Masteroppgave

NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for datateknologi og informatikk

Daniel Romanich

Generating program tracing exercises for introductory programming courses

Masteroppgave i Datateknologi
Veileder: Guttorm Sindre
Juni 2020

NTNU
Kunnskap for en bedre verden

Daniel Romanich

Generating program tracing exercises for introductory programming courses

Masteroppgave i Datateknologi
Veileder: Guttorm Sindre
Juni 2020

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for datateknologi og informatikk

NTNU
Kunnskap for en bedre verden

Abstract

Due to programming being more and more popular, CS1 courses are becoming more prominent. With the increased student mass, comes an increase in workload for the professors, lecturers, and teaching assistants responsible for the courses. They have to spend an increasing amount of time correcting exercises, rather than helping the students. A solution to this problem is to generate exercises and auto-correct them.

This thesis looks at two ways of generating program tracing exercises using templates, namely the CFG and the SCT approaches, and implements the CFG approach into a working prototype. Through iterative evaluations and a final test, the suitability of the CFG approach was evaluated. The analysis of the results suggests several advantages over the SCT approach, mainly by allowing for more advanced and exciting exercises to be generated. Several drawbacks with the technique were identified, including a large amount of boilerplate work required to set up and constraints on generating exercises for different target languages. However, the advantages outweigh the drawbacks, indicating that the CFG approach is a good technique for auto-generating program tracing exercises.

Sammendrag

Programmering har de siste årene blitt mer og mer populært, som et resultat har førsteårs-programmringskurs fått en betydelig økning i antall studenter. Med den økte student-massen kommer en økning i arbeidsmengden for professorene, foreleserne og undervis-ningsassistentene som er ansvarlige for kursene. De må bruke stadig mer tid på å rette øvinger, i stedet for å hjelpe studentene. En løsning på dette problemet er å både generere og rette øvinger automatisk.

Denne masteroppgaven ser på to ulike teknikker for å generere kodeforståelsesoppgaver på ved hjelp av maler: CFG og SCT teknikkene, og implementerer CFG teknikken i en fungerende prototype. Gjennom iterative evalueringer og en avsluttende test ble egne-theten til CFG teknikken evaluert. Analysen av resultatene viser flere fordeler med CFG i forhold til SCT, som hovedsakelig ligger i mulighetene for generering av mer avanserte og spennende oppgaver. Flere ulemper med teknikken ble også identifisert, blant annet at det kreves en stor mengde grunnarbeid for å sette opp et templatespråk, i tillegg til begren-sninger for generering av øvinger for flere forskjellige programmeringsspråk. Fordelene oppveier imidlertid ulempene, som indikerer at CFG teknikken er en god tilnærming for auto-generering av kodeforståelsesoppgaver.

Preface

I would like to offer a big thank you to my supervisor, Professor Guttorm Sindre, for providing valuable academic guidance and feedback throughout the project. Additionally, I would like to extend my gratitude to Børge Haugset, a professor at NTNU, for providing insightful feedback throughout the project.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Programming has seen a significant surge in popularity over the past decades. Naturally, courses teaching programming has also followed this trend. As a result, introductory programming courses are taught to large amounts of students at a time. Creating and correcting exercises to evaluate the students' knowledge during and after the course has become tedious, requiring much work from both the professors and the teaching assistants. Being able to automate parts of this process can, therefore, save much unnecessary time spent, and thus improve the courses by allowing the staff to spend more time helping the students, rather than creating and correcting exercises.

Ozmen and Altun has shown that one of the biggest causes of failure for students learning to program is the lack of practice [6]. When students are given the same exercises, the threshold for copying each other's answers is drastically reduced, resulting in reduced learning outcomes. By providing every student with unique exercises, the likelihood of cheating is massively reduced [7]. Creating one set of exercises to evaluate students is time-consuming. Creating individual exercises for every student by hand is, therefore, not feasible. Being able to auto-generate such exercises, accompanied by their correct answers, can prove itself a useful tool for reducing cheating.

Rørnes et al. [8] conducted a study exploring students' mental models when completing exercises related to references in Python. They found that students have trouble understanding simple assignment models, indicating a failure in understanding essential concepts of programming. A study conducted by Lahtien et al. [9] found that one of the best ways to learn programming concepts is by practical exercises. Providing a broad set of exercises, focusing on the simple, but essential concepts, could allow the students to practice and increase their understanding of the relevant concepts properly.

Another advantage of using unique exercises can be seen in an exam situation. Sindre and

Chirumamilla [10] showed that making exams unique can drastically increase the threshold of cheating, since no exam will have the same exercise/answer pairs. Uniquely generated exercises can, therefore, provide a safer exam environment by reducing the likelihood of cheating.

While there are many benefits related to auto-generating exercises, there are several issues that must be taken into account. In situations where the results of the exercises are graded, all exercises must be of the same difficulty. Without this assurance, the results are hard to determine, as the students' will not be evaluated on the same premises. It is therefore clear that knowing the difficulty of every exercise generated is required when used in exam environments. Additionally, the generated exercises should expose proper code conventions to avoid teaching students bad code practices.

1.2 Research Questions

The main focus of the project is to look at existing ways of creating templates for generating programming exercises, as well as finding the best approach for generating program tracing exercises for CS1 courses. The project focuses on the curriculum of the CS1 courses at NTNU, but the results are generalized for most CS1 courses. As a result, four main research questions looking at existing solutions and the best-fit solution for CS1 courses at NTNU as well as technical limitations regarding the approach have been outlined:

1. **Research question 1:** What solutions of auto-generating programming-exercises have already been proposed in research-literature?
2. **Research question 2:** What is the best approach for generating a large amount of program tracing exercises?
3. **Research question 3:** Which parts of CS1 courses would the generated exercises be most useful for?
4. **Research question 4:** Are there any significant technical limitations related to the approach chosen for generating exercises?

1.3 Scope

Currently, there are a lot of different exercise types used when teaching CS1 courses. It is not feasible to generate all of those exercise types. The main focus of this project is about generating program tracing exercises in Python for CS1 courses at NTNU. The two main courses in focus are Information Technology, Introduction (TDT4110/TDT4109) [11] [12]. This course's primary focus is to learn the basic concepts and elements in practical, procedure-oriented programming. As a result, only Python 3.x programs will be generated by the prototype.

The artifact is designed as a prototype, and is therefore not a complete system. Its focus is on generating exercises from templates, and therefore lacks proper infrastructure to be

set up as a fully functional system for generating, distributing, and correcting exercises. The main focus of the project is to study whether the chosen approach is a good way of generating program tracing exercises for the CS1 courses at NTNU or not. The code for the generator can be found at Autogenerator.

1.4 Outline

The report is structured in the following manner:

- **Chapter 1:** Introduces the thesis and its motivation.
- **Chapter 2:** Provides relevant background information on typical programming exercises and a previous project on the topic.
- **Chapter 3:** Describes related work on auto-generating exercises and relevant technologies.
- **Chapter 4:** Presents relevant research methodology, and its relevance to the report.
- **Chapter 5:** Reveals the results from the research, as well as relevant information to re-create the prototype.
- **Chapter 6:** Discusses and interprets the results from the previous section regarding the research questions.
- **Chapter 7:** Concludes the research questions and lists future work and research that can be done in the area.

Chapter 2

Background

2.1 Exercise types

A vast number of different exercise types exist, usually targeting different concepts and areas. As a result, it is crucial to investigate and evaluate the most common types to determine their suitability for the project at hand.

2.1.1 Code writing exercises

Code writing exercises are often considered the best way to measure the students' ability to create code. The BRACElet project investigated the hierarchy of programming skills [13]. They found that the bottom of the hierarchy consisted of basic constructs such as if statements, while the top of the hierarchy represents the ability to write code. However, it is hard to estimate the difficulty of code writing exercises [14]. There are several issues related to code-writing exercises:

- Code writing exercises usually require students to understand and utilize multiple concepts, even though the exercise was designed to look at only one concept.
- Weak students tend to spend a large amount of time on solving code-writing exercises, presenting limitations on how many of these exercises it is feasible to utilize as obligatory exercises.
- Weak students lacking the understanding of underlying concepts might try and fail, then look for examples that solve the problem, potentially providing the correct answer without understanding why.
- There are usually several solutions to a particular code writing exercise; some students might solve the problem without actually utilizing the concepts the exercise was supposed to expose.

Venables et al. [15] conducted a study comparing program tracing, program explanation and code writing in novice programmers and their relationship with the hierarchy of programming skills. The paper investigated the relationship between program tracing, explanation, and code writing in an exam containing three code writing, two code explaining, and one tracing exercise. By analyzing the results, they found a relationship between the three exercise types. Among the results, they found that the first skill the students acquire is the ability to trace. Once their tracing capability becomes reliable, the ability to explain the code emerges. When both of the former abilities are in place, the students' ability to write code improves. In essence, it was discovered that students capable of tracing and explaining what a selected piece of code does, are more likely to be able to write code. However, the results are purely based on statistics and thus do not prove, but rather imply the connection between these exercise types. While the results also imply a hierarchy of programming skills, this hierarchy might not be totally strict. Writing code means that the student needs to understand what the code does and explain it, while also needing to trace it. It is therefore clear that code writing might provide opportunities to improve code tracing and explaining skills, which in return improves the code writing skills.

A proposed way of testing the students' ability to write code is the Soloway's Rainfall problem [16]. The problem is aimed at testing multiple different concepts relevant to the curriculum of CS1 courses. It consists of creating a program that can calculate the average of the numbers provided through an input, then once the sentinel number is provided, stop the execution. Lakanen et al. [17] conducted a study where they presented students with a rainfall type question during the final exam of their CS1 course. Their proposed version of the rainfall exercise required the students to do the following:

- Ignore inputs after the sentinel has been hit.
- Negative values are to be ignored
- The sum of valid inputs have to be calculated
- Needs to calculate the occurrences of valid input numbers
- Needs to protect against zero division
- Needs to calculate the average input value based on the information stored

In addition to the Rainfall exercise, the exam contained 6 other exercises of different types (code tracing, modification, theory). The study looked at how suitable the rainfall code writing exercise was for measuring the students' ability to write code and measure their knowledge of the curriculum, as well as its suitability as an exam question. Through analysis of the exam results, there were mainly two interesting observations. The first observation is related to parameter passing in the rainfall exercise. They found that certain students struggled with assigning values to corresponding parameter values, even though they were perfectly able to understand the concept in other exercises. In contrast, it seemed that other students were perfectly able to comprehend variable usage and scoping in the rainfall problem, but not in other code tracing exercises. The students failing the tracing exercise might have been related to the fact that the course did not provide a lot of conceptual exercises such as program tracing and code explanation.

A proposed approach to help students to understanding code writing as well as evaluating their knowledge is *goals* and *plans* [18]. Goals are ”things” that need to be done in the code to solve the problem. The Soloway rainfall problem can be used as an example. The rainfall problem entails calculating the average of a stream of numbers, indicating the need for two goals, one to sum all of the numbers and one to count the amount of numbers summed. Based on these goals, two *plans* have to be established. A *sum-loop plan* and a *count-loop plan*. Solving the rainfall problem requires these two plans to be merged into one, where the counting and summing happens in one common loop. Explicitly defining and explaining these two concepts can help novice programmers in understanding and solving more complex problems. The study conducted by Costantini et al. [18] tested the approach of grading program writing exercises by looking at the plans made for the program. The study found that plans for more complex exercises and combining different concepts require a deep program comprehension. Despite this, they found that using plans can be a satisfactory approach to evaluating exercises and the students' knowledge. Although the test was only performed on students who passed the course (usually the students with the highest skill level), it seems that using plans to evaluate a students' skill levels is a different, but valid approach.

2.1.2 Single concept questions

It is well known that learning programming is hard. Successfully learning to program requires knowledge and understanding of a large set of concepts and skills. Even though students learn to understand and apply these concepts throughout a CS1 course, they still struggle to see the ”big” picture of a problem and how multiple of these concepts can be used to solve it. As a result, they tend to focus on smaller parts of the problem [19] [20].

An issue related to coding exercises are the concepts required to solve them. Even exercises that are designed to focus on specific concepts tend to require knowledge of more concepts than just the one in focus. If a student lack understanding in any one of the required concepts, their overall performance may tank considerably [14], which is the core of what makes code writing exercises particularly hard. Most exercises require knowledge of multiple concepts, but they also require the knowledge of which concepts to use and where to apply them.

A solution to the issue above is suggested by Zingaro et al. [21]. They propose an exercise type referred to as single concept questions. Single concept questions are code writing exercises, but it tries to tackle the issue with multiple concepts in one exercise by providing multiple questions, each focusing on one single concept instead of one broad question requiring multiple concepts. The study conducted by Zingaro et al. focused on one question requiring the knowledge of multiple concepts, and four smaller questions requiring the same concepts as the first. During the study, these five questions were given as a part of a CS1 exam.

The large, multi-concept exercise explicitly asked the students to iterate over a list of tuples, each containing a name and a list of grades. Their task was to calculate the average grade of each tuple in the list, representing students and their grades. Listing 2.1 shows

the variable *marks*, which is the list of tuples. A function skeleton to help the students get started was also provided, as shown in listing 2.1. This exercise required the students to know the four concepts; accessing a list element, iterating over a list, nested lists, and accumulating from the lists. Each of the four remaining questions, therefore, exposed one of these four concepts.

```
1    marks = [('dan11', [76,80,67]), ('jane23', [81,90,69]),
2    ('jones11', [77,79,55])]
3
4    def calc_average(a_num, marks):
5        '''Return the average mark on assignment a_num for
6        all students in list marks.
7        Precondition: marks contains an assignment
8        corresponding to a_num.'''
```

Listing 2.1: A snippet of the code writing exercise.

The exam scores were divided into four quartiles, where Q1 contained the weakest results and Q4 the best. The whole idea behind single concept questions is to split concepts into separate questions. It was therefore anticipated that the code writing exercise was going to score worse than all concept questions. All of the single concept questions except for nesting ended with better scores than the code writing exercise, indicating that nesting is a particularly hard concept. The quartile data also show that the weakest quartile students were mostly able to solve the list element access and iterate questions, while the other quartiles did pretty well all over, which is also reflected in the code writing exercise.

The results indicate a high correlation between single concept questions and plain code-writing exercises that expose the same concepts. Through the study, it was shown that single concept questions are better and more effective tools for feedback since weaker students are allowed to expose knowledge of concepts that would be missed in multi-concept code-writing exercises.

2.1.3 Program tracing exercises

Program tracing exercises require the student to trace the program through its execution [19]. The exercises usually require the student to provide the output of a function or a printed value, but can also involve explaining in short what the code does or even require the student to determine intermediate values during execution. Additionally, it is possible to create exercises with specific bugs the students have to identify. Program tracing exercises can range from simple exercises with a few lines to exercises consisting of multiple classes, but in general, they test the students' knowledge of concepts such as functions, loops, conditional statements, lists, and other essential concepts. Listing 2.2 illustrates a simple exercise where every number in a is summed in the variable b and printed out. The task here is to determine the final value of the variable b.

```
1    a = [1,2,3,4,5,6]
2    b = 0
```

```
3     for i in range(len(a)):
4         b += a[i]
5     print(b)
```

Listing 2.2: An example of a simple trace exercise.

Lopez et al. found that tracing code accounted for 46% of the variance exhibited in the code writing exercises of an exam [13]. During an exam, they found that students who performed poorly on code tracing exercises also performed poorly on code writing exercises, showing a linear relationship between the two. However, the relationship does not mean that students doing well on code tracing also do well on code writing. Instead, it only confirms the relationship between the two exercise types in cases where students perform poorly.

It has also been found that code tracing exercises require considerably less cognitive load compared to code-writing exercises, as explained in section 2.1.5, providing a great advantage over code writing. Additionally, program tracing exercises expose other advantages:

- Program tracing exercises can focus on specific concepts, making it easier to test that a student has understood that concept in particular.
- Solving the exercises are faster because they do not require any writing and has no compiler or run-time errors. Therefore, the exercises are less time-consuming, allowing for more of these exercises than an equivalent writing exercise.
- Program tracing exercises allow the exercises to test a wide variety of cases within the concept. E.g., for loops and index variables.

An important factor related to successfully solving code tracing exercises is sketching [22] [23]. Sketching is the notion of tracing the code and updating some external model as code is being ”executed”. A line-by-line tracing technique proposed by Xie et al. recommends having a separate memory model on paper that is updated every time a code line modifies a memory value. Both [22] and [23] showed that sketching could considerably improve the program tracing capabilities of students.

2.1.4 Parson’s Problems

The relevant work was reviewed, and the relevant background carried out in the project preceding this thesis [19] were identified. This is amended with a discussion of a few papers that have been studied after the project.

Learning to program can be both a tedious and time-consuming task, and it can often be compared to learning a new language [19]. Writing code properly requires high comprehension of both semantics and syntax, and without understanding a language’s syntactic rules, writing correct code is impossible. A common technique for learning a programming language is drill exercises, where one continuously solve exercises to gain knowledge. The issue with this form of learning, however, is that it is tedious and boring. Being able to engage the learner is, therefore, an important concept when learning students to program [24].

Learning to program is a hard task that, in some ways, can be compared to learning a new spoken language. Both of the cases require mastering of syntax and semantics. It is impossible to write in a programming language without knowledge of its syntactical rules. A technique that can be used to learn these rules are repetitive drill exercises. The problem, however, is that such exercises are boring. Task engagement in learning is, therefore, an important concept when teaching students to program [24].

Parson's problems focus on learning and memorizing syntactic constructs by using drag-and-drop style exercises [25]. Its main principles are:

- **Maximize the engagement:** By making the exercises puzzle-like, they become more engaging for the students, and thus ensure that students get enough repetition of the exercises.
- **Constrain the logic:** Parson's puzzles provide a good code structure, where the available code fragments constrain the logic, preventing the student from becoming sidetracked by not being able to find the correct logical or algorithmic approach to the given problem.
- **Permit common errors:** By providing distractors, the Parson's exercises intentionally give options that resemble errors students often make, allowing them to compare their mistakes with the correct solution.
- **Model good code:** By using fragments of good code, the student is exposed to good code practices.
- **Provide immediate feedback:** Beginner programmers cannot usually debug programs properly, and, therefore, spend considerable time doing so. In Parson's exercises, mistakes and errors can be immediately identified and shown to the student.

Parson's problems are providing students with a set of different text snippets (usually resembling one or more code lines) that are often referred to as code fragments. These fragments have to be placed correctly relative to each other to create a correct solution. It is possible to check the correctness of the current answer at any given time, and it is also encouraged to do so until the correct solution is found. Figure 2.1 shows a Parson's problem where students need to drag fragments from left to right and place them in the correct order and with correct indents.

Ericson et al. [26] performed an in-depth analysis of solving Parson's problems versus fixing and writing code. They performed a study where students were provided with one of three random exercises; a Parson's problem with distractors, fixing code with the errors equivalent to the distractors of the Parson's problem, or writing the equivalent code. The study was conducted in two different sessions. The first session consisted of solving one of the three exercises described, while the second session consisted of an exercise isomorphic to the first. Through the study, they found that solving the Parson's problem was significantly faster than fixing errors or writing the code from scratch, while still giving significant improvements from the first to the second session. The results show that Parson's problems could be classified as equally effective or even more effective in learning than correcting errors or writing code. Parson's problems can, therefore, be a handy tool

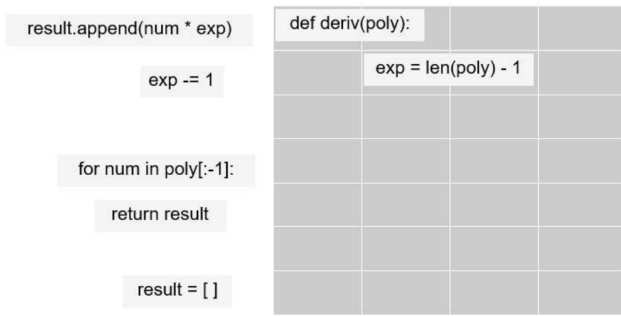


Figure 2.1: Illustration of a Parson's problem exercise where the student has to drag code fragments from left to right in the correct order and with correct indents. The illustration is reprinted from [1].

for learning the basics of syntax, semantics, and algorithms.

Paul et al. found that Parson's problems have a high correlation to code-writing [27]. They also come with the advantage of being easier to correct, since the amount of possible solution is restricted by the permutation of correct code fragments and distractors. Even though a correlation between Parson's problems and code-writing was found, it does not directly imply that students' being able to solve Parson's problems also can solve similar code-writing problems. The same study [27] conducted an experiment where the students were asked to solve a Parson's problem. Once solved, they were asked to solve the same problem with code. A lot of the students were unable to do so, indicating that students can solve Parson's problems without fully comprehending the whole meaning of the underlying code.

Initially, Parson's problems were designed to be combined with distractors. Distractors are code fragments that are not a part of the solution code, and therefore "distracts" the student. The idea behind the distractors is to make them resemble correct code fragments to make them harder to distinguish from the correct fragments. Harms et al. researched the topic of distractors in Parson's problems and found that distractors could prove itself detrimental to learning [28]. Because distractors come with an increased cognitive load, it reduces the students' ability to solve the problem and increases the time spent. As a result, it might harm the learning, rather than the opposite.

2.1.5 Program Completion vs. Program Generation

Merriënboer et al. conducted a study focused on investigating the differences in learning outcomes between observing, modifying, and extending well-designed programs (completion) versus creating new programs from scratch (generation) [29].

