Jeuring, Johan; Keuning, Hieke; Marwan, Samiha; Bouvier, Dennis; Izu, Cruz; Kiesler, Natalie; Lehtinen, Teemu; Lohr, Dominic; Petersen, Andrew; Sarsa, Sami

Towards Giving Timely Formative Feedback and Hints to Novice Programmers

Published in:
ITiCSE-WGR '22: Proceedings of the 2022 Working Group Reports on Innovation and Technology in Computer Science Education

DOI:
10.1145/3571785.3574124

Published: 27/12/2022

Document Version
Peer-reviewed accepted author manuscript, also known as Final accepted manuscript or Post-print

Please cite the original version:
Jeuring, J., Keuning, H., Marwan, S., Bouvier, D., Izu, C., Kiesler, N., Lehtinen, T., Lohr, D., Petersen, A., & Sarsa, S. (2022). Towards Giving Timely Formative Feedback and Hints to Novice Programmers. In B. Becker, & K. Quille (Eds.), *ITiCSE-WGR '22: Proceedings of the 2022 Working Group Reports on Innovation and Technology in Computer Science Education* (pp. 95–115). ACM. https://doi.org/10.1145/3571785.3574124

Towards Giving Timely Formative Feedback and Hints to Novice Programmers

Johan Jeuring*
Utrecht University
The Netherlands
j.t.jeuring@uu.nl

Hieke Keuning*
Utrecht University
The Netherlands
h.w.keuning@uu.nl

Samiha Marwan*
University of Virginia
Raleigh, USA
samarwan@ncsu.edu

Dennis Bouvier†
Southern Illinois University
Edwardsville / US Air Force Academy
USA
djb@acm.org

Cruz Izu
The University of Adelaide
Australia
cruz.izu@adelaide.edu.au

Natalie Kiesler
DIPF Leibniz Institute for Research and Information in Education
Germany
kiesler@dipf.de

Teemu Lehtinen
Aalto University
Finland
teemu.t.lehtinen@aalto.fi

Dominic Lohr
Friedrich-Alexander-Universität
Germany
dominic.lohr@fau.de

Andrew Petersen
University of Toronto Mississauga
Canada
andrew.petersen@utoronto.ca

Sami Sarsa
Aalto University
Finland
sami.sarsa@aalto.fi

ABSTRACT

Every year, millions of students learn how to write programs. Learning activities for beginners almost always include programming tasks that require a student to write a program to solve a particular problem. When learning how to solve such a task, many students need feedback on their previous actions, and hints on how to proceed. For tasks such as programming, which are most often solved stepwise, the feedback should take the steps a student has taken towards implementing a solution into account, and the hints should help a student to complete or improve a possibly partial solution.

This paper investigates how previous research on feedback is translated to when and how to give feedback and hints on steps a student takes when solving a programming task. We have selected datasets consisting of sequences of steps students take when working on a programming problem, and annotated these datasets at those places at which experts would intervene, and how they would intervene. We have used these datasets to compare expert feedback and hints to feedback and hints given by learning environments for programming.

Although we have constructed extensive guidelines on when and how to give feedback, we observed plenty of disagreement between experts. We also found several differences between feedback given by experts and learning environments. Experts intervene at specific moments, while in learning environments students have to ask for feedback themselves. The contents of feedback is also different; experts often give (positive) feedback on subgoals, which is not supported by most environments.

CCS CONCEPTS

• **Social and professional topics** → **Computing education**.

KEYWORDS

Learning programming, feedback and hints, sequences of programming steps, learning environments, automated feedback

ACM Reference Format:
Johan Jeuring, Hieke Keuning, Samiha Marwan, Dennis Bouvier, Cruz Izu, Natalie Kiesler, Teemu Lehtinen, Dominic Lohr, Andrew Petersen, and Sami Sarsa. 2022. Towards Giving Timely Formative Feedback and Hints to Novice Programmers. In *2022 ITiCSE Working Group Reports (ITiCSE-WGR '22), July 8–13, 2022, Dublin, Ireland.* ACM, New York, NY, USA, 22 pages. https://doi.org/10.1145/3571785.3574124

1 INTRODUCTION

When learning how to program, students almost always work on tasks in which they have to develop a program to solve a given problem. Many students would benefit from feedback on their programming actions, and hints on how to proceed. Formative feedback

*co-leader

†The views expressed in this article, book, or presentation are those of the author and do not necessarily reflect the official policy or position of the United States Air Force Academy, the Air Force, the Department of Defense, or the U.S. Government.

ITiCSE-WGR '22, July 8–13, 2022, Dublin, Ireland

ACM ISBN 979-8-4007-0010-1/22/07…$15.00
https://doi.org/10.1145/3571785.3574124

and hints are essential aspects of learning [61]. Intelligent tutoring systems (ITSs) that provide feedback and hints on steps of students have shown positive results [43, 44, 69]. For tasks such as programming, which are most often solved step by step, the feedback should take the steps a student has taken towards implementing a solution into account, and the hints should help a student to complete or improve a possibly partial solution. But *when* in this process do students need feedback and hints, and *how* should it be given? These questions are addressed in research on feedback [26, 61], but that research is not specifically about learning to program step by step. We investigate how can we use this research to give feedback on, or a hint at, a particular step a student takes when solving a programming task. We want to collect datasets consisting of sequences of steps students take when working towards a solution to a programming problem, and annotate these datasets at those places at which we think an expert should intervene, and how the expert wants to intervene. We create several of such expert-annotated datasets. These datasets are useful for several purposes: they are concrete examples of when and how to give feedback while solving programming tasks, they can be used to evaluate the quality of hints and feedback given by environments for learning programming, they can inform the design of such environments, but they can also be used for educational research looking at student behaviour.

RQ1 How should we annotate datasets consisting of steps students take towards solving a programming task with information about when and how to give feedback and hints?

There are many learning environments that support beginners learning how to program, including ITSs [14], online environments[1], and educational games [23]. Some of these learning environments give feedback on potentially partial student solutions, and hints on how to proceed with a partial solution [22, 43, 44, 52, 56].

Learning environments for programming often differ in the way they interact with users. This raises the question how learning environments should interact with novices to support solving a beginner's programming problem step by step. One way would of course be to perform experiments with different environments, and to compare the learning outcomes. Setting up such experiments is not easy [4], and if we find a difference we would still like to know the cause(s) for that difference. One reason that might explain why some learning environments better support students is the timeliness and quality of their feedback and hints. To determine this, we can compare them against expert-annotated datasets [53].

RQ2 How does expert feedback relate to the feedback found in learning environments for programming?

This paper proposes a method for annotating datasets consisting of steps students take when working towards a solution to a programming problem. The annotations specify when and how to intervene at steps. We annotate several datasets using this method. Then we study learning environments for programming, and investigate how they support the steps that students take towards a solution. We also investigate to what extent the feedback delivered by learning environments complies with the expert hints and feedback.

[1] e.g. Codecademy, Datacamp, Khan Academy, Code.org.

This paper is organised as follows. Section 2 discusses some relevant background for our research. Section 3 introduces and discusses the central concept of this paper: steps. Section 4 describes the characteristics and selection of datasets we use in this paper. Section 5 introduces the coding we use in annotating the datasets with expert information about when, and how to intervene at particular steps in the datasets. This coding is then applied to the datasets described in Section 4. Section 6 evaluates learning environments for programming using the results from the previous sections. Section 7 presents the findings in all aspects of this work. Section 8 concludes and describes future work.

2 BACKGROUND AND RELATED WORK

2.1 Feedback

Experts (or learning environments) support students, amongst others, by giving feedback and hints [41, 68]. Feedback is backward looking and usually focuses on negative issues, errors, or missing parts, which need to be addressed. Sometimes, we can also provide positive feedback about a program, for example: "the base case is completed" when writing a recursive function. Positive feedback could reinforce learning when the student is unsure, by increasing their confidence and may also prevent weaker students from undoing useful edits that received positive feedback [16, 42, 43]. Feedback is often not enough for a student to make progress. Thus, further help in the form of hints will help a student that is stuck [41, 68]. A next-step hint suggests a next step to take [68].

Hattie and Timperley [26] define four levels at which feedback operates: task, process, regulation, and self. We are specifically focused on task and process level feedback. There are several types of feedback described in the literature. Examples of such types are: KR 'knowledge of result', which simply indicates whether the solution is (in)correct; KCR 'knowledge of the correct results', which shows the expected solution; and EF 'elaborated feedback', which may consist of various kinds of elaborated feedback messages or hints. By inspecting existing feedback classifications, Narciss [48] found that feedback types have multiple characteristics: functional, content-related, and formal. Narciss proposes a new *content-related* classification of feedback messages, aimed at interactive learning tasks. Keuning et al. [31] have adopted and extended this classification for the programming domain. Section 6.1.2 describes this classification in detail.

Hao et al. [25] found that feedback design has measurable impact on student performance. More specifically, they found a statistically significant difference in student performance on programming assignments between those students that received only pass/fail results of test cases, and those that received results of test cases with some explanation. More plainly, providing only pass/fail results from unit testing is inferior to providing contextual information regarding the unit tests.

Another result from Hao et al. is equally relevant to our work: students reported they like positive feedback provided by the system. Our experts include positive feedback, when appropriate. The authors also report that students reported they appreciate that the system was "unobtrusive".

Dong et al. enumerate six reasons to intervene (i.e., provide a hint or feedback) with students in the process of programming [18] (rephrased here):

- missing key components
- using unnecessary components
- misusing needed components
- logical error
- unknown intent
- hint needed

This study also confirmed the need for positive feedback. They observed that students with correct but incomplete programs undo their work only to arrive at the same point some time later. By this, the authors suggest a seventh reason to intervene (not on the above list): confirming the code has achieved a subgoal. We agree that feedback that (a) affirms the correctness of the code-state and/or (b) an intervention when destructive behaviour is observed can assist the student.

2.2 Previous reviews on programming learning environments

A number of systematic literature reviews from the past six years were used as a starting point for the identification of adequate learning environments and their features [3, 12, 14, 15, 31, 35, 51, 62, 63, 66]. We briefly introduce the main points of this (non-exhaustive) list of references in chronological order starting in 2016. In addition, we outline their role for the present work.

For example, Souza et al. [63] reviewed thirty assessment tools with regard to their main characteristics for students' programming assignments. Although their analysis yielded a classification of the tools, the tools, respectively their last publication, date back to the 1980 to early 2010s. In a 2017 review, Kim and Ko [35] analyzed 30 online coding tutorials and emphasized the immature state of many so-called tutorial tools due to the lack of personalized learning and explanations of basic concepts as part of the offered feedback. The tools that allow learners to write code and receive feedback are considered relevant for the present research.

Another systematic literature review of online tools to support novice programming with publications between 1998 and 2018 identifies 34 papers addressing the development phase of CS1 [62]. Of these, 14 papers describe more recent programming environments for novice learners, such as the App Inventor, Alice, BlueJ and its continuation Greenfoot, as well as Scratch, and Snap! However, only six of them are still available online. Crow et al. [14] analyzed 14 intelligent tutoring systems for programming education and their adaptive or intelligent feedback. Nonetheless, there is no in-depth analysis of feedback types. Moreover, only two of the systems (and their publications) date back to 2016 and 2017 which explains yet again why many of the systems could not be accessed. The same is true for the 101 tools with automated feedback for programming exercises analyzed by Keuning et al. [31]. The detailed overview describes the applied feedback types, as well as their adaptability and the techniques used to generate feedback. Thus, the methodology can be replicated with recently available and accessible tools.

The most recent reviews include Deeva et al. [15] with a detailed literature review of automated feedback technologies published between 2008 and 2019. Eleven online learning environments were identified, five of which with publications in the past five years [1, 13, 22, 30, 70]. Cavalcanti et al. [12] summarize 63 learning management systems in the context of computing; 19 of them with a focus on programming. Few environments with recent publications (in the last five years) were identified (e.g., MicK tutor [2], ArTEMiS [37], Online Judge [71]). Furthermore, Paiva, Leal and Figueira [51] investigate 30 tools and their automated assessment techniques proposed between 2010 and the first half of 2021. They analyze, for example, the tools' feedback elements, and note significant gaps in pedagogy and the quality of automated assessment compared to human feedback (see [38, 39, 58]). This is also an underlying assumption of the present work. The 2022 systematic literature mapping by Barbosa Rocha et al. [3] aims to understand the main approaches used for providing and evaluating feedback in learning environments for novice programmers. The 39 papers from 2016 to 2021 distinguish six different feedback approaches of relevant tools, but the report does not provide information on their availability and accessibility. Strickroth and Striewe [66] provide the most comprehensive and recent corpus of task-based grading and feedback systems for learning and teaching programming, although feedback types are not part of the overview.

To conclude, several systematic literature reviews have been evaluated, but while the number of tools/learning environments is rather large, there is no overview of recent, accessible systems. Some systems are very old and hard to access, and the classifications in the reviews vary. As we are specifically looking for environments in the programming context, a recent investigation of online environments and their feedback types is required.

