Spark - A programming language for beginners

- Design, Definition and Implementation of Programming Languages -

Project Report

Group 6

Aalborg University
Sofware

This report is written in LaTeX Overleaf. Zotero integration with Overleaf is used for the sources and citations in the report. The program is written in IntelliJ IDEA with the programming language Java, and shared using GitHub. Google Docs is used for sharing notes, while Google Forms is used to make the survey. Diagrams are either found from other sources and cited, or made by the group members in Microsoft Excel or Draw.io based on data from other sources cited.

Software
Aalborg University
http://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Spark - A programming language for beginners

Theme:
Design, Definition, and Implementation of Programming Languages

Project Period:
Spring Semester 2023

Project Group:
6

Participant(s):
Goran Kirovski
Karina Botes
Magne Frank Dinesen
Mikkel Skovlund
Pernille Knudsen
Tobias Friese

Supervisor(s):
Simon Aagaard Pedersen

Copies: 8

Page Numbers: 79

Date of Completion:
May 25, 2023

Abstract:

The purpose of this project is to create a programming language and a compiler that can compile the created language into another language based on a problem area. The problem area chosen for this project is problems or difficulties that beginners learning to code might encounter. Data on the target group of "teenagers and young adults learning how to code" was collected through a questionnaire and multiple interviews.

The gathered data gave insight into the target group and problem area. Based on that data, the group started to form their own programming language targeting beginners in coding. The language created in this project is called Spark and it proposes a solution to the chosen problem statement. Spark uses minimal keywords and has easy-to-follow grammar rules which make the learning progress easier for beginners. Through the collection of data, the group found that beginners might have an easier learning curve when working with visual elements and based on this the group decided to implement the possibility to create visual elements with the Spark language. This was possible through the implementation of the "Processing" library in the compiler's target language Java.

Contents

1 Introduction

For this project, the group had the opportunity to focus on looking at problems linked to beginners learning how to code, as the group has personal experience with these problems. In this aspect, the group first started focusing on the popular learning platform Scratch to get inspiration. However, quickly expanded their attention from Scratch to coding a language aimed at beginners in general as research showed more languages common for beginners.

This project covers how the group has formulated a problem statement based on interviews, a questionnaire, and other research around problem areas linked to beginners learning how to code. Based on the problem statement the group then formed product requirements and developed a new programming language, that can be used as a beginning step in learning how to code.

To collect data from the target group,teenagers and young adults who are in the process of learning programming, the group has conducted interviews at Danish gymnasiums and sent out a questionnaire. An interview was also made with a volunteer from the organization Coding Pirates to investigate a slightly younger group, to see how they learn to code and what challenges they have. The collected data gave insight into what is useful and what is confusing and contributes to the difficulty in languages when learning how to code.

Based on the collected data from State of the Art, the questionnaire, and interviews, the group came to the conclusion that it is difficult for most people when they start learning how to code to learn the syntax and figure out the logic behind the code.

The finalized product consists of a viable prototype of the programming language Spark and the Spark compiler. The Spark language is designed to make learning to code easier with few keywords and with a focus on the possible use of visual elements as a learning tool. The Spark compiler has been designed to compile code written in Spark into the target language Java. The design choices and implementation of the Spark language and compiler are presented throughout this report.

Following the development of the compiler and language, tests were conducted to ensure that a few to no bugs were present in the code. Further tests can be used to increase user experience and coherence. The state of the product compared to the aim of solving the problem statement, is discussed and concluded in the Discussion and Conclusion section.

2 Methodology

When planning a larger project such as this one, it is important for the project's success that the group has an organized plan and structure throughout the project. The planning of this project has been done through the tools Backcasting and Gantt diagram which will be presented and discussed below [1].

2.1 Backcasting

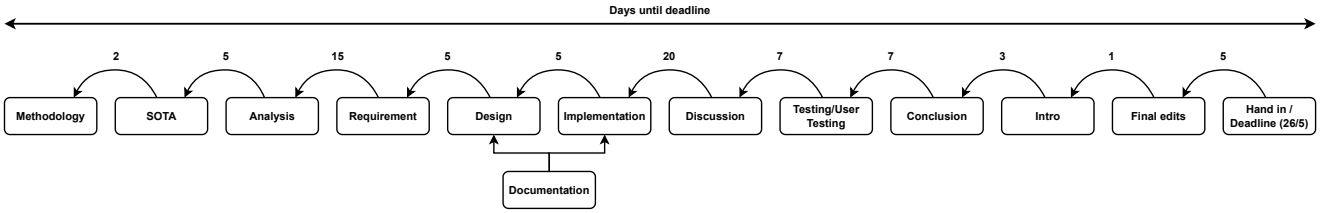


Figure 1: Backcasting

Figure 1 illustrates the time allocation for each phase of the project, and to gain a realistic overview of the phase duration, it has been done in a back-casting manner, meaning that the process of phase time allocation starts from the final phase and carries on backward [1]. This ensures that each phase receives enough resources to be thoroughly processed during the duration of the project, whereas the phase time allocation is based on the group's opinion, expectations, and previous experiences with time planning [1]. The figure consists of the total project duration of 66 days (exclusively weekdays), all the phases in chronological order with their respective amount of days allocated, and the total phase duration of 76 days. Additionally the phase "Documentation" runs perpendicularly to other phases, therefore it has not directly received time allocation in the figure, and lastly, the total phase duration surpasses the project duration, since some phases overlap one another, which can be seen in the Gantt diagram in figure 2.

2.2 Gantt diagram

The Gantt Diagram is a tool taught at Aalborg University in the course PBL [1]. The Gantt diagram is great at visualizing and keeping track of each phase of the project, the Gantt tool however specializes in keeping track of both the start and end date of each phase in the project. The phases can also be seen according to the weeks their start and end dates are placed in, visualizing the length and time consumption of the phases for the user.

However, a lot of unpredicted matters can happen within a project, especially in a larger project such as this and the group might experience becoming uncoordinated with the expected Gantt diagram. Therefore the group had an additional Gantt diagram, being the Actual diagram for the project process which was updated weekly throughout the project to keep track of the changes and their outcome. Then upon reflecting on the project, the group can see more clearly where problems might have occurred and how the group adapted to them.

Gantt Diagram - Expected

Tasks			Time Period				Week number														
			Overlap	Days	Start Date	End Date	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Startup		Metode		1	16/02/2023	17/02/2023															
SOTA				7	17/02/2023	28/02/2023															
Analysis			2	15	24/02/2023	17/03/2023															
Requirements			2	5	15/03/2023	22/03/2023															
Design	Dokumentation			5	22/03/2023	29/03/2023															
Implementation				20	29/03/2023	26/04/2023															
Testing/User Testing			1	7	25/04/2023	04/05/2023															
Discussion			2	7	02/05/2023	11/05/2023															
Conclusion				3	11/05/2023	16/05/2023															
Introduction				1	16/05/2023	17/05/2023															
Final adjustments				5	17/05/2023	24/05/2023															
Hand-in				1	24/05/2023	25/05/2023															
Deadline			-1	0	26/05/2023	26/05/2023															

Figure 2: Gantt diagram - Expected

The Expected Gantt diagram the group made for the project can be seen in figure 2. The Actual Gantt diagram was updated throughout the project and is a visual representation of how the progress of the project actually went. The Actual Gantt diagram can be seen in figure 3. When looking at the Expected Gantt diagram in comparison to the Actual Gantt diagram, it can be seen that some of the sections were moved or delayed. This can be first seen occurring at the "Requirements" section in week number 13. The section took a week longer than the group expected, which was followed by a huge amount of delay in the "Design" and "Implementation" sections, this delay happened due to the group encountering complications with understanding and learning how a compiler works and is properly implemented. This delay resulted in the work progress following becoming more asynchronous and the work progress of the following sections overlapped each other more than expected.

Gantt Diagram - Actual

Tasks			Time Period				Week number													
			Overlap	Days	Start Date	End Date	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Startup		Metode		1	16/02/2023	17/02/2023															
SOTA				7	17/02/2023	28/02/2023															
Analysis			2	15	24/02/2023	17/03/2023															
Requirements			2	10	15/03/2023	29/03/2023															
Design	Dokumentation			15	29/03/2023	19/04/2023															
Implementation			10	20	05/04/2023	03/05/2023															
Testing/User Testing			7	11	24/04/2023	09/05/2023															
Discussion			6	11	01/05/2023	16/05/2023															
Conclusion			1	7	15/05/2023	24/05/2023															
Introduction			1	2	23/05/2023	25/05/2023															
Final adjustments			1	1	24/05/2023	25/05/2023															
Hand-in			1	1	24/05/2023	25/05/2023															
Deadline				1	25/05/2023	26/05/2023															

Figure 3: Gantt diagram - Actual

2.3 Trello as a Kanban board

A Kanban board was made to continuously keep track of the group's current work progress. A Kanban board is a tool used to classify tasks into the boards' different categories: "To Do", "Doing", "Ready for review" and "Done" [2]. Upon creation, tasks are assigned to the "To Do" category and then moved into the different categories as they are being worked on, additionally, members can be assigned to different tasks, therefore the state and group members' responsibility for all the different tasks can easily be tracked, this creates an easy overview for the group of the project process.

The group has created a Kanban board using one of the more popular tools - Trello [3]. Trello is a website specialized in creating such Kanban boards as described above, and the group's Kanban board can be seen below in figure 4.

