Authoring Feedback for Novice Programmers in a Block-based Language

Luke Gusukuma, Dennis Kafura, Austin (Cory) Bart
Department of Computer Science
Virginia Tech
USA
{lukesg08, kafura, acbart} @vt.edu

***Abstract*— We present a block-based language for specifying feedback to novice learners about the programs they are constructing in a block-based programming language. In addition to feedback based on run-time and output checking, we are particularly interested in immediate feedback: corrective guidance given as the program is being constructed. Immediate feedback is a natural extension of the block-based language philosophy. Block-based languages prevent by design certain types of mistakes in all cases. Immediate feedback guides against, without fully preventing, problem-specific mistakes (i.e., constructions that are erroneous in only some cases). A feedback specification contains a block pattern and a set of actions that can be taken whenever the corresponding pattern is present or absent in the student's block program for a given problem. The paper illustrates the language through several examples derived from misconceptions found in the block-based programs of students taking a university-level Computational Thinking class. The feasibility of the proposed approach is shown by the translation of a specification using an evolving programmatic interface in BlockPy, a dual block/text programming environment for a subset of Python.**

Keywords—Immediate feedback, formative feedback, block-based languages, novice programming environments

I. INTRODUCTION

Feedback is an important element in theories of learning. Feedback played a significant role in each of five theories of learning in a meta-analysis of learning [1]. In [2], the importance of feedback as a means of shaping the learners behavior toward a desired performance (behaviorism), or assisting the individual learner in creating desired internal cognitive structures (cognitivism) is also noted. Beyond its role in supporting knowledge acquisition, feedback is also seen as a means of sustaining the motivation of students to learn [3] or guiding students to be self-regulated learners [4].

Our work considers feedback in block-based languages. By design, block-based languages obviate some forms of feedback (e.g. syntactic errors) and offer implicit feedback through the appearance (e.g., shape or color) and actions of blocks (e.g., two blocks that cannot be snapped together because the combination is not meaningful). However, this leaves a significant need for feedback related to the deeper cognitive barriers in learning to program. The envisioned research agenda is multidimensional, dependent not only on the block-based language and the form of the feedback, but also on the learners, and the context [3]. The feedback appropriate for children making games will be different from that needed by university-level students creating visualizations in a data-science context. One recent example in line with this agenda is a hint generation system for iSnap [5]. This proposal also aligns with the historical reflection on AgentSheets to "push blocks programming beyond syntax toward the support of semantics and even pragmatics" and is in the spirit of feedback via "Conversational Programming" [6]. In this paper we present another point in the spectrum.

Among the many kinds of feedback, this paper considers immediate formative feedback. Formative feedback can be defined as: "information communicated to the learner that is intended to modify the learner's thinking or behavior for the purpose of improving learning" and includes forms such as verification, explanation, hints, and worked examples [3]. Feedback can also be presented at different times. Submission-based feedback occurs when the student explicitly submits work for evaluation (e.g., runs a program against a supplied battery of test cases). Immediate feedback occurs as close in time to a student's exhibiting a correct or incorrect part of the work.

Our work with immediate feedback can be characterized by placing it in the context of recent work on hint generation systems. Hint generation systems (e.g., [7] [8]) have as their goal making suggestions to students on how to change the code they have submitted so that it is more likely to be correct. The challenge faced these systems is that the solution space is large as a result of the open-ended nature of the solutions. However, these systems can leverage two factors: off-line analysis using substantial computing resources and a large number of previous submissions to analyze. The trade-off made is that no effort is required of instructors but no explanation is provided to students, meaning that only an (at best) indirect improvement in student cognitive gains. Our goal is to provide immediate formative feedback to novice programmers (non-majors in a Computational Thinking class) authored by an expert (e.g., a teacher). The challenge here is that the analysis of student code must be highly efficient so as not to cause sluggish response. However, we can leverage two factors: the solution space is limited (due to the limited complexity of assignments for novice programmers) and students tend to have the same misconceptions (hence, make the same programming mistakes). The trade-off in work is that instructors must invest time in

This work is supported in part by National Science Foundation grants DUE 1624320, DUE 1444094, and DGE 0822220.

authoring formative feedback in the hope of higher (or more efficient) cognitive gains by students.

A block-based language for specifying feedback is similar in spirit to other work that uses a block-based approach for non-programming purposes as for example, in relational algebra [9] or database queries [10]. Our work also overlaps with hint systems for novice learners in a block-based language [5].

A specification language is important because it enhances the productivity of the expert instructor to produce high quality feedback without concern for enforcement details and can assist in organizing and maintaining the feedback over time. In a block-based programming environment expressing the specification in a block-based language is useful because the programming patterns to be detected are themselves arrangements of blocks. The feasibility of this specification language is shown by illustrating an enforcement mechanism in BlockPy [11].