2.3 Datasets

In 2015, an ITiCSE working group published a report titled: "Educational Data Mining and Learning Analytics in Programming" [27]. Their work surveyed the research literature on tools for data collection, and datasets of students working on programming problems. It lists 10 programming data collection tools and describes 10 datasets available at the time (2015). We looked at the two datasets that were listed as openly available: the Blackbox dataset [10] and the code.org dataset. Both datasets are unsuitable for our work, because in the Blackbox data we do not know the task on which a student is working, and the code.org dataset is not available anymore. The report concludes with a list of "Grand Challenges" for the community. Our work contributes in particular to the following challenge:

(1) to have researchers and practitioners commit to building and maintaining a multi-language, multi-institution, multi-nation learning process data and experiment result database

Price et al. [54] describe five datasets that are either not publicly available, or contain only final submissions of students.

3 STEPS AND SUBGOALS

Before we attempt any analysis we need to define the term "step" and to discuss how steps relate to the goal that they are intended to achieve. Step is a widely-used term that describes "an action, proceeding, or measure often occurring as one in a series" [64]. Thus, it is common to describe steps in a problem-solving process.

VanLehn found that feedback and hints are most effective when they are on the level of steps [69]. Though VanLehn's work is based

on studies of tutoring systems for other domains (e.g., physics, math); the granularity at which feedback is provided (i.e., step-based feedback versus only providing feedback at the conclusion) is very similar to the granularity at which feedback is provided to students working on programming problems.

We need to know the context to provide a more specific meaning to the term "step." In this paper we analyse activity (event) logs from environments that capture actions students take while working on solving programming tasks. As our work begins with the analysis of programming activity logs, our definition is based on the programming activities as recorded in the dataset under study.

3.1 Step

We define a "step" as a programming development action taken by the student. To be more precise, a **step** is one of

- a **code-edit** as recorded by the learning environment, which may range from a simple keystroke to adding a full line or pasting a code block
- a **user-action** that triggers an environment action, such as *compile*, *run*, *get hint*, *submit*, or *show solution*

The datasets we study capture sequences of steps students take when solving a programming task, which in general results in a sequence of program states. The difference between one code-state and the next is the addition or deletion of code: a keystroke, paste event, or cut event. A code-state may be incomplete and/or syntactically incorrect, so that it cannot be compiled or interpreted.

Although user-actions do not change the program state, they potentially change the student's perception of it and provide clues about their state of mind or intentions. The feedback we give to a student should take this into account. Consequently, user-actions are essential for determining when and how to interact with the programmer, and they sometimes provide a convenient time to provide a hint with less danger of interrupting a stream of thought. For example, choosing to run the program indicates the student is looking for feedback; perhaps because they believe they have completed a chunk, or simply because they wish to check that their syntax is legal. Not all environments provide all the user-actions we listed above, nor is our list exhaustive. Other environment-events such as syntax error highlighting or given feedback should also be included as part of the data.

Though in this paper we analyze data from students using text-based programming languages, the concept of a "step" also applies in block-based languages. Mouse-events are the dominate means for making code-edits, but it is the code-edit itself that defines the step a student makes.

Temporality. In many logs of student steps (e.g., the datasets used in this work), each event is annotated with the time the event occurred. In an interactive session, time is considered part of the event-stream. The utility of timestamps is discussed in Section 5.1.

Step size. The granularity of steps varies. In general, it is possible to convert from a very fine-grained step size to a more course granularity but not vice versa, so we prefer finer-grained collection. Using data with keystroke-level granularity ensures the data are compatible with other code analysis tools (e.g., CodeProcess Charts [60]), which can be useful in the investigation of students' programming processes.

The appropriate level of step-granularity may depend upon several factors, such as the student's ability. For example, a student starting a CS1 course may find writing an assignment to have multiple steps, but after a few weeks practice may view the same assignment as a single step [29].

3.2 Subgoals

A program is a collection of statements, or steps, that collectively achieve the goal of solving the given task. For all but the smallest tasks, the goal can be meaningfully divided into subgoals.

> "A subgoal represents the purpose of a set of steps" [11].

Decomposition, or breaking down a task into smaller subproblems that are solved individually and then assembled together, is at the core of most problem solving strategies. Each subproblem has a clearly identified purpose or subgoal. When writing code, subgoals support a top-down design by decomposing the algorithm into smaller steps that are then refined and implemented. Each subgoal is implemented using a plan [55]. Subgoals and their associated plans facilitate transfer by helping learners to identify when a new problem or task shares one or more subgoals with the already-learned procedure [11, 28].

When learning from worked examples, subgoals help students to better recognize the fundamental components of the problem solving process. Subgoal learning can be passive (subgoals are provided as comments in the code) or active (by asking students to write or generate those comments) by interpreting the purpose of the given block [47]. In short, subgoals provide organisation and help constrain the piece(s) of the procedure on which to focus.

A learning environment could use subgoals to recognise the intention of a subset of steps and provide targeted feedback for each subgoal. For example, when solving a recursive problem, we usually have two subgoals (1) complete the base case, (2) complete the recursive case. Note each subgoal is achieved by one or more steps. For example "complete the base case" can include the steps (1a) write a return statement, and (1b) provide the correct return value for the base case.

In this study, we use subgoals as a lens to frame student progress toward a solution, to associate particular steps with a purpose, and to generate suitable hints. This approach provides a framework for providing feedback. It supports the generation of more specific feedback, by identifying completed, missing, or incorrect subgoals, and it facilitates our ability to scale support from small programs with one or two subgoals to larger programs with multiple subgoals.

4 DATASETS

A main goal of this work is to collect, analyze, and annotate a selection of datasets consisting of individual student programming activity.

We first establish a list of desirable/common characteristics for the datasets we want to analyze. Then, we present the datasets we have selected. We publish the datasets we have analysed (either in full or their annotations only) for the community. Finally, we describe a tool that can be used to study the datasets.

4.1 Dataset characteristics

Our expert-annotated datasets need to have the following characteristics.

Rights. The rights of the datasets should allow for analysing the dataset, adding annotations to the dataset, and publishing the annotated dataset under an appropriate license. In some cases we are not allowed to publish the original dataset, in which case we publish the annotation separately.

Tasks. Given our focus on providing hints to students, we choose datasets that include tasks aimed at novice programmers. To give meaningful hints and feedback, we need to know the specific programming problem the student is working on. Thus, for each sequence we need to know the corresponding programming problem and its description. We include solutions when available, and develop correct solutions for exercises where they are not provided.

The programming problems should not be too complicated. We expect solutions to the problems to consist of a small number of lines of code, typically in the order of 10 lines. We have looked at some datasets that contain data from students solving rather challenging tasks, and noticed that it is hard to annotate steps when students are deviating far from a desired solution. We have seen sequences of hundreds of steps that lead nowhere, and for which it would be very hard to give feedback or hints. Focusing on a single subgoal at a time could be a way to deal with this complexity, but this is left for future work.

Sequences. We aim to analyse student steps that lead to a solution. So we are not looking at datasets containing only final student submissions to programming tasks, or sequences of such submissions. The analysis of final program submissions is the subject of other research [19, 63].

Programming languages. A dataset will contain programs in a particular programming language, but the programming language might be different for different datasets. We have found datasets using amongst others Python, Java, and Dart.

An important reason to collect datasets is to use them for evaluating learning environments. So we prefer to collect datasets for programming languages for which we also have access to learning environments. However, this is not required for including a dataset. If we have a learning environment for a particular programming language for which we have no dataset, we could choose to compare feedback and hints in the learning environment with a dataset with data from a *similar* programming language.

Granularity of steps. Code-edits can be recorded (or observed) at different levels of granularity. The ITiCSE'15 WG identified six different levels of granularity at which data were collected by systems at that time (2015) [27]. These are:

- keystroke
- line
- file-save
- compile
- execute
- submit

When studying and annotating the datasets we selected, we identified a seventh level between keystroke and line: token. That is, the data are collected each time a new token is detected. Token detection varies by programming language; often a token is detected when whitespace or a delimiter is used.

As one of our goals is to guide the development of systems that intervene at the best moment, waiting until a line is 'completed' may be too late. As a result, we can use datasets with granularity level ranging from keystroke to token edits or even more coarse grained. Since our expert annotators do not feel the need to annotate steps at the keystroke level, we pre-processed keystroke-level data to token-level data. First, code states i.e. the source code in the editor at a given instant must be reconstructed from the individual recorded keystrokes. Second, the code states that include incomplete tokens can be filtered so experts only investigate states where complete program tokens are inserted or removed.

System feedback. If the environment in which the data is collected gives feedback to students, this feedback is either included in the dataset, or can easily be reconstructed. Note that if an environment gives feedback beyond basic compiler errors and test results, the resulting dataset is harder to use to evaluate other learning environments.

4.2 Selected datasets

We searched for publicly available datasets in the DataShop repository (a repository for learning interaction data) [36], by studying datasets collected in related work [27, 54], and by approaching peers conducting work in this area.

Our search resulted in a list of five datasets that satisfy the characteristics described above. Table 1 shows the datasets and their key features. From now on, we refer to each dataset by their identifier in `teletype`.

Below we describe the selected datasets. They are collected using different learning environments and the environments that were relevant to this study have been included in our evaluation. Those environments are described in more detail in Section 6.

`CodingBat`. `CodingBat` has been obtained by watching videos of students developing two programs, returning the factorial or the fibonacci number of an input number respectively, in CodingBat, a programming environment for Java. The videos have been used by the data owner to create a dataset consisting of steps at a slightly higher level than tokens, called token$^+$: sometimes one or more expressions are entered in a single step. In addition, *compile*, *get hint*, and *show solution* actions are recorded. The dataset is freely available for scientific research, but we are not allowed to republish a fully annotated version of the entire dataset.

`FITech`. `Fitech` contains keystroke-level code edit actions as well as *run*, *submit*, and *request help* actions captured from online IDE embedded in a course platform for distance learning. The data is from a course that covers the principles of programming using Dart language and contains 64 different small programming tasks. The data we publish includes two simple tasks, reading input until it matches the required and returning a conditional value depending on a function parameter.

Table 1: Datasets considered for annotation.

Identifier	Name and Source	License	Language	Granularity	Sequences	Annotated[1]	Mean steps [2]
CodingBat	Kiesler [32, 33]	©DZHW	Java	Token$^+$	16	16	20.1±11.8
FITech	FITech 101 Introduction to Programming	CC BY-SA 4.0	Dart	Keystroke	25	25	144.1±120.7
Bielefeld	Python Programming Dataset [50]	GPL	Python	Pause	75	0	11.2±8.6
Utah	2021 CS1 Keystroke Data [20]	CC0	Python	Keystroke	554	0	796.1±1200.4
iSnap	iSnap - Fall 2017 [fixed] [52]	DataShop	Snap	Block-edit	285	0	565.8±472.2

[1]Total number step sequences (one sequence per task for each student)
[2]Mean and standard deviation of steps per sequence (after pre-processing)

Bielefeld. Bielefeld takes a snapshot of a student program after two seconds of inactivity. We call this level of granularity 'pause-level'. The students had five very structured tasks. The first task was to program a given mathematical function in Python. A tutor was present to give guidance and students could only proceed to the next task once the previous one was completed. The students proceeded to implement function's gradient and finally the gradient descent algorithm.

Utah. Utah records keystroke-level code edits and *run* actions from programming environment used in CS1 university course. Student work is available on 8 different weekly assignments for programming in Python. All of the tasks involve a relatively abstract problem, starting from testing which numbers fulfill 6 criteria of a 'Fluky' number and optimizing the program. The second task requires simulation of described events to extract estimates for different probabilities.

Unfortunately, we were unable to reconstruct code states from the keystroke data for 750 sequences i.e. the characters that were supposedly removed did not exist in the code state at that point. We have contacted the author of the dataset and some issues may be fixed by the data owner in updated versions of the Utah dataset.

iSnap. iSnap contains each step of creating a program in a block-based programming language Snap! as well as *get hint* actions. The data is collected from the iSnap learning environment. The tasks involve drawing in 'Turtle' style and a game to guess a random number. As a property of a block-based language the sequences do not have syntax errors.

The dataset stores the program state in a textual format. The described object hierarchy of the program is missing values and we could not find feasible methods to reconstruct visual program blocks which makes analysing steps in the iSnap dataset rather tedious.

The only two datasets that include system feedback beyond compiler errors and test results are FITech and iSnap. FITech gives feedback on code quality, which does not affect the kind of feedback we aim to give. The iSnap dataset includes *get hint* actions, which makes it more difficult to use this dataset to evaluate other learning environments.

4.3 Tools

We publish the annotated data sets in the well defined ProgSnap2-format [54], which we expand with custom columns for the annotation. The specification defines that a dataset has a single main table for all the events. Such common index is useful for machines reading the data. However, human inspection requires viewing one student and their sequence of steps for one assignment at a time. For the datasets we analysed, we split the main table into more manageable tables that each include a separate sequence.

We developed a data browsing tool that supports navigating sequences of steps by assignment and student. Furthermore, the tool allows hiding unnecessary columns and can highlight changes in code state and elapsed time. Our tool and our annotations of the datasets are available online[2].