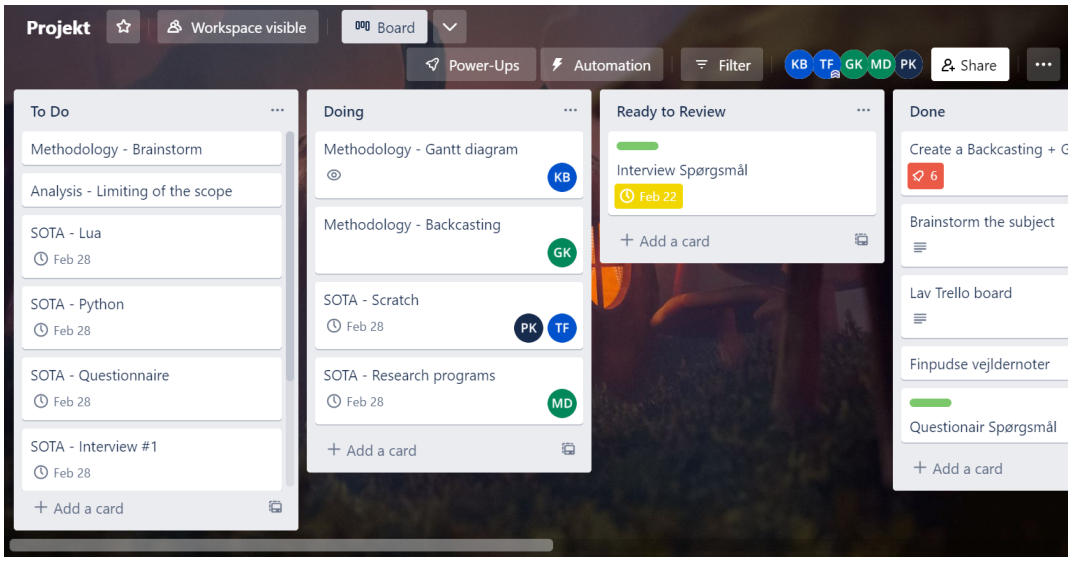


Figure 4: Overview of Trello as a Kanban Board

2.4 Mindmap

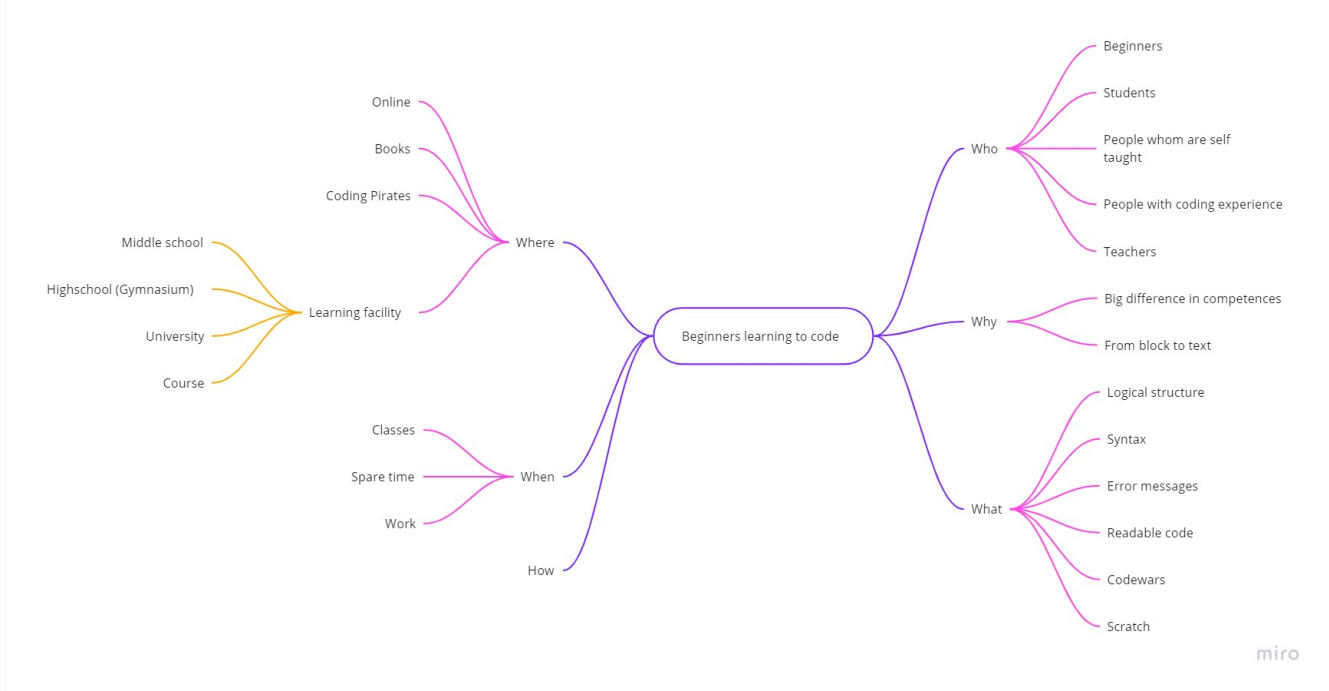


Figure 5: Mind-map over the problem area

Mind-map is a method, in which the group members engage in a brainstorming process - through discussion to analyze the topic at hand, with the purpose of generating and sorting thoughts and ideas to specify an accomplishable problem field. The current mind-map is based on the six questions "Where", "When", "How", "Who", "Why", and "What" [1], with the focal point being the topic "Beginners learning to code", illustrated in figure 5.

3 Configuration management

In this section, the tools used for developing the compiler, configuration management, and keeping track of version control will be presented and their selection will be discussed. The selection and use of these tools are important to understand as to gain a further understanding of the group's development process of the prototype of this project. The tools presented and discussed in this section are *Github* and *IntelliJ*.

The decisions of how to manage and use the compiler produced in this project will also be discussed, these decisions and considerations will be explained and confirmed why they were chosen. This can be read in this section under the **Management of the Spark compiler** part.

Github
A part of configuration management is keeping track and documenting changes and versions of the product [4]. Throughout this project, the group has used the GitHub tool to manage the project and document all changes made throughout the project. The group has used GitHub in previous projects and therefore the continuous use of GitHub in this project seemed obvious.

GitHub has been used in this project to give all the group members equal access to the project, this is possible through GitHub's opportunities for collaboration [5]. It has also been used to document version controls and changes made by the different group members, these version controls are always accessible and can be given a title and description of what changes have been made [5].

IntelliJ
The compiler has been written in the language Java using the integrated development environment *IntelliJ IDEA* [6]. The group decided to use IntelliJ due to their previous experience with the tool and it is one of the leading tools today to write Java code [6]. The group has previously found IntelliJ's method of keeping track of the project's folders, files, and libraries to be very organized and easy to keep track of and to keep a good overview. IntelliJ also has the option for multiple users to code live together on different computers, this is possible through IntelliJ's collaboration possibility [6], the group has found this tool very useful throughout the development of their code.

Management of the Spark compiler
The Spark compiler is built to compile code written in the language Spark to the language Java. The compiler has been designed to have few interactions required by the user, it only needs a text file placed in the input folder and then the user needs to run the compiler. The compiler will then automatically know where to find the input file, if it is placed correctly, then parse and translate the given input. Then the compiler will translate and print the translated code into an output file, but first, it will check if there is an already existing Java output file, if not it will create one.

So the only thing the user would have to do is place their input file as a text file into the Input folder and then run the compiler. If the compiler fails it will give the user an error message, which is designed to inform and help them solve the problem.

4 Finding the problem area

To gain more data on the problem area at hand, concerning beginners learning to program, the group decided to conduct a questionnaire and multiple interviews. The questionnaire, was targeted towards the project's target group which is beginner programmers, and the interviews were targeted towards the main influence on the target groups learning to code, namely experts in the field of teaching programming. This allows the group to gain insights on both sides of the coin, one side is learning to code, and the other is teaching to code, with the possibility to see what works for both sides.

The following section will go over the preparation, conduction, and results of the questionnaire and the interview respectively, and lastly, the findings will be discussed with the intention to further scope the problem area.

Throughout the collection of data phase, the group contacted the volunteer group called Coding Pirates, which is focused on teaching programming and computer science to kids and teenagers between the ages of 7 to 17 years old. The group deemed this organization as highly relevant for this project and therefore contacted them for further information and the possibility of sending them a questionnaire and meeting for an interview. The results will be discussed further in the following sections.

4.1 Questionnaire

A questionnaire or a survey is a tool to gather quantitative data [7]. The group decided to prepare and conduct a survey to gather quantitative data from the target group of this project.

This section describes the purpose, preparation, and conduction of the survey performed in the context of this project. It is followed by showcasing and describing the results gathered from the survey, additionally, the survey has been conducted in Danish, but due to the characteristics of the rapport being English, the results have been translated.

4.1.1 The purpose and preparation of the questionnaire

The purpose of the survey has been to gain an extensive understanding of the problem area of the project, as a result creating an opportunity to determine an angle worth focusing on, with an achievable size that has piqued interest in the group's eyes. The survey does not exclusively focus on a single point but focuses more on general knowledge of the problem area, therefore the survey consisted of many multiple-choice and open-minded questions. Both block-based and text-based programming languages are explored, and due to the group's experience of being introduced to Scratch as one of their first programming languages, a sizable portion of the survey regards Scratch, and transitioning from Scratch to a text-based language.

Based on the chosen problem area, it was decided that the survey should be aimed at students with at least a little experience, in the process of learning programming. The group wanted the target group to be more focused on the participants' skill level rather than their age. The age range of the survey is from the age of 13 to 23.

The participants of the survey were primarily gymnasium students who have programming in school, the group members sent the survey to a couple of gymnasium teachers, who sent the survey to their students. However, the group also sent the survey to some additional sources known to answer with sincere intentions and that fit into the problem area and can relate to problems linked with beginning to learn how to code, such as fellow students at Aalborg University and some friends who fit into the target group of beginner programmers.

