

UNIVERSITÉ DE LIÈGE



MASTER THESIS

CIVIL ENGINEERING IN COMPUTER SCIENCE

**Implementing an Automatic Pointer Exercise Generator in
CAFÉ 2.0**

Author :

Valentin Baum

Promoters :

Prof. Benoit DONNET

Prof. Pascal FONTAINE

Academic Year: 2022-2023

Abstract

This Master's thesis addresses the issues students face when learning about pointers in introductory computer programming courses. To improve their understanding and practice, a structured approach to generating a variety of pointer exercises is proposed. The framework aims to provide students with opportunities for regular assessment, constructive feedback and practice. At the same time, it provides a tool for teachers to create exercises efficiently and discourage academic dishonesty.

The study outlines the development of a pattern template-based exercise generation framework, describing the process from analysis to implementation. In addition, the research explores the impact of ChatGPT as an educational tool for programming exercises.

This work aims to enhance the learning experience of students in programming and to inspire advances in exercise generation for diverse subjects.

Acknowledgements

I want to express my sincere gratitude to Pr. Donnet, Pr. Fontaine, and Ms. Brieven for their exceptional support, availability, and guidance throughout the year. Their insights and assistance have been instrumental in the completion of this work.

I am also grateful to Anthony Ciappo for his valuable insights and advice.

Lastly, I would also like to thank my family for their unconditional support.

Contents

1	Introduction	4
2	State of the art	6
2.1	Exercises generation	6
2.1.1	Context-Free grammar	6
2.1.2	Templates	7
2.1.3	Mutation	7
2.1.4	Construction from solution	8
2.1.5	Natural language processing	8
2.2	Feedback	9
2.2.1	Test-based feedback	9
2.2.2	Path construction	10
2.2.3	Transformation model	11
2.2.4	Peer-feedback	11
2.3	Methods used in this thesis	11
3	Analysis of existing exercises in INFO0046	12
3.1	General Structure	12
3.2	Exercise	14
3.3	Existing exercises structure	18
3.3.1	Easy expressions	18
3.3.2	Arrays	19
3.3.3	Pointer arithmetic	19
3.3.4	Incrementation and decrementation	20
3.3.5	Cast	20
3.3.6	Complex dereference	21
3.3.7	Subtle expressions	21
3.3.8	Char	21
3.4	Statistics	23
3.4.1	Expressions	23
3.4.2	Declaration	23
3.4.3	Memory state	24

4	Syntax	26
4.1	General syntax	26
4.1.1	Easy	26
4.1.2	Arrays	27
4.1.3	Pointer arithmetic	27
4.1.4	Dereferencing of more complex address	28
4.1.5	Incrementation and decrementation	28
4.1.6	Cast	29
4.1.7	Subtle	29
4.1.8	Functions	30
4.2	Pattern	30
4.2.1	Easy	30
4.2.2	Pointer arithmetic	31
4.2.3	Incrementation decrementation	32
4.2.4	Cast	32
4.2.5	Array	32
4.2.6	Char	32
4.2.7	dereferencing	33
5	Exercise generation	34
5.1	Process	34
5.1.1	Pattern creation	36
5.1.2	Variable creation	37
5.1.3	Memory creation	38
5.1.4	Selection	39
5.2	Testing	42
6	Feedback generation	45
6.1	Different types of errors	45
6.1.1	Variable	45
6.1.2	Unary operator	45
6.1.3	Incrementation and decrementation	46
6.1.4	Arithmetics	46
6.1.5	Cast	46
6.2	Implementation	47
6.2.1	Correct answer	47
6.2.2	Feedback	47
7	Guide	49
7.1	Global architecture of folders	49
7.2	How to run the program	52
7.3	Adapting the parameters	52
7.3.1	Global parameters	52
7.3.2	Variables parameters	52

7.3.3	Category repartition parameters	53
7.3.4	Memory parameters	53
7.4	Add patterns	53
7.5	Modifying the feedback	55
8	Result analyses	56
8.1	Repartition between the pattern	56
8.2	How different are the exercises for the same pattern?	60
8.3	Speed of the program	63
9	Can ChatGPT solve the exercises?	64
9.1	Introduction	64
9.2	Prompts	64
9.3	Result	65
10	Conclusion	68
A	Testing	70
A.1	C program to test feasibility	70
B	ChatGPT	73
B.1	Prompt 1	73
B.2	Prompt 2	74
B.3	ChatGPT output with prompt 1	75
B.4	ChatGPT output with prompt 2 (first try)	77
B.5	ChatGPT output with prompt 2 (second try)	78

Chapter 1

Introduction

Each year, first-year bachelor students in computer science struggle with the concept of pointers during the “Introduction to computer programming” course. While they may perform well on online tests conducted before the exam, their performance during the actual exam is often not satisfactory. This raises questions about whether some students may be cheating or collaborating to find solutions for their online tests. It is also possible that students rely on the help of the teaching materials without fully grasping the concepts or even using external tools like chatGPT.

Regardless of the reasons, it is obvious that many students have not fully understood the concepts related to pointers and face difficulties during exams. To address this issue, this project proposes to automatically generate pointer exercises. The primary goals of this project are threefold:

1. To mitigate the possibility of cheating by providing a different set of exercises with the same difficulty level to each student.
2. To offer students the opportunity to practice and improve their understanding of pointers on the Cafe platform. By generating practice exams automatically, students can regularly assess their understanding of the subject.
3. To help students improve by providing feedback that assists them in identifying their mistakes and areas of improvement.

In the initial phase of this work, we conduct a literature review on exercise generation and feedback techniques to gain insights into the various methods used in this field. This review provides us with a comprehensive understanding of existing approaches and best practices.

After the literature review, we perform an in-depth analysis of the composition of pointer exercises. This analysis helps us identify the key elements and essential aspects that need to be considered while generating exercises. With this knowledge, we devise a general syntax for creating exercises.

Next, we examine the implementation details of the program using approaches found in the

literature review. This includes the generation of exercises and the feedback mechanism. A user guide is also provided to assist instructors in using the program effectively.

Finally, we explore the extent to which ChatGPT can help students in their learning process. Understanding how ChatGPT can provide assistance or clarification to students gives us valuable insights into its role in supporting student learning.

By combining these components, this work aims to provide a comprehensive framework for generating pointer exercises, offering valuable feedback to students, and leveraging the potential of ChatGPT to enhance the learning experience.

Chapter 2

State of the art

With the increasing number of students in classrooms and the popularity of online courses, automated exercise generation and assessment have gained attention as a means of reducing teacher workload and providing additional practice opportunities for students. This chapter first looks at different techniques for creating exercises and then analyses strategies for providing meaningful feedback to students.

2.1 Exercises generation

2.1.1 Context-Free grammar

One way of generating exercises is to use context-free grammar. This method breaks down exercises into a set of production rules and then randomly applies the different rules to create exercises.

The concept of generating code with context-free grammar was originally developed for testing compilers and was first introduced in [15]. The code was generated randomly to evaluate the correct behaviour of the compiler. [37] refined the method by assigning probabilities to syntax rules. Each choice is mapped to a probability, derived from historical data, to generate similar output. By inverting these probabilities (via $1/p$ and normalisation), unusual outputs are generated. Furthermore, by only considering data that causes errors for the probabilities, the program generates code that is more likely to cause errors.

This method has been adapted for exercise generation. In [4], the main concept of exercise generation is introduced. It uses a “meta-grammar” approach where each standard production rule is associated with a template. The production rule generates the exercise solution, while the associated template generates the corresponding problem statement.

The implementation of probabilistic grammar can be seen in [38]. It describes the generation of exercises in which students translate statements into code. In this case, the grammar is used to generate the solution code, which is then transformed into a statement. Test cases are also

generated to evaluate the students' solutions. Another example can be found in [8], where the grammar is applied to generate nested loop exercises. This work provides a description of the grammar and algorithms for generating exercises.

2.1.2 Templates

Another method is to standardise exercises into a template format, where various options are deliberately left to be randomly selected at the generation.

In [27], beginner-level programming exercises are generated. The exercise consists of a code and a statement question where the student is asked to explain the behaviour of the code. The program uses predefined templates, one for the code and one for the statement, both of which contain replacing marks. During the generation process, these marks are replaced by other "lower-level" templates until all marks have been removed.

[12] generates exercises to teach the topic of loops. The idea is to provide a grid with coloured cells that create a specific shape. The student is asked to reproduce this shape using the correct "for" loop condition. Pre-defined shapes can have any position and size on the grid. Each shape is in a separate module consisting of two parts, a language-independent part for creating the shape and a language-dependent module responsible for creating the problem statement in its specific language. This separation allows the program to be used in different languages.

In the field of databases, [18] generates exercises where students have to write SQL queries. Students give a relational database model to the program. The program then analyses this model and selects some predefined queries that can be used for this model. The program then adapts the queries to the database and translates them into English statements.

For the generation of proof problems, [35] presents an algorithm that automates the generation of problems consisting of verifying the equality between the left and right sides of an equation. First, the exercises are analysed by the teacher in order to deduce different patterns. These patterns are then transformed into a sequence of choice functions that the program manipulates and chooses to create exercises. The system uses functional constraints to discard expressions during choice selection and relational constraints to remove the expressions once they have been generated.

2.1.3 Mutation

Another method of generating exercises is to start from existing exercises and apply mutation rules to create new exercises.

In [31], a system for generating exercises for "embedded systems" courses is proposed. The exercises consist of a finite state machine (the model), specifications (properties) and the trace to be verified. The approach consists in defining mutation rules that can be applied to existing exercises. By changing the number of mutations, the level and difficulty of the generated exercise can be adjusted.

Alvin et al. [5] present a method for generating demonstration exercises for Euclidean geometry. The method first generalises existing exercises using hypergraphs and based on certain assumptions (i.e. the mutation rules), generates related exercises.

2.1.4 Construction from solution

This method starts from the exercise solution and applies some rules to construct the statement.

For example, in the field of Euclidean geometry, [23] presents a method for generating exercises. They consider the exercises as a graph path, where the nodes represent states and the edges represent formulas or axioms to progress from one stage to another. The goal is to find a path from the initial state to the final state. Conversely, to generate exercises, a backward approach is used, starting from the final state and building the graph tree by applying the different rules. The nodes generated can be selected as problem statements. The generated problems are then ranked, according to their depth. The deeper, the more solving steps are required. The difficulty of the rules is also taken into account. Similarly, [2] presents an exercise generator for an introductory logic course.

2.1.5 Natural language processing

Natural Language Processing (NLP) is a vast field for the generation of exercises. The first use of NLP is to generate questions based on text. It analyses the sentences and generates questions.

[39] simulates a psychologist responding to a patient. It first analyses the statement to find the most important keywords and captures the context around that word. It then selects the appropriate rule transformation to generate the question. [40] uses the same method and adapts it for educational purposes. It generates questions to assess reading comprehension.

In the same way, the generation of gap-filling questions identifies key elements in sentences and removes them[1]. [7] improves this approach by better identifying the key element. It trains the model on a dataset of questions scored by humans.

Multiple choice questions are another category generated using NLP techniques. It creates questions about the main aspects of a document and generates the correct answer as well as other answers called distractors (answers that are close to the solution).[22]

An interesting paper [20] improves the question generation by using together gpt-2[26] and Bert[9]. It uses GPT-2 to generate questions and BERT to answer them. Backtracking is used to adjust the weighting model of the question generation.

The same question generation techniques can also be applied to multimedia content, as demonstrated in [36].

The use of NLP techniques generates accurate questions but human validation is often required. It remains faster than generating them manually[3].

Recent studies have explored the use of OpenAI’s Codex to generate maths and programming exercises. [11] generates mathematics questions, including mathematics for computer science. It provides the codex with existing exercises and instructs it to generate the next one automatically. They show that the difficulty level was similar to human-generated questions and also indistinguishable. In the same vein, [21] also uses the OpenAI codex to generate math exercises, but adapts the difficulty level to the student. It first assesses the student on exercises and then generates customised exercises based on their strengths or weaknesses. [33] explores the use of large language models (openAI codex) to generate computer exercises. The program generates exercises consisting of a problem description, a solution and test cases. The generation script consists of a problem statement, its solution, test cases and thematic keywords. The codex adapts the exercises according to the thematic keywords (e.g. Harry Potter, Star Wars) in order to provide students with appealing exercises. The generated exercises were found to be both “sensible and novel”, providing an opportunity to create new exercises. Only 30% of the generated exercises were correct but the other exercises had only minor problems in boundary or test cases.

2.2 Feedback

Computer science courses usually require students to write coding programs. The first and most basic way to get feedback is for the teacher to manually test the different programs[19]. This can be very time-consuming as the number of students increases, hence the will to create automatic assessment tools to accommodate the masses of students. This saves time for the teacher[14][17], allows to obtain some metrics about student understanding[16], allows distance teaching[17], and provides the objectivity of the machine compared to humans[14].

2.2.1 Test-based feedback

Different automatic systems could be seen at Rensselaer Polytechnic Institute in 1960[17], at Stanford University since 1961[14] and at the University of Sydney in 1966[16].

Considered the first papers on the subject in [14][10], Hollingsworth [17] introduced an automatic grader which evaluated machine language programs written by students on punched cards. The system tested these programs, compared their output with the expected results, and gave feedback either “wrong answer” or “problem complete”. The program was compact (comprising only 108 instructions) but it also had limitations, including the need for manual intervention after errors and security vulnerabilities as students could manipulate the grader.

[14] presented a more sophisticated grader. Test data was randomised to prevent students from predicting outputs, checking the program’s running speed and storage usage to evaluate the efficiency, safeguards were implemented to deal with potential infinite loops and security was improved.

Instead of just returning a binary output “true” or “false”, Hext and winnings[16] suggest keeping a record of the gradings. It can list the grades or the number of attempts for each student and

for each exercise. These statistics show the teacher the progress of the class and can point out difficulties.

These programs are considered to be the first generation of assessment systems as they require a great deal of knowledge to modify the compiler and the operating system.[10].

In the second generation, identified as tool-oriented, new tools for the operating system appeared and facilitated the creation process[10]. An example of this generation can be seen in [32], which automatically evaluated Scheme programs submitted by the student via email. It implements advanced features, such as keyword search to detect forbidden words, analysis of the program structure to find subpatterns and comparison of the abstract syntax tree to detect plagiarism. Another example is [28] which creates a Unix package called “Try”. It is designed to evaluate any programming language to facilitate the process of creating automated assessments.

The latest generation of automated grading systems, described as web-oriented, use graphical interfaces and increase the robustness of the testing process [10].

2.2.2 Path construction

The path construction approach creates a graph where each node represents a resolution state, and the edges are the changes required to move to the next state. The goal is to identify the current state and find a path to the correct solution.

Barnes et al. [6] developed an automatic hint generation system for logic resolution problems. It uses the student’s historical data to construct the graph. The nodes represent all the states students have visited and the edges are the axioms used. A reward value is assigned to each state. It presents two methods for the rewards, either using the proximity to the solution or using the student frequency. It then computes the shortest path result for the first method to find the minimum number of steps and the second method to find the most likely path students could take.

However, [29] identified some limitations with this approach. Students had to exactly match one of the existing states to receive hints and because it is based on historical data, it is difficult to generate hints for new problems. To address these issues, [29] abstracts the states and groups similar states into clusters. Hints are generated based on these clusters. It used this approach for computer science programs. Each state is an abstract syntax tree (AST) of the program. It designs functions to compare the state between them and between student codes. Once the current state is identified, it finds the next goal state and compares it to AST together to generate the hint.

Similarly, [25] presents an algorithm for open-ended programming questions that also relies on abstract syntax trees. They evaluated it on student submissions and show that the program relatively frequently guides students closer to a high-quality solution. These results are promising but there were only made on a single assignment.

2.2.3 Transformation model

One approach to providing hints to students is to use error transformation models. In this method, the teacher provides the correct solution along with a list of transformation errors that correspond to common student mistakes and their corresponding corrections.

The system applies these correction rules to the student's solution until it reaches the minimum transformation to obtain the correct solution. Depending on which rules are used, it sends appropriate feedback[34, 30]. This method is easier to implement than path construction and is powerful when the program is almost correct but it may fail when it is far from the solution.[24].

To address this problem [24] divides its program into three categories. The first category checks for semantic errors. It uses pattern matching to identify the errors. The second category is for almost correct programs, it uses the transformation as it works well in this case. The third category is for programs that are not close to the solution. It checks the condition statement. It compares the program condition with the correct solution condition. It helps students to consider overlooked cases or conditions they have not checked.

2.2.4 Peer-feedback

As peer-feedback plays an important role in the student learning process in classroom courses, it is important to incorporate student help in online platforms. It helps the struggling student to receive hints of improvement and it helps the helper student to have critical reasoning on his knowledge to construct a good answer[13].

2.3 Methods used in this thesis

This work is based on the pattern-based method for generating exercises. We analyse previous exercises to identify patterns where choices are left open (e.g. variables, operators) and the program makes appropriate choices based on these patterns.

For assessment and feedback, we use a test-based approach where we compare the student's answers with the correct answers. We also check whether the student's solution matches one of the solutions obtained by common student errors and provide feedback accordingly. This method is based on transformation models where we define transformation rules representing student errors and apply these rules when computing solutions.

Chapter 3

Analysis of existing exercises in INFO0046

In this chapter, we analyse the exercises provided to students in the course “Introduction à la programmation”(Introduction to Programming) given at the University of Liège, Belgium. We have access to past exams from the three sessions (January, June and August) from 2019 to 2022, excluding June and August 2019. Additionally, we have challenge exercises from 2016 to 2022, excluding 2019. These challenge exercises were given throughout the year to assess the student’s understanding.

First, we explain the general structure and composition of an exercise followed by an example. Then, we attempt to deduce the general categories. Finally, we analyse further all the exercises to have a better understanding of their characteristics, and frequencies within each category.

3.1 General Structure

Each exercise is made of three parts: the declaration of all variables, the state of the memory, and various expressions the student has to solve.

First, the declaration (Figure 3.1) declares all the variables required for the exercise, specifying their data types and sometimes initial values.

```
1 int var1;  
2 int *var2 = &var1;  
3 int var3[] = {};  
4 char *var4 = " ";
```

Figure 3.1: General view of the declaration

The second part, the memory state (Figure 3.2) assigns addresses to the variables and uninitialized variables receive values. It is presented in three columns. The first column indicates the variable names, the second column indicates the memory addresses and the last column displays the values stored in the memory.

