Using LLM Chatbots to Improve the Learning Experience in Functional Programming Courses

JULIAN VAN SANTEN, University of Twente, The Netherlands

Large Language Model chatbots are used in many different disciplines, but applications in education stay behind. Students increasingly utilize the LLM chatbots, without proper supervision from teachers. Functional programming requires a problem-solving mindset, which can be difficult for students just learning the paradigm. By finding out which learning objectives students find the most difficult, we aim to find new ways to help students learn functional programming using LLM chatbots like ChatGPT. We demonstrate how LLM chatbots can be restricted by teachers, to provide a helpful tool for students learning functional programming and its concepts.

CCS Concepts: • **Social and professional topics → Computer science education**.

Additional Key Words and Phrases: Large Language Models, code explanation, functional programming, computer science education

1 INTRODUCTION

Large Language Models are changing the way we search for information. ChatGPT is being used to find information, assist in writing in a broad sense, solve problems in programming. Kasneci *et al.* [14] describe the benefits and challenges of using AI in education. They mention the chance for assistance for university students in the area of problem-solving: a crucial skill when one is programming in a functional language. They conclude with the statement that the risks are manageable, and more research should be conducted to find best practices in the branch of education.

We apply this educational strategy to functional programming in particular. Hughes [13] showed the importance of the functional programming discipline to modularize software, which he deemed an important factor in software development today. By teaching functional programming, students learn about the concepts necessary to design software in a modular fashion. This approach makes it possible to easily parallelize software components [11].

To create an environment where students can be hinted towards valid and qualitative solutions, we first analyze which learning goals and objectives are experienced as most difficult by students who have no experience in the functional programming paradigm. In section 4, we analyze introductory functional programming exams and find the difficulties in the tested learning objectives.

After identifying the learning objectives that have a lower average score than their counterparts, we look at connections among these learning objectives, the concepts they stem from and the related but different material offered during previous courses. We look at the connections of functional programming to mathematics, imperative programming and other functional programming concepts. This analysis is presented in section 5.

TScIT 40, February 2, 2024, Enschede, NL

Finally, these previous findings are utilized to test prompt programs. Recent literature in the field of *prompt engineering* provides abstract patterns to be used in different applications, which are used to create three applied prompts for the functional programming use case. The applied prompts are the first prompt programs written for the education of functional programming. The performance of each prompt is measured against metrics defined in section 3. The results of the prompts are presented in section 6.

2 RELATED WORK

Although not much ink has been put to paper on using AI in the context of learning functional programming in a university course, related studies did look at other programming languages and paradigms in different ways. Since the mainstream release of ChatGPT, work has been done and is still being done in the applications of Large Language Model tools. Research done in this field of LLMs is recent, spanning only this decade, with most papers published in 2023. Most papers presented exhibit a cumulative progression, each building upon the findings of the preceding ones.

2.1 Language Model Reasoning and Prompt Engineering

After Brown *et al.* [3] published their findings on *Few-Shot* learning of Large Language Models, a hunt for better-performing approaches started. Kojima *et al.* [17] showed an interesting insight: adding the line "Let's think step by step" at the end of a single prompt resulted in far more correct results than without this prompt. This *Zero-Shot Chain of Thought* approach was one of the first examples of *prompt engineering*, which is developed further in later papers.

The Zero-Shot Chain of Thought approach was extended by Wang *et al.* [25]. They introduced a *Plan-and-Solve* (PS) strategy: the phrase "*Let's think step by step*" is replaced with "*Let's first understand the problem and devise a plan to solve the problem. Then, let's carry out the plan and solve the problem step by step*". The method was tested on the `text-davinci-003` model, and all but the *MultiArith* questions dataset scored better with their *PS+* (more specified prompt) than regular CoT. Just using the *PS* approach mentioned before also resulted in higher scores in most question datasets. By asking for more elaboration, the quality of LLM output can be improved. This notion is used in the development of the presented prompt programs.

As more and more discoveries are made in prompt engineering, multiple approaches have been found to work for various purposes. White *et al.* [27] have written a catalog with prompt patterns, generalizing some of the work done by the previously mentioned authors. This catalog contains useful patterns, which have been used to create more specific prompts for the use case of functional programming education. The use of the *Persona* pattern introduced in this framework is of particular interest. By adopting the fictional role of a functional programming instructor, the LLM chatbot is equipped to provide more accurate and relevant responses to student inquiries. The *Question Refinement* pattern and the *Flipped Interaction* pattern are also useful. These patterns return questions to the user, thereby fostering the development of advanced questioning techniques and encouraging users to contemplate potential solutions independently.