The group contacted the organization Coding Pirate which is a volunteer organization. Unfortunately, the group did not have the opportunity to get answers from the children from Coding Pirates due to complications with getting approval from all the children's guardians.

The survey was made as a digital Google survey and shared with the participants through a link. The survey was designed in such a way that some specific answers gave access to other relevant questions. An example of this use is *Question 5. Do you have knowledge of Scratch or have used it?*, if the participants answer yes to this question, they will get asked an additional question, in this case *Question 7. How useful did you find what you learned from Scratch in the topic of coding.* The decision to only give access to question 7 by answering yes to question 5 was made to keep the relevance of the answers high, it would not make sense to ask question 7 to participants who do

not know of Scratch.

4.1.2 The questionnaire results

Background information

The first section of the survey contains some more basic questions about the participants, these were created to gather knowledge and insight into who the participants are since this might influence their answers throughout the survey.

Question 1. Have you programmed before?
The first question is created to confirm the participants have experience with coding prior to taking the survey. If the participant answers no to this question the participant will get removed from the survey since they are then not relevant.

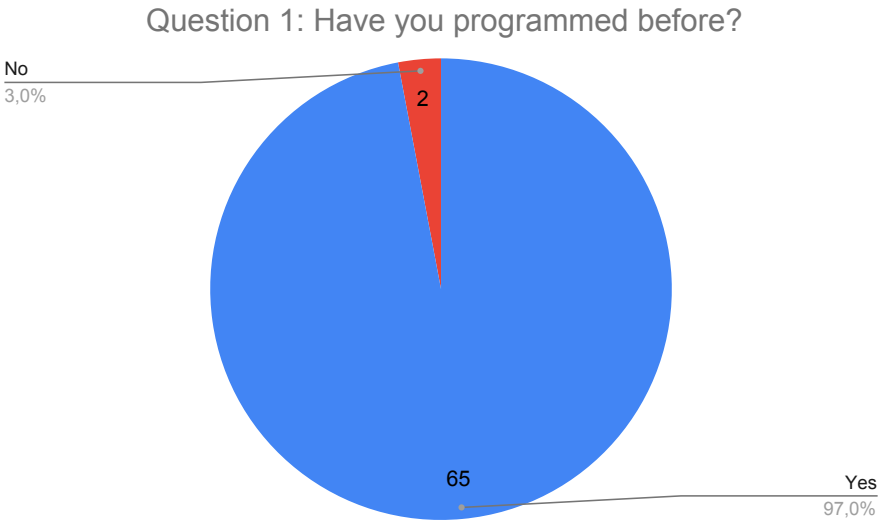


Figure 6: Results from question 1 in the survey

Based on this question it can be concluded that there were 65 participants in the survey that all had programmed prior. The two participants who answered no were removed from the survey and could not participate further.

Question 2. How old are you?
This question is created to gather empirical data on the participants who answered the survey. This gives a better idea of who the participants are.

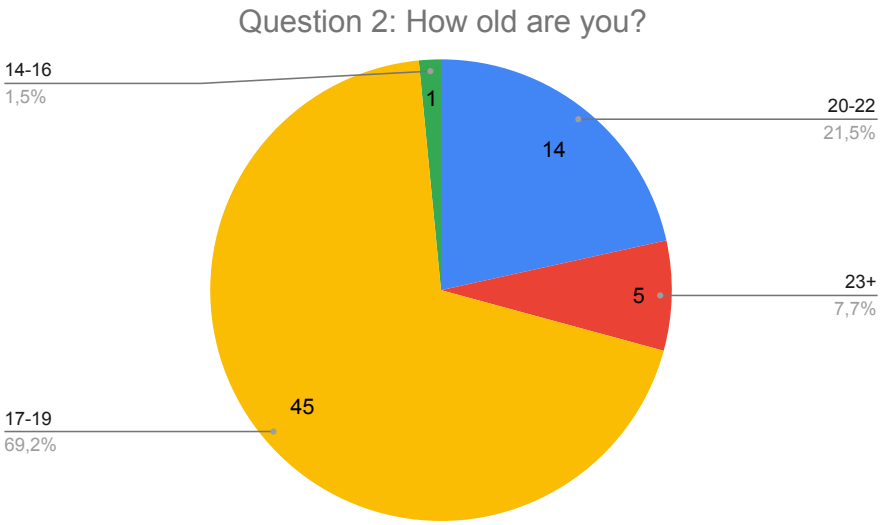


Figure 7: Results from question 2 in the survey

Based on this question it can be concluded the largest quantity of participants falls in the age of 17-19 years, followed by the second largest quantity being in the age of 20-22.

Question 3. How many years of programming experience do you have?
This question was created with the same intention as questions 1 and 2. To get empirical data and a better idea of who the participants are.

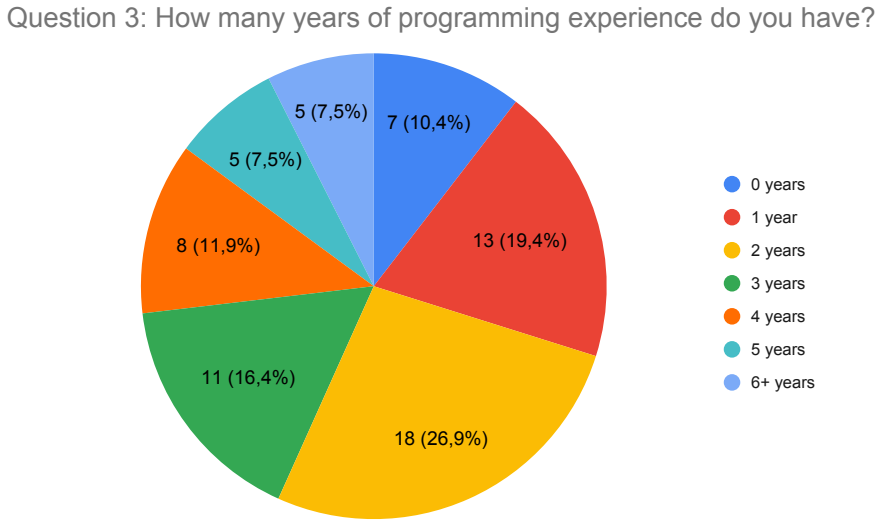


Figure 8: Results from question 3 in the survey

A wide variety of answers were collected to this question. The lowest amount of experience was 0 years with 10,4% of the answers, while the highest amount of experience was 5 years with 7,5% of the answers. Most answered 2 years with 26,9% followed by 1 year with 19,4% and 3 years with 16,4%. While a few of the participants did have either a lot or almost no coding experience, most had around 2 years of coding experience.

Question 4. At what age did you start learning to program?
This question gives insight into the participant's age when starting to understand code, this might influence how

they perceive and understand code now.

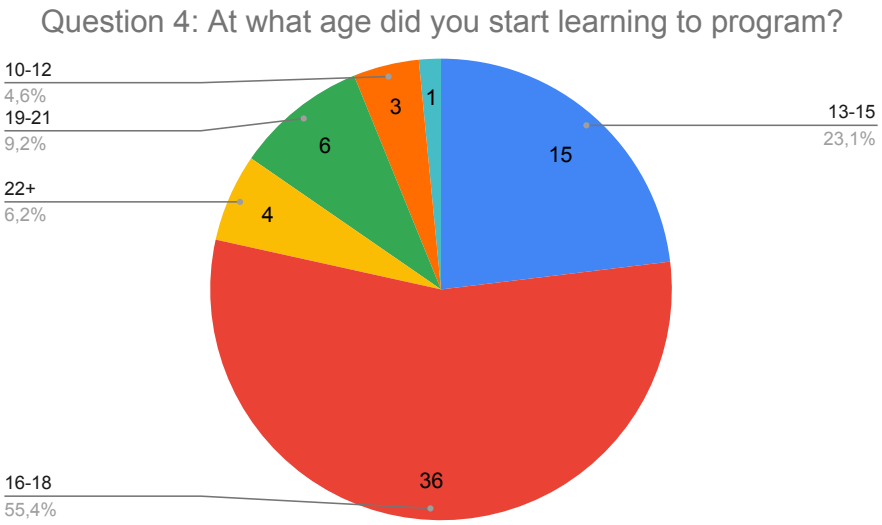


Figure 9: Results from question 4 in the survey

Based on these results it can be seen that the majority of 55,4% of the participants started learning how to code when they were the age of 16-18. Following that, 23,1% started learning how to code when they were between the age of 13-15 years old. The minority and rest of the participants started learning how to code when they were the ages of 10-12 (4,6%), 19-21 (9,2%), 22+ (6,2%).

Scratch and languages

Question 5. Have you used or do you know about Scratch?
This question has multiple purposes, firstly it gives insight into how large a quantity of the target group, that knows or has used scratch. Secondly, The participants, who answer yes to Scratch being their first language, get access to some extra Scratch-related questions only relevant to people who have knowledge of Scratch, this question is one of those question.

