Towards Synthesizing Qualitative and Diverse Programs for Block-Based Visual Programming

by

Akshay Dodwadmath

Master Thesis

Department of Computer Science
Saarland University

supervised by

Dr. Adish Singla

Reviewers

Dr. Adish Singla
Dr. Goran Radanović

July 28, 2023

Declaration of Authorship

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Datum/Date: ______________________________

Unterschrift/Signature: ______________________________

SAARLAND UNIVERSITY
Department of Computer Science

Abstract

Towards Synthesizing Qualitative and Diverse Programs for Block-Based Visual Programming
by Akshay Dodwadmath

Block-based visual programming environments are increasingly being used to teach K-12 students basic programming concepts, as well as to help them develop computational thinking. However, these environments often have a limited number of practice tasks, which hinders student learning. To solve this pedagogical challenge, we formalize the problem of synthesizing good quality codes, that can be used to produce new practice tasks. In particular, given an input code specification containing details such as code sketch and desired code length $(\mathrm{S}^{\mathrm{in}}, \mathrm{L}^{\mathrm{in}})$, we propose a methodology based on neural techniques, to synthesize a diverse set of good quality codes $\{\mathrm{C}^{\mathrm{out}}\}$. Our methodology is motivated by the fact that, it is challenging to handcraft all the rules that can encapsulate the features of good quality codes; hence using a rule-based approach is futile. Instead, our neural code generation algorithm operates by first performing behavior cloning on a set of expert generated codes. Then, the algorithm uses reinforcement learning with quality based rewards, to improve the quality of the generated codes. We demonstrate the effectiveness of our algorithm through an extensive empirical evaluation on a set of Karel specifications, which are based on *Intro to Programming with Karel* course by *CodeHS.com*.

Acknowledgements

First and foremost, I would like to thank Dr. Adish Singla, for giving me the opportunity to work under his supervision. His immense knowledge and expertise helped me to gain new perspectives, and improve my technical skills. I would like to express my sincere gratitude to my mentor Georgios Tzannetos, for his assistance and support at every stage of the project.

I am deeply grateful to all of my friends, who have been a constant source of happiness and fun during my stay at Saarbrücken. I am also thankful to my brother, who has constantly motivated and encouraged me throughout my studies. Finally, I would like to thank my wonderful parents, without whose unconditional and endless support, I would not have written this thesis.

Contents

Declaration of Authorship **3**

Abstract **5**

Acknowledgements **7**

1 Introduction **13**
1.1 Motivation . . . 13
1.2 Contribution . . . 15
1.3 Outline of the Thesis . . . 16

2 Background **19**
2.1 Block-Based Visual Programming . . . 19
2.2 Program Synthesis . . . 21
2.3 RL Framework for Program Synthesis . . . 23
2.4 Adding Diversity to Program Synthesis . . . 24

3 Related Work **27**
3.1 AI for Programming Education . . . 27
3.2 Program Synthesis . . . 27
3.3 Diverse Output Generation . . . 28

4 Problem Setup **31**
4.1 Preliminaries . . . 31
4.2 Formalizing the Objective . . . 33

5 Neural Code Generation Network - (CodeGen)Net **35**
5.1 Behavior Cloning . . . 36
5.2 Multi-Target RL Framework . . . 37
5.3 Integration with Beam Search . . . 38

6 Experiments **39**
6.1 Task Generation and Code Quality Scoring . . . 39
6.2 Evaluation Setup . . . 40
6.3 Results . . . 42

7 Conclusions **47**
7.1 Discussion . . . 47
7.2 Limitations & Future Work . . . 47
7.3 Broader Impact . . . 48

List of Figures **49**

List of Tables **52**

Bibliography **53**

Extra Background & Results **61**
A.1 Recap of REINFORCE Algorithm [60] . . . 61
A.2 Analysis of the Baselines . . . 62
A.3 More Qualitative Results . . . 65

I dedicate this thesis to my family, for their constant support and unconditional love. I love you all dearly!

Chapter 1

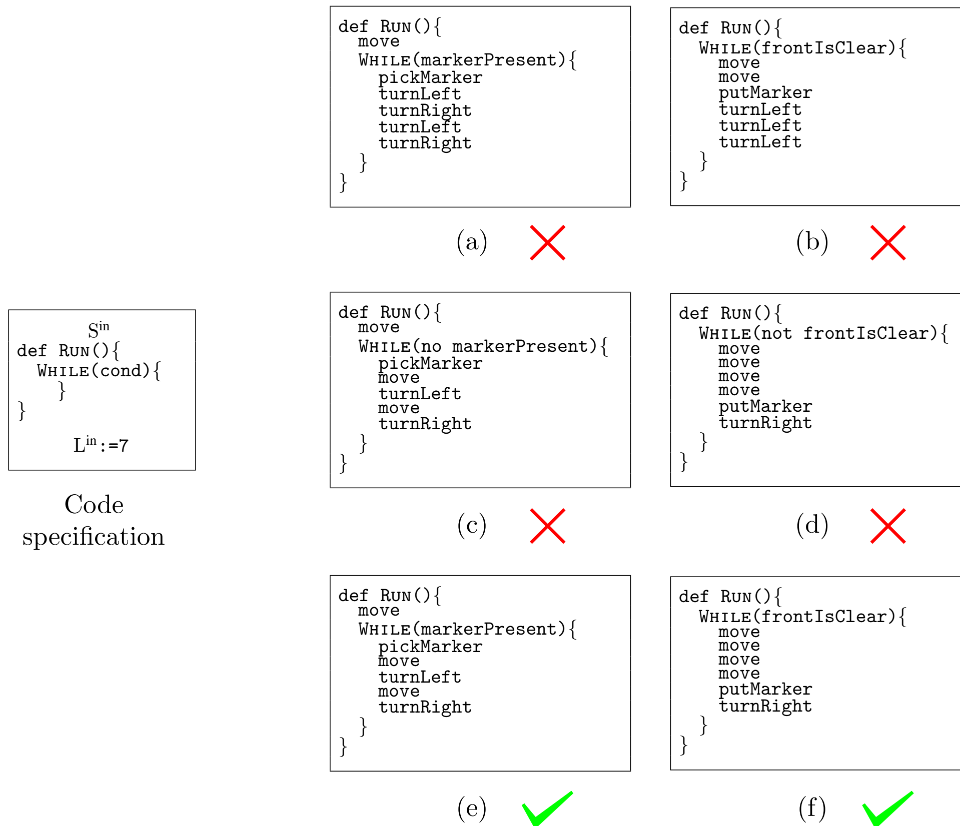
Introduction

1.1 Motivation

With the advent of online education, teachers are able to provide high quality education to students across the world. One of the popular forms of online education that has emerged recently is block-based visual programming, through environments like Scratch[47], initiatives like *Hour of Code* by *Code.org*[3], and online courses like *Stanford CS introductory programming* [1][5] and *Intro to Programming with Karel* by *CodeHS.com*[2]. These have a combination of simple syntax structure along with the ability to convey common programming paradigms such as control flow or routines, making them an ideal choice for new students learning to code [65][66].

Due to these factors, there has been increasing interest to improve the teaching mechanisms for the block-based programming environment using AI systems [45][46][64][66]. This can be seen through the many recent works in the field, such as using AI for hint-generation[43][71][39][18], automated feedback generation systems [53][44][68] or providing worked examples [72]. In spite of all these, the domain suffers from a limited number of programming tasks which are usually hand curated by experts, limiting the number of practice problems available to learn from. For example, HOC's *Classic Maze* challenge [4] provides only 20 puzzles, which can be insufficient for the students to fully comprehend the desired concepts.

To tackle this pedagogical challenge, one solution is to automatically generate diverse, good quality codes, and use the codes to create new practice tasks. Recently, Ahmed et al. [8] attempted such a solution by using a constraint based code generation framework followed by symbolic execution[27]. However, to generate codes, they made use of a set of hand-crafted constraints using expert knowledge, which limits the scalability of the approach. We hypothesize that, recent advancements in AI especially deep learning[31][61][29][38], can be utilized to learn the constraints automatically and be used to generate only those codes which are of good quality. Symbolic execution or any other code-to-task module can then be used to produce new tasks for student learning.

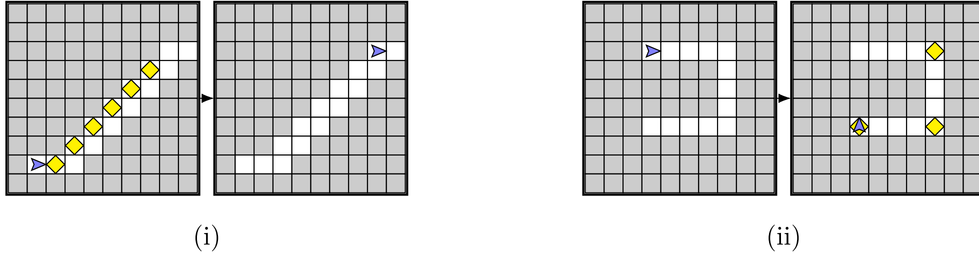


Six different codes (a) to (f) which satisfy the specification

FIGURE 1.1: Here we show a input code specification consisting of a code sketch and desired code length $(\mathrm{S}^{\mathrm{in}}, \mathrm{L}^{\mathrm{in}})$ on the left, and sample codes satisfying it from (a) to (f) on the right. The codes (a) to (d) are of bad quality, since they are semantically incorrect: (a) has consecutive action blocks which do not contribute to the output: *turnLeft* actions followed by *turnRight*, (b) has suboptimal action block sequence: three consecutive *turnLeft* which can be performed by a single *turnRight*, (c) and (d) always cause crash in execution if loops are entered: (c) has a condition to check for no markers followed by a *pickMarker* action, while (d) has a condition to check whether front is not clear followed by a *move* action. The codes (e) and (f) are semantically correct and are also good quality codes.

We formalize the problem of synthesizing codes for block-based visual programming as: given an input code specification containing details such as code sketch and desired code length $(\mathrm{S}^{\mathrm{in}}, \mathrm{L}^{\mathrm{in}})$, the goal is to synthesize a set $\{\mathrm{C}^{\mathrm{out}}\}$ of good quality codes which satisfy the specification. Good quality codes should be semantically correct and should also lead to new practice tasks useful for student learning. For example in Figure 1.1, codes (a) to (d) suffer from semantic irregularities and are considered as bad quality codes; while codes (e) and (f) are semantically correct and also lead to useful tasks (i) and (ii) respectively, shown in Figure 1.2. Thereby, codes (e) and (f) are considered as good quality codes. Finally, we provide another set of examples in Figure 1.3.

There are three key challenges that arise for the problem. First, the space of possible programs is large even for simple specifications, and intractable for more complex



(i) (ii)

FIGURE 1.2: Good quality codes should also lead to new practice tasks useful for student learning. Here, task (i) is produced using code (e) and task (ii) is produced using code (f) from Figure 1.1. Both tasks can be useful for student learning and hence, code (e) and code (f) are considered as good quality codes.

specifications. For example, for the specification shown in Figure 1.1, the number of syntactically correct codes that can be generated is over 100,000; thereby using exhaustive enumeration based techniques [54][7][9] is intractable. Second, out of all the syntactically correct codes, only a small number of them are of good quality. Thereby random generation of programs is not a viable solution(we try this as a baseline and show the results in Section 6.3). Third, the notion of good quality codes is an abstract concept, and it is improbable to define all the rules required to encapsulate the features of good quality codes; hence the constraint based approach of Ahmed et al. [8] or using any rule based approaches [11][35] is infeasible(we also try a constraint based approach in Section 6.3).

Our objective is to overcome these challenges by developing a neural methodology, that can learn the quality features by itself, and automatically generate diverse, good quality codes.

1.2 Contribution

In this work, we aim to solve the problem of synthesizing qualitative and diverse codes for block-based visual programming. To this end, we propose a neural code generation framework (CodeGen)Net which uses behavior cloning to learn from a set of expert demonstrations $\pi^E \sim (\mathrm{S}^{\mathrm{in}}, \mathrm{L}^{\mathrm{in}}) \rightarrow \{\mathrm{C}^{\mathrm{out}}\}^E$, and then uses multi-target reinforcement learning to generate diverse codes with improved quality $\pi^\theta \sim (\mathrm{S}^{\mathrm{in}}, \mathrm{L}^{\mathrm{in}}) \rightarrow \{\mathrm{C}^{\mathrm{out}}\}^\theta$. We demonstrate the effectiveness of our pipeline by performing an extensive empirical evaluation on a set of specifications in the Karel domain.

Our main contributions are summarized as follows.

1. We formalize the problem of qualitative and diverse program synthesis for block-based visual programming (Chapter 4).

2. We propose a neural code generation framework (CodeGen)Net which uses behavior cloning and multi-target reinforcement learning to generate diverse, good quality codes (Chapter 5).

3. We demonstrate the effectiveness of our approach by empirical evaluation of programs synthesized for a set of Karel specifications based on *Intro to Programming with Karel* course by *CodeHS.com* [2]. (Chapter 6).

1.3 Outline of the Thesis

We provide a brief overview of the background concepts required for the thesis in Chapter 2. Chapter 3 consists of an overview of the related works. We formalize the objective in Chapter 4. Our main contributions, proposals and methods are provided in Chapter 5, followed by empirical results in Chapter 6. Finally, we conclude with some discussions, limitations and possible future directions in Chapter 7.