II. Examples of Errors and Feedback

Example novice programs are used to illustrate the requirements for the description language. These programs are solutions constructed by students in an Introduction to Computational Thinking class. The misconceptions illustrated by these programs are representative of a significant group of students who had programs with the same misconception and are, thus, good targets for feedback. The example programs resulted from an automated analysis of all student submissions to identify programs with similar characteristics. These programs were solutions to the following problem: "The weather dataset provides information regarding weather records for two months over the summer. You have been given a weather block that returns a list of temperatures for Blacksburg. Iterate through the list and count the number of temperatures in the list."

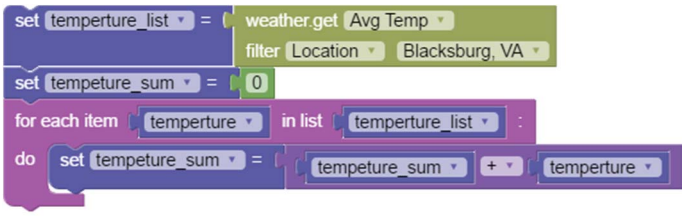

Figure 1 Example Student Error

A student's incorrect program for this problem is shown in Figure 1. This program illustrates a misunderstanding of the problem statement: rather than counting the number of temperatures in the list the student is summing the temperatures in the list. This was a commonly occurring mistake, one made by about one-third of the students.

Because this is a common mistake an instructor could specify how to detect this mistake and what corrective feedback to provide. The feedback for the first program is shown in Figure 2.

The top two elements in the example feedback give a name and a brief description of to the feedback. These elements are for management purposes as described later. The two primary

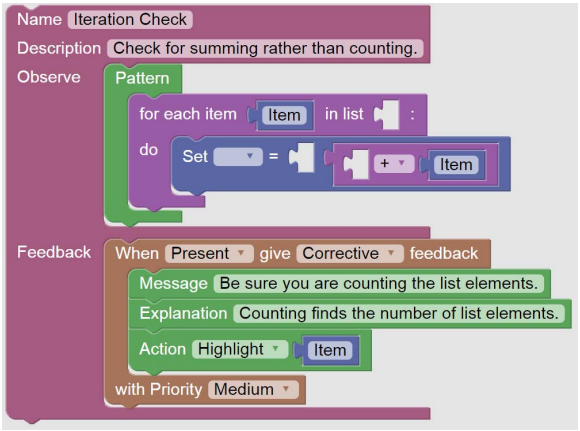

Figure 2 Example Feedback Specification

elements of the feedback is a pattern to be observed and the feedback to be given. The pattern is given as a block configuration where only the essential elements of the pattern are specified. In this case the feedback is given when the pattern is Present. It is also possible to provide feedback when the pattern is Absent (i.e., a missing required set of blocks). This feedback is categorized as Corrective (fixing an error). Other categories are Complementary (noting a correct aspect), and Completion (output evaluation). These categories anticipate that different user interface techniques might be used for each. The feedback consists of:

- Message: a short text description
- Explanation: a more elaborate explanation
- Action: a change made to the student's blocks (in this case to highlight the block in error).

Other feedback forms are possible. For example:

- Example: giving code to illustrate the feedback
- Resource: a link to a helpful reading, video, etc.
- Hint: a suggestion on what to use or how to proceed.

The priority setting anticipates multiple instances of feedback at a single time and allows a relative importance to be assigned.

More specifically, the pattern in Figure 2 matches all iterations that contain a set block for the addition of two items one of which must be the same as that used in the "for each" slot. The iteration may contain other blocks as well. There is no significance to the term "Item" other than being used in the two places shown. The translation of Figure 2 shown later in Figure 7 also helps to clarify the semantics. Our experience with authoring feedback for our students will guide the evolution of these semantics. Other environments or uses may choose different semantics as well.

In some cases the pattern cannot be expressed by the blocks alone. For example, another common mistake with iteration is filling in the iteration slots incorrectly as shown in Figure 3. In this case the student has reversed the placement of the list (the variable temperatures) and the iteration variable (temperature).

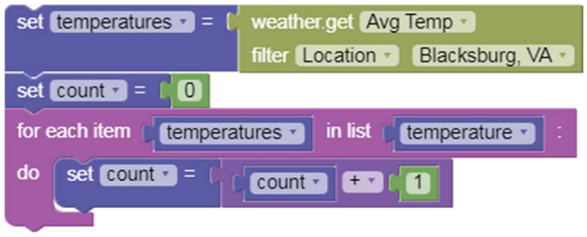

Figure 3 Example Error 2

The feedback specification for this example is shown in Figure 4. In this example the expression of the pattern requires not only the relevant blocks but also a condition that the pattern must satisfy. This condition, given by the "Where" element, is that the type of the misplaced block is a list. As discussed further in the Enforcement section, BlockPy has a limited ability to reason about types from static analysis. This example also illustrates another action, Eject, that ejects the misplaced block from its slot. This makes the feedback highly visible to the student.

