Journal of Computer Languages 85 (2025) 101372

Contents lists available at ScienceDirect

Journal of Computer Languages

journal homepage: www.elsevier.com/locate/cola

Goliath and the Cognitive Load Theory ☆

Tiago Carvalho Freitas [a], Alvaro Costa Neto [a,b,c,*], Maria João Varanda Pereira [b], Pedro Rangel Henriques [a]

[a] ALGORITMI Research Centre/LASI, DI - University of Minho, Campus de Gualtar, Rua da Universidade, Braga, 4710-057, Portugal
[b] Research Centre in Digitalization and Intelligent Robotics, Polythechnic Insitute of Bragança, Campus de Santa Apolonia, Bragança, 5300-253, Portugal
[c] Instituto Federal de Educação, Ciência e Tecnologia de São Paulo, Campus Barretos, Avenida C-1, 250, Barretos, 14781-502, SP, Brazil

ARTICLE INFO

Keywords:
Computer programming education
Artificial intelligence
Domain-Specific Languages
Programming exercises
Cognitive load theory

ABSTRACT

A successful teaching effort is usually dependant on several factors. From the right environment, to a precisely worded exercise statement, it rests on the teacher's shoulders the concoction of the most effective learning assets to their students. A significant part of this process lies on practise: students commonly solidify their knowledge by solving exercises. Creating new programming exercises, specially in high-demand environments such as large classrooms, is a repetitive and error-prone process, specially when stacked with other typical affairs that educators are required to attend to. Goliath, one of the two main contributions of this article, is a template-based, Artificial Intelligence (AI) supported exercise generator, that aims to facilitate the creation of exercise repositories. By using a Domain-Specific Language (DSL) to define exercise templates, combined with the automatic generation of different exercise types, educators can use Goliath's features to improve their exercise repositories, both in size and variety. This systematic approach allows for greater control and automatisation than using a Large Language Model (LLM) directly, as the exercises' main components can be pre-defined and pre-configured via their templates. Goliath, which is available online for free access, has been tested and its usability assessed. Combined with these functionalities, the content of the exercises themselves, the manner in which they are presented, and how they are rated for difficulty should also be considered in high regard when designing programming exercises. The Cognitive Load Theory (CLT) provides a conceptual foundation to understand problem-solving mechanisms that are commonly found in several aspects and situations of daily life, such as solving programming exercises. This foundation has been explored and systematically structured to construct the second main contribution of this article: guides to create exercise templates in Goliath founded on the Cognitive Load Theory, aiming to improve both teaching and learning computer programming.

1. Introduction

The path to learn computer programming is full of challenges, that range from technical, to personal [1,2]. Of the many traits required from students, persistence and practise are among the top contenders for importance. As computer education became a reality, many advances have being made to tools an techniques that aid students and teachers in this endeavour [3–5]. Solving programming exercises that complement what is taught in class, nonetheless, is still paramount for success. Through repetition, students can solidify their knowledge and find gaps in their understanding of the many topics in computer programming.

In order to achieve success, students commonly rely on programming exercises that have been previously created, verified and published by their teachers, peers, textbooks, websites, and others. Variety and immediate feedback are not commonly available in these cases, as the process of creating said exercises is neither quick, nor trivial [6]. Given current social and technological contexts where immediate gratification is expected at every tap and click, the patience to wait for new resources of this kind is increasingly low in students. While whims and fads should not dictate educational progress, it is reasonable to

☆ This work has been supported by FCT – Fundação para a Ciência e Tecnologia within the R&D Units Project Scope: UID/00319/2023. The work of Maria João and Alvaro was supported by national funds: UID/05757 - Research Centre in Digitalization and Intelligent Robotics (CeDRI); and SusTEC, LA/P/0007/2020 (DOI:10.54499/LA/P/0007/2020).

* Corresponding author at: ALGORITMI Research Centre/LASI, DI - University of Minho, Campus de Gualtar, Rua da Universidade, Braga, 4710-057, Portugal.
E-mail addresses: tiago10cf@hotmail.com (T.C. Freitas), alvaro@ifsp.edu.br (A. Costa Neto), mjoao@ipb.pt (M.J.V. Pereira), prh@di.uminho.pt (P.R. Henriques).

https://doi.org/10.1016/j.cola.2025.101372
Received 30 May 2025; Received in revised form 9 September 2025; Accepted 21 October 2025
Available online 29 October 2025
2590-1184/© 2025 The Authors. Published by Elsevier Ltd. This is an open access article under the CC BY-NC license (http://creativecommons.org/licenses/by-nc/4.0/).

assume that advances in feedback times and exercise variety would bring greater chances of success [7].

Goliath is an application designed and built to aid teachers in the process of creating and providing different programming exercises to students. It aims to help educators by leveraging both an AI-supported generation of exercise templates, and a DSL that allows automatic variation to be added to the exercises that result from these templates. It also supports exercise attribution and answering via dedicated interfaces, and the immediate feedback for students when they answer the exercises. This combination of technologies in a single system is currently unique, as our literature review shows in Section 2.

By freeing educators from manual aspects of distributing and varying exercises, Goliath also allows them to focus on the exercises' contents and how to correctly create challenges that improve students problem-solving capabilities. The Cognitive Load Theory (CLT), created by John Sweller [8], provides an structured approach in order to reach this goal. By defining limitations and mechanisms that explain how problems are mentally solved, CLT establishes a foundation that can be explored in Goliath to create focused and efficient programming exercises (and their variations).

This paper is an extended version of the one presented in FedCSIS 2024 [9]. Beyond Goliath's main functionalities that have been included and presented in the original paper, this article provides a theoretical foundation on how to better tackle cognitive load when creating exercise templates, and setting its difficulty level. This rationale orients decisions that are usually made intuitively by educators when they create programming exercises. This is the new contribution of this article. It includes the presentation of a novel framework for the identification and evaluation of cognitive load in computer programming, and how it can be tied to Goliath's exercise generation features.

This paper is divided into seven sections. After the introduction and contextualisation in Section 1, Section 2 contextualises resources for practising computer programming, while Section 3 presents how exercises are structurally constructed, based on foundational research. Section 4 briefly present a comparison of a few AI-supported methods for automatic and semi-automatic generation of Computer Programming exercises that were available when Goliath was designed. Section 5 details structural and functional aspects of Goliath and present an overview of the tests, results and feedback. Section 6 brings the major addition of this extended paper by conceptualising a theoretical foundation based on Cognitive Load Theory and how to apply it to construct and setup Goliath's exercise templates. Finally, Section 7 concludes the paper with final regards on Goliath's goals and achievements, and suggests future derivations and improvements within the scope of this research.

2. Resources for practising programming

When it comes to computer programming, practice support assumes a wide range of implementations. Lists of exercises (written and printed by teachers or tutors) and problems in textbooks are classical offerings in educational contexts. More modern tools, such as online guides to programming learning [10], online courses, program animation applications [11], and automatic evaluation tools [12].

Nonetheless, creating exercises is still a challenge. Many factors demand consideration in order to create clear, useful ones: the approach to the topic under practice, the difficulty it will present to a diverse student population, the correct and unambiguous wording of the problem statement to avoid misdirection and general confusion, *etc*. Once all of these challenges have been surpassed and a collection of quality exercises is reached, creating new ones, or even variations of those that already exist, is not trivial. Besides being a time-consuming task, it becomes ever more prone to errors as repetition allows for lost of focus to creep in, resulting in typos, missed information, and incoherence.

There are systems that automate these tasks, such as SIETTE [13], which allows the creation and management of exercises repositories, and R/exams [14], a package for the R language that provides mechanisms to create both HTML and LaTeX versions of parametrised exercise lists. Although successful in their own ways, the initial generation of the exercise components (more on that on Section 3) is done directly by the user, without any kind of initial guidance or suggestion.

On the other hand, LLMs such as ChatGPT and Gemini can provide mechanisms for obtaining different exercises directly. These applications provide the initial suggestion that the previously mentioned systems lack, going as far as generating complete exercises. Nonetheless, they still require educators to manually implement the layout, distribution and variation of the exercises, as the results from their prompts are static and complete responses.

Goliath's unique contribution lies in the intersection of these approaches, as it provides both an initial spark of creativity via AI-supported suggestions, as well as the controlled (and automated) variability and distribution of the exercises.

3. Structure of programming exercises

A formal structure for programming exercises had to be defined in order to automate their generation—which is, ultimately, Goliath's central objective. Exercises were classified into different *types* that (commonly found in tests, lists, websites, *etc*), and segmented into three main *components* (the *statement*, the *code*, and the *answer area*), each with its own responsibility in communicating the exercise's intent to the student.

3.1. Exercise types

From many definitions of exercise types available in the literature [15–17], Goliath relied on those published in [18], given their resemblance to how it implements their generation and which types it is capable of handling.

From all seven types in the source publication [18], Goliath adapted the implementation of three:

1. **Code from scratch:** students must write down the complete solution to a problem from scratch (as the name implies). No support code (or template) is provided, only a dedicated empty space for the answer;
2. **Code completion:** In order to solve this type of exercise, students must fill blanks that have been strategically positioned in a provided excerpt of code;
3. **Output or state prediction:** students are asked to find out either the output of a source code's execution, or the value of a variable throughout its lifetime.

Further details on the adaptation of these types will be presented in Section 5.

3.2. Exercise components

There are usually three main components to consider in a typical exercise: the problem *statement*, the accompanying *code*, and the *answer area* (see Fig. 1).

