Dataset of Student Solutions to Algorithm and Data Structure Programming Assignments

Fynn Petersen-Frey[1], Marcus Soll[2], Louis Kobras[3]
Melf Johannsen[4], Peter Kling[5], Chris Biemann[1]

[1] Language Technology Group, Dept. of Informatics, Universität Hamburg
[2] NORDAKADEMIE gAG Hochschule der Wirtschaft, Köllner Chaussee 11, 25337 Elmshorn
[3] Fachbereich Informatik, Universität Hamburg
[4] Center for Optical Quantum Technologies, Fachbereich Physik, Universität Hamburg
[5] Theory of Efficient Algorithms Group, Dept. of Informatics, Universität Hamburg
{fynn.petersen-frey, louis.kobras, peter.kling, christian.biemann}@uni-hamburg.de
marcus.soll@nordakademie.de, mjohanns@physnet.uni-hamburg.de

Abstract

We present a dataset containing source code solutions to algorithmic programming exercises solved by hundreds of Bachelor-level students at the Universität Hamburg. These solutions were collected during the winter semesters 2019/2020, 2020/2021 and 2021/2022. The dataset contains a set of solutions to a total of 21 tasks written in Java as well as Python and a total of over 1500 individual solutions. All solutions were submitted through Moodle and the Coderunner plugin and passed a number of test cases (including randomized tests), such that they can be considered as working correctly. All students whose solutions are included in the dataset gave their consent into publishing their solutions. The solutions are pseudonymized with a random solution ID. Included in this paper is a short analysis of the dataset containing statistical data and highlighting a few anomalies (e.g. the number of solutions per task decreases for the last few tasks due to grading rules). We plan to extend the dataset with tasks and solutions from upcoming courses.

Keywords: dataset, programming, algorithm, sourcecode, java, python

1. Introduction

Many people believe that the ability to write computer programs is an important skill to form the world we are living in (Körber-Stiftung, 2017), yet only a small part of the German society is able to actually handle the activity of programming (Initiative D21, 2021). Therefore, the idea of using natural language (e.g. English or German language) which is then automatically translated into computer code is quite old (Price et al., 2000; Begel, 2004). To improve these approaches, it would be useful to have datasets with tasks described in natural language and different valid computer programs which solve the task.

Motivated by this, we created a new dataset consisting of student solutions to natural language task descriptions. With this dataset, we hope to enable future research on learning programming, algorithms and data structures both for students and in machine learning contexts. The dataset is available under the permissive CC BY-NC 4.0 license.[1]

The dataset contains algorithmic programming assignments formulated originally in German together with accompanying test cases and currently 1500+ source code solutions in Java and Python by students participating in Bachelor-level *algorithm and data structures* courses at Universität Hamburg. Only correct solutions from students who consented on both collection and distribution are included in the published dataset. To make the dataset more easily accessible to others, we also provide carefully translated task descriptions in English. One example of such a task can be seen in Figure 1.

> Implement a class ”Queue” that works like a queue (as described in the lecture). The class should have at least the methods isEmpty(), head(), enqueue(x) and dequeue(). The queue does not need to hold more than 100 elements.
> Note 1: In the Java test, the stack is expected to store ”strings”. Since Python is dynamically typed, this does not apply.
> Note 2: It is not allowed to use ”import”!

Figure 1: Example task description

The task description further contains some usage examples and a skeleton of the class with empty method bodies. For brevity reasons, these can only be found in the dataset but are not shown here. A random, very concise Python solution to this task is shown in Figure 2.

As the tasks have an open solution space, the answers vary substantially in length, style, usage of comments as well as naming of variables and possible helper functions. To check the answers for correctness, each exercise has a number of test cases that a solution must pass. Table 1 shows a few of the test cases used to verify the Python solutions for the example task from Figure 1.