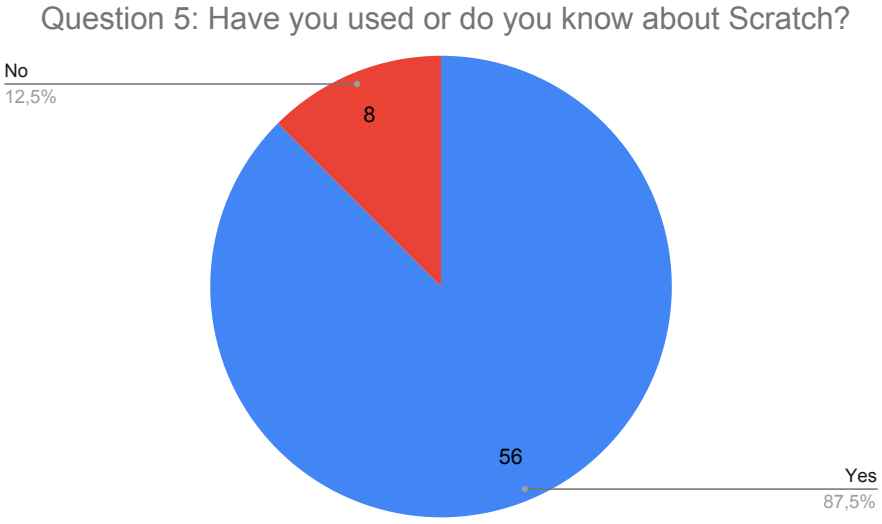


Figure 10: Results from question 5 in the survey

Based on the results from this question, it can be concluded that the majority of participants of 87,5% either know of or have used Scratch.

Question 6. What was the first programming language you learned?
This question was created to get inspiration and insight on which languages a complete beginner might choose to start learning to code through, this could help the group and give them the inspiration to look at these languages and understand what makes them easy to use.

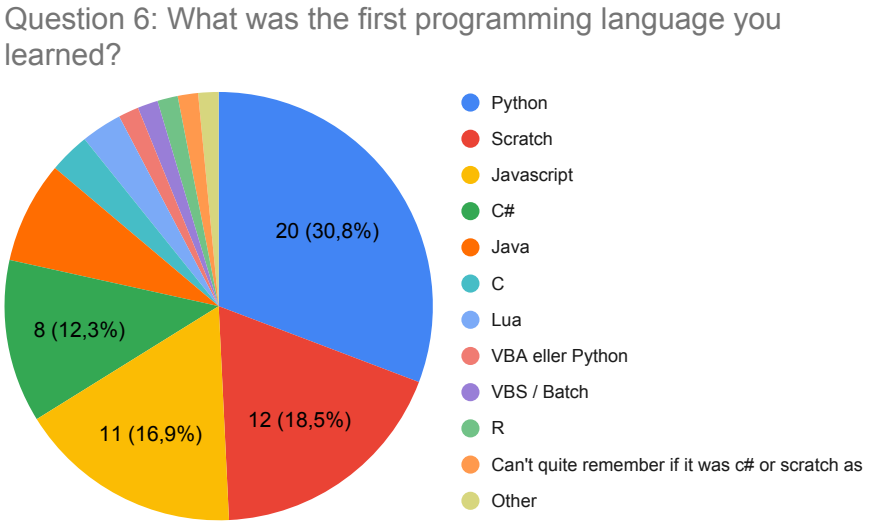


Figure 11: Results from question 6 in the survey

The participants had the option to write their own answers which makes the diagram look a bit clustered. However, it can be concluded that the majority of the participants' first programming language was Python, the majority being 30,8% of the participants.

It is, however, worth noticing that there are larger quantities of the participants who marked their first language to be Scratch (18,5%), JavaScript (16,9%), and C# (12,3%).

Smaller quantities of the participants marked their first language to be Java, C, and Lua, these are also worth taking a look at but the main focus should be on the languages with the largest quantities.

Question 7. How useful did you find what you learned from Scratch in the topic of coding?
This question was created to get insight into what the participants, who started learning with Scratch, think of Scratch and what their experience with learning through Scratch was. The participants were given 5 optional choices based on a Likert scale [8] to describe their experience: Very useful, useful, neutral, not particularly useful, and not useful at all.

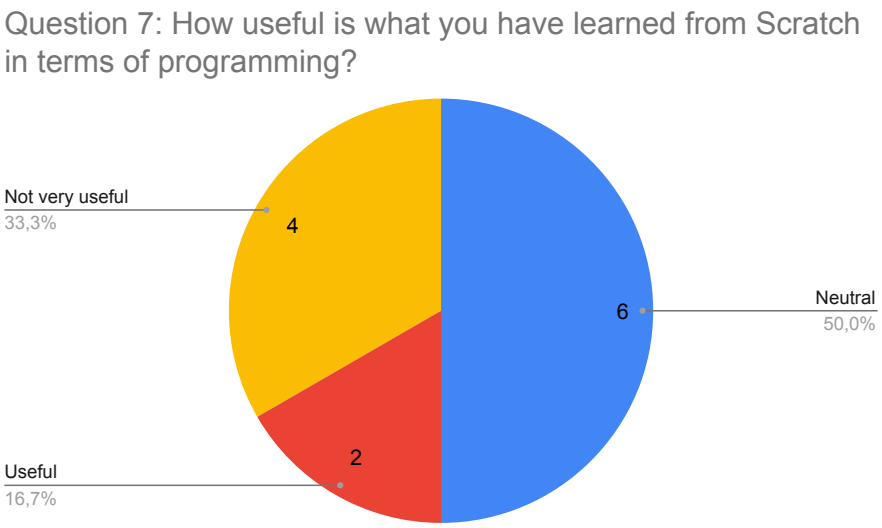


Figure 12: Results from question 7 in the survey

The results from this question give insight into what the participants actually think of Scratch. Overall the majority of 50% of the participants find Scratch as a learning tool "neutral". 16,7% find it useful, however, 33,3% of the participants find Scratch not particularly useful. Even though there are more participants who find Scratch neutral it is worth focusing on that every third of participants did not find Scratch useful for their own reasons. When looking at the results, 33,3% of the participants is a somewhat large amount of the participants and that confirms that Scratch might not be the best tool for this particular target group.

Question 8. Have you programmed in a text-based language? If yes, which language?
This question was created to sort the participants into two groups if necessary, the participants who answered yes to this gained access to the following question 9.

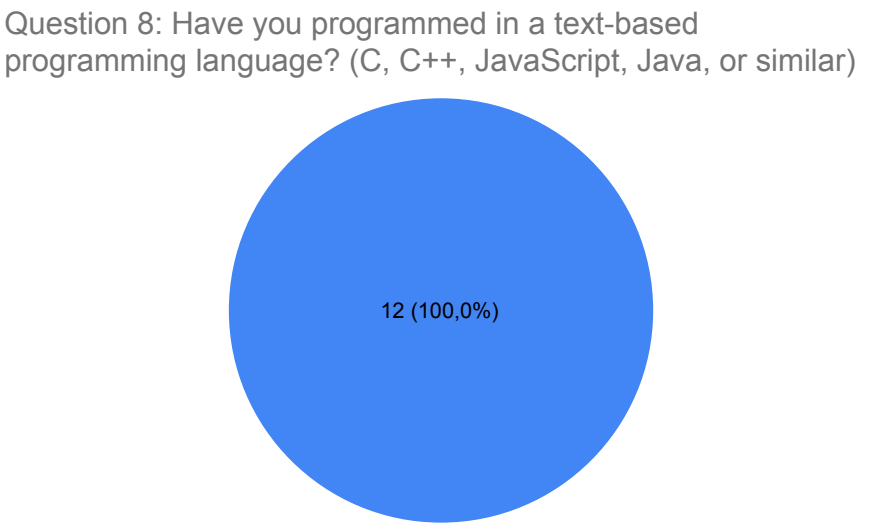


Figure 13: Results from question 8 in the survey

As can be seen, all 100% of the participants who reached this question, answered yes and got access to question 9.

Question 9. How did you find the transition between Scratch and text-based programming language?

This question gathered insight into how the participants, who both had experience with Scratch and text-based coding, experienced the transition between the two. This question will be a part of confirming that the transition might be an issue and that a better solution could exist.

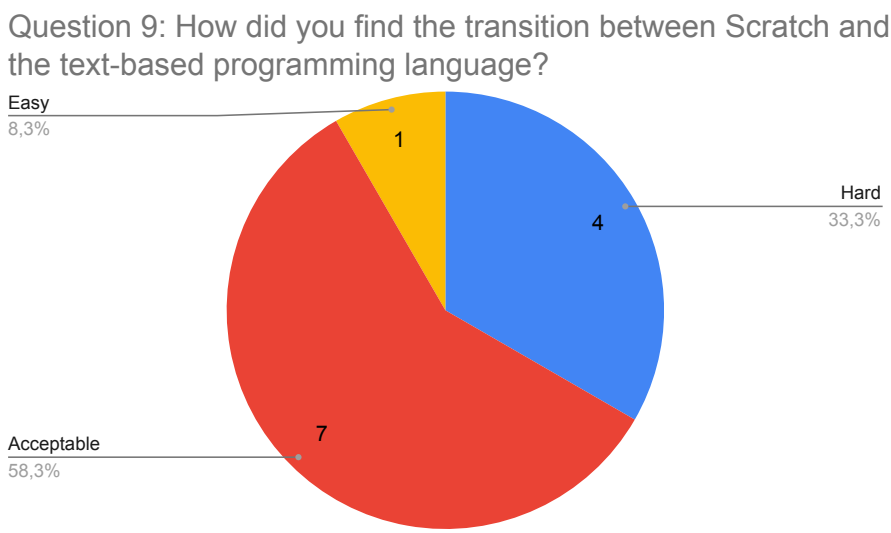


Figure 14: Results from question 9 in the survey

Here it can be seen that 58,3% found the transition acceptable, while 8,3% found the transition easy and 33,3% found the transition difficult. Even though the majority of participants found the transition acceptable or easy it is worth noting that a third of the participants found the transition difficult, that number is high especially when Scratch is so commonly used as a first language.