5 INTERVENTIONS

This section describes how we annotate steps in the datasets we collected in the previous section. In particular, we provide information about when to give feedback and hints to a student, and what kind of feedback and hints should be given to the student at that point. We call this information *interventions*, which are responses to a step or a sequence of steps. We draw up guidelines both for when to intervene and how to intervene. The guidelines are created by looking at some of the data, and by asking multiple experts to specify when they would intervene, and discussing the results. After establishing the guidelines, we use them to annotate several (parts of) the datasets we selected. The annotation is done by several experts, in several rounds, as we describe in Sections 5.1.1.

5.1 When to intervene

We draw up guidelines for intervening when a student is working towards a solution for a programming task. The guidelines should mimic the way a human expert, being both a *programming expert* as well as a *pedagogical expert*, would intervene in a one-on-one tutoring session. We aim to describe the guidelines in such a way that facilitates its automatic application by a learning environment.

Robertson et al. [57] advise learning environment's designers to resist the temptation to use immediate interruptions to "help" users find bugs. This advice aligns with the practice from experienced

[2]https://github.com/Programming-Steps-Working-Group-2022/public-datasets

Table 2: Guidelines that identify when and how to intervene.

Event	Example	Intervention Point (when)	Intervention Action (How)
Compiler error	Syntax or type error	If a student is not using the compiler often, or after the second compilation where the error goes unaddressed.	First, suggest that they compile if they have not done so. Second, offer an explanation.
Semantic error	Syntax is correct but wrong semantics, e.g. "=" instead of "=="	When a student moves to the next line, as these errors are hard to spot/debug.	Highlight the location of the error and offer an explanation.
Logical error		Once a student executes/tests the code or within 5 minutes if they choose not to run the code. If the error would lead to any further code being incorrect, intervene immediately.	Indicate the test case(s) that fail due to the error and provide a hint on how to fix it.
Deviation from specification	Changing required function signature	When a student leaves the line.	Provide clear statements from the assignment description as a reminder of the assignment specifications.
Trial and Error behaviour	Iterating through conditional operands	Once it becomes clear the edits are guessing – not experimentation.	Ask a student a question (e.g. an MCQ) about the purpose of the line. If they respond with a correct answer, provide a hint. Otherwise, suggest a (sub)goal to complete.
Hint or Feedback request	Pressing a "Hint" or "Show solution" button	Immediately when a student requests assistance.	A hint depends on the time a student requests it. If a student has a semantic/syntax error and asks for a hint, then offer a clear hint on how to fix the error. If a student has a logical error or is stuck in a specific subgoal, then offer a clear hint on how the subgoal/objective can be reached.
Subgoal completion	Correct base case(s) for recursive function	When a student completes all steps of a subgoal.	Provide positive feedback specific to the accomplishment.

instructors. Thus, we followed such advice by setting two general rules while reviewing and labelling student data logs. First, we decided not to interrupt students' train of thought when making small errors or writing incomplete lines. Second, we should give students the possibility (e.g. extra time to complete another step) to fix minor errors, instead of immediately reporting the errors.

5.1.1 Method. To identify intervention points, we performed a 2-step process. In the first step, three experts[3] reviewed a small set of logs, discussed opportunities to intervene and drew up a draft guideline for situations that call for an intervention.

In the second step, we used the resulting draft guidelines to annotate a collection of sequences from two datasets: `CodingBat` and `FITech`. This step involved multiple rounds of refinement, validation, and discussions between one to four experts in each round (described in detail below). This step resulted in a final list of "When to Intervene" situations, as shown in Table 2. Table 3 shows the assignment of experts in the data labelling rounds (all experts are authors of this paper). In each round, the experts annotated the sequences of a small number of students (5 - 15), where each has ~ 20 - 400 logs of data (i.e. code edits).

Table 3: Allocation of (E)xperts to data labelling (R)ounds.

Experts	CodingBat dataset			FITech dataset		
	R1	R2	R3	R4	R5	R6
E1	x					
E2	x					
E3	x	x	x			
E4	x	x		x		x
E5				x	x	x
E6					x	x

In section 3.2, we explained how we use subgoals to identify progress towards the main goal and to provide specific feedback. As subgoals were not documented in the existing task descriptions, the expert instructors constructed subgoals for each task before starting the rounds of data annotation.

Round 1 (`CodingBat`, Factorial exercise, 120 log entries, 6 students) annotates the sequences of novices (on average 20 rows of logged data per student) while implementing a recursive *Factorial* function (shown in Figure 4 in appendix

[3] All experts who reviewed, or labelled data in this work are from the authors' group. We chose to name them as "experts" since all of them have a multiple-year experience in CS education research and in teaching programming classes.

A). As this was our first attempt to use the guidelines, we allocated four experts to independently evaluate the logs. The experts then met to discuss the guidelines and resolve any annotation conflicts. We computed the inter-rater reliability by running Fleiss' Kappa test, which is an adaptation of Cohen's kappa for 3 or more raters. The Fleiss' Kappa result was 0.563, showing a moderate agreement between the four experts.

Round 2 (`CodingBat`, Fibonnaci exercise, 112 log entries, 5 students) contains log entries from novices implementing a recursive *Fibonacci* function. (shown in Figure 3 in appendix A). Two of the previous experts labelled this set. The Cohen's Kappa test result is 0.834, showing an almost perfect agreement.

Round 3 (`CodingBat`, Fibonacci exercise, 88 log entries, 5 students). Given the perfect agreement in the previous round, in this round only one expert labelled 88 additional entries from 5 students solving the *Factorial task*. It is worth to note that when the expert found a concern or a new situation, they discussed it with the experts from round 2 until conflicts were resolved.

Round 4 (`FITech`, Temperature exercise, 1k entries, 5 students). Using the `FITech` dataset, two experts reviewed data entries of 5 students for the Temperature exercise (shown in Figure 2 in appendix A). We computed the inter-rater reliability, where we found the Cohen's Kappa test result is 0.71.

Round 5 (`FITech`, Password exercise, ~ 800 entries, 5 students). Two experts reviewed ~ 160 code edit on average per student in the Password exercise (shown in Figure 1 in appendix A). The Cohen's Kappa test result is 0.895, showing a substantial agreement.

Round 6 (`FITech`, Temperature exercise, ~ 2k entries, 15 students). Finally, three experts reviewed ~ 2k data logs from 15 additional students solving the Temperature exercise. In this round, the experts divided the task by students so that each entry was reviewed by two of the three experts. While the previous round shows a substantial agreement, in this round the experts decided to continue labelling the data together since solutions of these exercises are longer than that of the `CodingBat` exercises, showing a larger space of students' solutions. Overall, in the `FITech` dataset, the experts reviewed ~200 entries per student, making a total of ~4k data entries.

Early rounds were testing the clarity and validity of the guidelines. The final rounds were performed in order to provide a substantial labelled dataset to the community. In Section 7.2, we discuss reasons behind experts' varying opinions while annotating the data, and how these variations can guide researchers and tool designers to develop more effective feedback in learning environments.

5.1.2 Results. In this Section, we answer the first part of RQ1: *"How should we annotate datasets consisting of steps students take towards solving a programming task with information about **when** to give feedback and hints"*. The answer to this question consists of the guidelines presented in Table 2, and the annotated datasets available on GitHub.

The labelling process confirms seven situations in which we think an intervention is desirable. Six events are remedial: three address errors introduced in the code (compiler error, semantic error and logical error) and three address high-level design issues (deviation from assignment, trial-and-error, hint request). The last event, subgoal completion, provides positive feedback.

If a student develops a solution by adding correct code line by line and self-correcting minor issues, then they will only receive positive feedback.

In the first three events (i.e. compiler error, semantic error, and logical error), we delay intervention to provide a student an opportunity to fix the errors without. For example, we only intervene in case a compiler error is repeated.

However, there are two specific events in which we intervene immediately: (1) when a *semantic error* is entered, and (2) when code edits clearly *deviate* from the task. Two examples of the latter behaviour are changing the function header (e.g. adding extra function parameters) and hard coding the answer.

While labelling the logs, we observed two behaviours at which we think we should not intervene: exploration (or productive tinkering [17]) and print debugging.

While exploring, it may appear that a student does not have a particular goal in mind; this can be an intentional behaviour to understand how a new construct or statement works or to try to build code by tinkering. Note that if tinkering does not eventually result in progress towards a goal, it will trigger an intervention.

When print debugging, a student adds unnecessary print statements to the code, followed by performing a user-action. Print debugging differs from tinkering in that a student does not try to construct code by trial and error, but investigates what the current code is doing. Again, no intervention is needed unless the print statements are not commented out when the student completes the subgoal.

5.1.3 Challenges and Reflections. Experts sometimes labelled the data differently. In this subsection, we discuss sources for these labelling conflicts, including examples. In addition, we reflect on useful log information that helped to define when to intervene.

Labelling conflicts. We describe some scenarios in which we found labelling conflicts, and we discuss how we addressed or resolved them.

In the first scenario experts agree on when they want to intervene (for example, usage of print command instead of return), but they choose to intervene at different code edits, either directly after the mistake was made (see Figure 1.(a)) or after it was clear the student is leaving the mistake by moving to another part of the solution (as in Figure 1.(b)). This was a frequent conflict caused by different pedagogical approaches: (a) an early intervention prevents a student from writing unnecessary code and spending extra time on an assignment, which may lead to student confusion and frustration, versus (b) a delayed intervention gives a student a chance to struggle productively [17, 18], which may improve student learning. We will discuss how to offer some flexibility in where to intervene in 7.2.

In the second scenario, we found some initial conflicts that were related to giving positive feedback for completed subgoals. For example, Figure 2 shows an example of a student working on the base case (or stopping condition) subgoal in the Fibonacci exercise. In this case, one expert felt giving positive feedback on subgoal completion was not required because it felt redundant, while other

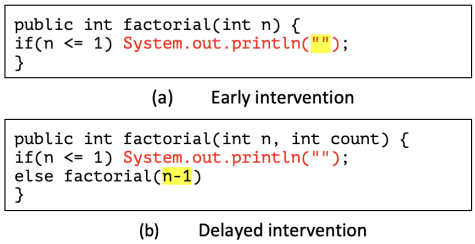


Figure 1: Example of a when-to-intervene conflict.

Code (last edit highlighted)	Intervention event

public int fibonacci(int n) {
if(n==0) return 0;
if(n==1) return 1;
 	
}
	Subgoal completion (implement base cases)
	Outcome – provide positive feedback

Figure 2: Example of a moment to provide positive feedback.

experts provided positive feedback only on completing the task. This is a minor conflict, as the student has already completed the task and hence needs no further help. However, we added in the final guideline to provide positive feedback when completing subgoals, which is suggested by prior work that providing positive feedback on subgoals can improve novices engagement and performance in programming tasks [42, 43].

A third conflict scenario is shown in Figure 3. In this case a student has made a change in the code to avoid a stack overflow, but the resulting code is incorrect. One expert wanted to intervene because "the student just tries out stuff and wants feedback, but does not know how to set the parameters of the recursive calls", while another expert interpreted the code edit as a sign of progress, without requiring intervention.

Code (last edit highlighted)	Action	Feedback received

public int fibonacci(int n) {
if(n==0) return 0;
if(n==1) return 1;
return n +
fibonacci(n+1);
}
	Go button	Expected Run
		fibonacci(0) → 0 0 OK
		fibonacci(1) → 1 1 OK
		fibonacci(2) → 1 StackOverflowError (line:4) X
		fibonacci(3) → 2 StackOverflowError (line:4) X
		fibonacci(4) → 3 StackOverflowError (line:4) X
		fibonacci(5) → 5 StackOverflowError (line:4) X
		fibonacci(6) → 8 StackOverflowError (line:4) X
		fibonacci(7) → 13 StackOverflowError (line:4) X
		other tests X

public int fibonacci(int n) {
if(n==0) return 0;
if(n==1) return 1;
return n + fibonacci(n-1);
}
	Go button	Expected Run
		fibonacci(0) → 0 0 OK
		fibonacci(1) → 1 1 OK
		fibonacci(2) → 1 3 X
		fibonacci(3) → 2 6 X
		fibonacci(4) → 3 10 X
		fibonacci(5) → 5 15 X
		fibonacci(6) → 8 21 X
		fibonacci(7) → 13 28 X
		other tests X

Figure 3: A student tests code, which results in a stack overflow, and then removes the incorrect call.

The fourth and last conflict scenario was when a student asked for a hint. One expert coded it as "no intervention" - as the system already has a hint mechanism in place and would provide a hint. However, other experts considered giving a hint as part of the intervention. Thus, we included hint guidelines to reflect this agreement.

Reflections on the components of the data. The core contents of the data are the edit steps students take towards solving a programming task. But besides these steps, we also found that timestamps and user-actions with their associated feedback results given by the system in which the data was collected are very informative when deciding about when to intervene.

Timing information (timestamps or other time references) in student log items provides experts a better picture of the student's current problem solving status. For example, two quick edits could be a sign of tinkering while the same edits made in a longer time span could be indicative of struggle.