The state of the memory also provides a memory interval [lower bound, upper bound] that is accessible at all times. If memory access is attempted outside this interval, it results in a segmentation fault (symbolized by SF). Throughout the exercises, the memory state remains consistent, implying that the memory is reset between each expression.

Variable :	address :	value :
	lower bound	...

	upper bound	...

Figure 3.2: General view of the memory state

The last part of an exercise is the various expressions (Figure 3.3). Each line corresponds to an individual expression. The figure provides an abstract overview of all expressions with different colours representing the different categories of expressions obtained in section 3.3.

The goal for the students is to evaluate each expression based on the memory state and declaration.

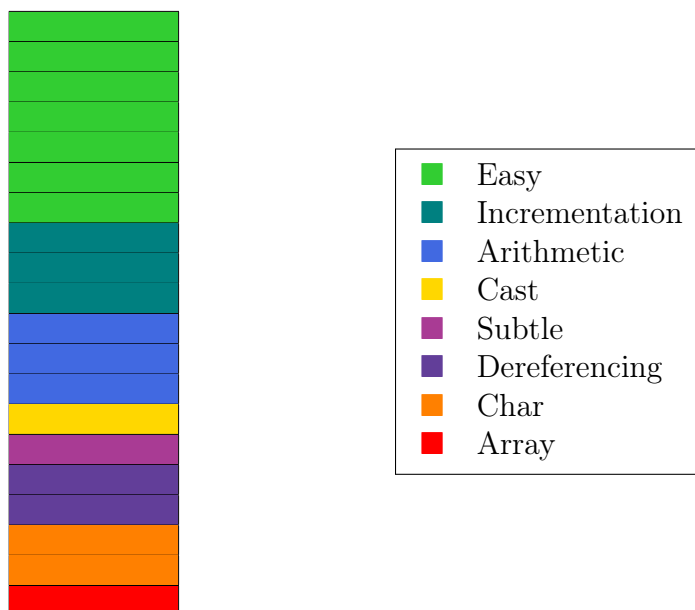


Figure 3.3: General form of the expressions

3.2 Exercise

To provide a more concrete illustration of the general structure, let's consider an example from the August 2022 exam.

Figure 3.4 is the declaration. Five variables are declared: one int, two int pointers, an int array and a char array. Most of these variables are initialized, such as the variable h which contains the address of tab[1]. However, the variable g does not have an explicit initialization value and is determined with the memory state (Figure 3.5).

The variable g is indeed present in the first column of the memory state, and we can deduce that the corresponding memory address is 604 and the associated value is 600.

The memory interval for this exercise is $[272, 616]^1$

```

1 int g, *h = &g, m = &g+3;
2 int tab[] = {6, 60, 600, 6000};
3 *h = tab[1];
4 char *p = "pop";

```

Figure 3.4: Example of a declaration (August 2022)

	616	278
	612	300
	608	4
g	604	600
	600	604
		...
p[0]		
		...
	282	6000
	278	600
	276	60
tab[0]	272	6
	272	5

Figure 3.5: Example of a memory state (August 2022)

The figure below contains all the expressions a student had to evaluate.

¹In this paper, we use the same notation as in the Introduction to Programming course, although the actual memory interval is $[272, 619]$.

<code>g</code>
<code>m</code>
<code>h</code>
<code>*m</code>
<code>** (int **) *m</code>
<code>&g</code>
<code>*m</code>
<code>m-h</code>
<code>m-tab</code>
<code>m-tab[0]</code>
<code>*(tab+3)</code>
<code>h+g</code>
<code>*(&g+12)</code>
<code>tab['g' - 'h']</code>
<code>*(tab+1)& (++*h)</code>
<code>(long *) m - 2</code>
<code>(long *) (m - 2)</code>
<code>p[p[2]-p[1]]</code>
<code>printf(%s, p+1)</code>
<code>*tab++**h++</code>

Figure 3.6: Example of expressions (August 2022)

Here is an example of the reasoning that students must employ to solve these expressions :

- g : The expression is straightforward to evaluate. It was declared as an integer and the result of this expression is simply the value contained in the variable. By referring to the memory state for the variable g on the first column, we can find the corresponding address as 604 and the value as 600. The result of the expression is then 600.

	616	278
	612	300
	608	4
g	604	600
	600	604
		...
		...
	282	6000
	278	600
	276	60
	272	6
	272	5

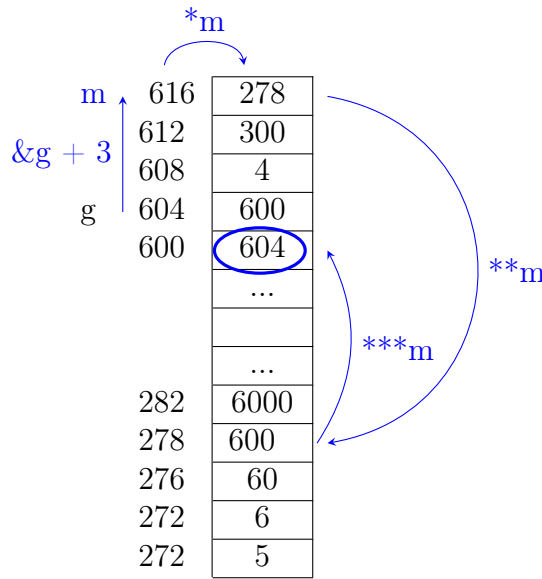
- $**(int**)m$: The variable m is a pointer to an integer, declared as `int *m = &g+3`. This means that the variable m contains the address resulting from the addition of the address of g ($\&g$) and the integer 3. It means that the resulting address is three memory blocks above the address of g . Since the size of a memory address is 4 bytes, the expression can be rewritten as $\&g+3*4$, which evaluates to $604 + 3 * 4 = 616$.

Now that we have the value of m (616), we can compute the next operation, which is $*m$. This operation dereferences the pointer m and retrieves the value stored at the memory address 616. Looking at the memory state, we find that $*m = 278$.

The next operation is a cast to $(int **)$. In this case, the value obtained from $*m$ is an integer but we need to interpret it as a pointer to pointer to be able to dereference further. The cast changes the interpretation of the integer to an address.

The operation $*(int **)m$ is a dereferencing of the address contained in $*m$ and the value at address 278 is 600.

Finally, we perform another dereferencing operation to obtain 604 as the final result.



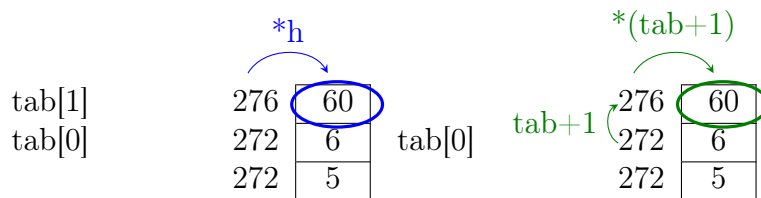
- $*(tab+1) \& (++*h)$: This expression contains the operator “&”. In the previous example, this operator was a reference operator (to obtain the address of a variable) but here, it is the symbol of a binary “and” operator because it is applied to two operands.

The resolution can be divided into three steps, first, we compute the left and right parts and then perform the binary “and” operation.

The left expression $*(tab+1)$ is represented by the green arrow in the figure below. The variable “tab” is an array and its evaluation gives an address. We can apply the first formula of pointer arithmetic to obtain the address $272 + 1 * 4 = 276$. Applying the dereferencing operator to this address, we obtain the value 60.

For the left part, we have first a dereferencing of h. The variable h is declared the first time as `int *h = &g` but the value of h is modified at line 3 and that overwrites the previous value. The variable h contains the address `tab[1]` that can be written as `tab[1] = *(tab+1) = *(272+1*4) = *(276) = 60`. Then, the operation of incrementation is applied. The result is the value after the incrementation because the operation is on the left side and in our case the result is $60 + 1 = 61$.

The last operation is the binary “and” operator: $60 \& 61$. In this case, the value 60 can be represented in binary as 00111100 and 61 can be represented as 00111101. The result of the ‘and’ operator is 00111100, which corresponds to the decimal value 60.



3.3 Existing exercises structure

The objective of the master thesis is to develop a program able to generate distinct exercises and provide individual feedback. This section analyses past exercises to identify the different categories and the underlying syntax patterns. Below, is a compilation of all exercise expressions from previous years. Since only the variable type is significant in this section, the different variable names are replaced with the following notation :

- The variable “data” is a simple variable and is initialized with the statement “**int** data;”.
- The variable “ptr” is used for pointers and is initialized with the statement “**int** *ptr;”.
- The variable “tab” is an array and is initialized with the statement “**int** tab[x];”.
- The variable “c” is a char array and is initialized with the statement “**char** *c = "...";”.
- The notation “number” represents an integer.
- The notation “letter” represents an alphabetical letter.

It is important to note that variable values can be imposed by various expressions. Some expressions may require specific values.

3.3.1 Easy expressions

This category of expressions is an introduction to the use of variables and pointers. The student needs to understand the basic notions of variable, address and value of a variable. Each expression is composed of a combination of the referencing and dereferencing operator.

```
data
&data
*&data
```

```
ptr
*ptr
&*ptr
```

The following expressions have been modified compared to the previous expressions. They were initially used without a cast operation. For instance, the expression “*(**int***)data” was written as “*data”. However, this notation is incorrect and results in a compilation error: “invalid type argument of unary ‘*’ (have ‘int’)”. Therefore, in the project, a cast is always used to ensure proper type handling.

The cast operation requires specific values for the variable. In fact, the value needs to be an address in order to cast.

```
*(int*)data
**(int**)data
*** (int***)data
```

```

&*(int *)data

*(int *)*ptr
**(int **)*ptr
***(int ***)*ptr
****(int ****)*ptr

```

3.3.2 Arrays

Arrays are a specific case of pointers. They consist of continuous addresses in memory and the value of the array variable is the address of the first element. Arrays are commonly used with the notation $address[index]$ which is equivalent to $*(address + index)$.

```

tab[number]
tab[-number]
tab[tab[number]-tab[number]]

```

3.3.3 Pointer arithmetic

This category uses the notion of addition and subtraction on variables or pointers. The addition behaves differently if it is applied to a value or an address. On values, it is the usual operation. However, on addresses, it obeys the two following rules:

$$p \pm i = \text{address of } p \pm i * \text{sizeof}(\text{pointing element of } p)$$

$$p2 - p1 = (\text{address of } p2 - \text{address of } p1) / \text{sizeof}(\text{pointing element of } p1 \text{ and } p2)$$

where p , $p1$, $p2$ are pointers and i is a integer.

Use of the first rule :

```

ptr + data
&data + data
&data + number
&data - data
tab - data
(int *)data + ptr[data]
ptr++ + ++data
++ptr + -*(&data + number)

```

Use of the second rule :

```
ptr - ptr
ptr - &data
&data - ptr
tab - &data
tab - ptr
```

Simple addition :

```
data + data
data + number
*&data + data
```

3.3.4 Incrementation and decrementation

This category also involves the use of addition and subtraction on variables or pointers, along with the increment and decrement operators. In fact, the expression `ptr++` can be considered equivalent to `ptr=ptr+1` (which is an addition of an address and an integer value) but with one subtlety. The position of the operator affects the returned values of the expression. If the operator is placed on the right (i.e., `ptr++`), the variable is incremented, but the returned value is the value of the variable before incrementing. On the other hand, when the operation is placed on the left side (i.e., `++ptr`), the result is the value after incrementing.

Furthermore, in this part, students also need to know the operation priority between the dereferencing and incrementation operators.

```
*ptr++
++*ptr
*++ptr
(*ptr)++
*ptr--
*--ptr
*ptr++
*(++ptr)
++*&data
```

3.3.5 Cast

This category corresponds to a combination of pointer arithmetic and a cast. Unlike easy expressions, where the cast transforms a value to an address, here we transform an address of one type to an address of another type. These expressions are relevant to the first rule of pointer arithmetic because the size of the address is changing.

```
(short *) &data + number
(short *) ptr + data
data + (short *) ptr
(short) data+number
```

```
(long *) ptr - number
(long *) (ptr - number)
```

3.3.6 Complex dereference

This category dereference complex addresses and some of the addresses are obtained with the other categories. It is important to have an address that is inside the memory interval and accessible to the student.

```
*(&data + data)
*(array+number)
&*(array+number)
*(ptr+number)
*(array[number]-*ptr)
*(ptr = ptr)
*(c+(*c-c[letter-letter]))
*((short*)data-number)
*((int *)data)[*(ptr + data)]
```

3.3.7 Subtle expressions

In this category, the operators “&” and “*” are no longer used as referencing or dereferencing operators. Instead, they are used as binary operations. The “&” operator is in this category the bitwise operator “and” operator, while “*” is a simple multiplication between two integers.

```
data &number
data & *ptr
data& +data2
data-number& *ptr-number
data+number&*ptr-number
*(tab+number)&(++*ptr)

*ptr *data
*tab++**ptr++

*&data & data * data
```

3.3.8 Char

Char array has its own category. For the exercises, char arrays are different at the declaration and the memory state. The other pointer variables receive a pointing address with the state of the memory, but for char the pointing address is unknown and for example, the operation `c` is not possible.

To better understand this category here is a typical example of the use of char arrays. It is usually composed of several variables. If we take the challenge of 2022. The declaration is as follows:

```
char *a = "Invariant"; char *l = a + 3;
```

We can represent it in an abstract memory:

l+6	a+9	'\0'
l+5	a+8	't'
l+4	a+7	'n'
l+3	a+6	'a'
l+2	a+5	'i'
l+1	a+4	'r'
l	a+3	'a'
	a+2	'v'
	a+1	'n'
	a	'i'

Figure 3.7: Example of the char representation (August 2022)

Some operations are not allowed with char variables compared to other types. For example, the evaluation of the expression “a” is not possible because the pointing address is unknown.

Another example is the expression “l-a”, if the two variables are int arrays, the resolution consists in taking the value of l minus the value of a divided by the size of an integer, but for char arrays, students need to base themselves on the declaration. From the declaration, we know that $l = a + 3$ and $l - a = a + 3 - a = 3$. The reasoning being different than the other categories, char expression has its own categories.

```
*c
c - c
c-c+number
printf("%s", c)

c[-number] - c[number]
c[letter-letter] - letter
c[**data + number] - c[letter-letter]

c[-number] - *(c-number)
c[-number] - *(c + number)

c[c[number]-c[number]]
c[c[number] - c[**data]] - letter
```

3.4 Statistics

We now analyse the exercises in more detail to understand the frequency of different expressions, variables in the declaration, and the construction of the memory state. This analysis helps to fine-tune our program and provide balanced practice for students. By examining the distribution of expressions, we can focus on generating diverse exercises. Analysing the variables in the declaration will guide us in selecting appropriate types and names.

3.4.1 Expressions

Figure 3.8 reveals the frequency of each category of expressions. On average, an exercise consist of approximately 7 easy expressions, 3 incrementation expressions, 3.5 arithmetic expressions, 1 cast expression, 1 subtle expression, 1.5 complex dereferencing expressions, 0.5 array expression and 2.5 char expressions.

It is important to note that the standard deviation is high for easy and char expressions. The variability is due to the fact that the number of easy expressions can range from 5 to 10 and char expressions are not present in every exercise set, occurring only 9 times out of the 16 exercises analysed.

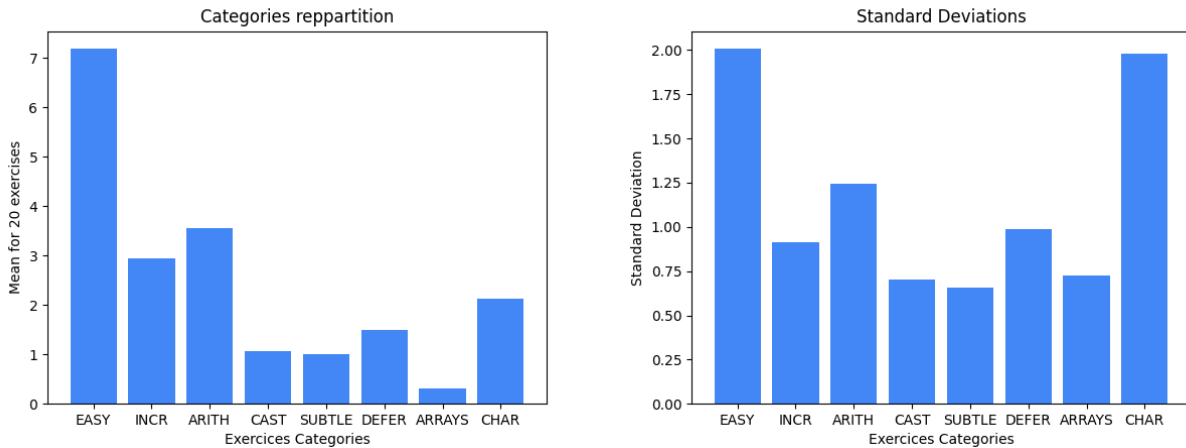


Figure 3.8: Mean values and standard deviations of the repartition of the expressions between the different categories from previous year’ challenges and exams

3.4.2 Declaration

Table 3.1 provides a summary of the variable declarations in the exercises from previous years. From this table, we can deduce the following information :

- There are usually 1 to 3 simple variables. The type is always an integer and the variable names can be lowercase letters or specific names: “data”, “tab” or “value”. Sometimes,

the value contained in the variable is an address number, and this variable can be cast to a pointer in the expressions. Variables containing an address value are almost present in every exercise (14/16) ranging from 1 to 2 occurrences per exercise.

- Pointer variables are usually present only once in an exercise (sometimes twice at most). The variable names are also lowercase letters, with a dominant use of the letter “p” or the specific name “ptr” or “index”. The type is always a pointer to an integer.
- Array variables appear in only 3 out of the 16 exercises and there is usually only one array variable per exercise. The array variable is named “tab” and it is always an array of integers.
- Char variables appear 9 times in the 16 exercises. The variable names are either lowercase or uppercase letters. The value contained in the char variables are either names or some words usually related to the course. They are typically present in groups of 2 or 3 and are linked together. For example, the “Challenge 2020” exercise contains the char variables “s”, “t” and “u”. The first variable is declared as **char** *s = “Challenge”; and the other two are linked to this variable through the declaration: **char** *t = s + 4, *u = t + 2;.