[1] `https://www.inf.uni-hamburg.de/en/inst/ab/lt/resources/data/ad-lrec`

Listing 1: A student's Python solution

```python
class Queue:
  def __init__(self):
    self.array = []
  def isEmpty(self):
    return len(self.array) == 0
  def head(self):
    return self.array[0]
  def enqueue(self, x):
    self.array.append(x)
  def dequeue(self):
    return self.array.pop(0)
```

Figure 2: Example task description

Test case	Correct output

`q = Queue()`
`print(q.isEmpty())`
	True

`q = Queue()`
`for i in range(1,101):`
`  q.enqueue(i)`
`for i in range(40):`
`  q.dequeue()`
`print(q.head())`
	41

`q = Queue()`
`q.enqueue("Kakao")`
`print(q.isEmpty())`
	False

Table 1: 3 out of 10 Python test cases for the task

In our previous work, we describe our lessons learned during the introduction of automatically assessed programming exercises to the algorithms and data structures course (Soll et al., 2020). In that work, we conducted a user study with 44 participants showing that the introduction of the new exercises was generally perceived well by the students, although improvements (e.g. better feedback to students) were still possible.

In this paper, we make the following contributions:

- a new dataset containing natural language task descriptions to source code with test cases and student solutions in two programming languages,
- details on the creation of the dataset,
- analysis of the dataset.

As compared to other similar datasets, the key differences and novelties of our dataset are the focus on more complex tasks from the domain of algorithms and data structures, tasks requiring to implement data structures, solutions without library usage and problem descriptions referring to course material.

2. Related work

Shin and Nam (2021) provide an extensive overview of the field of automatic code generation from natural language. They categorize approaches by their code, input and output form, e.g., systems transforming line-by-line natural language instructions into runnable soure code, systems translating abstract natural language descriptions into code snippets or when also given unit tests into fully runnable source code.

Oda et al. (2015) propose a method to automatically generate pseudo-code from source code using an adapted statistical machine translation approach. They provide the *Django* dataset consisting of pseudo code and source code pairs.

Lin et al. (2018) created *NL2Bash*, a parallel corpus of natural language instructions and Linux shell commands. The natural language instructions are short and usually consist of a single sentence. The corpus only includes shell commands that are formulated in a single line.

Based on *NL2Bash*, Agarwal et al. (2021) created a new dataset for the *NLC2CMD* competition by incorporating approximately 800 additional pairs of natural language to Linux shell commands collected from two other sources: The *Tellina* query log and the *NLC2CMD* data collection track. For use in the competition, the dataset is filtered by checking the validity of the Linux shell commands with a Bash parser.

Yin et al. (2018) created the *CoNaLa* dataset, a fine-grained, parallel corpus of natural language question/instruction and source code. They crawled data from *Stack Overflow*, filtered the pairs automatically and finally curated them manually via crowdsourcing.

Yao et al. (2018) created another dataset of question-code pairs from *Stack Overflow* named *StaQC*. Trough crawling and filtering with a neural network they produce a large collection of Python and SQL code snippets that are a standalone solution to the question.

Husain et al. (2019) created the *CodeSearchNet* challenge dataset, a parallel corpus of comment-code pairs extracted from open source libraries in six programming languages. The dataset is designed to train and evaluated systems retrieving relevant source code given a natural language query.

The *JuICe* dataset is large distantly supervised dataset for open domain context-based code generation based on online programming exercises (Agashe et al., 2019). Its test set further provides refined human-curated data. A large fraction of the natural language instructions is written in high-level, declarative style. The dataset mainly consists of code for real world applications, primarily for data science and machine learning.

Hendrycks et al. (2021) created the *Automated Programming Progress Standard (APPS)* benchmark dataset, a collection of 10,000 natural language coding problems with 131,777 test cases for checking solutions and 232,421 ground-truth solutions written by humans. In contrast to the previous datasets, the problem statements are considerably more complex with an average length of 293.2 words in natural language and are categorized into three levels of difficulty. The problems were crawled, deduplicated and manually curated from open-access coding sites. Hendrycks et al. (2021) further trained multiple GPT models to automatically generate solutions for these programming assignments and found GPT-Neo be able to solve 5% of the introductory-level problems.

Atzeni and Atzori (2018) work on the translation of natural language to source code using an unsupervised ontology-based approach. They describe a semantic approach to translate primarily mathematical, single-sentence natural language commands and questions into short Java code.

Spacco et al. (2015) perform an analysis of data collected by the CloudCoder programming exercise system from introductory courses taught in two programming languages. While they demonstrate the research potential of collecting and analyzing data from programming exercise, the data used in their analysis is not publicly available. Wang et al. (2017) use data from *Hour of Code* course exercise 18 to perform "knowledge tracing" by modeling a student's learning through analysis of the answers to the programming exercises.