The problem *statement* contains text that is presented to the student explaining the context and parameters of the problem, the type of answer that is expected, and other pertinent details about the exercise. An excerpt of *code* usually follows, containing snippets to be compared, completed, analysed or fixed. It supports the problem *statement* to establish a basis for solving the exercise. Finally, the *answer area* contains wither a blank space or the distribution of possible options for the student's answer.

Goliath implements this structure by in all its functionalities, from the AI-supported generation of the template's initial version, to the automatic generation of the answer options for multiple choice exercises.

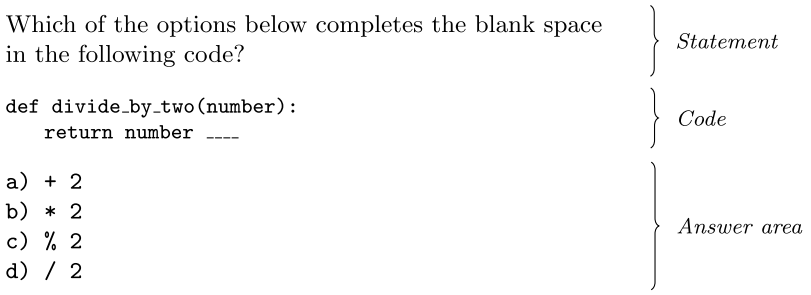

Fig. 1. Typical components of a programming exercise.

4. AI-supported exercise generation

Natural Language Generation (NLG) (a sub-field of Natural Language Processing (NLP)) encompasses theories and techniques that allow for the production of coherent and useful text in multiple languages, and it is commonly applied to create chat-bots, translators and similar tools [19–21]. The process involves taking input data, such as keywords, a set of facts or a starting piece of text, and transforming (or complementing) it into meaningful output text.

4.1. AI models to generate exercises

Goliath applies a multi-step workflow for the generation of exercises, in which the AI models represent an important part of it. Nonetheless, it was never intended for Goliath to rely solely on AI models for its functionalities. The main reason behind this decision was to avoid possible aberrations from the models to reach students, consequently causing confusion and degrading the entire educational process. Section 5 explains this workflow in greater detail, but one feature must be clear beforehand: the exercise components (statement, code and answers) generated by the models *constitute only suggestions for the teacher*. In other words, the teacher requests an initial version of an exercise based on keywords, such as "loops", "functions", and "pointers", and Goliath presents suggestions, generated by its models, for the exercise's statement and code. The teacher can then edit and augment these suggestions in the form of a template, which generates actual exercises that students can practice on. The results from the models are not presented directly to students.

Goliath uses two NLP models to generate its suggestions for exercise statements and code: KeysToText [22] and CodeT5 [23], respectively. This choice was based on an evaluation conducted in the second half of 2022,[1] when four candidates were considered [24]: GPT 3.5, GPT-2, KeyToText and CodeT5.

The main objective of this evaluation was to choose models that could be integrated into Goliath, given a few factors that would influence this decision:

Cost How much do models cost to run? Given that Goliath is an open-source and free application, paying for the models could potentially be harmful to its adoption, as institutions would be required to support this cost by themselves;

Hosting Can models be hosted in-house or are they necessarily remote services? Models require hardware and network resources that might be out of reach for some institutions, limiting their uses to services provided by remote servers. This approach could be prohibited entirely given security protocols in certain institutions, or just become a point of failure if the remote server becomes unavailable;

Input What type of input do models require? Are they complex natural language prompts or simple keywords? Can they interoperate by chaining the output of a model as the input for another? As Goliath's objectives contemplate more than the generation of the exercise components by the models, the complexity of their inputs could hinder their use as part of the full workflow (see Section 5 for an in-depth description of Goliath's functionalities and how they interact);

Output What is the output like? Is it a complete exercise, containing statement, code and answer options, or is it just one of those parts? Ultimately, the output of the models would require some kind of adjustment in order to fit the next step in Goliath's workflow: creating the exercise template (also further explained in Section 5). The easier this adjustment could be implemented, the better a model would fit.

Table 1 shows an overview of the main characteristics of each model for these factors. OpenAI's GPT-3.5 was the first to be evaluated. Despite returning accurate results (as in, correct and complete exercises for what was required in the input prompt), this version had a few shortcomings that made it less desirable for use in Goliath:

- It was free to use only within a gracing period, which would entail costs after deployment;
- The input prompt had to contain a complete description of the exercise and how it should be generated. This meant that Goliath would have to either compose this prompt based on a simpler input from users, or require them to write the prompt themselves—a more complex and less practical approach from their perspective;
- The output contained the exercise statement, code and answer in one single string, which would require post-processing in order to separate each component for consequent use in the exercise template;
- Even it was publicly available for download, it would be unfeasible to use it locally given its size. This meant Goliath would be dependent on OpenAI's servers, consequent communication and availability.

Listing 1 shows an example of how GPT-3.5 was evaluated. As can be seen by the contents of variable `prompt`, Goliath would need to construct a natural language prompt that almost contains the entire statement in order to integrate it. It would also require significant transformation of the output to split the exercise into its components (statement and code), a requirement for the consequent construction of the exercise template.

GPT-2 was evaluated as the second candidate. Since it has a more manageable size, the whole model was downloaded and fine-tuned with a dataset of 300 basic Python exercises [25]. The input consisted in a pre-formatted request for the code of a specific statement (see variable `prompt` in Listing 2), and the output resulted in both the statement and the code for the exercise. It was less accurate than GPT-3.5 — as can be

[1] At the time of the evaluation, GPT-3.5 was the newest version and ChatGPT was yet to be released.

Table 1
Comparison between different AI models and their features for programming exercises generation.

Model	Cost	Hosting	Input	Output
GPT-3.5	Paid	Remote	Full request and definitions	Statement, code and answer
GPT-2	Free	Local	Formatted text and statement	Statement and code
KeysToText	Free	Local	Keywords	Statement
CodeT5	Free	Local	Function statement	Code

Listing 1 Example of a programming exercise generation using GPT-3.5.

Source:

```
prompt = "Create a programming exercise in
    Python about string manipulation and write
    the answer."
exercise = openai.Completion.create(engine="text
    -davinci-003", prompt=prompt, temperature
    =0.7, max_tokens=500, top_p=1,
    frequency_penalty=0, presence_penalty=0)
```

Output:

```
Write a program to print out the last three
    characters of a string.

#Answer:
string = input("Enter a string: ")
length = len(string)
last_three_chars = string[length - 3:]
print(last_three_chars)
```

Listing 2 Example of a programming exercise generation using the fine-tuned GPT-2 model.

Source:

```
prompt = """Exercise

Instructions:
Write a Python program to reverse a string.

Code:"""

model.generate(prompt=prompt, max_length=500,
    temperature=0.3, top_p=0.9)
```

Output:

```
Exercise

Instructions:
Write a Python program to reverse the binary
    representation of a given integer.

Code:
def test(n):
  return int(bin(n)[::-1][:-2], 2)
```

seen by the misinterpretation of the instructions in Listing 2 — while some of the same shortcomings were also present, such as the necessity to provide most of the exercise as input (including the statement), and considerable post-processing of the output.

As the generation via a single model was creating undesirable side-effects in the form of input and output processing, two other models were evaluated in tandem, by dividing the process into individual tasks: KeyToText would generate the statement, while CodeT5 would generate the code.

Listing 3 Example of an exercise statement generation using the fine-tuned *Keys-To-Text* model.

Source:

```
keywords = ["remove", "list", "odd numbers"]
model.predict(keywords)
```

Output:

```
Write a function to remove odd numbers from a
    list.
```

Listing 4 Example of code generation using *CodeT5*.

Source:

```
prompt = "Function to remove odd numbers from a
    list."

ids  = tokenizer(prompt, return_tensors="pt").
    input_ids
code = model.generate(ids, max_length=128)
tokenizer.decode(code[0], skip_special_tokens=
    True)
```

Output:

```
def remove_odd_numbers(nums):
    return [n for n in nums if n % 2 == 0]
```

KeyToText[2] was fine-tuned using the Mostly Basic Python Problems dataset [27], containing 1000 entry-level programming problems. The model was able to generate accurate (albeit simple) exercise statements from a minimum of three keywords (see variable `keywords` in Listing 3). This solution was advantageous when compared to the GPT models, as it was able to automate the generation of the exercise statement from a relatively simple input, instead of requiring a full prompt.

The output of KeyToText (the exercise statement) was adapted and fed into CodeT5[3] (see variable `prompt` in Listing 4) to generate the code. It was trained with the CodeSearchNet [28] dataset collection, and was able to generate simple Python source code coherent with the exercise statement.

Beyond the fact that processing the input and output of KeyToText and CodeT5 was trivial, this strategy of using dedicated models had another benefit: the generation process could be taken piece-wise, allowing Goliath to check with users if the statement generated by KeyToText was a good result before inputting it into CodeT5. This pause in the generation process was a more reliable functionality overall, as

[2] KeyToText is based on the Text-to-Text Transfer Transformer (T5) architecture [26].

[3] CodeT5 also implements a T5 architecture.