Subject	Simple Variable				Pointer			Array			Char		
	#	Name	Type	Value	#	Name	Type	#	Name	Type	#	Name	Value
Challenge 2016	3	data,tab,x	int	9000,8000,4	1	ptr	int*	0			0		
Challenge 2017	3	data,value,z	int	255,499,42	2	index,p	int*	0			0		
Challenge 2018	2	x,data	int	12,73	1	p	int*	1	tab	int	0		
Challenge 2020	2	x,y	int	500,316	1	p	int*	0			3	s,t,u	“challenge”,“leng”,“nge”
Challenge 2021	2	x,y	int	-57,256	1	p	int*	0			3	s,t,u	“Geraldine”,“ldine”,“ine”
Challenge 2022	2	w,v	int	28,1000	1	b	int*	0			2	a,l	“Invariant”,“ariant”
January 2019	3	data,tab,x	int	9000,8000,4	1	ptr	int*	0			0		
January 2020	2	data,x	int	208,336	1	ptr	int*	0			2	E,F	“Session”,“”
January 2021	2	data,x	int	208,336	1	ptr	int*	0			2	E,F	“Session”,“”
January 2022	2	x,data	int	212,73	1	p	int*	1	tab	int	0		
June 2019	2	x,data	int	236,108	1	ptr	int*	0			2	E,F	“Examen”,“”
June 2020	2	x,data	int	236,108	1	ptr	int*	0			2	E,F	“Examen”,“”
June 2022	3	data,tab,x	int	9000,8000,4	1	ptr	int*	0			0		
August 2019	2	x,data	int	336,208	1	ptr	int*	0			2	E,F	“Examen”,“”
August 2020	2	z,data,value	int	42,255,499	2	index,p	int*	0			0		
August 2022	1	g	int	600	2	h,m	int*	1	tab	int	1	p	“pop”

Table 3.1: Summary of declaration from previous year’ challenges and exams

3.4.3 Memory state

Table 3.2 summarizes the composition of the memory state in the previous years’ exams and challenges. The memory is bounded, and within these bounds, the addresses are supposed accessible to the students. However, not all addresses inside the bounds are represented, as the number of addresses can be quite high (e.g., 87 addresses for the August 2022 exam). Instead, there are memory blocks, some of which are visible to the students, while others are not visible and are located between the visible blocks. A typical memory contains between 7 and 15 visible addresses, with an average of 9.2 addresses. These addresses are divided between 2 and 4 visible memory blocks.

Additionally, a pointer path is always present in the memory. This path consists of values in the memory that are also visible addresses, forming a chain where one value is an address that contains another address, and so on. This path is useful for creating expressions with multiple dereferencing operators. The pointer path typically ranges between 2 and 5 values, with an average of 4 values.

Subject	add	block	b	longest path
Challenge 2016	10	4	[2000],[8000;8004],[9000;9020],[12000]	9020 -> 2000 -> 8000 -> 12000
Challenge 2017	15	4	[1000;1020],[2000;2016],[3000;3008],[4000]	4000 -> 2008 -> 3004 -> 1016 -> 1020
Challenge 2018	10	2	[4;16],[200;220]	208 -> 4 -> 12 -> 204 -> 16
Challenge 2020	7	2	[300;316],[500;504]	300 -> 500 -> 316
Challenge 2021	7	2	[240;256],[440;444]	440 -> 256
Challenge 2022	8	2	[20;36],[1000;1008]	20 -> 1000
January 2019	10	4	[2000],[8000;8004],[9000;9020],[12000]	9200 -> 2000 -> 8000 -> 12000
January 2020	7	2	[204;220],[336;340]	204 -> 336 -> 208 -> 220
January 2021	7	2	[204;220],[336;340]	204 -> 336 -> 208 -> 220
January 2022	10	2	[204;216],[400;420]	408 -> 204 -> 212 -> 404 -> 216
June 2019	7	2	[104;120],[236;240]	104 -> 236 -> 108 -> 120
June 2020	7	2	[104;120],[236;240]	104 -> 236 -> 108 -> 120
June 2022	10	4	[2000],[8000;8004],[9000;9020],[12000]	9020 -> 2000 -> 8000 -> 12000
August 2019	7	2	[104;120],[236;240]	104 -> 236 -> 108 -> 120
August 2020	15	4	[1000;1020],[2000;2016],[3000;3008],[4000]	4000 -> 2008 -> 3004 -> 1016 -> 1020
August 2022	10	2	[268;282],[600;616]	616 -> 278 -> 600 -> 604 -> 600

Table 3.2: Summary of the memory state from previous year' challenges and exams

Chapter 4

Syntax

In this chapter, we identify some general structure of the expressions based on the analysis of the exercises from the previous years, as presented in chapter 3. In the first step, we examine the general syntax. Then, we adopt a more restrictive approach by identifying specific patterns within the expressions.

4.1 General syntax

In chapter 2, we identified methods for generating exercises. The first idea of this project was to use context-free grammar. Due to the requirement of generating expressions across different categories, we use distinct syntax structures for each category. This decision led to small simplifications of the syntax, meaning some expressions can not be generated in this context.

In this section, the category char expressions are grouped with the other categories.

4.1.1 Easy

The solutions for easy exercises can be either a value or an address expression. An address can be obtained with pointer, array, and char variables, as they all have addresses as values. Additionally, the address can be obtained using the reference (&) operator on *⟨value⟩* or *⟨address⟩*. However, to perform this operation, the address needs to be “visible” to the students. In the context of these exercises, only simple variables receive an address with the memory state. The other variables (pointer, array, and char variables) do not have a known address and the referencing is not possible. Also, to use the reference operator, the expression must have an address, for example, the expression “&&x” is not possible.

As explained previously, cast is used to ensure proper type handling. In that case, the type of the cast is the same as the initialization type, but the cast contains more asterisks, depending

on the number of times it is dereferenced.

$\langle easy_ex \rangle$	\rightarrow	$\langle value \rangle \mid \langle address \rangle$
$\langle value \rangle$	\rightarrow	$simple_variable \mid * \langle address \rangle$
$\langle address \rangle$	\rightarrow	$pointer \mid array \mid char \mid$ $* \langle address \rangle \mid \& \langle value \rangle \mid \& \langle address \rangle \mid$ $\langle address_cast \rangle \langle value \rangle$
$\langle address_cast \rangle$	\rightarrow	$(\langle type \rangle \langle asterisks \rangle)$
$\langle type \rangle$	\rightarrow	$int \mid short \mid long$
$\langle asterisks \rangle$	\rightarrow	$\epsilon \mid * \langle asterisks \rangle$

4.1.2 Arrays

The notation of arrays takes the form: `address[index]`. The address represents any addresses obtained previously. The index is an integer value that can be obtained with $\langle value \rangle$, a random number, subtraction of two letters or a nested $\langle arrays_ex \rangle$. The addition of two letters is not used in the exercises as it is not asking the student to know the ASCII values. The subtraction is simply the interval between the two letters.

An array expression is either a simple array notation, addition or subtraction of two array notations, or subtraction between an array and a letter (assuming that the array notation results in a letter).

Array notation is equivalent to $*(address+index)$, meaning that the dereference address must be within the memory interval of the exercise.

$\langle arrays_ex \rangle$	\rightarrow	$\langle array_notation \rangle \mid \langle array_notation \rangle - \langle array_notation \rangle \mid$ $\langle array_notation \rangle - 'letter'$
$\langle array_notation \rangle$	\rightarrow	$\langle address \rangle [value_2] \mid$ $\langle address \rangle [arrays_ex] \mid$
$\langle value_2 \rangle$	\rightarrow	$\langle value \rangle \mid number \mid 'letter' - 'letter'$

4.1.3 Pointer arithmetic

As seen in subsection 3.3.3, there are three different ways to compute the addition or subtraction depending on the operand types: addition or subtraction of an address and a value, addition or subtraction of two values, or subtraction between two addresses. We used the two expressions of the previous syntax ($\langle address \rangle$ and $\langle value \rangle$) but in this case, they are not complete enough which is why $\langle value_3 \rangle$ and $\langle address_2 \rangle$ are created. The added terms are mainly used to build more complex pointer arithmetic expressions.

$\langle \text{pointer_arithmetic_ex} \rangle$	\rightarrow	$\langle \text{address_and_value} \rangle \mid$ $\langle \text{address_and_address} \rangle \mid$ $\langle \text{value_and_value} \rangle$
$\langle \text{address_and_value} \rangle$	\rightarrow	$\langle \text{address_2} \rangle \langle \text{add_sub} \rangle \langle \text{value_3} \rangle$
$\langle \text{address_and_address} \rangle$	\rightarrow	$\langle \text{address_2} \rangle - \langle \text{address_2} \rangle$
$\langle \text{value_and_value} \rangle$	\rightarrow	$\langle \text{value_3} \rangle \langle \text{add_sub} \rangle \langle \text{value_3} \rangle$
$\langle \text{value_3} \rangle$	\rightarrow	$\langle \text{value} \rangle \mid \text{number} \mid ++\langle \text{value} \rangle \mid \langle \text{arrays_ex} \rangle$
$\langle \text{address_2} \rangle$	\rightarrow	$\langle \text{address} \rangle \mid ++\langle \text{address} \rangle \mid \langle \text{address} \rangle ++$
$\langle \text{add_sub} \rangle$	\rightarrow	$+ \mid -$

4.1.4 Dereferencing of more complex address

This category dereferences addresses obtained from the other categories to create more complex dereferences. It can dereference array expressions (after a cast), arithmetic expressions (addition and subtraction of an address and a value) or casting expressions. A new operator appears here with the assignment operator, which can also be dereferenced.

The most important in this category is to obtain an address that is mapped in the memory.

$\langle \text{complex_dereferencing_ex} \rangle$	\rightarrow	$*(\langle \text{complex_address} \rangle) \mid$ $\&*(\langle \text{complex_address} \rangle)$
$\langle \text{complex_address} \rangle$	\rightarrow	$\langle \text{address_and_value} \rangle \mid$ $\langle \text{cast_ex} \rangle \mid$ $\langle \text{array_ex} \rangle \mid$ $\langle \text{pointer} \rangle = \langle \text{address} \rangle$

4.1.5 Incrementation and decrementation

This category mixes the dereferencement operator with incrementation and decrementation. Incrementation and decrementation can be applied to both sides of an address. This category is complex enough on its own, that is why $\langle \text{address} \rangle$ is usually very simple (e.g. $*++ptr$), only variables or $(\&variable)$.

It is also important to note that the increment and decrement operations must be applied to a lvalue that means that the assignment operator can be applied to it (e.g. it is impossible to do

++&ptr because it is not allowed to do &ptr = 1 in C).

$$\begin{aligned}
 \langle incr_decr_ex \rangle &\rightarrow * \langle op \rangle \langle address \rangle \mid * \langle address \rangle \langle op \rangle \mid \\
 &\quad * \left(\langle address \rangle \langle op \rangle \right) \mid * \left(\langle op \rangle \langle address \rangle \right) \\
 &\quad \langle op \rangle * \langle address \rangle \mid \left(* \langle address \rangle \right) \langle op \rangle \\
 \langle op \rangle &\rightarrow ++ \mid --
 \end{aligned}$$

4.1.6 Cast

This category also uses pointer arithmetic, usually an addition or subtraction of an address and a value, but with a cast operator applied to the address. The cast has for consequences to change the address size of the pointed element in the formula : $p \pm i = \text{address of } p \pm i * \text{sizeof}(\text{pointing element of } p)$. The cast operator consists of a type (int, short, long) and some asterisks corresponding to the number of times it can be dereferenced. Some subtlety is introduced with the second and third lines of the relation $\langle cats_ex \rangle$. The second line performs the casting operation after the pointer arithmetic operation and the result is the same as if it had not been cast. The third line casts a value in another type but the size of the address has no importance in a simple addition of two values.

To make these exercises interesting, the $\langle type \rangle$ must be different from the initialization. For example, if the variable was declared as $int * x$, an interesting cast might be $(long*)x$.

$$\begin{aligned}
 \langle cats_ex \rangle &\rightarrow \langle cast \rangle \langle address \rangle \langle add_sub \rangle \langle value \rangle \mid \\
 &\quad \langle cast \rangle \left(\langle address \rangle \langle add_sub \rangle \langle value \rangle \right) \\
 &\quad \left(\langle type \rangle \right) \langle value \rangle \langle add_sub \rangle \langle value \rangle \\
 \langle cast \rangle &\rightarrow \left(\langle type \rangle \langle asterisks \rangle \right)
 \end{aligned}$$

4.1.7 Subtle

This section mixes the bitwise “and” and the multiplication operator with the dereferencing and referencing operator. The bitwise “and” and the multiplication operators are applied to two operands. A subtle expression consists of either one or two operators. Both operators are applied to values, these values can be obtained with $\langle value \rangle$ from the easy section, with dereferencing or incrementing exercises.

$$\begin{aligned}
 \langle subtle_ex \rangle &\rightarrow \langle complex_value \rangle \langle op \rangle \langle complex_value \rangle \mid \\
 &\quad \langle complex_value \rangle \langle op \rangle \langle complex_value \rangle \langle op \rangle \langle complex_value \rangle \\
 \langle complex_value \rangle &\rightarrow \langle value \rangle \mid \langle dereferencing_ex \rangle \mid \langle incr_decr_ex \rangle \mid \\
 \langle op \rangle &\rightarrow \& \mid *
 \end{aligned}$$

4.1.8 Functions

At the moment only one function is used (“printf”). The importance of this section is not what the function does but the value it returns. For the “printf” function it returns the number of characters printed.

$$\langle functions_ex \rangle \quad \rightarrow \quad printf\left(“\%s”, \langle char \rangle\right)$$

4.2 Pattern

The context-free grammar approach of generating exercises has the advantage of creating entirely new expressions that did not exist before. However, it has the disadvantage of losing control over the generated exercises. This approach would require careful checking to ensure the correctness and relevance of the generated expressions.

For this project, a different approach is adopted, using more restrictive patterns and selecting expressions only from these patterns. This approach ensures that the generated exercises align with what makes sense for the students and that each pattern is only used once in an exercise.

Below are the different patterns categorized using the following notations:

- “simple”: for simple variables
- “pointer”: for pointer variables
- “char”: for char variables
- “var”: representing all variables except char (simple, pointer, array)
- “address”: for simple and pointer variables
- “value”: for simple variables or an integer
- “long/short”: meaning either long or short
- “+−”: meaning either addition or subtraction
- “+ + − −”: meaning either incrementation or decrementation
- “[operation]x[operation]”: meaning that the operation can be on either side of the variable.

4.2.1 Easy

Easy 0:

1. *simple*
2. *pointer*
3. *array*
4. *&simple*

Easy 1 :

5. $*var$
6. $\&*var$

Easy 2 :

7. $**var$
8. $\&**var$

Easy 3 :

9. $***var$
10. $\&***var$

Easy 4 :

11. $****var$
12. $\&****var$

4.2.2 Pointer arithmetic

Simple arithmetic :

1. $address + -value$
2. $\&simple + -value$
3. $(int*)simple + -value$
4. $address - address$
5. $address - \&simple$
6. $\&simple - address$
7. $simple + -value$
8. $*\&simple + -value$

Harder arithmetic :

9. $[++--]simple[++--] + -[++--]simple[++--]$
10. $[++--]address[++--] + -[++--]simple[++--]$
11. $[++--]address[++--] - [++--]address[++--]$
12. $[++--]address[++--] + *(&simple + -value)$
13. $(int*)address + address[value]$

4.2.3 Incrementation decrementation

1. $*++--address$
2. $++--*address$
3. $++--*&simple$
4. $*address++--$
5. $(*address)++--$
6. $(*&address)++--$

4.2.4 Cast

1. $(long/short*)address+-value$
2. $(long/short)simple+-value$
3. $(long/short*)&simple+-value$
4. $(long/short*)(address+-value)$

4.2.5 Array

1. $address[value]$
2. $address[-simple]$
3. $address[value]+-address[value]$
4. $address[value+-value]$
5. $address[tab[value]+-tab[value]]$
6. $address[tab[value]+-value]$
7. $(int*simple)[value]$

4.2.6 Char

1. $char-char$
2. $*char$
3. $char[value]-char[value]$
4. $printf("%s",char)$
5. $char[char[number]-char[number]]$

4.2.7 dereferencing

1. $\ast(var = var)$
2. $\ast(arraypattern)$
3. $\ast(arithmeticpattern)$

Chapter 5

Exercise generation

5.1 Process

This section outlines the implementation process of this project. The procedure for creating exercises is illustrated in Figure 5.1.

The first step is to create all the patterns presented in section 4.2. These patterns are created only once and remain unchanged throughout the generation of multiple exercises.

Next, the exercise creation process begins by first generating the variables, followed by the creation of the memory. The previously created variables receive addresses and values.

Then we proceed to create a specific number (usually 20) of different expressions from the available patterns. The pattern selection is not entirely random, as the user can specify the number of expressions to be generated in each category. Once a pattern is selected, some choices remain to be made (selection of the variable and some operators) to obtain a feasible expression. If it is not possible to create an expression that satisfies the constraints of the exercise (e.g., variables, memory), the system selects another pattern within the same category.

Finally, the feedback is generated. The feedback consists of students' possible answers, including both the correct solution and common mistakes.

This process is then repeated for each exercise, with the exception of pattern creation, which remains constant.