3. Construction of the dataset

In this section, we describe how the dataset has been created. First, we outline why and how we added automatically graded programming exercises to the e-learning platform and briefly discuss the challenges we encountered. Second, we describe how we collected the data and what measures we applied to enable the distribution of student solutions under the data protection laws.

3.1. Challenges of creating automatically graded programming exercises

The Department of Informatics at Universität Hamburg uses *Moodle*[2] as a digital learning platform. In the algorithms and data structures course, *Moodle* has already been used in the past to organize the course work, provide a platform for communication within the course, and supply both lecture materials and manually-graded non-programming exercises.

Starting with the winter semester 2019/2020, programming exercises were added to the course in order to facilitate the students' understanding of the various algorithms and data structures by implementing a selection of them in actual code. In other courses offered by the Department of Informatics, automatically assessed tests had already been employed for multiple-choice questions and similar questions with a very limited solution space. Compared to those prior exercises, the integration of programming exercises presented a number of unique challenges.

While manual review of submissions by tutors is certainly possible, it is infeasible to do so for hundreds of source code solutions with only a handful of tutors. This issue was aggravated by the need for immediate feedback while a student is working on a task. Thus, we settled on a fully automatic, instantaneous assessment using unit tests.

We used the *CodeRunner* plug-in (Lobb and Harlow, 2016) to provide an interactive coding environment within *Moodle*. Thereby, a student was able to test a potential solution against a number of test cases defined for each exercise. A solution was seen as correct when it passed all test cases of the exercise, a subset of which were visible to the student. The non-visible set of test cases featured randomized inputs to prevent hard-coded solutions as well as test cases to prevent students from using library functions. Consequently, students mostly wrote the code for the algorithms and data structures from scratch as intended, although copying existing code from external sources could not feasibly be prevented.

For more details on the setup (e.g. design and deployment of the system) and our experiences, refer to our previous work (Soll et al., 2020).

3.2. Collecting the data

As task descriptions, test cases and student solutions are created and edited within the *CodeRunner* plug-in for *Moodle*, the e-learning platform is the single source of truth for the data. Since we collected task descriptions, test cases and student solutions after a course ended, we were able to ensure that they are in a consistent state, i.e., only the most recent, final version is used for the dataset. This is necessary because the task descriptions and/or some test cases were occasionally revised during the course following input from the students. Consequently, the students were allowed to revise and re-submit their solutions as well.

We included only those solutions that are considered correct in the context of the exercise, i.e., were graded with the maximum number of points achievable for each exercise.

Since the students produced their solutions for the exercises during a course at Universität Hamburg, the principle of data economy (data protection law in the EU) states that, without explicit consent, the students' data may only be used for the purpose they were collected for. That is, to ascertain passing the course and to archive all grading-related data as strictly regulated by law. To use student solutions for other purposes, e.g., research, we asked each student for permission during the course. The students could freely choose whether they wanted to allow the usage and redistribution of their solutions for research purposes. Each student's name was replaced by a random identifier to allow tracing the solutions of a student across different exercises. It was explicitly made clear that their progression and grading of the course is completely unaffected by their choice. Thus, students had to explicitly opt-in for their solutions to be included.

[2] Moodle is a free and open-source e-learning platform extensible with plugins. `https://moodle.org`

We converted the task descriptions to both PDF (via LaTeX) and plain text with LaTeX-math only for formulas. As the task descriptions are written in German for the course, we additionally provide a carefully translated English version for better accessibility and application of English NLP methods resp. models. The test cases were exported alongside the task descriptions.

3.3. License

The dataset is licensed under the Creative Commons Attribution-NonCommercial 4.0 International License. This allows an easy sharing and redistribution of both the original data as well as other datasets building on top of it while only requiring attribution and forbidding commercial usage (as the students were only asked for permission to use their solutions in a non-commercial way, like scientific usage).[3]

4. Dataset exploration

The dataset contains programming exercises for Java and Python from the domain of algorithms and data structures consisting of German and English task descriptions, test cases and student source code solutions for both programming languages. Currently, it contains 21 tasks, 533 test cases (almost 13 on average per task and programming language) as well as 1500+ source code solutions from students who consented on both collection and distribution of their solutions.