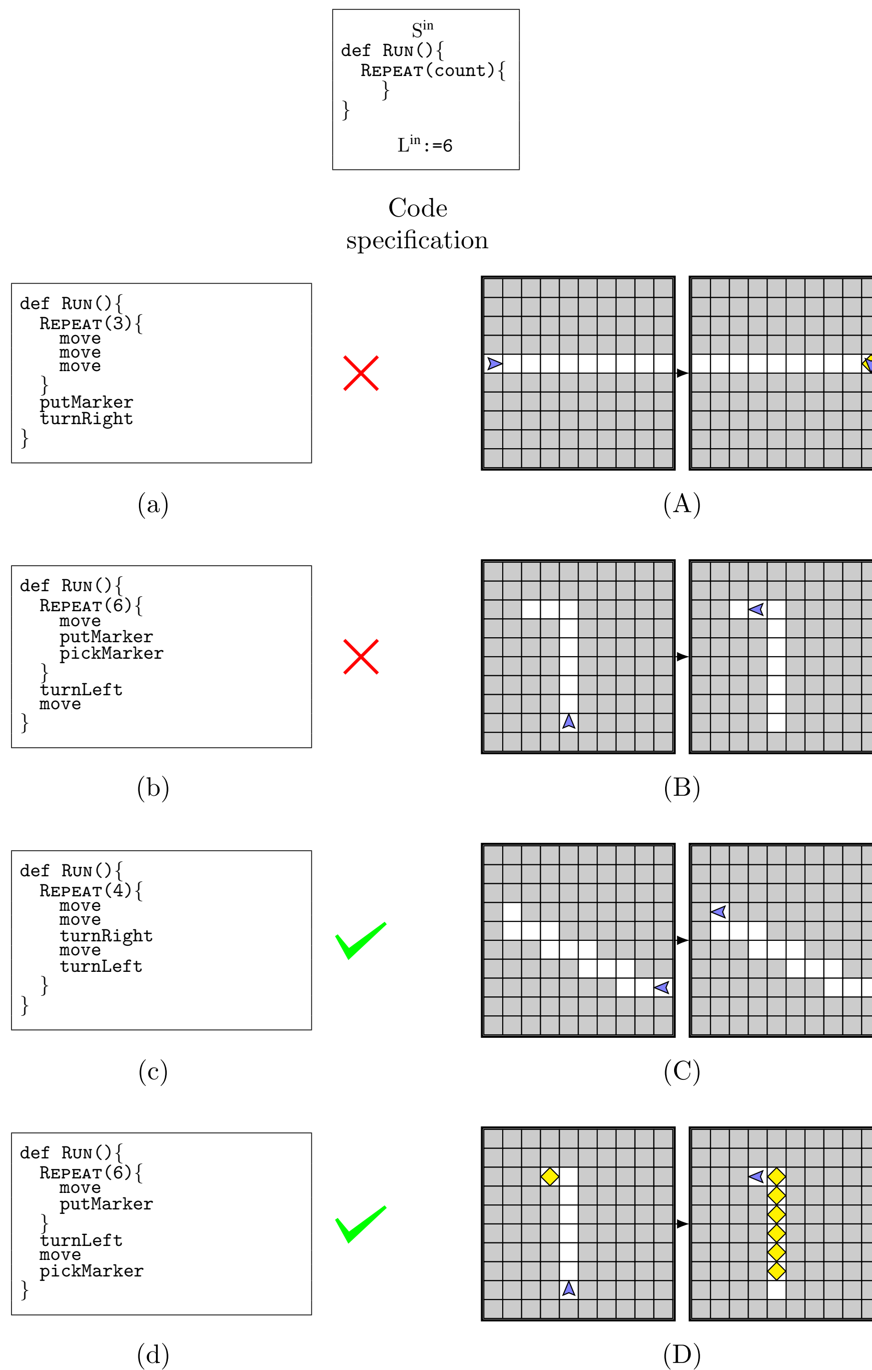


FIGURE 1.3: Here we show another example. Input code specification is shown at the top, sample codes for it are shown on the left side and an example task produced from each code are shown on the right side. The codes (a) and (b) are bad quality codes, since they are semantically incorrect: (a) has a suboptimal *repeat* block: three *move* action blocks can be replaced by a single *move* action block with an increase in *repeat* count, (b) has a redundant sequence of action blocks which do not contribute to the output: *putMarker* action followed by *pickMarker*. (a) and (b) also do not lead to useful practice tasks as shown in (A) and (B) respectively. The codes (c) and (d) are good quality codes, since they are semantically correct, and lead to useful practice tasks as shown in (C) and (D) respectively.

Chapter 2

Background

In this section, we discuss the necessary background for our thesis. We begin with block-based visual programming, and in particular Karel programming which is the environment we work on for our thesis. We then discuss the general problem of program synthesis as well as reinforcement learning framework for program synthesis, which act as a base to build our pipeline. Finally, we discuss the idea of adding diversity to program synthesis, which we is useful for our work.

2.1 Block-Based Visual Programming

Block-based visual programming [47][3][5] is a popular form of programming education which allows students to learn basic programming skills as well as to develop computational thinking. This domain is used to provide a systematic introduction to programming concepts by focusing on logical deduction and spatial reasoning rather than calculation and algebraic reasoning. Students are introduced to programming through a visual approach instead of text, where they can build a program by the use of a fixed set of pre-defined tokens or "blocks".

The development of exercises in these environments are usually dependent on the kind of programming concepts intended to be conveyed. For example, the 20 puzzles of HOC's *Classic Maze* challenge [4] each focus on different programming concepts. The initial puzzles focus on concepts such as creating instructions and learning the syntax structure, while the latter puzzles focus on understanding control flow and improving efficiency. Another example is Karel based course of *Stanford CS introductory programming* [5][1], where separate sections of exercises have been created depending on which concept the student wants to learn; such as exercises for routines or controls. Thereby, having an automated code generation system would be beneficial to develop new exercises based on the required programming concept.

There are various types of programming languages used for block-based learning such as Scratch [47], Hour of Code(HoC) [3] and Karel [41][6]. Karel has been used as a common

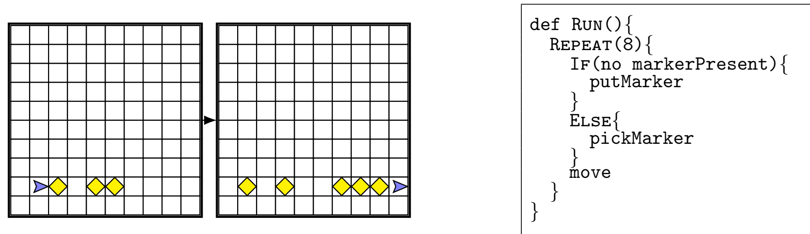


Figure 2.1: An example Karel programming problem, *One ball in each spot* from the *Intro to Programming with Karel* course by *CodeHS.com* [2].

```
code C     := def Run() DO s
rule s     := a | s;s | If(b) DO s | If(b) DO s Else s
              | While(b) DO s | Repeat(x) DO s
action a   := move | turnLeft | turnRight | pickMarker
              | putMarker
cond b     := markerPresent | leftIsClear |
              | rightIsClear | frontIsClear | not(b)
count x    := 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10
```

Figure 2.2: The syntax of Karel DSL.

platform for different areas of AI research such as neural program induction [15] and neural program synthesis [10][49]; we also make use of Karel language to develop our framework.

Karel programming language

A program in Karel is used to control a robot/agent in a virtual environment, in order to explore the environment and manipulate simple objects in it. The environment is a rectangular m $\times$ n grid world, where random cells can contain markers or walls (but not both); note that the agent cannot enter grid cells where there is a wall. A typical Karel programming problem consists of a task in the form of input-output grid, and a solution code that can maneuver the agent to change the environment from the input grid state to the output grid state. An example Karel problem from *CodeHS.com* [2] is shown in Figure 2.1.

For agent control and environment manipulation, five action blocks are available: *move* (the agent moves by one grid cell in the direction it is facing), *turnLeft* (the agent turns 90° left), *turnRight* (the agent turns 90° right) *putMarker* (the agent puts a marker on the grid cell he is standing at), *pickMarker* (the agent lifts a marker off the grid cell he is standing at). The agent can also query information about the current environment state: asking whether there is a marker in its present location(*markerPresent*), and whether there are walls next to it (*frontIsClear*, *leftIsClear*, *rightIsClear*). This can be done by means of conditional statement blocks. The following conditional blocks are available: branching statements (*if*, *if-else*) and loops (*while*, *repeat*). The complete Domain Specific Language(DSL) of Karel programming language is shown in Figure 2.2.

We use the Karel environment for all our proposals, methodology and experiments. This can be observed throughout the thesis.

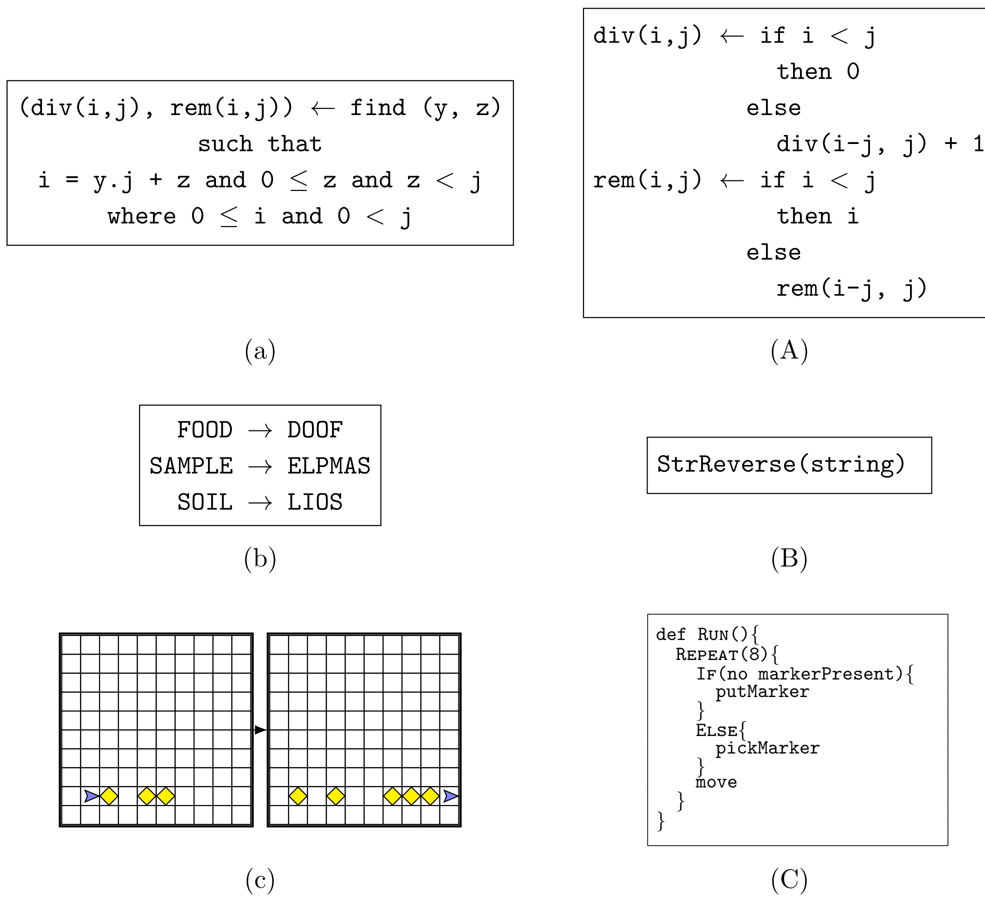


Figure 2.3: Different types of specifications that have been used in program synthesis along with their corresponding programs. (a) uses formal specification, (b) uses text based input-output examples, while (c) uses a visual input-output example. Corresponding programs are shown in (A), (B) and (C) respectively.

2.2 Program Synthesis

Program synthesis is the process of synthesizing a program that satisfies a given specification. The application of program synthesis can range from virtual environments such as pedagogy for student learning, to real world applications such as robot manipulation and control. Due to its wide applicability, the area of automated program synthesis has been researched since a long time [63][21][37][36]. However, only recently it has seen a number of works that make use of deep learning techniques [10][49][12], which is encouraging because using the latest AI advancements to synthesize codes automatically can lead to vast improvements in the field.

The conventional program synthesis problem consists of an input specification and output program/programs; we define both of them here.

Input specification. The input specification should express the purpose of the desired program directly. It should provide a precise idea of what the program is intended to do. The type of input specification can vary depending on the problem definition and

application. This can range from formal specifications [63][36] to input-output examples of text [40] or visual [10] data, as shown on the left side of Figure 2.3.

Output programs. Any program that satisfies the requirements of the input specification is a valid output program for the specification. The format of the output programs also vary based on the application/area of interest, ranging from conventional algorithm structure to syntax of a pre-defined DSL such as string transformation based DSL or Karel DSL, as can be seen in the examples on the right side of Figure 2.3.

In our case, we define specifications in terms of code sketch and code length, which encapsulates the required output codes' structure, contents and length information. We use the Karel DSL to generate the output programs.

Next, we formulate the problem of program synthesis.

Program synthesis formulation

The objective of program synthesis is to learn a code synthesizer σ that produces a program $\mathrm{C_n}$ for a given input specification ψ_n,

$$\sigma : \quad \psi_\mathrm{n} \;\rightarrow\; \mathrm{C_n} \tag{2.1}$$

Usually a pre-defined criteria is used to evaluate the performance of the synthesizer. The synthesizer σ could be any model: random generator, rule-based algorithm, neural network, etc.

Neural program synthesis

If σ is a neural network, the formulation is called as neural program synthesis. In neural program synthesis, a language model such as LSTM [10][49] or Transformer [19] is commonly used as the synthesizer σ.

Let each code be represented by $\mathrm{C} = [\mathrm{c_1}, ..., \mathrm{c_L}]$. The language model predicts a single token at a time. At each time-step, the input to the model is the concatenation of the embedding of the input specification as well as the last predicted token. Then the predicted tokens $\mathrm{c_1}, ..., \mathrm{c_L}$ across all time-steps are concatenated to generate a full program C.

This process can be represented as,

$$p_\theta\left(\mathrm{C_n} \mid \psi_\mathrm{n}\right) = \prod_{\mathrm{t=1}}^{\mathrm{L_n}} p_\theta\left(\mathrm{c_t} \mid \mathrm{c_1}, ..., \mathrm{c_{t-1}}, \psi_\mathrm{n}\right) \tag{2.2}$$

where θ are the parameters of the neural model.

Using MLE for neural program synthesis

The most common method to learn the parameters θ of the neural model, is to use maximum likelihood estimation (MLE) on a set of training data. Assuming we have a training data consisting of N input specifications $(\psi_1, ..., \psi_\mathrm{N})^T$ and corresponding target programs $(\mathrm{C}_1, ..., \mathrm{C}_\mathrm{N})^T$, the objective can be stated as,

$$\theta^\star = \underset{\theta}{\operatorname{argmax}}\ \mathcal{L}_{MLE}(\theta), \quad \text{where } \mathcal{L}_{MLE}(\theta) = \prod_{\mathrm{n}=1}^{\mathrm{N}} p_\theta\left(\mathrm{C_n} \mid \psi_\mathrm{n}\right) \tag{2.3}$$

Adding syntax conditioning to the MLE objective. In order to make learning easier, it is useful to incorporate syntax conditioning which can reduce the space of programs to explore, as well as increase the ability of the model to synthesize syntactically correct programs. This can be done by making use of predefined grammar rules or learning the grammar jointly along with synthesizing programs[10]. The model will then be able to discard any syntactically incorrect program before making a prediction.

We use the MLE objective for neural program synthesis(with syntax conditioning) for the first stage of our framework. The description of this can be found in Section 5.1.

2.3 RL Framework for Program Synthesis

Reward based objective

A framework for using reinforcement learning for program synthesis was introduced by Bunel et al.[10], by adding a reward function to the MLE based objective.

$$\theta^\star = \underset{\theta}{\operatorname{argmax}}\ \mathcal{L}_{RL}(\theta), \quad \text{where } \mathcal{L}_{RL}(\theta) = \sum_{\mathrm{n}=1}^{\mathrm{N}} \left(\sum_{\mathrm{C}} p_\theta\left(\mathrm{C} \mid \psi_\mathrm{n}\right) \mathrm{rew}\left(\mathrm{C}\right) \right) \tag{2.4}$$

Here, the reward function can be used in general to encode any notion of quality of the programs such as program correctness, run-time efficiency etc. For example, Bunel et al [10] used the function to measure the generalization accuracy of the synthesizer model. In our case, we use the reward function to measure the quality of the generated codes.

Using REINFORCE algorithm

The objective of equation 2.4 is intractable to compute, as the inner sum is over all possible programs. Instead, the gradient of this update can be approximated by using Monte Carlo samples, such that the expectation of the sample gradient is proportional to the actual gradient of the objective as a function of the parameter θ. This is called the REINFORCE trick [67], and can be expressed as,

$$
\begin{aligned}
\mathcal{L}_{RL}(\theta) &\approx \sum_{\mathrm{n}=1}^{\mathrm{N}} \sum_{\mathrm{r}=1}^{\mathrm{R}} \frac{1}{\mathrm{R}} \mathrm{rew}\left(\mathrm{C_r}\right), \quad \text{where } \mathrm{C_r} \sim p_\theta\left(\cdot \mid \psi_{\mathrm{n}}\right) \\
\nabla_\theta \mathcal{L}_{RL}(\theta) &\approx \sum_{\mathrm{n}=1}^{\mathrm{N}} \sum_{r=1}^{\mathrm{R}} \frac{1}{\mathrm{R}} \mathrm{rew}\left(\mathrm{C_r}\right) \nabla_\theta \log\left(p_\theta\left(\mathrm{C_r} \mid \psi_{\mathrm{n}}\right)\right)
\end{aligned}
\tag{2.5}
$$

where R is the total number of rollout samples for each specification. We do a recap of the REINFORCE algorithm in Appendix A.1.