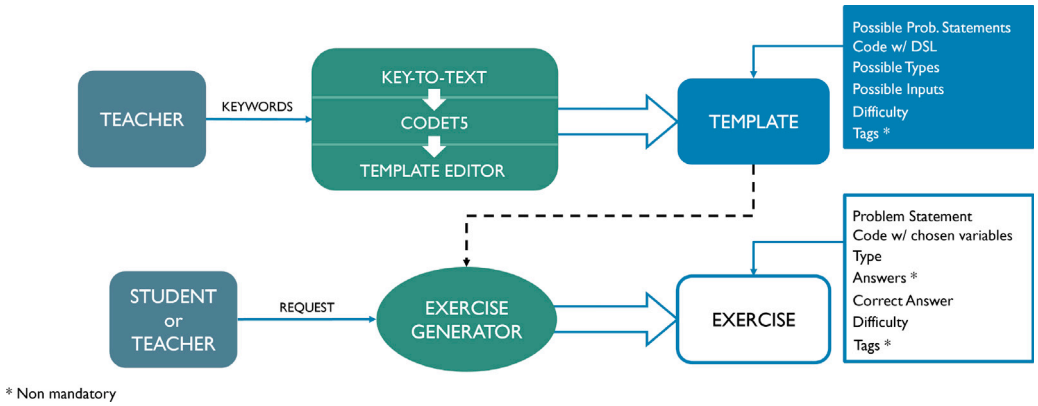

Fig. 2. Goliath's functionality and main features.

users could interfere in a timely manner, and correct any inaccuracies from the models before dealing with the next steps.

Finally, after several manual tests, it was clear that KeyToText and CodeT5 were able to generate comprehensible and coherent suggestions. Goliath implemented both models in a sequence of execution (further explained in Section 5.2) to generate initial templates for Python programming exercises.

5. Goliath

Goliath is an online application[4] for semi-automatic generation of programming exercises.[5] It is based on two AI models (KeyToText and CodeT5), a template system, a DSL, and other supporting functionalities. It allows teachers to create a repository of templates that generate programming exercises, and assign them to students in order to support their programming practices.

Fig. 2 shows the general design of Goliath's functionality and its main features. The workflow is divided into two processes: the creation of templates (top half of Fig. 2) and the generation of exercises (bottom half).

The template creation is based on a sequence of operations:

1. The teacher provides Goliath with (at least three) keywords to generate an exercise statement using KeyToText (left side of Fig. 3);
2. The statement, after reviewed by the teacher, is fed into CodeT5 to generate its accompanying code (right side of Fig. 3);
3. Both statement and code are presented to the teacher in a Template Editor;
4. The teacher embeds parameters into the template using commands from a DSL, specifying variations on the exercise;
5. The template is stored in a repository, along with a few other settings that the teacher can define (explained in Section 5.3).

Given that the teacher may not want to use the AI-supported functionalities, both steps 1 and 2 can be skipped, as long as he or she writes the statement and the code from scratch. Further details about the template system and the DSL are presented in Section 5.1. Nonetheless, creating a template does not generate an exercise. This only happens after a request is sent to the Exercise Generator by the student. This generation process is straightforward:

1. The student requests an exercise that was assigned by the teacher;
2. The corresponding template is fetch from the repository;
3. A version of the exercise is automatically generated by choosing randomly one of the pre-defined variations that the template specifies;
4. The answer alternatives are created *ad hoc*, based on the exercise version;
5. The exercise (statement, code and answer alternatives) is presented to the student who can answer it;
6. The correct answer is shown as feedback.

This division of workflows is intentional: by delaying the generation of the exercise, Goliath is able to add a layer of controlled randomisation to the process. This fact contributes to its *replayability*[6] and provides a sense of discovery to the students. Further explanations on the exercise generation are given in Section 5.2.

5.1. Goliath's DSL and the template system

Goliath's template system allows educators to automatically generate different versions of an exercise. It takes the initial version of the statement and the code (either suggested by the models or directly written by the teacher) and allows its parametrisation via a DSL that was specifically designed for this process.

As the core of the template system is the concept of *versions* which, in this context, represent variations of what is requested from students. As an example, a simple exercise statement could read "Write a function to remove all odd numbers from a list of integers". In order to change the request from *remove odd numbers* to *remove even numbers*, this statement needs only to be minutely changed. Furthermore, the statement could also be modified to request the removal of all positive numbers, all prime numbers, or any equivalent variation. Thus, if given the possibility of automatically generating these versions (odd, even, positive, prime, *etc.*) from one template, students could practice multiple times, while educators needed only to construct and parametrise the template once.

The template consists in the statement, the code, and a few added settings (explained in Subsection 5.3). The answer area is the only component that is not directly included in the template, as it is automatically created when the exercise is generated. For demonstration purposes, consider Listing 5 as a basis for a template that was suggested by the AI models.

[4] Accessible at: https://goliath.epl.di.uminho.pt/.

[5] Only Python is currently supported as the programming language for the exercises. This technical limitation was implemented due to two reasons: the AI models have been fine-tuned with datasets of Python source code, and internal verification mechanisms also assume Python as the language of choice for the exercises.

[6] Replayability indicates the possibility of reusing the system with a lower chance of encountering the same state more than once.

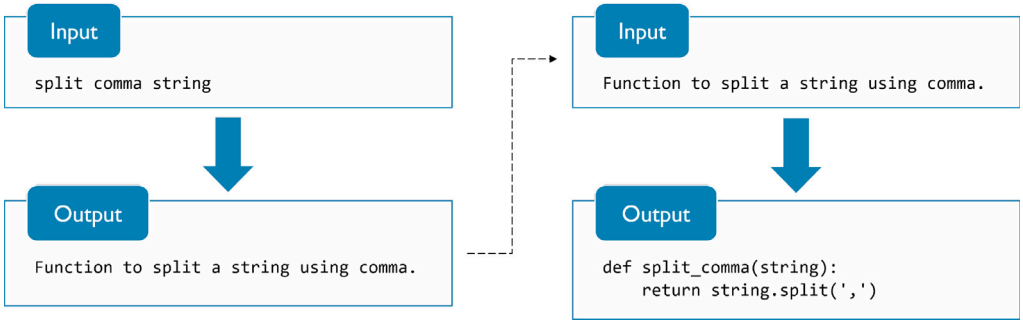

Fig. 3. AI-supported statement and code generation.

Listing 5 Initial template created by the keywords *list*, *remove*, and *numbers*.

Statement:

```
Function to remove all odd numbers from a list
    of integers.
```

Code:

```
def remove_odd_numbers(nums):
    return [n for n in nums if n % 2 == 0]
```

Listing 6 Grammar for the DSL that allows templates to generate several versions of an exercise. Note that the commands can not be recursively nested.

```
start       : declaration* commands

declaration: ID "=" STRING ("," STRING)+
commands    : (altern | conditional)*

altern      : STRING ("," STRING)* fail?
conditional: "case" vars ":" actions else? fail?
vars        : ID ("," ID)*
actions     : action (";" action)*
action      : STRING ("," STRING)* "=>" STRING

else        : ";" "else" ":" STRING
fail        : ";" "fail" ":" STRING ("," STRING)*
```

If left unchanged, this template would only generate one version of the exercise ("Function to remove all odd numbers from a list of integers"). In order to make it more flexible, it is necessary to parametrise it. A DSL was designed for this purpose, whose grammar, presented in Listing 6, was used with Lark [29] to develop an interpreter. This DSL is intended to be used in the same fashion as a markup language, whose commands must be interspersed into the text.

The commands of parametrisation are placeholders delimited by double curly brackets (`{{` and `}}`). They are substituted by a value (generally speaking, a piece of text) when the exercise is generated. There are three types of commands: *key declarations*, *conditional placements*, and *simple alternatives*.

5.1.1. Key declaration

Multiple keys can be declared in the exercise statement, representing variations on what is requested from the student. Each key defines a list of possible values. For example, in order to parametrise the template of Listing 5 and allow the generation of both *odd* and *even* versions of its exercise, the statement should be rewritten as:

Listing 7 Template with a key declaration in the statement.

Statement:

```
Function to remove all {{ x = 'odd', 'even' }}
    numbers from a list of integers.
```

Code:

```
def remove_odd_numbers(nums):
    return [n for n in nums if n % 2 == 0]
```

```
Function to remove all {{ x = 'odd', 'even' }}
    numbers from a list of integers.
```

The key `x` creates the variation for *odd* and *even*. When an exercise is generated from this template, its statement will read either "*...Function to remove odd numbers...*" or "*...Function to remove even numbers...*". Listing 7 shows the template with the parametrised statement.

Although unnecessary for this example, there could be multiple key declarations. The following statement would be able to generate four different versions of the exercise: "remove *odd* from *a list*", "remove *odd* from *an array*", "remove *even* from *a list*", and "remove *even* from *an array*".

```
Function to remove all {{ x = 'odd', 'even' }}
    numbers from {{ y = 'a list', 'an array' }}
    of integers.
```

Key declarations must only occur in the statement, as they establish variations on what the exercise will ask from the student. They would not make sense in the code, given that its logic is supposed to be derived from the statement, not the other way around.

5.1.2. Conditional placements

Assuming that an exercise must be entirely coherent, if there are possible variations on what is asked from student (via *key declarations* in the statement), a fixed code would probably be wrong. *Conditional placements* allow the code to adapt to the variations of the keys when the exercise is generated.

Conditional placements always start with the keyword `case`, followed by the list of keys that should be considered. For each combination of values, a result is specified using the *hash-rocket* notation (`=>`). It is similar to a `switch-case` construct in a conventional programming language.

