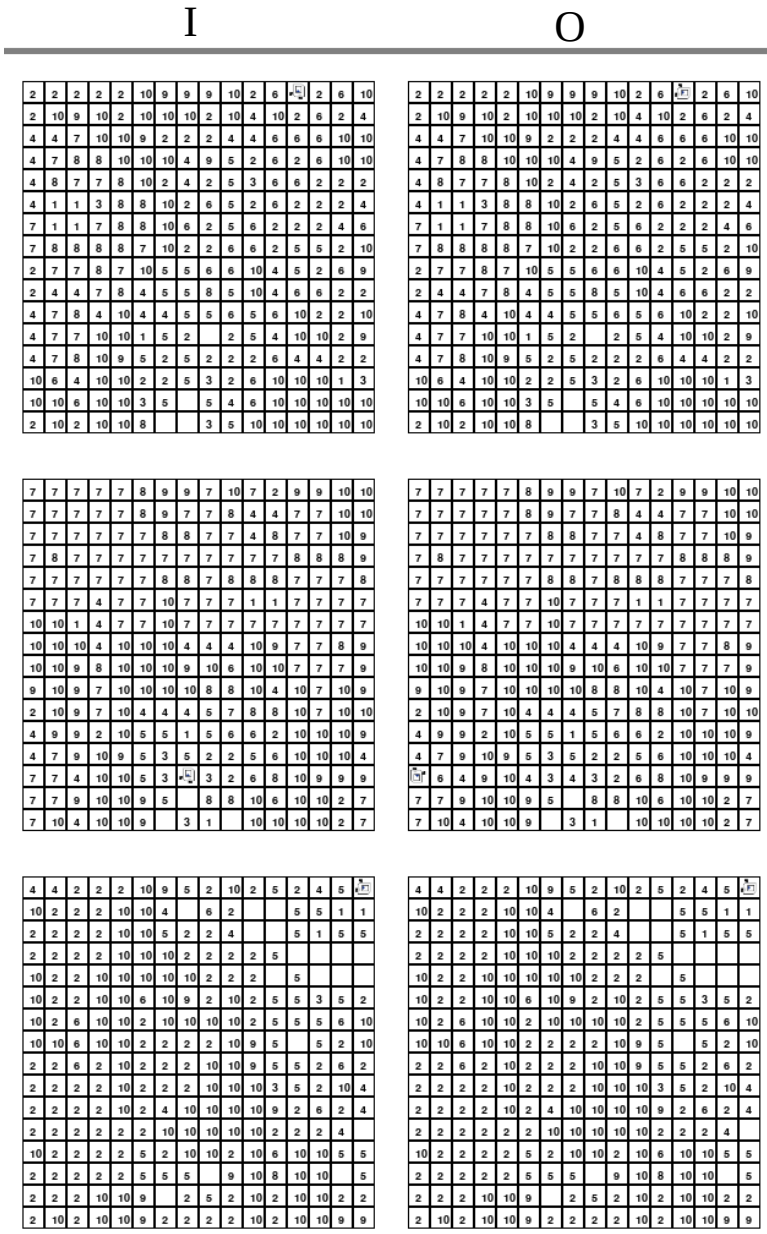
Figure 9: An example of list processing.



Program

```
def run():
   if markersPresent:
        putMarker
        if not rightIsClear:
            putMarker
            move
            pickMarker
        turnRight
        repeat R=2:
            turnLeft
        putMarker
        putMarker
        move
        putMarker
```

Figure 10: An query example of Karel. The number in the cell denotes the amount of markers.



Program

```
def run():
    repeat R=2:
        putMarker
        turnRight
        if frontIsClear:
            putMarker
            putMarker
            turnRight
        putMarker
        while rightIsClear:
            move
            move
            pickMarker
        turnRight
    turnLeft
    turnLeft
    move
    putMarker
    turnLeft
```

Figure 11: Another query example of Karel. The number in the cell denotes the amount of markers.