We use the RL framework for program synthesis for the second stage of our framework after MLE optimization. The description of this can be found in Section 5.2.

2.4 Adding Diversity to Program Synthesis

The standard MLE optimization objective of equation 2.3 is used to maximize the likelihood of a set of single reference target values for a corresponding set of input values. However, this objective makes it difficult to learn a model that can synthesize multiple, diverse programs for the same specification.

There have been some works that attempt to generate multiple outputs during inference even though the training is performed through a single reference corpus, such as methods based on sampling from a mixture of models [48][69] or decoding with diversity regularization [32][62]. However, these methods suffer from a discrepancy between training and inference. Recently, Lauchaux et al. [30] proposed a MLE based training method that makes learning a one-to-many mapping using a multi-reference corpus possible. In our work, we use a multi-reference corpus and use a similar objective as in [30], in order to synthesize diverse programs. Here we briefly define the modified objective.

Multi-code MLE optimization

The standard MLE objective of equation 2.3 is used to maximize the probability of a target code $\mathrm{C_i}$ for a input specification ψ_{i}. In a multi-target reference corpus of codes, for the same input specification there could be multiple target codes. To learn a one-to-many mapping, we introduce a set of intermediate key values, to which different target codes are mapped, and the language model has to predict the output looking at the specification as well as a key value.

The MLE optimization can then be re-framed as,

$$
\mathcal{L}_{MLE}(\theta) = \prod_{\mathrm{n}=1}^{\mathrm{N}} p_\theta\left(\mathrm{C_n} \mid \psi_{\mathrm{n}}, \mathrm{val_k}\right), \quad \text{where } \mathrm{val_k} \sim \mathrm{key}(\mathrm{C_n}) \tag{2.6}
$$

where the only difference from equation 2.3 is the use of additional key values $\mathrm{val_k}$, with K being the total number of possible key values. The mapping of key-values is performed by learning a neural encoder like Transformer [19] model. The intuition behind this method is that, the framework can learn different code representations corresponding to each key value, and generate different types of codes using these key values once training is complete.

Diverse code generation

Once the model is trained using equation 2.6, it can be utilized to generate diverse codes for a specification, using different key values. For this purpose, no target codes are used and only the different key values are presented along with the input specification, and the model must generate a code for each key value.

This can be represented as,

$$\sigma : \quad \psi_{\mathrm{n}}, \{\mathrm{val_1}, .., \mathrm{val_k}, .., \mathrm{val_K}\} \;\; \rightarrow \;\; \{\mathrm{C_1}, .., \mathrm{C_k}, .., \mathrm{C_K}\}_{\mathrm{n}} \tag{2.7}$$

We use the multi-code MLE optimization of equation 2.6 for the first stage of our neural framework, and the multi-code generation of equation 2.7 for the second and third stages of our neural framework. The description of this can be found in Section 5.1, 5.2 and 5.3.

Chapter 3

Related Work

3.1 AI for Programming Education

Using AI for the improvement in programming education has been increasingly researched as seen in the large amount of recent works. AI has been explored for improving student learning through applications such as hint generation [43][71][39][18], student knowledge understanding [64][25] and providing worked examples [72]. It has also been used to improve the learning platforms through automated feedback generation [53][44][68] and content generation [8][20]. A range of AI techniques have been used depending on the available resources, such as learning code embeddings from historical data [39][42], reinforcement learning in zero-shot setting [18][55], or using expert grammars to generate synthetic training data [68].

In particular, the amount of works focused on block-based visual programming has seen a surge in interest[45][46][64][66], confirming the popularity of this form of education. Closely related to our work, Ahmed et al. [8] introduced a framework to synthesize conceptually similar tasks for block-based visual programming, with a similar goal of developing an automated system to produce new practice tasks for students. However, they use a constraint based approach to generate codes which limits the scalability, whereas we use a neural approach.

3.2 Program Synthesis

The field of program synthesis has been researched since a long time. The initial works on program synthesis focused on constructing programs based on formal specifications of the input-output relation. Many of the works relied on theorem proving techniques [63][21][37][36], where the general idea was to prove a theorem that satisfied the set of input-output relations and then extract a program directly from the proof. Another set of works utilized rules to transform a specification into the desired programs [11][35].

However, the specifications in these needed to be in the form of predicate calculus, and providing such a specification was as complicated as writing the program itself.

This led to programming by examples(PBE) and programming by demonstration(PBD) methods [14][33], in which the specifications had to be provided in the form of input output examples or a set of demonstrations portraying required program behaviour. This simplified the problem of program synthesis since it was much easier to reason about concrete input states, as well as it avoided the need of providing formal specifications. Initially rule based approaches were able to deliver real-time applications through programming by examples such as FlashFill system in excel by Gulwani et al. [22]. However, it was difficult to extend such systems to more complex applications, and these also needed experts to define pruning rules for efficient search. Thereby, the focus shifted towards exploring deep learning techniques for program synthesis [51][12][13].

Neural Program Synthesis

Earlier neural approaches such as RobustFill [16] or Neuro-Symbolic Program Synthesis [40] continued the work on synthesizing programs for string input-output examples. More recently, Bunel et al. [10] focused on synthesizing programs for Karel domain by leveraging the syntax constraints of the Karel program language and a reinforcement learning framework. Shin et al. [49] made use of Karel interpreter to obtain intermediate execution traces and split the problem of specification based synthesis. Chen et al. [12] improved upon these by also making use of the semantic information, by obtaining intermediate grid states through program execution. There has also been some interest to focus on more challenging languages like C programming [13], but this has been limited so far. Deep learning for synthesis based on demonstrations has also been explored. For example, Sun et al. [58] and Duan et al. [17] have explored synthesizing programs using video frames as demonstrations, with the intention that the programs should summarise information and predict the underlying logic in the video frames.

Another interesting line of work is combining the neural synthesis process with program debugging [24][50]. The main advantage of this is it has more resemblance to human coders. Finally, some other works have focused on creating synthetic datasets [51][57] or query based datasets [26] for model training, which led to improvement in the generalization capabilities. Our work can also be utilized for developing synthetic examples in the block-based environment, with the advantage that the examples would be more suitable for the real world scenario.

3.3 Diverse Output Generation

There have been different approaches proposed to generate a diverse set of outputs for a given input, in machine learning. These include adding random noise to the latent

space of a VAE [28], sampling from a mixture of models [48][69], applying diversity regularization to decoding algorithm [32][62] and conditioning the decoding procedure with diverse signals [52][30].

One recent work of the last category was by Lachaux et al. [30], who made use of a neural framework that learnt a key value mapping of targets during training. These key values were then used to generate diverse outputs during inference. Their procedure was developed with respect to neural machine translation; we derive a similar procedure for diverse code generation based on key values for our work(see section 2.4).

Another recent work for diverse neural machine translation was by Lin et al [34], who made use of a multi-target reinforcement learning framework with rewards based on the quality and diversity of the outputs. We use a similar approach in the RL stage of our pipeline (see section 5.2). In their case, they defined rewards based on the language translations, while we define rewards based on the quality of codes generated.

Chapter 4

Problem Setup

```
code C    := def Run() DO s
rule s    := a | s;s | If(b) DO s | If(b) DO s Else s
             | While(b) DO s | Repeat(x) DO s
action a  := move | turnLeft | turnRight | pickMarker
             | putMarker
cond b    := markerPresent | leftIsClear |
             | rightIsClear | frontIsClear | not(b)
count x   := 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10
```

Figure 4.1: Karel Code DSL

```
sketch Q  := def Run() DO S
rule S    := ϕ | S;S | If(B) DO S | If(B) DO S Else S
             | While(B) DO S | Repeat(X) DO S
```

Figure 4.2: Karel Sketch DSL

4.1 Preliminaries

Code space. We define the space of all possible codes as $\mathbb{C}$ and represent them using a Domain Specific Language (DSL) [23]. For this work, we specifically employ the Karel DSL shown in Figure 4.1 to represent the codes. A code $\mathrm{C} \in \mathbb{C}$ has the following attributes: $\mathrm{C}_{\text{blocks}}$ is the set of types of code blocks used in C, C_{size} is the number of code blocks used, $\mathrm{C}_{\text{depth}}$ is the depth of the Abstract Syntax Tree of C and the nesting structure $\mathrm{C}_{\text{struct}}$ represents programming concepts exercised by C. For example for the code (b) in Figure 4.3, $\mathrm{C}_{\text{blocks}} = \{\textit{move}, \textit{turnLeft}, \textit{pickMarker}, \textit{turnRight}, \textit{while}\}$, $\mathrm{C}_{\text{size}} = 7$, $\mathrm{C}_{\text{depth}} = 2$, and $\mathrm{C}_{\text{struct}} = \{\textit{Run}\{\textit{while}\}\}$.

Sketch-Length space. We define a code sketch as a higher-level representation that encapsulates the essential components of a code. We derive a sketch DSL from the Karel DSL as shown in Figure 4.2 to define the sketch space $\mathbb{S}$. Similar to the Abstract Syntax Tree representation of a code, we represent a sketch as a tree having the programming constructs as its nodes and represent its depth by $\mathrm{S}_{\text{depth}}$, and the nesting structure by $\mathrm{S}_{\text{struct}}$. On the other hand, the length space $\mathbb{L}$ is the space of natural numbers. We

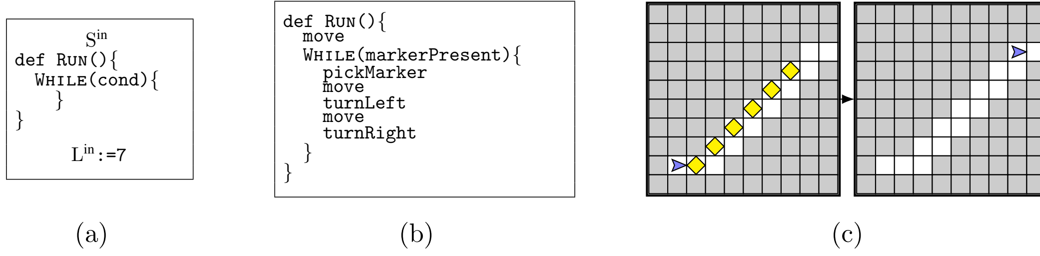


FIGURE 4.3: Reusing examples from Section 1 to define the preliminaries. Shown here is a specification (a), code (b) and visual task (c).

define desired code length L as the pre-determined number of code blocks constraining the size of codes. For example, for the specification (a) in Figure 4.3, $\mathrm{S}_{\mathrm{depth}} = 2$, $\mathrm{S}_{\mathrm{struct}} = \{Run\{while\}\}$ and $\mathrm{L} = 7$.

The mapping from the sketch-length space to the code space is captured by the one-to-many map, $\Phi : (\mathbb{S}, \mathbb{L}) \rightarrow \mathbb{C}$, i.e., the representation of a code C in $(\mathbb{S}, \mathbb{L})$ is given by $\Phi^{-1}(\mathrm{C})$. Note that for a given $(\mathrm{S}, \mathrm{L}) \in (\mathbb{S}, \mathbb{L})$ there can be multiple $\mathrm{C} \in \mathbb{C}$. Further, a code C is said to satisfy a specification (S, L), if $\mathrm{C}_{\mathrm{depth}} = \mathrm{S}_{\mathrm{depth}}$, $\mathrm{C}_{\mathrm{struct}} = \mathrm{S}_{\mathrm{struct}}$ and $\mathrm{C}_{\mathrm{size}} = \mathrm{L}$.

Task space. We define the space of tasks as $\mathbb{T}$. A task $\mathrm{T} \in \mathbb{T}$ consists of a visual task $\mathrm{T}_{\mathrm{vis}}$, a set of available types of code blocks (e.g., *move*, *turnLeft*) allowed in the solution code, and a limit on the length of solution code in terms of the number of code blocks. In this work, we restrict to the Karel tasks as used in *Intro to Programming with Karel* by *CodeHS.com* [2]. An example visual task can be seen in Figure 4.3 (c).

We relate the code and task space using the following definition.

Definition (Suitable visual task). A visual task $\mathrm{T}_{\mathrm{vis}}$ is suitable for a code C if the following holds: the visual task $\mathrm{T}_{\mathrm{vis}}$ can be successfully solved using C. We use $\{\mathrm{T}_{\mathrm{vis}}\}_{\mathrm{C}}$ to denote a set of different suitable visual tasks for C.

4.2 Formalizing the Objective

To formalize our objective, we use the domain knowledge about the block-based environment to arrive at an empirical measure of the quality of a code.

Measuring code quality. A good quality code C should not suffer from semantic irregularities such as redundant action sequence blocks, suboptimal action sequences etc., and also should have a suitable visual task $\mathrm{T_{vis}} \in \{\mathrm{T_{vis}}\}_{\mathrm{C}}$ that has some notion of *interestingness and complexity.* We measure the code quality with a environment specific function $\mathcal{F}_{\mathrm{codequal}}(\mathrm{C}) \in [0,1]$. We provide the specific instantiation of $\mathcal{F}_{\mathrm{codequal}}$ in the experiments chapter.

Objective. For any input specification consisting of a code sketch $\mathrm{S^{in}}$ and desired code length $\mathrm{L^{in}}$, our objective is to synthesize a set of output codes $\{\mathrm{C^{out}}\}$ satisfying the specification, where $\mathrm{C^{out}}$ has a quality score above δ, i.e. $\mathcal{F}_{\mathrm{codequal}}\left(\mathrm{C^{out}}\right) \geq \delta$, with δ being a pre-defined threshold.

Chapter 5

Neural Code Generation Network - (CodeGen)Net

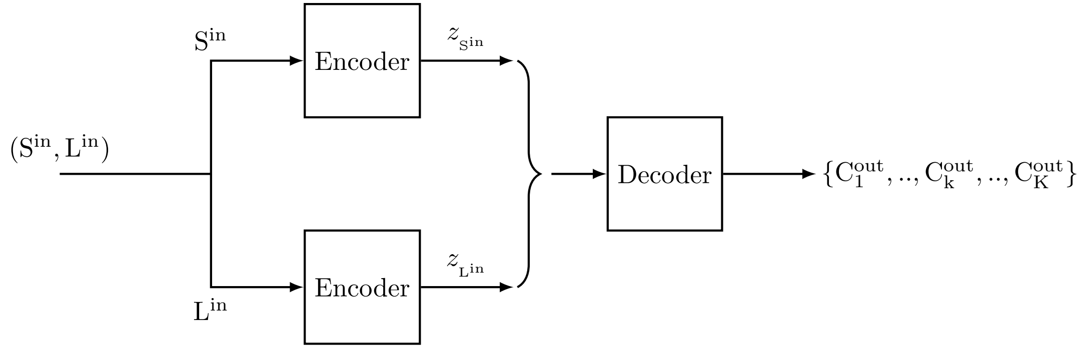


Figure 5.1: Our algorithm (CodeGen)Net takes as input a code specification (S^{in}, L^{in}) and generates a set of qualitative and diverse codes $\{C_1^{out}, .., C_k^{out}, .., C_K^{out}\}$.