Furthermore, some datasets recorded user-actions but did not include the feedback given to the student by the compiler or interpreter. Limited or no feedback information impacts "when to intervene" in two ways: (1) when a student performs a user-action that leads to an error message of the system, but the log data does not include this error message, it is hard to decide for an expert whether the error message is unclear and an intervention is needed; (2) when code compiles and runs, a student may receive feedback from the system on failing test cases; the annotator should not provide redundant information, and may use this feedback to determine if the next step is reasonable.

Concluding, although it is possible to intervene without either timestamps or user-actions, their inclusion facilitates a more informed intervention.

5.2 How to intervene

Intervention messages need to be specific, succinct, and phrased in terms familiar and relevant to the student population [21, 45]. However, since we do not know the students from whom the data was collected, we cannot provide specific messages; instead, we give well-defined suggestions for how to intervene, and leave the construction of specific messages (for example, the explanation for a semantic error) to those implementing the guidelines in a real situation. We describe the method to develop and refine the "how to intervene" guidelines, and discuss challenges found in this process.

5.2.1 Method. The methodology we use to develop guidelines for how to intervene is similar to the one presented in section 5.1.1, following the same rounds. Since "how to intervene" depends on the moments where experts decide to intervene, we decided to construct guidelines for "how to intervene" for each situation in the "when to intervene" guidelines. The same experts who labeled each round in the "when to intervene" methods, also labeled the data with the "how to intervene" guidelines. Note that this methodology is similar to the two-step process that we conducted above (Section 5.1.1).

In Round 1, experts tagged data from 6 students solving the Factorial programming assignment from the `CodingBat` dataset. The experts labeled 24 entries out of 120 in the 6 files, where these entries represent the moments the expert decided to intervene. For this round, the result of the inter-rater reliability test is 0.545, showing a moderate agreement. The experts then discussed the resulting annotations to come to consensus on how they wished to intervene.

This level of agreement is the result of variations in the strategies employed by the experts. For example, one expert might prefer to provide a hint where another prefers to instruct a student on the underlying problem. In essence, the experts varied in the amount and directness of the assistance they wished to provide. In addition, we identified three other factors that affected our experts' decisions on "how to intervene": (1) the type of messages provided by the compiler of the programming environment, (2) characteristics of the target audience, and (3) the resources available to the students from the problem itself. For example, one expert knew that the programming environment used for a particular dataset provided feedback on syntax errors – but did not provide useful feedback at compilation. Their feedback focused on supplementing information available in the environment.

Because of the impact of these classroom-related factors and to ensure a higher quality of data labeling, the experts decided to continue labeling the remaining data together. Whenever a conflict occurred, the experts discussed it until they reached consensus.

5.2.2 Results. In this section, we answer the second part of RQ1: *"How should we annotate datasets consisting of steps students take towards solving a programming task with information about **how** to give feedback and hints"*. The answer to this question consists of the guidelines presented in the last column of Table 2, and the annotated datasets available on GitHub.

As mentioned in the previous section, different experts interpreted the intentions and problems of students as exhibited in the student logs in different ways, and had different pedagogical practices. As a consequence, there was more variation in how experts annotate *how* to intervene than in *when* to intervene.

The amount of information provided or withheld in an intervention message is a well-known *assistance dilemma* [46]. For events that require immediate intervention, the response should provide detailed information: on a semantic error we should provide the location of the error and an explanation about how to fix it. For deviation from specification we should remind a student of the expected outcomes. For other events, experts differ in the granularity of their advice. Some experts favor high-level (i.e. less granular) suggestions about the task's subgoals, while others prefer more granular advice, such as a specific hint on the next step towards a subgoal. Our guidelines are flexible in this regard.

Furthermore, when looking at how to proceed at an intervention point, all experts agree on the importance of knowing the student's intent (e.g., asking whether we could provide a specific next-step hint without knowing exactly what the student was intending to write). As intent is sometimes implicit, a human tutor will often open a dialog with the student by asking a question about their intention. We suggest to follow this approach when intention is implicit or hard-to-guess. For example at a trial-and-error event, as shown in Table 2, the experts decided to use multiple choice questions (MCQs) to ask students about their intent, before giving a hint.

In situations where a student may receive feedback from the learning environment (such as compiler errors, or semantic errors), the experts agreed on providing intervention messages that encourage the student to compile or to pay attention to the information in the editor. However, in cases where a student did so but their error persisted, the experts agreed on the necessity of providing an alternate explanation or more informative support. This can be similar to the enhanced compiler messages suggested by Becker et al. [6, 7].

Note that the only intervention event triggered by students is *Hint or Feedback* request. If such a request is adequately dealt with by the environment in which the data is collected, then we do not need to do anything. Otherwise, there presumably is a problem in the student solution, which will be an instance of one of the other intervention events.

The response for positive feedback is simple - report which subgoal is completed. When implemented, the message may add words of encouragement or may indicate how many subgoals are left, as suggested by prior work [21, 42].

5.2.3 Challenges. We found two sources of labelling conflicts in the "how to intervene" phase. First, experts sometimes disagree on how to interpret particular edit steps or actions, and as a result, disagree on what response is appropriate at a particular location. For example, one student rapidly deleted and retyped several reserved words in the language. In one interpretation the student is exploring different options, in which case the student might benefit from a reminder of which subgoal might be attempted first. Another interpretation is that the student is unsure how to implement a particular subgoal, in which case an expert could either let them continue to further explore or nudge them towards a solution.

The second labelling conflict is found in the expert responses to *repeated* mistakes. The initial guidelines only required to use recent steps, as this simplifies the process by not keeping previous feedback history. This means that when a student makes the same mistake twice, the response will be exactly the same the second time. We propose to deal with repeated mistakes in two possible ways: (1) ignore the same mistake and do not intervene or (2) intervene by providing additional information. For example, if an expert decides to intervene the second time the student makes the same mistake, then consider providing an example or more specific guidance instead of just the (apparently ineffective) feedback message. A similar scenario occurs when a student repeatedly ask for hints. In this case, we may change the response into a dialog, such as in the trial-and-error event, to understand the student's intent, and provide them with the kind of feedback or hint they expect.

6 EVALUATING LEARNING ENVIRONMENTS

In this final part of the study, we focus on programming learning environments, investigating how they support the steps that students take towards solving a programming problem. We also investigate to what extent expert hints and feedback align with the feedback delivered by learning environments.

6.1 Method

We have selected a number of programming learning environments, categorized the feedback they provide, and attempted to replay sessions from the student datasets to learn how they would be supported. This section elaborates on the selection process (Section 6.1.1) and the coding of the learning environments (Section 6.1.2).

Table 4: Classification of feedback for programming.

Label	Description
	Simple feedback
KP	Knowledge of performance
KR	Knowledge of result/response
KCR	Knowledge of the correct results
	Elaborated feedback
KTC	Knowledge about task constraints
TR	Hints on task requirements
TPR	Hints on task-processing rules
KC	Knowledge about concepts
EXP	Explanations on subject matter
EXA	Examples illustrating concepts
KM	Knowledge about mistakes (basic or detailed)
TF	Test failures
CE	Compiler errors
SE	Solution errors
SI	Style issues
PI	Performance issues
KH	Knowledge about how to proceed
EC	Bug-related hints for error correction
TPS	Task-processing steps
IM	Improvement hints
KMC	Knowledge about meta-cognition

6.1.1 Selection of programming learning environments. We investigated a number of popular and well-known digital programming learning environments (also referred to as 'systems' or 'tools') found online and in the literature. We identified these systems by a literature study of programming tool reviews (see Section 2.2), by searching for online systems on Google, and by collecting a list of systems from ourselves and colleagues. We do not aim for a complete list, but strive to give a broad overview of current systems. We set the following inclusion criteria that a learning environment should adhere to:

General The environment should be specific for learning programming. It should support a textual programming language, because the programs in our currently annotated datasets are also textual. The interface of the system should be available in a language spoken by the researchers.

Features The system should support the following features:
- The student can enter pieces of code, such as a snippet, method, or class.
- The system can analyze and provide feedback on the code the student has written. For example, there may be a button to ask for feedback, or the system may deliver feedback at certain times.

Availability The system should be accessible for downloading or running it online.

Relevancy We investigate only systems we can currently access online, or systems from the last 10 years based on the latest publication of a paper and/or the software.

6.1.2 Data coding. For each system, we collected information on the supported exercises, the type of feedback, and the timing of the feedback it provides. Below we explain this in more detail.

Exercises. We require that students can enter code in some editor. In the datasets students work on specific tasks, but because we could not find many accessible systems supporting tasks, we have also included more open-ended exercise types. We distinguish the following types:

- Tasks. The student has to solve a specific task that is specified in the system.
- Examples. The system provides example code that can be executed and modified.

Feedback type. Narciss [48] proposed a classification of the contents of feedback for computer-based learning environments, in which several instructional aspects (i.e., task rules, errors, and procedural knowledge) are considered. This classification has been extended for the programming domain by Keuning et al. [31] and has been applied to over 100 programming tools. We use this classification, extended with Narciss' *simple* feedback components. Table 4 shows the labels used and their short description. Below we describe in more detail the labels from the classification that we found in the systems from this study (taken and adapted from Keuning et al.).

Narciss describes three *simple* feedback components: 'Knowledge of performance' (KP), 'Knowledge of result/response' (KR), and 'knowledge of the correct results' (KCR). We do not consider 'Knowledge of performance' (KP), because this performance level feedback is on a *sequence* of tasks, while we focus on individual tasks. We include 'Knowledge of result/response' (KR), which conveys whether a solution is correct or incorrect. In the context of programming, a correct solution may (1) pass all test cases, (2) be equal to or similar to a model solution, (3) satisfy a number of constraints, or a combination of the preceding. In addition, we consider 'knowledge of the correct results' (KCR), which is a description or indication of a correct solution.

The next main types are *elaborated* feedback components. Each type addresses an element of the instructional context.

Feedback with 'knowledge about task constraints' (KTC) focuses on the task itself, and has the following subtypes:

- Hints on task requirements (TR). Examples are messages indicting to use a particular language construct, or not to use library methods.
- Hints on task-processing rules (TPR). These static hints provide some general help on how to approach the exercise, not taking the student's current solution into account.

For 'knowledge about concepts (KC)' two subtypes are distinguished. We only found messages with 'examples illustrating concepts' (EXA).

'Knowledge about mistakes (KM)' is the category found in most tools in Keuning et al.'s study [31]. KM messages have a type and a level of detail: basic (such as a numerical value, location, or short identifier, denoted by ○) or detailed (a more elaborate description, denoted by ●). In this study we also distinguish 'enhanced feedback' (★) as a level, indicating that a standard compiler/tool message has been converted to a more informative, student-friendly message. There are five different subtypes for KM messages:

- Test failures (TF). Many tools run test cases on student programs, presenting their results to students.
- Compiler errors (CE). These messages can point to syntax errors or semantic errors, and are not specific for an exercise.
- Solution errors (SE).
  This type of feedback deals with programs showing incorrect behaviour for a particular exercise. Examples are runtime errors, or logic errors (e.g. a loop that is executed one time more or less than expected).
- Style issues (SI). These issues do not affect the functionality of a program; however, many teachers consider learning a good programming style important for novice programmers. For example: formatting and documentation issues, structural issues, and problems with algorithm implementations.
- Performance issues (PI). This type could indicate a problem with the execution time of a program, or using too many resources.

Feed-forward messages contain 'knowledge about how to proceed' (KH). In this study we identified two of three types:

- Bug-related hints for error correction (EC). These messages clearly focuses on what the student should do to correct an error.
- Task-processing steps (TPS). These are next-step hints that should bring the student closer to a solution.

Feedback timing. The feedback given can be provided on demand or can be provided automatically by the system [48, 49]. We distinguish the various events from Table 7 as a possible trigger for a certain type of feedback.

Process. This part of the study was conducted by three authors. These authors each studied multiple systems and performed the initial coding. This coding was then checked by one of the other two authors, until an agreement on the coding was reached. For some systems, the second checker was an author of the system we had contacted (and not a member of this working group). In addition, we replayed a number of student sessions taken from the data sets and compared the system's feedback at the intervention points. Finally, we explored to what extent we could implement the expert's feedback into one of the systems.

6.2 Results

In this section we answer RQ2 by showing the results of analysing automated feedback in several programming learning environments, and relating that feedback to the expert-authored feedback. We have selected 18 programming learning environments for analysis. We show an overview of their characteristics in Table 5. From now on, we refer to each system by their name in SMALLCAPS. We distinguish several types of systems, of which the first two types were taken from Kim and Ko [35].

Interactive tutorials Defined as systems that "require learners to interact with command window, text editor, or equivalent in order to pass successive stages." Examples from this study include DataCamp, Codecademy, and Khan academy.

Web references These systems "play the role of a 'dictionary.' Tutorials under this genre ... help learners properly code against a library, API, or platform. Some web references ... provide code editors or command windows for learners who might want extra practice for reference code." Examples include W3Schools and LearnPython.