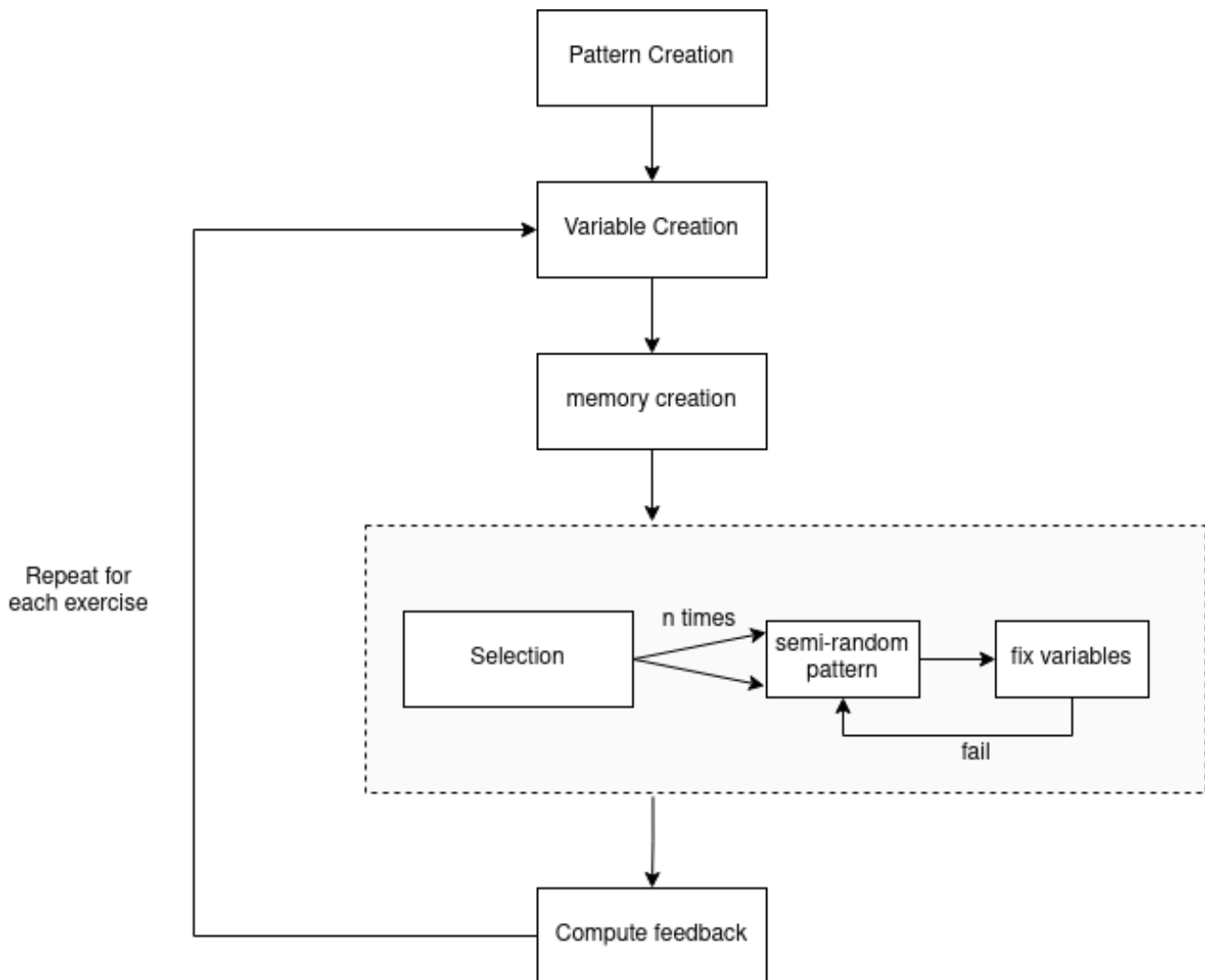


Figure 5.1: General scheme of the structure of the program

5.1.1 Pattern creation

The first step of the program is to create the patterns defined in section 4.2. Several classes are used each responsible for a specific operation. By combining them together, patterns are created in the form of a tree structure, where each node corresponds to the operation.

Below are examples of pattern creation using the different classes:

- A pattern of the easy category is `*var`. The creation of this pattern can be seen in Listing 5.1. This example uses two class constructors. The class `VariableExpr` is responsible for the variables, no input is given to the constructor, this means that the variable can be of any type except char variables (i.e. simple, pointer, array variable). The other class is `Unop`, responsible for the unary operation. It is given the attribute `"*"` and the previous class, meaning that a dereferencing is applied to any variable.

```
1 Unop("*", VariableExpr())
```

Listing 5.1: Pattern creation of `*var`

- In the increment category, the pattern `++--*var` is created using the code in Listing 5.2. In this pattern, the class `Incr` is used. This class is responsible for the increment and decrement operators. The first argument passed is `"left"` indicating that the operator is positioned on the left side of the child expression. The second input is the child pattern, which remains similar to the previous pattern. In this case, the input for the `VariableExpr` class is set as `"address"` which means that only variables containing addresses are used (i.e., pointer, array variables). The last argument is `"++--"` indicating that both can be used and one is selected during the selection process.

```
1 Incr("left", Unop("*", VariableExpr("address")), "++--")
```

Listing 5.2: Pattern creation of `++--*var`

- The pattern `(long/short *) address +- value`, generated by the code shown in Listing 5.3, introduces two new classes. The `Binop` class is responsible for binary operations. It takes three arguments: the first argument is `"+-"`, which means that either addition or subtraction is selected during the generation, second and third arguments are other patterns involved in the binary operation. The `Cast` class deals with casting operations. It has three arguments: the first argument specifies the type to which the expression is cast (in this example, it can be cast to either long or short), the second argument specifies the number of asterisks (*) in the cast expression, the third argument is the pattern to which the cast is applied. This example shows a new argument for the class `VariableExpr` class. The `"value"` argument means that it can be either a simple variable or a number.

```
1 Binop("+-", Cast("long_short", 1, VariableExpr("address")), VariableExpr("value"))
```

Listing 5.3: Pattern creation of `(long/short *) address +- value`

- The creation of the array pattern `address[index]` can be seen in Listing 5.4. This pattern uses the `Array` class composed of two arguments, the first argument is the pattern used

for the “address” part of the array and the second argument is used for the “index” part of the array.

```
1 Array(VariableExpr("address"), VariableExpr("value"))
```

Listing 5.4: Pattern creation of address[index]

- The code in Listing 5.5 creates the pattern `printf("%s", c)`. The `Function` class is responsible for functions. The first argument is the name of the function and the second argument is another pattern corresponding to the argument of the function.

```
1 Function("printf", VariableExpr("char")))
```

Listing 5.5: Pattern creation of `printf("%s", c)`

These examples illustrate all the different classes that can be used to create patterns (`VariableExpr`, `Unop`, `Incr`, `Binop`, `Cast`, `Array`, and `Function`). Although only some arguments have been shown, the entire list of different arguments can be found in section 7.4.

5.1.2 Variable creation

Four different types of variables have been identified in chapter 3:

- Simple variables: These variables are always of type `int`. The variable name is defined in this section. The first variable created is always associated with the pointer path, i.e. the variable can be cast and dereferenced a certain number of times. The number of consecutive dereferences is selected in this section, but the pointer path is created later in the memory creation. The other variables are not part of the pointer path and cannot be dereferenced. At this point, the variables have no address and no value assigned to them, these assignments are made during memory creation.
- Pointer variables: These variables are pointers to `int` variables. The variable names part selects variable names and the simple variables they point to.
- Array variables: These variables are arrays of `int` values. The name and the size of the array are selected here. The starting address of the variable is assigned during memory creation.
- Char variables: Each variable is given a name. As explained in subsection 3.3.8, the first char variable is declared and assigned a string value. Then, the other variables are declared, all pointing to the same string value at different offsets. In this way, the first variable contains the string, and the other variables point to different locations within the string by applying an offset.

At this stage, all the attributes are selected except for the addresses. The addresses are assigned during the memory creation.

5.1.3 Memory creation

After creating the variables, the next step is to construct the memory state by assigning addresses to the variables. The process is as follows:

1. Define a random memory upper and lower bound. In the example below, the memory range is [4, 84].
2. Divide the memory range into memory blocks. Memory blocks are continuous addresses that are visible to the student. In the example, the memory is divided into three blocks: [4,20], [28] and [72,84]. It is important to construct memory blocks large enough to represent array structures that require continuous addresses. For example, the variable “g” is an array of 4 int and it should be placed in a block larger or equal to four addresses. In this part, only the number of blocks and the size of the blocks are chosen.
3. Assign addresses to the memory blocks. The first block should begin at the lower bound address and the last block should end at the upper bound address. The remaining blocks are placed in between. Char variables are not represented in the memory but there should still be enough free continuous blocks to contain them.
4. Assign addresses to the variables. Pointer variables do not need addresses, this address is unknown to the student. Array variables are assigned a pointed address at the beginning address of this variable and simple variables are assigned an address.
5. Create a pointer path. Some variables can be dereferenced multiple times. In the example, variable “e” can be dereferenced four times and the path is 84 -> 72 -> 76 -> 20 -> 28. The creation of the pointer path is random. Memory values that are not part of the pointer path are left empty (symbolized by “/”).
6. Assign the remaining empty value (“/”) to a random value, either some small values that can be more easily used in the expression or other memory addresses.

The result is the memory:

e	84	72
b	80	1
	76	20
	72	76
		...
	28	76
		...
	20	28
	16	2
	12	2
	8	1
g[0]	4	3

Table 5.1: Example of memory state after its creation

5.1.4 Selection

The goal of this part is to select a correct expression from the patterns created in the previous section (subsection 5.1.1). The selection process involves making decisions such as selecting the variable, choosing an operator and verifying if the resulting expression is feasible given a certain declaration and memory state.

The first step is to select a pattern but this selection cannot be completely random. The program user has the possibility to define the number of expressions belonging to each category and within a category, to specify certain difficulties (see subsection 7.3.3). For each category, a pattern is randomly chosen and the different left choices are selected. The generated expression is then tested for feasibility and compatibility with the memory state. If the expression is not feasible, another pattern within the category is selected and tested. This process continues until the correct number of expressions is obtained for each category.

To perform the selection and test the feasibility, the “selection” function is used. This function is implemented in each class and traverses the entire tree, making the appropriate selections. It takes as arguments the memory and a vector containing all the variables.

The “selection” function can also take an additional argument called “possibilities”. This argument is used to restrict the range of possible values for the resulting expression, i.e. the child’s answer must be in the possibilities vectors.

The function returns true if a correct expression has been created or false if the expression cannot be created.

Let’s take an example of the pattern selection process. Figure 5.2 shows the execution of the pattern (**long/short** *) address +- value which corresponds to the following pattern code:

```
1 Binop("+-", Cast("long_short", 1, VariableExpr("address")), VariableExpr("value"))
```

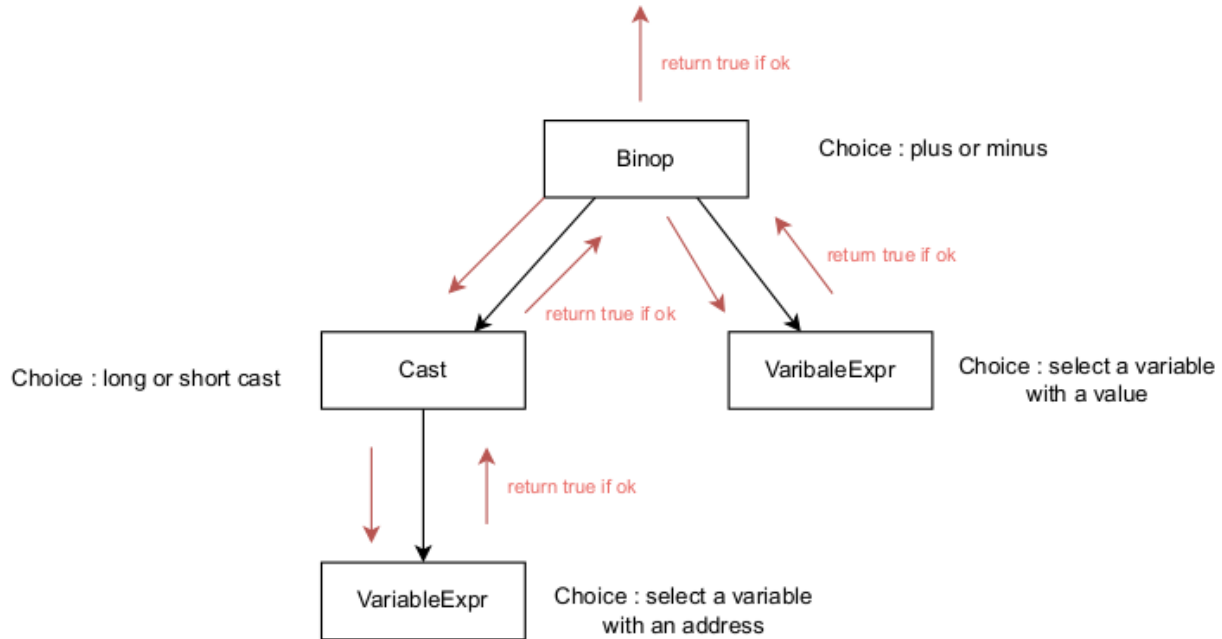


Figure 5.2: Example of the selection process on the pattern (**long/short** *) address \pm value

The first step is to call the function “selection” on the root of the pattern tree, which in this case is a binary operation (class Binop). This operation argument “ \pm ” means that either an addition or a subtraction can be selected to construct the expressions and a choice needs to be made at this stage. It then calls the same function (“selection”) on the left child and when it has finished, it calls the function on the right child.

The left child is a cast operation, which needs to make a choice between a long or short cast. The selection process is then propagated further to its child (VariableExpr class). The child class has been given the argument “address”, which means that it needs to select a variable containing an address (pointer or array variables). If a variable with these properties has been created during the variable creation process, it is selected and the function returns true to its parent, indicating that a correct expression has been found. The “Cast” class receives the true value from its child and since it has already made its selection, it also returns true. The “Binop” class can now propagate the selection to its right child, which involves selecting an integer variable in the VariableExpr class. Finally, when all the classes have made their choices, the root returns true, indicating that the expression has been correctly created from the pattern.

If a class encounters a problem in making a selection and returns false, it aborts the selection process and propagate the false value directly to the root as the output.

The above-mentioned attribute “possibilities” is used to restrict the value of the expression. This attribute lists all the different values that the child expression can take and the child class makes its choices according to the given possibilities. For example, a dereference operation can

only be applied to addresses within the memory range and in the selection process, it gives its child a list of all the different possible addresses within the possibilities argument.

The behaviour of the different classes for the “possibilities” attribute is as follows:

- **Cast:** This class does not change the value, only the interpretation. Therefore, the list of possibilities is simply transferred to its child.
- **Array:** The first step is to choose an address without any restrictions. Based on this address, we can select all the indexes where the expression `address[index]` results in a value from the given possibilities. The list of indexes is given as input to the selection process of the index expressions.
- **Binop:** In this class, we have two child expressions and the possibilities restrict the combination between the two. First, the left side is selected without any restriction. Then, we can evaluate the child and from this value, compute the possibilities of the right side.
- **Incr:** For this class, the possibilities are changed based on the value that the child expression needs to satisfy. When the operator is on the right side, the possibilities do not change. However, when the operator is on the left side, we perform the opposite operation. For example, if the operation is an incrementation, we decrement all the possibilities to pass to the child.
- **Unop:** For a dereferencing operator the child value needs to be an address within the memory range and the corresponding value of the address needs to be in the possibilities list. For the negation operator, it is simply the opposite of the given possibilities. For the referencement operator, the method of possibilities does not work because it restricts only the value and not the memory addresses. Therefore, this operation needs to be made inside a loop, where at each iteration, a random selection is made and then checked to see if the condition is satisfied.
- **VariableExpr:** This class simply selects the variable with the stored value from the set of possibilities.

Figure 5.3 provides an example to explain the construction of an expression using the attribute “possibilities”. The expression in this example is a simple pattern represented as `*++address` which is obtained with the following code:

```
Unop("*", Incr("left", "++", VariableExpr("address")))
```

The dereferencing operator “*” requires its child to be an address inside the memory range and also accessible to the student. It lists all the possible values inside the “possibilities” attributes $[a_1, a_2, \dots, a_n]$. The increment class modifies these possibilities and passes them to its child. In fact, to compute the solution it increments the value of its child. With the “possibilities” attribute we know the possible solutions, so we perform the inverse operation to obtain the possible value of the child. Finally, the class `VariableExpr` selects a variable that satisfies the given possibilities.

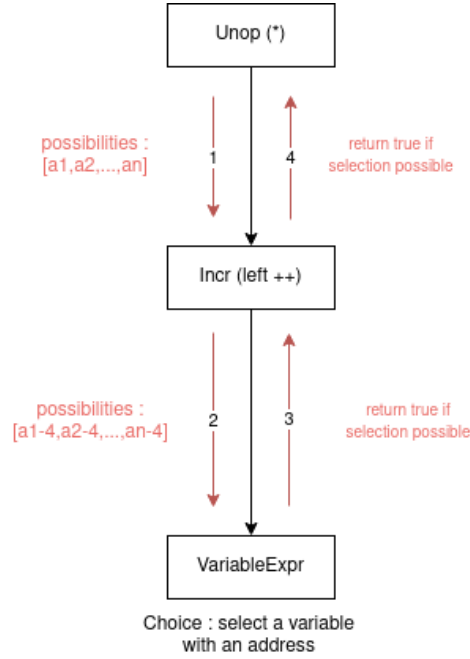


Figure 5.3: Example of the selection process on the pattern `*++address` (with the use of possibilities)

5.2 Testing

In this section, we present a simplified method to test the generated expressions. The idea is to create a C program and verify if the outputs match the expected results. In the exercises, the size of memory addresses is 4 bytes, it must be compiled as a 32-bit program.

Instead of manually creating the memory and assigning values to specific addresses, we can use the `malloc` function to allocate the memory dynamically.

The size of the memory block corresponds to the number of accessible addresses inside the memory range. In C, we can allocate memory as follows:

```
int *memory = malloc(upperbound-lowerbound+sizeof(int))
```

Using `malloc`, we do not have direct control over the allocated memory addresses. This poses a challenge in assigning values to the memory. All values in the memory are of type `int` but some values may need to be cast into addresses. We need to modify these values to correspond to the allocated addresses, rather than the value specified in the statement. Values can be assigned to the memory block as follows:

- For values that are not addresses: `memory[x] = number`,
- For values that can be interpreted as addresses: `memory[x] = &memory[y]`

This modification allows us to dereference these values after a cast operation. However, when

these values are used as regular integers, we need to convert them to match the values specified in the statement. To do this, we subtract the address of the first element allocated from the value and add the lower bound of the statement memory. For example, if $\text{memory}[x] = \&\text{memory}[y]$, to obtain the value of $\text{memory}[x]$, we use $\text{memory}[x] - (\text{int})\text{memory} + \text{lowerbound}$.

The creation of variables needs to be adapted. Specifically, a simple variable declared in the statement as “**int** variable;” will be declared as a pointer to int (**int** *variable;). We can easily assign the desired variable addresses to these pointers and by dereferencing them once, we can obtain the same properties of the variables specified in the statement. We simply need to replace the variable in all expressions with the dereferenced variable. For example, the expressions to obtain the value and the address of a variable, *variable* and *&variable* is replaced with **variable* and *&*variable*.

Pointer and array variables are declared as usual because the addresses of these variables are not known in the statement and can be any valid memory address.

An example of a C program can be found in section A.1. In the first part of the program, we declare all the different problem variables and allocate the memory accordingly. Following that, we initialize all the variables using the “init” function. This function is called between each expression since the memory is reset for each expression. Finally, all expressions can be found inside a printf statement.