In this section, we present our technique (CodeGen)Net, which takes as input a code specification containing sketch S^{in} and desired length L^{in} and generates a set of diverse codes $\{C_1^{out}, .., C_k^{out}, .., C_K^{out}\}$. The goal is for this set of codes to satisfy the objective of Section 4.2. To achieve this goal, (CodeGen)Net operates in three stages: (i) In Stage-1, we make use of a set of expert demonstrations consisting of specifications(sketch and length) and good quality codes, and use behavior cloning to train a neural model. (ii) In Stage-2, we design a reinforcement learning framework with quality based rewards to fine-tune the neural model from Stage-1. (iii) In Stage-3, we integrate the model from Stage-2 with beam search and release it for validation.

The core architecture remains the same across the three stages, and we describe it first.

(CodeGen)Net architecture. We use linear encoders to generate embeddings for the input sketch $z_{S^{in}}$ and the input length $z_{L^{in}}$, which are passed on to the decoder. On the decoder side, each output code is modelled one token at a time using an LSTM network. At each time-step, the input embeddings and the last predicted token are concatenated

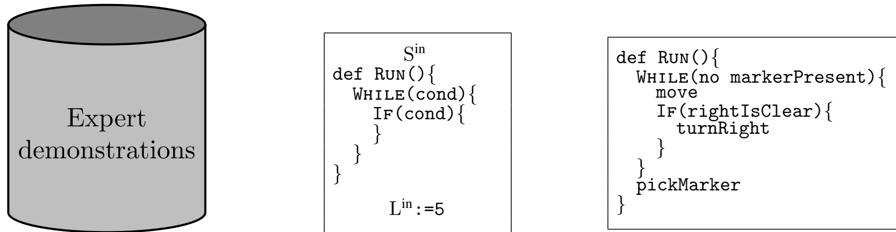


FIGURE 5.2: Stage 1 of our technique-behavior cloning uses a set of expert demonstrations containing code specifications and good quality codes.

and passed as input to decoder. The predicted tokens are collected across the time-steps and combined to generate a code $\mathrm{C}_{\mathrm{k}}^{\mathrm{out}}$.

Now we describe the three stages of our algorithm.

5.1 Behavior Cloning

We use behavior cloning in Stage-1 to pre-train our neural pipeline. In this stage, we make use of a set of expert demonstrations $\pi^E \sim (\mathrm{S}^{\mathrm{in}}, \mathrm{L}^{\mathrm{in}}) \rightarrow \{\mathrm{C}^{\mathrm{out}}\}^E$, containing code specifications and expert generated codes, and utilize the encoder-decoder network to pre-train a neural policy $\pi^\theta \sim (\mathrm{S}^{\mathrm{in}}, \mathrm{L}^{\mathrm{in}}) \rightarrow \{\mathrm{C}^{\mathrm{out}}\}^\theta$. We perform behavioral cloning(BC) to directly estimate the expert policy π^E, with maximum likelihood estimation (MLE). Using the standard MLE objective for specification based synthesis defined in 2.3, the training objective can be stated as,

$$\theta^\star = \underset{\theta}{\operatorname{argmax}}\, \mathcal{L}_{BC}(\theta), \quad \text{where } \mathcal{L}_{BC}(\theta) = \prod_{\mathrm{n}=1}^{\mathrm{N}} p_\theta \left(\mathrm{C}_{\mathrm{n}}^{\mathrm{out}} \mid (\mathrm{S}_{\mathrm{n}}^{\mathrm{in}}, \mathrm{L}_{\mathrm{n}}^{\mathrm{in}}) \right) \tag{5.1}$$

where θ are the parameters of the model π^θ, N is the total number of available specifications, $(\mathrm{S}_{\mathrm{n}}^{\mathrm{in}}, \mathrm{L}_{\mathrm{n}}^{\mathrm{in}})$ are code sketch and code length of a specification, and $\mathrm{C}_{\mathrm{n}}^{\mathrm{out}}$ is a expert generated code. Note that we also add syntax conditioning as part of the objective as described in Section 2.2.

Multi-code optimization: Due to the one-to-many relation of the sketch space to the code-space, a single specification could have multiple codes in the expert demonstrations. However, the standard MLE objective of equation 5.1 is not suitable for a multi reference training corpus. Moreover, using a single reference demonstration set is not a viable option, as this limits the diversity in the output codes that can be generated during inference. To deal with this issue, we use the modified MLE objective defined in Section 2.4 that enables multi-code MLE optimization and diverse code generation. Please refer to 2.4 for more details about the modified MLE objective.

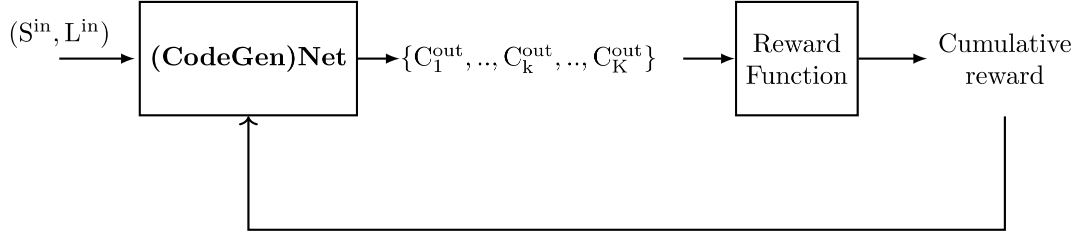


FIGURE 5.3: Stage 2 of our technique-fine tuning the neural model from stage 1 with a multi-target reinforcement learning framework.

5.2 Multi-Target RL Framework

Next, we describe stage 2 of our algorithm. In this stage, we try to guide the neural model from stage 1 to improve the quality of code generation using a multi-target reinforcement learning (RL) framework. For this purpose, we ask the neural model from stage 1 to generate K output codes for each specification and compute a quality based reward for all K codes. Specifically, for a specification $(\mathrm{S}^{\mathrm{in}}_\mathrm{n}, \mathrm{L}^{\mathrm{in}}_\mathrm{n})$ we ask the BC model to generate $\{\mathrm{C}^{\mathrm{out}}_1, .., \mathrm{C}^{\mathrm{out}}_\mathrm{k}, .., \mathrm{C}^{\mathrm{out}}_\mathrm{K}\}$. Note that, we generate multiple codes again using the diverse code generation setting of Section 2.4. We then compute a reward value for each $\mathrm{C}^{\mathrm{out}}_\mathrm{k}$ using a reward function and the cumulative reward is used to optimize model parameters. The overview of this stage is illustrated in Figure 5.3.

Cumulative reward function: The easiest approach to add a quality based reward function is to use the same code quality scoring function $\mathcal{F}_{\mathrm{codequal}}$ introduced in Section 4. Thereby, we can define the cumulative reward as,

$$\mathrm{rew}(\{\mathrm{C}^{\mathrm{out}}_1, .., \mathrm{C}^{\mathrm{out}}_\mathrm{k}, .., \mathrm{C}^{\mathrm{out}}_\mathrm{K}\}) = \sum_{\mathrm{k}=1}^{\mathrm{K}} \mathcal{F}_{\mathrm{codequal}} \left(\mathrm{C}^{\mathrm{out}}_\mathrm{k}\right) \tag{5.2}$$

Model Training: In contrast to the previous stage, each training instance of this stage consists of a specification and the K different predicted codes. The cumulative reward over the predicted codes is the cost that needs to be maximized. We train the model using the REINFORCE based objective defined in equation 2.5, with the difference being we optimize over K codes for a specification instead of one; i.e we want to maximize

$$\begin{gathered}\mathcal{L}_{RL}(\theta) \approx \sum_{\mathrm{n}=1}^{\mathrm{N}} \sum_{\mathrm{r}=1}^{\mathrm{R}} \frac{1}{\mathrm{R}} \mathrm{rew}\left(\{\mathrm{C}^{\mathrm{r}}_1, .., \mathrm{C}^{\mathrm{r}}_\mathrm{k}, .., \mathrm{C}^{\mathrm{r}}_\mathrm{K}\}\right), \\ \text{where } \{\mathrm{C}^{\mathrm{r}}_1, .., \mathrm{C}^{\mathrm{r}}_\mathrm{k}, .., \mathrm{C}^{\mathrm{r}}_\mathrm{K}\} \sim p_\theta\left(\cdot \mid (\mathrm{S}^{\mathrm{in}}_\mathrm{n}, \mathrm{L}^{\mathrm{in}}_\mathrm{n})\right)\end{gathered} \tag{5.3}$$

where R is the total number of rollouts for each specification. Using the REINFORCE policy gradient method, we update the parameters of the neural network π^θ.

5.3 Integration with Beam Search

In this stage, we take the RL-fine tuned model from the previous stage and integrate its decoder with beam search. This enables multiple hypothesis to be maintained at every step of the decoding process, resulting in non greedy local decisions. We carry out beam search for every target code prediction $\mathrm{C}_{\mathrm{k}}^{\mathrm{out}}$, and this results in a considerably improvement in the performance of the model. We show this empirically in Section 6.3. After integration with beam search, we release our neural model for validation.

Chapter 6

Experiments

6.1 Task Generation and Code Quality Scoring

In this section, we describe our process of obtaining suitable visual tasks for generated codes and define the scoring function to validate the quality of codes.

Obtaining visual tasks. For each generated code $\mathrm{C}^{\mathrm{out}}$ by a model, we generate suitable visual tasks $\{\mathrm{T}_{\mathrm{vis}}\}_{\mathrm{C}^{\mathrm{out}}}$ in order to measure the quality of $\mathrm{C}^{\mathrm{out}}$. We achieve this by using a version of the task synthesis framework of Ahmed et al. [8] consisting of symbolic execution and best-first search techniques. The best possible $\mathrm{T}_{\mathrm{vis}}^{\mathrm{out}} \in \{\mathrm{T}_{\mathrm{vis}}\}_{\mathrm{C}^{\mathrm{out}}}$(the one with the highest $\mathcal{F}_{\mathrm{combqual}}$ value-see below) is used to measure the quality of $\mathrm{C}^{\mathrm{out}}$.

Note that, our framework is adaptable to any other task generation technique such as neural generation or human-induced generation.

Code quality scoring. We first define a scoring function that evaluates a code $\mathrm{C}^{\mathrm{out}}$ in combination with each suitable visual task $\mathrm{T}_{\mathrm{vis}}^{\mathrm{out}} \in \{\mathrm{T}_{\mathrm{vis}}\}_{\mathrm{C}^{\mathrm{out}}}$. This combined scoring function should encapsulate the desired quality criteria. To this end, we define the function with the following constituent parts:

1. $\mathcal{F}_{\mathrm{nocrash}}\left(\mathrm{C}^{\mathrm{out}}, \mathrm{T}_{\mathrm{vis}}^{\mathrm{out}}\right) \in 0, 1$, which evaluates to 0 in case the agent crashes into a wall and 1 otherwise.
2. $\mathcal{F}_{\mathrm{nocut}}\left(\mathrm{C}^{\mathrm{out}}, \mathrm{T}_{\mathrm{vis}}^{\mathrm{out}}\right) \in 0, 1$ which evaluates to 0 if there is a shortcut sequence of actions smaller than $\mathrm{C}_{\mathrm{size}}^{\mathrm{out}}$ size that solves $\mathrm{T}_{\mathrm{vis}}^{\mathrm{out}}$ and 1 otherwise.
3. $\mathcal{F}_{\mathrm{cov}}\left(\mathrm{C}^{\mathrm{out}}, \mathrm{T}_{\mathrm{vis}}^{\mathrm{out}}\right) \in 0, 1$, which evaluates to 1 in the event of complete coverage of code $\mathrm{C}^{\mathrm{out}}$ by task $\mathrm{T}_{\mathrm{vis}}^{\mathrm{out}}$ and 0 otherwise.
4. $\mathcal{F}_{\mathrm{taskqual}}\left(\mathrm{C}^{\mathrm{out}}, \mathrm{T}_{\mathrm{vis}}^{\mathrm{out}}\right)$ which acts as a measure of the *interestingness and complexity* of the visual task, and is approximated as the sum of the normalized counts of ‘moves’, ‘turns’, ‘segments’, ‘long-segments’, ’pick-markers’ and ’put-markers’ in

the grid; where segments and long segments are sequences of ≥ 3 and ≥ 5 move actions respectively.

That is,

$$\mathcal{F}_{\text{taskqual}}\left(\mathrm{C}^{\text{out}}, \mathrm{T}_{\text{vis}}^{\text{out}}\right) = \frac{3}{4} \cdot \frac{1}{4}\left(\frac{\#\text{moves}}{2n} + \frac{\#\text{turns}}{n} + \frac{\#\text{segments}}{n/2} + \frac{\#\text{long-segments}}{n/3}\right) + \frac{1}{4} \cdot \frac{1}{2}\left(\frac{\#\text{pick-markers}}{n} + \frac{\#\text{put-markers}}{n}\right) \tag{6.1}$$

Overall, this combined scoring function is given by,

$$\mathcal{F}_{\text{combqual}}\left(\mathrm{C}^{\text{out}}, \mathrm{T}_{\text{vis}}^{\text{out}}\right) = \left(\mathcal{F}_{\text{nocrash}}\left(\mathrm{C}^{\text{out}}, \mathrm{T}_{\text{vis}}^{\text{out}}\right) = 1\right) \cdot \left(\mathcal{F}_{\text{nocut}}\left(\mathrm{C}^{\text{out}}, \mathrm{T}_{\text{vis}}^{\text{out}}\right) = 1\right) \cdot \left(\mathcal{F}_{\text{cov}}\left(\mathrm{C}^{\text{out}}, \mathrm{T}_{\text{vis}}^{\text{out}}\right) = 1\right) \cdot \left(\mathcal{F}_{\text{taskqual}}\left(\mathrm{C}^{\text{out}}, \mathrm{T}_{\text{vis}}^{\text{out}}\right)\right) \tag{6.2}$$

The best suitable visual task is then used to evaluate the quality of the code, i.e.

$$\mathcal{F}_{\text{codequal}}\left(\mathrm{C}^{\text{out}}\right) = \max_{\{\mathrm{T}_{\text{vis}}\}_{\mathrm{C}^{\text{out}}}} \mathcal{F}_{\text{combqual}}\left(\mathrm{C}^{\text{out}}, \mathrm{T}_{\text{vis}}^{\text{out}}\right) \tag{6.3}$$

6.2 Evaluation Setup

Baselines and models evaluated. We evaluate two baselines (Rand), (Rand)Cstr along with two models from our pipeline (BC)Net and (CodeGen)Net.

Given a code specification, (Rand) generates random but syntactically valid codes. The number of syntactically valid codes for any specification is too large, especially for more complex specifications. To be feasible, we ask (Rand) to generate the same number of codes as our system (CodeGen)Net.

(Rand)Cstr is an extension of (Rand), where we use the human selected constraints of Ahmed et al. [8] to describe the desired semantics of the generated codes. These constraints filter the randomly generated codes. In particular, we use the following constraints. (i) Minimality: this ensures redundant sequences such as *turnRight-turnLeft* or *pickMarker-putMarker* which do not affect the output, as well as *turnLeft-turnLeft-turnLeft* which could be achieved by a single *turnRight* are eliminated. (ii) Action sequence constraint within nested conditionals: this prevents invalid actions within conditionals such as *pickMarker* after *noMarkerPresent* condition. (iii) Optimal *repeat* counter: this ensures action sub-sequences before and after a *repeat* block are not nested within it.