Developing a broad knowledge base requires variation. A wide variety of exercises can provide such variation:

1. Running and tracing programs, designing and coding algorithms, modifying, extending and debugging programs

2. Presentation of a large set of problems with different underlying solutions, and programs that are correct solution to different programming problems

In terms of variety, code completion exercises expose the variation above. Another significant advantage of code completion is that the students are presented with a partial solution, providing exposure to a wide variety of problems and, therefore, well-designed solutions for these problems [29]. Variation is not as quickly exposed in code generation exercises. Since the time spent both designing and implementing complete programs often takes much time, variety is harder to achieve.

In addition to exposing variety, program completion exercises have another critical advantage over code generation; it has a natural usage of examples. The reasoning behind the importance of this is directly connected to how students transition from passive, declarative knowledge to program state behavior. In more concrete terms, it means that students are rather picky when selecting material to learn from, and they usually favor examples directly related to the problem at hand.

During the study, Merriënboer et al. conducted an experiment where they tested a set of generation and completion exercises on two different groups of high-school students, then tested their acquired knowledge. It was found that both groups wrote approximately the same amount of code lines during the experiments, but the completion group had, on average more correct lines. Additionally, it was found that the completion group also had produced higher-quality programs. Further, they found that during the course, the generation group felt the difficulty increased, while the completion group found all exercises to be of the same difficulty. Further research found that the generation exercises increased in difficulty, resulting in processing overload. The completion group did not encounter this issue and had an even processing load throughout the experiment. This is attributed to its variation in activities such as tracing, modification, completion, and reading. Code generation can, therefore, be deemed inferior to code completion when compared in terms of processing load.

2.2 Concept inventories and exercise meta-data

Concept-inventories (CI) are assessments that are specifically designed to measure the learning of core concepts [30]. Concept inventories are common and well used in other sciences such as Physics. The main idea behind concept inventories is to develop ways to measure student knowledge across core concepts of science. As a result, CI's provide educators with ways to measure and compare the learning outcomes of students across instructors, institutions, and curriculum. This allows for educators to improve the efficiency and other teaching-related factors based on the feedback from the CI.

Although concept inventories are well established in other sciences, CS is still a young science currently in development. As a result, concept inventories are still lacking. Current issues related to CI's for computer science are the constant changes in the field. Since new languages are developed continuously, the languages used in CS courses change over time. As a result, CI's specific for a language might, therefore, not be valid over a more extended period. A possible solution to the problem is language independent CI's, but this

does not solve the problem over time. Since different courses use different programming paradigms, such as Object-oriented or functional programming, which use very different memory models, a single pseudo-language cannot capture the context for these different approaches.

A possible way of developing new CI's for computer science could be through analyzing results from multiple-choice questions containing meta-data such as topics, difficulty, and language [30]. These questions are rather easy to administer, allowing researchers to compare the results across different contexts, such as instructors and courses. By utilizing accompanied meta-data, researchers could utilize the information available to further develop CI's for CS.

2.3 Symbol table

A symbol table is a way of storing variables and their types, as well as functions available at any given program scope [31]. Symbol tables are often used to aid the traversal of the parsing tree, as seen in Figure 3.2. There are multiple ways of implementing a symbol table, but it is commonly done by defining unique tables for every program's scope and linking them together. With this approach, a base symbol table, at the highest level, is instantiated. Once a new scope is found, a new table is created, pointing to the previous table. Thus, all available variables in any given scope is contained either in the symbol table from the current scope, or in the symbol table in any of the parent scopes. The creation of a symbol table can be illustrated with an example using the code presented in Listing 2.3. In the code, there is a total of three scopes. The initial scope, containing the variables a and b, and two deeper scopes with separate variable assignments. Figure 2.2 illustrates the symbol table created from the code in Listing 2.3. An example of using this symbol table would be to check for available variables in the ***else:*** scope. Here, the current symbol table is the child table to the right in Figure 2.2. By first extracting the variables in that symbol table, then looking at the variables in the parent, it is possible to determine that the variables, a = 5, b = 6, c = 9 and d = 10 is available in the ***else:*** scope.

```
a = 5
b = 6
if a > b:
    a = 6
    c = 7
    d = 8
else:
    c = 9
    d = 10
```

Listing 2.3: A simple template for generating random ways of iterating a for loop

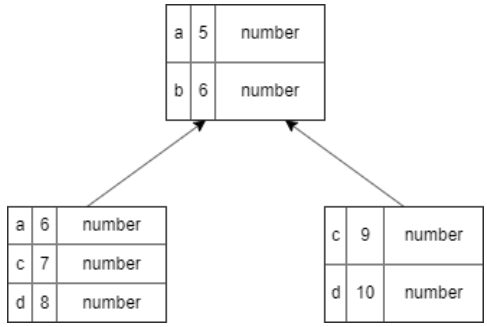


Figure 2.2: The symbol table generated from the code snippet in Listing 2.3.

2.4 Comparison of exercise types

Throughout the section, a lot of different exercises have been presented and described. The most obvious choice for testing a students' code writing skills are code-writing exercises. As mentioned in section 2.1.1, there are several drawbacks linked to code-writing exercises. Additionally, generating satisfactory code-writing exercises is a rather hard problem. On the other hand, single concept questions aim to mitigate some of the problems linked to code-writing exercises. The main idea is to split the concepts within a code-writing exercise into smaller sub-exercises. This approach has been proven to help students' show their knowledge, but as with code writing exercises, they can be hard to generate since their solution generation is rather complicated.

On the other hand, program tracing exercises are not as good for testing the students' knowledge as the formerly presented exercises, but they require less of a cognitive load and still gives a decent indication of the students' code-writing skills. Section 2.1.3 also present some of the other advantages of using program tracing exercises. Compared to code-writing, a big advantage of program tracing is its ability to test many different aspects of a concept. Another great advantage of the program tracing exercises is their solution generation. Since program tracing relies on the student to determine the output of some code, running said code would also generate the solution. Although program tracing has many advantages over code-writing exercises, they are not automatically better. Tracing exercises allow for quick and easy testing of concepts, providing reasonable grounds for expanding into code-writing exercises with the same concepts. Therefore, the two exercise types can be looked upon as complementary, as the ultimate goal is to teach students to write code. Finally, program-tracing exercises are not just relevant for beginners. Working with IT usually require the candidate to understand code written by others, requiring the ability to trace.

Parson's Problems are also presented, and can, to a certain extent, be looked at as an extension of program tracing. The big difference is that they usually require a better description of program tracing exercises and require the code to be split up into fragments. Thus, Parson's problems would be a natural expansion from program tracing exercises.

2.5 Auto-generating program tracing exercises

In a previous project, related to the pre-project, a prototype for a generator was implemented and evaluated. The project was finished autumn 2019 as a part of a 15 points course. Its choice of exercise type to generate was based on the same reasoning presented in section 2.4. Section 3.4 also discusses the different models, and gives the reasoning behind the best model for this project, which is the same reasoning the pre-project is based on.

Based on the presented reasoning, the pre-project implemented a prototype capable of generating simple program tracing exercises using a context-free grammar (CFG). The goal of the project was to prove that the CFG approach is a good way of generating program tracing exercises, and the prototype was evaluated and improved through design iterations.

The first focus of the project was to implement a bare-bones solution capable of generating program tracing exercises with some form of randomization. To achieve this, ANTLR4 was chosen as a framework for implementing the CFG. The initial CFG implementation contained simple logic for variable assignment, if statements, for statements, expressions, and randomized functions, which are briefly explained below.

2.5.1 Initial grammar

As explained, the first solution contained grammar for variable assignment, if statements, for statements, expressions, and randomized functions. Below is a brief explanation of the different elements and their implementation in the prototype.

2.5.1.1 Variable assignment

Variable assignment is a crucial part of almost any programming language, including Python, and is therefore required to generate proper exercises. An assignment is divided into two types: assign variable and assign list. The former represents an assignment of any variable. The latter explicitly targets list assignments, which requires a more careful definition to allow more than one element to be defined in a list. In Appendix I the Productions 7.9, 7.10, and 7.11 show the three productions required to define a list. The last Production defines the restrictions for a variable name, which states that variables can have names containing numbers from 0 to 9 and a to z, but is not allowed to start with a number.

2.5.1.2 If-statements

If-statements are just as crucial as variable assignment, and therefore had to be implemented in the most simple prototype. The if-statements implemented was designed to allow any number of if, else if and else clauses to simulate how if-statements in Python is implemented. Appendix I shows the implementation of if-statements in the Productions 7.12 to 7.14.

2.5.1.3 For-statements

As with if-statements, for-statements are an important tool, especially when working with lists. The initial approach only supported one type of for statement, namely **for *var_name* in range(0, n)**. While there are several other ways to iterate over a set of values or a list, for in range is the most common, and was therefore chosen as the initial way to express a simple loop.

2.5.1.4 Expressions

Expressions are the bread and butter of almost any grammar, as productions usually end in one or more expressions. The grammar defines arithmetic, relational and atomic expressions. Arithmetic expressions are mathematical expressions, usually used for numbers and certain String operations. Relational expressions are essential to define conditions for if-statements, and atomic expressions are primarily used to define atomic variables of different types. The grammar used in the prototype explicitly defines numbers, strings, and booleans as atomic variable types.

2.5.1.5 Custom functions

The final part of the initial grammar was custom functions. These functions are specific for this grammar and define parts of the grammar that is to be randomized. The initial implementation allowed for randomization of arithmetic and relational operators, random int values, and random int arrays. Randomization of these custom functions is implemented during the traversal of the parse tree, by substituting the functions with real values. Figure 2.3 shows a simple parsing tree with the custom randInt function that is substituted for a random value within its parameters. As explained, this process is done for every custom function implementation every time an exercise is generated.

2.5.2 Further development

The grammar explained in the sections above was the bare-bones implementation of the prototype. During the project, the prototype was further developed to generate better and more interesting exercises. As mentioned it was evolved through an iterative process. Based on the evaluation and feedback of the exercises generated by the initial grammar, a second iteration was used, where two new key features were implemented into the prototype. These two features were randomized iteration and if relational statements generated with available variables at the given scope.

2.5.2.1 Randomized iteration

The first feature, implemented as a result of the first evaluation was randomized iteration. The idea behind this feature is to provide a language construct that allows the writer of the template to define that a loop should be used, but not explicitly define what kind of loop. There exists multiple ways of iterating over a list, and the three most common is the for-in-range, while and for-each approaches. A vital hindrance of a unified definition of a for loop is how to access the list element currently iterated. Since for-each explicitly defines the

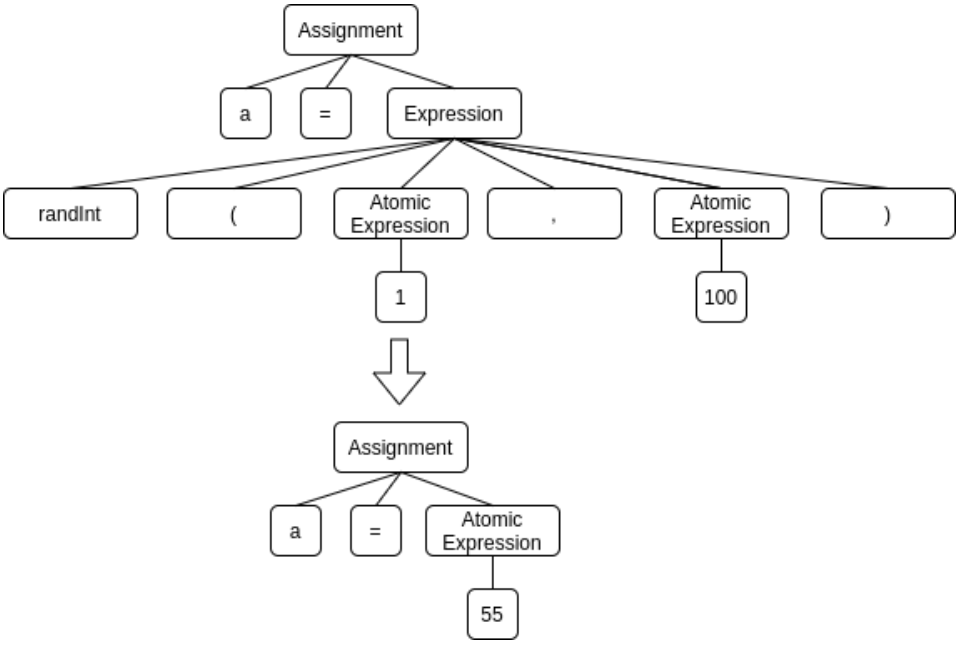


Figure 2.3: The parse three before and after the randInt visitor has modified it with a random value.

current element accessed in a list, while for-in-range and while loops usually define some index to be iterated over, these approaches also handle accessing elements differently. To handle this issue, a pre-determined value, *it*, was defined. This value is used as a general way of ”pointing” at the current element used. Upon exercise generation, all *it* references inside of a randomly generated loop are substituted with the correct way of accessing the list element. Listing 2.4 illustrates a simple example of how the template is translated into an exercise when it contains a randomized iteration.

```
# The template used for generating
a = [1,2,3,4,5]
sum = 0
for (a) {
    sum = sum + it
}
print(sum)

# Exercise 1 generated
a = [1, 2, 3, 4, 5]
sum = 0
i = 2
while i < len(a):
    sum = sum + a[i]
    i = i + 1
print(sum)

```

```
# Exercise 2 generated
a = [1, 2, 3, 4, 5]
sum = 0
for e in a[0:len(a) - 2]:
    sum = sum + e

print(sum)
```

Listing 2.4: A simple template and two generated exercises containing the random for loop construct.

2.5.2.2 Composed relational expressions

The second feature implemented was the ability to generate relational expressions consisting of variables available at the given scope, and usually also contain the type of the variables. A symbol table was implemented to keep track of variables available at any given scope. The particular symbol table implemented is as described in section 2.3. The first step of actually generating a composed expression is to extract the variables to be used. By traversing the available symbol tables, available variables of the correct type can be selected. Only variables that are defined as numbers are selected by the prototype to simplify the generation process. Once the available variables have been extracted, they are combined iteratively by placing relational operators between them. Listing 2.5 show an example of a simple template with a composed relational expression and two of the exercises generated from it.

```
# Template used for generating
a = 5
b = 6
c = 7
d = 8
if composedStatement() {
    print(a + b)
} else {
    print(c + d)
}

# Exercise 1 generated
a = 5
b = 6
c = 7
d = 8
if c < d or a > b:
    print(a + b)
else:
    print(c + d)

# Exercise 2 generated
a = 5
b = 6
c = 7
d = 8
if d <= b and c < a:
```

```
    print(a + b)
else:
    print(c + d)
```

Listing 2.5: A simple template and two generated exercises containing one composed relational expression.

2.5.3 Application used for evaluation

In addition to the prototype, a simple application was set up to make the evaluation of the prototype easier. The application consists of a simple graphical interface implemented in React [32] and a small web-server containing the prototype as well as a small REST-API used for communication between the front-end interface and the prototype. The REST-API is written in Java with the Spring-Boot [33] framework. Figure 2.4 shows the architecture of the application and how the different components communicate with each other. This application was used during interviews with professors while evaluating the prototype. It simplified the process of showing the capabilities of the prototype by allowing them to see templates and exercises generated from them in real-time, with proper support for checking whether the answer to the exercise is correct or not. Images of the web application can be found in Appendix J.

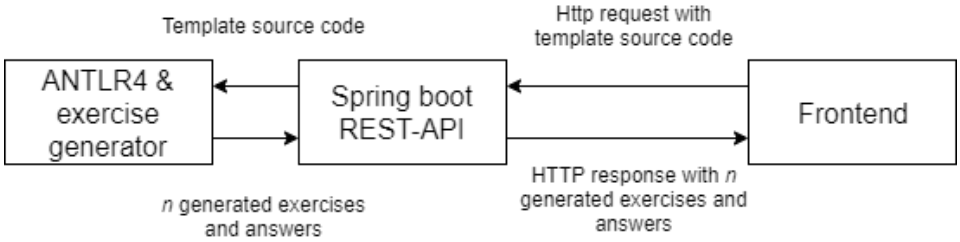


Figure 2.4: A simple illustration of the prototype architecture and the communication between the components.

Chapter 3

Related work

3.1 Good program tracing exercises

In section 2.1.3, tracing was presented and compared to program writing exercises. The most common way of evaluating tracing exercises is compared to program writing exercises, where the students' scores on program tracing exercises are directly related to their scores on program writing exercises focusing on the same concepts.

Venables et al. conducted a study where they looked at the outcomes of simple and complex program tracing exercises and code-writing exercises [15]. Their definition of simple program tracing exercises, were exercises with only one loop, while the complex had more than one. Through the study, they found a causal relationship between the simple program tracing and code writing exercises. Their data indicated a relationship between the two that indicates the existence of a minimum tracing skill requirement to enable code writing, but the minimal program tracing skill does not suffice by itself to enable code writing. It becomes clear that too simple program tracing exercises might not capture the minimum required skill to write code. The study also made a comparison between complex and simple exercises. They found that the complex tracing exercises required a systematic approach to the tracing, e.g., sketching [22] [23]. This, in return, indicates that students require a deeper understanding of the whole execution process when answering more complex tracing exercises. Through comparison with code writing exercises, they found clear evidence that complex tracing would relate better to performance on code writing than simple tracing exercises. Such exercises might be better suited for exercises where the students' skills are being evaluated. On the other hand, the relationship between the complex tracing exercises and the combined code-writing exercises does not expose a direct linear relationship. A possible reason for this is that complex program tracing exercises are very error-prone, and a small mistake might, therefore, cause the answer to be completely wrong, even though the student understood all of the critical concepts in the exercise.

While solving complex tracing exercises might make the student more capable of writing code, it does not necessarily imply that simple program tracing exercises does not bring anything to the table. Simpler exercises might not provide proper evaluation of a student; however, they might still help improve the students' knowledge of simple concepts in programming, and help them understand how to apply them. Therefore, it is essential to distinguish between different use-cases when evaluating the usefulness of the tracing exercises at hand.

3.2 Specification, Configuration and Templates (SCT)

A review of the current related work and state of the art was performed, and relevant background material from the preceding project was identified [19]. Through this process, no new material was found. The presentation from the project report is included below.

Radovsevic et al. proposed a generator framework called Specification, Configuration, and Templates (SCT) that allows for the generation of specified and personalized exercises [2] [19], by providing a model represented through a set of artifacts, the framework allows for program synthesis. The artifacts consist of application parameters and templates which can be synthesized according to the generator's configuration. Additionally, the SCT allows for a multi-level design, where higher-level generators can be defined through one or more lower-level generator. This way, generators can be re-used, striking a similarity to Object-Oriented Programming.

3.2.1 The SCT Model

The three core elements of the SCT model are: Specification, Configuration, and Templates. An SCT frame represents the three elements, which are illustrated in figure 3.1. The three elements can be used in conjunction to generate unique and customized exercises. Below is a short description of each of the elements:

- **Specification** The specification contains features of the generated application in the form of attribute-value pairs.
- **Template** contains the source code in the desired programming language, together with a set of *connections.*
- **Configurations** defines the *connection* rules between the specification and the templates.

3.2.2 Generating source code with SCT

To generate source code, the three elements: Specification, configuration, and template, have to be defined. Each of the elements play an essential role in the generation process.

A **Specification** defines concrete values that are used in the connections. Below is an example of a specification:

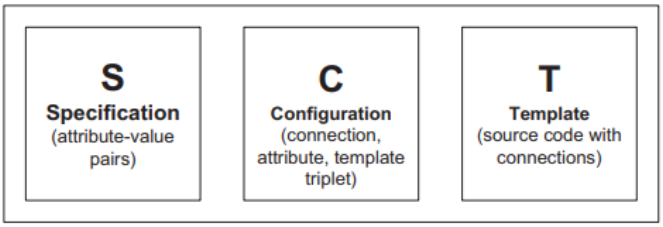


Figure 3.1: Illustration of the three elements that together makes a SCT frame. Reprinted from [2].

```
1  OUTPUT:out1
2  out1:output/Linked_list.cpp
3  field_int:student_id
4  field_char:surname_name
5  field_int:year_of_study
6  field_char:note
```

Listing 3.1: An example of how a specification is defined.

The specification defines an output file, Linked_list.cpp, which has four attribute pairs defining their type and the attribute name.

Configurations define how the connections defined in the templates are to be replaced. The listing 3.2 shows an example of a configuration file:

```
1  (1) #1#,,main.template
2  (2) #fields_declarations#,field_*,field_*.template
3  (3) #data_entry#,field_*,data_entry.template
4  (4) #field#,field_*
```

Listing 3.2: An example of how a configuration is defined.

The first rule defines the starting template to be used. The second rule replaces the occurrence of #field_declarations# in the template file with all the fields and their templates defined in the specification. Similarly, rule 3 replaces the #data_entry# connection with every definition starting with field_ in the specification. The fourth rule is used to replace the #field# connection in the data_entry template.

Templates are program fragments where connections are explicitly defined. Listing 3.3 shows a template:

```
1  cout << "#field#: ";
2  cin >> new_element->#field#;
```

Listing 3.3: A simple template

The SCT model defined above goes through every variable defined in the specification and prints its name and value.

3.2.3 Using SCT to generate personalized student exercises

The SCT generator framework was initially designed to generate specified exercises tailored for students. However, it can also be used to generate randomized exercises as proposed by Radovsevic et al. [7]. To generate randomized exercises, the specification part of the generator has to contain constant attributes, but randomized values. When the attributes remain constant, the configuration and the template elements can be kept constant for every instance of the specification, meaning it is the only element that has to be changed for randomization.

3.3 Syntactic Generation of Programs using Context-free grammar (CFG)

State of the art and related work were reviewed, and an identification of the relevant background material was carried out in the project preceding this thesis [19]. This is amended with a discussion of a few papers that have been studied after the project.