4.1. Dataset overview

The dataset consists of three CSV files: Task descriptions, test cases and student solutions. The first file lists each task description with its identifier, original German title, translated English title, plain text German task description, translated English task description, solution code skeleton, shared code for the test cases and a solution.

The task identifier is built from the semester, the exercise block, the exercise number and programming language. An example is `19_20-4-2-java`; a task from the course in the winter semester 2019/20, exercise block 4, exercise number 2 within the block and Java as programming language. In the *Moodle* courses, topically related exercises were grouped together in an exercise block.

Plain text task descriptions in both German and English still contain LaTeX math formulas enclosed in $ signs. As these formulas are often essential to solve a task, we decided to keep them as is. The solution code skeleton was shown to the students to use as basis for their custom solution. These code fragments usually define the basic parts, i.e., names of functions or classes, to be compatible with the automated tests. The tester is shared code that is required to run the test cases. It provides classes and methods used in the individual test cases to make them concise.

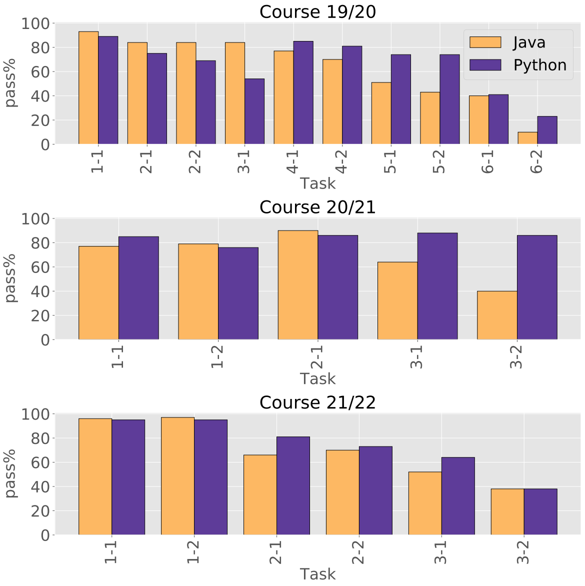

Figure 3: Percentage of correct solutions from the students who consented to collection

The test cases for all assignments are listed in the second file with the task identifier to link a test case to the corresponding task description. For each task, the test cases are numbered and apart from the test code and expected output also feature a flag indicating whether the test case is an example, i.e., was shown to the students. Since the test cases originate from *Moodle*, they differ from typical Java or Python unit tests. Instead of using asserts, the test cases produce an output that has to be compared externally with the expected output.

The third file contains one solution together with the task identifier and randomized student identifier. As students could freely choose which programming language to use for each exercise, students could to do some task in both languages, some task in Java, the next in Python or not all. Thus, the number of solutions per exercise can vary greatly.

4.2. Dataset analysis

In this section, we describe the results of our quantitative analysis of the dataset and give insights into certain anomalies we found that can only be explained with knowledge of the courses. Table 2 gives an overview of the dataset by comparing the data of the three years currently contained in the dataset. It gives the number of exercises in each year, the number of students that consented to collection of their solutions, the number of total and correct solutions of these students as well as the total number of test cases. The table also displays the average length of all task descriptions (word count) as well as the average length of student solutions by lines of code (LOC), including comment-only lines but excluding empty (whitespace-only) lines, both global average and average by programming language.

[3] More detailed and precise legal information can be found at `http://creativecommons.org/licenses/by-nc/4.0/` and in the license file accompanying the dataset

Course	19/20	20/21	21/22
Exercises	10	5	6
Students	85	91	128
Correct solutions (abs.)	541	415	570
Correct solutions (rel. %)	68.5	75.0	73.3
Test cases	241	142	150
Avg. task descr. length	122.7	200.4	201.0
Avg. LOC	25.3	21.8	16.6
Avg. LOC (Java)	28.8	26.1	20.0
Avg. LOC (Python)	19.7	17.7	12.7

Table 2: Dataset statistics overview for the courses in 2019/20, 2020/21 and 2021/22. Task description length measured in words. LOC is short for lines of code.