(BC)Net is the behavior cloned only model(without RL fine-tuning) from our pipeline which is integrated with beam search. (CodeGen)Net is the final model from our

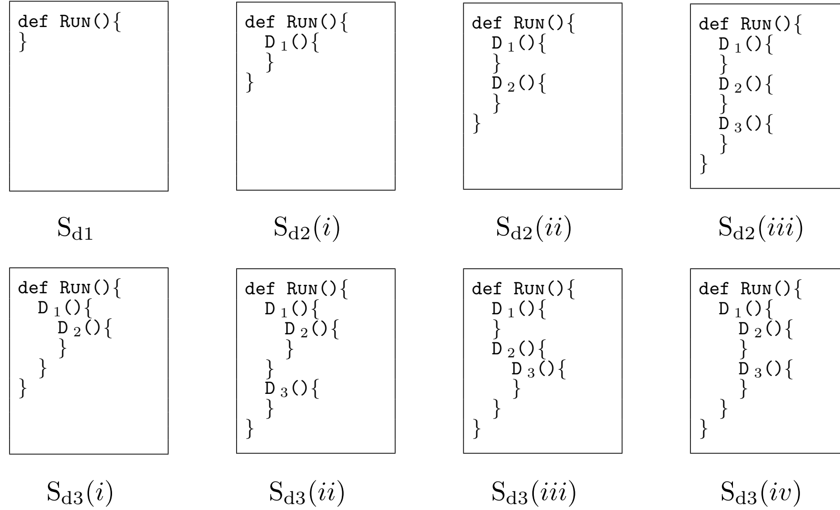


FIGURE 6.1: Illustration of the 8 different templates that we work with. All our specifications are based on these 8 different templates, which are divided here according to their depth. Here S_{d1}, $S_{d2}[(i)$ to $(iii)]$ and $S_{d3}[(i)$ to $(iv)]$ correspond to all sketches of depth 1,2 and 3 respectively. Note that, D_1,D_2 and D_3 can correspond to any control block.

pipeline, i.e. the model that has undergone all the three stages(behavior cloning, RL-fine tuning and integration with beam search) of our system.

Hyperparameters. For our neural models, we used linear encoders to encode the specification(code sketch and code length). On the decoder side, we followed Bunel et al. [10] and used a two layer LSTM network with hidden size of 256. The code sketch, code length and token at previous step are all embedded in the form of 256 dimensional vectors and passed as input to the LSTM. The output of the LSTM at each time step is passed through a handwritten syntax checker and a probability distribution over the next tokens is obtained. All training is performed with Adam optimizer with a learning rate of 10^{-4}. Behavior cloning used a batch size of 128 and RL training used a batch size of 8. Beam search was used with a beam size of 64.

For the multi-target optimization component, we used a transformer encoder to learn the mapping to key values. The number of key values was set to 10.

Code specifications and expert demonstrations. We use 1000 training specifications and 100 validation specifications of Karel, each differing in their sketch/length /both. The specifications are based on 8 different templates as shown in Figure 6.1, with a maximum depth S_{depth} of 3 and maximum action blocks of 15. The training specifications also had 10 expert generated codes, with each code having a score $\mathcal{F}_{\text{codequal}} \geq 0.5$. These were used to train the neural models, and the remaining 100 were used to validate the baseline as well as the neural models.

Method	Fraction of codes with $\delta \geq 0.5$				Fraction of codes with $\delta \geq 0.7$			
	S_{d1}	S_{d2}	S_{d3}	$\mathbf{S_{all}}$	S_{d1}	S_{d2}	S_{d3}	$\mathbf{S_{all}}$
(Rand)	0.23	0.13	0.13	**0.14**	0.00	0.02	0.07	**0.06**
(Rand)Cstr	0.90	0.31	0.29	**0.31**	0.00	0.13	0.18	**0.16**
(BC)Net	0.90	0.69	0.48	**0.52**	0.00	0.14	0.25	**0.22**
(CodeGen)Net	0.90	0.78	0.71	**0.73**	0.00	0.13	0.45	**0.39**

TABLE 6.1: Results for the Karel specifications for two different quality thresholds: $\delta = 0.5$ and $\delta = 0.7$. Results are shown separately for different specification templates S_{dX} and for all specifications combined S_{all}. See Section 6.3 for more description.

Evaluation Criteria. We evaluate the generated codes based on the objectives of Section 4.2. Since our objectives are encapsulated within the code quality scoring function, we calculate the quality score of each generated code and compare with a desired threshold value δ. Specifically, we ask each model to generate 10 codes for each validation specification, and evaluate the performance depending on the fraction of the total generated codes that satisfy the threshold.

6.3 Results

Performance comparison.

Table 6.1 shows the performance of the models for validation specifications of depth levels 1,2,3 separately, and all combined; and for two quality threshold values (δ): 0.5 (good quality) and 0.7 (very good quality).

Let us consider δ of 0.5 first. The simple baseline (Rand) performs poorly independent of the depth level. This supports our belief that, the fraction of good quality codes among the syntactically valid codes is small, making random generation not a viable option. The other baseline (Rand)Cstr seems to improve compared to (Rand). This is especially true for depth level 1 specifications, where it meets the quality threshold for 90% of the generated codes, thereby getting a score of 0.9. However, for higher depth levels(2 and 3), the number of generated codes satisfying δ drops significantly, indicating the difficulty in developing handcrafted constraints that can capture all the features of good quality codes. On the other hand, our neural models (BC)Net and (CodeGen)Net are able to keep up with the constraint based baseline for depth level 1 specifications. However, they are able to vastly improve for higher depth levels of 2 and 3, indicating that a neural framework is better able to learn the good quality codes' features. Among the neural models, (CodeGen)net provides a slight improvement over (BC)net for depth level 2 specifications, but a high improvement of 0.33 for depth level 3 specifications. We hypothesize that, this happens because the number of patterns that lead to good quality

codes increases with the increase in depth level, and using RL, the (CodeGen)net model is able to explore and find more of these patterns.

The performance of the models are on similar lines even if we consider a higher quality threshold, δ of 0.7. None of the generated codes for depth 1 specifications meet the threshold, across the models; this indicates that the codes of depth 1 have a maximum quality score of less than 0.7. Interestingly for depth 2 specifications, the performance of the the (Rand)Cstr baseline and neural models are similar, indicating that the neural models find it difficult to find patterns that can lead to very good quality codes for lower depth levels. But as the depth increases, i.e. for depth 3 specifications, the neural models are able to perform much better than the (Rand)Cstr baseline. This is especially true for (CodeGen)Net, which sees an improvement of 0.27 over the (Rand)Cstr baseline.

Qualitative Analysis.

We perform qualitative analysis for a specification with $\text{S}_{\text{depth}} = 3$, $\text{S}_{\text{struct}} = \{\text{Run}\{\text{WHILE}\{\text{IF}\}\}\}$ and $\text{L}^{\text{in}} = 5$. The codes generated by our model (CodeGen)Net are shown in the Figure 6.2 from (a) to (j). We also show example suitable visual tasks (A), (B) corresponding to two good quality codes (a), (b), and (H), (I) corresponding to two bad quality codes (h), (i) in Figure 6.3. We use δ of 0.5.

(CodeGen)Net is able to satisfy the threshold for 7 out of the 10 codes(code (a) to code (g)), it was asked to generate. This is close to the 0.71 overall score for all the specifications of depth 3, as shown in Table 6.1. Most of the codes that satisfy this threshold are free from semantic irregularities, and can lead to interesting tasks, examples of which can be seen in the two tasks (A) and (B) produced using codes (a) and (b); these can be used as new practice tasks for student learning. However, this is not always the case. For example, code (e) in Figure 6.2 contains an *if* block with *leftIsClear* condition followed by a *turnRight* action: this does not seem a good quality code sequence intuitively, but the code still gets a good score. This indicates a limitation in the quality scoring function. Also, code(c) has limited utility since it can only lead to tasks with simple agent movements; however this can be filtered with a higher δ value.

On the other hand, code (h) to code(j) suffer from severe semantic irregularities: code (h) and code (j) always lead to crash in execution/infinite loop in case the *if* block of the codes is entered, while the *if* block in code (i) is redundant. These codes also cannot lead to any interesting task, as seen through the corresponding visual tasks (H) and (I) for code (h) and code (i) respectively. Hence, there are bad quality codes and do not satisfy the threshold.

Further, we also perform analysis for the same specification for the baselines (Rand) and (Rand)Cstr, and describe it in the appendix section A.2.

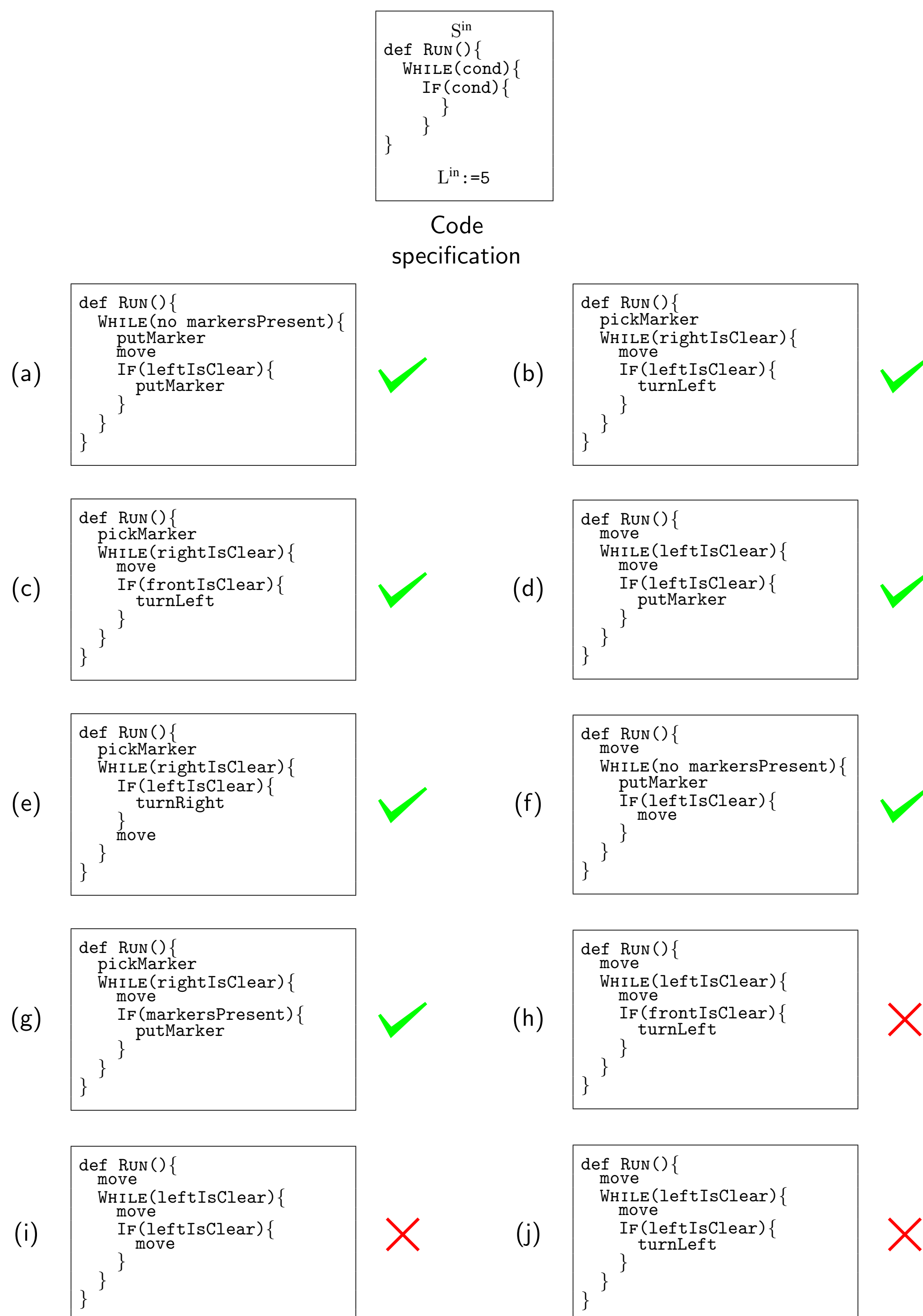


FIGURE 6.2: Illustration of 10 different codes (a) to (j) generated by our model (CodeGen)Net for the code specification shown at the top. In this case, the model generated 7 good quality codes (a) to (g), and 3 bad quality codes (h) to (j), for δ of 0.5.

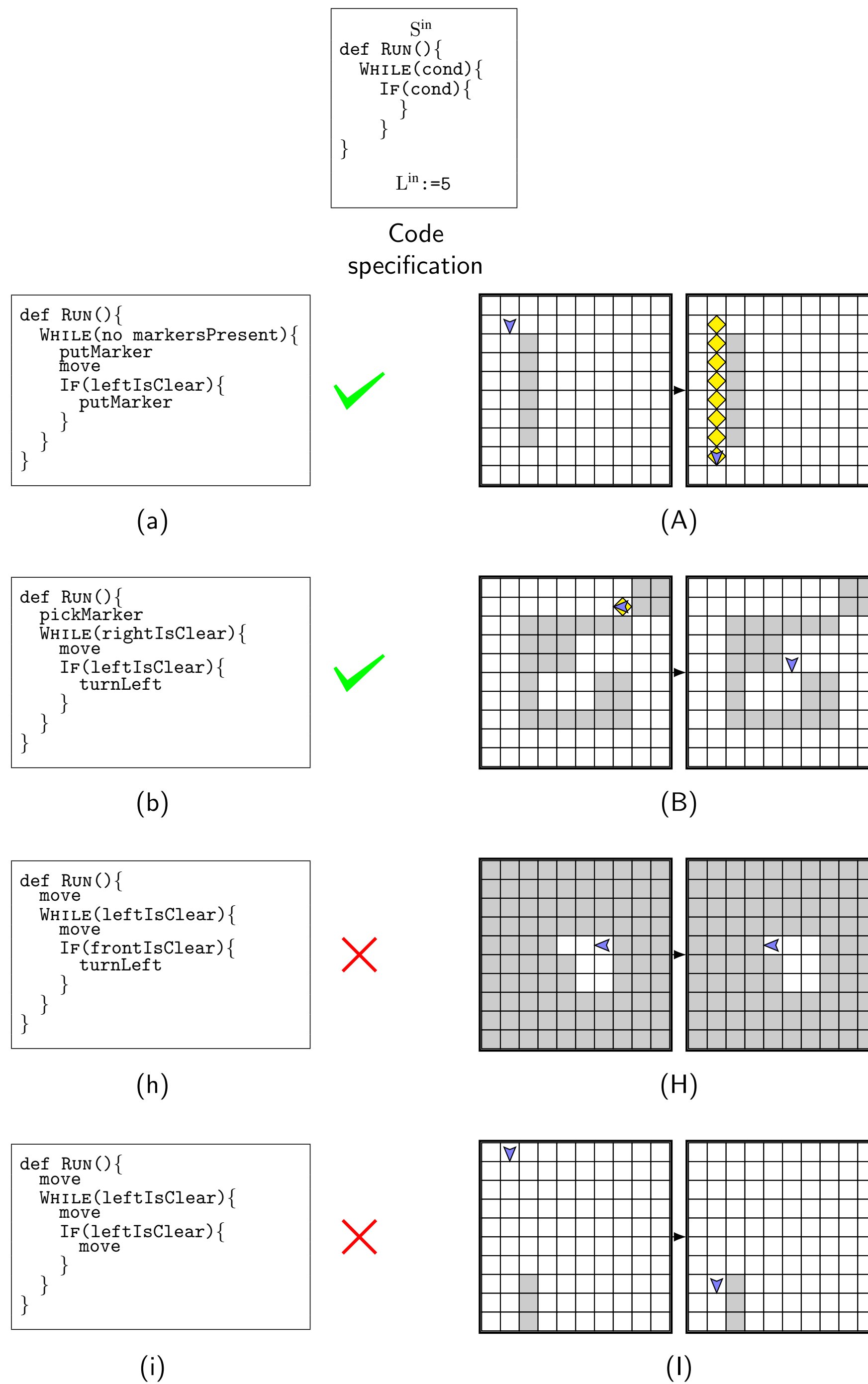


FIGURE 6.3: Reusing two good and two bad quality codes from the previous Figure 6.2 to illustrate the codes along with one of their suitable visual tasks. The good quality codes (a) and (b) can lead to interesting practice tasks as shown in (A) and (B) respectively, while the bad quality codes (h) and (i) do not lead to interesting tasks, as shown in (H) and (I) respectively.