An approach to generating a large number of exercises based on a context-free grammar (CFG) was presented by Ade-Ibijola [3]. The solution utilizes a CFG, which is mainly used for recognizing or generating languages. The grammar used during the study was designed to emulate Python to a high degree, and thus making it easier to convert the grammar to complete and working Python code.

The study focused on generating *program tracing* exercises. As a result, all exercises generated requires at least one print statement that the student is supposed to determine the output of. The main focus of the generator was on three different types of concepts: Loops, conditional statements, and arithmetic operations. The generator also allowed for the usage of built-in math functions to allow for more complex and interesting exercises.

3.3.1 CFG

A context-free grammar (CFG) is used to specify the syntax of a language. A grammar describes the hierarchical structure of a programming language. Formally, a CFG is often expressed as a four-tuple: G = (N, Σ, P, S) [31], where:

- N is a set of *non-terminals*, usually called ”syntactic variables”. Each non-terminal represents a set of strings of terminals.
- Σ is a set of *terminal* symbols, often referred to as ”tokens”. The terminals are elementary symbols of the language defined by the grammar.
- P is a set of *productions*, where each production consists of a non-terminal, called the left side of the production, an arrow, and a sequence of terminals and/or non-terminals, often called the right side of the production.
- S is a designation of one of the non-terminals used as the *start* symbol.

3.3.2 Lexical analysis and Syntax analysis

Two essential steps in capturing the logic of a language for a CFG are lexical- and syntax analysis. These are two of three steps in the front-end of a compiler. The front-end of the compiler is concerned with taking a source program and converting it to some form of intermediate representation. An example of this is a compiler converting C code into assembly instructions, where C is the source code, and the assembly instructions are the intermediate representation.

Lexical analysis is the notion of splitting the text into tokens, the terminal symbols described by the CFG. With the help from the tokens generated, the syntax analyzer, often referred to as a parser, can create a parse tree and validate the tokens based on the CFG. A parse tree is a logical representation of the flow of the language. Since all productions in the CFG are deterministic, the syntax analysis will always have a set of expected tokens. If an unexpected token is provided to the analyzer, it means the language provided is not correct in regards to the CFG. Figure 3.2 shows the three steps in the front-end of a compiler. In this case, the end output, usually known as the intermediate representation, is the generated python code.

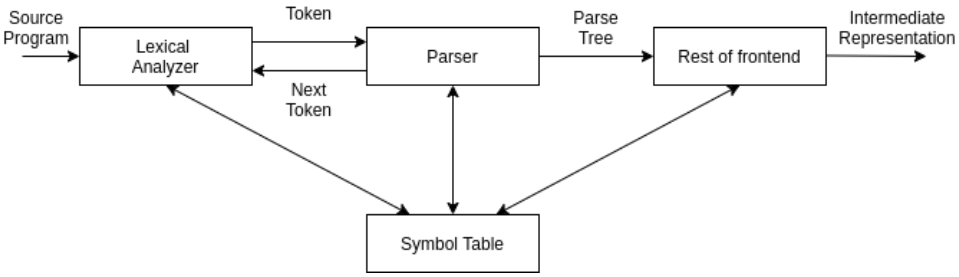


Figure 3.2: The three steps in the front-end of a compiler. The symbol table is used to store symbols such as variable names and other values that might be required in any of the three steps.

3.3.3 CFG as a template

The CFG proposed by Ade-Ibijola describes a subset of the language rules for identifiers, control rules, and structures of the programming language Python. Production 3.1 to 3.5 below are a few of the productions he proposes for terminal symbols. They describe terminal productions for letters, digits, and different operators used.

$$\texttt{<letter>} \longrightarrow l \in \Sigma \tag{3.1}$$

$$\texttt{<digit>} \longrightarrow d \in 0 \mid ... \mid 9 \tag{3.2}$$

$$\texttt{<rel\_op>} \longrightarrow \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{!=} \mid \texttt{==} \tag{3.3}$$

$$\texttt{<arth\_op>} \longrightarrow \texttt{+} \mid \texttt{-} \mid * \mid \texttt{/} \mid \% \tag{3.4}$$

$$\texttt{<logi\_op\_infix>} \longrightarrow \texttt{and} \mid \texttt{or} \tag{3.5}$$

As mentioned, every CFG has a symbol S, denoting a non-terminal start symbol. This is required to construct the root of the parsing tree used for syntax analysis. The proposed solution defines the non-terminal <prog> as shown in production 3.10 to 3.12 to be the start symbol. Productions 3.6 to 3.8 show the productions for the three different program types.

<prog_arth_expr_eval> ⟶ <ident_init><assignments><display> (3.6)

<prog_cond_expr_eval> ⟶ <ident_init><if_stmt><tab_in><display> | (3.7)

⟶ <ident_init><if_stmt><tab_in><display>

((<elif-stmt><tab_in><display>) | λ) (3.8)

<else_stmt><tab_in><display>

<prog_loop_expr_eval> ⟶ <ident_init><for_loop> | <while_loop> (3.9)

<prog> ⟶ <prog_arth_expr_eval> | (3.10)

⟶ <prog_cond_expr_eval> | (3.11)

⟶ <prog_loop_expr_eval> (3.12)

3.3.4 Generating exercises from a CFG

To generate a set of distinct exercises from a parsed program, traversing the parse-tree and looking at every leaf node is required. Every leaf node that solely consists of some syntactic variable can be printed out, as the CFG is designed to emulate the syntax of Python. Leaf nodes that contain relational operators, arithmetic operators, and numerical values can be substituted for other randomized values. Multiple randomized and different exercises can be generated by doing this for multiple iterations of the parse tree.

Additionally, special syntax was defined for generating a random number of variables. Because the parse tree is linear to the program flow, it is easy to keep track of the available variables at any given state in the program. This makes it possible to generate mathematical expressions between all available numerical variables to create even more randomized output.

3.3.5 Solution generation

Generating solutions for the programming exercises made this way is a rather trivial task. Since all the programs are related to program tracing, the only output made from the programs are printed to the console. Generating the solution of any given program require each generated Python program has to be run individually, then storing their output. Figure 3.3 shows the simple process of generating solutions for the generated programs.

3.4 Comparison of models for generating exercises

A few fundamental features can differentiate the two models for generating exercises described above. The syntactic generator is built by defining a grammar very similar to

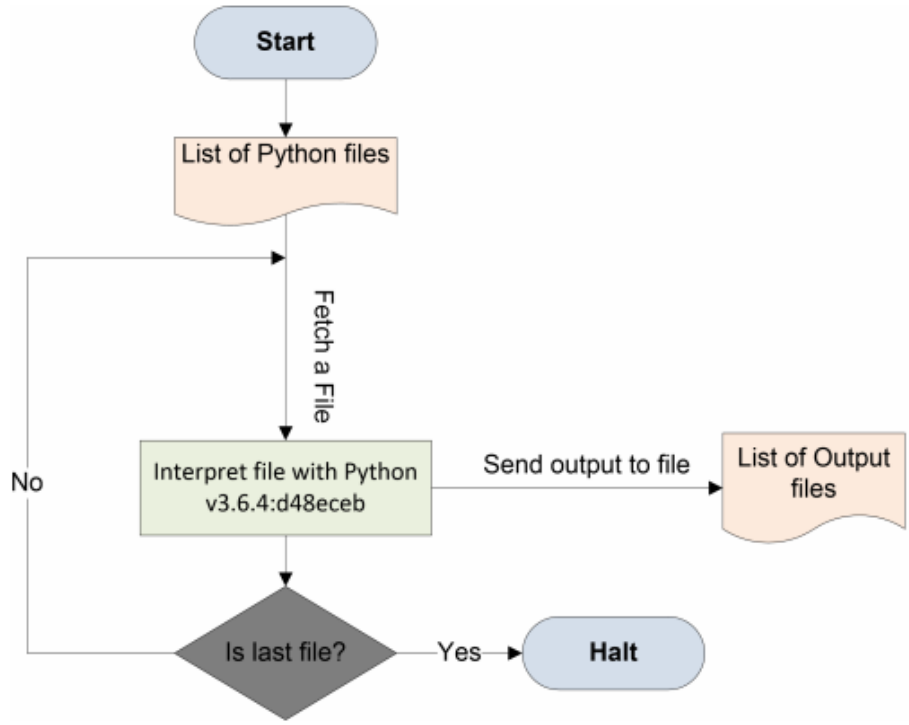


Figure 3.3: Solutions are generated by iterating over all of the newly generated Python files and saving their outputs to appropriate files. Reprinted from [3].

Python, then parsing said template and analyzing it to generate distinct exercises in the specified output language. On the other hand, the SCT approach is based on templates in any given language. It works by inserting connections into the templates to randomize certain elements. Their use-cases may differ due to their differences; therefore, it makes sense to compare them when generating program tracing exercises for Python.

One of the SCT models' most significant strengths is that it is not confined to a single target language, like the syntactical model. It is language-independent, and therefore does not require a template to be written for a specific language. Thus, using such a generator would allow exercises to be generated for many courses, utilizing different programming languages. This flexibility can prove itself extremely valuable if the generator model is to be used for multiple courses. The syntactic generator approach has a different focus: generating exercises for a single output language, rather than any language. It is, however, possible to generate exercises for more than one language. This requires considerable work, since the whole tree traversal and generating step would have to be written for the given output language. Hence, the syntactic generator approach is most suitable for single target language use-cases.

Although the syntactic generator is used for generating exercises for a single language,

it allows for far more complex analysis and generation than the SCT model. The SCT model is based on simple substitutions through connections, while the syntactic generator allows for more in-depth analysis and, therefore, more complex substitutions. Since the syntactic generator utilizes parse trees, it can construct Symbol tables, allowing it to generate randomized comparisons based on the available variables at any given state of the program. Ade-Ibijola [3] showed that the syntactical generator is capable of generating if statements containing different permutations of variables available at any given scope. Hence, the syntactic approach is superior in cases where a wider variety of more randomized exercises for a single language is preferred over simpler exercises for a large number of languages.

Based on the comparison above, it is clear that the syntactic approach would be a better fit for generating program tracing exercises for CS1 courses where Python is the only language used.

3.5 ANTLR4

ANTLR4 (ANother Tool For Language Recognition) is a robust parser generator framework. The framework takes in a grammar that specifies a language and generates a recognizer for the specified language. The input grammar is defined as a Context-free grammar. The framework supports generating code in Java, C#, C++, JavaScript, Python, Swift, and Go.

Chapter 4

Methodology

4.1 Design Science and Behavioral Science

The two main paradigms that characterize the research in the information systems genre is behavioral and design science [4]. While both of the paradigms are important to the genre, their focus' differ;

- **Behavioral science**: Behavioral science usually focus on an IT-artifact implemented in a given context. Usually, theories are created, seeking to explain or predict human or organizational behavior. The paradigm address research through developing and justifying an IT-artifact.
- **Design science**: Design science seeks to extend the boundaries of human and organizational capabilities by creating new and innovative artifacts. It is concerned with solving problems by creating innovative and effective solutions to solve problems.

From the characterizations of the two paradigms, it is clear that behavioral science is concerned with "why" an IT-artifact works, while design science focuses more on whether it works or not. Even though they differ, they are also highly co-dependent. Since design science focuses on the creation of new and innovative IT-artifacts, it also relates to behavioral science. Creating useful and innovative IT-artifacts requires existing theories to be applied, tested, modified, and extended through experience, creativity, intuition, and problem-solving [4]. It is, therefore, clear that both of these paradigms depend on each other to drive research forward.

4.2 Design Science: A Framework for IS Research

Design Science has a great focus on both the process and the product to be made. There are two processes and four design artifacts produced by design science research in information

systems [34]. The two processes described are *build* and *evaluate*. These processes define an iterative workflow where the artifact is built then evaluated. The evaluation step is used to further build and modify the artifact. The four design artifacts described are:

- **Constructs**: Defines the language in which problems and solutions are defined and communicated.
- **Models**: Uses constructs to represent a real-world situation, the design problem, and its solution space.
- **Methods**: Defines the processes. Provides guidance on how to solve problems; e.g., search the solution space.
- **Instantiations**: Shows how constructs, models, and methods can be used together in a working system.

Figure 4.1 shows the framework proposed by Hevner et al. [4]. The environment section is the problem space. It consists of people, organizations, and their existing or planned technologies. In combination, they define the problem. As mentioned, the IS research stage consists of two processes; developing/building and justify/evaluating. From the justification/evaluation phase, weaknesses can be identified in the theories/artifacts, allowing the researchers to refine and reassess. The knowledge base is the last section, and it consists of prior research in the area and methodologies for evaluation.

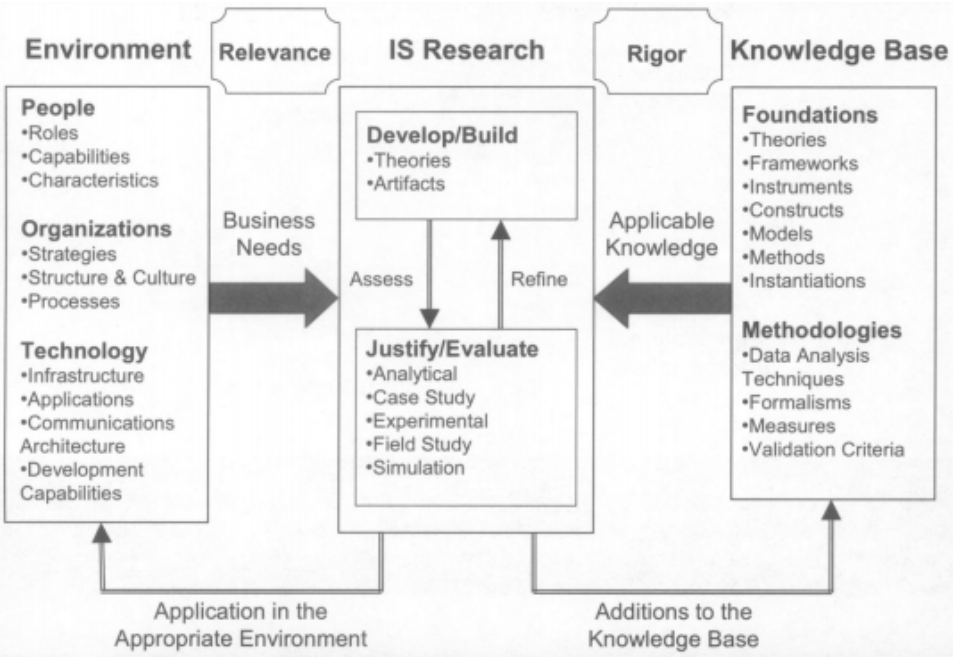


Figure 4.1: The framework for information system research. Reprinted from [4].

In addition to the framework, Hevner et al. proposes seven guidelines for design sci-

ence:

1. **Design as an Artifact**: The result of the design-science research is to create an artifact that solves an important organizational problem. It should also provide enough description to enable efficient implementation in the problem domain.
2. **Problem Relevance**: The objective of IS research is to acquire knowledge and understanding to enable the development and implementation of technology-based solutions to unresolved problems. Hence, the problem that is being researched must be significant enough to warrant an efficient technical solution.
3. **Design Evaluation**: Evaluation is a crucial step in the research process. Proper evaluation is fundamental to the success of the research, and without it, the building phase suffers.
4. **Research Contributions**: Effective and good research must provide clear contributions to the problem. The final assessment for any research is "What are the new and interesting contributions?".
5. **Research Rigor**: Design science relies on the application of rigorous methods in both construction and evaluation.
6. **Design as a Search Process**: To create an effective artifact, the researcher must utilize available means to reach the desired result, while also satisfying the laws of the problem environment.
7. **Communication of Research**: The design research results must be properly presented and conveyed to both technology-oriented and management-oriented audiences.

These guidelines assist in understanding the requirements for effective design-science research.

4.3 Application of Design Science

Both Behavioral science and Design Science play integral parts to the information science research genre. While behavioral science mostly focuses on *why* something works, design science is mainly interested in *if* something works. This report is mostly concerned with the finding out *if* there is a good approach to generating a large amount of program tracing exercises. In the light of this problem, design science provides the best tools. Therefore, the focus of this report is to answer the research questions provided in the context of design science.

As stated in the introduction, the first research question proposed is: **What solutions of auto-generating programming-exercises have already been proposed in research-literature?** This question mainly supports the second and third research question, and it is deeply rooted in the second design guideline, namely, problem relevance.

The second research question, however, **What is the best approach for generating a large amount of program tracing exercises?** warrants proper usage of the design science

principles. This question is also affected by the methods used for testing and evaluating the prototype.

The third question: **Which parts of CS1 courses would the generated exercises be most useful for?** is related to the usefulness of the generator in CS1 courses. Since this report mainly focuses on the technical aspects of the exercise generation, the research question is not within the main focus. The question, however, is important, because it allows the prototype to be discussed in the light of the problem that has to be solved, which is rooted in the **problem relevance**.

The fourth question is related to the limitations of the generation approach: **Are there any significant technical limitations related to the approach chosen for generating exercises?** This question requires exploration of the different aspects of the implementation. Since design science is an inherently iteration-based approach, different aspects of the implementation will naturally be explored, and thus, limitations will also be discovered during development.

Table 4.1 explains the relevance of the seven guidelines to the project, as well as how they have been taken into account.

4.4 Evaluation technique

As mentioned in Table 4.1, the evaluation form for the project chosen was semi-structured interviews with professors in the CS1 courses at NTNU. There are many ways of evaluating the generator and the exercises generated, such as having user tests where students test the exercises. The choice, however, fell on semi-structured interviews with professors between every iteration based on a few critical reasons. Since this project mainly focuses on whether the technique for generating exercises is suitable for the CS1 courses at NTNU, the professors in charge of the courses, who have many years of experience, sit on valuable information related to the quality of the exercises and the generator. A semi-structured interview then allows for a more in-depth discussion of features related to the generator, resulting in a good way of evaluating the usefulness of the exercises generated without directly setting up a test with students.

Another essential reason behind this choice was the time constraints of the project. Setting up a test with students is time-consuming and highly relies on the generator being able to generate sensible and correct exercises at the time of the test. Hence, such a test would only be sensible to perform after the final iteration and also requires access to a large number of CS1 course students. Discussing with professors is less time-consuming and allows for evaluation of every design iteration.

4.4.1 Testing

Instead of directly interviewing professors with examples of generated code, the last iteration was evaluated through a prototype test. The professor was given access to the prototype, along with documentation of how it works. They were given free reigns to test different templates as they pleased, but were also provided with a few examples to get

some inspiration. This freedom was given because, in addition to being testers, they are also experts in the field. Thus, they had a good idea of what features they would want from an auto generator.

Since the purpose of the test was to test the auto generator's ability and usefulness, the professor was given access to the prototype without any supervision and were given a form with a set of questions to answer after they had done the testing.

4.5 Functional requirements

Another important aspect of the methods used to develop the artifact is requirements. The term functional requirements are fundamental concerning this aspect. A functional requirement can mainly be defined in two different ways [36]:

- The first aspect is that a functional requirement is a requirement stating a function the system should be able to perform.
- The second aspect defines the behavioral aspects of the system. Thus, a functional requirement can be defined as "*those requirements that specify the inputs (stimuli) to the system, the outputs (responses) from the system, and behavioral relationships between them*".

The functional requirements are essential to the project as they help define the needs of the system. Since the project is based on design science, iterations are an essential part of the development process. To properly structure and apply the evaluation feedback from every cycle, functional requirements to be implemented in the next iteration are to be made after every evaluation round. By providing good and thorough functional requirements, it is easier to evaluate and test the new features of the system, as their functionalities have been explicitly defined.

4.6 System development method

Developing the system requires a set of methods and processes to be chosen to support the development properly. The subsections below describe different techniques and tools used to aid the development process.

4.6.1 Agile software development

The domain of auto-generating programming exercises using syntactic methods is rather unexplored, and the requirements for the prototype at the beginning of the first iteration were somewhat open-ended. As a result, a development method, such as the waterfall model, would not be appropriate for the project. The waterfall model is highly based on an initial plan, and therefore not a good fit for the project. An agile-development process, on the other hand, would fit the project very well.

Agile development method is highly linked to an environment where the requirements and solutions continuously evolve and change. Such a method fits very well with the iterative

nature of design science and the current state of the domain. Since many of the approaches that are being tested in the project has not been explicitly tested before, it is crucial to be able to adjust and modify the requirements during the project to test and evaluate the solution space properly. Even though the project is only handled by one person, a tool for keeping track of the requirements during the project is highly recommended. Kanban was, therefore, chosen to handle this aspect.

4.6.2 Kanban

A Kanban board, which comes from the lean development method Kanban, was decided to be used. The board contains multiple columns and allows for requirements within each of the columns. By using the Kanban board, visualizing and keeping track of the progress during the project was simplified by a large margin and helped structure the work. Trello, an online Kanban board website, was chosen as the tool for implementing the board. Using enables a clean way of structuring requirements and tasks that has to be done. It also helps with structuring and presenting the tasks that have to be done every sprint by having an easy to use interface that makes it easy to make changes on the fly.