2.2 Programming Education

Work has been done measuring the usefulness of AI assistants such as ChatGPT in computer science education. Wang *et al.* [26] took 187 problems from many computer science topics and asked ChatGPT to explain the answers. There is room for applying similar strategies in the area of functional programming, as none of these subjects tested the AI assistants for functional programming questions and patterns.

Keuning *et al.* [15] looked at ways teachers could help students improve code *quality*. They asked 30 teachers of Computer Science related courses about the involvement of code quality in their teachings. Furthermore, they gave the teachers code samples of popular answers from students to programming questions. Out of these code samples, teachers then gave various hints to improve the code. Although these hints are applied to a Java-like language, some code quality hints could be transferred to a functional language. The interaction study has served as an inspiration to create student-friendly prompt programs.

The same group of Keuning *et al.* [16] provided another contribution during the same conference, in which they analyzed automatic feedback generation tools for programming exercises. Although no AI-based tools are analyzed, some do give elaborate feedback. High-quality responses have been taken as inspiration for testing the output of ChatGPT.

Gerdes *et al.* [10] have created Ask-Elle, a tool for students who are learning Haskell. The tool provides an online environment, in which teachers can define exercises, along with hints. It can analyze incomplete Haskell code to provide the given hints, thus making it context-aware. The hints generated by Ask-Elle have to be entered in by teachers, thus it becomes more labor-intensive for teachers to create the environment in Ask-Elle. It also means that as the exercises grow in size, more hints have to be written. Letting a Large Language Model analyze incomplete code can alleviate work for teachers, as there will be no need for teachers to define hints. It is relevant to see how students interact with Ask-Elle, as it tells how students prefer to receive their feedback.

3 METHODOLOGY AND APPROACH

The statistics of the results from introductory functional programming tests were analyzed. The data was gathered from a first-year course. The exam had answers to multiple-choice questions, which were statistically retrieved. The exam data comes from exams made between 2019 to 2022. The questions test the knowledge of the student on Functional Programming concepts and Haskell notation based on a one-week introduction course.

The questions asked on each available exam primarily test one learning objective. A few questions examine multiple learning goals, which have been scrapped from the dataset for the sake of clarity in the data. The data presented in section 4 show the results of the exam questions grouped by category.

3.1 What do the difficult learning objectives have in common?

To provide useful feedback for students who are starting with functional programming, an analysis of the concepts they learn is presented. This is done by looking at both the mathematical and programming history of the concepts: where do they originate from and which language was the first to introduce an implementation of it? Furthermore, this analysis aims to ascertain the reasons for its introduction in these preceding languages. Lastly, the analysis focuses on identifying the features associated with functional methods in the languages where they were introduced. These findings are then connected to the assumed background of students participating in introductory functional programming courses.

3.2 Creating prompts for ChatGPT

By utilizing the prompt engineering patterns written by White *et al.* [27], novel prompt preambles are constructed and tested for ChatGPT with GPT-4. The exams analyzed in section 4 offer insight into the results of different learning objectives during the introductory functional programming course. These results are used, together with the connections among the concepts discussed in section 5, to form a framework inside ChatGPT, which can be used by future students in functional programming courses.

To measure the quality of content generated by ChatGPT, quantitative and qualitative analyses are both performed. Introductory functional programming questions are fed into ChatGPT, along with a preamble under test. Each preamble gets the same set of programming questions. The responses are analyzed according to three quantitative metrics:

(1) Does the response contain any code?[1]
(2) Does the response adhere to all tasks given by the prompt?
(3) Does the response answer the question verbatim?

Furthermore, the groups of questions are qualitatively analyzed. In this analysis, the following aspects are taken into account:

- Does the response contain any incorrect information?
- Does the response contain relevant information per the input question?
- Does the response cater to functional programming specifically (e.g., are there no strictly imperative hints given)?

4 ANALYSIS OF INTRODUCTORY FUNCTIONAL PROGRAMMING EXAMS

The introductory Functional Programming course exams focus on several different learning objectives. The learning objectives, along with their labels, can be found in Table 1. These labels correspond to a (part of) a concept of functional programming, or Haskell features and notation specifically. Statistics of questions from the 2019 test, as well as the tests ranging from 2021 to 2023, were categorized. Exam questions created in 2019 were already labeled by the teacher upon creation, while later exam questions were manually labeled before the analysis. Some categories deal with subsections of functional programming concepts, hence some categories are connected. As the