Method	Type	Fraction of codes with $\delta \geq 0.5$			
		S_{d1}	S_{d2}	S_{d3}	**S_{all}**
(BC)Net	without multi-code optimization	0.80	0.46	0.30	**0.33**
(BC)Net	with multi-code optimization	0.90	0.69	0.48	**0.52**

Table 6.2: Illustration of difference in performance of (BC)Net with and without multi-code optimization.

Method	Type	Fraction of codes with $\delta \geq 0.5$			
		S_{d1}	S_{d2}	S_{d3}	**S_{all}**
(BC)Net	without beam search	0.87	0.43	0.24	**0.28**
(BC)Net	with beam search	0.90	0.69	0.48	**0.52**
(CodeGen)Net	without beam search	0.90	0.64	0.50	**0.52**
(CodeGen)Net	with beam search	0.90	0.78	0.71	**0.73**

Table 6.3: Illustration of difference in performance of the neural models with and without beam search.

Ablations.

We perform ablation studies to analyze the advantage of using multi-code optimization during behavior cloning, and integration of beam search to the neural frameworks and report the results in Table 6.2 and 6.3 respectively. We again use δ of 0.5 to validate the quality of all the generated codes.

First, in Table 6.2, (BC)Net with multi-code optimization is the same behavior cloned-only model described in Section 6.2, (BC)Net without multi-code optimization is also a behavior cloned-only model, but trained on a single reference dataset with the standard MLE optimization objective of 5.1. We observe that (BC)Net with multi-code optimization consistently outperforms the (BC)Net without multi-code optimization model, across different specification depth levels. This confirms our claims that, using multi-code optimization helps to learn the one-to-many mapping of specifications and codes, and results in diverse good quality code generation.

Second, in Table 6.3, we compare the performance of (BC)Net and (CodeGen)Net with and without beam search. We observe that, using beam search leads to better performance for either models. For (BC)Net the improvement is by a high score of 0.24 for all specifications, while for (CodeGen)Net is by a equally high score of 0.21. This result is in line with other areas such as neural program synthesis [10][49] or neural machine translation [30][70], which have also seen beam search as an important tool for performance improvement.

Chapter 7

Conclusions

7.1 Discussion

We presented a novel code generation framework (CodeGen)Net for synthesizing qualitative and diverse codes. The main novelties lie in using neural methodologies to automatically learn features of good quality codes leading to an automated code synthesizer. We empirically showed that our neural framework performs significantly better than random and constraint based techniques for a set of Karel based specifications. Furthermore, we demonstrated through several examples, that having such a high performing synthesizer can lead to new practice tasks useful for tutors and students in block-based visual programming environments. We believe that, this is an important step towards mitigating the problem of limited availability of practice exercises, and hence can vastly improve the success of pedagogy in block-based visual programming environments.

7.2 Limitations & Future Work

We successfully developed a neural framework for qualitative code synthesis. However, we still see some limitations in the current work, which can be improved upon in future works.

1. **Code quality scoring improvement.** Though the current scoring function is able to cover many of the things that humans associate to good quality codes, it is still handcrafted based on the environment knowledge and is not perfect. For example, in Figure 6.2, code (e) contains an *if* block with *leftIsClear* condition followed by a *turnRight* action: this does not seem a good quality code sequence intuitively, but still the code gets a good score by the quality scoring function. Automatically learning the code quality scoring function, or using a teacher/human-in-the-loop for quality feedback could be possible future directions that could cover better all the desired code quality features.

2. **Code diversity improvement.** Our framework has the ability to inherently learn to generate diverse codes due to the use of multi-code optimization and generation techniques. However, we see this is not always the case, and for some specifications the model can collapse to generate similar, multiple codes. This is increasingly seen after RL fine-tuning, possibly due to the well known problem of reward hacking [56]. A possible future work could include a combination of quality and diversity based reward as in [34], leading to further improvement in diversity among the generated codes.

3. **Use of more advanced neural techniques.** In this work, we focused on developing a neural pipeline to solve our problem. However, the performance of the models could be further improved by using improved neural techniques that have been developed recently. For example, using a transformer decoder for output code generation, or using more advanced RL algorithms such as PPO for RL optimization. It could be interesting to see the difference in performance after utilizing these techniques.

4. **Extending to other programming environments.** In this work, we restricted the empirical evaluation to the Karel environment. However, our methodology can easily be extended to other block-based environments such as HoC or Scratch. More broadly, it could be interesting to extend our methodology to general-purpose programming languages such as Python.

7.3 Broader Impact

We believe our work can help in improving the success of pedagogy in block-based visual programming environments. Our code generation framework generates a large number of good quality codes, which can be used in a variety of different ways in these environments. It can be used to generate new tasks corresponding to available expert tasks, which are similar in terms of the programming concepts they exercise, helping the students to master each concept. Or it can be used to create a personalized curriculum of problems based on current student knowledge, or a general curriculum based on evolution in complexity in terms of programming concepts; these can increase the efficiency of student learning process. Or just in the general sense, it can be used to create new practice tasks for students to learn from, or to provide new assignments/homework for students to check their knowledge.

List of Figures

Figure 1.1 Here we show a input code specification consisting of a code sketch and desired code length $(\mathrm{S}^{\mathrm{in}}, \mathrm{L}^{\mathrm{in}})$ on the left, and sample codes satisfying it from (a) to (f) on the right. The codes (a) to (d) are of bad quality, since they are semantically incorrect: (a) has consecutive action blocks which do not contribute to the output: *turnLeft* actions followed by *turnRight*, (b) has suboptimal action block sequence: three consecutive *turnLeft* which can be performed by a single *turnRight*, (c) and (d) always cause crash in execution if loops are entered: (c) has a condition to check for no markers followed by a *pickMarker* action, while (d) has a condition to check whether front is not clear followed by a *move* action. The codes (e) and (f) are semantically correct and are also good quality codes. 14

Figure 1.2 Good quality codes should also lead to new practice tasks useful for student learning. Here, task (i) is produced using code (e) and task (ii) is produced using code (f) from Figure 1.1. Both tasks can be useful for student learning and hence, code (e) and code (f) are considered as good quality codes. 15

Figure 1.3 Here we show another example. Input code specification is shown at the top, sample codes for it are shown on the left side and an example task produced from each code are shown on the right side. The codes (a) and (b) are bad quality codes, since they are semantically incorrect: (a) has a suboptimal *repeat* block: three *move* action blocks can be replaced by a single *move* action block with an increase in *repeat* count, (b) has a redundant sequence of action blocks which do not contribute to the output: *putMarker* action followed by *pickMarker*. (a) and (b) also do not lead to useful practice tasks as shown in (A) and (B) respectively. The codes (c) and (d) are good quality codes, since they are semantically correct, and lead to useful practice tasks as shown in (C) and (D) respectively. 17

Figure 2.1 An example Karel programming problem, *One ball in each spot* from the *Intro to Programming with Karel* course by *CodeHS.com* [2]. . . 20

Figure 2.2 The syntax of Karel DSL. 20

Figure 2.3 Different types of specifications that have been used in program synthesis along with their corresponding programs. (a) uses formal specification, (b) uses text based input-output examples, while (c) uses a visual input-output example. Corresponding programs are shown in (A), (B) and (C) respectively. 21

Figure 4.1 Karel Code DSL 31
Figure 4.2 Karel Sketch DSL 31
Figure 4.3 Reusing examples from Section 1 to define the preliminaries. Shown here is a specification (a), code (b) and visual task (c). 32

Figure 5.1 Our algorithm (CodeGen)Net takes as input a code specification $(\mathrm{S}^{\mathrm{in}}, \mathrm{L}^{\mathrm{in}})$ and generates a set of qualitative and diverse codes $\{\mathrm{C}_1^{\mathrm{out}}, .., \mathrm{C}_{\mathrm{k}}^{\mathrm{out}}, .., \mathrm{C}_{\mathrm{K}}^{\mathrm{out}}\}$. 35
Figure 5.2 Stage 1 of our technique-behavior cloning uses a set of expert demonstrations containing code specifications and good quality codes. . . 36
Figure 5.3 Stage 2 of our technique-fine tuning the neural model from stage 1 with a multi-target reinforcement learning framework. 37

Figure 6.1 Illustration of the 8 different templates that we work with. All our specifications are based on these 8 different templates, which are divided here according to their depth. Here $\mathrm{S_{d1}}$, $\mathrm{S_{d2}}[(i)$ to $(iii)]$ and $\mathrm{S_{d3}}[(i)$ to $(iv)]$ correspond to all sketches of depth 1,2 and 3 respectively. Note that, $\mathrm{D_1}$,$\mathrm{D_2}$ and $\mathrm{D_3}$ can correspond to any control block. 41
Figure 6.2 Illustration of 10 different codes (a) to (j) generated by our model (CodeGen)Net for the code specification shown at the top. In this case, the model generated 7 good quality codes (a) to (g), and 3 bad quality codes (h) to (j), for δ of 0.5. 44
Figure 6.3 Reusing two good and two bad quality codes from the previous Figure 6.2 to illustrate the codes along with one of their suitable visual tasks. The good quality codes (a) and (b) can lead to interesting practice tasks as shown in (A) and (B) respectively, while the bad quality codes (h) and (i) do not lead to interesting tasks, as shown in (H) and (I) respectively. 45

Figure A.1 Illustration of 10 different codes (a) to (j) generated by the (Rand) baseline for the code specification shown at the top. In this case, the model generated 2 good quality codes (a) and (b), and 8 bad quality codes (c) to (j), for δ of 0.5. 63
Figure A.2 Illustration of 10 different codes (a) to (j) generated by the (Rand)Cstr baseline for the code specification shown at the top. In this case, the model generated 5 good quality codes (a) to (e), and 5 bad quality codes (f) to (j), for δ of 0.5. 64

Figure A.3 Illustration of 10 different codes (a) to (j) generated by our neural framework for the code specification shown at the top. In this case, the model generated 8 good quality codes (a) to (h), and 2 bad quality codes (i) and (j), for δ of 0.5. 66

Figure A.4 Reusing two good and two bad quality codes from the previous Figure A.3 to illustrate the codes along with one of their suitable visual tasks. The good quality codes (a) and (b) can lead to interesting practice tasks as shown in (A) and (B) respectively, while the bad quality codes (i) and (j) do not lead to interesting tasks, as shown in (I) and (J) respectively. 67

List of Tables

Table 6.1 Results for the Karel specifications for two different quality thresholds: $\delta = 0.5$ and $\delta = 0.7$. Results are shown separately for different specification templates S_{dX} and for all specifications combined S_{all}. See Section 6.3 for more description. 42
Table 6.2 Illustration of difference in performance of (BC)Net with and without multi-code optimization. 46
Table 6.3 Illustration of difference in performance of the neural models with and without beam search. 46

Bibliography

[1] Stanford Karel IDE. https://stanford.edu/~cpiech/karel/learn.html, . Accessed: 2023-03-29.

[2] CodeHS platform. Intro to Programming with Karel the Dog. . https://codehs.com/info/curriculum/introkarel, . Accessed: 2023-03-31.

[3] Hour of Code. http://hourofcode.codehs.com/, . Accessed: 2023-03-07.

[4] Code.org platform. Hour of Code: Classic Maze Challenge. . https://studio.code.org/s/hourofcode, . Accessed: 2023-03-31.

[5] Stanford CS106A course page. https://see.stanford.edu/Course/CS106A, . Accessed: 2023-03-07.

[6] Karel programming language documentation. http://mormegil.wz.cz/prog/karel/prog_doc.htm, . Accessed: 2023-03-07.

[7] Umair Ahmed, Sumit Gulwani, and Amey Karkare. Automatically generating problems and solutions for natural deduction. pages 1968–1975, 08 2013.

[8] Umair Z. Ahmed, Maria Christakis, Aleksandr Efremov, Nigel Fernandez, Ahana Ghosh, Abhik Roychoudhury, and Adish Singla. Synthesizing tasks for block-based programming. *CoRR*, abs/2006.16913, 2020. URL https://arxiv.org/abs/2006.16913.

[9] Chris Alvin, Sumit Gulwani, Rupak Majumdar, and Supratik Mukhopadhyay. Synthesis of geometry proof problems. *Proceedings of the National Conference on Artificial Intelligence*, 1:245–252, 06 2014. doi: 10.1609/aaai.v28i1.8745.

[10] Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. In *International Conference on Learning Representations(ICLR)*. arXiv, 2018. URL https://arxiv.org/abs/1805.04276.

[11] R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, jan 1977. ISSN 0004-5411. doi: 10.1145/321992.321996. URL https://doi.org/10.1145/321992.321996.

[12] Xinyun Chen, Chang Liu, and Dawn Xiaodong Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2018.

[13] Xinyun Chen, Dawn Song, and Yuandong Tian. Latent execution for neural program synthesis beyond domain-specific languages. *CoRR*, abs/2107.00101, 2021. URL https://arxiv.org/abs/2107.00101.

[14] Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky, editors. *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, MA, USA, 1993. ISBN 0262032139.

[15] Jacob Devlin, Rudy R Bunel, Rishabh Singh, Matthew Hausknecht, and Pushmeet Kohli. Neural program meta-induction. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper/2017/file/3bf55bbad370a8fcad1d09b005e278c2-Paper.pdf.

[16] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy I/O. *CoRR*, abs/1703.07469, 2017. URL http://arxiv.org/abs/1703.07469.

[17] Xuguang Duan, Qi Wu, Chuang Gan, Yiwei Zhang, Wenbing Huang, Anton van den Hengel, and Wenwu Zhu. Watch, reason and code: Learning to represent videos using program. In *Proceedings of the 27th ACM International Conference on Multimedia*, MM '19, page 1543–1551, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368896. doi: 10.1145/3343031.3351094. URL https://doi.org/10.1145/3343031.3351094.

[18] Aleksandr Efremov, Ahana Ghosh, and Adish Kumar Singla. Zero-shot learning of hint policy via reinforcement learning and program synthesis. In *Educational Data Mining*, 2020.

[19] Abhay Garg, Anand Sriraman, Kunal Pagarey, and Shirish Karande. Are transformers all that karel needs? In *Advances in Programming Languages and Neurosymbolic Systems Workshop*, 2021. URL https://openreview.net/forum?id=qGDIkNmWydG.