Design Science Guideline	Usage in the project
Design as an artifact	The artifact designed is specifically created to solve an organizational problem related to creating exercises. The design is created to allow lecturers and professors to save valuable time they otherwise would have spent creating exercises. Through the result section, the implementation is carefully explained, providing clear information on how to implement it.
Problem Relevance	The relevance and motivation for the problem is presented in Section 1, where it explains why the artifact could provide large organizational benefits. In addition, section 2 gives insight into existing solutions.
Design Evaluation	The end users of the artifact is the lecturers and professors in various CS1 courses (and the end-users are the students the exercises are generated for). Evaluation of the project was therefore done by qualitative interviews with professors in CS1 courses on NTNU. The interview style used was semi-structured interviews [35]. A semi-structured interview is a more open-ended interview technique, where the interviewer follows a set of pre-determined questions and allows for conversation that is not strictly related to the question at hand. Through functional requirements derived from the evaluation, changes to be made in the new iteration are established.
Research Contributions	The conclusion section of this report found in Section 7 gives a clear conclusion to the research questions proposed, and also gives a clear summary of the contributions of this report.
Research Rigor	Research rigor is achieved by performing a careful evaluation of the possible techniques that can be used to generate program tracing exercises before choosing what technique to utilize. Additionally, an iterative process was used to allow for valuable evaluation of the artifact during its development.
Design as a Search Process	The inherent iterative process of design science was utilized in the project. Through iteration, a search was performed to find a good implementation that solved the presented research questions.
Communication of Research	Through the three final chapters, the report presents both the technical aspects of the artifact, as well as its implications for the audience without any technical expertise.

Table 4.1: A table of the seven guidelines in design science and their relevance to the report.

Chapter 5

Results

Since this project was a continuation of the project described in section 2.5, the main focus was on improving the prototype with novel features that allowed for more complex and interesting exercises to be generated. As mentioned in Section 4, the prototype was developed through an iterative process with a basis from design science. During the project, three iterations were completed. The first iteration was an extension from the previous project, as described in Section 2.5. The two remaining iterations were purely to develop and evaluate new prototype features.

5.1 The initial iteration (Iteration 1)

This iteration was done to evaluate the features implemented during the final parts of the pre-project. Two of the features from the pre-project, however, were not fully finished. The prototype from the pre-project was found to have the following shortcomings:

- Randomized iterations
  - Nested iterations would use each others' index variables
  - The *it* variable would not work as intended in nested loops
- Lists were implemented in such a way that did not allow for lists within lists
- The final product of the pre-project was not evaluated

The following subsections explain how these issues were addressed.

5.1.1 Finalizing randomized iterations

While randomized iterations were implemented in the previous project, they were not fully completed. There were mainly two issues related to the iteration construct that had to be

solved. Both of them were directly related to the nesting of loops.

The first issue occurred when two randomized iterations were nested into each other and using the same substituted variable. If the randomly chosen loop types were a for in range and while loop, the loops could potentially terminate early or run indefinitely because the for loop would tamper with the while loops index variable. The problem was caused by the generator not choosing different index variable names for each of the loops. Solving the issue required using the symbol table again, allowing the randomized constructs to check for taken variable names while ensuring their own selected variable would not be taken by another structure. Thus, the randomized constructs would have to choose another name if the one they wanted to select was taken.

The second issue was caused by how the *it* variable was handled in nested loops. Since the variable pointed to by the *it* variable was reset upon exiting of any randomized iteration, the *it* variable would stop working properly if *it* was referenced below a randomized construct but within another. Line 7 in Listing 5.1 shows a line were the *it* variable would not work as intended. Solving the problem involved utilizing a stack for holding the currently reference *it* variables. The current referenced *it* variable would be added onto the stack when a for iterator is created and popped from the top of the stack once *it* is exited. This solution would ensure that the top of the stack is always the correct variable at any given time.

```
1 a = [1,2,3,4,5]
2 b = [6,7,8,9,10]
3 for a {
4     for b {
5         print(it)
6     }
7     print(it) # This it would not be substituted for the variable iterated
       over in the variable a, because the for construct above it would
      reset the reference.
8 }
```

Listing 5.1: An example of a case were the it variable would not work.

5.1.2 Re-writing list definition

One of the problems with the existing list definition was the fact that a list definition was defined as a separate production, and not part of an expression. As a result, a multi-dimensional list could never be instantiated since the contents of a list was required to be an expression. To solve this problem, the way a list was defined had to be changed. Initially, the two productions 5.1 and 5.2 were created to specifically handle list declarations. This solution, however, was far from optimal, as the definition of list meant it would not be counted as an expression and thus not work as intended.

The solution was to create a new expression production. This way, multi-dimensional lists are naturally created, since a list can be used as a bottom level expression. Production 5.3 illustrate the new production that was added as an expression. Additionally, the empty list definition was baked into the expression, reducing the total amount of productions.

```
<assign_list>
                                                                    (5.1)
            ⟶ '[' <expr> ( ',' expr)* ']'
            ⟶ '[' ']'                                               (5.2)
<expr>      ⟶ '[' (<expr> ( ',' expr)*)* ']'                        (5.3)
```

The other great advantage of re-writing this implementation is that it is now part of the default variable assignment code. As a result, there is no need to have more than one visitor for handling variable declaration.

5.1.3 Evaluation

As mentioned in Section 4, semi-structured interviews with Professors and lecturers in the CS1 courses at NTNU was used to evaluate the artifact. Table 5.4 lists the two professors who participated during the first evaluation and how long they have been teaching in the relevant course.

Professor	Years of experience
A	2
B	17

Table 5.1: Table of professors interviewed during the first evaluation.

Since this evaluation was performed on the prototype from the previous project, the two main features in focus was the randomized iteration construct and composed statement. The interview therefore consisted of three overarching questions:

1. How useful would the addition of randomized iterations be for creating interesting and educational exercises?

2. How useful would the addition of composed statements be for creating interesting and educational exercises?

3. The ground-works for the first part of the syllabus of the CS1 course at NTNU has now been implemented in the artifact. In the light of the CS1 course, which concepts would be most beneficial to focus on?

Before these three questions were asked, the artifact and some examples of exercises using the two new features were shown. The two first questions were used to start a discussion to evaluate the usefulness of the two novel features. The last question is concerned with uncovering new potential features that would improve the learning outcomes of the exercises. Table 5.2 in section 5.1.4 summarizes the answers and highlights differences between the answers.

5.1.3.1 Question 1: How useful would the addition of randomized iterations be for creating interesting and educational exercises?

Teacher A was very positive to the feature, and clearly stated that it could open the door for many interesting exercises. The significant advantage is that the difference between the different loops and iterations changes up exercises enough to the point were they are somewhat similar but different enough to prevent students from answering the next question correctly based on recall of previous answers rather than the actual tracing of the code. During the first part of the interview, an exercise where two randomized iterations were nested was shown. This exercise was rather short, but all of the professors pointed out that such an exercise would usually exceed the ceiling of how difficult exercises in the subject could be. One of the big problems with it was that such exercises combined different sorts of iteration, such as a for in range with a while loop, which is not as relevant for CS1 courses. Hence, such usage of the randomized iterations would not be very beneficial. Another problem with both of the new features is related to the text description of the exercises. With more deluded and different exercises, it becomes harder to give an accurate description of every exercise, since they might deviate a lot from each other.

5.1.3.2 Question 2: How useful would the addition of composed statements be for creating interesting and educational exercises?

Both teachers thought this feature could prove itself extremely useful when generating large amounts of exercises. Since it allows exercises to combine available variables into an expression, a large number of different exercises can be made. Additionally, it serves very well in combination with the randomized iteration construct. While there were no clear complaints about the feature, it is clear that it is currently quite limited. Since the current implementation of the artifact mostly focuses on number variables, other variables such as strings might still be implemented. Thus, the composed statement implementation would have a limited number of combinations in exercises where variables such as strings are used. A possible solution to this problem is to use techniques for converting the other variable types into numbers.

5.1.3.3 Question 3: The ground-works for the first part of the syllabus of the CS1 course at NTNU has now been implemented in the artifact. In the light of the CS1 course, which new concepts would be most beneficial to focus on?

A clear concept many students struggle with are dictionaries. Dictionaries are harder to grasp than other concepts in the course, and it is necessary to solve a considerable number of exercises to understand the concept properly. Additionally, this means that exercises using dictionaries does not need to be overly complicated, but instead teach the basics. Also, both strings, lists, and tuples are essential parts of the curriculum, and should also be prioritized. These three concepts can generally be viewed as pretty similar, and in most exercises, they are used in similar ways. While these concepts are useful on their own, they are often used in conjunction with built-in functions such as *list.append* or *string.join*. A requirement for the implementation of these three concepts is, therefore, a construct that allows built-in functions to be called on these types. The last feature discussed by all of

the professors were functions. Functions are an essential part of almost any programming language, and students usually fail to understand certain concepts involving functions such as parameter passing, immutable and mutable variables, and return values. Teacher A also suggested looking at file I/O, with a focus on reading from text-files. Here the generator would generate files with a particular structure and generate code that handles this file structure. Teacher B, on the other hand, iterated that I/O should not be in focus, only because it requires more work than the other features, and might not yield as good results compared to the work required.

5.1.4 Summary of answers

Question	Summarized answer
Question 1	Both Teacher A and Teacher B thought the generator could generate some great exercises for learning some of the simpler concepts. Additionally, Teacher A thought some of the randomly generated nested for loops could be a bit too complex due to its mixed usage of loops (e.g., a while loop within a for loop). In general, the features were well received
Question 2	Both of the teachers thought that the composed statement generation would be a great fit for most exercises. The only drawback pointed by Teacher B is the fact that it would only work on numbers, and thus, would not apply to certain exercises that did not explicitly work with numbers.
Question 3	Both of the teachers are responsible for the same course, and thus agreed on many of the new concepts to focus on. Lists, tuples, dictionaries, sets, and strings were all pointed out by both of the teachers. Teacher A also stated that functions with and without return values are something students' struggle with, and therefore should be included. Teacher A and B had conflicting views related to I/O. Teacher A thought it could make for some interesting exercises, while Teacher B thought it could potentially be too much work compared to the results.

Table 5.2: A summary of the answers during the interview for the first evaluation.

5.1.5 Requirements for the next iteration

As mentioned in Section 4, the development part in each iteration is done according to a set of requirements extracted from the evaluation of the previous iteration. From the interviews, a few specific features were pointed out as natural concepts to implement into the artifact. Those were:

- Functions
- Sets
- Dictionaries
- Strings
- Tuples

- Built-in Python functions for lists, strings and dictionaries

I/O is missing from this list. It was decided that I/O would not be as high of a priority as the other features listed, simply because the other features were requested by both teachers, while I/O was a conflicted topic.

Looking at the new proposed features in the light of the previously implemented features, a few changes would also be required to support them properly. The randomized iterator construct would need to support randomized iteration over dictionaries, sets, strings, and the currently implemented lists. To properly handle this issue, the current type system needs to be expanded to allow the construct to check the type of the variable it is iterating over, to customize the iteration to the given type. Additionally, the composed statement should be expanded to allow other variable types to be included by converting them to numbers.

A few concrete requirements for the next iteration can be extracted from these points:

- The prototype should support the generation of tracing tasks where the code includes functions definitions and function calls.
  - The functions should be able to generate both void functions and return value functions.
  - The functions should be able to support input parameters.
  - The functions defined should be callable with parameters.
- The prototype should support dictionaries.
  - Dictionaries should be defined as a variable declaration.
  - Dictionary entries should be accessed through its variable name, followed by square brackets.
  - Randomized iterations should be able to iterate over dictionaries.
- The prototype should support Strings as a primitive type.
  - A String should be defined as a variable declaration.
  - Randomized iterations should be able to iterate over Strings.
  - Concatenation of String should be allowed through the arithmetic operator '+'.
- The prototype should support sets.
  - Sets should be defined in the same manner as lists, but surrounded by parenthesis instead of square brackets.
  - Randomized iterations should be able to iterate over sets.
- The composed statement should support String variables by converting them to numbers using the *len* function.

- The type system should support the proper type checking of statically defined variables.
- The artifact should support the usage of specific built-in Python functions for lists, strings, and dictionaries.
- The artifact should support randomized string concatenation.

5.1.6 Summary of requirements

Table 5.3 presents the overarching topics for each requirement, who suggested the requirement and the requirement itself. The source of the requirement is either one or more of the teachers or the author.

Id	Overarching topic	Source	Requirement
FR1	Functions	Teacher A	The functions should be able to generate both void functions and return value functions.
FR2	Functions	Both teachers	The functions should be able support input parameters.
FR3	Functions	Both teachers	The functions defined should be callable with parameters.
FR4	Dictionaries	Both teachers	Dictionaries should be defined as a variable declaration.
FR5	Dictionaries	Both teachers	Dictionary entries should be accessed through its variable name, followed by square brackets.
FR6	Dictionaries	Author	Randomized iterations should be able to iterate over dictionaries.
FR7	Strings	Both teachers	Strings should be defined as a variable declaration.
FR8	Strings	Author	Randomized iterations should be able to iterate over Strings.
FR9	Strings	Author	Concatenation of strings should be allowed through the arithmetic operator '+'.
FR10	Sets	Both teachers	Sets should be defined in the same manner as lists, but surrounded by parenthesis' instead of square brackets.
FR11	Sets	Author	Randomized iterations should be able to iterate over sets.
FR12	Composed statements	Teacher B & Author	The composed statement should support String variables by converting them to numbers using the *len* function.

FR13	Type system	Author	The type system should support proper type checking of statically defined variables.
FR14	Built in Python functions	Author	The artifact should support usage of specific built-in Python functions for lists, strings and dictionaries.
FR15	Strings	Author	The artifact should support randomized string concatenation.

Table 5.3: A summary of the requirements from the first round of evaluations.

5.2 Iteration 2

The requirements for this iteration is summarized in table 5.3. The sections below explain the implementation of the requirements.

5.2.1 Function calls

Functions are essential to almost any programming language, including Python. Implementing functions require the usage of several constructs used in the implementation of previous features. There are two main aspects of implementing the generation of program tracing exercises where the code includes functions. The first is allowing for functions to be defined with a name and parameters, while the second aspect is allowing for function calls to be used in expressions and statements.

To allow for functions to be defined, one of the root productions of the grammar had to be changed. The code block is the highest grammar rule, which allows for 0 or more statements to be defined within it. Since functions in Python can only be defined at the highest level, a code block must have to allow for functions and statements to be defined. Production 5.4 and 5.5 represent the two productions used to define a function. The former shows the structure of a function, which consists of defining that it is a function, its name, and then a parameter list. The parameter list is a list of zero or more variable names. To properly implement a function definition, the symbol table also has to be included. Since the parameters count as standard variables inside the function scope, they have to be added to the symbol table representing this scope. An issue with doing this is that there is no way of knowing the variable type of the parameters until the function is called with values. Generally, this would not be a problem, but since other features of the generator, such as composed statements, utilize the type of the variables to combine them, the variables utilized in the function have to be assigned some type if they are to be correctly integrated in the prototype.

Additionally, the function has to be added to the highest level symbol table. Since Python explicitly requires a function to be defined before its called, all functions need to be added to the symbol table. Since the parse-tree traversal always traverses the tree linearly, a faulty

function call can be detected by checking if the function in question is stored in the highest level symbol table.

$$\begin{aligned} \texttt{<function\_definition>} \longrightarrow \texttt{<function><var\_name>} \\ \texttt{'('} \ \texttt{<parameter\_list>} \ \texttt{')'} \ \texttt{<stat\_block>} \end{aligned} \tag{5.4}$$

$$\texttt{<parameter\_list>} \longrightarrow \texttt{(<var\_name>} \ \texttt{(','} \ \texttt{<var\_name>)*)?} \tag{5.5}$$

The second part is allowing a function to be called, as, without this part, the function would be useless. The Productions 5.6 and 5.7 show the two productions required to properly call a function with its parameters. Production 5.6 simply refers to the function name and the parameters required. As mentioned, knowing the type of the variable passed into the function parameters is also essential. However, this issue does not have a simple solution, because a parameter might have different types in two different function calls. As a result, the parameters are given an open type that is always accepted when generating composed expressions, unless they are specified to not part of it. While this in no way is the most optimal solution, it is viable and enables functions to be used in conjunction with other features.

$$\begin{aligned} \texttt{<function\_call>} \longrightarrow \texttt{<var\_name>} \\ \texttt{'('} \ \texttt{<parameter\_expression\_list>} \ \texttt{')'} \end{aligned} \tag{5.6}$$

$$\texttt{<parameter\_expression\_list>} \longrightarrow \texttt{(<expr>} \ \texttt{(','} \ \texttt{<expr>)*)?} \tag{5.7}$$

$$\texttt{<return\_stat>} \longrightarrow \texttt{<return>} \ \texttt{<expr>} \tag{5.8}$$

The last required feature to allow proper function calls is the return statement. Without it, the function would only be capable of either printing values or mutating mutable variables. Production 5.8 displays the structure of the return statement.

Listing 5.2 shows how function parameters can be combined to be utilized with composed expressions.

```
# The template
a = 5
def fun(b, c, d) {
    if composedStatement() {
        return b + c
    } else {
        return a + d
    }
}
print(fun(6,7,8))

# Exercise 1
a = 5
def fun(b,c,d):
```

```
15      if c <= d or a > b:
16          return b + c
17      else:
18          return a + d
19
20  print(fun(3,5,8))
21
22  # Exercise 2
23  a = 5
24  def fun(b,c,d):
25      if d <= b and a > c:
26          return b + c
27      else:
28          return a + d
29
30  print(fun(3,5,8))
```

Listing 5.2: A template and exercises with a function call that incorporates composed expressions.

5.2.2 Tuples

Just like lists, tuples are indexed collections; however, tuples are immutable. It, therefore, has no concrete built-in Python functions to mutate it in any way. Parenthesises are used to define a tuple, and each element within it is defined the same way as a normal list. Production 5.9 show the structure of a tuple definition. Even though tuples are immutable lists, they still behave just like lists when iterating over them. Therefore, the tuples had to be added to the randomized iteration structure to allow randomly generated iterations.

$$\texttt{<assign\_tuple>} \longrightarrow \texttt{'('\ (<expr>\ (','\ <expr>)*)*\ ')'} \tag{5.9}$$

5.2.3 Dictionaries

Dictionaries are an important part of the curriculum in the CS1 courses at NTNU. It is, therefore, essential to give templates the ability to utilize dictionaries and their use-cases. Assigning a dictionary is very similar to list assignment, and is illustrated in Production 5.10. This production involves both assigning empty and populated dictionaries. The big difference between the two is that a dictionary entry is defined with two values; a key and a value. This is reflected in Production 5.11, which represents a dictionary entry. Since both the key and the value are defined as two expressions, their type can be determined upon definition. This data is stored in a separate data structure that is passed to higher-level productions during tree-traversal. This also includes storing the variable defined in the symbol table, containing a HashMap of all its values. Doing this allows other constructs in the generator to use dictionaries in a better way. An example would be to have the randomized iterator determine what type of variable it is iterating over, and then adjust its output based on this information.

```
<assign_dictionary> ⟶
                    '{' (<dict_entry> (',' <dict_entry>)*)* '}'
```
(5.10)

```
<dict_entry> ⟶ <expr> ':' <expr>
```
(5.11)

5.2.4 Sets

A set is an unordered and unindexed collection. They are defined by curly brackets, just as dictionaries, but instead of having a key-value pair for every entry, it only contains values for every entry. An empty set is achieved in Python by the call set(), since {} yields an empty dictionary. Production 5.12 shows the simple production for defining a set. In addition to adding a way of defining sets, an entry for sets was also added to the randomized iterator construct. Since sets behave rather similar to dictionaries, they are not indexed, and thus, using a while or for in range loop does not make sense. A for-each implementation was, therefore, the only iteration structure utilized for sets.

```
<set> ⟶ '{' <expr> (',' <expr>)* '}'
```
(5.12)

5.2.5 Strings

Strings are essential and one of the building blocks of CS1 courses. In the grammar, they have to be defined as the lowest level variable, to allow them to be added to the symbol table by the system currently in place. Production 5.13 shows a regular expression determining the rules for defining a string.

```
<string> ⟶ '"' (~[\r\n] | '""')* '"'
```
(5.13)

To allow a string to be concatenated, it needs to be defined as an expression. In the grammar, all other primitive variable types are defined as atomic expressions, and thus, the string is also included here.

5.2.6 Improved Type Checking

One of the underlying features of the grammar is that the atomic expressions have explicit types (number, string, or boolean). While this feature was fundamentally implemented in the grammar, it was never utilized in the parser other than to determine the variable type to be inserted into the symbol table. Since other features of the prototype, such as the randomized iterator, could significantly benefit from having the ability to check the type of variables, a system for passing variable types through the tree was implemented. The idea behind this system is to allow lower-level productions such as atomic expressions or variable declarations to return a value containing information about the type and variable

name. By doing this, higher level productions are given access to more information about the variables utilized.

Since ANTLR4 generates visitors for every production that is allowed to return generic values, this was utilized to pass information through the tree. The type to be passed as a return value from the visitors was named AtomicVariable, which took two parameters, first the value as a generic object, as this value could vary based on the type of variable to be returned. The other parameter contains information about what type of variable it is. This implementation also made it easier to extend the Symbol Table, as it allows the symbol table to be populated at a higher level than previously.

Another major issue related to types are data-structures that can contain variables of multiple types. Lists or dictionaries are examples of such structures. The issue with these structures is that there is that the generator cannot with one hundred percent certainty tell what type each element within the structure is. Because the prototype operates at compile-time, it is unaware of what might potentially happen at runtime. The issue arises from the parameter variables in functions. Since a function's parameter can be of any type, depending on what is passed to it during a function call, mutation of lists in a function is hard to track. A result of this is that when generating composed statements with elements from a list, there is no trivial way to determine what type each element is. While this is a hard problem, it is still possible to solve by making an assumption. It is possible to assume that if every element in the list is of a single type when its assigned, then no other type than the initial will be appended to the list. This assumption makes it possible to check the elements of a list when it has been assigned and give it a specific type (e.g., a list of strings). Doing this allows constructs such as the composedStatement to accurately determine the type of every element, giving it more information to create better statements. The assumption, however, is at the mercy of the template creator. If the template creator decides to append elements with a type that is not in the initial list, the composedStatement has the potential to generate erroneous statements, resulting in invalid Python code. Therefore, this feature requires a more in-depth evaluation to determine if making such an assumption provides more benefits than potential problems. Due to these potential problems, a simple property was added to determine if such an assumption is to be made. Hence, the assumption can be removed by merely disabling it in the settings.