Coding practice systems These systems offer a large set of programming exercises, often at various levels, to practice programming. Similar systems are sometimes called online judges, or drill-and-practice systems. Examples are CodeWars and Kattis.

Automated assessment (AA) systems These systems focus on large scale assessment of programming submissions. This assessment is often summative (used for grading), but can also provide formative feedback to help students improve their submissions. Examples are JACK and GATE.

Another type of system we expected to find is the Intelligent Tutoring System (ITS). These systems help students solving problems step by step with feedback and hints. Although there are several ITSs for programming [14], we did not find any such systems easily available.

Table 6 shows the characteristics of the feedback the systems provide. Only JACK had been classified before [31]; which we have redone here based on the latest version and by testing the actual system as opposed to deriving information from publications. In the remainder of this section, we describe a number of systems in more detail, as well as the results of replaying student sessions from the datasets.

6.2.1 *Overview of Feedback in Learning Environments.*

In this section we present the learning environments with the most distinguishing features in terms of feedback and hints.

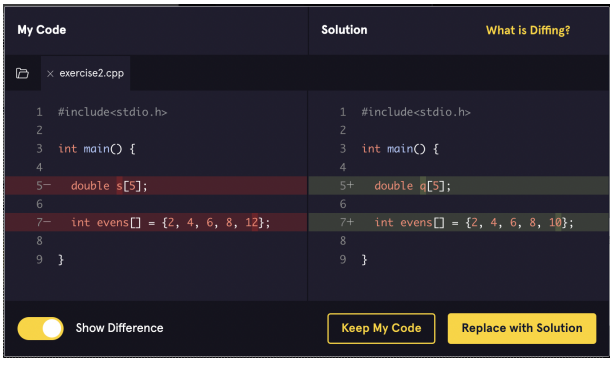
Codewars. Codewars is an online system for practicing and improving programming skills in a large number of programming languages. Programming challenges are offered by a big community and provide test-based feedback and the opportunity to compare own solutions with solutions from other (e.g. more experienced) programmers. The tasks are divided into different levels of difficulty and topics and allow the learner to practice specific topic areas. The quality and scope of the feedback depends on the unit-tests provided by the task creator. The feedback provided automatically by the system is KM-CE after clicking on a test button (run code). The same event gives information about the results of the public tests (KM-TF-d), if they were provided by the task creator. In addition to the event *Test*, there is a possibility to *Submit* the solution. If Secret Tests have been provided, the learner will receive information about the results of these tests (KM-TF-d). The system itself does not generate additional feedback, however it is possible for the learner to receive additional feedback through community discussions about the tasks and code comparison. Codewars relies heavily on gamification. Each successfully completed code challenge increases the experience points of the user's profile, which are associated with a certain skill level. This public information provides guidance for other programmers when comparing code. A special feature of the platform is that after successfully solving programming tasks, the learner can see the code of all other members for this task (i.e., receive KCR feedback) and compare it with his own or get inspiration for more efficient or smarter solution approaches.

Table 5: General characteristics of learning environments.

Name	Type	URL	Languages
Codewars	Coding practice	codewars.com	> 20
Codecademy	Interactive tutorial	codecademy.com	~15
Coderbyte	Coding practice	coderbyte.com	~10
CodingBat	Coding practice	codingbat.com	Java, Python
DataCamp	Interactive tutorial	datacamp.com	Python, R
FITech 101	Coding practice	fitech101.aalto.fi	Dart
FreeCodeCamp	Interactive tutorial	freecodecamp.org	JavaScript
Funprogramming	Web reference	funprogramming.org	Processing
GATE [65]	Automated assessment	gate.ifi.lmu.de	Java
HackInScience	Coding practice	hackinscience.org	Python
JACK [67]	Automated assessment	jack3-alpha.paluno.uni-due.de/demo/	Multiple
Kaggle	Coding practice	kaggle.com/learn	Python
Kattis	Coding practice	open.kattis.com/problems	22
Khan Academy	Interactive Tutorial	khanacademy.org/computing	JavaScript
LearnJS/Java/C	Web reference	learn-js.org	JS, Java, C
LearnPython	Web reference	learnpython.org	Python
Python Tutor [24]	Visualisation tool	pythontutor.com	Java, Python, JS, C, C++
W3Schools	Web reference	w3schools.com	Python, JS, Java, C++, and more

Codecademy. Codecademy is an online platform that offers coding classes for beginners and advanced programmers. The learning environment is based on a modular system where small learning units are combined to Skill Paths. If a learner finishes one of those units, multiple Skill Paths will be updated. The learning units are already divided into sub-goals, which must be worked through in a given sequence. Besides KR and KCR, a static hint is provided for each subgoal (KTC-TPR). When pushing the Run Button the system provides full compiler messages (if the submitted code has compile errors) (KM-CE R-d) and some additional hints on task processing rules (KTC-TPR) which are static and do not depend on the code of the learner. For each task it is possible to ask for a full solution and show the differences between the written code and the expected solution (see Figure 4). Since this is not only a comparison of the solution and the submitted code, but the differences were also highlighted by visual feedback, we have classified this as basic KM-SE.

Figure 4: Screenshot of the Compare Solution Feature in Codecademy.



CodingBat. CodingBat is an online coding practice system. It offers several programming problems to solve in Java or Python, in various categories and levels, typically by writing one method. When clicking on the 'go' button, several public and hidden test cases are run on the solution, and the results are shown to the student (KR, KM-TF, KM-CE).

The `Codingbat` dataset was collected in this system. The responses from the system are also recorded in the dataset, so replaying these can simply be done by inspecting the dataset. Figure 5 shows a screenshot of the feedback on a student step from the dataset, in which a list of passed and failed test cases is shown. Clicking on 'show hint' shows a predefined hint that does not take the current code status into account, referring to subgoals of establishing a base case and adding a recursive call (KTC). However, not all feedback types (e.g., KCR) are available for all tasks [34]. Below each task, additional material (e.g., similar, worked examples and solutions) is available as support. However, we deliberately did not classify this as KM-EXA, as it does not represent feedback on actions in the learning platform.

Figure 5: Screenshot of the feedback in CodingBat.

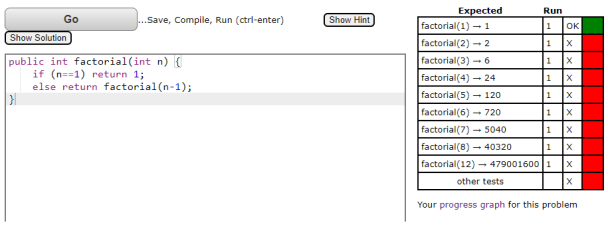


Table 6: Feedback characteristics of learning environments. The columns are described in Section 6.1.2. Table 7 shows the legend for the other letters and symbols used. We have omitted the feedback types we did not find in tools.

		Simple		KTC		KC		KM					KH		
Name	**Exercises**	KR	KCR	TR	TPR	EXP	EXA	TF	CE	SE	SI	PI	EC	TPS	IM
Codewars	Tasks	sb	sl					r→●	r→●						
Codecademy	Tasks	r	sl		h, r				r→●	sl→ ○					
Coderbyte	Tasks	r	sl					r→●	r→●						
CodingBat	Tasks	sb	sl	h	h			r→●	r→●						
DataCamp	Tasks	sb	sl		h			r→●	r, sb→●				sb	sb	
FITech 101	Tasks	sb						sb→●	p, r→●		p→●				
FreeCodeCamp	Tasks							r→●	p→●	r→●					
Funprogramming	Examples								r→●						
GATE	Tasks	sb						sb→★	sb→●	sb→●	sb→○	sb→○			
HackInScience	Tasks	sb	sl	sb					sb→★	sb→●			sb		
JACK	Tasks			sb				sb→★	sb→●	sb→●	sb→●/★				
Kaggle	Tasks	f+ r	f+ r		f+ r			r→●	r→●						
Kattis	Tasks							sb→○	sb→○			sb→○			
Khan Academy	Tasks								p→○, h→★	p→●			p		
LearnJS/Java/C	Tasks, examples	r	sl						r→●						
LearnPython	Tasks, examples	r	sl		r				r→●						
Python Tutor	Examples						r		r→●						
W3Schools	Examples								r→●						

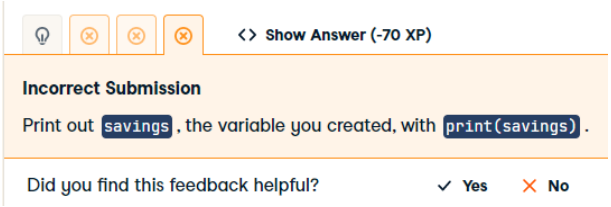
Table 7: Legend for feedback coding.

Symbol	Meaning
p	Code
r	Run
c	Compile
sb	Submit
h	Ask for hint
sl	Ask for solution
f	Function call
→	*leads to*
○	basic feedback
●	detailed feedback
★	enhanced/extended feedback

DataCamp. DataCamp is a popular online coding tool that offers several courses containing exercises to practice coding in Python and R. The system provides several options for help. Hints and solutions are limited by a points system. Correct solutions increase the amount of points. Using a hint decreases the amount of points. It is only possible to get a full solution if you have asked for a hint before. Moreover, detailed KM-TF, is displayed, but it is limited to one (the first) failed test case. The feedback is presented in two areas: the shell and a separate frame after submitting the solution. It is also possible to rate the helpfulness of the feedback given by the system. Figure 6 is a screenshot of the system showing feedback on an incorrect submission.

It was impossible to fully test our datasets because the freely available courses in DataCamp do not deal with recursion. However, based on the tasks examined, we conclude that the implementation of expert feedback in this system is possible but limited to generic hints that do no not relate to a user's input.

Figure 6: Screenshot of the feedback in DataCamp



FITech 101. FITech 101 is an e-book course platform developed by Aalto University. It offers a series of courses that are mainly targeted for lifelong learners. The content is offered only in Finnish and it covers introduction level courses for basic programming, databases, and web and mobile development. Dart is used as the

programming language, and Flutter is the framework used for mobile development. Programming exercises are embedded within the e-book materials of the courses and incorporate a customised version (added support for entering input) of DartPad[4]. The FITech 101 dataset was collected from this platform.

Dartpad is a browser-based IDE built by the Dart tools team as an online playground for Dart that also supports Dart for web and Flutter programming. The Dartpad environment allows executing Dart code directly in the browser (the Dart code is compiled to JavaScript for browser execution) and, similar to a desktop IDE, provides realtime feedback on syntax errors (KM-CE) and code quality (KM-SI) such as unused variables or dead code.

Task related feedback is received by submitting an exercise solution and the feedback is given by unit tests (KM-TF). Besides the DartPad IDE for code quality feedback and unit tests for task feedback, the learning platform provides an option to request help via a button next to the programming environment. These help requests are responded to by the course teachers within the learning platform. Learners may view their help requests, their responses and respond to teacher responses in a separate view in the course platform.

GATE. GATE is a web-based, platform-independent system for improving programming education and supporting tutors. Syntax and functional tests, as well as special tests for UML tasks, enable automatic correction and the provision of automated feedback. A special feature of the system is that feedback can only be requested in limited numbers (e.g. once). When submitting their answer, the students can choose between (a) only submit and (b) submit with feedback (limited to 1 request). Another special feature of the GATE system is that it is possible to provide randomized tasks. This means that variables can be used in tasks, which are assigned with individual values for each student. This allows learners to discuss tasks without having to worry about plagiarism for which the GATE system checks by means of an automatic plagiarism checker.

JACK. JACK is an e-assessment system that provides automated feedback to exercise solutions using different means of static and dynamic program analysis. Besides the ability to pass all compiler messages to the student (KM-CE) the system can execute test cases as white-box-tests and record all steps and variable values occurred during execution. From these data it can also detect unused code in terms of lines of code that are not reached by any test case and can create visualization of data structures. The system can execute teacher-defined test cases and provide both arbitrary feedback on test results and generic feedback on Java runtime exceptions (KM-TF). It can perform teacher-defined rule-based checks on the syntax graph of a solution and provide feedback on unwanted or missing code structures (KTC-TR, KM-SI). Static checks are formulated using the GReQL query language, which is based on the GRAL graph specification language. Learners can ask for hints, but they are predefined and not based on the actual state of the code (KTC-TPR)[67].

Kaggle. While in most of the learning environments studied by the working group hints, solutions and test results were provided via buttons, in Kaggle they have to be requested via a function call (see Figure 7). If a learner wants a hint for the task q3 they have to call the hint-function (`q3.hint()`). These functions print a simple static message. If there are unit-tests provided, then their results can be checked via `qr.check()`. When a student is stuck, or wants to compare their solution against a good solution, they can call the function `q3.solution()` to receive this information.

Figure 7: Screenshot of Kaggle.