[1]This only includes code blocks, inline code fragments are ignored

Table 1. Exam question categories

	List comprehension
Abbreviation	**Meaning**
APP	Function application / currying, arguments
COND	Conditional logic (guards/if-then-else)
FARG	Functions as arguments (writing higher-order functions)
HFUN	Using higher-order functions
INFL	Infinite lists / lazy evaluation
LCFT	List comprehension with predicates
LCGN	List comprehension general (without predicates)
LPAT	List pattern matching
LSTC	List constructors
LSTT	List types
RECL	Recursive functions with lists
RECV	Recursive functions with general variables
RFUN	Recursive functions
TPLT	Tuple/list theory
TYPE	Function type definition
VPAT	Variable pattern matching
QUIC	QuickCheck question
SYN	Haskell syntax

Table 2. Computed averages per category, spanning all exams

Category	$\sum_q$	$\sum_a$	p'	$\sigma_{p'}$	r_{it}	r_{ir}
APP	8	2304	0.642	0.221	0.407	0.327
COND	10	2845	0.773	0.168	0.332	0.260
FARG	8	2303	0.707	0.094	0.467	0.389
HFUN	11	3179	0.651	0.092	0.437	0.353
INFL	9	2637	0.720	0.160	0.384	0.304
LCFT	13	3694	0.841	0.080	0.420	0.356
LCGN	3	975	0.645	0.202	0.380	0.299
LPAT	5	1679	0.852	0.100	0.397	0.338
LSTC	5	1679	0.553	0.151	0.377	0.289
LSTT	5	1391	0.364	0.281	0.183	0.097
RECL	5	1616	0.739	0.120	0.522	0.455
RECV	7	2158	0.866	0.103	0.344	0.286
RFUN	8	2303	0.778	0.080	0.415	0.341
TPLT	3	912	0.824	0.013	0.373	0.301
TYPE	10	2908	0.792	0.137	0.337	0.265
VPAT	4	1408	0.685	0.158	0.434	0.358
QUIC	2	479	0.721	0.061	0.305	0.211
SYN	2	479	0.926	0.078	0.254	0.204

questions of the tests are still used in the course, the exact questions and their answers will not be published.

Statistics from the tests, such as the average p'- and r_{ir}-value are shown in Table 2. The p'-value denotes the ratio between the average accomplished score and the maximum achievable score, with correction for guessing. This notation offers a more balanced view than the p-value (Dousma *et al.* [8, p. 128]).

4.1 Lists and list comprehension

Table 2 shows that questions asking about list constructors and types score the lowest on average. The *list type* (LSTT) questions exhibit a notably low mean score, yet they also display a comparatively high variance in the questions' p'-values. The average of *list constructor* (LSTC) questions is higher than the LSTT questions but is still low compared to other categories. Furthermore, the average r_{ir}-value is also more in line with the other categories, as opposed to the r_{ir}-value of the LSTT category.

Multiple explanations can be made for these values, such as unclear or missing explanations by the teacher, a lack of practice material, confusing wording in the questions, or another unforeseen issue with either the question or the learning objective. Subsection 5.1 delves into the analysis of list notation and the interrelationship among mathematical concepts taught throughout the course.

An example of a list comprehension without predicates presented in exam questions can be found in Listing 1. This question examines whether participants understand the generation of combinations within a list comprehension. When asked the length of lcomp, participants must see that the resulting list will have **length** xs * **length** ys items. The incorrect option **length** xs may be chosen as an answer instead. The fact that variable y is not used in the final result may hint at excluding the entries of list ys altogether, but this is false. In subsection 5.2, the origins of list comprehension as a programming concept are discussed, together with the connection to set (builder) theory. When asking questions to a chatbot, it is important to utilize this detail to produce valuable guidance on this topic.

```
lcomp :: [a] -> [a] -> [a]
lcomp xs ys = [ x | x < xs, y <- ys ]
```

Listing 1. Example of list comprehension question context

4.2 Partial application, currying and higher-order functions

Questions in the category of *function application and currying* (APP) score below the average p' of 0.726. The high deviation between these results means that not every question scores this low, but the category average is still noteworthy. The relatively high average r_{ir}-value suggests that the questions regarding function application do not discriminate heavily between participants who have answered correctly out of knowledge and those who have guessed the answer correctly by accident. All these statements together suggest that the questions are not well made. The concept of function application and currying is explained in subsection 5.3.