Going back to the example in Listing 7, if the key `x` assumes the value `odd` in the exercise, the second line of code must read:

```
return [n for n in nums if n % 2 == 0]
```

which guarantees that only even numbers are left in the list, effectively removing odd numbers. On the other hand, if `even` is chosen for `x`, the line must be changed to:

```
return [n for n in nums if n % 2 == 1]
```

The template can be parametrised for this variation through a conditional placement on `x`:

```
return [n for n in nums if n % 2 == {{ case x: '
    odd' => '0'; 'even' => '1' }}]
```

This conditional placement results[7] in `0` if `x` is `odd`, and `1` if `x` is `even`. Since `x` has only two possible values, the command could also be written using an `else` clause:

```
return [n for n in nums if n % 2 == {{ case x: '
    odd' => '0'; else: '1' }}]
```

This clause acts as a complementary branch in the conditional placement. It is similar to the `default` branch in a classic `switch-case`.

The `fail` clause can be used to specify invalid options for the placement:

```
return [n for n in nums if n % 2 == {{ case x: '
    odd' => '0'; else: '1'; fail: '2', '3' }}]
```

The definition of invalid options using the `fail` clause is important as Goliath is unable to automatically come up with wrong answers for the exercise versions (further explanations in Section 5.2).

Finally, a conditional placement can take multiple keys into consideration:

```
{{ case x, y: 'odd',  'a list'   => '0';
              'even', 'an array' => '1';
        else: '-1';
        fail: '2','3' }}
```

5.1.3. Simple alternatives

Unlike conditional placements, *simple alternatives* do not take keys into consideration. They simply create variations, with the added possibility of specifying invalid options:

```
return [n for n in nums if n {{ '\% 2'; fail: '/
    2', '* 2', '+ 2' }} == 0]
```

The correct option for that portion of code is `% 2`, while the others (defined by the `fail` clause) are invalid and will generate incorrect alternatives for the answer in the exercise.

Simple alternatives can also be used to create equivalent variations in the code:

```
h = {{ 'n / 2.0', 'n * 0.5' }}
```

Both variations of the code above are equivalent in their objective (assigning the half of `n` to `h`) and would result in two different, albeit valid, versions.

5.1.4. Final remarks on the DSL

The three types of commands (key declarations, conditional placements and simple alternatives) allow for the definition of both valid and invalid variations. Both are equally important for the generation of the exercise, given that Goliath is not able to self-determine which changes in the code would result in correct answers and which would not. In order to that, Goliath would be required to not only perceive what is asked by the statement — meaning, the actual problem that the exercise entails — but also to check if the code's logic is coherent with it. The Halting Problem guarantees this impossibility.

Typical error checking is done during the interpretation of the DSL commands, including the use of undeclared keys, syntactic mistakes and invalid commands. After all commands have been successfully interpreted, an internal JSON-like representation of the keys and their variations is created, and stored with the template (see Listing 8).

Listing 8 Internal representation of a template's keys and variations.

```
"keys": {
    "x": ["odd", "even"]
},

"variables": {
    "alt1": {
        "correct": [
            { "value": "\% 2" }
        ],
        "wrong": ["/ 2", "* 2", "+ 2"]
    },
    "alt2": {
        "correct": [
            {
                "value": "0",
                "conditions": [
                    { "key": "x", "index": 0 }
                ]
            },
            {
                "value": "1",
                "conditions": [
                    { "key": "x", "index": 1 }
                ]
            }
        ],
        "wrong": ["2", "3"]
    }
}
```

Listing 9 Template containing all three constructs.

Statement:

```
Function to remove all {{ x = 'odd', 'even' }}
    numbers from a list of integers.
```

Code:

```
def remove_{{ x }}_numbers(nums):
    return [n for n in nums if n {{ '\% 2'; fail
    : '/ 2', '* 2', '+ 2' }} == {{ case x: 'odd'
     => '0'; else: '1'; fail: '2', '3' }}]
```

Listing 9 shows the complete version of the exercise template, including all three types of commands.

5.2. Exercise generation

As previously mentioned, a template acts as a blueprint that generates different versions of an exercise. The generative process begins with a request from a student, when he or she accesses an assignment created by the teacher.[8] In this context, the real exercise can be considered an instantiation of the assignment's template, in a similar fashion to a an object-class relationship.

[7] As previously explained, all commands from the DSL are placeholders. The result of a command refers to the value that will substituted it in the exercise.

[8] Teachers may create an assignment at any time, as long as the template has already been created and stored in the repository.

A few important considerations must be made in order to understand the exercise generation:

- There are three types of exercises available in Goliath: *code selection* (adapted from *code from scratch*), *input/output*, and *code completion* (also adapted).[9] This limitation was imposed by the mechanisms implemented in the exercise generation. Additional types would require more commands in the DSL and more settings in the template, which was unfeasible for this version of Goliath;
- Answers are presented in multiple choice format for the *code selection* and *code completion* types.[10] Implementing an open answer format for these types would either negate the immediate feedback to students, as they waited for the teacher to correct the answers, or require the implementation of an extremely accurate NLP model. *Input/output*, on the other hand, is presented in open answer format, as it can be trivially verified by executing the code of the exercise itself;
- Both the statement and the code in the template need to follow specific discourses. Statements must start with "*Function to...*", "*Function that...*", *etc*. while the code should always contain a single function definition. This requirement exists because Goliath needs to adapt them to the exercise type when the generation occurs, which requires complementing the statement text and running the code's function.

Goliath follows a simple routine to generate an exercise:

1. The type of the exercise is randomly picked within the range of possible options;
2. The value for each key in the statement is randomly chosen;
3. The code is adjusted to the values of the keys;
4. The answer alternatives are calculated;
5. The exercise is constructed from the three components (statement, code and answer alternatives).

The template carries more than just the statement and the code. It also contains other settings that are used to determine the possible exercise types, answer alternatives, correctness, level of difficulty and category (more on this in Section 5.3). One of these settings is a list of valid inputs for the function defined in the code. This list is used to generate exercises of the *input/output* type, which read "*What is the output of the following function when the input is ...?*". It is also used to check the student's answer by comparing it to the output of the function when fed with the valid input.

The incorrect alternatives for the other two types of exercises are constructed using the values of the `fail` clauses in the code. In the end, the presence or absence of valid inputs and `fail` clauses determine which exercise types can be generated. Listing 10 shows one example for each type of exercise based on the template of Listing 9. In these examples, `even` was chosen for the key `x` of the statement.

5.3. Goliath's interface

Goliath was implemented as an online application using a mix of technologies and languages (a Programming Cocktail) containing Python, Flask, Lark and MongoDB. Its interface follows the general workflows presented in Fig. 2, with a few added pages to manage users, control access to the several parts of the application, define assignments, and manage templates and exercises.

The main pages of the application are the *AI suggestion page*, the *template edit form*, the *exercises management page* and the *exercise page*.

[9] See Section 3.1 for the description of each type.

[10] This is the reason Goliath implements adapted versions of the original exercise types.

Listing 10 Three exercise types generated from the same template.

Code selection:

```
Which of these options is a function to remove
    all even numbers from a list of integers.

a) def remove_even_numbers(nums):
       return [n for n in nums if n % 2 == 0]
b) def remove_even_numbers(nums):
       return [n for n in nums if n % 2 == 1]
c) def remove_even_numbers(nums):
       return [n for n in nums if n * 2 == 0]
d) def remove_even_numbers(nums):x
       return [n for n in nums if n % 2 == 3]
```

Input/output:

```
What is the output of the following function
    when the input is [1, 2, 3, 4, 5].

def remove_even_numbers(nums):
    return [n for n in nums if n % 2 == 1]

Answer: _______
```

Code completion:

```
Which of these options complete the following
    function to remove all even numbers from a
    list of integers.

def remove_even_numbers(nums):
    return [n for n in nums if n % 2 == ___]

a) 1
b) 0
c) 2
d) 3
```

The AI suggestion page (Fig. 4) is the starting point to the creation of a new template in which the teacher inputs keywords for the KeyToText model to process and suggest an statement. After review, the teacher can send the statement for the CodeT5 to generate the associated code and fill the next page, the template edit form.

The template edit form (Fig. 5) allows the teacher to edit the statement, the code and the other settings for the template. It comes after the AI models have made their suggestions (or if they were skipped entirely). The settings for the template are:

- **Difficulty:** a difficulty level from easy to hard designed to guide students in approaching the easier exercises first. *Defined by the teacher;*
- **Tags:** a form of categorisation for the template. Specifies the general programming concepts that the exercises of this template will tackle;
- **Inputs:** a list of valid inputs for the function defined in the code. As explained at the end of Section 5.2, these inputs will be used to generate and verify the answers to exercises of the *input/output* type.

Finally, the exercise page (Fig. 6) allows the student to answer an exercise and obtain immediate feedback of his or her response. The image shows an exercise of type *code completion* begin answered.

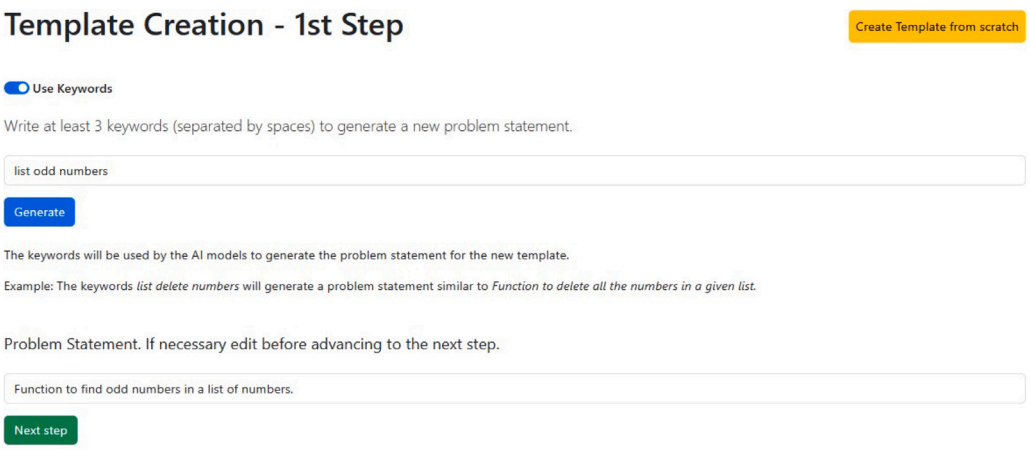

Fig. 4. Keyword input for the AI model.