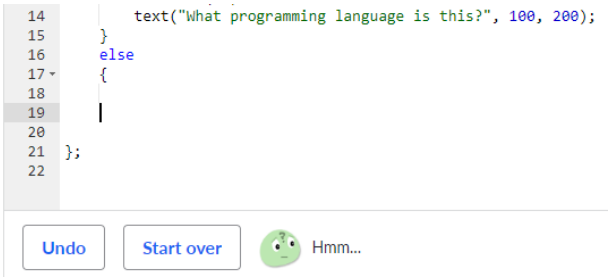
```
# Uncomment to get a hint
q3.hint()

# Uncomment to view solution
#q3.solution()

# DO NOT REMOVE: Check your answer
q3.check()
```

Khan academy. Khan academy is a learning platform that offers courses on a range of subjects, such as maths, economics, arts and humanities. For the subject of computer science a JavaScript course is available containing programming exercises on drawing and moving objects. Usually the student needs to expand starter code to achieve a certain goal. What stands out in this system is the timing of the feedback. Right after typing something incorrect, a green blob character, shown below the editor, responds with 'hmm…' (shown in Figure 8). Clicking on it leads to enhanced compiler error messages (KM-CE-e) and by clicking on 'show me where' the location (line) is highlighted in the editor (KM-CE-b). The system is also able to give more detailed feedback on specific errors in the student solution.

Figure 8: Screenshot of the feedback in Khan Academy.

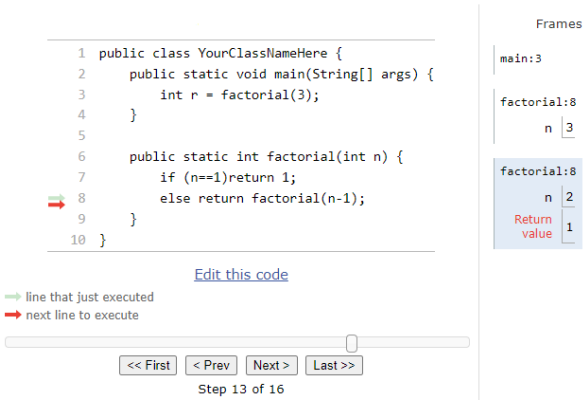


Python Tutor. The Python Tutor by Philip Guo [24] is an online system that focuses on visualizing code. Students can write code in several languages or select a program from a list of examples. The system gives standard compiler error messages (KM-CE) when the user presses "visualize execution." When the program compiles, the user can execute it step by step. We classified this feedback as KC-EXA, 'examples illustrating concepts'. Although slightly different, it can prove to be a valuable support in understanding a program and identifying problems to fix.

[4]https://dart.dev/tools/dartpad

We used a session from the `Codingbat` dataset for the Factorial exercise to replay in this system. A screenshot from the replay is shown in Figure 9. First, a class and main method need to be written, and the method needs to be called. The first snapshot that can be compiled can be stepped through. The student would see that the return value remains 1. In a subsequent snapshot the student extends the code with another recursive call, leading to 48 steps instead of 16 for factorial(3). When stepping through, it is shown that the step limit has been reached by calls with a negative parameter. The student then goes back to the previous state and later adds the correct calculation, showing the correct execution.

Figure 9: Screenshot of the visualisation in PythonTutor.



6.2.2 Human-authored vs Automated Feedback. In the following, we illustrate how some of the examined systems differ from the experts' consensus and guidelines on when and how to intervene. From inspecting the annotated datasets, we can conclude that different experts choose (based on clues like previous behaviour, code state, or timestamps) to give feedback at different locations in/-times during the programming process. This does not match with how learning environments provide feedback; these environments mostly leave it to the learner to perform an action (run, compile, or a hint button), after which feedback or a hint is shown. When students exhibit hint-avoiding behaviour, they are not helped in any way by the system. We propose that learning environments consider our guidelines for giving timely feedback, taking into account the previous actions of the student.

The experts' suggestion when and how to intervene in the `CodingBat` dataset related, for example, to positive feedback on the reached subgoals, such as writing the base case(s) with a viable if and return statement. Moreover, experts would intervene as soon as students falsify their code (e.g., B02, Fibonacci task), and when they change the given, correct code (e.g., B03, Fibonacci task). Experts further agree to intervene when observing tinkering behavior or continuous try-and-error submissions without a concept (e.g., B03, Fibonacci). In this context, experts suggest additional feedback addressing students' motivation to avoid dropouts. After long pauses without a student input, experts further suggest a question like "Do you need help?" (e.g., B05, Fibonacci). Codingbat, however, neither intervenes, nor does it offer these types of feedback. It thus completely relies on students requesting feedback. The only positive feedback provided by CodingBat refers to the positive output of the unit tests. In contrast to that, experts would combine correct unit test results with hints, as students do not seem to fully understand the display of the unit test results (e.g., A01, factorial). Similarly, experts would paraphrase and enhance compiler messages to (e.g., A05, factorial), but they have different views on how much assistance they want to provide.

In the `FITech` dataset, experts intervened early on when students were merely working towards the test cases and not aiming for a universally applicable solution. They pointed out that there are also hidden test cases and that a solution that only addresses the public test cases would not be effective. However, in some cases it is questionable whether it is really coding to the test or if a special case is used in a first step to build up the code structure and then adapt the algorithm in a second step. It also happened that learners deleted the `main()` method at the beginning, making any feedback in the FITech system impossible. The experts pointed this out with a short feedback message, but were still able to provide further feedback on the method to be written, which was no longer possible for the system. While the system only provides the compiler message pointing out a missing return statement, the experts used targeted feedback to draw attention to the difference between a print statement and a return statement.

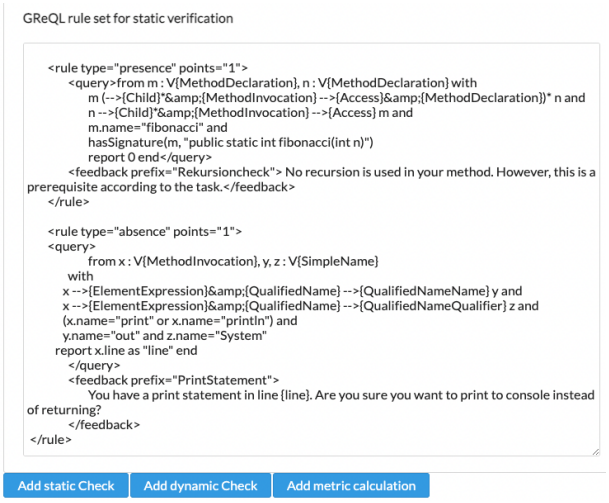
6.2.3 Implementing the Experts' Feedback. The working group investigated the extent to which the guidelines developed by the experts (see Table 2) can be implemented in the learning platforms and what problems arise in making expert feedback available in an automated manner. Most of the learning environments listed in Table 5 have predefined sequences of tasks including feedback and do not allow adding own tasks to the system or implementing extra feedback. This was only possible for us in CodeWars, FITech, GATE and JACK. Among the systems studied, the JACK system was the most suitable for this exploration because it offers many different ways to generate feedback. Moreover, it was the most accessible due to a close exchange with the developer. For the exploration, the programming task "Fibonacci" from the codingbat dataset was recreated in the JACK System so that the original feedback (unit tests and compiler messages) would be identical to the feedback messages in the dataset. Subsequently, the annotated expert feedback from the datasets was mapped to the systems' feedback options based on the elaborated feedback rules (see Table 2).

When implementing the feedback, the following two limitations were noted. First, the dynamic and static checks used to provide automated feedback only run when the code can be compiled without any errors. Thus, it must first always be ensured that there are no syntactic errors in the solution. However, in a context of teaching novice programmers, it is sometimes better to ignore syntax errors in a first step and address e.g. an critical logical error with valuable feedback first. The second limitation is the timing of the feedback. The experts guidelines in Table 2 *when* to intervene refer to the amount of compilation where the error goes unaddressed. Since JACK, like many e-assessment systems, is based on solution submission, no information is available to the system about how many times the code was compiled between two submissions.

Besides these limitations, it was possible to implement the expert feedback into the system. Figure 10 shows two of the GReQL rule

sets (static code checks) which we have entered into the JACK system as part of the working group. The first rule checks whether recursion (as a requirement of the task) is used in the submitted answer. The second rule checks if there are print statements in the response and indicates that values must be returned and not output to the console. Similarly, other rules could be written which, for example, check whether a base case is in the solution and then give positive feedback.

Figure 10: Static code checks in the JACK System using GReQL.



7 DISCUSSION

7.1 Datasets

We found five datasets with the desired characteristics. The datasets use multiple programming languages, and have varying levels of granularity. It was surprisingly hard to find such datasets. Datasets mentioned in previous work were either not available anymore, like the code.org dataset, or unsuitable for various reasons.

We discarded multiple datasets because the level of granularity was too high. For example, the submission data from the 2nd-CSEDM data challenge[5], available on DataShop, does not give us enough information at the desired level of detail about the progress of students.

Another common problem is that we cannot find the problem descriptions for the tasks students worked on, which makes it hard to for example give hints. For this reason we did not include the BlackBox [10] dataset.

The ITAP dataset ('ICER - All Attempts - All Steps - ITAP Goal'), available on DataShop, contains sequences of steps of students programming in the ITAP learning environment for programming in Python [56]. The dataset contains a bit more than 25,000 steps, from interactions with 89 students. This is an interesting dataset, but we discarded it because we could not determine complete sequences of students from starting to work on a solution to completing it.

[5]https://sites.google.com/ncsu.edu/csedm-dc-2021/home

7.2 Interventions

In this paper, we designed interventions from a human expert perspective. The experts aimed to develop feasible intervention guidelines, when and how, by setting simple rules while also trying to provide as much meaningful feedback as possible. As expected, it is a significant challenge to design such a system.

All our experts (authors of this paper) have extensive programming experience over many years with several languages. They all have taught programming to novice students but they vary in their pedagogical experience. They often chose to provide feedback at different locations/times (e.g., when the student paused versus immediately), at different levels of specificity (e.g., about a particular line versus about a concept), and using different tone and language (e.g., recommendations, requests, reflection prompts). These variations in feedback are artifacts of differences in pedagogical philosophy and/or differing assumptions about the context of the learning and are not unlike the differences between experts reported by Dong et al. [18] such as disagreeing on "which issue to fix first". Most differences were attributed to "preferences in tutoring style" [18].

For a number of datasets we have little knowledge about the context in which the data was collected. Besides the task prompts, the context and the sub-goals that are relevant for the task are very useful for providing feedback. While knowledge about the students could be derived from earlier data, experts typically only get to consider a student's ability at the moment they look at particular steps.

The guidelines tend to delay intervention (except when a hint is explicitly requested) but allow for flexibility on how long to wait. Most interventions are triggered by identified struggles, such as when a student is struggling to complete a subgoal. However, positive feedback was provided when a subgoal is completed (i.e. when an achievement takes place). Positive feedback plays two roles: it provides encouragement to a student, and it helps students that are unsure about their work, and reduces the chance that they undo correct steps [16, 42]. Furthermore, positive feedback enhances self-efficacy [59], although he also noted that "this increase will be temporary if subsequent efforts turn out poorly".

Immediate and extensive feedback is not always the best way to learn. According to Bjork et al. [8] desirable difficulties, such as delayed feedback, may lead to better learning.

Our guidelines give students a chance to process the system's feedback on their own, to learn how to deal with syntax and logical errors, for example. When they fail to do so, the feedback from the system (as a result from a run-action) should not be duplicated but complemented. To be able to do this, datasets should log the feedback from the system given to the students.

The annotater variation in intervention locations/times is an opportunity, both for researchers who can directly compare different intervention points and methods, and for tool developers, who have flexibility in implementing "human-like" feedback. We urge further work in this area to help guide tool designers to providing effective methods for feedback.

7.3 Learning environments

Research vs practice. In our literature study, we found several systems that are considered Intelligent Tutoring systems. However, none of these systems could be easily reviewed in practice. In the reviews (e.g., [31]) we found many examples of feedback aimed at identifying logical errors and providing hints on how to correct them, usually by employing static analysis methods. However, in practice, we see only a few systems that actually integrated this type of feedback. Also, we observe that only a few systems provide feedback on code style and quality.

Finally, studies have shown that binary (correct/incorrect) feedback is not very effective or could even have an adverse effect [25, 38]. We note that most systems we investigated go slightly beyond simple binary feedback, but there is still room for improvement.

Accessibility. We are aware of but did not review several systems that support students taking steps towards a solution by providing hint and feedback on steps. We were unable to include them in this study, because they were not (easily) accessible to us. For instance, ITAP[6] uses data-driven hints and is available on GitHub, but has no instructions on how to deploy it. The CloudCoder project[7] ended in early 2022. In fact, although several papers refer to public repositories, demoing or deploying such systems is challenging. In addition, the corresponding systems found in the literature studies were too old or did not provide a link to download or access the system. These problems have been raised by other researchers, and we found this was still the case. For example, another 2022 working group studies problems related to the accessibility, discoverability, and dissemination of various tools [9].

A second issue when evaluating learning environments is related to the types of exercises that are (not) offered. Students often work on specific tasks to practice programming, and our datasets contain student steps for solving specific tasks. Ideally, we would like to investigate all of the programming tools using the same task – to be able to add the tasks from our dataset. However, we found that this was not trivial: very few tools made access as a problem author easy.