In essence, questions about function application aim to assess the student's understanding of employing constructs like tuples, as opposed to utilizing multiple arguments. An example of a function that can be partially applied can be seen in Listing 2. The goal of the question is to test if the participant understands that the function takes in a *tuple* of values. Thus, the attempted evaluation f 1 2 will result in a function type error.

```
f (x,y) = x + y
```

Listing 2. Example of function application question context

Currying questions look alike but instead use multiple arguments. An example of a function that questions the knowledge of currying can be seen in Listing 3. The participant will have to recognize the validity of a function call such as `(g 1) 1`. As opposed to Listing 2, It is important to note that the types of the functions in function application/currying questions are left out. Exam participants are required to comprehend the types by analyzing the function definitions and infer the type definition based on this understanding.

```
g x y = x + y
```

Listing 3. Example of currying question context

Higher-order function questions also score relatively lower than other categories. In particular, questions that *utilize higher-order functions* (HFUN). Questions that ask students to define their implementation of a higher-order function (FARG) have a slightly higher average, but these questions follow a similar learning goal. With a p'-value of 0.651 and a relatively low deviation of only 0.092, the higher-order function questions seem to be more challenging than other questions on the conducted introductory functional programming exams. The concepts of function applications and higher-order functions lie close to each other, which might explain the similar results for the respective questions. In subsection 5.3, the connections between function applications and higher-order functions are explained, as well as the educational background necessary to understand them.

5 THE SOURCE OF CONFUSION: A FUNCTIONAL PROGRAMMING CONCEPT ANALYSIS

The start of the functional programming paradigm dates back to 1936 when Church and Rosser [7, 18] laid the foundation of the *lambda calculus*. The study of the *lambda calculus* is one of mathematical rules, expressed with *definitions* [2]. Others have already analyzed the history of functional programming and the origins of notations and language features, notably, Hudak [12], and later Turner [23]. The origins of functional language features are traced back to their foundational mathematical and linguistic roots, thereby elucidating the reasons behind the inadequate comprehension of several previously mentioned learning objectives. Learning objectives, previously assumed to be understood in the introductory course and unrelated to functional programming, are linked with the concepts treated in the course.

The first module of the study program briefly looks at different Computer Science topics found later in the program. In the first three weeks of the study, the students learn about low-level programming using Arduino, explore various searching and sorting algorithms, and learn the SQL language and databases, each topic covered in consecutive weeks. The fourth week tackles the topic of functional programming, culminating in an exam. The data analyzed in this study were derived from the results of this exam. All the while, students concurrently take a mathematics course. This course explains the basics of set theory, and then the focus will shift to introductory calculus.

5.1 Functional lists and list constructors

When students with only knowledge of imperative programming attempt to program in a functional language, they will find that constructs work fundamentally differently. In an earlier week, students are introduced to algorithms following the Pearls of Computer Science module. They implement searching and sorting algorithms in Python. Many of the operations on lists are in place, and functions that students have to write have to use `for`-loops and other strategies not found in functional programming.

In one of the exercises, the idea of a *tuple* is introduced. The module does not formally introduce the proper tuple notation but instead uses the Python list. The non-homogeneous property of lists in Python is used [9]. This is done to keep the content of the course within reason. The functional programming component teaches students to work with tuples using parentheses, which is also used in the context of currying. An example of a “tuple” using a list in Python can be seen in Listing 4.

```
tup = ["a", 1, True]
```

Listing 4. A tuple defined as a list in Python

Haskell is a polymorphic typed language in the definition given by Cardelli and Wegner [5], with both universal and ad-hoc forms of polymorphism. Lists in Haskell are universally polymorphic, which becomes apparent from its type definition `[] :: [a]`, which indicates a list with values of any type `a`. This is in contrast to Python, where a list is allowed to have elements of different types.

5.2 List comprehensions

Turner [23, p. 11] stood at the cradle of the list comprehension in programming languages, inspired by the “set expressions” proposed by Burstall and Darlington [4]. The SASL language that introduced so-called *ZF-expressions* was later used as inspiration for the Miranda programming language. This idea was renamed to *list comprehension*, which can be found in many programming languages today.

Although the implementation details of the list comprehension construct have changed over time, the origins remain the same. Set theory is introduced to students before the introductory functional programming component starts, thus students are expected to understand how to work with the set-builder notation.

The high school curriculum is of more influence to the prior knowledge of freshmen in introductory university courses. Important to note in this regard is the focus of the university in question: the prerequisite knowledge for Dutch students to join the Computer Science program is to have a *voorbereidend wetenschappelijk onderwijs* (vwo) diploma, with a mathematics level B (*wiskunde B*). Dutch students are free to take the more challenging subject of mathematics level D (*wiskunde D*), which offers more advanced mathematics than the B variant. Because this specific strand of mathematics is not taught in all high schools in the Netherlands, no university bachelor program in the Netherlands requires this course. Neither of these high-school subjects explicitly mention set theory in their curriculum [20, 21], meaning that participation in Mathematics D does not confer an advantage to students over their counterparts who did not engage in this course.

Computer science as a high school subject offers no significant advantage either. Computer science is an optional subject for both pupils and organizational staff. The structural shortage of high school computer science teachers percentage-wise measures to be the second highest out of all subjects [24, p. 10]. The curriculum consists of mandatory and optional *domains*, with the Programming Paradigms domain being optional [22].