5.4. Tests and feedback

In order to evaluate Goliath's functionality, a survey was conducted with teachers from Computer Programming background. The survey consisted in ten questions: two for establishing the respondent's background, seven to evaluate Goliath's features and usability — in a scale from 1 (terrible) to 5 (excellent) — and a final open question for additional remarks and feedback. The overall feedback was positive, with the results showing that the most requested area of improvement to be the initial generation of the statement and code by the models.

The questions, in order of appearance, were:

1. Programming experience (in years).
2. Experience in programming teaching (in years).
3. Regarding general usability, how do you rate the ease of navigation and interaction with the application?
4. How do you rate the ease of generating text (instructions and code) from the AI models?
5. How do you rate the quality of the text generation results (instructions and code) of the AI models?
6. How do you rate the way the templates are structured?
7. How do you rate the influence of the DSL in generating different exercises from the same template?
8. How do you rate the ease of using the DSL?
9. How do you rate the quality of the generated exercises?
10. In this section, you are asked to comment on any aspects not covered by the questions and to report errors/bugs that have appeared while using the application.

The survey was answered by 10 people and the results produced the charts in Figs. 7 and 8.

The first two questions revealed that the survey was answered by people with different levels of experience, including those that have never taught programming. This result indicates that the evaluation for the next seven questions are not skewed towards teachers, but a general overview of the application.

The results were mostly positive, especially in usability and ease of use of the main functionalities. The question with the lowest rating is the fifth (*Quality of the text generated by the AI models*), which indicates that the AI models have not been completely effective. Although, some respondents also indicated that the lower grades where due to instability of the models. The processes that run the models hung up the application a few times when the server was under heavier load from other sources. Nevertheless, the evaluations show that, despite a few inefficiencies, their preparation and the detailed processing of the input and output texts, resulted in average to good results.

The feedback obtained from the last question centred around suggestions of features that would complement the application, such as the possibility to create entire tests inside the application, and even a layout suitable for printing. Small bugs were also reported, which were addressed, making the application as consistent as possible.

5.5. Threats to validity

Besides the overall positive feedback, there are a few points that could be considered threats to validity of Goliath's evaluation:

- The number of respondents was relatively low, which could impact the overall statistical validity of the results;
- Given the time frame for the evaluation of Goliath's functionalities (a couple of months), the questionnaire focused on the more practical aspects of its use, so that the test users could effectively evaluate it. While there was a possibility answering with any kind of suggestions in the end of the questionnaire, no direct questions about Goliath's impact in the educational process were present. This limitation signals that, while the feedback was generally positive, it is not necessarily representative of Goliath's impact in computer programming education;
- The users were from two different universities in different countries (Portugal and Brazil). While these represent different contexts and reinforce the results' validity, it is not representative enough for overall generalisation.

Given what was asked of users in the evaluation process and the overall positive results, none of these points pose significant threat to Goliath's usefulness. The major concern relates to the necessity of a long to mid-term evaluation process, that could take several months, if not years. This effort for better validation of Goliath's impact in programming education is currently considered a future work, as presented in the conclusion of this paper (Section 7).

6. Exercises and the cognitive load theory

The Cognitive Load Theory was formally presented by John Sweller in his 1988 seminal paper [30], where he laid the groundwork for the development of an entire research field. By proposing a framework for the measurement of a person's cognitive burden when solving problems, Sweller demonstrated that, in fact, different types of problems (with their respective types of solutions) can be more or less taxing on the solver. In the same paper, Sweller also proposed a model for simulating and analysing cognitive load based on a production system framework that he had developed. Despite the fact that the main objective of the paper was to prove that open-ended problems are

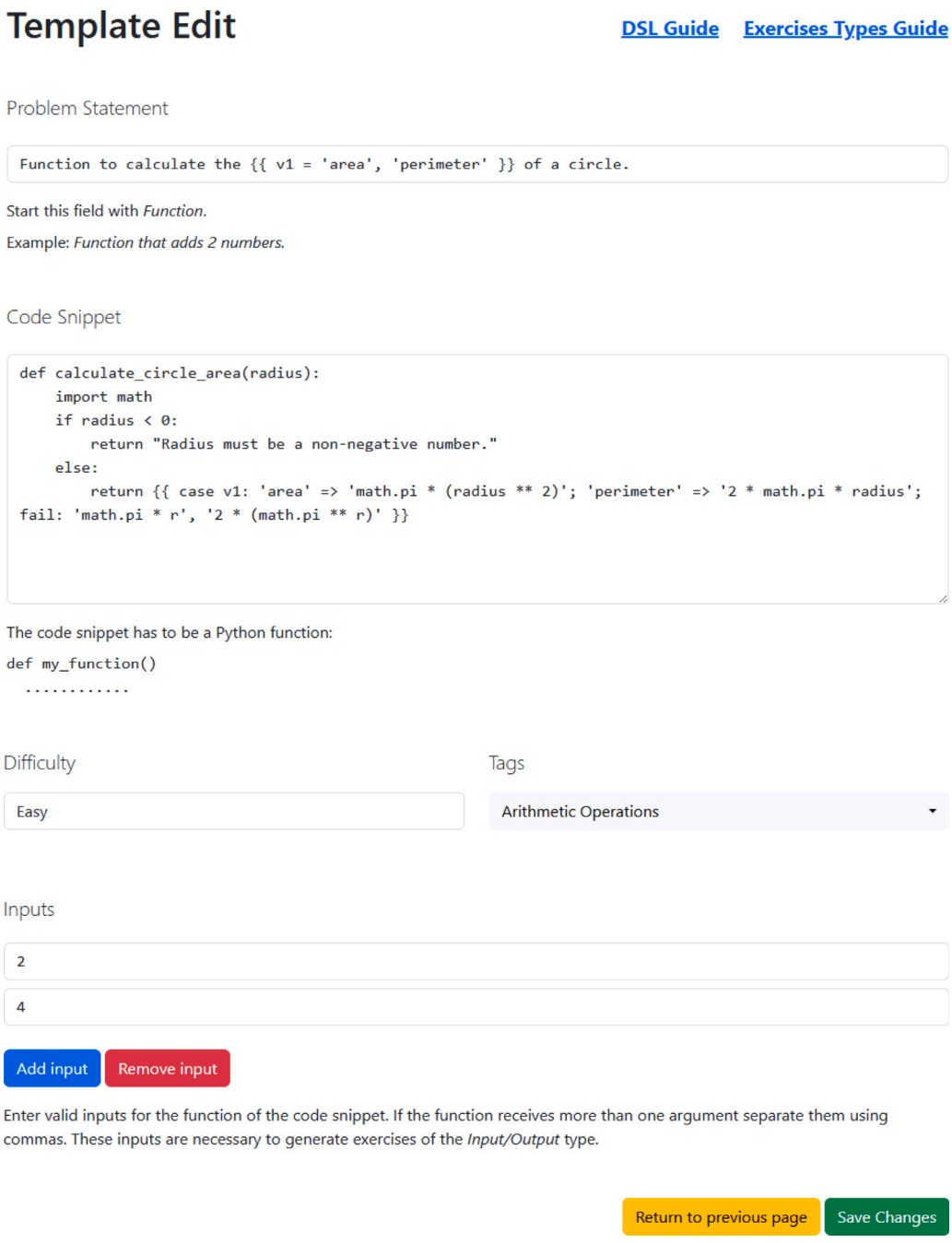

Fig. 5. Template edit form.

less cognitively burdening than their closed-ended counterparts, the formalisation of the Cognitive Load Theory concepts and the production system model became, since then, more prominent in the field of cognitive theory than those results *per se*.

6.1. The main concepts of cognitive load theory

Cognitive Load Theory is based on the idea that memory is divided between two types [30]:

Long-term Memory The portion of memory in which knowledge that has been constructed is stored for long-term use. It has no known limit and usually holds information for many years. It is the equivalent of non-volatile computer memory, such as hard drives and flash circuitry;

Working Memory Holds new information for the construction of knowledge. It is volatile, has a very small capacity, and determines how much information a person can work on at the same time. In computer engineering terms, it is equivalent to cache memory.