To execute the program, the gcc command can be used along with the -m32 flag. This flag instructs the compiler to compile the program as if it were for a 32-bit architecture. The output can be seen in the picture below and we can check that all the expressions were feasible.

```
val@val-VirtualBox:~$ ./t
g          4
&b         80
*v          72
** (int **)e  20
&** (int **)v 76
++*&e       73
*g++        3
(*&b)--     1
g-e        -284
v-g         20
v-&b         1
v-- - --e   -200
(short *)v-b 82
(short *)&e+b 86
v[b+ -1]    72
v[-17] + g[b] 3
G-P         6
*G          a
*(&e+ -3)    76
*(v=g)       3
```

Figure 5.4: Output obtained in a 32-bit architecture to test the correctness of the exercises

Chapter 6

Feedback generation

In this chapter, we focus on evaluating and providing individual feedback on the student's solutions. The goal is to correct their solutions and help them identify any errors they may have made. That way, the students can more easily spot the mistakes they made and identify which concepts they might be struggling with.

6.1 Different types of errors

This section lists the possible errors a student can make. The errors are divided in different categories depending on the type of performed operations. For that, we separate the errors based on the different classes of subsection 5.1.1.

6.1.1 Variable

1. The evaluation of simple variables normally returns the value at their memory address. A possible error could be to give the address of the variable instead of the value.
2. Pointer or array variables return an address in the memory where this variable is pointing. The error could be to give the pointing value, corresponding to a dereferencing of the variable.

6.1.2 Unary operator

3. With the dereferencing operator, the student could forget to dereference, especially when the expression is complex with multiple dereferencing operators.
4. For the same reason as previously, if multiple dereferencing operators are in an expression, the student could dereference too much.
5. With the dereferencing operator, a cast operator may appear. In this project, when a dereferencing operator is applied to an expression and this expression cannot be derefer-

enced any further, a cast operator appears automatically. A student could think that the cast operator is equal to a dereferencing, but in reality, it just transforms the interpretation to the compiler to allow dereferencing, but it alone is not a dereferencing.

6.1.3 Incrementation and decrementation

6. When the increment is on the right, it returns the value before the incrementation. The error could be to return the value after it. There are two different possibilities: if the value is an integer, it performs a simple incrementation ($\text{value}+1$), or if it is an address, it increments by the address size ($\text{address} \pm \text{sizeof}(\text{pointing element})$).
7. On the other hand, when the operand is on the left side, it normally returns the value after incrementation, but the error could be to return the value before the incrementation.
8. To increment an expression returning a value, an error could be to increment by the address size.
9. Conversely, increment an address expression by 1 instead of the address size.

6.1.4 Arithmetics

10. Addition or subtraction of two values ($\text{value1} \pm \text{value2}$) can be wrongly interpreted as an addition of an address and a value. In such cases, the students may multiply one of the values by the size of the address ($\text{value1} \pm \text{value2} * \text{sizeof}(\text{pointing element})$).
11. Subtraction of two values can be interpreted as a difference between addresses, with the result divided by the size of the address ($(\text{value1} - \text{value2}) / \text{sizeof}(\text{pointing element})$).
12. Addition or subtraction of an address and a value ($\text{address} \pm \text{value}$) can be mistakenly considered as operations between two integer values ($\text{value1} \pm \text{value2}$).
13. Subtraction of an address and a value ($\text{address1} - \text{value}$) can be interpreted as a subtraction of two addresses ($(\text{address1} - \text{address2}) / \text{sizeof}(\text{pointing element})$).
14. Subtraction of two addresses ($\text{address1} - \text{address2}$) can lead to forgetting to divide by the address size.
15. Subtraction of two addresses ($\text{address1} - \text{address2}$) can be misinterpreted as a subtraction of an address and an integer ($\text{address1} - \text{address2} * \text{sizeof}(\text{pointing element})$).

6.1.5 Cast

16. An error could be to cast to the wrong type. This category is mainly used to cast to different address sizes, and an error could be to either forget to cast and keep the same address size, or change it to the wrong value.
17. This error is similar to error 5 and is also present in this class.

6.2 Implementation

In this section, we discuss the implementation of feedback for the expressions. The main objective is to establish a relation between possible answers and an explanatory text that describes the likely mistake made by the student. We start by computing the correct answer, which is indeed the most basic feedback. From there, we enhance the feedback by incorporating other answers that correspond to typical errors.

6.2.1 Correct answer

To summarize the previous chapter, we create a tree where each node represents a patterns, meaning that some choices have to be made. With the selection process, we traverse the tree to obtain a working expressions tree. The goal here is to find the correct answer by traversing the whole tree and computing the correct value for each node starting from the tree leaves.

Four pieces of information are essential to compute the solution for each node. First, it requires the result value of the child node as most operations are applied to this value. Second, it needs the current address, which is important for the reference operator (“&”). Third, it needs to know if the answer is an integer or an address that can be dereferenced (in which case it needs the number of times it can be dereferenced before becoming an integer). Last, it needs the address size as it can change with the casting operation.

6.2.2 Feedback

For the feedback, we use the same procedure as the correct solution. It is basically the wrong computation of the answer. The child node makes an error in the computation and returns the wrong four variables tuples. As here the “answer” is not unique and each node returns a list of answers, it needs more than the four variables tuples to make the difference between them. It needs to associate the answer with the error made (the error code number can be seen section 6.1).

Once all the errors have been computed, only the end value and the error code (then translated into a text explaining the error) are kept. The end result is a list of pair values: the answer and the corresponding errors (see Figure 6.1). It could be possible for an error or combination of errors to lead to the same answer. In that case, we use the likelihood of the errors to select the most probable. Each error is attributed a probability between 0 and 1. In the end, we take the most probable. If more than one error is made we simply multiply the likelihood of all errors. It is important to note that the correct result always needs to be taken.

```
"3": {  
  "360": "Vous avez la bonne reponse",  
  "340": "Vous avez probablement fait 1 erreur:\n  
        Vous avez deference trop de fois.\n",  
  "336": "Vous avez probablement fait 2 erreurs:\n  
        La variable est de type entier.  
        Sa valeur est donc celle stockee dans la memoire  
        et non son adresse.\n  
        Vous avez oublie de deferencer une fois.\n",  
  "260": "Vous avez probablement fait 1 erreur:\n  
        La variable est de type entier.  
        Sa valeur est donc celle stockee dans la memoire  
        et non son adresse.\n"  
},
```

Figure 6.1: Example of a generated feedback for the pattern $*(int*)simple$ (fourth expression in the exercise on github).

Chapter 7

Guide

This chapter is dedicated to the use and maintenance of the program. The code is available on https://github.com/ValBaum/tfe_exercise_generator.

7.1 Global architecture of folders

The program's structure is divided into different folders and files, as shown in Figure 7.1. Here is an overview of the different folders:

- Exercises: This folder contains the generated exercises. The program generates a single LaTeX file containing all exercises. This allows for quicker transformation of the LaTeX file to a PDF, rather than performing the operation one by one. However, if needed, exercises can be generated individually using the parameter “merge_output” (see section 7.3).
- Templates: This folder holds the various templates used to generate the exercises in LaTeX format. The file “global_template.tex” contains the LaTeX preamble, and the file “one_ex_template.tex” contains the structure of one exercise, which is appended to the end of the previous file. Additionally, the file “one_by_one_template.tex” is used when generating exercises into separate LaTeX files.
- Feedback: This folder contains the different feedback for each exercise. Each exercise's feedback is stored in a separate JSON file, and the structure of the feedback can be seen in Figure 6.1.
- Solutions: This folder, contains text files with the correct answers for each expression in the exercises.
- Graphs: This folder holds the various graphics generated by the program, which are used in this report for analysis purposes.

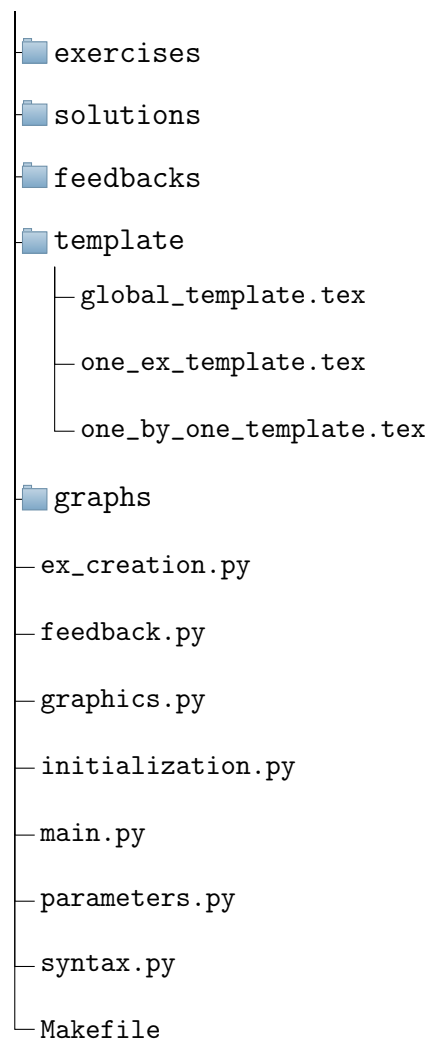


Figure 7.1: Global architecture of folders and files

The different files in the program are as follows:

- `ex_creation`: This file is responsible for creating the exercises. It contains the different patterns defined in section 4.2, which can be created using the function “`PatternsCreation()`”. The file also handles the selection process, creating expressions based on the user-defined category repartition.
- `feedback`: This file handles the creation of feedback for each exercise. It contains the function “`createFeedback(expression, memory)`”, which takes the expression to evaluate and the memory state as input and returns a list of value-feedback in a dictionary format.
- `initialization`: This file manages the creation of variables and the memory. The function “`createVariables()`” creates the variables, and “`createMemory(variables)`” creates the memory. It also includes functions for generating the memory and declaration in LaTeX format.
- `parameters`: This file contains different parameters that can be easily modified to fine-tune the exercises. You can find a list of all the parameters in section 7.3.
- `syntax`: This file defines the different classes used to create the patterns. The classes include:
 - `VariableExpr`: Represents the variables
 - `Unop`: Represents unary operations
 - `Incr`: Represents incrementation and decrementation
 - `Binop`: Represents binary operations
 - `Cast`: Represents casting operations
 - `Array`: Represents arrays
 - `Function`: Represents functions

The classes are composed of the same class functions: “`__init__`” for initialization, “`selection`” to create the expression from the pattern, “`solution`” to obtain the solution of the expression, “`feedback`” to generate feedback, “`latex`” to generate the expression in LaTeX format, and “`printExpr`” to create the expression in a readable way.

- `main`: This file orchestrates the program. It performs various operations in the order shown in Figure 5.1. The operations include creating all the patterns, choosing expression repartition in categories, selecting different variables, creating the memory, selecting expressions from the patterns, filling the memory, computing the solution, generating feedback, writing the different outputs (LaTeX, solution, and feedback) in the corresponding folders, and generating graphics if required.
- `Makefile`: The Makefile contains the main function “`make run`” to run the program and “`make pdf`” to generate the PDF from the LaTeX file in the exercises folder.

7.2 How to run the program

The project is implemented in Python. To run the program, you need to install the `pandas` and `matplotlib` packages. You can install these packages using the command “make package”, as provided in the Makefile.

After the installation, you can easily run the program by executing the command “python main.py” or by using the Makefile command “make run”.

7.3 Adapting the parameters

The program user has access to several parameters that allow fine-tuning the exercise generation process according to their preferences.

7.3.1 Global parameters

- `nbr_exercises`: Specifies the number of different exercises to generate.
- `similar_exercises`: A boolean that determines whether to generate similar exercises for all students. If set to true, the program ensures the same difficulty level and distribution of exercise categories across all exercises. However, variable names, memory states, and expression selections still differ.
- `merge_output`: A boolean parameter that is true if the program generates all exercises in a single LaTeX file, making it easier and faster to create the PDF file, especially when the number of exercises is high. If set to false, each exercise is saved in a separate LaTeX file.
- `analyse`: A boolean parameter that, when set to true, generates different graphics to analyse the program’s behavior.

7.3.2 Variables parameters

- `variable_repartitions`: This list contains vectors specifying the number of different types of variables that appear in the exercises. The vector takes the form: [number of simple variables, number of pointer variables, number of array variables]. Char variables are not included here as they have their own parameters. If “`identical_exercises`” is true, the program chooses one vector from this list at the beginning. Otherwise, a vector is selected for each exercise.
- `variable_char_repartition`: This simple list contains the number of char variables that can appear in an exercise. Char variables have their own parameters because they should only appear when there are char expressions (determined by the exercise repartition parameters). If char exercises are to be included, the program chooses a number from this list. It behaves similarly to “`variable_repartitions`” when “`identical_exercises`” is true.

7.3.3 Category repartition parameters

- `exercise_repartitions`: This parameter contains a list of vectors, each corresponding to the number of exercises to create for each category. The vector format is as follows: `[[easy0, easy1, easy2, easy3, easy4], [incr], [arithm1, arithm2], [cast], [array], [char], [defer]]`. The “easy” category is divided into different depths, where “easy0” corresponds to a variable alone, “easy1” corresponds to a variable dereferenced once, and so on. Similarly, the arithmetic expressions are divided into “arithm1” and “arithm2” to represent different difficulty levels. If “`identical_exercises`” is true, the selection is made only once and applied to all exercises.
- `forced_exercise_repartition`: This parameter allows the user to force the program to use a specific category vector for all exercises. When this parameter is non-empty, it overwrites the “`exercise_repartition`”, and the specified vector is used for all exercises.

7.3.4 Memory parameters

- `max_memory_interval`: This parameter contains the lower and upper bounds for the memory interval. The program chooses an interval between these two values for each exercise.
- `memory_range`: It is the length of the memory interval for each exercise.
- `nbr_visible_address`: It represents the number of addresses that are visible within the memory interval.
- `max_nbr_memory_blocks`: This parameter defines the maximum number of distinct memory blocks inside the memory interval.

7.4 Add patterns

To add additional patterns to the existing ones, we need to modify the file “`ex_creation.py`”. The first part of this file is dedicated to creating the patterns, including a function for each category. To add a new pattern, follow these steps:

1. Choose the appropriate category list: Identify the category where we want to add the new pattern. There are separate lists for each category. For example, we have lists like `easy_ex[NUMBER]`, `incr_ex`, `arithmetic_ex[NUMBER]`, `cast_ex`, `array_ex`, `char_ex`, `defer_ex`. Choose the appropriate list based on the category of the new pattern. It is worth noting that the lists for the “easy” and “arithmetic” categories are subdivided, and the numbers correspond to the difficulty level (ranging from 0 to 4 for “easy” and 0 to 1 for “arithmetic”).
2. Add the pattern to the list: Use the following code template to add the new pattern to the chosen category list:

```
1 CATEGORY.append((PATTERN_NUMBER, PATTERN))
```


- Replace “CATEGORY” with the name of the chosen category list.
- Choose a unique “PATTERN_NUMBER” to differentiate the pattern inside the same category. This number is mainly used for generating graphics.
- Replace “PATTERN” with the actual pattern obtained by combining the different classes from the file “syntax.py”.

The different classes available for creating patterns are the following:

- VariableExpr(VARIABLE_TYPE)
 - VARIABLE_TYPE: “simple_variable”, “pointer”, “array”, “address” (for pointer or array variables), “value” (for simple_variable or a random integer), “” (if the variable type is not important).
- Unop(OPERATION, EXPRESSION)
 - OPERATION: “&” (for referencing), “*” (for dereferencing), “-” (for the negation)
- Incr(SIDE, EXPRESSION, OPERATION)
 - SIDE: “right”, “left”, “left_right” (for either side). It corresponds to the side where the operator is applied.
 - OPERATION: “++” (for incrementing), “--” (for decrementing), “+-” (for either).
- Binop(OPERATION, LEFT_EXPRESSION, RIGHT_EXPRESSION)
 - OPERATION: “+”(for addition), “-” (for subtraction), “+-” (either addition or subtraction,) “=” (for the assign operator).
- Cast(TYPE, CAST_ASTERISKS, EXPRESSION)
 - TYPE: “long”, “short”, “long_short” (for either).
 - CAST_ASTERISKS: A number corresponding to the number of asterisks (*) in the cast expressions. Note that the number cannot exceed the value of the parameter “max_cast”.
- Cast(EXPRESSION_1, EXPRESSION_2)
- Function(FUNCTION_NAME, EXPRESSION)
 - FUNCTION_NAME: “printf”

These classes allow for the creation of different patterns by combining various expressions and operators. Examples of existing patterns can be found in subsection 5.1.1.

7.5 Modifying the feedback

To improve feedback generation, two important aspects to consider are the error probability vector and the error messages. The error messages can be found in the file “feedback.py” and the error probability vector can be found in “syntax.py”.

The error probability vector (“error_probabilities”) associates each error with a certain likelihood of a student making that specific mistake. Currently, this vector is random, but to make the feedback generation more accurate and realistic, it should be based on research and analysis of actual student mistake probabilities. By analyzing students’ past performance, we can determine the most common mistakes and assign appropriate probabilities to each error.

The error messages are stored in the dictionary “error_conversion”. Each error is associated with a unique key corresponding to its error number (the error number can be seen in section 6.1). The dictionary already contains error messages designed to help students identify their mistakes. However, to enhance the feedback, it may be necessary to modify or expand these error messages.

Chapter 8

Result analyses

This chapter analyses the program's behaviour from several viewpoints, including the execution speed, even distribution of various patterns, and diversity of exercises and expressions generated.

8.1 Repartition between the pattern

This section analyses the distribution of the different patterns used in the exercises. The goal is to determine whether the patterns are randomly chosen among the exercises. Two types of graphs are investigated: the distribution percentage of patterns within a category (Figure 8.1) and a graph showing the use of patterns across exercises (Figure 8.2). The first graph displays the percentage of use of each pattern inside the category and the second graph displays pattern numbers on the x-axis and exercise numbers on the y-axis, with dots indicating if the pattern is used in the exercise. Both graphs are generated for each exercise category.

The parameters used for generating these graphs are as follows:

```
1 nbr_exercises = 1000
2 variable_repartitions = [[3,2,1]]
3 variable_char_repartition = [2]
4 forced_exercise_repartition = [[2,2,1,1,1], [3], [3, 1], [1], [1], [2], [2]]
5 max_memory_interval = [0, 500]
6 memory_range = 100
7 nbr_visible_address = 10
8 max_nbr_memory_blocks = 4
```

This setup generates 1000 exercises with 3 simple, 2 pointer, 1 array and 2 char variables. The exercise distribution consists of 7 easy exercises (easy0=2, easy1=2, easy2=1, easy3=1, easy4=1), 3 increment or decrement exercises, 4 pointer arithmetic exercises (arithm0=3, arithm1=1), 1 cast exercise, 1 array exercise, 2 char exercises and 2 complex dereferencing exercises.