Getting insight into the participants understanding and use of coding?

Question 10. What problems did you experience when you first had to learn/understand programming?
This question was created to gather insight and data on what beginners may struggle with when starting to code. The participants had free choice to write which problems they might have encountered when first starting.

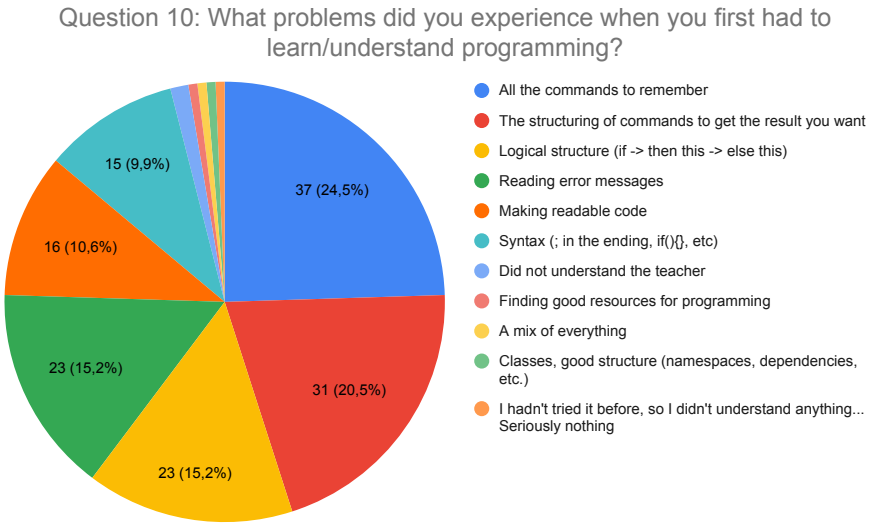


Figure 15: Results from question 10 in the survey

Based on the result from the question seen above in figure 15, it can be concluded that the majority of the participants taking up 24,5% encounter "All the commands to remember". When looking at the figure it can also be seen that larger quantities of the participants had struggles with "The structuring of commands to get the result you want" (20,5%), "Logical structure" (15,2%), "Reading error messages" (15.2%), and "Syntax" (9.9%).

It is worth looking into what could improve and help with these struggles that beginner programmers are experiencing.

There are however some of the participant's answers such as "Making readable code", "Did not understand the teacher", and "I hadn't tried it before so I didn't understand anything..." that can not be a focus of this project due to it being too complex, not relevant, and/or out of the projects time frame to solve.

Question 11. Were there any things that you could use from other subjects/subjects/interests that helped with some aspects of programming?
Through this question inspiration, insight and ideas could be gathered on what beginners find helpful with understanding coding when first beginning

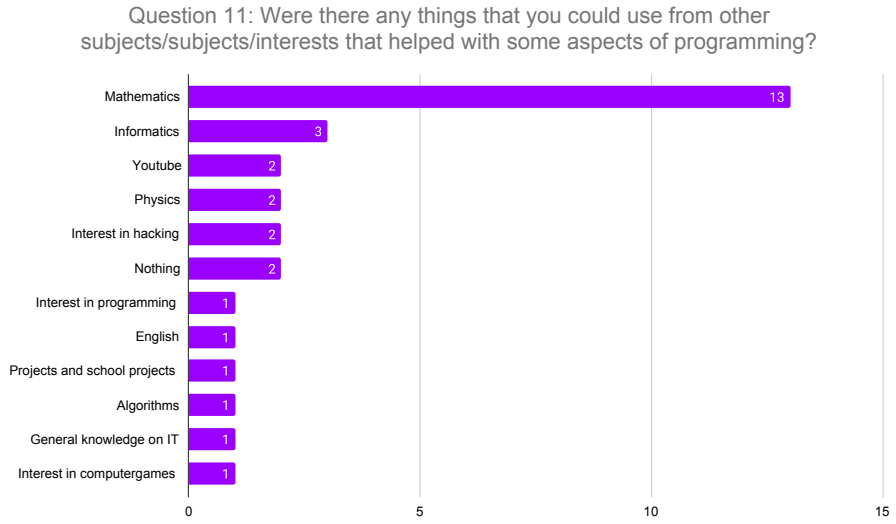


Figure 16: Results from question 11 in the survey

It can be seen in figure 16 that the majority of participants answered that "Mathematics" was the most helpful and relevant course/class when learning coding. The second most marked course/class is "Informatics" which is a class taught at HTX gymnasium that teaches the basics of computers, coding, and the basics of IT. "Informatics" is closely followed by "Physics", "Interest in hacking" and "Youtube". Youtube is a great platform to find tutorials or clear confusion. Mathematics and coding often go together in a lot of aspects, it might therefore be of best interest for the group to examine how the aspects of Mathematics or Physics could be implemented in a language to make them easier to use and understand for beginners.

Question 12. Which tools or features in the different coding languages did you particularly like?
This question was created to gather ideas of which features beginners like in programming languages and which might be a great idea to focus on.

- Error messages
- info-boxes when the mouse hovers over a command from IntelliJ
- Lambda methods
- Pointers
- Good visualization
- Scratch with visualization for easy understanding
- Python made it easy to write something new
- Github copilot
- No type restriction
- The structure and logic needs to be easy to understand by looking
- Java's references and documentation
- The structure and use of libraries
- The simplicity and quantity of libraries/documentation in Python
- Generally the graphical tools in Processing

Figure 17: Results from question 12 in the survey

For this question, the survey allowed the participants to write their own answers since it would be hard for us to list different features from different languages. These answers should be considered when constructing the project's own language since they are features wanted by users.

Question 13. Which coding language(s) do you primarily use today?
This final question was created to gather information on which languages the participants use today, this can be compared to their starting languages and the progress and differences can be analyzed.

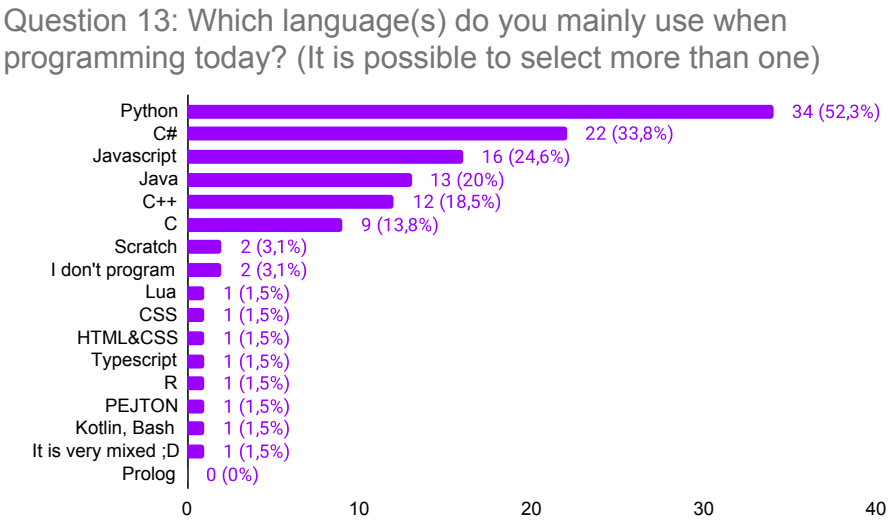


Figure 18: Results from question 13 in the survey

The results seen in figure 18 showcase the languages the participants primarily use today. It can be seen that the majority use Python (52,3%), C# (33,8%), JavaScript (20%), C++ (18,5%), and so on.

It is worth noting that the majority of the languages marked in this question are multi-paradigm languages. A pattern can be seen that some multi-paradigm languages have in commonality imperative programming or object-

oriented programming, and in the top rankings we can see the object-oriented language Java, and the imperative language C, therefore it can be speculated that these paradigms are preferred.

4.1.3 Conclusion of the questionnaire

The questionnaire had 65 participants that fit the criteria, most of the participants are between the ages of 17-19 (69,2%) followed by 20-22 (21,5%), having programming experience for 2 years (26,9%) followed by 1 year (19,4%), and have started programming at the ages of 16-18 (55,4%) and 13-15 (23,1%), therefore it is evident that the participants are introduced and/or learn programming either during gymnasium or shortly before they start gymnasium.

The languages that the participants first learned were Python (30.8%), Scratch (18.5%), JavaScript (16,9%), C# (12,3%). A multiple-choice question reveals that the most used programming languages today amongst the participants are, Python (52,3%), C# (33,8%), JavaScript (24,6%), Java (20%). A pattern can be seen that both the start and end languages are primarily multi-paradigm, and is an aspect worth considering. Languages that have a uniqueness separating them one from another will be looked into in a later section.

Regarding block-based programming languages, more specifically Scratch, it was found out that even though majority of the participants have knowledge about Scratch, only a small portion of 12 (18,5%) have used it. Furthermore, only 4 (33,3%) participants who had used Scratch thought of it as "Not very useful", and again only 4 (33,3%) found the transition towards text-based language "Hard". It can be seen that not only a relatively small amount of participants have used Scratch, but also the majority had a neutral or positive experience transitioning to text-based languages, therefore the group deemed this problem angle insufficient, and decided to diverge from it.

The participants had the opportunity to point out features they liked and disliked, the features that left a positive remark are, for example, "Error messages", "Good visualization", and "No type restriction", and features that left a negative remark are "All the commands to remember", "The structuring of commands to get the result you want", and "Logical structures". On that account the following needs to be considered, the relevant liked features should be implemented, and a solution should be developed for the disliked features.