In its simplest, most direct form, *cognitive load* represents how occupied the working memory is, specially when solving problems and learning. This occupation may present itself through three basic forms of cognitive load [31]:

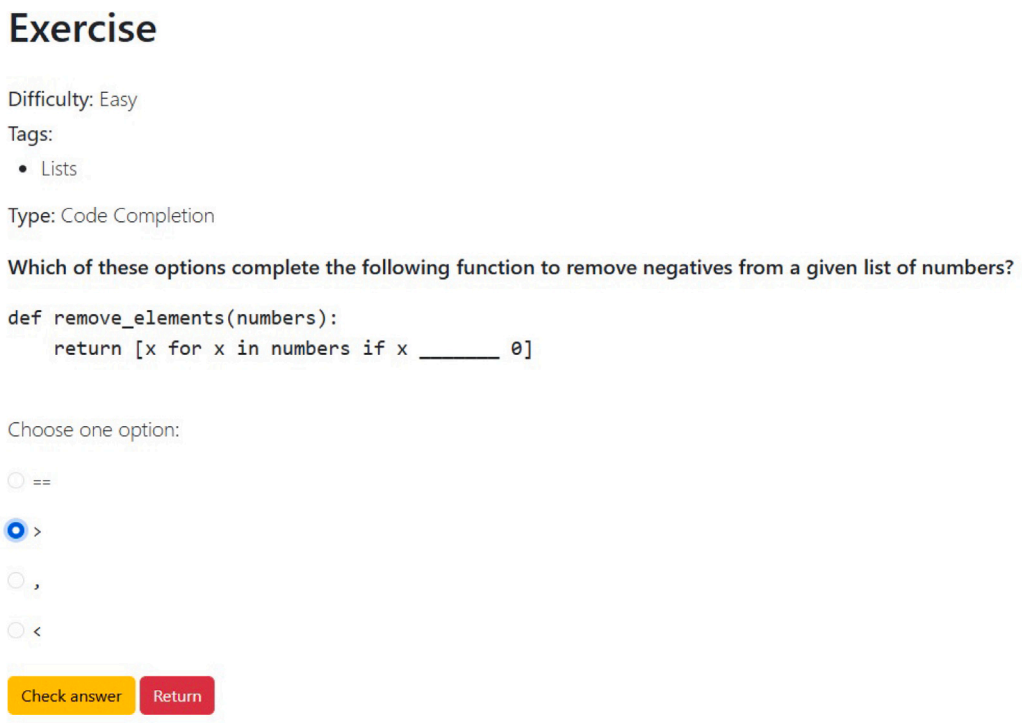

Fig. 6. Exercise page.

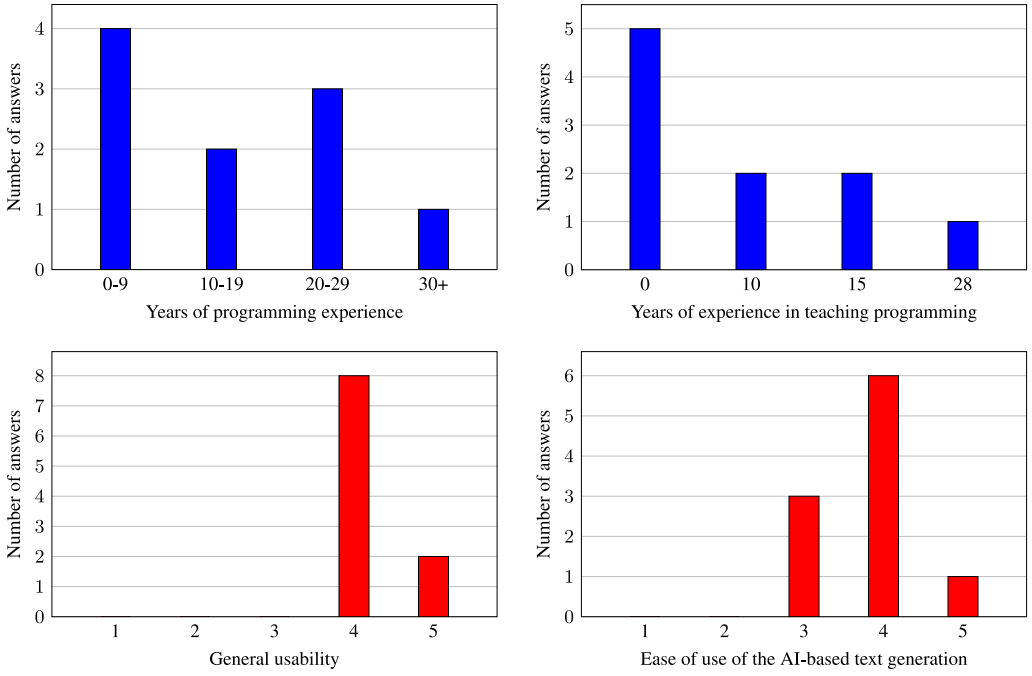

Fig. 7. Results from the survey.

Intrinsic The amount of information that is directly related to what is being processed. In a learning scenario, intrinsic cognitive load would entail the data, formulae, statements and options that are usually presented in an exercise;

Extraneous Any information that has no contribution to the main objective (learning or problem solving). Represents distractions that degrade the cognitive process and slow the transfer of useful information to the Long-term Memory. Again in the learning scenario, extraneous cognitive load would probably present itself in noises of all types (audio and visual), unnecessary graphs, pictures, or explanations;

Germane Represents the cognitive translation of information from the working memory to its long-term equivalent. It is closely related to its intrinsic relative and does not occupy working memory *per se*.

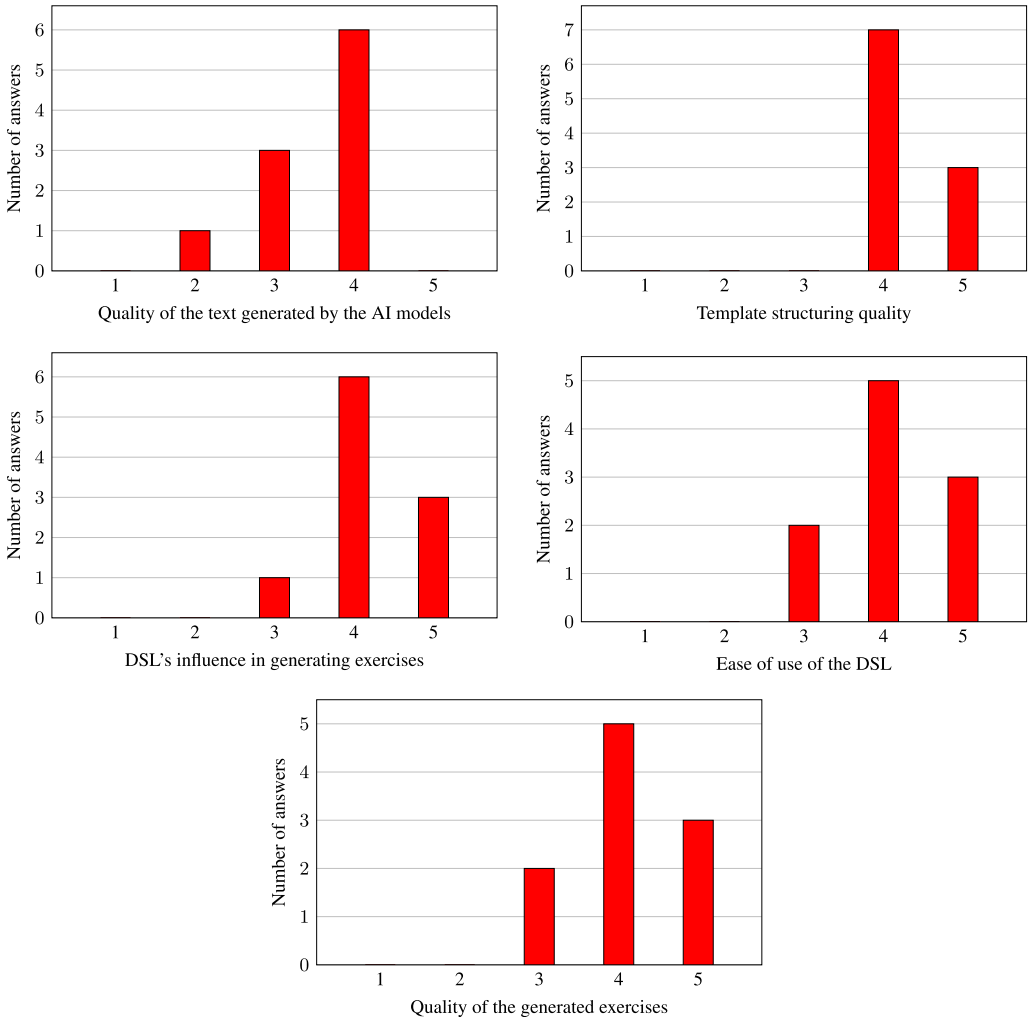

Fig. 8. Results from the survey (continuation).

Founded on this original idea, we created a simplified mathematical representation to express cognitive load (c) as a sum of both its intrinsic (i) and extraneous (e) parts:

$$|c(t)| = |i(t)| + |e(t)|$$

This (abstract) representation pertains c, i, and e as mappings of a moment in time t to a set of *cognitive units*[11] being held in working memory (M_W):

$$c, i, e : T \rightarrow U, U \subset M_W$$

The notation of set cardinality ($|c(t)|$, for example) is equivalent to the level of working memory occupation for each basic form of cognitive load (i.e. the number of cognitive units in their respective sets).

Given that there is an upper limit (L) to how many cognitive units the working memory can hold at any given time [32], the amount of available cognitive "space" for new information and processing ($a(t)$)[12] would be represented as:

$$a(t) = L - |c(t)|$$

[11] While there is no universal concrete definition for these units, they can be thought as representing information (in general terms, not merely data) in working memory [30].

[12] Meaning the amount of working memory that is "free".

6.2. Cognitive load in programming exercises

In order to create exercises that are lighter, more objective and efficient, teachers must strive to avoid distractions by minimising the use of elements that contribute to extraneous cognitive load, while choosing the right characteristics for snippets of code. In the mathematical sense, it would be ideal to reduce $e(t)$ to zero:

$$\lim_{t \to t'} |e(t)| = 0, \forall t' \in T$$

which would entail:

$$\lim_{t \to t'} |c(t)| = |i(t)|, \forall t' \in T$$

meaning that, ideally, all the cognitive load present when solving an exercise would be directly related to the problem and the topic in question, without any kind of distracting detail. This, of course, as symbolised by the limit, is purely ideal. Even brief and seemingly innocuous data, words, graphs or punctuation incur in some type of extraneous cognitive load.