5.2.7 Expanding the randomized iterator to include different types

To expand the randomized iterator to include different types, the symbol table needs to be used. Since some types, such as dictionaries, do not have an index to iterate over, loops such as for in range will not work on this data-structure. To solve this, the generator needs to determine the type of the variable to iterate over before it generates a loop. A list containing all variable types that can be iterated over was therefore created. This list contains: Lists, Tuples, Dictionaries, Sets, and Strings. Iterating over lists, strings, and tuples can be done in the same way, as a tuple is just an immutable list, and a string is essentially a tuple of characters. The list implementation is therefore done the same way for lists, strings, and tuples. Dictionaries and sets have to be handled differently, however. Since these are not indexed by numbers (and sets are not indexed at all), the for in range implementation is not possible. A while loop over dictionaries and sets are technically

possible; however, it does not make much sense. Hence, only the for-each implementation was included for these two data-structures.

5.2.8 Built-in Python Functions

Another feature that would allow for better exercise generation is to allow certain built-in functions from Python to be used. Examples of this would be list.append() and list.clear(). These features are commonly used in the CS1 courses, and should, therefore, also be reflected in the prototype. To properly add such functions, the type checking system needs to be utilized. Since every function is to be applied to a variable of some type, the visitors need to be able to determine if the function called from the current variable exists for that type. An example would be to call the .append() function on an integer, which would not work. To solve this problem, every built-in function gets a separate production consisting of the variable being called and the specifications for that function. Productions 5.14 to 5.19 show five of the implemented functions. Each production has its own generated visitor that handles type checking for that specific function. The # behind every production denotes that the productions have a specific visitor. This allows multiple productions to have the same visitor. The idea behind labeling productions is that some of the productions can be handled in the same way. It is, therefore, sensible to allow multiple productions to utilize the same visitor. Hence, functions on lists can all be handled by the same visitor.

To validate the variable type, the symbol-table is consulted. Based on the type of information, it decides if the function exists for the given variable. As seen, the first part of each production is the same. While this looks redundant, it is required because every production needs to specifically know the variable name to do the type checking.

```
<variable_function>
```
(5.14)
```
        ⟶ <var_name> '.'  <append> '(' <expr> ')'#listFunction
```
(5.15)
```
        ⟶ <var_name> '.'  <remove> '(' <expr> ')'#listFunction
```
(5.16)
```
        ⟶ <var_name> '.'  <clear> '(' <expr> ')'#listFunction
```
(5.17)
```
        ⟶ <var_name> '.'  <join> '(' <expr> ')'#stringFunction
```
(5.18)
```
        ⟶ <var_name> '[' <expr>?  ':'  <expr>?
        (':')  <expr>?)?  ']'#lis
```
(5.19)
```
        ⟶ <var_name> '.'  <random_slice> '(' ')'#randomStringSlice
```

As seen above, Production 5.18 has a different structure than the other four. This is because the production handles list slicing. A decision was made to treat list slicing as a function since it only can be performed on lists and tuples, and thus requires the same type-checking as the other productions. The production simply allows the template to define list slices where either of the three parameters in the square-brackets can be taken. An example would be ***string[::-1]***, which simply reverses a whole string (Note that a string can be looked at as a special tuple, and therefore also applies to this production).

Production 5.19 were also added, and essentially generates random slices using the same syntax seen in Production 5.18. To generate random slices, it utilizes the same constructs as the randomized iteration construct. It generates random lower and upper boundaries and a direction to slice (backward indicates that the string is reversed). Based on these values, it creates a string slice. An important feature of this function is that certain boundaries, such as start slice from index 0 or end at the length of the string is omitted since these are implicitly specified unless otherwise defined. A concrete example would be ***list[0:len(a):2]*** and ***list[::2]***, which are equivalent. Omitting implicit values was implemented to make the generated code snippets as realistic as possible. If the generated snippet simply had reverted boundaries (from the length of the string to the index 0) and a negative step length, it essentially means the list is to be reverted. The best practice for this is by simply writing ***list[::-1]***, which the generator achieves by properly checking the boundaries and step direction.

All of the productions above except 5.18 and 5.19 is implemented in a very general way, where the main distinction between them is what type the function is allowed to be called from. Such a solution makes adding more built-in functions easy, as the only required information is what type the function call needs to be allowed.

5.2.9 Evaluation

The evaluation for iteration 2 was done in conjunction with three professors. The evaluation's main focus was to determine the suitability of the features implemented and look at the possibilities for the final iteration.

Professor	Years of experience
A	7
B	2
C	17

Table 5.4: Table of professors interviewed during the second evaluation.

While many features were implemented during the iteration, much of the work was not as visible as previous iterations. A clear example is the improved type-system, which required much work, but was only visible in a small part of the generator. As a result, more focus was put into the future features than in previous iterations. The questions asked and discussed around during the evaluation were the following:

1. How useful would the addition of functions be for generating interesting and educational exercises?
2. How useful would the addition of strings, sets, tuples, and dictionaries be for generating interesting and educational exercises?
3. There is one iteration left of the project. Based on the current implementations, what features would you like to see in the last iteration?

Before the three questions above were asked, the prototype was shown with some sample exercises utilizing the new features. The two first questions were used to spark a discussion

to evaluate the features of iteration 2, while question 3 was used to determine new features to be implemented in the final iteration.

5.2.9.1 Question 1: How useful would the addition of functions be for generating interesting and educational exercises?

Teacher A gave a positive response to the feature but thought the usefulness for generating new exercises over time would be reduced because it had no features that allowed for direct randomization. However, since functions are a significant part of most CS1 courses, the feature makes sense and enables the generation of exercises relevant for the course. A suggestion was to allow for usage of pre-determined common functions for algorithms such as sorting. These functions would be pasted into the code during exercise generation to test the students' knowledge. Both teachers B and C gave positive feedback and, in general, expressed that the usefulness of the generator increased with the addition of functions. None of the two teachers had any direct criticism or changes that should be made, and teacher B stated that learning the concepts of the functions is most important.

5.2.9.2 Question 2: How useful would the addition of strings, sets, tuples, dictionaries, and built-in functions be for generating interesting and educational exercises?

The exercises generated using the mentioned features with randomized constructs create very technical exercises. Teacher A thought the exercises were very specific and would test precise knowledge, such as understanding exactly how slicing works. While such features could create interesting exercises, they might not be as attractive in the larger picture. Teacher B thought the features would be interesting and that they would increase the diversity of the generator. On the other hand, these features, in conjunction with the randomized constructs, could increase the risk of exercises becoming too obscure, thus reducing the exercises' ability to test the students understanding of concepts. As a result, the exercises generated might be difficult because of obscurity rather than testing difficult concepts. Teacher C gave a positive response and thought the implemented features would enhance the exercises generated.

5.2.9.3 Question 3: There is one iteration left of the project. Based on the current implementations, what features would you like to see in the last iteration?

As mentioned briefly, during question 1, teacher A thought that having a set of commonly used functions available in the generator would make much sense. Since there are certain functions students should learn during a course, having these as a part of various exercises would provide better learning outcomes. An example would be to have allowed the usage of merge-sort, where the function that does the sorting is pasted to the top of the exercise during generation. This would not provide any more randomization to the exercises, but it would allow students to understand essential functions better.

Both Teacher B and C thought that new features to the generator should not be the main focus since there was only one iteration left. Both expressed that another important aspect

of the generator and the template language would be how easy it is to use. This was emphasized in terms of the quality of feedback the ”compiler” creating exercises from the template would respond with. An interesting approach both of the teachers lightly touched on would be to look at how to give good error messages to the template creator if parts of the template has invalid syntax or logic. Such a feature would make it easier to learn how to use the generator.

While discussing the features, Teacher C also thought the ability to tag exercises with relevant concepts would be a great idea. Tagging could prove useful when storing the exercises, as it would be easier to find relevant exercises by fetching them based on their tags.

5.2.10 Requirements for iteration 3

Based on the feedback from section 5.2.9, three new features came up:

1. Error handling and responses
2. Tagging of exercises
3. Commonly used functions should be included with code in every exercise if used

The first feature is all about making it easier to utilize the generator and its template language. Feature 2 looks at ways to classify the exercises generated based on the concepts it utilizes, while feature 3 aims to increase the students' understanding of important functions by including them in the exercises.

Table 5.5 below illustrates the functional requirements derived from the feedback in section 5.2.9.

Id	Overarching topic	Source	Requirement
FR1	Error & Feedback	Teacher A & B	The generator should give feedback of the line, position and offending symbol when a syntax error occur.
FR2	Error & Feedback	Teacher A & B	The generator should give feedback of the line and error when a logic error occur.
FR3	Tagging	Teacher C & Author	The generator should be able to tag exercises with concepts it tests.
FR4	Common functions	Teacher A	The generator should have a set of commonly used functions that are copied into exercises upon generation when used in the template.

Table 5.5: A summary of the requirements from the second round of evaluations.

5.3 Iteration 3

5.3.1 Tagging topics of an exercise

As mentioned in section 2.2, meta-data linked to exercises could allow researchers to develop existing CI's further and also create new ones. This makes tagging exercises with relevant topics and concepts relevant. Additionally, having specific tags for topics would allow students to target exercises containing specific concepts they want to practice directly. Therefore, it is desirable to automatically allow the auto-generator to tag exercises with topics and concepts utilized in each exercise as they are generated. However, to tag the exercises, proper and correct tags have to be identified.

Chinn et al. conducted a study examining exams and found a set of important topics covered during exams [37]. A small subset of these topics are illustrated below:

- Loops
- Arrays
- Assignment
- I/O
- Parameter passing
- Strings
- Arithmetic operators

The topics presented by Chinn et al. serves as a good list for topics to be used for tagging generated exercises.

Since the generator utilizes ANTLR4 to define and parse the specified grammar, it has access to visitors for all productions within the grammar. These visitors allows the generator to determine what constructs are being used in an exercise specifically. An example would be a loop. Since a loop is specifically defined within a production, the visitor that handles that production could automatically add a tag to that exercise if it is executed. Figure 5.1 shows a very simplified version of the parse-tree generated from the exercise template illustrated in Listing 5.3.

```
a = [1,2,3,4,5]
for e in a {
    print(a[i])
}
```

Listing 5.3: Code illustrating a for each iterator being used on a list.

For each node in Figure 5.1, there exists a visitor. Even though the parse-tree has been drastically simplified compared to the real parse-tree, it still contains enough information to generate relevant topic tags. By looking at the small list of topics above, both assignments and loops are part of the suggested topics. As seen, the assignment node exposes

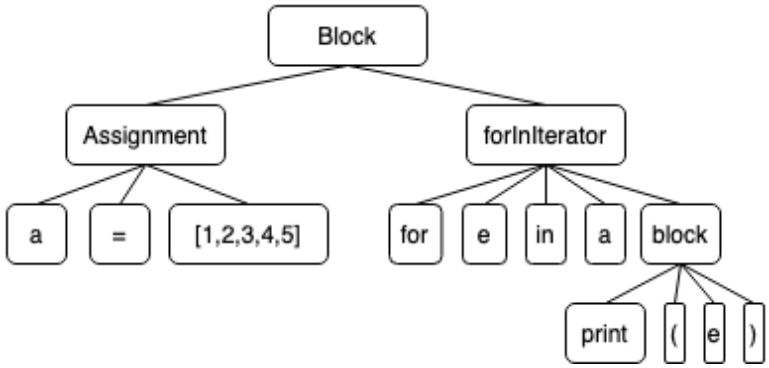


Figure 5.1: A simplified parse tree generated from the snippet in Listing 5.3.

the assignment topic, meaning the assignment visitor could add the assignment tag to the exercise. The same principle goes for the loop, as the forInIterator visitor could assign the loop tag. Additionally, the code also iterates over a list (array). Since the symbol table also stores what type of variables have been assigned, one could simply look up all variables referenced to see if any of them are of the type list. In this case, the *a* reference in the for-each loop references a list, and therefore that visitor could apply the array tag.

5.3.2 Error handling and feedback

Another essential aspect when looking at the technique for generating exercises is its capabilities to give error messages and warnings to the user during exercise generation. Even though it might be capable of generating interesting exercises, its usefulness is diminished if it is hard to learn or fails to give proper feedback to the user. Therefore, creating a good generator based on templates is an integral part of creating a sound system for handling feedback and errors.

Generally, there are two types of errors that have to be handled; syntactic and logical errors. The former is directly related to the characters used in the template. A syntax error can formerly be defined as an error that occurs when the parser fails to find an available production that fits the next character in the template, and therefore causes the template to be invalid with respect to the grammar. A logic error can be defined as an error that occurs when the syntax is correct, but the logic of the template is faulty. An example would be using the plus (+) operator between an int and a list. While the grammar might allow this, the logic in itself does not make sense (when looking at the features of Python), and the generator should, therefore, report this issue. The two subsections below explain how error handling and responses were implemented in the prototype.

5.3.2.1 Syntax errors

As mentioned, syntax errors are issues where the input does not match the productions of the grammar. They are therefore detected upon parsing the template language. The generator is implemented with the parser-generator framework named ANTLR4. ANTLR4

allows the user to attach custom error listeners to the parses. Doing this allows the developer to be notified when an error occurs. During parsing, the only error possible are errors where the syntax is not valid. Hence, the detection of syntax errors is, therefore, achieved in the generator by attaching a custom error listener to the parser.

When the error listener is invoked, it will extract information from the error at hand. Since the error is a result of a syntax error, it contains information about what caused it. The relevant information extracted by the listener is the line number and the character position that caused the error. Additionally, it can extract the offending symbol, e.g., the symbol that caused the error. Once the information is acquired, the listener throws a custom SyntaxErrorException, which can be caught by the generator. The message attached to the SyntaxErrorException is crafted to an easily readable message consisting of the information extracted from the parser. When the generator catches the SyntaxErrorException, it responds to the exercise generation request with the error message.

5.3.2.2 Logical errors

The other category of errors are linked to invalid logic. These errors can only occur if there were no syntax errors, because these errors are generated during tree traversal, implying that the syntax was valid and hence a parse-tree could be generated. There are multiple logic errors, some are inherited by Python, while some are specific to the template-language. The way of handling the errors is equal to the SyntaxErrorException, where an exception is thrown, then caught by the generator, and passed on as a response.

Errors inherited by Python are mostly related to expressions between different variable types. The example with adding using the logic operator plus (+) between a list and an integer is such an error. To catch these errors, a set of rules has to be established. These rules explicitly determine what types can interact with each other and with what operators. As such, the rules state that a list can be multiplied (*) by an int and that the expression should be evaluated as a new list. To implement these rules, all productions with expressions had to be re-written. Previously they would not care about what types they were given, which could lead to errors, but since there was no logic in place to determine what was wrong, a proper error message would not be possible to generate. With the new logic, error messages are entirely able to determine if an expression is valid, and if it is not, it can explicitly tell what is wrong, by pointing to the variable types and the operator. Another issue related to Python is function calls. While functions could be defined and called in the template, there was no logic in place to determine if the function call was valid. An example would be a function **fun** with three parameters. If **fun** were called with two parameters, nothing would happen, as this error could only be detected during parsing. To adequately detect and report such errors, logic was implemented to count the number of parameters in the function definition and throw an IllegalParameterExpression if it was called with the wrong number of parameters.

Errors specific to the template-language are related to the constructs mentioned in previous chapters. These are usually caused by constructs receiving variables of invalid types. An example would be the randomized iteration construct. It allowed for any iterable type to be used, but if an invalid type were provided to the construct, it would fail with an exception,

but the exception did not contain important information about why it failed. The exceptions were therefore expanded to utilize custom exceptions. While error handling in these constructs were already implemented, new exceptions had to be generated to categorize the errors correctly.

All of the logic above, including exceptions, also contain the offending line number in the template.

5.3.3 Evaluation

As mentioned, the evaluation of the last iteration was performed through a test where the professors were given access to the prototype itself, along with documentation. Because the professors themselves are the experts at creating exercises for obligatory deliveries and exams, they were given free reigns to test generating exercises they could see fit. To inspire their creativity, the documentation also contained multiple different example templates that could help with inspiration.

The test's main objective was to see how suitable the generator would be for their CS1 course while also looking at how easily they were able to understand and use the template language. Based on the objectives, qualitative responses would be the most interesting to look at. Each participant was, therefore given a set of questions to answer after the testing was done:

1. How many years of experience with CS1 courses do you have?
2. Did the error messages and warnings affect your ability to create correct templates?
3. Could the exercises you generated have use cases in your course?
4. Is there any features you wish would be implemented to further improve the generators ability to create exercises?
5. Do you see any drawbacks with using the prototype to generate program tracing exercises?
6. Would you be interested in using such a tool to create exercises in the future?
7. Do you have more feedback not included in the previous questions?

Due to the Covid-19 related lockdown of Norwegian university campuses in the Spring of 2020, performing all of the planned tests became a significant challenge. All of the planned participants were teachers of different CS1 courses at NTNU, and due to the lockdown, their courses required re-structuring. The large amount of additional work meant the test needed to be down-prioritized, and as a result, only one professor had the opportunity to participate. In the sections below, the participant is referred to as Teacher A.

5.3.3.1 Did the error messages and warnings affect your ability to create correct templates?

Teacher A had mixed thoughts about the error messages and warnings. While some of the errors gave specific feedback about what went wrong and where it went wrong, others

did not give clear enough information about what went wrong. Unexpected errors partially caused this during the testing. Teacher A stated that it could be used in a real context based on the error messages that gave proper information.

5.3.3.2 Could the exercises you generated have use cases in your course?

Based on the examples from the documentation as well as the testing performed by Teacher A, a tool like the prototype could be very useful for creating program tracing exercises. In its current state, however, a certain degree of manual inspection must be done to assure that the exercises are of decent quality, as the quality of the generated exercises varies. Additionally, it would have to have a support system around it. For example, storing templates and exercises for different parts of the curriculum in a manner that allows the students to access them easily.

5.3.3.3 Is there any features you wish would be implemented to further improve the generators ability to create exercises?

Teacher A thought that the generator covered most of the important aspects of their CS1 course. In general, the most important focus point would be to fix all of the small errors (Such as wrong return value for specific built-in functions) contained in the template language to make it more fluent. The one feature not covered in the CS1 course is I/O handling, which could be something to expand upon. Additionally, it could be interesting to generate textual descriptions of the exercises. Such a feature would allow a new type of exercise where the students' are given a description of the exercise and tasked with writing the code that matches the description. These exercises could then be corrected by comparing them to the exercise generated.

5.3.3.4 Do you see any drawbacks with using the prototype to generate program tracing exercises?

Teacher A was a bit unsure of any particular drawbacks with the prototype. In general, the most significant drawback was the small errors linked to the template language. Additionally, it was a bit hard to understand which parts of the template language causes changes in each exercise. It could potentially be helpful to have even more complex examples to show other sides of the generator.

5.3.3.5 Would you be interested in using such a tool to create exercises in the future?

Teacher A thought such a tool could be handy for generating exercises. The concept has potential, but it would require a more stable and polished prototype. For it to be useful in a course, it would have to be implemented in a system that is capable of storing the exercises as well as testing the students.

5.4 Duplicate exercises

During the development, code was implemented to measure duplicates for different templates. The code simply compares the code for every generated exercise, and if two code snippets are identical, they are deemed as duplicates. The percentage of duplicates out of total exercises was measured on some of the templates used during evaluation, and is presented in figure 5.2. The figure measure duplicates for 100, 500, and 10000 exercises. Each set size of generated exercises was generated 100 times, and the results are the average of the 100 times generated. The corresponding table with absolute numbers of duplicate exercises and the corresponding standard deviation is shown as Table 5.6. The four templates used to sample the duplicate count can be found in Appendix K.1, in Listing 7.2, 7.3, 7.4 and 7.5. Note that template 4 has correct data entries; there are just 0 duplicates for all entries.

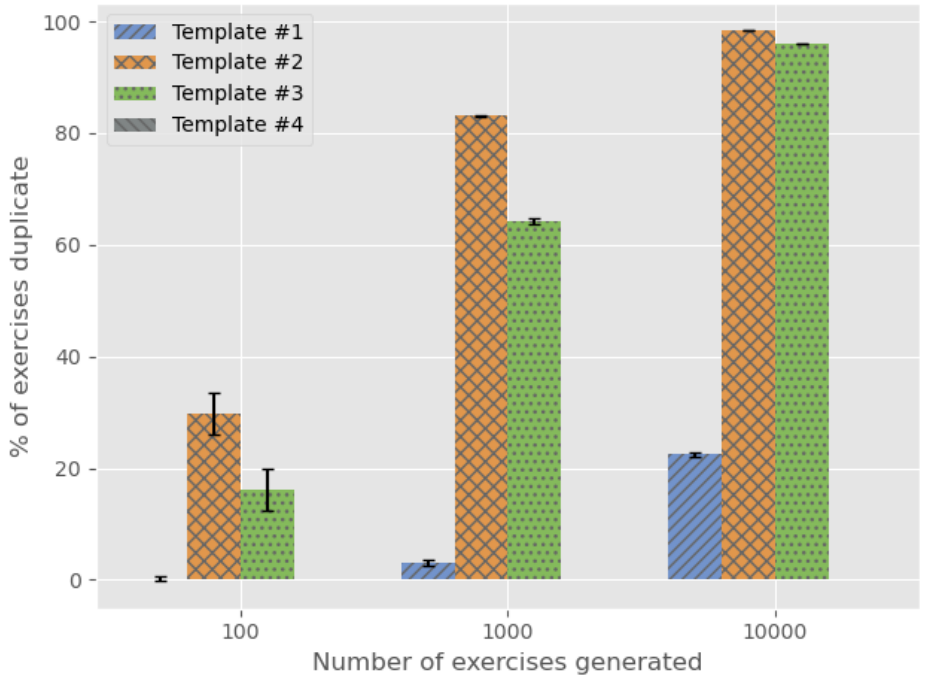


Figure 5.2: Percentage of duplicate exercises from four selected templates.