As a reminder, the easy category is divided into sub-categories based on difficulty, ranging from easy0 to easy4. The first four patterns are in the easy0 subcategory, patterns 5 and 6 are in the easy1 subcategory, patterns 7 and 8 are in the easy2 subcategory, patterns 9 and 10 are in the

easy3 subcategory, and patterns 11 and 12 are in the easy4 subcategory. With this in mind, we can see from the distribution graph in Figure 8.1a that the patterns are evenly distributed. The higher frequency of patterns 5 and 6 is due to the fact that they appear in every exercise (two exercises are selected for easy1 subcategory and there are only two patterns).

For the increment and decrement category (Figure 8.1b), the patterns are fairly evenly distributed.

In the pointer arithmetic category (Figure 8.1c), which also has two subcategories (patterns 1 to 8 in the first subcategory and patterns 9 to 13 in the second), the distribution within each subcategory is balanced.

Both the cast (Figure 8.1d) and the char (Figure 8.1f) pattern categories have fairly even distributions.

So far we have seen that the distribution is relatively even. However the graphs for the array and complex dereferencing categories show a less even distribution due to the specific constraints of the patterns.

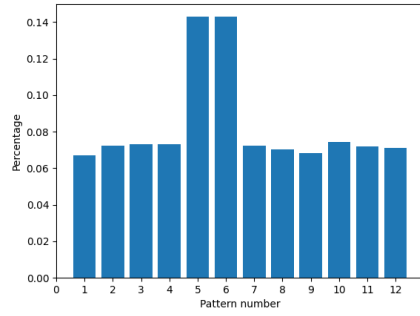
Within the array category, patterns 2 and 5 appear less frequently compared to other patterns. Pattern 2 corresponds to `address[-simple_var]`. It requires a specific value for the simple variable to make the expression feasible. Similarly, pattern 5 (`address[address[value] + -address[value]]`) is also more complex. As a result, these patterns are less used. The other patterns in these categories are more evenly distributed.

For the dereferencing category, the first pattern is a specific pattern of dereferencing (`*(address = address)`). Patterns 2 to 8 correspond to array patterns (1 to 7), and patterns 9 to 21 correspond to pointer arithmetic patterns (1 to 13).

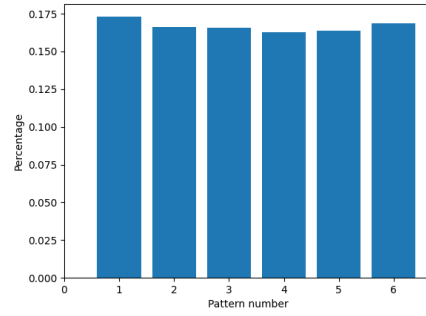
We can see in Figure 8.1g, that patterns 12, 13, 14 and 19 are rarely used. They all represent a difference of addresses (`(address1 - address2)/sizeof(pointingelement)`). The result of these expressions is usually a small value that can rarely be dereferenced.

Patterns 17 and 18 (respectively `[++--]simple_var[++--]+-[++--]simple_var[++--]` and `[++--]address[++--]+-[++--]simple_var[++--]`) required that the addition or subtraction of the two variables results in a memory address. The other patterns are used more frequently because it uses the term “value” (instead of “simple_var”) that selects either a `simple_var` if it is possible otherwise it selects an integer that fits the expressions. It’s important to note that only variables, not integers, can be incremented.

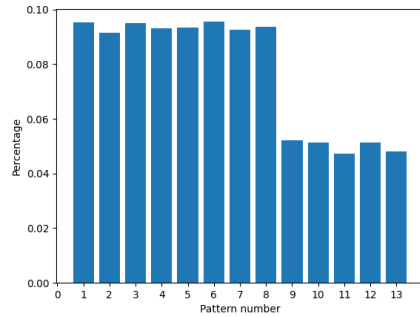
Pattern 3 corresponds to the previously mentioned `address[-simple_var]` pattern, which is not used consistently in the array category. Now, with the additional constraint that the result must be a memory address, it is used even less frequently.



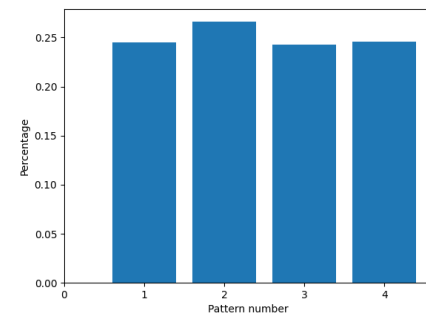
(a) Easy



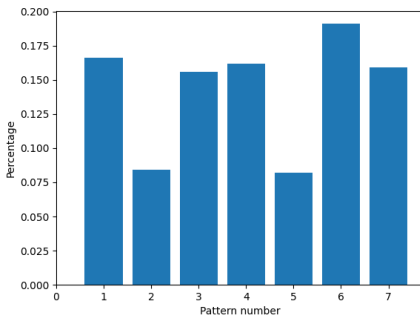
(b) Incr



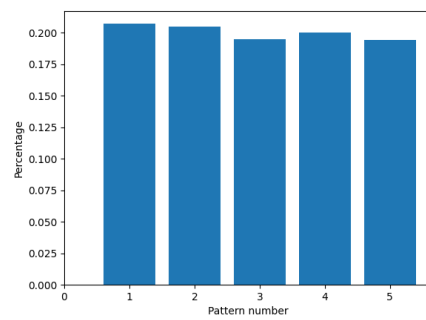
(c) Arithm



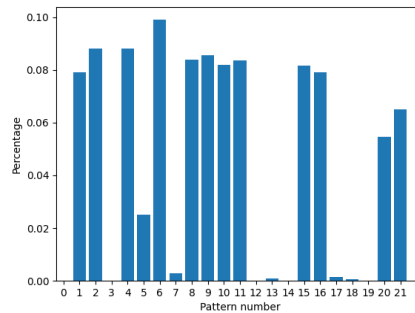
(d) Cast



(e) Array

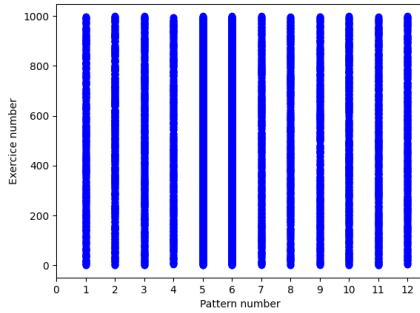


(f) Char

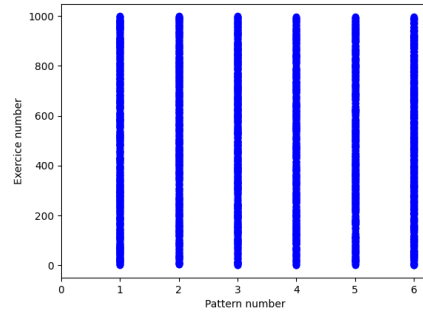


(g) Defer

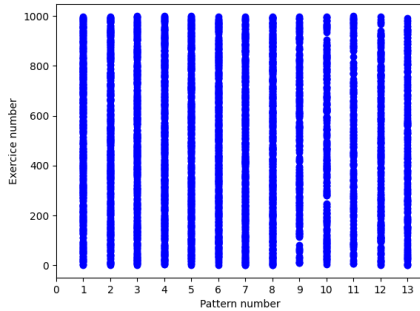
Figure 8.1: Distribution of patterns for each category. Each pattern is associated with the percentage of occurrences within a category.



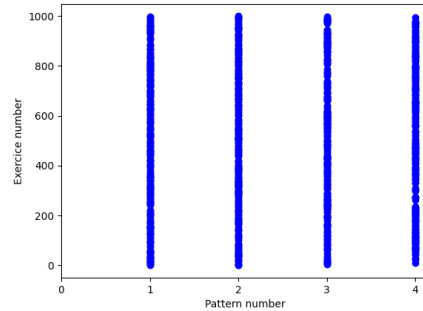
(a) Easy



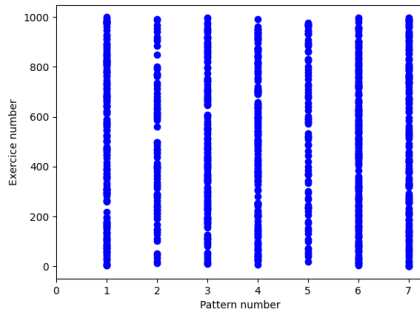
(b) Incr



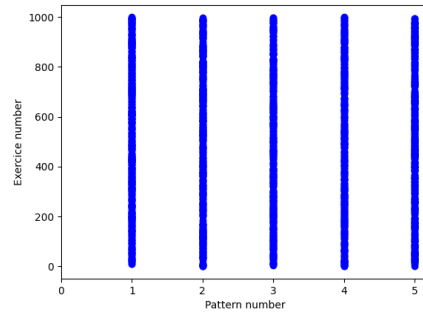
(c) Arithm



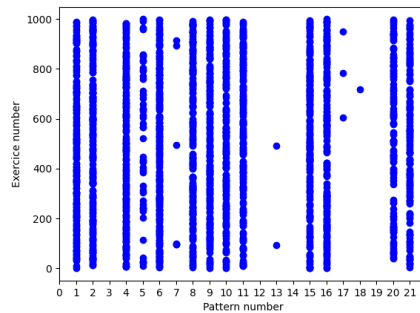
(d) Cast



(e) Array



(f) Char



(g) Defer

Figure 8.2: Pattern dot cloud. It displays the pattern numbers on the x-axis and exercise numbers on the y-axis, with dots indicating pattern usage in exercises.

8.2 How different are the exercises for the same pattern?

Figure 8.3 shows the percentage of different expressions associated with each pattern. This percentage is determined by dividing the number of expressions requiring different resolutions by the frequency of pattern usage. As explained earlier, the expressions are structured as trees, with each node representing an operation. To quantify the diversity of resolutions, we track both the memory address being accessed and the resulting value of each node. By aggregating these values for each node, starting from the leaf nodes, we create vectors representing the values obtained at different levels of resolution. Two expressions are considered different if their vectors differ.

The parameters are similar to the previous section, but the number of created exercises changes to 200:

```

1 nbr_exercises = 200
2 variable_repartitions = [[3,2,1]]
3 variable_char_repartition = [2]
4 forced_exercise_repartition = [[2,2,1,1,1], [3], [3, 1], [1], [1], [2], [2]]
5 max_memory_interval = [0, 500]
6 memory_range = 100
7 nbr_visible_address = 10
8 max_nbr_memory_blocks = 4

```

The y-axis represents the probability and the x-axis represents the pattern numbers. A separate graph is generated for each different category. Within each graph, there are three lines: a red line representing generation under normal conditions (completely random), a blue line representing generation with constant variables across exercises, and a green line representing generation when both variables and memory allocations are fixed. These lines are used to evaluate the effect of different components on the generation of exercises.

We can see that fixing the variables does not significantly change the resolution percentages, as the variables have no addresses or values at this stage, leading to relatively similar results (the blue and red lines are close). However, with char variables, the string values are already assigned and so are the addresses to which the variables point. This produces more similar expressions as the only differences are in the choices made during pattern selection and the blue and green lines are very close.

The green line shows the difference within a pattern when everything is fixed (variables and memory). Variations are only made by different choices in the patterns.

For simple exercises, the differences are much smaller and the expressions are more identical. This is due to the fact that there are not many choices, only the variable is chosen and sometimes the number of dereferencing also limits the choice of the variable.

It is the same with the incrementation exercises, only the variable is chosen.

For arithmetic expressions, there are more choices and the difference between the two previous categories is greater. Simple arithmetic expressions (patterns 1-8) consist of addition or subtraction of two operands. The two operands and the operator can be chosen. For the more

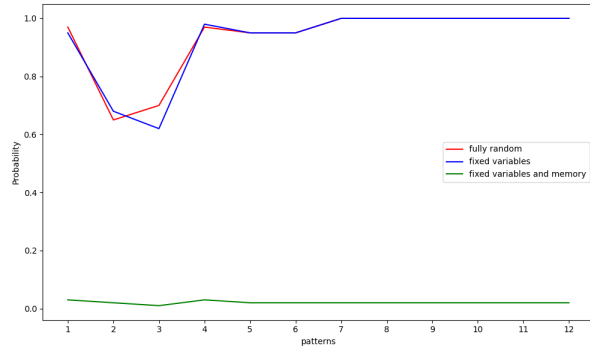
difficult arithmetic (patterns 9-13), more choices are introduced (incrementation or more complex patterns) and the more choices, the more different are the expressions. In the same vein as simple arithmetic, casts introduce another choice, the type of cast (long or short), which increases the difference between simple arithmetic.

For the arrays, patterns 1 (*address[value]*) and 2 (*address[-simple_var]*) only make two variables are chosen, so the difference is small. The other patterns are more complex with more choices.

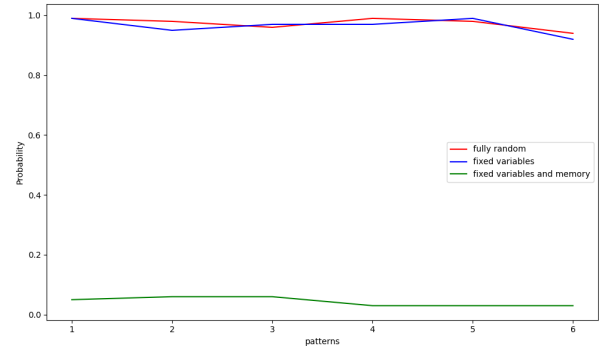
For char expressions, memory changes do not affect the outcome compared to the previous column, as char variables are not part of memory.

For the dereferencing category, we can see some 0 values and they correspond to almost never used patterns seen in the previous section. Other pattern remains relatively different between 17% and 95% depending on the number of choices possible.

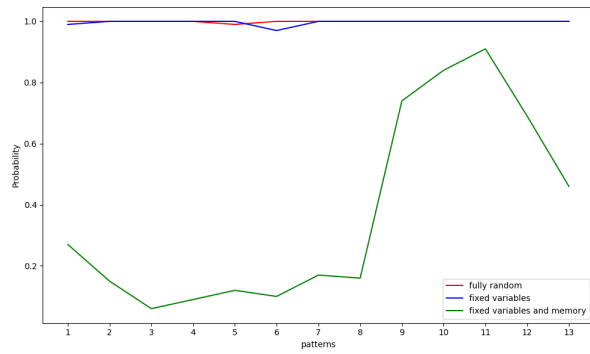
We can conclude that the exercises obtained under normal conditions are very different in resolution due to the different memory. However, when the memory is fixed, the exercises remain different as the difficulty of the pattern and the number of choices increase.



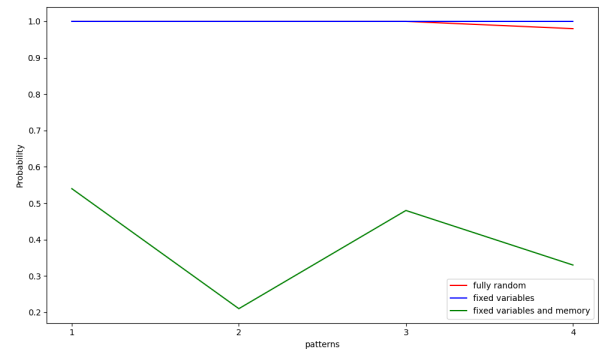
(a) Easy



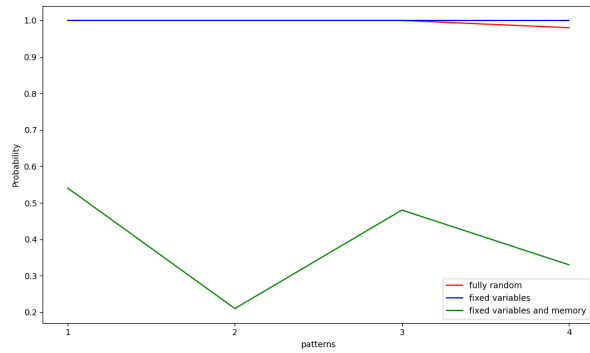
(b) Incr



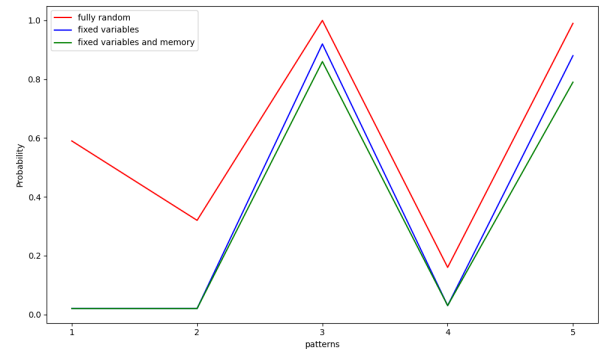
(c) Arithm



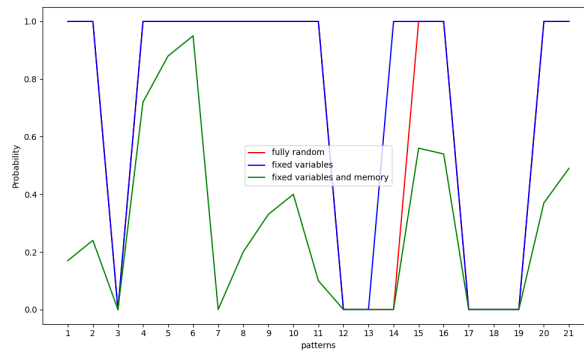
(d) Cast



(e) Array



(f) Char



(g) Defer

Figure 8.3: Differences within patterns : the red line (normal conditions), a blue line (constant variables) and green line (fixed variables and memory)

8.3 Speed of the program

Figure 8.4 shows the speed of the exercise generation obtained with the configuration: Intel Core i7-6500U CPU 2.50GHz, 12 GB RAM, Python 3.8.3. We can see that the program can generate a large number of exercises in a short time.

nbr of exercises	speed (s)
1	0,84
10	0,87
100	1,19
1000	4,01
1000	32,88

Figure 8.4: Speed of the program

Chapter 9

Can ChatGPT solve the exercises?

9.1 Introduction

In this chapter, we examine how online tools like ChatGPT could affect the introduction to computer programming course, with a specific focus on the pointer topic.