While the main concepts of the Cognitive Load Theory are well established and have been consistently observed via their effects in learning and solving problems [31], measurements of cognitive load are highly heterogeneous. There are general ways of measuring cognitive load in an abstract from [31], but there is not a general formula to relate the cognitive load to the topic at hand, given its innate dependency on the context of measurement and the area of knowledge that

Table 2
Example of cognitive productions for computer programming.

#	Conditions	Actions	Strength
1	Algorithm to be implemented; It is a repetition.	New objective: loop implementation	1
2	Loop to be implemented; There is a precondition.	Register the *while* loop statement; Completed: loop implementation.	1
3	Loop to be implemented; There is a precondition; The condition relates to the number of iterations.	Register the *for* loop statement; Completed: loop implementation.	2
4	Loop to be implemented; There is a postcondition.	Register the *do...while* loop statement; Completed: loop implementation.	1
5	Loop to be implemented; There is a collection to be traversed.	Register the *for...in* loop statement; Completed: loop implementation.	1
6	Loop has been implemented.	Completed: algorithm implementation	1

is being analysed. A few approaches have been proposed for computer programming that could contribute to define which source code metrics relate to cognitive load, ranging from the number of lines of code, to indentation level and number of execution control statements. A valid relationship between certain characteristics of source code snippets with the cognitive load perceived by programmers has been found, but not yet proven [33].

Another possibility for measuring cognitive load when programming a new piece of code (or even an entire program) relies in a similar model as the one proposed by Sweller in his original paper: a Production System Model. In this model, each small logic step for solving a problem becomes a cognitive production that, when considered during the act of programming, raises intrinsic cognitive load [30]. This method takes into account characteristics that are not as immediate to calculate as lines of code while, on the other hand, allowing for a deeper analysis into the fundamental structures of a program implementation and how much these weight into the programmer's working memory [34].

Table 2 shows an example of the Production Model for the implementation of a repetition algorithm (loop). Each line forms a cognitive production, composed of conditions for its execution and actions that will be taken when it is fired. The strength of a production serves a purpose similar to that of operator precedence: when two or more productions are enabled (that is, their conditions are met, such as would happen to productions 2 and 3 in the case of a counting loop), the ones with greater strength are executed before those with lower strength. This can be used to model the better fit of a specialised construct, such as *for* in the case of counting loops.

Redundant structures contribute with more cognitive productions, extending the table. One example lies again in productions 2 and 3, as the behaviour of a *for* loop can be directly implemented through an equivalent *while* loop.[13]

A full table containing all steps for all *programming tasks*[14] would be composed of many more productions, similar in nature as the ones presented in Table 2. Beyond repetition though, other *programming tasks*, such as conditional execution, modularisation, database connection and input parsing would also be modelled. In any case, and independently of the context, these *tasks* would be inspected and broken down into their constituent steps, forming the lines of the table (i.e. the cognitive productions).

[13] In fact, if *break* is accounted for, every single type of loop can be implemented via *while*. This has not been added to the table as it would complicate the example and add little to no contribution to the main discussion.

[14] A *programming task* is akin to the implementation of an algorithm, ranging from an entire program, to a small piece of source code inside a function.

As presented in Section 6.1, germane cognitive load transfers cognitive units from working to long-term memory. This process is dependant on the units that are currently forming the intrinsic cognitive load. In a computer programming exercise, these units represent its data, the statements that the problem presents and the cognitive productions that are fired to form the solution. By controlling these elements (such as which productions are more relevant to learn the topic at hand) the exercise can reduce extraneous cognitive load by avoiding unrelated data, overcomplicated wording and redundant cognitive productions. This control, in turn, would allow more working memory space ($a(t)$) for intrinsic and germane cognitive load, consequently improving the chances of learning.

6.3. Goliath templates and cognitive load

It is common for teachers to create exercises intuitively, taking into account several details such as difficulty, length, type, topic, *etc.* In the specific case of creating exercise templates for Goliath, the production model and the types of cognitive load may form a structured and formalised approach that relies less in intuition and more on cognitive analysis.

Two main objective compose an overall strategy for creating exercise templates in Goliath that take into account the minimisation of cognitive load:

1. Removal of unnecessary elements that could become extraneous cognitive load in the solution;
2. Reduction of cognitive productions that the solution requires.

Both of these objectives need to be tailored to the intended goal of the exercise *per se*. If there is a need to intentionally complicate the exercise from a valid educational point-of-view, such objectives should be taken lightly. Otherwise, and from this point on in this discussion, it should be assumed that this strategy is considered positive and should be applied whenever possible.

Goliath's restrictions on the format of a template's statement and code already provide support for achieving the first objective. The seemingly restrictive rules for the composition of both elements (as explained in Sub Section 5.1.4) actually perform as an implicit guide of sorts, in which the teacher must focus its exercise in a simple function that can be easily parametrised via the DSL commands. Further extraneous care includes:

- The statement, even if parametrised using keys, should not require many results to be calculated nor features to be added to the code;
- Comments in the code should be kept to a minimum and never point features that are evident in the code itself;

Listing 11 Template that might generate a higher cognitive load version.

Statement:

```
Function that returns true if a number is {{x = 'the first of', 'repeated in'}} a global list "l".
```

Code:

```
def f(n):
    result = {{ case x: 'the first of' => 'l[0] == n';
                else: 'len([x for x in l if x == n]) > 1' }}
    return result
```

- Data that is relevant to the exercise should be presented only once, without redundancy, and as close to where it is needed as possible;
- Data that is relevant only to some variations of the exercise should either be conditionally placed or avoided at all;
- The code should be kept as minimalistic as possible, taking extra care to be both (a) directly related to the statement, and (b) maintain its simplicity in all of its variants.

The second objective is not as direct as the first one when it comes to recommendations. It requires not only care, but also a minimal model of the cognitive productions that can be fired during the solution of the exercise, to serve as an evaluation guide. Despite this fact, there are two main concerns to be observed:

- The choice of keys for the statement should be evaluated for all its possible combinations, as they might create exercise versions that require the use of more cognitive productions to understand the code;
- As one of the derivations of the simplicity rule previously mentioned, the code should avoid alternative means of implementation, such as different possibilities for loops, types or functions. Special care must be taken if the conditional placements deal with the main topic at hand, as it implies variations that Goliath will choose at random when presenting the exercise to the student. If a conditional placement adds to a topic that is already being trained by the exercise, more cognitive productions will be necessary to understand and solve the problem.

Listing 11 presents a template in which the choice of the key `x` might present higher cognitive load in one of its versions. The value `within` is very simple to implement and only deals with a conditional expression—the central topic for the exercise. The alternative value (`repeated in`) requires list comprehension, a higher order operation that implies repetition (even if indirectly).

6.4. Determining templates difficulty levels

Goliath allows the teacher to define a difficulty level to each template. This decision is usually done intuitively, just as the rest of the exercise construction. Taking into account the cognitive concepts explained in the previous sections, it can be better defined, even if still in relative form.

Each exercise deals not only with a specific programming topic (such as variable declarations, types, functions, *etc.*), but also with the details that the statement and the code bring to the problem. As exemplified by Listing 11 and its discussion, the keys and conditional placements generate variations that must be taken into account when considering which cognitive productions the student will require to solve the exercise.

Beyond the intuitive definition of the topic's challenges as a difficulty measure, the teacher can also take this cognitive load analysis as a metric for determining the template's difficulty. As an example, if a template generates exercises that deal only with one type of loop statement, and this restriction is directly presented to the student in the code, it is logical to assume it should be less cognitively taxing than one that allows many versions with different kinds of loops. However, not all cognitive productions are so directly identified. The conditional placement in Listing 11 allows for a solution that deals with list comprehension. While the operation itself can be understood as one cognitive production, the fact that the list will be traversed in some shape or form is undeniable, and this "background loop" will be considered by students that already know what list comprehension is. In fact, this presents an interesting situation where the exercise with list comprehension will be less cognitively burdening to students that only have a superficial understanding of the topic, than to those with deeper knowledge.

An exercise's type also contributes to its difficulty level, as some types of answers provide clues in the form of alternatives that others do not. Further and more in depth correlations between the exercise types and the cognitive productions are still under research, but it is safe to assume that they effect the difficulty levels perceived by students, as a similar rationale for mathematical problems has already been presented by Sweller [8].

In summary, the difficulty of a template should take into account the teachers intuitive notion of the hardship that students usually go through for a certain topic, but also how many cognitive productions the target audience for the exercises will need to understand it.

7. Conclusion

This extended paper presented Goliath, an online application to support semi-automatic generation of exercises. Goliath's DSL allows educators to create several versions of an exercise using only one template. This functionality provides replayability and diversification for students in order to aid them in the computer programming tasks.

As the main contribution beyond the original paper, this article also presented the Cognitive Load Theory, its main concepts and a new (and still in development) form of measuring cognitive effort in computer programming. Through the analysis of cognitive productions for typical programming tasks (such as declaring a variable, creating a loop, *etc.*) it is possible to go beyond intuitive construction of exercises, taking into account how many cognitive units one might need to solve them.