When comparing these statistics for the three courses a few aspects stand out. While the number of exercises was cut in half from the 2019/20 course to the 2020/21 course, the average exercise description length increased by over 60% and stayed the same in the most recent course. This could be an indicator that the exercises of the 2020/21 and 2021/22 courses were fewer in number but more challenging. When creating the newer exercises, we wanted to keep the difficulty roughly at the same level. Since the percentage of correct solutions increased slightly over the 2019/20 course, a more likely reason is that the exercise descriptions in the later courses are more detailed and provide more context information. The average solution length decreased slightly for both Java and Python from the first to the second course. In the most recent course, the average solution lengths decreased stronger. Combined, it decreased even more because at the same time more students used the more concise Python language in the more recent courses than before. The number of test cases per task and programming language remained similar, increasing on average from 12.1 in the first course to 12.8 in the second course and decreasing slightly to 12.5 in the third course.

Figure 3 shows the relative number of correct solutions for each exercise in the dataset. In the 2019/20 course, the Java solutions exhibit a strong decline from more than 80% in the first four exercises down to mere 10% in the last exercise. While Python has a lower ratio of correct solutions for the first four exercises, it is the opposite during the remaining six exercises. Still, the last two exercises reach only 41% resp. 23% of correct solutions. We identified four main reasons for low numbers on the two exercises in the last exercise block. First, the students were only required to pass one of both exercises to achieve full points in the exercise block. Thus, they likely looked at both exercises but only put effort into solving one of them. Second, most students did simply not require the points from the last exercises to pass the course as they already collected enough points during the other exercises. Thirdly, except for having a passing grade on a certain number of programming tasks to pass the course, there was no further organizational incentive for students to interact with the tasks after a certain point. Consequently, some likely did not see a point in putting more time and thought into their last solutions and instead focused on the theoretical exercises for the exam. The programming exercises and the theoretical exercises had to be passed individually to pass the course. Lastly, the exercises were set to auto-submit at due date such that all students who opened an exercise for any reason, even if only for looking at the task, had submitted a solution, which then probably would not have been correct. We were able to observe a large number of students who looked at an exercise in both programming languages but worked on it only in one. Thus, the pass%-value especially of the later tasks of the course of 2019/20 is probably skewed due to a large number of “empty” responses.

In the 2020/21 course, the ratio of correct solutions is rather constant at approximately 80% across all five exercises. Only for the last two exercises (belonging to one exercise block) the percentage of correct Java solutions decreases. In contrast to the previous course, the theoretical assignments and programming exercises had to be passed as a whole. As such, students could compensate their performance on the theoretical assignments with the programming exercises and vice versa. However, the points to be obtained from the programming exercises only account for roughly 10% of the total points in the course. The exercises were primarily intended to motivate students by applying the theoretical concepts in actual code and to get a better understanding of the algorithms and data structures by implementing them.

In the 2021/22 course, the ratio of correct solutions decreases continuously from almost 100% in the first two tasks to less than 40% in the last task. The tasks are generally becoming more difficult throughout the course. While the first tasks require implementations of algorithms such as exponentiation by squaring, least common multiple or edit distance, later tasks require writing and applying data structures such as queue, stack or search trees. As in the previous course, the theoretical assignments and programming exercises had to be passed as a whole. Again, the programming assignments were not meant to stop anyone from passing the course, but to help understand algorithms and data structures.

Table 3 goes into more detail than Table 2, separating the information not only by year but also by exercise and programming language. The dataset only contains the solutions that passed all tests, i.e., were seen as correct, for a total of 1526 entries at the time of writing.

We further encountered some special cases in the exercises that may not be evident from the statistics but that might be important for any approach to automatically translate the exercise descriptions into source code. In the 2019/20 course, the exercises in the fifths block built on each other. While exercise `5-1` describes context (graphs), problem and required solution (breadth-first search) in great detail, exercise `5-2` builds on it and