As seen in the graph, there are templates 1 and 4 with 0 or very close to 0 duplicates for 100 generated exercises. Templates 2 and 3, however, have around 30% duplicates, and both have a standard deviation of around 3.5% (Depicted as the black lines on the top of the bars). When increasing to 1000, the number of duplicates increases drastically for templates 2 and 3, while the standard deviation drops very close to 0. n, as depicted by the little line at the top of the graphs. Template 1 has around 3% duplicates. When increasing

Template #	Duplicate of 100	σ	Duplicate of 1000	σ	Duplicate of 10000	σ
1	0.22	0.44	30.29	5.71	2245.92	31.11
2	29.89	3.73	831.68	1.21	9830.0	0.0
3	16.13	3.71	642.47	5.82	9592.0	0.0
4	0	0	0	0	0	0

Table 5.6: Table containing the average number of duplicate exercises for each generated template and the standard deviation when running the generation 100 times.

to 10000 generated exercises, both Template 2 and 3 stabilizes at 98% and 96% duplicates respectively with precisely 0 in standard deviation. As seen in the Table 5.6, there are precisely 9830 and 9592 duplicates. During all of the tests, there were no duplicates in template 4.

Chapter 6

Discussion

This chapter will be discussing the results from the design science process in the light of the four research questions:

1. What solutions of auto-generating programming-exercises have already been proposed in research-literature?
2. What is the best approach for generating a large amount of program tracing exercises?
3. Which parts of CS1 courses would the generated exercises be most useful for?
4. Are there any significant technical limitations related to the approach chosen for generating exercises?

6.1 Evaluating the generator

6.1.1 Implementation and testing

The results from the test, as shown in Section 5.3.3, highlighted both advantages and disadvantages of the generator. From the testing, it became clear that one of the main drawbacks of the prototype, in particular, was its susceptibility to errors related to the template language. Since a large part of the workload related to the generator revolved around creating and implementing a functioning template language, it is also naturally more prone to errors. Since all types, functions, and constructs has to work exactly as described when creating a programming language, any implementation mistakes could easily cause errors when using said language. While the type system of the prototype was in place, it became clear that the system could use even more testing. It would be possible to test the prototype even more by conducting a more extensive test on even more lecturers and teachers to identify and fix the template language's remaining issues. Additionally, creating a suite of

tests matching the language's specifications could also provide an excellent way to identify bugs. By implementing unit tests checking the type and return values of variables and built-in functions, small errors could be caught early without direct human testing, and new errors could also be prevented during further development.

Another point presented during the test was the ability to learn how to use the generator. Before the test was conducted, the two primary tools for learning were the documentation given to the testers, as well as the error and warning messages given by the generator. Based on the response from Teacher A in Section 5.3.3.1 and Section 5.3.3.4, some time was spent understanding how the generator worked and how it converted a template into exercises. Initially, the subject struggled with understanding the difference between some of the generated exercises, but as they tried new templates, it became more evident. For the prototype to be even more usable, a more in-depth description of how it generates exercises is required. One of the most natural approaches to solve the problem would be to provide an even more thorough documentation explaining more of the generator's underlying techniques, as well as concrete examples of what happens when the generator converts a template to a set of exercises. Since the system's users are seasoned programmers, having such a technical description and explaining the inner workings of the system would be feasible. Additionally, the error messages and warnings could be further developed to give an even more explicit description of the problem, since responding with error and warning messages in ”dangerous” situations are important [38].

The last point discussed during the test was the prototype's usefulness in the CS1 course taught by the test subject. While the subject agreed the prototype could prove useful in their course, based on the feedback, two main issues have to be resolved for the prototype to be fully use-able:

1. The template language has to be stable, without any errors or small bugs
2. The generator has to be incorporated into a helping system for the students

The first issue is linked to the technical implementation of the generator itself, and is described further in section 6.3.

Issue #2 is related to the system around the generator. While the generator itself is capable of generating exercises that can be used in various CS1 courses, its usefulness is diminished if it cannot be implemented into existing or novel e-learning systems. Making the prototype more useful for these courses would require a design that allows for integration with other systems. In its current state, the prototype does utilize a REST-API to retrieve templates and return exercises and answers, but this is also the only end-point implemented. Making a standardized API with proper documentation would allow for implementation into existing and novel learning systems without much modifications to the generator.

6.1.2 Duplicate exercises

As seen in the graph from Section 5.4, templates 2 and 3 receive an increasing percentage of duplicates when increasing the number of exercises generated. When hitting 10000 exercises generated, they have a standard deviation of precisely 0, which is a clear indicator

that they are hitting a cap of possible exercises to generate. When looking at the increased percentage of duplicates while increasing the number of exercises generated, this makes sense. Since there is a maximum number of exercises a concrete template can generate, the percentage of duplicate increases as its getting closer to having generated all of the exercises possible, which explains the statistics of templates 2 and 3.

Based on the way the generator creates exercises, it is also possible to calculate the total number of possible exercises a template can generate. The number of possible exercises is directly affected by the different constructs utilized in the templates. Many of the constructs also rely on randomization, and an increase in the number of such constructs also increases the number of possible exercises. To calculate all of the different exercises a single template can generate, one has to take the product of all the different outputs of the random constructs in a template. Equation 6.1 shows the formula. Note that each entry in the set *con* represents the number of different outputs a construct can generate. A simple example of this formula could be calculating the maximum number of exercises for the simple template shown in 6.1. In the template, there are two randomized constructs, namely the randInt functions for a and b. These both range from 1 to 10, meaning each construct has 10 different outputs. Using the equation, we find that there are 100 different combinations. This can be confirmed by generating an adequate amount of exercises (where the standard deviation is zero) and confirming that the number of duplicates is n - 100, where n is the number of exercises generated.

```
a = randInt(1, 10)
b = randInt(1, 10)
if a > b {
    print(a)
} else {
    print(b)
}
```

Listing 6.1: A very simple template used to illustrate the concept of calculating duplicates for the exercises.

The equation can also be used to confirm the duplicate numbers in Table 5.6 for template 2 and 3 when generating 10000 exercises. As seen, template 2 shown in Listing 7.3 from Appendix K.1 utilizes a randomized list and a randomized iteration. The randomized list has five different outputs, and iteration construct has a total of 34 different ways to loop over the list. Using the equation gives a total of 170 unique exercises for template 2, which confirms the number of duplicates shown in Table 5.6 when subtracted from the total number of exercises generated. Template 3 also has two different randomized constructs: a randomized iteration and a randomized string slice. The randomized iteration has 34 different outputs, while the string slice has 12 different outputs. This gives a total of 408 unique exercises, which also matches the table entry for template 3.

As seen in the graph from Figure 5.2, the number of duplicates highly depends on the template. While some templates allow for a vast amount of unique exercises, a large variety of exercises might not be required. Template 2 and 3 from the graph was shown to have 170 and 408 distinct exercises, respectively. In a course containing 1000 students, the

$$exercises = \prod_{n \in con} n \tag{6.1}$$

Figure 6.1: To equation to calculate the number of unique exercises generated by a given template. The set named *con* represents every randomized construct in the template and the number of different outputs the construct can generate.

two templates mentioned would generate more than enough exercises. Since the exercises generated are rather simple and do not require much time to solve, students are less likely to cheat, regardless of them having the same exercise. Additionally, it is beneficial to have a certain number of students per exercise to facilitate analysis. Having too few students per exercise makes it harder to determine the difficulty of each exercise generated by the template. In practice, it means that cheating by getting the answer from someone else or someone sitting next to the student is severely reduced due to the low chance of them having the same exercise.

Duplicates can also prove themselves problematic if the creator of the template is not aware that the generator creates duplicates. Also, duplicate exercises can be extremely problematic if the generated exercises are used in exams, because it runs the risk of students receiving the same question more than once. Having duplicate exercises in a formative evaluation such as self-testing could provide the students with a false sense of accomplishment, since they might end up learning the correct answer of a duplicate exercise, rather than learning the underlying concepts. A possible solution to the duplicate issue would be to utilize the formula presented in Equation 6.1 to show the creator of the template how many unique exercises could be generated with the current template. Also, an option could be provided to the generator, where it only selects unique exercises, resulting in no duplicate exercises.

6.1.3 Ability to expand

Another essential feature to consider when evaluating the quality of the generator is its ability to be expanded with new features. A significant advantage of using the context-free grammar approach for generating exercises is not only its ability to extract much information from the template but also the ability to modify the grammar to make the template language fit the user's needs. These two advantages, in conjunction, give the developer great freedom to implement new features that can be manifested either in the generator itself or in the template language.

Being able to expand the generator past just generating program tracing exercises is also a highly valued attribute. A natural expansion from program tracing is program tracing with multiple choice answers where one answer is correct, while the other answers are wrong, but could resemble the correct answer. Generating multiple choice answers where the distractors do not utilize the exercise context is a lot easier than using distractors that utilize the context. Hence, generating "dumb" multiple choice answers could be done regardless of the approach used to generate the exercise. Both the CFG and the SCT approach can easily generate the answer for a program tracing exercise and could, therefore, pick random

answers, very similar to the correct answer as distractors. Such an approach, however, does not allow for exciting distractors.

A way to expand the generator to generate interesting distractors is to look at the way the generator itself generates exercises. Since it highly relies on randomizing certain constructs defined in the template language, it is possible to generate answer distractors by tweaking the randomly generated constructs. An example would be when generating an exercise with a randomized iterator. Tweaking the number of iterations done by, e.g., changing the start or end index would allow for a very similar answer to be generated that could easily capture a mistake. Such an approach is easily implementable in the prototype. Generators using the SCT approach, on the other hand, lack the information about loops and iterators that would enable the generation of such distractors. The same technique could be applied for other randomized constructs as well, and it is generally easier to generate distractors utilizing the context of the exercise, such as the example above, when utilizing the CFG approach.

6.2 Usage in CS1 courses

Exercises are used in several parts of a CS1 course. It can range from exam exercises to simple self-testing exercises where the exercises aim to teach more about the related concepts or to perform self-evaluation. This discussion below focus' on the usefulness of the generator concerning three specific use cases in CS1 courses:

1. Exam exercises
2. Obligatory exercises required to take exam
3. Voluntary test

6.2.1 Exams

Out of the three use-cases mentioned above, the hardest one to satisfy the requirements of is the exam exercises. Exam exercises have a set of requirements that has to be satisfied to be utilized in an exam. Bloom's taxonomy describes six different levels of questions that are used to evaluate their quality [39]. Figure 6.2 shows the six different parts of the taxonomy, and their related levels. The taxonomy is related to all types of questions, and as such, all of its parts are not directly applicable to the program tracing exercises generated by the prototype. In general, the two most applicable classifications to program tracing exercises are analyzing and applying. Since the program tracing exercises generated are based on the concepts, the students' are required to analyze the questions and apply their knowledge about the relevant concepts to solve them. According to the taxonomy, the exercises would fit within the classifications of exercises used on an exam.

However, satisfying parts of the taxonomy does not imply that the questions themselves are good exam questions. If exams contain randomized questions for every student, fairness between the exercises is extremely important. The prototype mostly relies on randomization techniques to generate unique exercises. This randomization creates opportunities for

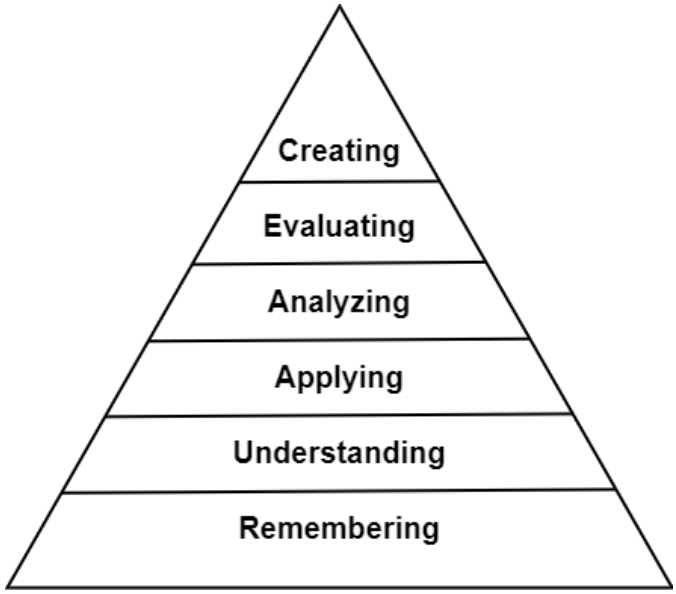


Figure 6.2: The six classifications of Bloom's taxonomy.

some exercises to be "lucky". In reality, this could mean the exercise depends on some statement that never becomes true, meaning that the student only needs to evaluate one simple if statement rather than the whole exercise, and thus the exercise becomes trivial. Mathew Hillier suggested that one of the biggest concerns students' had when taking digital exams containing questions from a random question bank is the fairness of the exercises [40].

Solving the exercise difficulty issue would require some form of metric to measure the difficulty of each exercise generated before the exam, then assigning equally difficult exercises to each student. Kasto and Whalley propose a set of metrics to measure the difficulty of program tracing exercises in a Java CS1 exam [5]. While the target language was Java, many of the metrics are still relevant for the prototype. Figure 6.4 display the metrics. All of the metrics not related to object-oriented programming in the *basic* column is accessible through the parsing/traversing steps of the generator. Additionally, most of the other metrics marked in the *structural* column can be measured in the generator. The two measures for complexity: Cyclomatic complexity and nested block depth are also assessable, though a bit more complicated than the former metrics. For program tracing exercises, the researchers found a significant correlation between performance and the metrics: cyclomatic complexity, nested block depth, and a dynamic metric counting statements executed along the correct path of execution, indicating that these metrics could be useful for the generator to calculate.

The cyclomatic complexity can be measured by using equation 6.3 [41], where V(G) is the complexity, e is the number of edges and n is the number of nodes. Using the grammar, the number of nodes can be calculated by counting the number of statements, while the edges

$$V(G) = e - n + 2 \tag{6.2}$$

Figure 6.3: Formula for cyclomatic complexity.

are found by counting all productions using branching (e.g., if statements, loops). The nested block depth calculation is shown in Section 3.3.5. During this step, it is possible to look at the total number of scopes made during tree traversal, as this represents the nested block depth. Calculating the number of statements executed during the correct path is a bit more complex, and would not be possible to calculate with the generator. Since it is dependent on the decisions made at run-time, a tool that either interprets the generated code or instruments code to determine the path of execution in Python would be required.

Metric Type	Metric	Programming Paradigm		
		imperative	structural	object oriented
Basic	Number of lines of code	✔	✔	✔
	Number of *blank* lines of code	✔	✔	✔
	Number of comment lines of code.	✔	✔	✔
	Number of comment words.	✔	✔	✔
	Number of statements	✔	✔	✔
	Number of methods.		✔	✔
	Average line of code per method.		✔	✔
	Number of parameters.	✔	✔	✔
	Number of import statements.		✔	✔
	Number of arguments.		✔	✔
	Number of methods per class.			✔
	Number of classes referenced.			✔
	Average number of attributes per class			✔
	Number of constructors.			✔
	Average number of constructors per class.			✔
	KLCID	✔	✔	✔
Complexity metrics	Cyclomatic complexity	✔	✔	✔
	Nested block depth.	✔	✔	✔
Halstead metrics	Number of operands.	✔	✔	✔
	Number of operators.	✔	✔	✔
	Number of unique operands.	✔	✔	✔
	Number of unique operators.	✔	✔	✔
	Effort to implement.		✔	✔
	Time to implement.		✔	✔
	Program length.		✔	✔
	Program level.		✔	✔
	Program volume.		✔	✔
	Maintainability index.		✔	✔
Object oriented	Weight method per class.			✔
	Response for class.			✔
	Lack of cohesion of methods.			✔
	Coupling between object classes.			✔
	Depth of inheritance tree.			✔
	Number of children.			✔

Figure 6.4: Metrics for measuring difficulty of program tracing exercises. Reprinted from [5].

Another way to solve the issue of difficulty is to randomly pick exercises from the generated set and give every student the same exercise. Such an approach would make cheating more straightforward, as every student has the same set of exercises. It is possible to randomize the order in which the exercises appear on the exam to make cheating harder.

Randomizing the order makes cheating by looking at other students a lot harder, but still does not entirely prevent cheating. Since every student has the same exercises, it is still possible for friends to communicate the answers. Another approach to make cheating even more difficult would be to have exercises with subtle differences. These differences would hardly be visible, but would in return, slightly change the correct answer, punishing students who cheat [42].

It is still possible to utilize the generated exercises in an exam without knowing their exact difficulty. Such a case would require a sufficient amount of students and relies on scaling the points given for a question based on their relative difficulty. An example would be to choose a set of around 50 to 100 exercises for an exam with 1000 students, where the difficulty of each exercise has to be roughly known. It is then possible to randomly distribute these exercises to different students in the group. It is also important to distribute these exercises to random students with different skill levels, e.g., different study programs to ensure that every exercise is evenly distributed between students of different skill levels. When grading the exam, the grades of each student is scaled based on the relative difficulty of the question given; thus, a student who received exercises with an overall lower score due to hard exercises would still receive a fair evaluation because it is taken into account.

Additionally, to utilize program tracing exercises in exam situations, a requirement is that the students do not have any way to run the code, as it would trivialize the question. At NTNU, the current exam system utilized, Inspera, only supports syntax highlighting, meaning the trace exercises generated by the prototype could be utilized. In the future, Inspera might end up with supporting code execution. In such a case, program tracing exercises would not make sense. A requirement for tracing to be feasible in this exam situation would be to force parts of the exam to be done without the ability to execute code.

6.2.2 Obligatory exercises

Another potential use is for obligatory exercises during the course that is required before taking the exam. These exercises usually test the students in the curriculum during the course, and the students' are often required to complete a certain amount of these exercises to take the exam. Compared to the exam exercises, the difficulty of these exercises does not need to be as equal. The difficulty problem could be overcome by giving each student enough questions to increase the probability of every student receiving approximately an equal distribution of exercises in regards to difficulty.

While the difference in the difficulty of each exercise might pose itself as a problem, it is also an opportunity. Adaptive e-learning systems are tailored to give every student exercises that fit their skill level. These systems provide students with exercises that fit their current skill level, and as their skills increase, so does the difficulty of the exercises. Such systems rely on having access to exercises of different difficulties. Utilizing such a system would prevent skilled students from receiving easy exercises and less skilled students from receiving too hard exercises. By giving students exercises that fit their skill level, they are also more likely to achieve Flow [43]. An important part of achieving Flow

is dependent on the difficulty of the task at hand. To achieve Flow, the student should be given a reachable goal, i.e., complete tasks within their skill level, but not too easily. If the student is presented with too easy exercises, they are likely to become bored, while presenting too hard exercises might lead to anxiety. An adaptive e-learning system can, therefore, help the student achieve Flow by providing the students with exercises that are challenging, but solvable. Having a system to distribute the exercises adaptively would also increase the usefulness of the exercises. In obligatory exercises, it could be possible to create a limit where the students' have to complete a given number of exercises tailored at their skill level to pass. Such an approach would ensure that every student has to complete a set of exercises to take the exam in the course, but would also provide each student with exercises that fit their skill level, potentially increasing their learning to a more substantial degree than exercises of static difficulty would.

A problem that arises for obligatory exercises is the cheating possibilities. These exercises are performed at home, outside of a controlled environment. As a result, they can be solved simply by executing the code and copying the output. It is, however, possible to prevent or reduce the likelihood of cheating. A possible technique is to display the exercise in a way that makes it impossible to copy, and thus requires the student to write out the whole code snippet to find the answer. It is also possible to utilize characters that are not accepted by the Python interpreter or use a combination of spaces and tabs, which would look just like runnable Python code, but would cause exceptions upon execution.

Another way of preventing cheating could be to include function calls to functions that are not directly included in the program tracing exercise. An example is a function called *number_to_text*, which takes in a number and returns its text without spaces, converting 402 to four hundred two. Running the code to solve such an exercise would require the student to create the function being called, which would need more work than solving the tracing exercise. A requirement for such an approach would be a clear and concise description of what the function does. Without a concise explanation, the exercise's difficulty might shift from the tracing itself to understanding what the function does.

6.2.3 Voluntary tests

Voluntary test exercises are the easiest type of questions to generate. These type of questions does not require the exercises to be marked with any difficulty level, making this area the easiest to generate exercises for. Knowing the difficulty of each exercise, however, is still important for self-examination questions. Specific e-learning systems rely on tailoring the exercises given based on the students' progression, providing the difficulty of each exercise is therefore required for them to be utilized appropriately in said system [44]. Another vital requirement to use exercises for self-examination is to ensure that the exercises generated from the same template are sufficiently different from each other. If the exercises become too similar, it could cause the student to memorize the answer or a specific process for extracting the answer. A result of such exercises could be reduced learning outcomes, as the student stops learning how the specific concepts work, but instead relies on remembering the answers.