[20] Ahana Ghosh, Sebastian Tschiatschek, Sam Devlin, and Adish Singla. Adaptive scaffolding in block-based programming via synthesizing new tasks as pop quizzes, 2023.

[21] Cordell Green. Application of theorem proving to problem solving. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, IJCAI'69, page 219–239, San Francisco, CA, USA, 1969. Morgan Kaufmann Publishers Inc.

[22] Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, aug 2012. ISSN 0001-0782. doi: 10.1145/2240236.2240260. URL https://doi.org/10.1145/2240236.2240260.

[23] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017. ISSN 2325-1107. doi: 10.1561/2500000010. URL http://dx.doi.org/10.1561/2500000010.

[24] Kavi Gupta, Peter Ebert Christensen, Xinyun Chen, and Dawn Song. Synthesize, execute and debug: Learning to repair for neural program synthesis. *CoRR*, abs/2007.08095, 2020. URL https://arxiv.org/abs/2007.08095.

[25] Joy He-Yueya and Adish Singla. Quizzing policy using reinforcement learning for inferring the student knowledge state. *International Educational Data Mining Society*, 2021.

[26] Di Huang, Rui Zhang, Xing Hu, Xishan Zhang, Pengwei Jin, Nan Li, Zidong Du, Qi Guo, and Yunji Chen. Neural program synthesis with query, 2022.

[27] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7): 385–394, jul 1976. ISSN 0001-0782. doi: 10.1145/360248.360252. URL https://doi.org/10.1145/360248.360252.

[28] Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2022.

[29] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[30] Marie-Anne Lachaux, Armand Joulin, and Guillaume Lample. Target conditioning for one-to-many generation. pages 2853–2862, 01 2020. doi: 10.18653/v1/2020.findings-emnlp.256.

[31] Yann LeCun, Y. Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521:436–44, 05 2015. doi: 10.1038/nature14539.

[32] Jiwei Li, Will Monroe, and Dan Jurafsky. A simple, fast diverse decoding algorithm for neural generation, 2016.

[33] H. Lieberman. Your wish is my command: Programming by example. 01 2001.

[34] Huan Lin, Baosong Yang, Liang Yao, Dayiheng Liu, Haibo Zhang, Jun Xie, Min Zhang, and Jinsong Su. Bridging the gap between training and inference: Multi-candidate optimization for diverse neural machine translation. In *Findings of the Association for Computational Linguistics: NAACL 2022*, pages 2622–2632, Seattle, United States, July 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.findings-naacl.200. URL https://aclanthology.org/2022.findings-naacl.200.

[35] Z. Manna and R. Waldinger. Synthesis: Dreams → programs. *IEEE Transactions on Software Engineering*, SE-5(4):294–328, 1979. doi: 10.1109/TSE.1979.234198.

[36] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, jan 1980. ISSN 0164-0925. doi: 10.1145/357084.357090. URL `https://doi.org/10.1145/357084.357090`.

[37] Zohar Manna and Richard J. Waldinger. Toward automatic program synthesis. *Commun. ACM*, 14(3):151–165, mar 1971. ISSN 0001-0782. doi: 10.1145/362566.362568. URL `https://doi.org/10.1145/362566.362568`.

[38] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL `http://arxiv.org/abs/1312.5602`.

[39] Benjamin Paaßen, Barbara Hammer, Thomas William Price, Tiffany Barnes, Sebastian Gross, and Niels Pinkwart. The continuous hint factory - providing hints in vast and sparsely populated edit distance spaces. *CoRR*, abs/1708.06564, 2017. URL `http://arxiv.org/abs/1708.06564`.

[40] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. *CoRR*, abs/1611.01855, 2016. URL `http://arxiv.org/abs/1611.01855`.

[41] Richard E. Pattis. *Karel the Robot: A Gentle Introduction to the Art of Programming.* John Wiley amp; Sons, Inc., USA, 1st edition, 1981. ISBN 0471089281.

[42] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. Learning program embeddings to propagate feedback on student code, 2015.

[43] Chris Piech, Mehran Sahami, Jonathan Huang, and Leonidas Guibas. Autonomously generating hints by inferring problem solving policies. In *Proceedings of the Second (2015) ACM Conference on Learning @ Scale*, L@S '15, page 195–204, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334112. doi: 10.1145/2724660.2724668. URL `https://doi.org/10.1145/2724660.2724668`.

[44] Thomas Price, Rui Zhi, and Tiffany Barnes. Evaluation of a data-driven feedback algorithm for open-ended programming. 06 2017.

[45] Thomas W. Price and Tiffany Barnes. Position paper: Block-based programming should offer intelligent support for learners. In *2017 IEEE Blocks and Beyond Workshop (BB)*, pages 65–68, 2017. doi: 10.1109/BLOCKS.2017.8120414.

[46] Thomas W. Price, Yihuan Dong, and Dragan Lipovac. Isnap: Towards intelligent tutoring in novice programming environments. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, page 483–488, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450346986. doi: 10.1145/3017680.3017762. URL `https://doi.org/10.1145/3017680.3017762`.

[47] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: Programming for all. *Commun. ACM*, 52(11):60–67, nov 2009. ISSN 0001-0782. doi: 10.1145/1592761.1592779. URL `https://doi.org/10.1145/1592761.1592779`.

[48] Tianxiao Shen, Myle Ott, Michael Auli, and Marc'Aurelio Ranzato. Mixture models for diverse machine translation: Tricks of the trade. *CoRR*, abs/1902.07816, 2019. URL `http://arxiv.org/abs/1902.07816`.

[49] Eui Chul Shin, Illia Polosukhin, and Dawn Song. Improving neural program synthesis with inferred execution traces. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL `https://proceedings.neurips.cc/paper/2018/file/7776e88b0c189539098176589250bcba-Paper.pdf`.

[50] Richard Shin, Illia Polosukhin, and Dawn Xiaodong Song. Towards specification-directed program repair. In *International Conference on Learning Representations*, 2018.

[51] Richard Shin, Neel Kant, Kavi Gupta, Christopher Bender, Brandon Trabucco, Rishabh Singh, and Dawn Song. Synthetic datasets for neural program synthesis. *CoRR*, abs/1912.12345, 2019. URL `http://arxiv.org/abs/1912.12345`.

[52] Raphael Shu, Hideki Nakayama, and Kyunghyun Cho. Generating diverse translations with sentence codes. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 1823–1827, Florence, Italy, July 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1177. URL `https://aclanthology.org/P19-1177`.

[53] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. *SIGPLAN Not.*, 48 (6):15–26, jun 2013. ISSN 0362-1340. doi: 10.1145/2499370.2462195. URL `https://doi.org/10.1145/2499370.2462195`.

[54] Rohit Singh, Sumit Gulwani, and Sriram Rajamani. Automatically generating algebra problems. *Proceedings of the National Conference on Artificial Intelligence*, 2, 01 2012. doi: 10.1609/aaai.v26i1.8341.

[55] Adish Singla, Anna N. Rafferty, Goran Radanovic, and Neil T. Heffernan. Reinforcement learning for education: Opportunities and challenges, 2021.

[56] Joar Skalse, Nikolaus H. R. Howe, Dmitrii Krasheninnikov, and David Krueger. Defining and characterizing reward hacking, 2022.

[57] Alexander Suh and Yuval Timen. Creating synthetic datasets via evolution for neural program synthesis. *CoRR*, abs/2003.10485, 2020. URL `https://arxiv.org/abs/2003.10485`.

[58] Shao-Hua Sun, Hyeonwoo Noh, Sriram Somasundaram, and Joseph Lim. Neural program synthesis from diverse demonstration videos. In *Proceedings of the 35th International Conference on Machine Learning*, 2018.

[59] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. URL `http://incompleteideas.net/book/the-book-2nd.html`.

[60] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansõur. Policy gradient methods for reinforcement learning with function approximation. In S. Solla, T. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 1999. URL `https://proceedings.neurips.cc/paper/1999/file/464d828b85b0bed98e80ade0\a5c43b0f-Paper.pdf`.

[61] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc. ISBN 9781510860964.

[62] Ashwin K. Vijayakumar, Michael Cogswell, Ramprasaath R. Selvaraju, Qing Sun, Stefan Lee, David J. Crandall, and Dhruv Batra. Diverse beam search for improved description of complex scenes. In *AAAI Conference on Artificial Intelligence*, 2018.

[63] Richard J. Waldinger and Richard C. T. Lee. Prow: A step toward automatic program writing. IJCAI'69, page 241–252, San Francisco, CA, USA, 1969. Morgan Kaufmann Publishers Inc.

[64] L. Wang, Angela Sy, Larry Liu, and Chris Piech. Learning to represent student knowledge on programming exercises using deep learning. In *Educational Data Mining*, 2017.

[65] David Weintrop and Uri Wilensky. To block or not to block, that is the question: Students' perceptions of blocks-based programming. IDC '15, page 199–208, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450335904. doi: 10.1145/2771839.2771860. URL `https://doi.org/10.1145/2771839.2771860`.

[66] David Weintrop and Uri Wilensky. Comparing block-based and text-based programming in high school computer science classrooms. *ACM Trans. Comput. Educ.*, 18 (1), oct 2017. doi: 10.1145/3089799. URL https://doi.org/10.1145/3089799.

[67] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3–4):229–256, may 1992. ISSN 0885-6125. doi: 10.1007/BF00992696. URL https://doi.org/10.1007/BF00992696.

[68] Mike Wu, Milan Mosse, Noah D. Goodman, and Chris Piech. Zero shot learning for code education: Rubric sampling with deep learning inference. *CoRR*, abs/1809.01357, 2018. URL http://arxiv.org/abs/1809.01357.

[69] Xuanfu Wu, Yang Feng, and Chenze Shao. Generating diverse translation from model distribution with dropout. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1088–1097, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.82. URL https://aclanthology.org/2020.emnlp-main.82.

[70] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation, 2016. URL https://arxiv.org/abs/1609.08144.

[71] Jooyong Yi, Umair Z. Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. A feasibility study of using automated program repair for introductory programming assignments. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 740–751, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450351058. doi: 10.1145/3106237.3106262. URL https://doi.org/10.1145/3106237.3106262.

[72] Rui Zhi, Thomas W. Price, Samiha Marwan, Alexandra Milliken, Tiffany Barnes, and Min Chi. Exploring the impact of worked examples in a novice programming environment. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, page 98–104, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450358903. doi: 10.1145/3287324.3287385. URL https://doi.org/10.1145/3287324.3287385.

Appendix A

Extra Background & Results

A.1 Recap of REINFORCE Algorithm [60]

Consider a standard reinforcement learning framework in which a learning agent interacts with a Markov Decision Process(MDP) [59]. A policy in such a framework defines the way in which agent should act given different environment states. Policies can be either non-parametric or parametric. Consider a parameterized policy with a parameter vector $\boldsymbol{\theta}$, with the probability of taking an action a in an environment state s defined as $\pi(a|s,\boldsymbol{\theta})$. For parameterized policies, a popular way to learn the policy parameters is based on the gradient of a cost function. That is, if $\mathcal{L}(\theta)$ is the cost function, then the parameters can be learnt using the update,

$$\theta_t = \theta_{t-1} + \alpha \nabla \mathcal{L}(\theta_{t-1}) \tag{A.1}$$

where θ_t are the parameters of the policy at time step t, and α is the learning rate. To obtain the gradient of the cost function, policy gradient theorem can be used [60],

$$\nabla \mathcal{L}(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a Q_\pi(s,a) \nabla \pi(a \mid s, \boldsymbol{\theta}) \tag{A.2}$$

where $Q_\pi(s,a)$ is the action-value estimate of being in a state s and following action a from that state, and $\mu(s)$ is the state distribution under π.

This expression is intractable to compute for large state-action spaces, as it requires summing over all possible states and actions. Instead, this update can be approximated by sampling at each time step, such that the expectation of the sample gradient is proportional to the actual gradient of the cost function as a function of the parameter θ. This is called the REINFORCE trick [67], and can be expressed as,

$$\nabla \mathcal{L}(\boldsymbol{\theta}_{t-1}) = \alpha G_{t-1} \frac{\nabla \pi\left(A_{t-1} \mid S_{t-1}, \boldsymbol{\theta}_{t-1}\right)}{\pi\left(A_{t-1} \mid S_{t-1}, \boldsymbol{\theta}_{t-1}\right)}. \tag{A.3}$$

where G_{t-1} is the return, A_{t-1} is the action and S_{t-1} is the state at time $(t-1)$.

Using this sample as part of the stochastic gradient ascent update of equation A.1 yields,

$$\theta_t = \theta_{t-1} + \alpha G_{t-1} \frac{\nabla \pi \left(A_{t-1} \mid S_{t-1}, \boldsymbol{\theta}_{t-1}\right)}{\pi \left(A_{t-1} \mid S_{t-1}, \boldsymbol{\theta}_{t-1}\right)}. \tag{A.4}$$

A.2 Analysis of the Baselines

In this section, we investigate the difficulties faced by the baselines to generate good quality codes. We do this by analyzing the codes generated by the baselines (Rand) and (Rand)Cstr for the same specification of $\mathrm{S}_{\text{depth}} = 3$, $\mathrm{S}_{\text{struct}} = \{\text{Run}\{\text{WHILE}\{\text{IF}\}\}\}$ and $\mathrm{L}^{\text{in}} = 5$, that we analyzed for our neural framework in Section 6.3. The codes generated by the (Rand) baseline are shown in the Figure A.1 and the codes generated by the (Rand)Cstr baseline are shown in the Figure A.2. We use δ of 0.5 for both the baselines.

(Rand) generates only 2 codes((a) and (b) in Figure A.1) out of the 10 syntactically correct codes it was asked to generate, that satisfy the threshold. This is in agreement with our claim that only a small amount of syntactically correct codes are also semantically correct. The randomly generated codes can suffer from basic quality issues, such as redundant sequence of action blocks *turnLeft-turnRight* or *putMarker-pickMarker*((c) and (d) in Figure A.1), using a *while* block with a *no markersPresent* condition just after a *putMarker* action ((e) in Figure A.1) etc. Note that, even the 2 codes that satisfy the threshold 0.5 are of not great quality, since they always result in simple agent actions, and will not be able to satisfy a higher threshold value.