				correct solutions		
ex.	lang.	WC	tests	abs.	rel. %	LOC
			Course 19/20			
1-1	Java	93	5	51	93	16.9
1-1	Python	93	5	31	89	9.7
2-1	Java	68	11	49	84	35.1
2-2	Java	241	13	49	84	32.3
2-1	Python	68	10	27	75	31.6
2-2	Python	231	13	25	69	20.4
3-1	Java	121	11	48	84	17.8
3-1	Python	121	11	19	54	13.3
4-1	Java	105	8	41	77	28.9
4-2	Java	98	10	37	70	46.1
4-1	Python	98	8	23	85	18.9
4-2	Python	91	10	22	81	37.5
5-1	Java	205	11	24	51	26.6
5-2	Java	45	11	20	43	26.2
5-1	Python	191	11	23	74	14.2
5-2	Python	32	11	23	74	14.0
6-1	Java	65	21	12	40	37.7
6-2	Java	211	20	3	10	30.7
6-1	Python	65	21	9	41	17.9
6-2	Python	211	20	5	23	19.6
			Course 20/21			
1-1	Java	160	6	48	77	15.7
1-2	Java	148	6	49	79	40.0
1-1	Python	160	20	46	85	14.3
1-2	Python	149	11	41	76	24.9
2-1	Java	323	17	47	90	16.8
2-1	Python	288	17	40	86	11.8
3-1	Java	220	21	35	64	25.2
3-2	Java	166	10	22	40	39.6
3-1	Python	220	21	49	88	16.2
3-2	Python	170	13	38	68	21.9
			Course 21/22			
1-1	Java	142	15	71	96	17.0
1-2	Java	157	17	72	97	20.3
1-1	Python	149	17	56	95	12.5
1-2	Python	146	18	56	95	12.6
2-1	Java	324	17	52	66	20.4
2-2	Java	212	5	55	70	17.7
2-1	Python	289	17	54	81	11.7
2-2	Python	235	7	49	73	11.7
3-1	Java	162	7	31	52	18.5
3-2	Java	215	10	23	38	34.3
3-1	Python	162	7	32	64	10.6
3-2	Python	219	13	19	38	22.5

Table 3: Exercise statistics for the three courses showing for each exercise (ex.) and programming language (lang.) the word count (WC) of the task description, the number of test cases, number of solutions that passed all test cases (abs.), their percentage of all submitted solutions (rel.%) and the average lines of code (LOC) of the solutions in the dataset. All solution statistics were calculated from the subset of solutions where students consented on the scientific usage of their data.

only asks to implement a similar algorithm (depths-first search). Tasks `19_20-5-1`, `20_21-3-2` and `21_22-3-2` further use an image depicting an example graph. In the 2020/21 and 2021/22 courses, most exercises explicitly state a maximal run time for the solution both in big-$\mathcal{O}$ notation and as a processing time limit in seconds. These time limits have been enforced by the *CodeRunner* plugin and are thus not automatically enforced in the extracted test cases.

5. Conclusion and future directions

In this paper we presented a new dataset containing 21 algorithmic programming tasks (accompanied by test cases) and a total of 1500+ solutions created by computer science students in a Bachelor course at Universität Hamburg. The dataset is made available under the *Creative CommonsAttribution-NonCommercial 4.0 International License*.

The data was collected during the winter semesters of 2019/2020, 2020/2021 and 2021/2022 by exporting submitted solutions from *Moodle* / *Coderunner* for all students who gave their consent. Together with the dataset we presented some basic statistical data as well as informed about noteworthy observations concerning the development of the solution set over the semester.

For the time being, we plan to continually add more exercises with corresponding test cases and students solutions from future courses to the dataset. We hope that by expanding the dataset it will become even more useful in future research. Possible future experiments include the application of state-of-the-art machine learning methods to automatically translate natural language to source code.

6. Acknowledgements

We would like to thank all students who have participated in the courses and agreed to distribute and use their assignment solutions for scientific purposes. Additionally, we would like to thank Matin Urdu and Ahmad Shallouf for integrating further programming assignments for the courses during the winter semesters 2020/2021 and 2021/2022.

Marcus Soll, Louis Kobras and Melf Johannsen were funded by MINTFIT Hamburg[4]. MINTFIT Hamburg is a joint project of the four STEM universities in Hamburg: Hamburg University of Applied Sciences (HAW), HafenCity University Hamburg (HCU), Hamburg University of Technology (TUHH), University Medical Center Hamburg-Eppendorf (UKE) as well as Universität Hamburg (UHH) and is funded by the Hamburg Authority for Science, Research and Gender Equality[5].

Fynn Petersen-Frey was partly supported by the Cluster of Excellence CLICCS (EXC 2037), Universität Hamburg, funded through the German Research Foundation (DFG).