6.3 Technical limitations

As presented in the Results section, there are many benefits to using the CFG approach for generating program tracing exercises. However, certain drawbacks were discovered during the prototype development. Some of the biggest technical limitations encountered during the development that could potentially become issues in the future are:

- Work required to implement the boilerplate system
- Implicit type system and randomized constructs
- Generating exercises for multiple languages

The subsections below explain and discuss the different limitations.

6.3.1 Required work

While there are several significant advantages of using a context-free grammar approach for generating exercises, there are also drawbacks. One of the most significant drawbacks that translates to much work for the writer of the generator is that the generator gets almost nothing for free from the target language. In practice, this means that if the template language wants to use a construct, function, or other features present in the target language, it has to make sure to support these features explicitly. Designing a good template language that allows for usage of concepts present in the target language and providing proper error messages and feedback to the user of the generator is, therefore, an enormous task. While defining these constructs in the template language gives the generator full control over every feature, basic features such as built-in functions might require an amount excessive work to be appropriately implemented. In an SCT generator, on the other hand, no error handling or validation is present, and all features of the generator stems from the features of the target language. Thus, creating a generator using the SCT approach requires substantially less effort at the cost of generator complexity. From the implementation of the prototype, it has become clear that the amount of work to implement constructs for generating randomized exercises in addition to adding simple features present in Python requires much work that could potentially be saved by using the SCT. On the other hand, the control given by explicitly defining every feature of the template language gives the generator a great ability to give feedback and error messages to the user of the generator, giving it a significant edge in usability.

Due to the boilerplate requirements of the template language, such as validating logic, types, and other constructs of the language, much time has to be spent on areas of the generator that is not directly linked to features for generating exercises. Additionally, the test performed after the last iteration showed that a lot of small bugs could break specific templates. From the test, it became clear that much work also has to be put in to validate all of the features in conjunction with each other, as one feature is could potentially break another. Thus, significant overhead is attached to implement interesting and novel features properly.

6.3.2 Type system and randomized constructs

One of the most significant limitations of the approach is the fact that it operates at compile-time and lacks information about certain things happening at run time. The first issue this caused was related to Pythons implicit type-system. The prototype template-language was designed to imitate Python as much as possible, making it easy to learn the language. As a result, the implemented grammar was designed to be type implicit (just like Python). Type implicit languages do not explicitly define the type of variables or parameters; instead, it is inferred from its definition. This sparked problems when trying to utilize different variable types to generate constructs such as composedStatements—generating these statements requires the generator to be 100% certain of the type of each variable used, since some have to be converted to integers before being compared in such an expression. An example would be a String that has to be converted to an int by taking its length. The problem with the implicit type system and composedStatements emerges from function calls. Since the parameters of function calls are not explicitly typed, they can be of any type. This means that if a function, named **fun** with one parameter, **a**, is called twice; one time where a is a number, then another time where a is a string, the generator cannot determine a single type for the input parameter a. If the function then adds this parameter to a list defined outside of the function, the generator no longer with 100% certainty what types the list contains. It is, therefore, clear that without explicit types, the generator struggles with utilizing all types of variables in its different constructs.

As mentioned in the code, one way to solve the mutation issue with the function parameter being added to the list, is by making an assumption. By assuming that a list only contains one type, the constructs of the generator can still be utilized. One of the most significant drawbacks of this assumption is that more responsibility is given to the template writer. If the template writer fails to ensure single types in a list with this assumption, the generator may create erroneous exercises that fail to run.

Another way to solve this issue would require a thorough re-write of the grammar used for the prototype. The template language could utilize explicit typing even though the target language, Python, uses implicit typing. By making all variable declarations and parameters require explicit typing, the issue described above could be avoided, since the generator would catch any errors with type mismatches. Even though Python does not utilize explicit types, forcing the template to utilize explicit types would make sense in terms of what is allowed in the generated exercises. While forcing a list only to allow elements of one type does not happen in Python, it does not make sense to utilize a list with elements of different types in the generator. As explained above, having lists containing multiple types would make many of the randomized constructs impossible to utilize without adding a lot of unnecessary and bloated logic to the exercises themselves, which in practice would ruin the exercises. If, for some reason, the template needs to support lists of multiple types, it is possible to utilize the **Any** type, which is accepted by most constructs. Utilizing this type would tell the generator that there are certain limitations to the variable, limiting the features of the constructs.

6.3.3 Generating exercises for multiple programming languages

As presented in the Section about the CFG generator, a problem with it is that it targets a specific language. During the development of the prototype, it became clear that creating a generator for multiple languages would require a large amount of work compared to a generator for a single language. The amount of work would, of course, depend on how significant the differences between the target languages are and what concepts the exercises should focus on.

An example would be generating exercises for Java and Python from the same grammar. To be able to do this, a few considerations have to be made. The first consideration is that Java is only object-oriented, while Python is Procedure oriented (can be object-oriented though). Generating exercises for these two languages would require the generator to create the boilerplate constructs around the exercise (such as the main function and its surrounding class) for Java, while Python could be generated as explained in this report. The second consideration is that Java has explicit typing. When comparing these two ways of deciding the type of a variable, explicit typing is more strict. In reality, this would mean that the grammar needs to have explicit typing to satisfy the Java requirements. The third consideration is related to what types of exercises one would utilize in courses that teach Java versus courses that teaches Python. Since Java is object-oriented, many courses utilize this language to teach object-oriented programming. Generating object-oriented exercises would be different from simple procedure-oriented exercises, resulting in little re-use of logic between the two target languages.

The example above shows that while it is possible to utilize a single generator & grammar to generate exercises for different languages, this approach might require much work. Additionally, different languages might expose different problems, meaning that the difference in exercise types might warrant utilizing two different grammars and generators to efficiently generate exercises that fit the course's need in mind.

Chapter 7

Conclusion and Future Work

Due to programming being more and more popular, CS1 courses are becoming more prominent. Consequently, professors, lecturers, and teaching assistants have to spend an increasing amount of time correcting exercises rather than helping the students. A way of combating this issue is to create templates that can be used to create exercises similar to the template in question. With the ability to both generate a broad set of exercises and auto-correct them, the time spent by those responsible for the course can be shifted from creating and correcting exercises to helping students. This thesis looks at an approach for creating templates for generating program tracing exercises, then evaluates the suitability of such an approach for CS1 courses.

Research question 1: *What solutions of auto-generating programming-exercises have already been proposed in research-literature?*

There were mainly two approaches discovered during the literature review, the Specification, Configuration & Template (SCT) and the context-free grammar (CFG) approaches. The SCT technique is based around having templates in the target language, and utilizing configurations and specifications for the template to substitute values in the template. The configurations define connections that are directly used in the templates. These connections are linked to values defined as specifications. Upon generation, the connection entries in the template is substituted for the values from the specification. Therefore, the SCT template approach can create unique exercises by randomizing the values placed in the template. Additionally, its implementation allows the templates to be written in the target language, meaning the generator would work for any programming language. The CFG approach is based on defining a context-free grammar that is parsed in and converted to exercises. The main idea behind the approach is to parse in the template language and substitute different sub-trees within the parse-tree of a template to randomize the output (each exercise). Such an approach gives the exercise generator much context of the template at hand, allowing for more advanced randomization of the template. The downside of using a context-free grammar is that the generator has to be programmed to output ex-

ercises in the target language, thus, to make it generate exercises for multiple languages require extra work.

Research question 2: *What is the best approach for generating a large amount of program tracing exercises?*

Through the project, a prototype was developed to determine the potential and limitations of generating exercises through a CFG approach. The study conducted showed that the CFG approach opens the door for generating interesting program tracing exercises for CS1 courses. By using a template language customized for the task at hand, the prototype gained insight into the exercise templates, allowing for more advanced exercise generation. On the other hand, developing a template language requires much work and is prone to errors. Through the final test, several small errors were found, diminishing the usefulness of the generator. While some templates used in the prototype might not lead to many thousands of distinct exercises, having a smaller amount of unique exercises is still useful.

Compared to the SCT approach for generating exercises, the CFG approach has a significant advantage because it uses its own template language that is parsed and analyzed, allowing for more nuanced and complex exercises. This advantage comes at the cost of a much larger workload to get the generator working, due to more boilerplate requirements. Additionally, the CFG approach generates exercises for a single language, making it less diverse than the SCT approach. Regardless of the drawbacks found during the project, the potential of the CFG generator outweighs the drawbacks, and compared to the SCT approach, it comes out ahead. By comparing the different approaches for generating exercises discovered during the literature review and looking at the project results, the CFG generator is the best approach to generator program tracing exercises.

Research question 3: *Which parts of CS1 courses would the generated exercises be most useful for?*

There are mainly three areas of a CS1 course where exercises generated by the prototype would be the most useful; exams, obligatory exercises, and self-examination. Exams are the most strict area and usually require a way of measuring the difficulty of each exercise generated. Scaling the grades for each question on the exam based on the overall performance is a feasible approach, but even then, having a rough estimate of the difficulty of each exercise is required.

On the other hand, obligatory exercises do not require each exercise to be equal in difficulty. The main problem in this use-case is deferring students from cheating by executing the code. This can be circumvented to a certain degree by ensuring it takes more time to run the code than by just solving it. There are several approaches to achieving this:

- Utilize signs in the code that are not accepted by the Python interpreter, resulting in exceptions
- Make it impossible to copy the code, forcing the student to write every line manually
- Include functions with obvious functionality that is not included in the exercise, forcing the student first to write the function to execute the code.

By utilizing one of these techniques, the generated exercises could be useful for reducing cheating in obligatory exercises.

The last area is self-testing, and generating exercises for this area can be looked upon as the easiest. Since self-testing does not affect the students' grades in the CS1 course, each exercise does not require a specific difficulty estimate. However, knowing the difficulty of each exercise would still be of great help when utilizing the exercises. Since some e-learning systems rely on providing students with exercises based on their knowledge and progression, they also require an estimate of the exercise difficulty. Despite this, using the exercises generated could still provide excellent learning outcomes by providing unique exercises for different concepts.

Research question 4: *Are there any significant technical limitations related to the approach chosen for generating exercises?*

The main drawback of the CFG technique is the amount of boilerplate work required to generate interesting exercises properly. The two main points posing limitations are:

- Implicit type system
- Generating exercises for multiple languages

Since the idea behind the CFG approach is to define a grammar to represents a template language, much work is required to ensure that the logic of the template is correct. To properly create randomized constructs and other features, a proper type system needs to be in place, requiring a substantial amount of work. Additionally, much work is required to allow the parse-tree traverses to check for logical errors. In general, it can be seen as creating a rather simple compiler for a new programming language. Much effort must be put into validating the logic, type, and other aspects of the template language, rather than directly focusing on features for generating exciting exercises. These features are also error-prone, meaning much time needs to be put into fixing bugs and testing different edge cases.

Finally, multiple target languages pose limitations to the approach. Since the CFG approach requires the developer to implement logic to generate exercises for a specific language, additional logic must be implemented to support more languages. The CFG technique, therefore, scales poorly when looking at more than one target language.

7.1 Future work

7.1.1 Multiple choice distractors

A natural expansion for the program tracing problems is to generate multiple-choice answers (distractors). The distractors could be of any value, but should ideally be very similar to the correct answer, either in terms of their actual value, or by reflecting common errors related to the question. There are multiple ways to approach this problem; one way could be to accumulate answers for a given question and pick the most common wrong answers to be used as distractors. Another way is by tweaking the exercise generated in a small way to generate an answer that could be mistaken for the correct one.

7.1.2 Parson's problems

Parson's problems would also be a natural way to expand the prototype. These problems consist of dividing a block of code into smaller fragments that must be ordered in the correct way to create the correct output. While distractors are common with Parson's problems, it has been shown that their impact might not be significant in any way. Thus, the main job of generating Parson's problems is to divide the generated exercises into sensible code fragments.

7.1.3 Evaluating difficulty

As mentioned during the discussion, an essential aspect of every exercise is their difficulty. Knowledge about each exercises' difficulty could improve their usability in the various areas of a CS1 course. As discussed, there are multiple ways of trying to determine the difficulty of an exercise. Due to the importance of knowing the difficulty of each exercise, it is certainly worth looking into approaches for classifying their difficulty.

7.1.4 New generating features

During the project, much work was put into making the template language and the underlying features of the generator. This work diverted resources from creating features to generate novel features to make the exercises better and more exciting. More work should be put into looking at how the features available in the prototype can be utilized to create new and exciting constructs.

7.1.5 Further testing

From the testing, some of the issues with the template language surfaced. The biggest issue found during the test was small errors and bugs within the language, preventing the generator from being fully utilized. Minimizing the number of errors and bugs is crucial, and further testing should, therefore, be conducted to identify bugs and issues. Additionally, more focus could be put on the end-users of the exercises generated. During the project, the evaluation was performed in conjunction with the professors who were going to use the system, but no testing was performed on students. Therefore, it would be highly efficient to test the generated exercises on students to evaluate the capabilities of the generator further. Different types of tests could also be done. One possibility is to test the exercises either in a real exam environment or an environment emulating an exam situation. Such a test would provide great insight into the usefulness of the exercises.

7.1.6 Exercise descriptions

One of the suggestions presented after testing was expanding the generator to create a textual description of each exercise generated. Having such a feature would allow the generator to create code-writing exercises, where the student are tasked with implementing code based on the textual description given. Correcting these exercises would, in return, require a comparison between the code of the exercise generated and the code created by the student. Such a comparison would not always work since there are usually different

ways of solving a problem. A more elaborate way of correcting each exercise would, therefore, have to be implemented to catch situations where the generated code does not match the student-written code, even though the students' code is correct.

7.1.7 Final thoughts

While this report mainly looks at whether the chosen approach for generating exercise would be suitable for program tracing exercises in CS1 courses, it would be interesting to look at it in the context of its usability in the courses. As discussed, adaptive e-learning systems could greatly benefit from having access to exercises of different difficulty. In the future, implementing the metrics presented by Kasto & Whalley [5] in the generator could provide an excellent basis for usage in adaptive e-learning systems. Utilizing the exercises with the given metrics in adaptive systems would then provide the basis for evaluating the exercises on a larger scale with students of different skill levels. As it currently stands, there are no features for storing and organizing the exercises generated. Making the generator more accessible would require some system to both generate and store the exercises in a sensible manner. By expanding the current prototype using the already implemented REST-API, one could implement the generator into an already existing e-learning system, or design a stand-alone system that handles both generation and exercise storage.

Bibliography

[1] G. Sindre, "Analyse av oppgaver og oppgavesjangre i en digital eksamen i innledende programmering," in *NOKOBIT - Norsk konferanse for organisasjoners bruk av informasjonsteknologi*, vol. 27, 2019.

[2] D. Radošević and I. Magdalenić, "Source code generator based on dynamic frames," *Journal of Information and Organizational Sciences*, vol. 35, no. 1, pp. 73–91, 2011.

[3] A. Ade-Ibijola, "Syntactic generation of practice novice programs in Python," in *Annual Conference of the Southern African Computer Lecturers' Association*. Springer, 2018, pp. 158–172.

[4] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design science in information systems research," *MIS quarterly*, vol. 28, no. 1, pp. 75–105, 2004.

[5] N. Kasto and J. Whalley, "Measuring the difficulty of code comprehension tasks using software metrics," in *Proceedings of the Fifteenth Australasian Computing Education Conference-Volume 136*, 2013, pp. 59–65.

[6] B. Özmen and A. Altun, "Undergraduate students' experiences in programming: difficulties and obstacles," *Turkish Online Journal of Qualitative Inquiry*, vol. 5, no. 3, pp. 1–27, 2014.

[7] D. Radošević, T. Orehovački, and Z. Stapić, "Automatic on-line generation of student's exercises in teaching programming," in *Radošević, D., Orehovački, T., Stapić, Z:" Automatic On-line Generation of Students Exercises in Teaching Programming", Central European Conference on Information and Intelligent Systems, CECIIS*, 2010.

[8] K. M. Rørnes, R. K. Runde, and S. M. Jensen, "Students' mental models of references in Python."

[9] E. Lahtinen, K. Ala-Mutka, and H.-M. Järvinen, "A study of the difficulties of novice programmers," *ACM SIGCSE Bulletin*, vol. 37, no. 3, pp. 14–18, 2005.

[10] G. Sindre and A. Chirumamilla, "E-exams versus paper exams: A comparative anal-

ysis of cheating-related security threats and countermeasures," *Presented at Norwegian Information Security Conference (NISK)*, Nov 2015.

[11] NTNU. (2019) TDT4110 - Information Technology, Introduction. [Online]. Available: https://www.ntnu.edu/studies/courses/TDT4110

[12] NTNU. (2019) TDT4109 - Information Technology, Introduction. [Online]. Available: https://www.ntnu.edu/studies/courses/TDT4109

[13] M. Lopez, J. Whalley, P. Robbins, and R. Lister, "Relationships between reading, tracing and writing skills in introductory programming," in *Proceedings of the fourth international workshop on computing education research*. ACM, 2008, pp. 101–112.

[14] A. Petersen, M. Craig, and D. Zingaro, "Reviewing CS1 exam question content," in *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '11. New York, NY, USA: ACM, 2011, pp. 631–636. [Online]. Available: http://doi.acm.org/10.1145/1953163.1953340

[15] A. Venables, G. Tan, and R. Lister, "A closer look at tracing, explaining and code writing skills in the novice programmer," in *Proceedings of the fifth international workshop on Computing education research workshop*, 2009, pp. 117–128.

[16] E. Soloway, "Learning to program= learning to construct mechanisms and explanations," *Communications of the ACM*, vol. 29, no. 9, pp. 850–858, 1986.

[17] A.-J. Lakanen, V. Lappalainen, and V. Isomöttönen, "Revisiting rainfall to explore exam questions and performance on CS1," in *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, 2015, pp. 40–49.

[18] U. Costantini, V. Lonati, and A. Morpurgo, "How plans occur in novices' programs: A method to evaluate program-writing skills," in *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, 2020, pp. 852–858.

[19] D. Romanich, "Generating program tracing exercises for introductory programming courses," Department of Computer Science, NTNU – Norwegian University of Science and Technology, Project report in TDT4501, Dec. 2019.

[20] R. Lister, B. Simon, E. Thompson, J. L. Whalley, and C. Prasad, "Not seeing the forest for the trees: novice programmers and the SOLO taxonomy," *ACM SIGCSE Bulletin*, vol. 38, no. 3, pp. 118–122, 2006.

[21] D. Zingaro, A. Petersen, and M. Craig, "Stepping up to integrative questions on CS1 exams," in *Proceedings of the 43rd ACM technical symposium on Computer Science Education*. ACM, 2012, pp. 253–258.

[22] B. Xie, G. L. Nelson, and A. J. Ko, "An explicit strategy to scaffold novice program tracing," in *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. ACM, 2018, pp. 344–349.

[23] K. Cunningham, S. Blanchard, B. Ericson, and M. Guzdial, "Using tracing and sketching to solve programming problems: Replicating and extending an analysis of

what students draw," in *Proceedings of the 2017 ACM Conference on International Computing Education Research*. ACM, 2017, pp. 164–172.

[24] G. Kearsley and B. Shneiderman, "Engagement theory: A framework for technology-based teaching and learning." *Educational technology*, vol. 38, no. 5, pp. 20–23, 1998.

[25] D. Parsons and P. Haden, "Parson's programming puzzles: a fun and effective learning tool for first programming courses," in *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*. Australian Computer Society, Inc., 2006, pp. 157–163.

[26] B. J. Ericson, L. E. Margulieux, and J. Rick, "Solving parsons problems versus fixing and writing code," in *Proceedings of the 17th Koli Calling International Conference on Computing Education Research*, 2017, pp. 20–29.

[27] P. Denny, A. Luxton-Reilly, and B. Simon, "Evaluating a new exam question: Parsons problems," in *Proceedings of the fourth international workshop on computing education research*. ACM, 2008, pp. 113–124.

[28] K. J. Harms, J. Chen, and C. L. Kelleher, "Distractors in parsons problems decrease learning efficiency for young novice programmers," in *Proceedings of the 2016 ACM Conference on International Computing Education Research*. ACM, 2016, pp. 241–250.

[29] J. J. Van Merriënboer, "Strategies for programming instruction in high school: Program completion vs. program generation," *Journal of educational computing research*, vol. 6, no. 3, pp. 265–285, 1990.

[30] C. Taylor, D. Zingaro, L. Porter, K. C. Webb, C. B. Lee, and M. Clancy, "Computer science concept inventories: past and future," *Computer Science Education*, vol. 24, no. 4, pp. 253–276, 2014.

[31] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[32] Facebook. (2019) React. [Online]. Available: https://reactjs.org/

[33] Pivotal. (2019) Spring boot. [Online]. Available: https://spring.io/projects/spring-boot

[34] S. T. March and G. F. Smith, "Design and natural science research on information technology," *Decision support systems*, vol. 15, no. 4, pp. 251–266, 1995.

[35] A. Tjora, *Kvalitative forskningsmetoder*. Gyldendal, 2017, kvalitative forskningsmetoder 3. utgave ISBN: 978-82-05-50096-9.

[36] M. Glinz, "On non-functional requirements," in *15th IEEE International Requirements Engineering Conference (RE 2007)*. IEEE, 2007, pp. 21–26.

[37] D. Chinn, M. de Raadt, A. Philpott, J. Sheard, M.-J. Laakso, D. D'Souza, J. Skene, A. Carbone, T. Clear, R. Lister *et al.*, "Introductory programming: examining the exams," in *Proceedings of the Fourteenth Australasian Computing Education Conference-Volume 123*, 2012, pp. 61–70.

[38] R. Parasuraman and V. Riley, "Humans and automation: Use, misuse, disuse, abuse," *Human factors*, vol. 39, no. 2, pp. 230–253, 1997.