Feedback. Several well-known and widely used systems provide limited feedback to students. For example, the web reference W3Schools has the option to try out code snippets and provides a console with extended compiler messages and backtraces. This kind of feedback might be discouraging and difficult to interpret for novices learning to program. We refer to the extensive research on compiler error messages for an in-depth analysis (see e.g., Becker et al. [5]).

In many systems, hints and a solution are readily available to students. We have also noticed an increase in the introduction of credit systems: in DataCamp every user has an amount of 'XP' (experience points). XPs are defined as "a way of measuring your engagement within DataCamp. It calculates automatically based on courses, exercises, or other content that you complete." When asking for a hint or solution, the user loses points. The GATE system even allows only one request for feedback. Students must first test their code themselves and can then request feedback once.

[6] https://github.com/krivers/ITAP-django
[7] http://cloudcoder.org/

The working group's examination of learning platforms also shows that the provided feedback is primarily focused on symptoms and does not focus on the cause of errors. Research shows that students view the latter as an important criterion for valuable feedback [40].

7.4 Threats to validity

We observe the following threats to validity related to the datasets. First of all, the selection of datasets constitutes a limitation. As discussed, several reasons lead to the selected datasets, as others were not available, or did not provide the required details. However, more data would certainly increase validity. Second, although experts' ratings were confirmed and evaluated by other coders in terms of intercoder-reliability, the annotated datasets exhibit the biases of our experts. Third, the exercises in the annotated datasets were small and relatively simple. We expect annotating student steps to be more difficult and perhaps less reliable when exercise difficulty increases.

Another aspect is that only a few student submissions per problem have been analyzed due to the extensive effort required for qualitative analysis. More comparable student data for the same tasks/problems would help increase the results' validity. This challenge is reflected in the wide range of learning environments, which do not all offer the same tasks/problems, or even type of practice. Moreover, the access to learning environments was in many cases restricted, preventing us from reviewing more systems and corresponding feedback.

For some of the datasets, we do not have the complete knowledge about the context in which the data was collected. For instance, the Codingbat dataset was recorded in a usability laboratory and others were not, so the impact of such conditions had to be ignored in this study. Therefore, while our expert-annotated datasets represent our best effort by our panel of experts, we would not want to call these datasets 'Golden Datasets'.

8 CONCLUSIONS AND FUTURE WORK

This section revisits and answers the research questions formulated in the introduction, and describes future work.

RQ1 How should we annotate datasets consisting of steps students take towards solving a programming task with information about when and how to give feedback and hints?

To answer this question, we first looked for datasets with fine-grained steps, that is, datasets that have steps denoting progress towards final submissions. In the process, we identified several of these datasets of different granularities. The finest granularity was that of keystroke level, which we found too fine-grained for annotating steps manually. For this reason, we converted keystroke level steps into token level steps, using a tool developed as part of this work (see Subsection 4.3). Besides keystroke level data, we also found datasets with steps at pause level, token$^+$ level, and block-edit level (in block-based languages) granularity.

Based on the literature and experts' recommendations, we designed guidelines for when to give feedback and hints, and how to do that. We used these guidelines to annotate several sequences of student steps from several datasets. In most cases, the different annotators reached a reasonable agreement on the annotations of the

dataset. The annotated datasets have been made publicly available, together with a tool for browsing through the datasets more easily.

RQ2 How does expert feedback relate to the feedback found in learning environments for programming?

We analysed multiple recent learning environments and categorised them by the types of feedback they provide and when they provide the feedback. We then compared the feedback in the annotated datasets against that provided by the analysed learning environments. We found that experts tend to provide timely feedback based on student behaviour while learning environments mostly wait for a student explicitly requesting feedback via e.g. a hint button. Furthermore, if possible and available, experts take context into account when constructing feedback, and try to identify the strategy of the students and possible misconceptions. We did not find learning environments that would provide such personalised feedback. We note, however, that there was plenty of disagreement between experts on when to intervene and also some disagreement on how to intervene.

Future work. An important contribution of this working group is that we found that there is still much work to be done to determine what kind of feedback to give, how to assess the feedback given by learning environments, and ultimately how to improve these environments. There are several directions for future work. First, since we think the expert-annotated datasets are a very useful resource, we encourage researchers to annotate more data. Second, we want to use the expert-annotated datasets to generalize how experts give feedback and hints on steps students take when working on programming tasks. These insights will be used to describe the design of a system giving automated feedback on student steps. Third, we also want to annotate steps students take when working on slightly larger tasks. Since possibilities to go off track are larger here, we need to think of a method to deal with this, probably involving connecting student steps to subgoals of a task. Fourth, we want to study what kind of feedback students expect at the various steps they take, and how this compares against the feedback specified by experts.

REFERENCES

[1] Haiyang Ai. 2017. Providing graduated corrective feedback in an intelligent computer-assisted language learning environment. *ReCALL* 29, 3 (2017), 313–334.

[2] Hugo Arends, Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2017. An intelligent tutor to learn the evaluation of microcontroller I/O programming expressions. In *Proceedings of the 17th Koli Calling International Conference on Computing Education Research*. 2–9.

[3] Hemilis Joyse Barbosa Rocha, Patrícia Cabral De Azevedo Restelli Tedesco, and Evandro De Barros Costa. 2022. On the use of feedback in learning computer programming by novices: a systematic literature mapping. *Informatics in Education* (2022).

[4] Joseph E. Beck, Kai Min Chang, Jack Mostow, and Albert Corbett. 2008. Does help help? Introducing the Bayesian Evaluation and Assessment Methodology. In *Proceedings of the International Conference on Intelligent Tutoring Systems*.

[5] Brett A. Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, Janice L. Pearce, and James Prather. 2019. Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education* (Aberdeen, Scotland Uk) *(ITiCSE-WGR '19)*. Association for Computing Machinery, New York, NY, USA, 177–210. https://doi.org/10.1145/3344429.3372508

[6] Brett A Becker, Graham Glanville, Ricardo Iwashima, Claire McDonnell, Kyle Goslin, and Catherine Mooney. 2016. Effective compiler error message enhancement for novice programming students. *Computer Science Education* 26, 2-3 (2016), 148–175.

[7] Brett A Becker, Kyle Goslin, and Graham Glanville. 2018. The effects of enhanced compiler error messages on a syntax error debugging test. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. 640–645.

[8] Elizabeth Bjork and Robert Bjork. 2011. Making things hard on yourself, but in a good way: Creating desirable difficulties to enhance learning. *Psychology and the Real World: Essays Illustrating Fundamental Contributions to Society* 2 (01 2011), 56–64.

[9] Jeremiah Blanchard, John R Hott, Vincent Berry, Rebecca Carroll, Bob Edmison, Richard Glassey, Oscar Karnalim, Brian Plancher, and Seán Russell. 2022. Leveraging Community Software in CS Education to Avoid Reinventing the Wheel. In *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 2*. 580–581.

[10] Neil Christopher Charles Brown, Michael Kölling, Davin McCall, and Ian Utting. 2014. Blackbox: A Large Scale Repository of Novice Programmers' Activity. In *Proceedings of the ACM Technical Symposium on Computer Science Education (SIGCSE)*.

[11] Richard Catrambone. 2012. *Subgoal Learning*. Springer US, Boston, MA, 3230–3233. https://doi.org/10.1007/978-1-4419-1428-6_55

[12] Anderson Pinheiro Cavalcanti, Arthur Barbosa, Ruan Carvalho, Fred Freitas, Yi-Shan Tsai, Dragan Gašević, and Rafael Ferreira Mello. 2021. Automatic feedback in online learning environments: A systematic literature review. *Computers and Education: Artificial Intelligence* 2 (2021), 100027. https://doi.org/10.1016/j.caeai.2021.100027

[13] Helder Correia, José Paulo Leal, and José Carlos Paiva. 2017. Enhancing Feedback to Students in Automated Diagram Assessment. In *6th Symposium on Languages, Applications and Technologies, SLATE 2017, June 26-27, 2017, Vila do Conde, Portugal (OASICS, Vol. 56)*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. https://doi.org/10.4230/OASIcs.SLATE.2017.11

[14] Tyne Crow, Andrew Luxton-Reilly, and Burkhard Wuensche. 2018. Intelligent Tutoring Systems for Programming Education: A Systematic Review. In *Proceedings of the 20th Australasian Computing Education Conference* (Brisbane, Queensland, Australia) *(ACE '18)*. ACM, New York, USA, 53–62. https://doi.org/10.1145/3160489.3160492

[15] Galina Deeva, Daria Bogdanova, Estefanía Serral, Monique Snoeck, and Jochen De Weerdt. 2021. A review of automated feedback systems for learners: Classification framework, challenges and opportunities. *Computers & Education* 162 (2021), 104094. https://doi.org/10.1016/j.compedu.2020.104094

[16] Barbara Di Eugenio, Davide Fossati, Stellan Ohlsson, and David Cosejo. 2009. Towards explaining effective tutorial dialogues. In *Annual Meeting of the Cognitive Science Society*. Citeseer, 1430–1435.

[17] Yihuan Dong, Samiha Marwan, Veronica Catete, Thomas W. Price, and Tiffany Barnes. 2019. Defining Tinkering Behavior in Open-ended Block-based Programming Assignments. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. ACM, 1204–1210.

[18] Yihuan Dong, Preya Shabrina, Samiha Marwan, and Tiffany Barnes. 2021. You Really Need Help: Exploring Expert Reasons for Intervention During Block-Based Programming Assignments. In *Proceedings of the 17th ACM Conference on International Computing Education Research* (Virtual Event, USA) *(Icer 2021)*. Association for Computing Machinery, New York, NY, USA, 334–346. https://doi.org/10.1145/3446871.3469764

[19] Christopher Douce, David Livingstone, and James Orwell. 2005. Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing (JERIC)* 5, 3 (2005), 4–es.

[20] John Edwards. 2022. 2021 CS1 Keystroke Data. https://doi.org/10.7910/DVN/BVOF7S

[21] Davide Fossati, Barbara Di Eugenio, Stellan Ohlsson, Christopher Brown, and Lin Chen. 2015. Data Driven Automatic Feedback Generation in the iList Intelligent Tutoring System. *Technology, Instruction, Cognition and Learning* 10, 1 (2015), 5–26.

[22] Alex Gerdes, Bastiaan Heeren, Johan Jeuring, and L Thomas van Binsbergen. 2017. Ask-Elle: an adaptable programming tutor for Haskell giving automated feedback. *International Journal of Artificial Intelligence in Education* 27, 1 (2017), 65–100.

[23] Andreas Giannakoulas and Stelios Xinogalos. 2020. A review of educational games for teaching programming to primary school students. *Handbook of Research on Tools for Teaching Computational Thinking in P-12 Education* (2020).

[24] Philip J Guo. 2013. Online python tutor: embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM technical symposium on Computer science education*. 579–584.

[25] Qiang Hao, David H Smith IV, Lu Ding, Amy Ko, Camille Ottaway, Jack Wilson, Kai H Arakawa, Alistair Turcan, Timothy Poehlman, and Tyler Greer. 2022. Towards understanding the effective design of automated formative feedback for programming assignments. *Computer Science Education* 32, 1 (2022), 105–127.

[26] John Hattie and Helen Timperley. 2007. The Power of Feedback. *Review of Educational Research* 77, 1 (2007), 81–112. https://doi.org/10.3102/003465430298487 arXiv:https://doi.org/10.3102/003465430298487

[27] Petri Ihantola, Arto Vihavainen, Alireza Ahadi, Matthew Butler, Jürgen Börstler, Stephen H. Edwards, Essi Isohanni, Ari Korhonen, Andrew Petersen, Kelly Rivers, Miguel Ángel Rubio, Judy Sheard, Bronius Skupas, Jaime Spacco, Claudia Szabo, and Daniel Toll. 2015. Educational Data Mining and Learning Analytics in Programming: Literature Review and Case Studies. In *Proceedings of the 2015 ITiCSE on Working Group Reports* (Vilnius, Lithuania) *(ITiCSE-WGR '15)*. Association for Computing Machinery, New York, NY, USA, 41–63. https://doi.org/10.1145/2858796.2858798

[28] Cruz Izu, Violetta Lonati, Anna Morpurgo, and Mario Sanchez. 2021. An Inventory of Goals from CS1 Programs Processing a Data Series. In *2021 IEEE Frontiers in Education Conference (FIE)*. 1–8. https://doi.org/10.1109/FIE49875.2021.9637360

[29] Cruz Izu, Amali Weerasinghe, and Cheryl Pope. 2016. A Study of Code Design Skills in Novice Programmers Using the SOLO Taxonomy. In *Proceedings of the 2016 ACM Conference on International Computing Education Research* (Melbourne, VIC, Australia) *(ICER '16)*. Association for Computing Machinery, New York, NY, USA, 251–259. https://doi.org/10.1145/2960310.2960324

[30] Samantha Jiménez, Reyes Juárez-Ramírez, Víctor H Castillo, Guillermo Licea, Alan Ramírez-Noriega, and Sergio Inzunza. 2018. A feedback system to provide affective support to students. *Computer Applications in Engineering Education* 26, 3 (2018), 473–483.