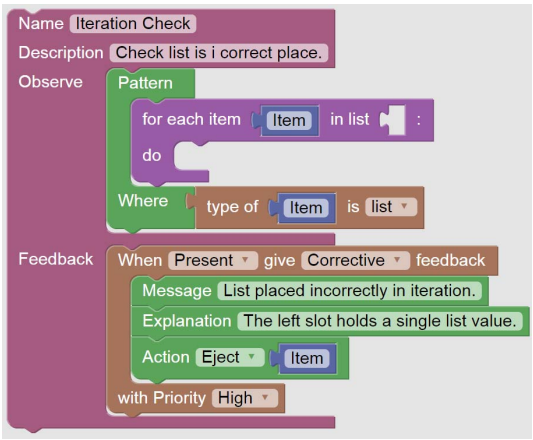

Figure 4 Specification of Feedback for Error 2

Traditional output checking can also be specified as shown in Figure 5. The feedback is based on whether the correct answer (the number 61) is contained in the observed output. This example also illustrates that feedback can be given both on the presence and absence of the same observation. Further, the feedbacks can be in different categories and have different priorities.

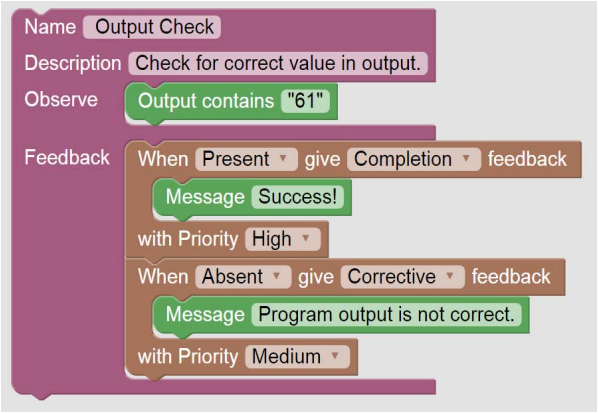

Figure 5 Output Checking

It is anticipated that there will be many feedback specifications for a given problem. It was seen above each specification has a unique Name and a Description. These elements aid the instructor in organizing and recalling what each specification is about. Specifications can be organized using a grouping block is shown in Figure 6.

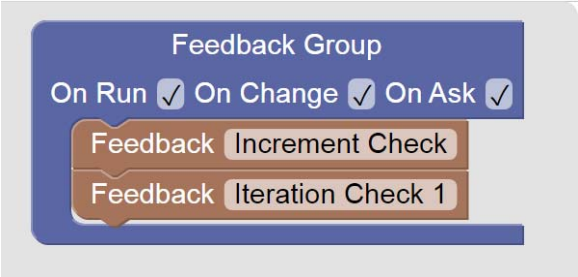

Figure 6 Organizing Specifications

In addition, the grouping block specifies when in time the instances of the group should be evaluated. Three distinct times are anticipated:

- On Run: the program is executed
- On Change: any change is made in the block program, and
- On Ask: the student asks for feedback.

These times are distinguished for two reasons. First, not all checks are reasonable at all times. For example, output checks should be performed most reasonably at On Run. Second, for performance reasons care must be taken not to perform so many On Change checks that the interface becomes sluggish. Greater response time latitude is possible On Ask.

III. Enforcing Specifications in BlockPy

The feasibility of the specification language is illustrated by showing how to enforce key parts of an example specification using a Python programmatic interface in BlockPy [10]. Figure 7 shows the translation of specification given in Figure 2.

The enforcement shown in Figure 7 inspects the AST of the student code. All iterations (indicated by a "For" node) are inspected and the iteration property (the variable holding the current list value) is extracted. All binary operations (indicated by a "BinOp" node) are inspected and feedback is generated if the operation accesses the iteration property. The code shown in Figure 7, for simplicity, does not show the full set of safety checks (e.g. whether there are any binary operations). The value of the specification approach can be seen by comparing the expression of the feedback in Figure 2 to the detailed programming in Figure 7.