[39] J. Buckley and C. Exton, "Bloom's taxonomy: A framework for assessing programmers' knowledge of software systems," in *11th IEEE International Workshop on Program Comprehension, 2003.* IEEE, 2003, pp. 165–174.

[40] M. Hillier, "The very idea of e-exams: student (pre) conceptions." Australasian Society for Computers in Learning in Tertiary Education, 2014.

[41] G. K. Gill and C. F. Kemerer, "Cyclomatic complexity density and software maintenance productivity," *IEEE transactions on software engineering*, vol. 17, no. 12, p. 1284, 1991.

[42] R. J. Fendler and J. M. Godbey, "Cheaters should never win: Eliminating the benefits of cheating," *Journal of Academic Ethics*, vol. 14, no. 1, pp. 71–85, 2016.

[43] M. Csikszentmihalyi, *Flow: The psychology of happiness*. Random House, 2013.

[44] L. Kolås, A. Staupe, A. Sterbini, and M. Temperini, *QUIS requirement specification for a next generation e-learning system*. Technical Report, TISIP, Trondheim, 2007.

Appendix

H Grammar used in the artifact

The grammar is how the grammar is defined in ANTLR4. It is also accessible here. Note that the grammar is expressed as

`<production_name> : <another_production> | <another_production1>;`

Where the colon (:) equals to the right arrow ($\longrightarrow$) of the productions previously mentioned in this report.

```
grammar Autogenerator;

grammar Autogenerator;

parse
    : block EOF
    ;


block
    : (stat | function_definition) *
    ;

stat
    : assignment        #regularStatement
    | expr              #regularStatement
    | if_stat           #regularStatement
    | for_stat          #regularStatement
    | random_for_stat   #customStatement
    | print             #regularStatement
    | return_stat       #regularStatement
    ;

assignment
    : VAR_NAME ('[' expr ']')* ASSIGN expr
    ;

if_stat
    : IF condition_block (elif_stat)* (else_stat)?
    ;

elif_stat
```

```
33      : ELIF condition_block
34      ;
35
36  else_stat
37      : ELSE stat_block
38      ;
39
40  return_stat
41      : RETURN expr
42      ;
43
44  condition_block
45      : expr stat_block
46      ;
47
48  stat_block
49      : OBRACE block CBRACE
50      | stat
51      ;
52
53  for_stat
54      : FOR VAR_NAME IN RANGE OPAR expr (',' expr)? (',' expr)? CPAR
        stat_block       #forInRange
55      | FOR VAR_NAME IN expr stat_block
                 #forInIterator
56      | WHILE expr stat_block
                 #whileLoop
57      ;
58
59  random_for_stat
60      :
61      FOR VAR_NAME stat_block
62      ;
63
64  print
65      : PRINT OPAR expr CPAR
66      ;
67
68  function_definition
69      : FUNCTION VAR_NAME OPAR parameter_list CPAR stat_block
70      ;
71
72
73  function_call
74      : VAR_NAME OPAR parameter_expression_list CPAR
75      ;
76
77  parameter_list
78      : (VAR_NAME (',' VAR_NAME)*)?
79      ;
80
81  parameter_expression_list
82      : (expr (',' expr)*)?
83      ;
84
85  dictionary_entry
86      : expr COL expr
```

```
87      ;
88
89  // TODO - Rewrite this to have production for every function.
90  variable_function
91      : VAR_NAME '[' (expr)? ':' (expr)? ((':') expr?)? ']'
          #stringSliceFunction
92      | VAR_NAME DOT RANDOM_SLICE OPAR CPAR
          #randomStringSlice
93      ;
94
95
96
97
98  expr
99   : function_call                                                    #
       function_callExpr
100  |  MINUS expr                                                       #
       unaryMinusExpr
101  | NOT expr                                                         #notExpr
               // DONE
102  | expr op=(MULT | DIV | MOD | PLUS | MINUS) expr                   #
       mathematicalExpr // DONE
103  | expr op=(LTEQ | GTEQ | LT | GT) expr                             #
       relationalExpr   //DONE
104  | expr op=RAND_ARITHM_OP expr                                      #
       randomArithmOpExpr
105  | expr op=RAND_REL_OP expr                                         #
       randomRelOpExpr
106  | expr op=(EQ | NEQ) expr                                          #
       equalityExpr   // DONE
107  | expr IN expr                                                     #inExpr
             // DONE
108  | expr AND expr                                                    #andExpr
             // DONE
109  | expr OR expr                                                     #orExpr
             // DONE
110  | atom                                                             #atomExpr
             // DONE
111  | RAND_INT OPAR expr ',' expr CPAR                                 #
       randIntExpr
112  | RAND_INT_List OPAR expr ',' expr ',' expr CPAR                   #
       randIntArrayExpr
113  | LEN OPAR expr CPAR                                               #lenExpr
114  | STR OPAR expr CPAR                                               #strExpr
115  | COMPOSED_EXPRESSION OPAR VAR_NAME* (',' VAR_NAME)* CPAR          #
       composedExpression
116  | '[' (expr ( ',' expr )*)* ']'                                    #listExpr
117  | '{' (dictionary_entry ( ',' dictionary_entry )*)* '}'            #dictExpr
118  | '{' expr ( ',' expr)* '}'                                        #setExpr
119  | '(' (expr ( ',' expr)*)* ')'                                     #tupleExpr
120  | expr DOT VAR_NAME OPAR parameter_expression_list CPAR             #
       dynamicVariableFunction
121  | variable_function                                                #
       varFunctionExpr
122  ;
123
124 atom
```

```
 : OPAR expr CPAR        #parExpr
 | (INT | FLOAT)         #numberAtom
 | (TRUE | FALSE)        #booleanAtom
 | VAR_NAME '[' expr ']'#varNameAtomAccessIndex
 | VAR_NAME              #varNameAtom
 | STRING                #stringAtom
 | NONE                  #noneAtom
 ;

OR : 'or';
AND : 'and';
EQ : '==';
NEQ : '!=';
GT : '>';
LT : '<';
GTEQ : '>=';
LTEQ : '<=';
PLUS : '+';
MINUS : '-';
MULT : '*';
DIV : '/';
MOD : '%';
POW : '^';
NOT : 'not';

SCOL : ';';
COL : ':';
ASSIGN : '=';
OPAR : '(';
CPAR : ')';
OBRACE : '{';
CBRACE : '}';

TRUE : 'True';
FALSE : 'False';
NONE : 'None';
IF : 'if';
ELSE : 'else';
ELIF : 'elif';
FOR : 'for';
IN : 'in';
RANGE : 'range';
WHILE : 'while';
PRINT : 'print';

DOT : '.';
RANDOM_SLICE : 'randomSlice';

RAND_INT : 'randInt';
RAND_INT_List : 'randIntList';
LEN : 'len';
STR : 'str';

RAND_ARITHM_OP : 'randArithmOp';
RAND_REL_OP : 'randRelOp';
COMPOSED_EXPRESSION : 'composedStatement';

```

```
RETURN : 'return';
FUNCTION : 'def';

VAR_NAME
 : [a-zA-Z_] [a-zA-Z_0-9]*
 ;

PYTHON_FUN_NAME
 : [a-zA-Z_] [a-zA-Z_0-9]*
 ;

INT
 : [0-9]+
 ;

FLOAT
 : [0-9]+ '.' [0-9]*
 | '.' [0-9]+
 ;

STRING
 : '"' (~["\r\n] | '""')* '"'
 ;
COMMENT
 : '#' ~[\r\n]* -> skip
 ;
SPACE
 : [ \t\r\n] -> skip
 ;

OTHER
 : .
 ;
```

Listing 7.1: The ANTL4 grammar representing the auto-generator template language.

I Relevant productions from the pre-project

$$\texttt{<parse>} \longrightarrow \texttt{<block>}\lambda \tag{7.1}$$
$$\texttt{<block>} \longrightarrow \texttt{<statement>}* \tag{7.2}$$
$$\texttt{<statement>} \longrightarrow \texttt{<assign>}\ | \tag{7.3}$$
$$\longrightarrow \texttt{<if\_statement>}\ | \tag{7.4}$$
$$\longrightarrow \texttt{<for\_statement>}\ | \tag{7.5}$$
$$\longrightarrow \texttt{<random\_for\_statement>}\ | \tag{7.6}$$
$$\longrightarrow \texttt{<print>} \tag{7.7}$$
$$\tag{7.8}$$

$$\texttt{<assign>} \longrightarrow \texttt{<var\_name><assign><expression>}\ | \tag{7.9}$$
$$\longrightarrow \texttt{<var\_name><assign><assign\_list>} \tag{7.10}$$
$$\texttt{<var\_name>} \longrightarrow \texttt{[a-zA-Z\_][a-zA-Z\_0-9]}* \tag{7.11}$$

$$\texttt{<if\_stmt>} \longrightarrow \texttt{'if' <cond\_block> (<elif\_stmt>)}* \texttt{ (<else\_stmt>)?} \tag{7.12}$$
$$\texttt{<elif\_statement>} \longrightarrow \texttt{'elif' <condition\_block>} \tag{7.13}$$
$$\texttt{<else\_statement>} \longrightarrow \texttt{'else' <statement\_block>} \tag{7.14}$$

$$\texttt{<condition\_block>} \longrightarrow \texttt{<expression> <statement\_block>} \tag{7.15}$$
$$\texttt{<statement\_block>} \longrightarrow \texttt{'\{' <block> '\}'} \tag{7.16}$$
$$\longrightarrow \texttt{<statement>} \tag{7.17}$$

$$\texttt{<print>} \longrightarrow \texttt{'print' '(' <expr> ')'} \tag{7.18}$$

J Image of the prototype used during evaluation of the project

As mentioned in chapter 2.5.3 a simple web-application was built to make testing of the prototype easier. Below is a set of images of the web-application used during evaluation. Figure 7.1 and 7.2 show the initial design used during the two first iterations. Figure 7.3 illustrate the new version used during the final test. The code input section was upgraded to be more programming friendly.

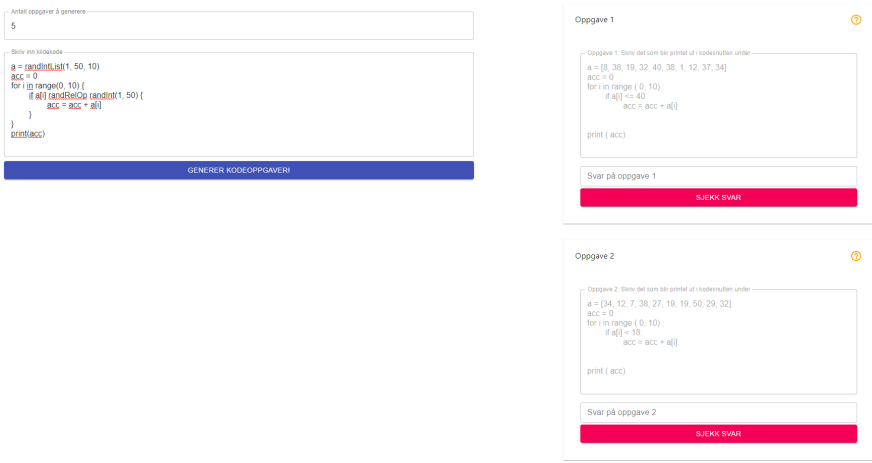


Figure 7.1: The website implemented to use for the interviews.

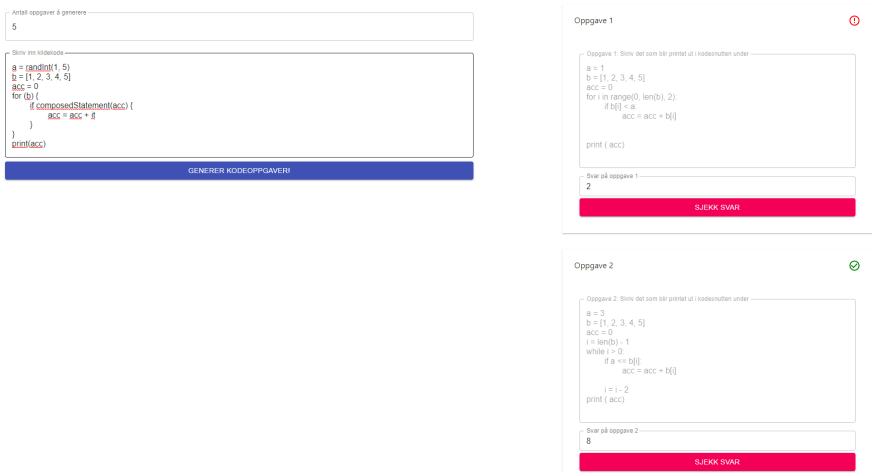


Figure 7.2: The website implemented to use for the interviews.

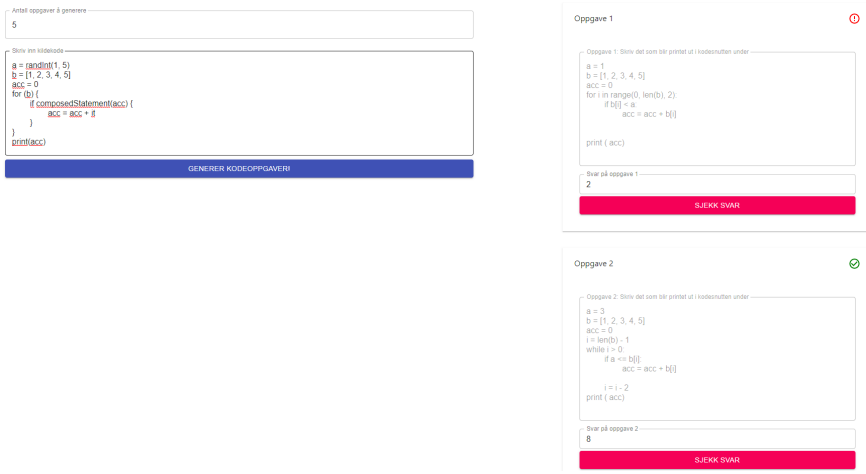


Figure 7.3: An upgraded version of the website used during the final test.

K Template examples

K.1 Templates used to measure duplicates

```
a = [1,2,3,4,5]
sum = 0
b = 6
c = 7
d = 8
for a {
  if composedStatement(a, sum) {
    sum = sum + it
  }
}
print(sum)
```

Listing 7.2: Template #1 used for measuring duplicates.

```
a = [5] * randInt(4, 8)
sum = 0
for a {
  sum = sum + it
}
print(sum)
```

Listing 7.3: Template #2 used for measuring duplicates.

```
words = ["Hello", "Wonderful", "Python", "Exercise", "World", "!"]

def myst(a) {
  b = ""
  for a {
    b = b + it.randomSlice() + " "
  }
  return b
}

print(myst(words))
```

Listing 7.4: Template #3 used for measuring duplicates.

```
table = {}
elements = randIntList(1, 10, 20)

for elements {
  if it in table {
    table[it] = table[it] + 1\n" +
  } else {
    table[it] = 1
  }
}
```

```
11
12 print(table[randInt(1, 10)])
```

Listing 7.5: Template #4 used for measuring duplicates.

K.2 Templates used in evaluations

```
1  # 1
2  a = [1,2,3,4,5]
3  sum = 0
4  b = 6
5  c = 7
6  d = 8
7  for a {
8    if composedStatement(a, sum) {
9      sum = sum + it
10   }
11 }
12 print(sum)
```

Listing 7.6: Template used in the first round of evaluation.

```
1 a = [5] * randInt(4, 8)
2 sum = 0
3 for a {
4   sum = sum + it
5 }
6 print(sum)
```

Listing 7.7: Template used in the first round of evaluation.

```
1 a = [[1,2,3],[4,5,6],[7,8,9]]
2 sum = 0
3 for a {
4   for it {
5     sum = sum + it
6   }
7 }
8
9 print(sum)
```

Listing 7.8: Template used in the first round of evaluation.

```
1 a = [[randInt(2, 8)] * randInt(4, 8)] * 3
2 sum = 0
3 for a {
4   for it {
5     sum = sum + it
6   }
7 }
```

```
print(sum)
```

Listing 7.9: Template used in the first round of evaluation.

```
list = ["This", "is", "a", "short", "list", "of", "strings"]
count = 0
for list {
  count = count + len(it.randomSlice())
}
print(count)
```

Listing 7.10: Template used in the second round of evaluation.

```
def fun(l, a, b) {
  l1 = []
  for l {
    l1.append(it * a + b)
  }
  return l1
}


print(fun(randIntList(2,5,5), randInt(1, 3), randInt(5, 15)))
```

Listing 7.11: Template used in the second round of evaluation.

```
words = ["Hello", "Wonderful", "Python", "Exercise", "World", "!"]

def myst(a) {
  b = ""
  for a {
    b = b + it.randomSlice() + " "
  }
  return b
}

print(myst(words))
```

Listing 7.12: Template used in the second round of evaluation.

```
table = {}
elements = randIntList(1, 10, 20)

for elements {
  if it in table {
    table[it] = table[it] + 1
  } else {
    table[it] = 1
  }
```

```
}

print(table[randInt(1, 10)])
```

Listing 7.13: Template used in the second round of evaluation.

```
def c(e) {
  a = {}
  for e{
    if it in a {
      a[it] = a[it] + 1
    } else {
      a[it] = 1
    }
  }
  r = randInt(1, 10)
  if r in a {
    return a[r]
  } else {
    return 0
  }
}


print(c(randIntList(1, 10, 20)))
```

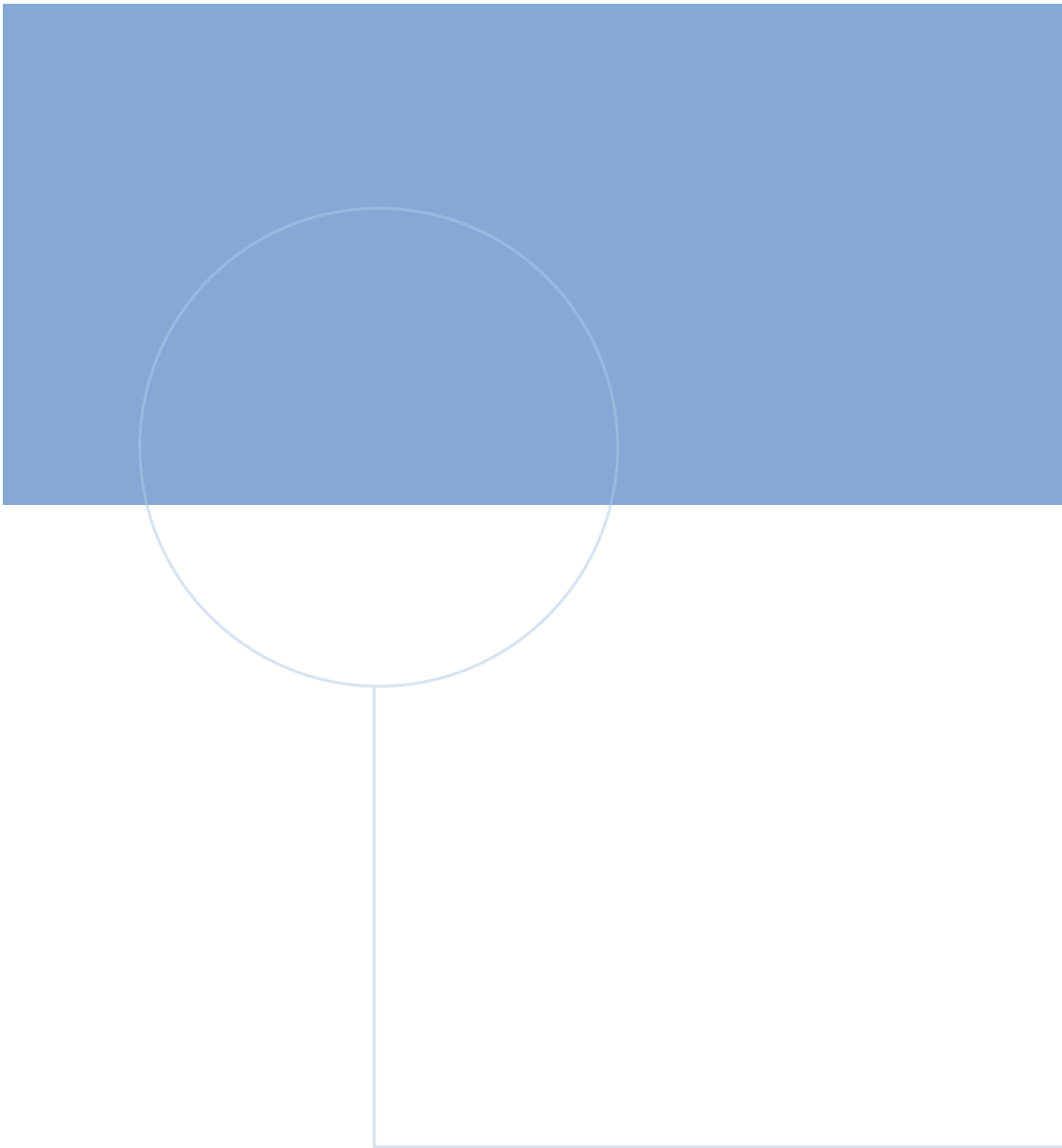
Listing 7.14: Template used in the second round of evaluation.

```

a = ["This", "is", "a", "list", "of", "strings"]

def m(b) {
  for i in range(0, len(a)) {
    s = a[i]
    a[i] = s.randomSlice()
  }
}

m(a)
print(a)
```

Listing 7.15: Template used in the second round of evaluation.

L Documentation

The documentation for the generator can be found at:
https://github.com/danielromanich/Autogenerator.



NTNU

Kunnskap for en bedre verden