(Rand)Cstr is able to improve upon (Rand) and generates 5 codes((a) to (e) in Figure A.2) out of the 10 codes it was asked to generate, that satisfy the threshold. This suggests that the use of constraints can resolve some quality issues; however developing handcrafted constraints that can cover all the possible code quality features becomes a difficult challenge. Note that, it might be possible to add some more constraints then we use, but it becomes increasingly difficult to introduce new constraints as the complexity of the specifications keeps increasing. Further, another example can be observed here which indicates limitations in the quality scoring function: code (c) in Figure A.2 cannot lead to any agent movement within the grid, but is still given a high value by the scoring function.

```
S^in
def Run(){
  While(cond){
    If(cond){
    }
  }
}
L^in:=5
```

Code specification

(a)
```
def Run(){
  move
  While(not rightIsClear){
    move
    If(not leftIsClear){
      move
    }
  }
}
```

(b)
```
def Run(){
  move
  While(not leftIsClear){
    turnRight
    If(frontIsClear){
      putMarker
    }
  }
}
```

(c)
```
def Run(){
  While(not rightIsClear){
    If(not rightIsClear){
      turnLeft
      turnRight
      turnRight
    }
  }
}
```

(d)
```
def Run(){
  turnLeft
  While(rightIsClear){
    If(no markersPresent){
      putMarker
      pickMarker
    }
  }
}
```

(e)
```
def Run(){
  putMarker
  While(no markersPresent){
    putMarker
    If(frontIsClear){
      move
    }
  }
}
```

(f)
```
def Run(){
  While(markersPresent){
    putMarker
    If(not frontIsClear){
      putMarker
      turnRight
    }
  }
}
```

(g)
```
def Run(){
  pickMarker
  While(not frontIsClear){
    turnLeft
    If(not rightIsClear){
      move
    }
  }
}
```

(h)
```
def Run(){
  While(no markersPresent){
    If(not rightIsClear){
      putMarker
      move
      turnLeft
    }
  }
}
```

(i)
```
def Run(){
  move
  pickMarker
  While(no markersPresent){
    If(not rightIsClear){
      putMarker
    }
  }
}
```

(j)
```
def Run(){
  While(not rightIsClear){
    If(not rightIsClear){
      pickMarker
      turnLeft
      turnLeft
    }
  }
}
```

FIGURE A.1: Illustration of 10 different codes (a) to (j) generated by the (Rand) baseline for the code specification shown at the top. In this case, the model generated 2 good quality codes (a) and (b), and 8 bad quality codes (c) to (j), for δ of 0.5.

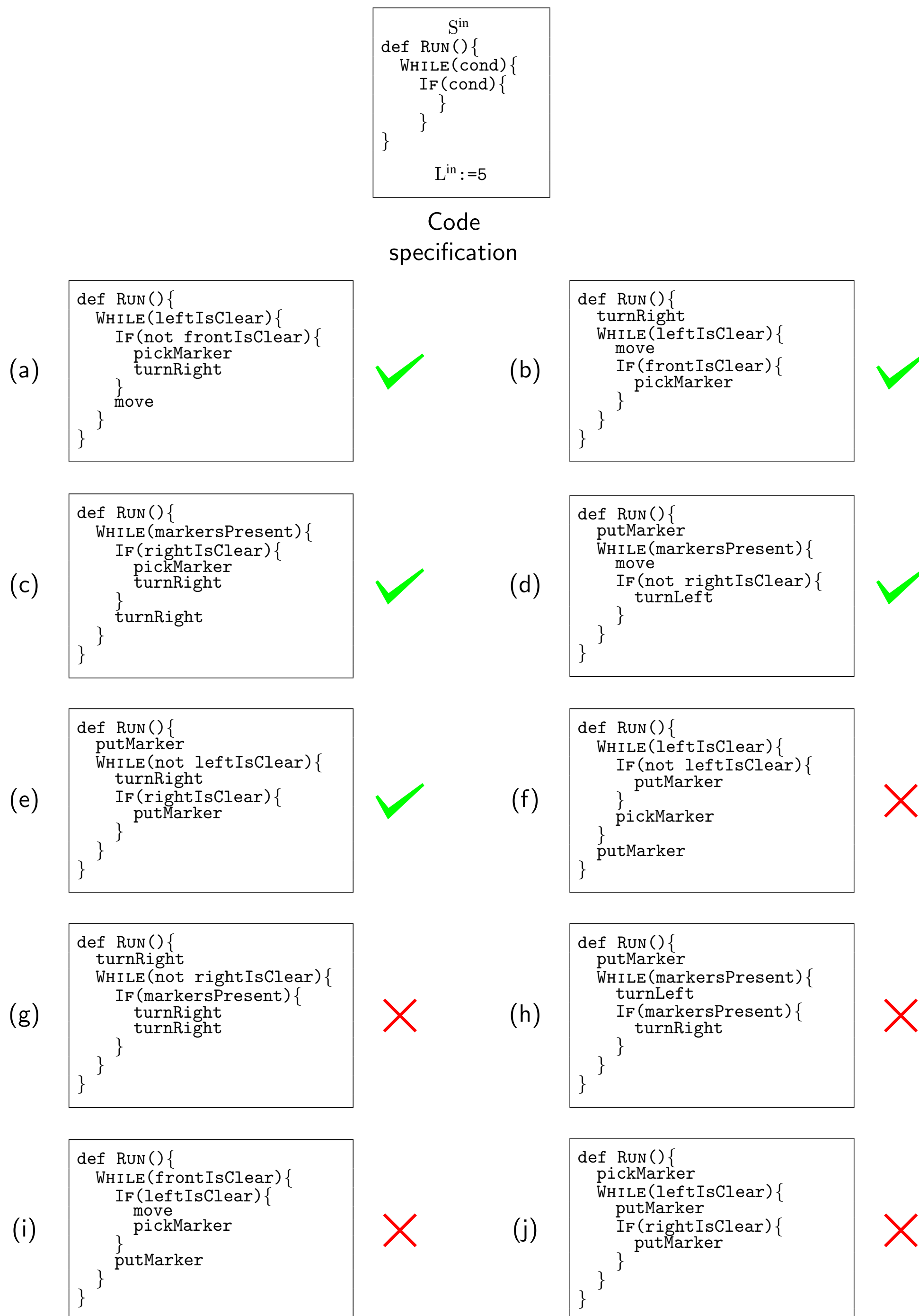


FIGURE A.2: Illustration of 10 different codes (a) to (j) generated by the (Rand)Cstr baseline for the code specification shown at the top. In this case, the model generated 5 good quality codes (a) to (e), and 5 bad quality codes (f) to (j), for δ of 0.5.

A.3 More Qualitative Results

Here we perform qualitative analysis for another specification with $\mathrm{S}_{\mathrm{depth}} = 2$, $\mathrm{S}_{\mathrm{struct}} = \{\mathrm{Run}\{\mathrm{WHILE}\}\}$ and $\mathrm{L}^{\mathrm{in}} = 4$. The codes generated by our neural framework are shown in the Figure A.3 from (a) to (j). We also show example suitable visual tasks (A), (B) corresponding to two good quality codes (a), (b), and (I), (J) corresponding to two bad quality codes (I), (J) in Figure A.4. We use δ of 0.5.

The model is able to satisfy the threshold for 8 out of the 10 codes(code (a) to code (h)), it was asked to generate. This is close to the 0.78 overall score for all the specifications of depth 2, as shown in Table 6.1. These 8 codes do not have any visible semantic irregularities, and can lead to interesting tasks, examples of which can be seen in the two tasks (A) and (B) produced using codes (a) and (b); these can be used as new practice tasks for student learning. However, these codes seem to have limited diversity, considering they use only 3 of the 5 available action blocks {*move*, *turnLeft*, *putMarker*} as well as 4 of the 8 available condition blocks {*frontIsClear*, *rightIsClear*, *leftIsClear*, *no markersPresent*}.

On the other hand, code (i) and code (j) are semantically incorrect: code (i) has a suboptimal *while* block: three *move* action blocks can be replaced by a single *move* action block, and the *while* loop in code (j) is either redundant or a never ending loop depending on the environment. Moreover, code(i) can only lead to straight agent movements within the grid, and code (j) allows only few number of agent actions before termination, and hence these codes also cannot lead to any interesting task. This can be seen through the corresponding visual tasks (I) and (J). Hence these codes are bad quality codes and do not satisfy the quality threshold δ.

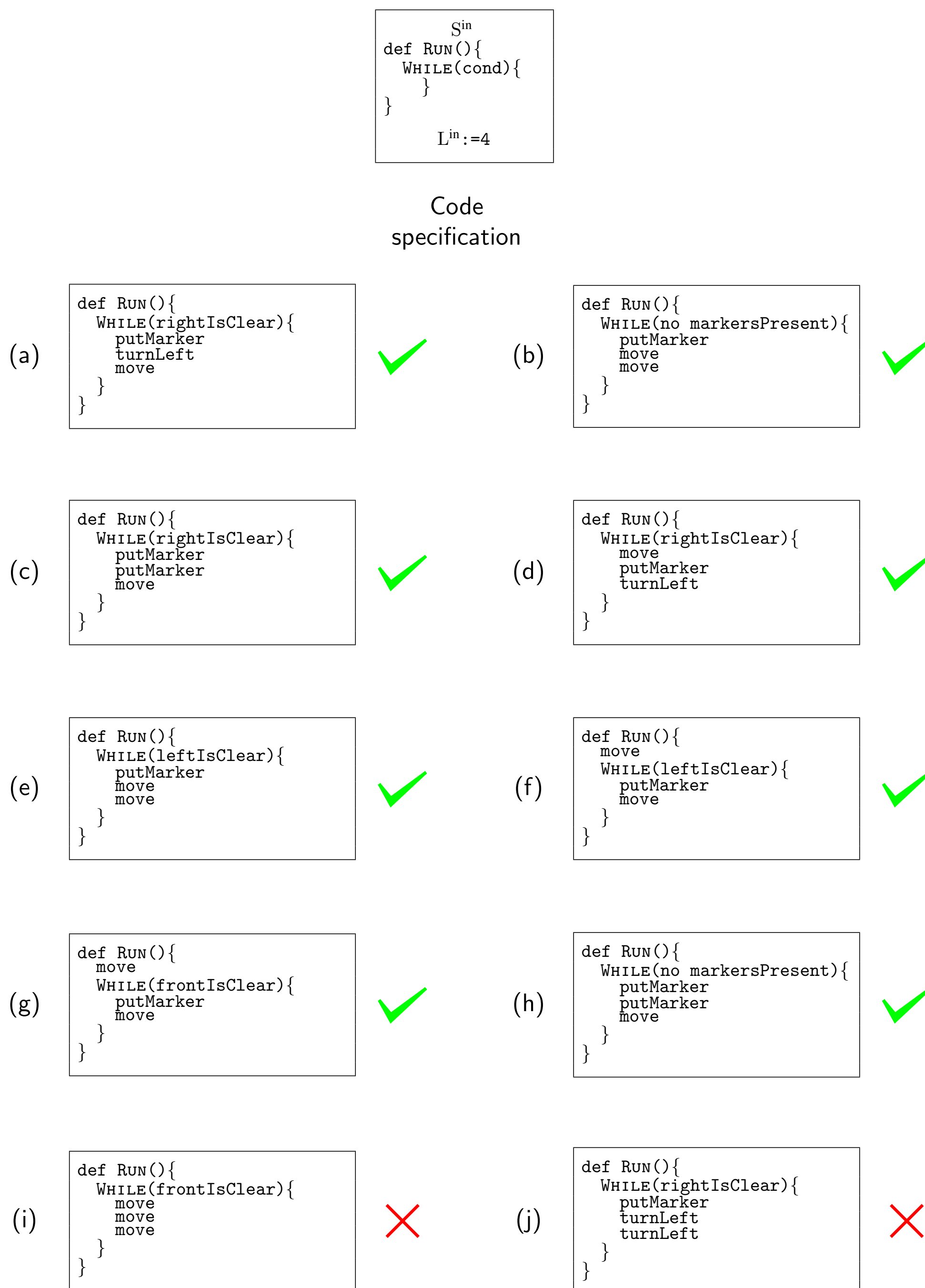


FIGURE A.3: Illustration of 10 different codes (a) to (j) generated by our neural framework for the code specification shown at the top. In this case, the model generated 8 good quality codes (a) to (h), and 2 bad quality codes (i) and (j), for δ of 0.5.

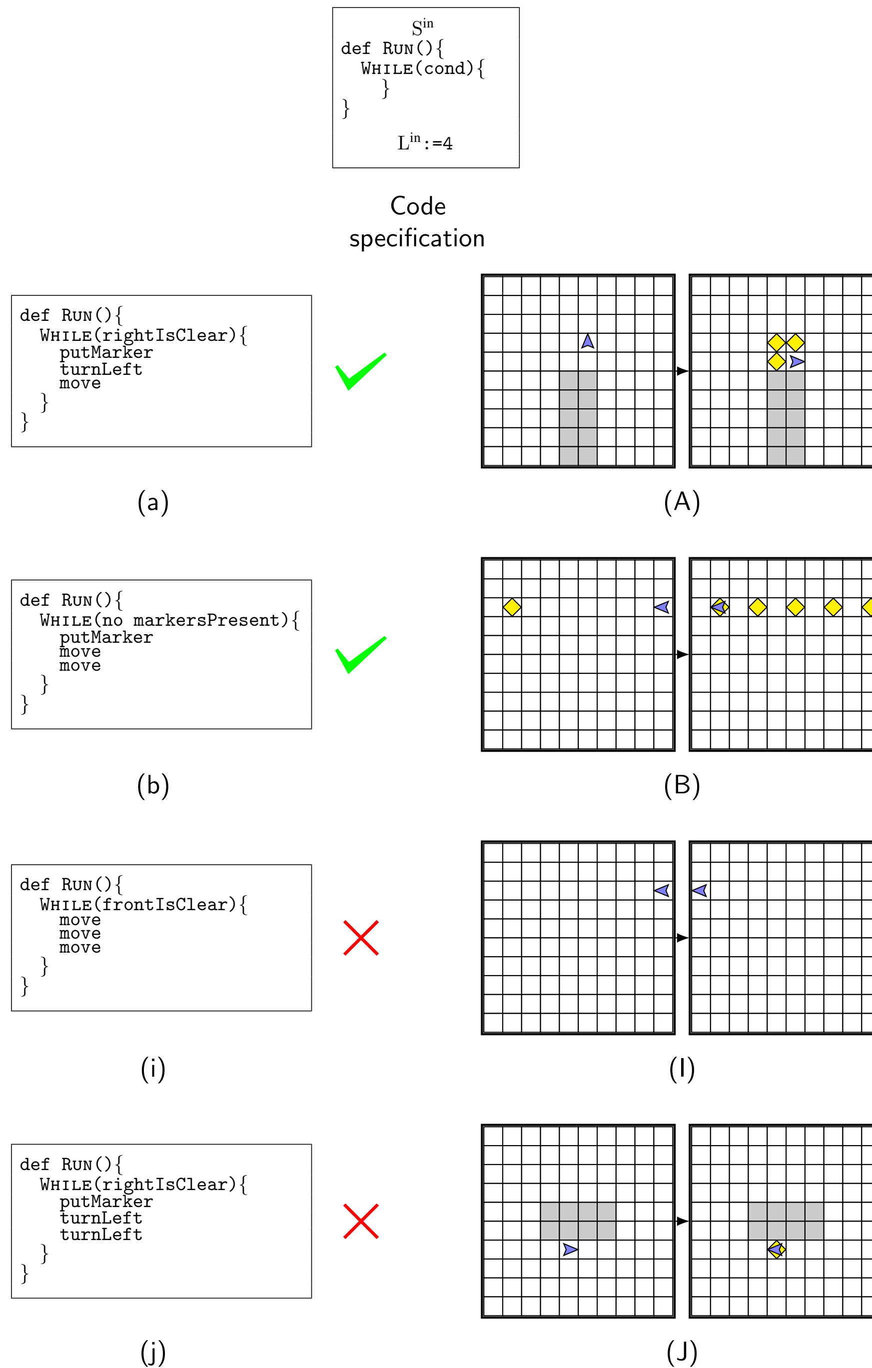


FIGURE A.4: Reusing two good and two bad quality codes from the previous Figure A.3 to illustrate the codes along with one of their suitable visual tasks. The good quality codes (a) and (b) can lead to interesting practice tasks as shown in (A) and (B) respectively, while the bad quality codes (i) and (j) do not lead to interesting tasks, as shown in (I) and (J) respectively.