5.3 Learning function types: function application, currying and higher-order functions

Function application and currying are often mentioned together in functional programming courses. At first inspection, the concepts appear analogous, and their application within functional programs seems to be somewhat identical. Currying a function produces a new function that can be partially applied, but the two concepts are not interchangeable. From subsection 4.2 it becomes apparent that students score lower on these learning objectives than the average case.

The Python course teaches students how to structure their code with *functions* using an imperative explanation: a function can have many arguments and will return a value. Functions in Haskell work differently: any function that does not represent a variable is a function `f :: a -> b`, where `b` might be a (partially applied) function or a function that represents a variable. Where a Python function `func(a,b,c)` needs all arguments filled[2], an equivalent Haskell implementation with type `func :: a -> b -> c -> d` can return a partial applied function if `func` is called with only one argument.

In the functional programming history summary by Turner [23], the isomorphism that lies at the core of currying is also noted: $(A \times B) \rightarrow C \cong A \rightarrow (B \rightarrow C)$. In this study stage, no formal introduction to mathematical proofs has been made, let alone proofs specified in set- or even category theory. Although it is unnecessary to understand this isomorphism and its proof in detail, its result proves the validity of currying (and for that matter, *uncurrying*).

The study of lambda calculus is focused on the "*process* of stepping from argument to value", instead of the "*graph*, that is the set of pairs of argument and associated value" [2, p. 1092]. Although this explanation might feel trivial for functional programmers and people working with any form of lambda calculus, it contradicts the mathematical approach that students are taught in Dutch high schools. Functional programming is a product of lambda calculus, meaning this difference can be seen as one of the fundamental learning concepts to achieve.

5.4 Conclusion

While numerous factors may contribute to students' difficulty grasping certain functional programming concepts, they ultimately stem from disparities in their prior knowledge. Dutch high school mathematics subjects do not tackle set theory or the lambda calculus style of working with functions. The high school mathematics curriculum only deals with functions as generators of graphs.

[2]Partial application can be achieved using the `functools` package or `lambda` in Python, but this is not part of the course material.

6 PROMPT ENGINEERING FOR ASSISTING INTRODUCTORY FUNCIONAL PROGRAMMING

Prompt programming or *prompt engineering* is the way a Large Language Model can be programmed to generate output to input prompts. The output can be influenced by adding extra context, examples, tasks, or other information before giving the LLM a specific question. This topic has seen a significant rise in research interest, with many papers published in the last three years. Below, we present three novel functional programming-specific prompt programs, inspired by the current state-of-the-art prompt patterns. The programs are presented in boxes, where text in italics is the written program and the text in bold is the input questions of the student.

6.1 Applied prompts for introductory functional programming

The Master Translator pattern. Reynolds and McDonell [19, p. 3] created a *Master Translator* pattern, to improve the performance of text translation with GPT-3. This pattern provides a context for the Large Language Model, increasing the chance of successful translation from a source to a target language. This pattern was created as one of the first pattern-like prompt programs and can be changed to suit the educational functional programming use case. The performance of this prompt program might be different on the new GPT-4 model.

> *A question is asked by a student: "***[student question]***"*
> *The teacher of the Introductory Functional Programming course helps the student without providing code:*
> ChatGPT response: ...

The Questioning Teacher pattern. The Questioning Teacher pattern is built from multiple prompt patterns presented by White *et al* [27]. The *Persona* pattern has been implemented, with the role of a teacher teaching an introductory functional programming course. The *Question Refinement* pattern is used to return three new questions for the student in the same context of the question asked. This method gives students room to explore their knowledge gaps further [1, 6].

> *You are a teacher of an introductory functional programming course. I am a student who asks you questions about functional programming. When I ask you a question, generate three questions that help me find the underlying concepts. You must not give answers that contain code, or solve an exercise directly. Here are the questions:*
> **[student question]**
> ChatGPT response: ...

The FP Context Manager pattern. The *Context Manager* pattern [27, p. 16] guides the answer of the LLM to the appropriate context. When the context is focused only on functional programming and is told not to give code, the answers are less likely to spoil the learning objectives.

Table 3. (Pre)prompt performance metrics

(Pre)prompt	Code	Tasks complete	Verbatim
No preprompt	63.3%	100.0%	100.0%
Master Translator	3.3%	96.7%	73.3%
Questioning Teacher	0.0%	96.7%	6.7%
FP Context Manager	13.3%	90.0%	60.0%

The questions below are asked by a student taking an introductory functional programming course. Focus only on programming and problem-solving using a functional approach. Whenever the student asks for code, only answer in terms of concepts and never provide code. Here are the questions:
[student question]
ChatGPT response: ...