A final point to be made is that due to time constraints, the data is very specific, because it is gathered from a few specific locations, instead of being widespread, by gathering data from a variety of locations. Gathering a smaller quantity of data is not ideal, but this trade-off is not necessarily bad, as the survey was directed to trustworthy sources that will do its diligence, and therefore the data will be of greater quality and gathered faster.

4.2 Interview

The method of conducting semi-structured interviews has been chosen by the group, as a way to collect qualitative data via researching experts in a specific field. This method allows the interviewers (the group) to construct a basic script, ensuring that the same topic is covered in each interview, though the less strict nature of the script, allows for a more open-ended, and more flexible conversation with the interviewees, opening up the opportunity of enlightenment in different aspects of the topic, that is yet to be explored.

4.2.1 Interview preparation work

For an overall smoother interview and analysis process, the project group was divided into two, three-member groups, each responsible for two of the four interviews, allowing the project group to conduct more interviews in a shorter time period, thus maximizing time efficiency, whilst maintaining a high interview quality.

An interview contract was constructed, seen in *Appendix A*, issued to the interviewees at the start of the interview, thus ensuring the group has the right to capture an audio recording of the interview, and the right to use information gathered from the interviews. The role distribution during the interviews is as follows, one member takes the leading role of the interview and holds a dialog with the interviewee, whilst the two remaining group members take notes, and when needed they supplement the discussion with additional questions or information. During the concluding part of the interview, the interviewee is presented with a small exercise regarding syntax, with the intention of gaining insights on preferred teaching syntax that provides the best learning experience, the structure of the exercise paper can be seen in *Appendix B*, and the interviewee results can be seen in *Appendix C*.

The prepared interview script is followed through completely, but the interview is open to flow in any direction, and the questions of the interview script will be explored in table 1, where the questions are divided into research questions which are the reason behind the questions and the actual interview questions. Upon conducting all interviews the groups will analyze the gathered interview data, subsequently, the main points from each interview will be extracted, so the project group can compare and contrast the different interviews' points of view, and on that basis draw a conclusion.

Research Questions	Interview Questions
Introduction.	• Begin with an introduction about the group and the motivation behind the interview.
Formalities.	• Fill out the contract and ensure that we can record the conversation.
Facts/background knowledge about the person we are interviewing.	• What is your name and position? • For how long have you been teaching programming?
Facts about the students.	• Who do you teach programming? • What levels of programming do you teach?
What is the teacher's experience as a teacher of programming?	• How is your experience teaching programming? • What languages and tools do you use to teach programming on all levels? • Do you know Scratch or other block coding programs?
How do you introduce programming to beginners?	• What is the first language you use to teach people who have never tried programming? • What is a problem you often see done by someone new to programming? • What is the easiest thing to understand for someone new to programming?
How does the teaching take place when you teach programming?	• Is help often needed in your lessons? • What tools are used if problems or confusion arise? • What languages do you use in teaching? • What is your impression of the student's meaning and use of (Language X)?
Get the teacher's views on what shortcomings there are in a language such as Scratch and suggestions that we can use for the development of our language.	• What do you think the advantages and disadvantages of (Language X) are in relation to teaching? • How do you facilitate the transition between (Language X) and the more classical advanced languages? • Imagine that a completely new language was created which is developed to teach children/young people how to program. •What do you think this language should consist of?
Closure	

Table 1: A table over the interview script

4.2.2 The results

The first interview arranged by the group was done at Hillerød Teknisk Gymnasium, where lector Christian Reinholdt participated in answering the prepared interview questions. Christian Reinholdt is teaching Programming for all three years of high school. The students have Mathematics on the A level and programming on the B level, and Christian Reinholdt thinks it is an advantage that the students have Mathematics on a high level since it teaches the students logical thinking. They learn some machine learning where mathematics is particularly useful. Some of the students have already coded a bit on a hobby level before starting at the Gymnasium, but he has experienced that it is not always an advantage that they have coded prior to studying it. He says this because some students have acquired some bad habits and may not have learned about syntax rules. And therefore may not have an overall overview of the code and are not able to make a structured plan for it beforehand.

Christian Reinholdt prefers to start out with Python as the first coding language, as it has a simple syntax. Besides this, it is open source and people are very helpful in various forums when it comes to Python. When the students have figured out how Python works, they are quicker to understand how to program in C and C++, according to

Christian Reinholdt. One of the more tricky parts in the transition from Python to C or C++ is learning about pointers, and understanding that they need to be in control of the memory. As well as to use access modifiers on their variables. Generally, he likes the syntax from Python such as forced indentation and no type restriction.

The students learn to make object-oriented programming, including making objects, UML Diagrams (Unified Modeling Language diagram), Class Diagrams, and State Diagrams. After they have learned object-oriented programming in Python they move on to imperative programming in C.

Christian Reinholdt sees it as a problem that many people believe that coding is a very geeky subject and therefore abstain from it, whereas he sees it more as a creative and innovative subject. Some problems that often occur when students have to learn to program are when the language is case-sensitive, to avoid syntactical errors such as remembering to use semicolons and curly brackets, and to write more readable code. Another problem is understanding error messages. The students tend to google the answers to the messages, because of a lack of patience, instead of trying to understand them, which in the long term takes more time.

Christian Reinholdt dislikes programs such as Scratch, as it is not used in the industry and cannot be used in a professional sense, and therefore only has value as a learning tool. It contributes to the student's motivation that they can use the language in "real life" and some students can even get a student job through experience with this language. Based on this he does not see Scratch as very useful when teaching gymnasium-level programming. Besides this, he does not think that it is easier to teach them Scratch compared to Python. However, he sometimes uses App Inventor by MIT which is reminiscent of an extended version of Scratch. He thinks that it is more useful because you can make apps for Android and the level is a little higher than Scratch as, for example, you are able to create functions.

Generally speaking, Christian Reinholdt was very enthusiastic about Python when teaching beginners. He likes that statements end with a new line, forced indentation, no pointers, and the use of object-oriented programming.

The second interview was held at KMG (Københavns mediegymnasium), conducted with the interviewee Louise Høpfner, with the position of leader in the digital department, and previously having experience as an instructor in the subject's Programming and Physics. Louise Høpfner used to teach Programming courses on C-level for beginners and thus points out that due to time limitations of the course, the purpose of the course is for the students to learn the process of solving a problem with programming, and therefore the language choice is less relevant since not all students will continue programming.

For more student engagement, JavaScript p5.js , which is also showcased in section 5.3, was chosen since it is graphical and any code changes the student does, can be seen visually in real time granting instant feedback. Additionally, if Louise Høpfner would undertake a class of students in the study field of mathematics or physics, Python would be the programming language of choice. Louise Høpfner gives insights about the downsides with p5.js for the students, such as the no-type restriction, which does not teach the students about types, which is especially problematic when the code does not work as intended or errors occur, as well as error handling not being that exceptional, as it does not show many errors messages, and the few it shows are not very specific or useful.

The biggest challenge in teaching is the students' level difference, some students have programming experience and others don't. Also, the learning curve of each individual student is different, therefore the key factor in teaching is finding the right level to teach amongst the student's contrast in experience. The most common difficulties students tend to encounter are, no knowlegde of how to start or finish coding a program, debugging errors, understanding statements (with the exception of if-statements), the expectation of their code to be flawlessly written on the first try, and for it to work as expected on execution. Therefore the students need a helping hand to guide them during the whole programming process.

Louise Høpfner explains that she has little personal experience with block programming, and the only language she knows is mBlock [9], though she never taught students any block programming, she further elaborates that block programming could be useful for the less experienced individuals if used, especially in learning statements. Louise Høpfner mentions that the students would benefit more if they based their code on the concept of objects, thus preferring object-oriented programming over the other paradigms. Louise Høpfner seemed somewhat neutral on type restriction, stating that having it would result in an extra abstraction for the students but they would gain more knowledge. On the other hand, not having it would help the students use more energy on other programming concepts.

The third interview was with a programming teacher Daniel P. Withenstien from Københavns mediegymnasium who teaches at both C and B levels. While the skills the students should learn are predetermined in Daniel P. Withenstiens course, the preexisting knowledge among the students varies greatly. Daniel P. Withenstien uses processing, which the group researched in section 5.3, with Java at the beginning of the course to teach the fundamentals of programming and afterward moves to Unity with C#. Daniel P. Withenstien makes use of these tools mainly because they make it easier for the student to create programs with visual elements which enhances the learning by visualizing what the code does. Though he argues that Processing can be limiting for experienced students and that Unity has many features that are unrelated to coding. Processing can be limiting due to its lack of functions regarding more complex functionalities. For processing Daniel P. Withenstien uses a purpose build editor also made by the processing foundation. For programming in Unity, the editor visual studio code is used, which is an open-source editor made by Microsoft [10]. As well as GitHub for online file management and project sharing, which is also a Microsoft service [11]. Furthermore, Daniel P. Withenstien allows his students to utilize ChatGPT by OpenAI [12] and Github copilot by Microsoft [11] to help with syntax correction and code completion, as long as the student understands the code and can explain it.

Daniel P. Withenstien has chosen not to use Scratch in his course as he thinks that it would be a disservice to his students due to its limitation at the level of programming he teaches. But thinks Scratch is well-suited for students in elementary school. Daniel P. Withenstien would prefer a programming language that makes use of new lines to separate statements but argues that this attribute makes it harder for students to understand that each line gets executed.

The Programming paradigm that Daniel P. Withenstien would prefer is imperative programming and object-orientated programming, furthermore noted that functional programming could be added but a syntax like Lisp [13] can be difficult to learn or understand. Lastly, Daniel P. Withenstien thinks that Logic programming is too different from the other paradigms, so it would not be suitable for first-time learners.