[31] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2018. A Systematic Literature Review of Automated Feedback Generation for Programming Exercises. *ACM Trans. Comput. Educ.* 19, 1, Article 3 (sep 2018), 43 pages. https://doi.org/10.1145/3231711

[32] Natalie Kiesler. 2022. Dataset: Recursive problem solving in the online learning environment CodingBat by computer science students. Online. https://doi.org/10.21249/DZHW:studentsteps:1.0.0 Datenerhebung: 2017. Version: 1.0.0. Datenpaketzugangsweg: Download-SUF. Hannover: FDZ-DZHW. Datenkuratierung: İkiz-Akıncı, Dilek.

[33] Natalie Kiesler. 2022. *Daten- und Methodenbericht Rekursive Problemlösung in der Online Lernumgebung CodingBat durch Informatik-Studierende.* Technical Report. https://metadata.fdz.dzhw.eu/public/files/data-packages/stu-studentsteps$/attachments/studentsteps_Data_Methods_Report_de.pdf

[34] Natalie Kiesler. 2022. An Exploratory Analysis of Feedback Types Used in Online Coding Exercises. https://doi.org/10.48550/ARXIV.2206.03077

[35] Ada S. Kim and Amy J. Ko. 2017. A Pedagogical Analysis of Online Coding Tutorials. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (Seattle, Washington, USA) *(SIGCSE '17)*. Association for Computing Machinery, New York, NY, USA, 321–326. https://doi.org/10.1145/3017680.3017728

[36] K.R. Koedinger, R.S.J.d. Baker, K. Cunningham, A. Skogsholm, B. Leber, and J. Stamper. 2010. A Data Repository for the EDM community: The PSLC DataShop. In *Handbook of Educational Data Mining*, C. Romero, S. Ventura, M. Pechenizkiy, and R.S.J.d. Baker (Eds.). CRC Press: Boca Raton, FL.

[37] Stephan Krusche and Andreas Seitz. 2018. Artemis: An automatic assessment management system for interactive learning. In *Proceedings of the 49th ACM technical symposium on computer science education*. 284–289.

[38] Angelo Kyrilov and David C Noelle. 2016. Do students need detailed feedback on programming exercises and can automated assessment systems provide it? *Journal of Computing Sciences in Colleges* 31, 4 (2016), 115–121.

[39] Abe Leite and Saúl A Blanco. 2020. Effects of human vs. automatic feedback on students' understanding of AI concepts and programming style. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. 44–50.

[40] Dominic Lohr and Marc Berges. 2021. Towards Criteria for Valuable Automatic Feedback in Large Programming Classes. (2021).

[41] Samiha Marwan. 2021. *Investigating Best Practices in the Design of Automated Hints and Formative Feedback to Improve Students' Cognitive and Affective Outcomes.* PhD. North Carolina State University.

[42] Samiha Marwan, Bita Akram, Tiffany Barnes, and Thomas W Price. 2022. Adaptive Immediate Feedback for Block-Based Programming: Design and Evaluation. *IEEE Transactions on Learning Technologies* (2022), 406–420.

[43] Samiha Marwan, Ge Gao, Susan Fisk, Thomas W. Price, and Tiffany Barnes. 2020. Adaptive Immediate Feedback Can Improve Novice Programming Engagement and Intention to Persist in Computer Science. In *Proceedings of the ACM Conference on International Computing Education Research (ICER)*.

[44] Samiha Marwan, Joseph Jay Williams, and Thomas Price. 2019. An Evaluation of the Impact of Automated Programming Hints on Performance and Learning. In *Proceedings of the International Computing Education Research Conference*.

[45] S. Marwan, N. Lytle, J. J. Williams, and T. W. Price. 2019. The Impact of Adding Textual Explanations to Next-step Hints in a Novice Programming Environment. In *Proceedings of the 24th Annual ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE19 (forthcoming)*.

[46] Bruce M. Mclaren, Tamara Gog, Craig Ganoe, David Yaron, and Michael Karabinos. 2014. Exploring the Assistance Dilemma: Comparing Instructional Support in Examples and Problems. In *12th International Conference on Intelligent Tutoring Systems - Volume 8474* (Honolulu, HI, USA) *(ITS 2014)*. Springer-Verlag, Berlin, Heidelberg, 354–361. https://doi.org/10.1007/978-3-319-07221-0_44

[47] Briana B. Morrison, Lauren E. Margulieux, and Mark Guzdial. 2015. Subgoals, Context, and Worked Examples in Learning Computing Problem Solving. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research* (Omaha, Nebraska, USA) *(ICER '15)*. Association for Computing Machinery, New York, NY, USA, 21–29. https://doi.org/10.1145/2787622.2787733

[48] Susanne Narciss. 2008. Feedback strategies for interactive learning tasks. *Handbook of research on educational communications and technology* (2008), 125–144.

[49] Susanne Narciss. 2020. Feedbackstrategien für interaktive Lernaufgaben. *Handbuch Bildungstechnologie*, 369–392.

[50] Benjamin Paaßen. 2019. Python Programming Dataset. (2019). https://doi.org/10.4119/unibi/2941052 Bielefeld University.

[51] José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. 2022. Automated Assessment in Computer Science Education: A State-of-the-Art Review. *ACM Trans. Comput. Educ.* (jan 2022). https://doi.org/10.1145/3513140 Just Accepted.

[52] Thomas W. Price, Yihuan Dong, and Dragan Lipovac. 2017. iSnap: towards intelligent tutoring in novice programming environments. In *Proceedings of the ACM SIGCSE Technical Symposium on computer science education*.

[53] Thomas W. Price, Yihuan Dong, Rui Zhi, Benjamin Paaßen, Nicholas Lytle, Veronica Cateté, and Tiffany Barnes. 2019. A comparison of the quality of data-driven programming hint generation algorithms. *International Journal of Artificial Intelligence in Education* 29, 3 (2019).

[54] Thomas W. Price, David Hovemeyer, Kelly Rivers, Ge Gao, Austin Cory Bart, Ayaan M. Kazerouni, Brett A. Becker, Andrew Petersen, Luke Gusukuma, Stephen H. Edwards, and David Babcock. 2020. ProgSnap2: A Flexible Format for Programming Process Data. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education* (Trondheim, Norway) *(ITiCSE '20)*. Association for Computing Machinery, New York, NY, USA, 356–362. https://doi.org/10.1145/3341525.3387373

[55] Robert S. Rist. 1991. Knowledge Creation and Retrieval in Program Design: A Comparison of Novice and intermediate Student Programmers. *Human–Computer Interaction* 6, 1 (1991), 1–46. https://doi.org/10.1207/s15327051hci0601_1 arXiv:https://doi.org/10.1207/s15327051hci0601_1

[56] Kelly Rivers and Kenneth R Koedinger. 2017. Data-driven hint generation in vast solution spaces: a self-improving Python programming tutor. *International Journal of Artificial Intelligence in Education* 27, 1 (2017).

[57] T. J. Robertson, Shrinu Prabhakararao, Margaret Burnett, Curtis Cook, Joseph R. Ruthruff, Laura Beckwith, and Amit Phalgune. 2004. Impact of Interruption Style on End-User Debugging. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Vienna, Austria) *(CHI '04)*. Association for Computing Machinery, New York, NY, USA, 287–294. https://doi.org/10.1145/985692.985729

[58] Manuel Rubio-Sánchez, Päivi Kinnunen, Cristóbal Pareja-Flores, and Ángel Velázquez-Iturbide. 2014. Student perception and usage of an automated programming assessment tool. *Computers in Human Behavior* 31 (2014), 453–460.

[59] Dale H. Schunk. 1995. *Self-Efficacy and Education and Instruction.* Springer US, Boston, MA, 281–303. https://doi.org/10.1007/978-1-4419-6868-5_10

[60] Raj Shrestha, Juho Leinonen, Arto Hellas, Petri Ihantola, and John Edwards. 2022. CodeProcess Charts: Visualizing the Process of Writing Code. In *Australasian Computing Education Conference*. Association for Computing Machinery, New York, NY, USA, 46–55. https://doi.org/10.1145/3511861.3511867

[61] Valerie J. Shute. 2008. Focus on formative feedback. *Review of Educational Research* 78, 1 (2008).

[62] Tze Ying Sim and Sian Lun Lau. 2018. Online tools to support novice programming: A systematic review. In *2018 IEEE Conference on e-Learning, e-Management and e-Services (IC3e)*. IEEE, 91–96.

[63] Draylson M Souza, Katia R Felizardo, and Ellen F Barbosa. 2016. A systematic literature review of assessment tools for programming assignments. In *2016 IEEE 29Th international conference on software engineering education and training (CSEET)*. IEEE, 147–156.

[64] Step. 2022. *In Merriam-Webster.com.* https://www.merriam-webster.com/dictionary/step

[65] Sven Strickroth and Florian Holzinger. [n.d.]. Supporting the Semi-Automatic Feedback Provisioning on Programming Assignments. *Methodologies and Intelligent Systems for Technology Enhanced Learning* ([n. d.]).

[66] Sven Strickroth and Michael Striewe. 2022. Building a Corpus of Task-based Grading and Feedback Systems for Learning and Teaching Programming. *The International Journal of Engineering Pedagogy* (2022).

[67] Michael Striewe. 2015. *Automated analysis of software artefacts-a use case in e-assessment.* Ph.D. Dissertation. Duisburg, Essen, Universität Duisburg-Essen, Diss., 2014.

[68] Kurt VanLehn. 2006. The behavior of tutoring systems. *International journal of artificial intelligence in education* 16, 3 (2006), 227–265.

[69] Kurt VanLehn. 2011. The Relative Effectiveness of Human Tutoring, Intelligent Tutoring Systems, and Other Tutoring Systems. *Educ. Psych.* 46, 4 (2011).

[70] Burkhard C Wünsche, Edward Huang, Lindsay Shaw, Thomas Suselo, Kai-Cheung Leung, Davis Dimalen, Wannes van der Mark, Andrew Luxton-Reilly, and Richard Lobb. 2019. CodeRunnerGL-An interactive web-based tool for computer graphics teaching and assessment. In *2019 International Conference on Electronics, Information, and Communication (ICEIC)*. IEEE, 1–7.

[71] Wenju Zhou, Yigong Pan, Yinghua Zhou, and Guangzhong Sun. 2018. The framework of a new online judge system for programming education. In *Proceedings of ACM Turing Celebration Conference-China*. 9–14.

A TASK DESCRIPTIONS

Figure 1: Prompt and solution for the Password problem. Translated from Finnish.

```
Write a one-parameter function askPassword. When
the function is called, it asks the user for a pass-
word until the user enters the correct password.
The correct password is given to the function as
a parameter. When the user enters the correct
password (i.e., the user's input corresponds to the
value received as a parameter), the program prints
the message 'Thank you!' To ask for a password,
use the string 'Enter password.'

Below is an example of the function when the function
is called in the form askPassword('turnip');

Enter password.
< cauliflower
Enter password.
< carrot
Enter password.
< turnip
Thank you!
----------

askPassword(correct) {
  while (true) {
    print('Enter password.');
    var input = stdin.readLineSync();
    if (input == correct) {
      break;
    }
  }
  print('Thank you!');
}
```

Figure 2: Prompt and solution for the Temperature problem. Translated from Finnish.

```
Write a one-parameter function named explanation
that, when called, returns information about whether
the number given as a parameter is positive, neg-
ative, or zero. If the number is positive, the func-
tion should return the string 'Positive!' If the
number is negative, the function should return the
string 'Negative!' Otherwise, the function should
return the string 'Zero!'
----------

explanation(number) {
  if (number > 0) {
    return 'Positive!';
  } else if (number < 0) {
    return 'Negative!';
  } else {
    return 'Zero!';
  }
}
```

Figure 3: Prompt and solution for the Fibonacci problem.

```
The Fibonacci sequence is a famous bit of mathe-
matics, and it happens to have a recursive defi-
nition. The first two values in the sequence are 0
and 1 (essentially 2 base cases). Each subsequent
value is the sum of the previous two values, so
the whole sequence is: 0, 1, 1, 2, 3, 5, 8, 13, 21
and so on.

Define a recursive fibonacci(n) method that re-
turns the nth fibonacci number, with n=0 repre-
senting the start of the sequence.

fibonacci(0) -> 0
fibonacci(1) -> 1
fibonacci(2) -> 1
----------

public int fibonacci(int n) {
    if (n==0) return 0;
    if (n==1) return 1;
    return fibonacci(n-1)+ fibonacci(n-2);
}
```

Figure 4: Prompt and solution for the Factorial problem.

```
Given n of 1 or more, return the factorial of n,
which is n * (n-1) * (n-2) ... 1. Compute the result
recursively (without loops).

factorial(1) -> 1
factorial(2) -> 2
factorial(3) -> 6
----------

public int factorial(int n) {
    if (n == 1)
        return 1;
    else
        return n * factorial(n - 1);
}
```