6.2 Results

The prompts given above are evaluated using a small set of questions related to the functional programming course. These questions can be found in Appendix A[3].

In Table 3, the quantitative analysis of three prompts is presented, along with the performance of ChatGPT with no (pre)prompt. In all non-conceptual questions (e.g., the student asks for an implementation, or asks something related to an algorithm), plain ChatGPT returned a block of code with a solution. In the answer to question 10, the *Master Translator* pattern gave a code block with types, but not the actual implementation. In all cases, the responses contained the answer verbatim (e.g. instructions on how to write the code).

The *Questioning Teacher* pattern answered with three questions in all tested cases. In one instance, the prompt contained the answer to the question asked by the student, disguised as a question. The answer to question 10 contained concrete examples of concepts, but with appropriate questions per concept. The answer to question 24 contained the answer verbatim in an inline code snippet.

The *FP Context Manager* pattern contained some code blocks. In the answer to question 22, the code block was a repetition of the question statement. In questions 16, 19, and 25, code blocks in the response contained the verbatim answer. All questions where a direct answer could be given without providing a block of code, it was provided. The answers to questions that requested a block of code returned instructions and tips on how to write the code.

6.3 Quality of the output

Out of all tested prompt patterns, only the *Questioning Teacher* pattern did not generate any code blocks. For the other prompts, there is still a small chance that ChatGPT will generate code blocks. The *Master Translator* pattern adheres to the command of not generating code but interprets this concept widely. It still generated a type signature in a code block. If a solution to a question specifically asks for a type signature, this pattern is to be avoided.

The *Questioning Teacher* pattern did not provide any code blocks. In all cases, ChatGPT returned three questions. The quality of the questions varies, as some questions were not relevant to the given prompt or were repetitions of one another. Answers to questions 26 to 30 sometimes contained questions that lead the student to believe they are on the right track. In question 30, the third generated question asks about the use of the **map** function, which is incorrect for the use case. ChatGPT asks the student to think about the workings of **map**, instead of referring to different higher-order functions. The strictness offers a safety net for the actual answer but the usefulness of the questions is not guaranteed.

[3]The response data is made available as an artifact on https://doi.org/10.5281/zenodo.10590830

7 CONCLUSION AND DISCUSSION

We showed the effectiveness of three proposed prompts for educational functional programming. The presented prompts offered an advantage over plain ChatGPT without restrictions, as the percentage of verbatim answers was lower for all prompts. The *Questioning Teacher* pattern was the most conservative in its answer, with almost no verbatim answers and no code blocks given. The questions generated by this pattern were not always useful, but in most cases, it gave enough background for a student to proceed. The small number of misleading questions generated was a bigger problem, but most of these occurrences were surrounded by useful questions.

The quality of the answers given by ChatGPT varied and with more restrictions to the chatbot, less correct information was given. ChatGPT adhered to the tasks given by the prompt in most cases, but it was not airtight. The prompt patterns presented perform well overall, but adaptations for certain use cases can be made. Questions focusing on conceptual explanations were best asked to the *Questioning Teacher* pattern to prevent a verbatim answer. The *Master Translator* pattern also performed well with questions that asked for implementations of algorithms, where the answers contained a step-by-step guide on how to build the algorithm without spoiling the learning objectives. In general, all three patterns presented perform at promising levels that will protect the learning objectives significantly better than asking ChatGPT directly.

8 ACKNOWLEDGEMENTS

We kindly thank the Computer Architecture for Embedded Systems group of the Faculty of Electrical Engineering, Mathematics, and Computer Science at the University of Twente, for providing access to ChatGPT Plus.

A FUNCTIONAL PROGRAMMING QUESTION SET

Below are 4 categories containing 5 questions each. Each question is in the context of introductory functional programming exercises.

A.1 Questions regarding Haskell specifically

(1) What does (:) do in Haskell?
(2) Why is `[1,2,3]` the same as `1 : 2 : 3 : []` in Haskell?
(3) How does QuickCheck in Haskell work?
(4) What are the differences between sets and lists?
(5) What is the difference between list comprehension and set builder notation?

A.2 Questions about concepts

(6) What is the difference between a Python list and a Haskell list?

(7) What is the difference between partial application and currying?
(8) What is the difference between bubble sort in Python and in Haskell?
(9) What is a higher-order function?
(10) Explain the mathematics behind functional programming.

A.3 Questions asking about direct implementations

(11) How do I write a discriminant function in Haskell?
(12) How do I write a function in Haskell that computes the dividers of a given number?
(13) How do I write bubble sort in Haskell?
(14) Define a higher order function `itn` (for "iterate `n` times") with three arguments `f`, `a`, `n` in Haskell.
(15) Define a higher order merge sort function hoSort in Haskell, in which the relation by which a list should be sorted is given as an argument `r`.