The execution of the instructor-defined feedback checking is performed as part of a larger process. The complete process is performed when the student code is executed. The entire process consists of the following phases:

```
'''
Name: Iteration Check
Description: Checking for summing rather than counting
'''
def Iteration_Check():
    for_loops = ast.find_all("For")
    for loop in for_loops:
        item = loop.target
        assignments = loop.find_all("Assign")
        for assignment in assignments:
            binops = assignment.find_all("BinOp")
            if len(binops) == 1:
                binop0 = binops[0]
                if binop0.has(item) and binop0.op == "Add":
                    set_feedback("Corrective",
                                 priority="Medium",
                                 message="Be sure you are counting the list elements.",
                                 explanation="Counting finds the number of list elements.",
                                 highlight=item)
```

Figure 7 Enforcing a Specification

- (Student Code) Verification: check for empty code
- (Student Code) Parsing: generates an AST.
- (Student Code) Analysis: performs abstract interpretation to identify types/algorithm issues.
- (Student Code) Execution: runs the student code, collects runtime exceptions, trace, output.
- (Instructor Specifications) Enforce: runs the instructor code generated from the instructor's feedback specification.

At each phase, a report is generated, which can be used in the Instructor phase to access data from previous phases (AST, trace, variable behavior, issues, runtime exceptions, etc.). In Figure 7, for example, the first line of code accesses the AST. If a phase fails, then the report will indicate that with a "success" field.

After all reports have been generated the default action of the system is to present the feedback from each failed phase to the student. However, the instructor API can be used to prevent the feedback from a particular phase from being presented. Thus, the instructor feedback can give more specific or informative feedback than another phase. Currently, a "suppress" method is used but other mechanisms are possible (e.g., give specific feedback in such a way that it automatically suppresses all other kinds).

IV. Discussion

Other meta-information beyond the name and description could be added to the specifications. For example, each feedback specification could be annotated with a specific misconception. By capturing the occurrence of misconceptions over time a learning curve [12] could be generated automatically. The learning curves could inform both the learner, as an additional part of the feedback over multiple problems, and the instructor, providing an assessment of an individual student but also of the class overall.

Another possibility is to make the feedback mechanism adaptive [2] so that the feedback for a given student is moderated by the feedback that student received on earlier problems. Such adaptation is likely to make the feedback more targeted and specific. However, can such adaptability be added orthogonally to the feedback specification? Or is it necessary for rules for adaptability be included in the specification as well.

The feedback description language is silent on the user interface through which the feedback is presented to the student. This includes not only the look/feel details but also two other aspects. First, how intrusive or reserved is the feedback presentation? An intrusive presentation might require a window to be dismissed before proceeding while a reserved interface may simply indicate that feedback is available. Second, how much control does the student have over the feedback presentation. Can the student, for example, silence the feedback? Or see only the high priority feedback?

Finally, the specification language presented here and its enforcement in BlockPy are both evolving. Critique from the community will inform their evolution.

References

[1] Marieke Thurlings, Marjan Bermeulen, Theo Bastiaens, Sjef Stijnen, Understanding feedback: A learning theory perspective, Educational Research Review, Vol 9. p. 1-15, 2013.

[2] Le, Nguyen-Thinh, A Classification of Adaptive Feedback in Educational Systems for Programming, Systems, Vol. 4, No. 2, 2016.

[3] Valerie J. Shute, Focus on Formative Feedback, Review of Educational Research, vol. 78, no. 1, p. 153-189, 2003.

[4] David J. Nicol and Debra Macfarlane-Dick, Formative assessment and self-regulated learning: a model and seven principles of good feedback practice, Studies in Higher Education, vol. 31, no. 2, p. 199-218, 2006.

[5] Thomas W. Price, Yihuan Dong, and Dragan Lipovac. iSnap: Towards Intelligent Tutoring in Novice Programming Environments. SIGCSE '17, p. 483-488.

[6] Reppening, Alexander, Moving Beyond Syntax: Lessons from 20 Years of Blocks Programming in AgentSheets, Journal of Visual Languages and Sentient Systems, Vol 3, 2017, p. 68-91.

[7] Rivers, K. and Koedinger, K.R. (2015). Data-Driven Hint Generation in Vast Solution Spaces: A Self-Improving Python Programming Tutor. International Journal of Artificial Intelligence in Education, 1-28.

[8] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. Learning program embeddings to propagate feedback on student code. ICML'15, p 1093-1102.

[9] Jason Gorman, Sebastian Gsell, and Chris Mayfield. 2014. Learning relational algebra by snapping blocks. SIGCSE '14. ACM, New York, NY, USA, 73-78.

[10] Yasin N. Silva and Jaime Chon. 2015. DBSnap: Learning Database Queries by Snapping Blocks. SIGCSE '15, p.179-184.

[11] Austin Cory Bart, Javier Tibau, Eli Tilevich, Clifford A. Shaffer, Dennis Kafura, "BlockPy: An Open Access Data-Science Environment for Introductory Programmers", Computer vol. 50 no. 5, p. 18-26, 2017.

[12] Kelly Rivers, Erik Harpstead, and Ken Koedinger. 2016. Learning Curve Analysis for Programming: Which Concepts do Students Struggle With?. Proceedings ICER.