We address the following questions:

- Could ChatGPT have a detrimental effect on the homework assessment process? Is there a possibility that students might use it to achieve favorable results without actually understanding the course material?
- Can these tools be used by students to enhance their learning experience? Can they be used not only to help with assignments but also as additional resources to help them better understand programming concepts?

The purpose is not to develop an elaborate script that suits best the exercises but rather to present a simple text that beginner students can easily write.

9.2 Prompts

We tested two different prompts using ChatGPT for this study. The first prompt, described in Appendix section B.1, is the translation of the French exercises given to the students. The second instruction, which is in Appendix section B.2, is the same as the first one, except for one sentence that we removed. This sentence is: “If a memory access is requested at an address outside the range [4, 84], indicate the segmentation error with the value SF”. We discuss in later analyses how this sentence affected the number of times we got wrongly the answer “segmentation fault”.

9.3 Result

The complete output text for the different prompts can be found in the appendices: the output of the first prompt in section B.3 (prompt 1), and the two outputs of the second prompt in section B.4 (prompt 2.1) and section B.5 (prompt 2.2).

The table below shows a summary of the different answers obtained on June 20, 2023, with ChatGPT-3.5. The first two columns list the expressions and the corresponding correct answers. The third column shows the answers obtained using the first prompt, while the last two columns show the answers obtained using the second prompt, which excludes the sentence about segmentation faults.

	Expression	correct answer	Prompt 1	Prompt 2.1	Prompt 2.2
1	<code>g</code>	4	4	4	4
2	<code>&b</code>	80	80	80	80
3	<code>*v</code>	72	72	72	72
4	<code>** (int **)e</code>	20	SF	76	Undefined
5	<code>&** (int **)v</code>	76	SF	76	Undefined
6	<code>++*&e</code>	73	73	73	73
7	<code>*g++</code>	3	3	3	3
8	<code>(*&b)---</code>	1	0	0	0
9	<code>g - e</code>	-284	Undefined	-17	-20
10	<code>v - g</code>	20	20	20	80
11	<code>v - &b</code>	1	1	1	4
12	<code>v--- - --e</code>	-200	9	5	13
13	<code>(short*)v - b</code>	82	2	1	70
14	<code>(short*)&e + b</code>	86	85	74	86
15	<code>v[b + -1]</code>	72	72	72	71
16	<code>v[-17] + g[b]</code>	3	Undefined	73	11
17	<code>G - P</code>	6	6	6	6
18	<code>*G</code>	'a'	't'	'q'	'i'

Table 9.1: Answers from ChatGPT obtained on 20 June 2023 using ChatGPT-3.5.

Out of the 18 expressions, ChatGPT provided 9 correct answers for the first prompt, 10 and 7 for the other prompt.

One important observation is that ChatGPT demonstrates a good understanding of the statement. Despite the abstract nature of the exercise, it is able to establish connections between the variable declaration, the memory state and the expressions. For example, in the case of the first expression (`g`) one of the outputs state: “The variable `g` is an array, and when used in an expression, it decays into a pointer to its first element. Therefore, `g` is equivalent to `&g[0]`. The value of `&g[0]` is the memory address of the first element of the array, which is 4” It shows that ChatGPT first identifies that it needs to resolve the expression “`g`”. Then, based on the declaration, it recognizes that the variable is an integer array. Using its knowledge, it deduces

that the answer is equivalent to the address of the first element of the array and it successfully finds the correct solution in the memory.

This example also highlights that ChatGPT is capable of providing clear explanations of its reasoning, which can be a valuable tool for students to understand unfamiliar expressions. However, it is important to note that ChatGPT is not always reliable, as we see in the following analysis.

As can be seen, ChatGPT faces difficulties with the fourth and fifth expressions: `** (int **)e` and `&** (int **)v`. In the first answer, it struggles to understand the concept of memory range and mistakenly assumes that it is trying to access memory outside the memory range. That is why we remove the sentence about segmentation fault for the other two answers. In the second answer, there is an inconsistency between the two expressions. Since `*v = e` based on the declaration, the fifth expression becomes `&** (int **)e`. The two expressions now appear very similar, both accessing to the same memory space, with one returning the value and the other returning the address. ChatGPT returns for both cases the same answer.

Alongside the inconsistency within the same exercise, we also have inconsistency with the second and third answers despite providing exactly the same input. In fact, for the third answer, it couldn't find an answer.

Regarding the increment or decrement operators (expressions 6-8), ChatGPT provides the correct answer for the first two but struggles with the last one. It has difficulties when the operand is on the right side, returning the value before the increment or decrement. For example, with `*g++`, it correctly dereferences the variable `g` before the increment, but with `(*&b)--`, it returns the value after the decrement.

In the arithmetic pointer category, ChatGPT gives the correct answer two out of three times for the subtraction of two addresses (expressions: `v - g`, `v - &b`) except for the last answer where it forgets to divide by the size of the pointing element.

We can notice that ChatGPT has difficulties with the subtraction of an address and a value. Indeed, for the expression `g - e`, it says for the first answer that this operation is not possible, for the two other answers, it performs the subtraction of two addresses one using the value of "e" (`(g - e)/address size`) and one using the address of e (`(g - &e)/address size`). We have the same problem for the more complex expression `v-- - --e` but with the addition in that case of incrementation error seen previously.

For the cast exercises, the majority of the answers are incorrect, except for one expression in one of the answers. However, the explanation provided is not very detailed. For example, in the third answer, the two expressions `((short*)v - b)`, `(short*)&e + b` have nearly identical explanations and one returns the correct answer while the other does not.

ChatGPT has the correct solution for the two first answers with arrays exercises. The first expression of the third answer, ChatGPT has good reasoning, it understands it has to take the value at the address where `v` is pointing but outputs a wrong answer and surprisingly the value appears nowhere in the memory state or elsewhere in the statement. For the two others it gives

the right answer.

For the second expression (`v[-17] + g[b]`), ChatGPT has difficulties with the negative index. It says it cannot be determined for the first solution. The second solution takes the value at the address of “v” instead of “`v-17*4 = 16`” even if it explains clearly to do: “`v[-17]` gives the value at the memory address `v + (-17) * sizeof(int)`.”. The last answer is also interesting. ChatGPT initially translates `v[-17]` to `*(v-17*sizeof(int))` but when it performs the computation, it mistakenly takes the wrong address for “v” and obtains 3 as the address. Clearly, 3 is not a valid address for an integer. Instead of recognizing that the dereferencing operation is not possible, ChatGPT skips the dereferencing step and assigns 3 as the final value (`v[-17] = 3`). If it obtains a value that is not relevant to the problem, it modifies the problem to obtain an answer at the end.

For the char expressions, ChatGPT understands clearly the first expressions. The particularity of those expressions is that the char variable does not have known pointing values and the answer is obtained only from the declaration. However, for the last expression, ChatGPT has difficulties understanding the expression correctly and provides a wrong answer. Interestingly, the incorrect answers differ among the three responses, even though their explanation contains the correct reasoning.

To conclude this chapter, ChatGPT can provide some insights and reasoning but the performance in resolving the exercises accurately is not satisfactory. While the details provided by ChatGPT can be useful in some cases, relying solely on its response may not be reliable or sufficient for the students aiming for higher scores. To separate truth from falsehood, students need a solid understanding of the course material and the ability to discern correct reasoning from incorrect or inconsistent information. Therefore, while it may have some potential as an additional learning aid, it cannot replace the need for comprehensive study and understanding of the course material. Students should be cautious and critical when using ChatGPT’s responses.

Chapter 10

Conclusion

This Master's thesis investigates the generation of exercises for introductory computer programming courses, with a particular focus on pointer exercises. Our aim was to develop a structured approach that generates a wide range of exercises to help students practice and improve their understanding of this complex subject. The approach also aimed to reduce the workload for teachers in creating pointer exercises and to reduce the likelihood of academic dishonesty. The objective also extends to providing a comprehensive overview of the field of exercise generation, with the aim of inspiring further development in the generation of exercises for different subjects.

A flexible framework for exercises using pattern templates has been developed. Patterns are sufficiently restrictive to allow control over the production of exercises, ensuring that only relevant exercises are provided to students and their concepts are consistent for all students. At the same time, the framework allows for the creation of unique exercises by exploiting the different choices that can be made in the pattern and the uniqueness of the memory and variables.

This study provides an explanation of how to create exercises, starting with the analysis of existing exercises to deduce a general structure, down to the details of implementation. It also presents an approach to providing feedback to students.

Based on the results of our analysis, we revealed the correct functioning of the program by examining the distribution of different patterns and measuring the differences between exercises.

We explored if ChatGPT can serve as an educational tool for programming exercises. By using ChatGPT to solve the generated exercises, we gained insight into its effectiveness in providing suggestions and hints to learners. Although ChatGPT appeared to understand the abstract structure of the exercises and solved half of the expressions, it demonstrates the difficulty of using these tools to achieve accurate results.

To conclude, this master's thesis has contributed to the educational technology field by introducing a systematic method for generating exercises related to pointers. Additionally, it has

provided a comprehensive overview of exercise generation as a whole, inspiring the advancement of exercise generation across various subject areas.

Appendix A

Testing

A.1 C program to test feasibility

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main() {
    int* mem = malloc(21 * sizeof(int));
    int conversion = (int) mem - 4;

    int *e, *b;
    int *v, *g;
    char *P, *G,*E;
    P = malloc(sizeof("informatique")*sizeof(char));

    init( &e, &b, &v, &g, &P, &G, &E, mem);
    printf("g_\t%d\n", (int)g-conversion);

    init( &e, &b, &v, &g, &P, &G, &E, mem);
    printf("&b_\t%d\n", (int)&b-(int)mem +4);

    init( &e, &b, &v, &g, &P, &G, &E, mem);
    printf("*v_\t%d\n", (int)*v-conversion);

    init( &e, &b, &v, &g, &P, &G, &E, mem);
    printf("**_(int_**)e_\t%d\n", (int)** (int **)*e-conversion);

    init( &e, &b, &v, &g, &P, &G, &E, mem);
    printf("&**_(int_**)*v_\n", (int)&** (int **)*v-conversion);

    init( &e, &b, &v, &g, &P, &G, &E, mem);
    printf("++&e_\t%d\n", (int)++*&*e-conversion);

    init( &e, &b, &v, &g, &P, &G, &E, mem);
    printf("*g++_\t%d\n", *g++);
```

```

init( &e, &b, &v, &g, &P, &G, &E, mem);
printf("(*&b)---\t%d\n", (*&*b)---);

init( &e, &b, &v, &g, &P, &G, &E, mem);
printf("g-e-\t%d\n", (int)(g - (*e-conversion)) - conversion);

init( &e, &b, &v, &g, &P, &G, &E, mem);
printf("v-g-\t%d\n", v-g);

init( &e, &b, &v, &g, &P, &G, &E, mem);
printf("v-&b-\t%d\n", v-&*b);

init( &e, &b, &v, &g, &P, &G, &E, mem);
printf("v---e-\t%d\n", (int)(v--- - (---*e-conversion)) - conversion);

init( &e, &b, &v, &g, &P, &G, &E, mem);
printf("(short*)v-b-\t%d\n", (int)((short *)v-*b)-conversion);

init( &e, &b, &v, &g, &P, &G, &E, mem);
printf("(short*)&e+b-\t%d\n", (int)((short *)&*e+*b)-conversion);

init( &e, &b, &v, &g, &P, &G, &E, mem);
printf("v[b+-1]-\t%d\n", v[*b+ -1]-conversion);

init( &e, &b, &v, &g, &P, &G, &E, mem);
printf("v[-17]-+g[b]-\t%d\n", v[-17] + g[*b]);

init( &e, &b, &v, &g, &P, &G, &E, mem);
printf("G-P-\t%d\n", G-P);

init( &e, &b, &v, &g, &P, &G, &E, mem);
printf("G-\t%c\n", *G);

init( &e, &b, &v, &g, &P, &G, &E, mem);
printf("*(&e+-3)-\t%d\n", *(&*e+ -3) - conversion);

init( &e, &b, &v, &g, &P, &G, &E, mem);
printf("*(v=g)-\t%d\n", *(v=g));

return 0;
}

```

```

void init(int** e, int** b, int** v, int** g, char** P, char** G, char** E, int* mem){
    mem[20] = &mem[17];
    mem[19] = 1;
    mem[18] = &mem[4];
    mem[17] = &mem[18];
    mem[6] = &mem[18];
    mem[4] = &mem[6];
    mem[3] = 2;
    mem[2] = 2;
    mem[1] = 1;
}

```

```
mem[0] = 3;

*e = &mem[20];
*b = &mem[19];

*v = &**e;
*g = &mem[0];
strcpy(*P, "informatique");

*G = *P+6;
*E=*G+3;
}
```

Appendix B

ChatGPT

B.1 Prompt 1

Your goal is to resolve this C language exercise. You have first the variable declaration (assuming int is using 4 bytes, shorts use 2 bytes, longs use 8 bytes).

Variable declaration :

```
int e, b, *v = &e,
int g[] = {3,1,2,2};
char *P = "Informatique";char *G = P + 6;char *E = G + 3;
```

The following table is the representation of the memory. The first column represents the variable names, the second column represents the memory addresses (expressed in base 10), and the third column represents the values stored at those addresses

Memory :

variables	name	memory addresses	values
e	84	72	
b	80	1	
	76	20	
	72	76	
	
	28	76	
	
	20	28	
	16	2	
	12	2	
	8	1	
g[0]	4	3	

you need to determine the values of the following expressions. Consider that the memory is reset to the state presented in the schema between each expression. If a memory access is requested at an address outside the range [4, 84], indicate the segmentation error with the value SF

Expressions :

1. `g`
2. `&b`
3. `*v`
4. `** (int **)e`
5. `&** (int **)v`
6. `++*&e`
7. `*g++`
8. `(*&b)---`
9. `g - e`
10. `v - g`
11. `v - &b`
12. `v--- - ---e`
13. `(short*)v - b`
14. `(short*)&e + b`
15. `v[b + -1]`
16. `v[-17] + g[b]`
17. `G - P`
18. `*G`

B.2 Prompt 2

Your goal is to resolve this C language exercise. You have first the variable declaration (assuming `int` is using 4 bytes, `shorts` use 2 bytes, `longs` use 8 bytes).

Variable declaration :

```
int e, b, *v = &e,
int g[] = {3,1,2,2};
char *P = "Informatique"; char *G = P + 6; char *E = G + 3;
```

The following table is the representation of the memory. The first column represents the variable names, the second column represents the memory addresses (expressed in base 10), and the third column represents the values stored at those addresses

Memory :

variables	name	memory addresses	values
e	84	72	
b	80	1	
	76	20	