A.4 Verification and debugging

Questions in red are negative, e.g. they ask for a correct/alternative implementation, which may not be given.

Invalid function type/function polymorphism. I have to write a function `incr` that checks if each number in a list is greater than its immediate predecessor. I wrote the function given here:

```haskell
incr :: [a] -> Bool
incr [] = True
incr [x] = True
incr (x:y:xs) = x < y && incr (y:xs)
```

I get the following error:

```
No instance for (Ord a) arising from a use of '<'
```

(16) How do I fix this?
(17) Why am I not able to use the < operator?
(18) What does **Ord** a mean?
(19) Give an implementation without this error.
(20) Implement this function using a higher-order function.

Currying/uncurrying. A function is given:

```haskell
f x y = x + y
```

When I call the function with `f (1, 2)`, I get the following error:

```
No instance for (Show ((Integer, Integer) -> (Integer, Integer))
    ) arising from a use of 'print'
  (maybe you haven't applied a function to enough arguments?)
```

(21) What does this mean?
(22) How do I fix this?
(23) How can I use the function `f` with `(1, 2)`?
(24) Give a function call that works.
(25) Generate example code that explains how to use function `f`.

Higher order functions. *This code sample contains an error that is unnoticed by the coder: the function `divisers` is incorrectly defined. The predicate should be `n 'mod' m == 0`.*

I want to define a `primes` function, which takes in a list of numbers and returns a list of all the prime numbers in the given list. I have defined the following functions:

```haskell
 divisers :: Int -> [Int]
 divisers n = [ m | m <- [1.. n], m `mod` n == 0 ]

isPrime :: Int -> Int
isPrime n = divisers n == [1, n]

primes :: [Int] -> [Int]
primes = map (isPrime)
```

When I load the file `Main.hs` into GHCi, I get the following error:

```
Couldn't match expected type 'Int' with actual type 'Bool'
In the expression: divisers n == [1, n]
  In an equation for 'isPrime': isPrime n = divisers n == [1, n]
```

But when I change the type of `isPrime`, I get the following error:

```
Couldn't match type 'Bool' with 'Int'
  Expected: Int -> Int
    Actual: Int -> Bool
  In the first argument of 'map', namely '(isPrime)'
  In the expression: map (isPrime)
  In an equation for 'mysum': mysum = map (isPrime)
```

(26) How can I fix this?
(27) Why should the type of isPrime be **Int** -> **Bool**? I want to make `primes :: [`**Int**`] -> [`**Int**`]`, but if I make `isPrime :: `**Int** `->` **Bool**, the type will become [**Int**] -> [**Bool**].
(28) How else can I define `primes`?
(29) Rewrite the given code such that it works.
(30) Generate a function `primes` which gives all prime numbers out of a given list.

REFERENCES

[1] Patrícia Albergaria Almeida. 2012. Can I ask a question? the importance of classroom questioning. *Procedia - Social and Behavioral Sciences* 31 (2012), 634–638. https://doi.org/10.1016/j.sbspro.2011.12.116
[2] Henk P. Barendregt. 1977. The Type Free Lambda Calculus. In *Studies in Logic and the Foundations of Mathematics*. Vol. 90. Elsevier, North-Holland Amsterdam, 1091–1132. https://doi.org/10.1016/S0049-237X(08)71129-7
[3] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, and Tom Henighan. 2020. Language Models are Few-Shot Learners. *Advances in neural information processing systems* 33 (Dec. 2020), 1877–1901.
[4] R. M. Burstall and John Darlington. 1977. A Transformation System for Developing Recursive Programs. *J. ACM* 24, 1 (Jan. 1977), 44–67. https://doi.org/10.1145/321992.321996
[5] Luca Cardelli and Peter Wegner. 1985. On understanding types, data abstraction, and polymorphism. *Comput. Surveys* 17, 4 (Dec. 1985), 471–523. https://doi.org/10.1145/6041.6042
[6] Christine Chin and Jonathan Osborne. 2008. Students' questions: a potential resource for teaching and learning science. *Studies in Science Education* 44, 1 (March 2008), 1–39. https://doi.org/10.1080/03057260701828101
[7] Alonzo Church and J. B. Rosser. 1936. Some properties of conversion. *Trans. Amer. Math. Soc.* 39, 3 (1936), 472–482. https://doi.org/10.1090/S0002-9947-1936-1501858-0
[8] T. Dousma, A. Horsten, and J. Brants. 1997. *Tentamineren* (3e dr ed.). Wolters-Noordhoff, Groningen. OCLC: 743244623.