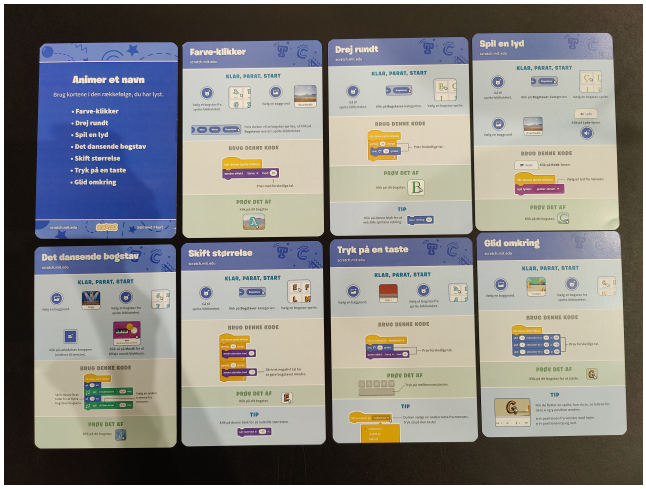
The fourth interview was the last interview, and it was with a volunteer from Coding Pirates. The other interviewees had a more focused target group for their teaching, while Coding Pirates has a variety of courses specified for their participants. Another thing to note is that the students take the courses because of their own or parents' interest and do not get any school credits for taking these courses.

The interviewee was Kristoffer Hansen, who is a helper, teacher, and administrator for the ITU Coding Pirates branch. He works a full-time job as a backend software developer and has volunteered at Coding Pirates for 8 years in his spare time. They have seven courses in this particular branch, where Kristoffer Hansen is a substitute teacher and helper in all seven courses in the current semester.

Kristoffer Hansen talked about some of the Coding Pirates courses that use Scratch, Ozobots, Python, and Unreal Engine (C++) for their programming, depending on the age and experience of the kids. Where Scratch and Ozobots are taught to young kids and Unreal Engine is more focused on experienced or older teenagers. He believes that Scratch is a bit too simple for teenagers who will get more out of jumping straight to the more advanced and professional tools.

Something he talked a lot about was that Coding Pirates focuses on completing projects and having the kids do code with a self-chosen goal in mind. They try to do theory later on, so the kids already have some practical knowledge and code to compare with. This way the kids have the motivation to code and learn theory, as they have a personal goal and a completed program in the end that they can take home and use. They focus a lot on keeping the kids motivated, by doing projects that are fun or have practical use in the end. Kristoffer Hansen mentioned that theory is important for learning code but it is also what makes it a bit dry and boring. This is one thing that he feels that Coding Pirates does better than most programming courses in school since the participants can take theory more relaxed throughout the project. Whereas theory in school courses is very heavy and the primary focus when teaching. One thing they do a lot is use gamification in their teaching process. To help them with the gamification of Scratch, they use colorful flashcards with small projects the kids can use to create interactable games.

(a) The different flashcard stacks that the kids can choose.



(b) A close-up of one of the assignments.

Figure 19: Flashcard for teaching kids.

The biggest problem they face when teaching the kids to program is getting them to understand how a computer thinks and interacts. Kristoffer Hansen tells that most of the kids they teach do not know what a computer is capable of.

Kristoffer prefers the object-oriented and imperative programming paradigm when teaching kids how to code. He does not think that pointers, forced indentation, or type restriction are necessary when teaching kids about programming. He has also expressed that statements should be ended with a semicolon, blocks should be indicated with curly brackets, and comments should be indicated with a double slash (//) or slash asterisk followed by text and an asterisk slash (/**/).

4.2.3 Conclusion of the interview

Out of the four interviews, it can be derived that the teachers prefer to use different programming languages when students start out with programming. However, all interviewees make use of visual languages such as Processing, p5.js, Scratch, or App Inventor to help teach programming. Two interviewees use Python as their first text-based language, one uses p5.js as their first programming language, and the remaining interviewee teaches Processing. After the first text-based language, they transition to different languages including C, C# in Unity, and C++ in Unreal Engine.

Something that most of the interviewees did, was use imperative programming to start with when teaching and end with object-oriented programming (OOP). And even though one of the interviewees did not do it in this way, they were all four very fond of OOP. This was also a trend that was noticed in the questionnaire, where most students ended up using an OOP language as their primary language in the end. When prompted about the difficulties when starting out programming, half of the interviewees noted that error messages can often be problematic for beginners to understand and use.

Regarding the syntax and features in the different languages, all four interviewees agree that using new lines to indicate statement ends, and forced indentation for nesting statements is the most beginner friendly. It should then be noted that half of the interviewees also believe that using curly brackets to indicate nesting statements is good teaching, as they are not too difficult and are more widespread in programming.

The group asked the interviewees about their opinion on type restriction, half of them preferred to have it in a language for beginners, and the other half preferred not to. The arguments against type restriction were that it is easier to program when you do not have to take them into consideration, as then you do not need to know the difference between types. The argument for it was that you probably have to learn them at some point anyways so you might as well learn it from the beginning.

In regards to pointers in the language, three of the four interviewees strongly suggest it is automatically done by the

language and not something the programmer should think about and do manually, while the remaining interviewee was inconclusive in their opinion of pointers. There were also quite a few opinions only mentioned by singular interviewees. While all of these are valid opinions and will be kept in mind, will the common and conflicting opinions be more heavily weighted, when figuring out the problem to tackle and the design of the solution.

4.3 Analysis and discussion of the findings

The upcoming section will revolve around further enlightenment of topics that are the main focus points of the project area. This will serve as background to create a foundation for defining the problem statement, that needs to be resolved during the duration of the project. The topic discussed will be based on the previously gathered data from the analysis, and the topics being explored are target group, common barriers, and additional thoughts.

4.3.1 Target group

This section will discuss and analyze the target group and conclude which target group the project should be focused on and why.

As mentioned previously in the Introduction, section 1, this project is based on the topic "Beginners learning to code". This decision opened up the possibility of creating a language well-suited for people learning how to code, therefore the target group should be aimed at beginners.

Based on the results in the survey question 4, it can be concluded that the project's target group could be aimed at beginners between the age of 16-18 and 13-15 years old. It can be backed up by the data gathered from the participants in this question, which shows a more significant amount started to learn to code in that age gap. It would be ideal to target a beginner's coding language to the age where people learn to code.

When looking further into what the target group for this project could be, there are two clear contenders, which are teenagers and young adults, as the interviews and questionnaire show that this group of people can benefit from a text-based programming language that is targeted to beginners. From the interviews and the questionnaire, the group obtained a lot of data about young adults who attend a gymnasium and who have just started programming. A beginner language could be a good starting point before programming with more complicated programming languages, such as C, later on.

The target group for this project based on this section was chosen to be teenagers and young adults learning to code. This target group will be kept in mind throughout the project, the creation of the problem statement, and the product.

4.3.2 Common barriers

Another topic worth addressing is the common barriers that beginner programmers encounter when they are learning to code. It was observed that the following barriers were persistent throughout the analysis: code knowledge and error messages.

Code knowledge in this context covers both the syntax and the keywords/commands of programming, and also refers to intuition which works hand in hand with both memory and understanding of code. In question 10 in the questionnaire regarding the first problems encountered when learning code, the following answers were commonly chosen: all the commands to remember, syntax, and structure of commands. The same commonalities can be seen throughout the interviews as well, they prefer languages that are easier for the user to learn, with simpler syntaxes. It was mentioned that the wide range of commands causes issues remembering for the target group and that understanding statement logic is challenging. To lessen the burden on the target group's memory, while designing a solution for the problem, the following things should be taken into account: having a more intuitive syntax and commands, as well as a relatively lower amount of commands.

Error messages A large number of participants have encountered error messages as problematic upon learning programming, as seen in question 10 regarding encountering problems when first learning to code. Some of the interviewees had the following remarks about error messages: "They are problematic because they are hard to understand", "not very specific/useful", and "Students tend to google the error messages instead of trying to understand them", as seen in section 4.2.2. More descriptive error messages, that possibly propose a solution or a

more descriptive message should be highly considered.

4.3.3 Additional thoughts

A worthy topic to be considered is visual programming, relating to gaining a visual representation of one's code. This idea of visual programming has been presented in a positive light by all the interview participants, a commonality could be seen that each interviewee made use of a visual programming language as a teaching tool, and was also one of the more prevalent answers in the questionnaire question 12 regarding features that the target group liked. On that basis implementing some sort of way to program visually could prove beneficial, and therefore should be considered.

And the final topic worth considering is programming paradigms. In the questionnaire question 13 it was found that most languages used were multi-paradigm, and the group speculated that either object-oriented or imperative paradigm was used. This speculation was confirmed by the interviews, which mentioned imperative and object-oriented paradigms are preferred for teaching programming. The product should be at minimum an object-oriented or an imperative paradigm, or even be a multi-paradigm language supporting imperative and object-oriented programming.

4.4 Problem statement

Based on the analysis section, a problem statement for the project has been brainstormed and discussed. The following is the problem statement the group has decided to work towards solving:

How can a programming language be designed and implemented to address the most common barriers for teenagers and young adult programming beginners?

- How can the keywords be intuitive and memorable?
- How can the programming languages syntax be intuitive?
- How can the programming language use visual elements and leverage graphics to enhance learning?
- How can error messages be helpful and understandable for programming beginners?

5 State of the Art

With the problem statement done, the focus for this project is to make a programming language mainly for beginners and therefore the group has gained insight into which problems often occur when teenagers and young adults learn programming. The knowledge of these problems was gained through interviews, a questionnaire, and target group analysis.