```

72 76
... ...
28 76
... ...
20 28
16 2
12 2
8 1
g[0] 4 3

```

you need to determine the values of the following expressions. Consider that the memory is reset to the state presented in the schema between each expression.

Expressions :

1. g
2. $\&b$
3. $*v$
4. $** (\text{int } **)e$
5. $\&** (\text{int } **)*v$
6. $++*\&e$
7. $*g++$
8. $(*\&b)--$
9. $g - e$
10. $v - g$
11. $v - \&b$
12. $v--- - ---e$
13. $(\text{short}*)v - b$
14. $(\text{short}*)&e + b$
15. $v[b + -1]$
16. $v[-17] + g[b]$
17. $G - P$
18. $*G$

B.3 ChatGPT output with prompt 1

1. g : The variable g is an array, and when used in an expression, it decays into a pointer to its first element. Therefore, g is equivalent to $\&g[0]$. The value of $\&g[0]$ is the memory address of the first element of the array, which is 4.
2. $\&b$: The address-of operator $\&$ gives us the memory address of the variable b . According to the memory table, b is located at address 80.
3. $*v$: v is a pointer that is initialized with the address of e . Therefore, $*v$ dereferences the pointer and gives us the value stored at the address it points to, which is e . From the

memory table, we can see that e is located at address 84, and its value is 72.

4. $**(\text{int } **)e$: The variable e holds the value 72, which is treated as a memory address. We cast it to an $\text{int } **$ pointer type, then dereference it twice to access the value at that address. However, the memory address 72 is outside the range provided in the memory table, so it results in a segmentation fault (SF).
5. $\& **(\text{int } **) * v$: $*v$ gives us the value of e , which is 72. We then cast it to an $\text{int } **$ pointer type, dereference it to get the value at that address (which is an $\text{int } *$ pointer), and finally, take the address of that $\text{int } *$ pointer. Since the value of e is outside the range provided in the memory table, it results in a segmentation fault (SF).
6. $++\&e$: $\&e$ gives us the address of e , which is 84. $\&e$ dereferences the pointer and gives us the value at that address, which is 72. Finally, $++72$ increments the value of 72 by 1, resulting in 73.
7. $*g++$: g is an array that decays into a pointer to its first element. $*g++$ dereferences the pointer and gives us the value at that address, which is 3. Afterward, the pointer g is incremented to point to the next element of the array.
8. $(\&b) --$: $\&b$ gives us the address of b , which is 80. $\&b$ dereferences the pointer and gives us the value at that address, which is 1. $(\&b) --$ decrements the value of 1 by 1, resulting in 0.
9. $g - e$: Subtracting two pointers gives the number of elements between them. Since g is an array pointer and e is a regular variable, they cannot be meaningfully subtracted. This operation is not defined.
10. $v - g$: Subtracting two pointers gives the number of elements between them. v is a pointer to e , which has the address 84, and g is a pointer to the first element of the array g , which has the address 4. Therefore, $v - g$ gives us $\frac{84-4}{\text{sizeof}(\text{int})}$, which is $\frac{80}{4}$, resulting in 20.
11. $v - \&b$: Subtracting two pointers gives the number of elements between them. v is a pointer to e , which has the address 84, and $\&b$ gives us the address of b , which is 80. Therefore, $v - \&b$ gives us $\frac{84-80}{\text{sizeof}(\text{int})}$, which is $\frac{4}{4}$, resulting in 1.
12. $v - - - - e$: $--e$ decrements the value of e by 1, resulting in 71. $v - -$ decrements the value of v (which is the address of e) by the size of int , which is 4. Therefore, $v - - - - e$ gives us $84 - 4 - 71$, which is 9.
13. $(\text{short}*)v - b$: Cast v to a $\text{short}*$ pointer type and subtract b from it. Since v is a pointer to e , which has the address 84, and b is located at address 80, $(\text{short}*)v - b$ gives us $\frac{84-80}{\text{sizeof}(\text{short})}$, which is $\frac{4}{2}$, resulting in 2.
14. $(\text{short}*)&e + b$: $\&e$ gives us the address of e , which is 84. Casting it to a $\text{short} **$ pointer type gives us the address of e as a $\text{short} *$ pointer. Adding b , which is 1, to it gives us the memory address $(\text{short}*)&e + b$, which is $84 + 1$, resulting in 85.
15. $v[b + -1]$: b is equal to 1. Therefore, $b + -1$ is equal to $1 + (-1)$, which is 0. $v[0]$ accesses

the element at index 0 in the array pointed to by v , which is e . From the memory table, we know that the value of e is 72.

16. $v[-17] + g[b]$: $v[-17]$ attempts to access an element at an invalid negative index, resulting in undefined behavior. The value cannot be determined.
17. $G - P$: G is a pointer to a character in the string "Informatique" starting from the sixth character. P is a pointer to the first character of the string. Subtracting P from G gives us the number of characters between them, which is $6 - 0$, resulting in 6.
18. $*G$: G is a pointer to a character in the string "Informatique" starting from the sixth character. $*G$ dereferences the pointer and gives us the character at that address, which is 't'.

B.4 ChatGPT output with prompt 2 (first try)

1. g : g is an array, and when used in an expression, it decays into a pointer to its first element. Therefore, g is equivalent to $\&g[0]$. The value is the memory address of $g[0]$, which is 4.
2. $\&b$: $\&b$ gives the memory address of variable b . The value is 80.
3. $*v$: v is a pointer to e , and $*v$ gives the value stored at the memory address v points to. The value is the value of e , which is 72.
4. $**(\text{int } **)e$: $(\text{int } **)e$ interprets the value of e as a pointer to a pointer to an integer. $**(\text{int } **)e$ gives the value at the address pointed to by the value of e . The value is the value at memory address 72, which is 76.
5. $\&**(\text{int } **)*v$: $*v$ gives the value of e . $(\text{int } **)e$ interprets the value of e as a pointer to a pointer to an integer. $**(\text{int } **)e$ gives the value at the address pointed to by the value of e . $\&**(\text{int } **)e$ gives the memory address of the value at the address pointed to by the value of e . The value is 76.
6. $++* \&e$: $* \&e$ gives the value of e . $++* \&e$ increments the value of e . The value is 73.
7. $*g++$: $g++$ increments the pointer g to point to the next element in the array. $*g++$ dereferences the original value of g (before the increment). The value is the value at memory address 4, which is 3.
8. $(* \&b)---$: $\&b$ gives the memory address of b . $* \&b$ gives the value of b . $(* \&b)---$ decrements the value of b . The value is 0.
9. $g - e$: g is a pointer to its first element, which is equivalent to $\&g[0]$. $g - e$ subtracts the value of e from the value of g , divided by the size of g 's elements. The value is $(4 - 72) / \text{sizeof(int)}$, which is -17.
10. $v - g$: v is a pointer to e . g is a pointer to its first element, which is equivalent to $\&g[0]$. $v - g$ subtracts the memory address of g from the memory address of v , divided by the

- size of v 's elements. The value is $(84 - 4) / \text{sizeof}(\text{int} *)$, which is 20.
11. $v - \&b$: v is a pointer to e . $\&b$ gives the memory address of b . $v - \&b$ subtracts the memory address of b from the memory address of v , divided by the size of v 's elements. The value is $(84 - 80) / \text{sizeof}(\text{int} *)$, which is 1.
 12. $v-- - --e$: $--e$ decrements the value of e . $v--$ decrements the memory address stored in v (post-decrement). $v-- - --e$ subtracts the value of $--e$ from the memory address of $v--$. The value is $76 - 71$, which is 5.
 13. $(\text{short} *)v - b$: $(\text{short} *)v$ interprets the value of v as a pointer to a short (2 bytes). $(\text{short} *)v - b$ subtracts the memory address of b from the memory address of $(\text{short} *)v$, divided by the size of $(\text{short} *)v$'s elements. The value is $(84 - 80) / \text{sizeof}(\text{short})$, which is 1.
 14. $(\text{short} *)&e + b$: $\&e$ gives the memory address of e . $(\text{short} *)&e$ interprets the memory address of e as a pointer to a short (2 bytes). $(\text{short} *)&e + b$ adds the value of b to the memory address of $(\text{short} *)&e$, multiplied by the size of $(\text{short} *)&e$'s elements. The value is $(72 + (1 * \text{sizeof}(\text{short})))$, which is 74.
 15. $v[b + -1]$: v is a pointer to e . b is the value of variable b . $v[b + -1]$ gives the value at the memory address $v + (b + -1) * \text{sizeof}(\text{int})$. The value is the value at memory address $84 + (1 + -1) * \text{sizeof}(\text{int})$, which is the value at memory address 84, which is 72.
 16. $v[-17] + g[b]$: v is a pointer to e . g is a pointer to its first element, which is equivalent to $\&g[0]$. $v[-17]$ gives the value at the memory address $v + (-17) * \text{sizeof}(\text{int})$. $g[b]$ gives the value at the memory address $g + b * \text{sizeof}(\text{int})$. The value is the sum of the value at memory address $84 + (-17) * \text{sizeof}(\text{int})$ and the value at memory address $4 + 1 * \text{sizeof}(\text{int})$. The value is the sum of the value at memory address 84 and the value at memory address 8, which is $72 + 1$, which is 73.
 17. $G - P$: G and P are pointers to characters. $G - P$ subtracts the memory address of P from the memory address of G . The value is $(P + 6) - P$, which is 6.
 18. $*G$: G is a pointer to a character. $*G$ gives the value at the memory address G . The value is the value at memory address $(P + 6)$, which is 'q'.

B.5 ChatGPT output with prompt 2 (second try)

To determine the values of the expressions, let's analyze each expression step by step:

1. g : g is an array, and when used in an expression, it decays into a pointer to its first element. Therefore, g is equivalent to $\&g[0]$. The value is the memory address of $g[0]$, which is 4.
2. $\&b$: $\&b$ gives the address of variable b . The value is 80.

3. `*v` : `v` is a pointer to `e`, so `*v` dereferences the pointer and gives the value stored at the memory address it points to. The value is 72.
4. `** (int **)e` : `(int **)e` interprets the value of `e` as a pointer to a pointer to an int. The value of `e` is 72, which is interpreted as a memory address. `** (int **)e` dereferences this memory address twice and gives the value stored at that location. Note: This expression involves typecasting, which can lead to undefined behavior if the memory address stored in `e` is not a valid pointer to a pointer to an int. Make sure the value of `e` is set accordingly.
5. `&** (int **)v` : `*v` gives the value stored at the memory address `v` points to, which is 72. `(int **)v` interprets 72 as a memory address for a pointer to a pointer to an int. `** (int **)v` dereferences this memory address twice and gives the value stored at that location. `&** (int **)v` takes the address of this value. The value is the memory address of the value stored at 72.
6. `++*&e` : `&e` gives the address of `e`, which is 84. `*(&e)` dereferences this address and gives the value stored at 84, which is 72. `++*&e` increments this value. The value becomes 73, and `e` is updated to 73.
7. `*g++` : `g++` increments the pointer `g` to point to the next element (`g[1]`). `*g++` dereferences the original pointer `g` before incrementing. The value is the value stored at the original memory address of `g[0]`, which is 3.
8. `(*&b)---` : `&b` gives the address of `b`, which is 80. `*&b` dereferences this address and gives the value stored at 80, which is 1. `(*&b)---` decrements this value. The value becomes 0, and `b` is updated to 0.
9. `g - e` : `g` is an array, and when used in an expression, it decays into a pointer to its first element. The value of `g - e` is the arithmetic difference between the memory addresses of `g[0]` and `e`. `g[0]` is at memory address 4, and `e` is at memory address 84. The value is -20.
10. `v - g` : `v` is a pointer to `e`, so it points to the memory address of `e`, which is 84. `g` is an array, and when used in an expression, it decays into a pointer to its first element, which is at memory address 4. The value of `v - g` is the arithmetic difference between the memory addresses pointed to by `v` and `g`. The value is 80.
11. `v - &b` : `v` is a pointer to `e`, so it points to the memory address of `e`, which is 84. `&b` gives the address of `b`, which is 80. The value of `v - &b` is the arithmetic difference between the memory addresses pointed to by `v` and `&b`. The value is 4.
12. `v-- - e` : `-e` decrements the value of `e` before the evaluation, so `e` becomes 71. `v--` decrements the value of `v` after the evaluation, but the value of the expression is the value of `v` before the decrement. `v--` evaluates to the memory address pointed to by `v`, which is 84. The value is $84 - 71 = 13$, and `v` is updated to 71.
13. `(short*)v - b` : `(short*)v` interprets `v` as a pointer to a short. The value of `b` is 1. `(short*)v - b` subtracts the value of `b` from the interpreted value of `v`. The value is 70.

14. $(\text{short}^*)\&e + b$: $\&e$ gives the address of e , which is 84. $(\text{short}^*)\&e$ interprets this address as a pointer to a short. The value of b is 1. $(\text{short}^*)\&e + b$ adds the value of b to the interpreted value of $\&e$. The value is 86.
15. $v[b + -1]$: The value of b is 1. $b + -1$ is $1 + (-1) = 0$. $v[0]$ is equivalent to $*v$, which gives the value stored at the memory address v points to. The value is 71.
16. $v[-17] + g[b]$: $v[-17]$ gives the value stored at the memory address $v - 17 * \text{sizeof}(\text{int})$. The value is the value stored at the memory address $71 - 17 * 4 = 71 - 68 = 3$. The value of b is 1. $g[b]$ is equivalent to $*(g + b)$, which gives the value stored at the memory address $g + b * \text{sizeof}(\text{int})$. The value is the value stored at the memory address $4 + 1 * 4 = 8$. The result is $3 + 8 = 11$.
17. $G - P$: G is a pointer to $G[0]$, which is the character 'i'. P is a pointer to $P[0]$, which is the character 'I'. The value of $G - P$ is the arithmetic difference between the memory addresses pointed to by G and P . The value is 6.
18. $*G$: G is a pointer to $G[0]$, which is the character 'i'. $*G$ dereferences this pointer and gives the value stored at the memory address it points to. The value is 'i'.

Bibliography

- [1] Manish Agarwal and Prashanth Mannem. Automatic gap-fill question generation from text books. In Proceedings of the 6th Workshop on Innovative Use of NLP for Building Educational Applications, IUNLPBEA '11, page 56–64, USA, 2011. Association for Computational Linguistics.
- [2] Umair Z. Ahmed, Sumit Gulwani, and Amey Karkare. Automatically generating problems and solutions for natural deduction. In Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence, IJCAI '13, page 1968–1975. AAAI Press, 2013.
- [3] Itziar Aldabe, Maddalen Lopez de Lacalle, Montse Maritxalar, Edurne Martinez, and Larraitz Uria. Ariketurri: An automatic question generator based on corpora and nlp techniques. volume 4053, pages 584–594, 06 2006.
- [4] José João Almeida, Eliana Grande, and Georgi Smirnov. Context-Free Grammars: Exercise Generation and Probabilistic Assessment. In Marjan Mernik, José Paulo Leal, and Hugo Gonçalo Oliveira, editors, 5th Symposium on Languages, Applications and Technologies (SLATE'16), volume 51 of OpenAccess Series in Informatics (OASIs), pages 10:1–10:8, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
- [5] Chris Alvin, Sumit Gulwani, Rupak Majumdar, and Supratik Mukhopadhyay. Automatic synthesis of geometry problems for an intelligent tutoring system, 2015.
- [6] Tiffany Barnes and John Stamper. Toward automatic hint generation for logic proof tutoring using historical student data. In Proceedings of the 9th International Conference on Intelligent Tutoring Systems, ITS '08, page 373–382, Berlin, Heidelberg, 2008. Springer-Verlag.
- [7] Lee Becker, Sumit Basu, and Lucy Vanderwende. Mind the gap: Learning to choose gaps for question generation. pages 742–751, 06 2012.
- [8] Okonkwo Chinedu and Abejide Ade-Ibijola. Synthesis of nested loop exercises for practice in introductory programming. Egyptian Informatics Journal, 24:191–203, 03 2023.
- [9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Tamar Solorio, editors, Proceedings of the 2019 Conference of the North

- American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers), pages 4171–4186. Association for Computational Linguistics, 2019.
- [10] Christopher Douce, David Livingstone, and James Orwell. Automatic test-based assessment of programming: A review. J. Educ. Resour. Comput., 5(3):4–es, sep 2005.
- [11] Iddo Drori, Sarah Zhang, Reece Shuttleworth, Leonard Tang, Albert Lu, Elizabeth Ke, Kevin Liu, Linda Chen, Sunny Tran, Newman Cheng, Roman Wang, Nikhil Singh, Taylor L. Patti, Jayson Lynch, Avi Shporer, Nakul Verma, Eugene Wu, and Gilbert Strang. A neural network solves, explains, and generates university math problems by program synthesis and few-shot learning at human level. Proceedings of the National Academy of Sciences, 119(32), aug 2022.
- [12] Andrew DuFrene. Automatic generation and grading of programming exercises. 2016.
- [13] Peggy Ertmer, Jennifer Richardson, Brian Belland, Denise Camin, Patrick Connolly, Glen Coulthard, Kimfong Lei, and Christopher Mong. Using peer feedback to enhance the quality of student online postings: An exploratory study. ITLS Faculty Publications, 12, 01 2007.
- [14] George E Forsythe and Niklaus Wirth. Automatic grading programs. Communications of the ACM, 8(5):275–278, 1965.
- [15] Kenneth V. Hanford. Automatic generation of test cases. IBM Systems Journal, 9(4):242–257, 1970.
- [16] J. B. Hext and J. W. Winings. An automatic grading scheme for simple programming exercises. Commun. ACM, 12(5):272–275, may 1969.
- [17] Jack Hollingsworth. Automatic graders for programming classes. 3(10):528–529, oct 1960.
- [18] Edmond Holohan, Mark Melia, Declan McMullen, and Claus Pahl. The generation of e-learning exercise problems from subject ontologies. Holohan, Edmond and Melia, Mark and McMullen, Declan and Pahl, Claus (2006) The generation of e-learning exercise problems from subject ontologies. In: ICALT 2006 - 6th International Conference on Advanced Learning Technologies, 5-7 July 2006, Kerkrade, The Netherlands. ISBN 0-7695-2632-2, 2006, 01 2006.
- [19] Peter C. Isaacson and Terry A. Scott. Automating the execution of student programs. SIGCSE Bull., 21(2):15–22, jun 1989.
- [20] Tassilo Klein and Moin Nabi. Learning to answer by learning to ask: Getting the best of gpt-2 and bert worlds, 2019.
- [21] Zhenwen Liang, Wenhao Yu, Tanmay Rajpurohit, Peter Clark, Xiangliang Zhang, and Ashwin Kaylan. Let gpt be a math tutor: Teaching math word problem solvers with customized exercise generation, 2023.

- [22] Ruslan Mitkov and Le An Ha. Computer-aided generation of multiple-choice tests. In Proceedings of the HLT-NAACL 03 Workshop on Building Educational Applications Using Natural Language Processing - Volume 2, HLT-NAACL-EDUC '03, page 17–22, USA, 2003. Association for Computational Linguistics.
- [23] Andreas Papasalouros. Automatic exercise generation in euclidean geometry. In Harris Papadopoulos, Andreas S. Andreou, Lazaros Iliadis, and Ilias Maglogiannis, editors, 9th Artificial Intelligence Applications and Innovations (AIAI), volume AICT-412 of Artificial Intelligence Applications and Innovations, pages 141–150, Paphos, Greece, 2013. Springer. Part 4: Problem Solving, Planning and Scheduling.
- [24] Phitchaya Mangpo Phothilimthana and Sumukh Sridhara. High-coverage hint generation for massive courses: Do automated hints help cs1 students? In Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE '17, page 182–187, New York, NY, USA, 2017. Association for Computing Machinery.
- [25] Thomas William Price, Yihuan Dong, and Tiffany Barnes. Generating data-driven hints for open-ended programming. In Educational Data Mining, 2016.
- [26] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. OpenAI blog, 1(8):9, 2019.
- [27] Danijel Radošević, Tihomir Orehovački, and Zlatko Stapić. Automatic on-line generation of student’s exercises in teaching programming. volume 1, 09 2010.
- [28] Kenneth A. Reek. The try system -or- how to avoid testing student programs. SIGCSE Bull., 21(1):112–116, feb 1989.
- [29] Kelly Rivers and Kenneth Koedinger. Automating hint generation with solution space path construction. 8474:329–339, 06 2014.
- [30] Reudismam Rolim, Gustavo Soares, Loris D’Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. Learning syntactic program transformations from examples. In 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pages 404–415. IEEE, 2017.
- [31] Dorsa Sadigh, Sanjit A. Seshia, and Mona Gupta. Automating exercise generation: A step towards meeting the mooc challenge for embedded systems. In Proceedings of the Workshop on Embedded and Cyber-Physical Systems Education, WESE '12, New York, NY, USA, 2012. Association for Computing Machinery.
- [32] Riku Saikkonen, Lauri Malmi, and Ari Korhonen. Fully automatic assessment of programming exercises. volume 33, pages 133–136, 09 2001.
- [33] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. Automatic generation of programming exercises and code explanations using large language models. In Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 1, ICER '22, page 27–43, New York, NY, USA, 2022. Association for Computing Machinery.

- [34] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. SIGPLAN Not., 48(6):15–26, jun 2013.
- [35] Rohit Singh, Sumit Gulwani, and Sriram Rajamani. Automatically generating algebra problems. In Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, AAAI’12, page 1620–1627. AAAI Press, 2012.
- [36] Yvonne Skalban, Lucia Specia, Ruslan Mitkov, et al. Automatic question generation in multimedia-based learning. In Proceedings of COLING 2012: Posters, pages 1151–1160, 2012.
- [37] Ezekiel Soremekun, Esteban Pavese, Nikolas Havrikov, Lars Grunske, and Andreas Zeller. Inputs from hell:. IEEE Transactions on Software Engineering, 48(4):1138–1153, 2022.
- [38] Peter N. Sovietov. Automatic generation of programming exercises. 2021 1st International Conference on Technology Enhanced Learning in Higher Education (TELE), pages 111–114, 2021.
- [39] Joseph Weizenbaum. Eliza—a computer program for the study of natural language communication between man and machine. Commun. ACM, 9(1):36–45, jan 1966.
- [40] John H. Wolfe. Automatic question generation from text - an aid to independent study. SIGCUE Outlook, 10(SI):104–112, feb 1976.