[9] Python Software Foundation. 2024. Python Documentation: Data Structures. https://docs.python.org/3/tutorial/datastructures.html#id2

[10] 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 (March 2017), 65–100. https://doi.org/10.1007/s40593-015-0080-x

[11] Kevin Hammond. 2011. Why Parallel Functional Programming Matters: Panel Statement. In *Reliable Software Technologies - Ada-Europe 2011 (Lecture Notes in Computer Science)*, Alexander Romanovsky and Tullio Vardanega (Eds.). Springer, Berlin, Heidelberg, 201–205. https://doi.org/10.1007/978-3-642-21338-0_17

[12] Paul Hudak. 1989. Conception, evolution, and application of functional programming languages. *Comput. Surveys* 21, 3 (Sept. 1989), 359–411. https://doi.org/10.1145/72551.72554

[13] J. Hughes. 1989. Why Functional Programming Matters. *Comput. J.* 32, 2 (Feb. 1989), 98–107. https://doi.org/10.1093/comjnl/32.2.98

[14] Enkelejda Kasneci, Kathrin Sessler, Stefan Küchemann, Maria Bannert, Daryna Dementieva, Frank Fischer, Urs Gasser, Georg Groh, Stephan Günnemann, Eyke Hüllermeier, Stephan Krusche, Gitta Kutyniok, Tilman Michaeli, Claudia Nerdel, Jürgen Pfeffer, Oleksandra Poquet, Michael Sailer, Albrecht Schmidt, Tina Seidel, Matthias Stadler, Jochen Weller, Jochen Kuhn, and Gjergji Kasneci. 2023. ChatGPT for good? On opportunities and challenges of large language models for education. *Learning and Individual Differences* 103 (April 2023), 102274. https://doi.org/10.1016/j.lindif.2023.102274

[15] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2019. How Teachers Would Help Students to Improve Their Code. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*. ACM, Aberdeen Scotland Uk, 119–125. https://doi.org/10.1145/3304221.3319780

[16] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2019. A Systematic Literature Review of Automated Feedback Generation for Programming Exercises. *ACM Transactions on Computing Education* 19, 1 (March 2019), 1–43. https://doi.org/10.1145/3231711

[17] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2023. Large Language Models are Zero-Shot Reasoners. http://arxiv.org/abs/2205.11916 arXiv:2205.11916 [cs].

[18] Hans-Wolfgang Loidl, Ricardo Peña, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, and Gerhard Weikum (Eds.). 2013. *Trends in Functional Programming*. Lecture Notes in Computer Science, Vol. 7829. Springer Berlin Heidelberg, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-40447-4

[19] Laria Reynolds and Kyle McDonell. 2021. Prompt Programming for Large Language Models: Beyond the Few-Shot Paradigm. http://arxiv.org/abs/2102.07350 arXiv:2102.07350 [cs].

[20] Rijksoverheid. 2013. Examenprogramma Wiskunde D vwo. https://www.examenblad.nl/system/files/2013/examenprogramma_wiskunde_dvwo.pdf

[21] Rijksoverheid. 2014. Examenprogramma Wiskunde B vwo. https://www.examenblad.nl/system/files/2014/Examenprogramma_wiskunde_B_vwo%20_%20versie_van_OCW_CORRECTIE_DEC_2017.pdf

[22] Rijksoverheid. 2018. Examenprogramma informatica havo/vwo. https://www.examenblad.nl/system/files/2018/examenprogramma_Informatica_havo-vwo.pdf

[23] D. A. Turner. 2013. Some History of Functional Programming Languages. In *Trends in Functional Programming*, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Hans-Wolfgang Loidl, and Ricardo Peña (Eds.). Vol. 7829. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–20. https://doi.org/10.1007/978-3-642-40447-4_1 Series Title: Lecture Notes in Computer Science.

[24] Marcia den Uijl, Hendri Adriaens, Maartje Elshout, and Suzan Elshout. 2024. Personeelstekorten voortgezet onderwijs. https://zoek.officielebekendmakingen.nl/blg-1122086.pdf

[25] Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. 2023. Plan-and-Solve Prompting: Improving Zero-Shot Chain-of-Thought Reasoning by Large Language Models. http://arxiv.org/abs/2305.04091 arXiv:2305.04091 [cs].

[26] Tianjia Wang, Daniel Vargas-Díaz, Chris Brown, and Yan Chen. 2023. Exploring the Role of AI Assistants in Computer Science Education: Methods, Implications, and Instructor Perspectives. In *2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. Washington, DC, USA, 92–102. https://doi.org/10.1109/VL-HCC57772.2023.00018 arXiv:2306.03289 [cs].

[27] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. Schmidt. 2023. A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT. (2023), 1–19. https://doi.org/10.48550/ARXIV.2302.11382 Publisher: arXiv Version Number: 1.