Future works include both new features and improvements to the existing ones. New modules that would allow Goliath to be applied in tests and other practice-oriented situations would be beneficial for teachers, supervisors and tutors. Also, features that provide greater independence to students would be ideal, such as the automatic generation of complete lists of exercises based on their history and background. The quality of both the statement and code generated by the AI models should be improved, in order to obtain more variety and complexity in their suggestions. In fact, given that more advanced models have been released since Goliath's original implementation, it would be interesting to evaluate possible improvements by integrating with newer versions of GPT, ChatGPT, Gemini and alike. Finally, reliability and efficiency in the communication between Goliath's internal parts and the models could also be improved.

CRediT authorship contribution statement

Tiago Carvalho Freitas: Writing – original draft, Validation, Software, Methodology, Investigation, Data curation, Conceptualization. **Alvaro Costa Neto:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Project administration, Methodology, Investigation, Conceptualization. **Maria João Varanda Pereira:** Writing – review & editing, Writing – original draft, Validation, Supervision, Project administration, Methodology, Investigation, Funding acquisition, Conceptualization. **Pedro Rangel Henriques:** Writing – review & editing, Writing – original draft, Validation, Supervision, Project administration, Methodology, Investigation, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

References

[1] A. Gomes, A.J. Mendes, Learning to program: Difficulties and solutions, in: Proceedings of the 2007 International Conference on Engineering and Education, ICEE, International Network on Engineering Education and Research, 2007, pp. 283–287, URL: http://icee2007.dei.uc.pt/proceedings/papers/411.pdf.

[2] J. Figueiredo, F.J. García-Peñalvo, Building skills in introductory programming, in: F.J. García-Peñalvo (Ed.), Proceedings of the Sixth International Conference on Technological Ecosystems for Enhancing Multiculturality, ACM, New York, 2018, pp. 46–50, http://dx.doi.org/10.1145/3284179, URL: https://dl.acm.org/doi/10.1145/3284179.3284190.

[3] M.J.V. Pereira, P.R. Henriques, Visualization/animation of programs in alma: Obtaining different results, in: Proceedings of the IEEE Symposium on Human Centric Computing Languages and Environments, 2003, pp. 260–262, http://dx.doi.org/10.1109/HCC.2003.1260242, URL: https://ieeexplore.ieee.org/document/1260242.

[4] R.R. Fenichel, J. Weizenbaum, J.C. Yochelson, A program to teach programming, Commun. ACM 13 (1970) 141–146, http://dx.doi.org/10.1145/362052.362053, URL: https://dl.acm.org/doi/10.1145/362052.362053.

[5] S.A. Robertson, M.P. Lee, The application of second natural language acquisition pedagogy to the teaching of programming languages: a research agenda, ACM SIGCSE Bull. 27 (4) (1995) 9–12, http://dx.doi.org/10.1145/216511, URL: https://dl.acm.org/doi/10.1145/216511.216517.

[6] F. Paas, Training strategies for attaining transfer of problem-solving skill in statistics: A cognitive-load approach, J. Educ. Psychol. 84 (1992) 429–434, http://dx.doi.org/10.1037/0022-0663.84.4.429.

[7] O.J. Ajogbeje, Enhancing classroom learning outcomes: The power of immediate feedback strategy, Int. J. Disabil. Sport. Heal. Sci. 6 (2023) 453–465, http://dx.doi.org/10.33438/ijdshs.1323080.

[8] J. Sweller, Cognitive load during problem solving: Effects on learning, Cogn. Sci. 12 (2) (1988) 257–285, http://dx.doi.org/10.1207/s15516709cog1202_4, URL: https://onlinelibrary.wiley.com/doi/abs/10.1207/s15516709cog1202_4.

[9] T.C. Freitas, A. Costa Neto, M.J.V. Pereira, P.R. Henriques, Goliath, a programming exercises generator supported by AI, in: M. Bolanowski, M. Ganzha, L. Maciaszek, M. Paprzycki, D. Ślęzak (Eds.), Proceedings of the 19th Conference on Computer Science and Intelligence Systems, FedCSIS, in: Annals of Computer Science and Information Systems, vol. 39, IEEE, 2024, pp. 331–342, http://dx.doi.org/10.15439/2024F8479.

[10] M.V.P. Almeida, L.M. Alves, M.J.V. Pereira, G.A.R. Barbosa, EasyCoding: Methodology to support programming learning, in: R. Queirós, F. Portela, M. Pinto, A. Simões (Eds.), in: Open Access Series in Informatics (OASIcs), vol. 81, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, pp. 1–8, http://dx.doi.org/10.4230/OASIcs.ICPEC.2020.1, URL: https://drops.dagstuhl.de/opus/volltexte/2020/12288.

[11] Python Tutor, Python tutor, 2023, URL: https://pythontutor.com.

[12] Beecrowd, Beecrowd, 2024, URL: https://beecrowd.com.

[13] A. Rios, J.L. Pérez de la Cruz, R. Conejo, SIETTE: Intelligent evaluation system using tests for TeleEducation, in: WWW-Based Tutoring Workshop At 4th International Conference on Intelligent Tutoring Systems, 1998, URL: https://www.siette.org.

[14] A. Zeileis, R/exams, 2023, URL: https://www.r-exams.org.

[15] M. Bower, A taxonomy of task types in computing, SIGCSE Bull. 40 (3) (2008) 281–285, http://dx.doi.org/10.1145/1597849.1384346.

[16] A. Ruf, M. Berges, P. Hubwieser, Classification of Programming Tasks According to Required Skills and Knowledge Representation, vol. 9378, 2015, http://dx.doi.org/10.1007/978-3-319-25396-1_6.

[17] N. Ragonis, Type of questions - the case of computer science, Olymp. Inform. 6 (2012) 115–132.

[18] A. Simões, R. Queirós, On the Nature of Programming Exercises, in: R. Queirós, F. Portela, M. Pinto, A. Simões (Eds.), First International Computer Programming Education Conference, ICPEC 2020, in: OpenAccess Series in Informatics (OASIcs), vol. 81, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2020, pp. 24:1–24:9, http://dx.doi.org/10.4230/OASIcs.ICPEC.2020.24, URL: https://drops.dagstuhl.de/opus/volltexte/2020/12311.

[19] E. Reiter, R. Dale, Building Natural Language Generation Systems, Cambridge University Press, 2000.

[20] E. Reiter, R. Dale, Building applied natural language generation systems, Nat. Lang. Eng. 3 (2002).

[21] A. Celikyilmaz, E. Clark, J. Gao, Evaluation of text generation: A survey, 2020, CoRR, abs/2006.14799, arXiv:2006.14799, URL: https://arxiv.org/abs/2006.14799.

[22] G. Bhatia, Keytotext, 2023, URL: https://github.com/gagan3012/keytotext,

[23] Y. Wang, W. Wang, S. Joty, S.C. Hoi, CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, in: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 2021, pp. 8696–8708, URL: https://aclanthology.org/2021.emnlp-main.685.

[24] T.C. Freitas, A. Costa Neto, M.J.V. Pereira, P.R. Henriques, NLP/AI Based Techniques for Programming Exercises Generation, in: R.A. Peixoto de Queirós, M.P. Teixeira Pinto (Eds.), 4th International Computer Programming Education Conference, ICPEC 2023, in: Open Access Series in Informatics (OASIcs), vol. 112, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2023, pp. 9:1–9:12, http://dx.doi.org/10.4230/OASIcs.ICPEC.2023.9, URL: https://drops.dagstuhl.de/opus/volltexte/2023/18505.

[25] w3resource, Python exercises, practice, solution, 2023, URL: https://www.w3resource.com/python-exercises/.

[26] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, P.J. Liu, Exploring the limits of transfer learning with a unified text-to-text transformer, J. Mach. Learn. Res. 21 (1) (2020).

[27] J. Austin, A. Odena, M. Nye, et al., Program synthesis with large language models, 2021, arXiv preprint arXiv:2108.07732.

[28] H. Husain, H. Wu, T. Gazit, M. Allamanis, M. Brockschmidt, CodeSearchNet challenge: Evaluating the state of semantic code search, 2019, CoRR, arXiv:1909.09436, arXiv:1909.09436, URL: http://arxiv.org/abs/1909.09436.

[29] E. Shinan, Lark, 2023, URL: https://lark-parser.readthedocs.io/en/stable/.

[30] J. Sweller, Cognitive load during problem solving: Effects on learning, Cogn. Sci. 12 (2) (1988) 257–285, http://dx.doi.org/10.1207/s15516709cog1202_4, URL: https://onlinelibrary.wiley.com/doi/abs/10.1207/s15516709cog1202_4.

[31] J. Sweller, J.J.G. van Merriënboer, F. Paas, Cognitive architecture and instructional design: 20 years later, Educ. Psychol. Rev. 31 (2019) 261–292, http://dx.doi.org/10.1007/s10648-019-09465-5.

[32] F. Paas, J.J.G. van Merriënboer, Cognitive-load theory: Methods to manage working memory load in the learning of complex tasks, Curr. Dir. Psychol. Sci. 29 (4) (2020) 394–398, http://dx.doi.org/10.1177/0963721420922183.

[33] A. Abbad-Andaloussi, On the relationship between source-code metrics and cognitive load: A systematic tertiary review, J. Syst. Softw. 198 (2023) 111619, http://dx.doi.org/10.1016/j.jss.2023.111619, URL: https://www.sciencedirect.com/science/article/pii/S0164121223000146.

[34] Y. Wang, Cognitive complexity of software and its measurement, in: 2006 5th IEEE International Conference on Cognitive Informatics, vol. 1, 2006, pp. 226–235, http://dx.doi.org/10.1109/COGINF.2006.365701.