[4] https://www.mintfit.hamburg/en

[5] https://www.hamburg.de/bwfg

7. Bibliographical References

Agarwal, M., Chakraborti, T., Fu, Q., Gros, D., Lin, X. V., Maene, J., Talamadupula, K., Teng, Z., and White, J. (2021). Neurips 2020 nlc2cmd competition: Translating natural language to bash commands. In Hugo Jair Escalante et al., editors, *Proceedings of the NeurIPS 2020 Competition and Demonstration Track*, volume 133 of *Proceedings of Machine Learning Research*, pages 302–324. PMLR.

Agashe, R., Iyer, S., and Zettlemoyer, L. (2019). JuICe: A large scale distantly supervised dataset for open domain context-based code generation. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 5436–5446, Hong Kong, China. Association for Computational Linguistics.

Atzeni, M. and Atzori, M. (2018). Translating natural language to code: An unsupervised ontology-based approach. In *2018 IEEE First International Conference on Artificial Intelligence and Knowledge Engineering (AIKE)*, pages 1–8.

Begel, A. (2004). Spoken Language Support for Software Development. In *2004 IEEE Symposium on Visual Languages - Human Centric Computing*, page 271–272.

Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D., and Steinhardt, J. (2021). Measuring coding challenge competence with APPS. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*.

Husain, H., Wu, H., Gazit, T., Allamanis, M., and Brockschmidt, M. (2019). Codesearchnet challenge: Evaluating the state of semantic code search. *CoRR*, abs/1909.09436.

Initiative D21. (2021). Digital Skills Gap - So (unterschiedlich) digital kompetent ist die deutsche Bevölkerung. `https://initiatived21.de/app/uploads/2021/08/digital-skills-gap_so-unterschiedlich-digital-kompetent-ist-die-deutsche-bevlkerung.pdf`.

Körber-Stiftung. (2017). Programmieren eröffnet Welten. *Körber-Stiftung Pressemeldungen*.

Lin, X. V., Wang, C., Zettlemoyer, L., and Ernst, M. D. (2018). NL2Bash: A corpus and semantic parser for natural language interface to the linux operating system. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, Miyazaki, Japan. European Language Resources Association (ELRA).

Lobb, R. and Harlow, J. (2016). Coderunner: A tool for assessing computer programming skills. *ACM Inroads*, 7(1):47–51.

Oda, Y., Fudaba, H., Neubig, G., Hata, H., Sakti, S., Toda, T., and Nakamura, S. (2015). Learning to generate pseudo-code from source code using statistical machine translation. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 574–584.

Price, D., Riloff f, E., Zachary, J., and Harvey, B. (2000). NaturalJava: A Natural Language Interface for Programming in Java. In *Proceedings of the 5th International Conference on Intelligent User Interfaces*, IUI ’00, page 207–211, New York, NY, USA. Association for Computing Machinery.

Shin, J. and Nam, J. (2021). A survey of automatic code generation from natural language. *J. Inf. Process. Syst.*, 17(3):537–555.

Soll, M., Kobras, L., Johannsen, M., and Biemann, C. (2020). Enhancing a theory-focused course through the introduction of automatically assessed programming exercises - lessons learned. In Tom Broos et al., editors, *Proceedings of the Impact Papers at EC-TEL 2020, co-located with the 15th European Conference on Technology-Enhanced Learning ”Addressing global challenges and quality education” (EC-TEL 2020), Virtual, September 14-18, 2020*, volume 2676 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Spacco, J., Denny, P., Richards, B., Babcock, D., Hovemeyer, D., Moscola, J., and Duvall, R. (2015). Analyzing student work patterns using programming exercise data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE ’15, page 18–23, New York, NY, USA. Association for Computing Machinery.

Wang, L., Sy, A., Liu, L., and Piech, C. (2017). Deep knowledge tracing on programming exercises. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale*, L@S ’17, page 201–204, New York, NY, USA. Association for Computing Machinery.

Yao, Z., Weld, D. S., Chen, W., and Sun, H. (2018). Staqc: A systematically mined question-code dataset from stack overflow. In Pierre-Antoine Champin, et al., editors, *Proceedings of the 2018 World Wide Web Conference on World Wide Web, WWW 2018, Lyon, France, April 23-27, 2018*, pages 1693–1703. ACM.

Yin, P., Deng, B., Chen, E., Vasilescu, B., and Neubig, G. (2018). Learning to mine aligned code and natural language pairs from stack overflow. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR ’18, page 476–486, New York, NY, USA. Association for Computing Machinery.