The group chose to research existing programming languages for the purpose of identifying features and qualities that should be considered in the design of the programming language made for this project. As the programming language is developed for the purpose of learning to code, this section will put a special emphasis on the learning aspect of the programming languages examined. Lastly, the sheer quantity of programming languages makes it impossible to research them all, and therefore the group has chosen a selection of programming languages that can be distinguished from one another by the different interesting features that make them better for learning.

5.1 Scratch

Scratch is a free visual programming language that is developed for children over the age of six, and is created by the Lifelong Kindergarten group at the MIT Media Lab [14]. It is used for teaching programming in primary schools, high schools, and also at some universities. Scratch has more than three million users registered [14] and they refer to themselves as the world's largest coding community for children [15]. It can be useful to learn Scratch before starting text-based programming because it can teach you several concepts which are used in programming [14]. It is a creative way to learn to code and it is block-based, which means that you are able to place different pre-made coding blocks together to create a working program. The blocks and the way they can be moved around can give associations to Lego blocks because they fit together in a certain way. The blocks can not be placed wrong so mistakes can not be made. It is not possible to make variables in the same way as in text-based programming languages, they can however be changed by changing the numbers and colors in the circular shapes on blocks.

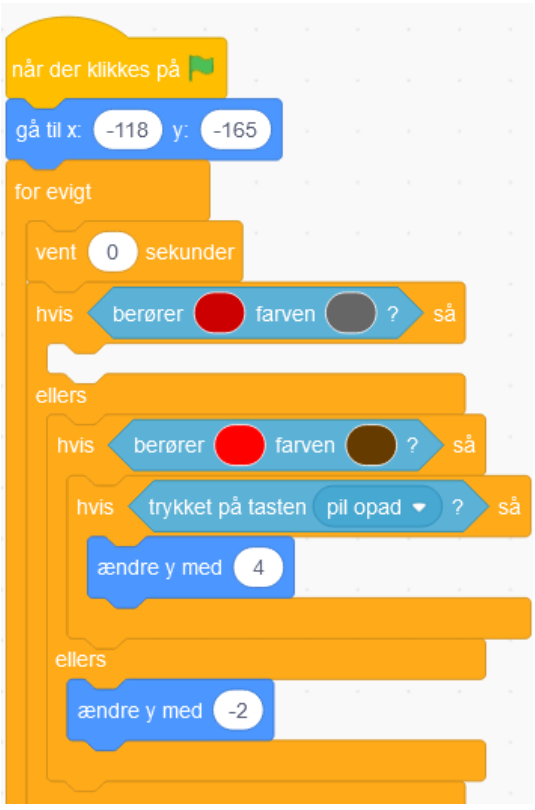


Figure 20: Example from the Scratch game "Platform tutorial" showing an if statement [16].

5.2 Python

On the basis of the results gathered through the questionnaire, in section 4.1, more specifically question 6, Python overall was perceived by the majority as the most favorite programming language and two of the interviewees uses Python as the first text-based language, as seen in the conclusion of the interview 4.2.3, and thus the language will be researched.

The Python programming language was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum in the Netherlands and is currently maintained by a non-copyright organization The Python Software Foundation (PSF) [17]. The Python language can be used to solve a wide variety of problems through web development, software development, mathematics, or system scripting, on different platforms (Windows, Linux, Mac etc.) [18]. Python code can be written in the imperative paradigm, object-oriented paradigm, or functional paradigm, as it is designed to be able to work with all three paradigms. Its syntax is designed for readability, allowing for a more compact code, and is very similar to the English language. Some syntax examples are: new lines are used for ending a command, indentation is used for scope defining [18], and variables are auto-assigned a type upon declaration (though a type can be manually assigned with casting) [19], the mentioned syntax can be seen in the code example in figure 21. The previous observations imply that Python is a powerful, high-level, general-purpose, multi-paradigm, and portable programming language [20].

```
# Python 3: Fibonacci series up to n
>>> def fib(n):
>>>     a, b = 0, 1
>>>     while a < n:
>>>         print(a, end=' ')
>>>         a, b = b, a+b
>>>     print()
>>> fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

Figure 21: A code example from python.org

5.3 Lua

The Lua programming language is designed, implemented, and maintained by a team at the Pontifical Catholic University of Rio de Janeiro in Brazil and is a lightweight, multi-paradigm language [21]. Lua is embedded in many games like Roblox [22], World of Warcraft [23], Garry's mod [24], and Factorio [25] to make add-ons and other modifications to the games. It can therefore be the starting point for many peoples when learning how to program. Lua has several features that make it easier to learn for people who are new to programming. Firstly, Lua has a small set of keywords compared to other written programming languages [26] [27] [28], which means the programmer has less to memorize. Secondly, Lua has dynamic variable typing which means that the programmer does not have to specify the type of a variable [29]. Lastly, Lua only has one type of collection called a table which is implemented as an associative array, that by default uses numbers as identifiers, but the identifier can be set to any string value [30].

```lua
for i = 1, 100 do
    if i % 15 == 0 then
        print("FizzBuzz")
    elseif i % 3 == 0 then
        print("Fizz")
    elseif i % 5 == 0 then
        print("Buzz")
    else
        print(i)
    end
end
```

Figure 22: FizzBuzz made in Lua

5.4 Processing

Processing is a library, which means it is a collection of pre-written code and functions that can be used by the programmer to more easily create different functions and potentially optimize their code [31]. The Processing library was first created in 2001 by Ben Fry and Casey Reas [32] and was created to use as a teaching tool for learning how to code [33]. The team behind Processing has created an IDE which is built around and incorporates the Processing library into it, to better help use the tools and commands it provides [34].

The original Processing library is built to extend Java, but in 2012 they started the Processing foundation with the goal of promoting teaching software in multiple places [35]. This led to the creation of a Processing library for both JavaScript also known as p5.js [36], Python [19], and Android [37][35].

The original Processing library that extends Java was created to be as simple as possible and with visualizing data in mind [38]. The point was not to create anything new, but to motivate students by visualizing their work and programs [38], which granted the creators an award for their work in 2005 [39], and Ben Fry, one of the creators, an award in 2011 [40].

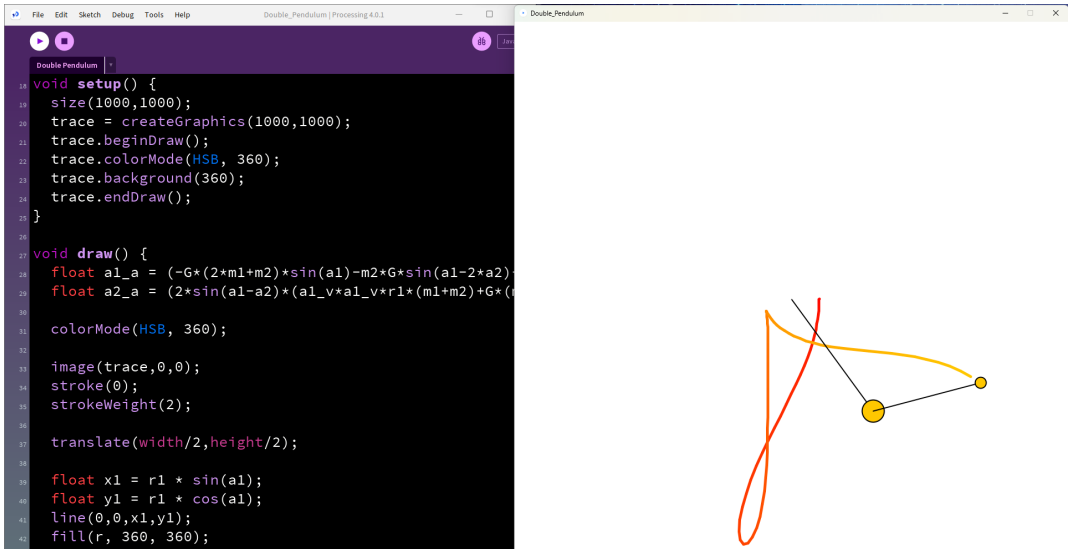


Figure 23: A snapshot of a simple double pendulum code made in Processing and the executing and visualizing of it.

6 Requirements

In this section of the report, the requirements for the product will be determined and described. This has been done through the use of the MoSCoW model, which consists of four categories of requirements: "Must have", "Should have", "Could have" and "Wont have". The creation process of the product requirements and use of the MoSCoW model will be seen throughout the following sections.

6.1 MoSCoW model

With the current programming languages examined in State of the art, in section 5, the results from the questionnaire in section 4.1, and the interviews in section 4.2, the group is now able to structure requirements in order to develop a language made to solve the problem statement. And for this part, the MoSCoW model will be utilized. The MoSCoW model is a prioritization technique used to categorize requirements [41].

The model is divided into four segments each containing requirements based on prioritization. The four segments are "Must have", "Should have", "Could have", and "Won't have". And in this project, the MoSCoW model is prioritized on requirement importance based upon data gathered from the analysis, the group's skillset, and time limit.

The top prioritized requirements should be placed into the "Must have" segment, which should be vital to the product. These must be focused on completing to have the minimum viable product.

The next segment is "Should have" where the requirements are essential to the product but not vital. These requirements can significantly improve the product's quality and/or capabilities.

Following is the segment "Could have". These requirements are smaller quality-of-life features and will only be worked on if the "Must have" and "Should have" segments are completed.

Lastly is the "Won't have" segment. The "won't have" requirements are the last category in the MoSCoW model. These are features discussed but will not be implemented in the product. This decision is based on a lack of time to implement the feature, the feature contradicts the problem statement and focus, or something in between.

6.2 MoSCoW Requirements

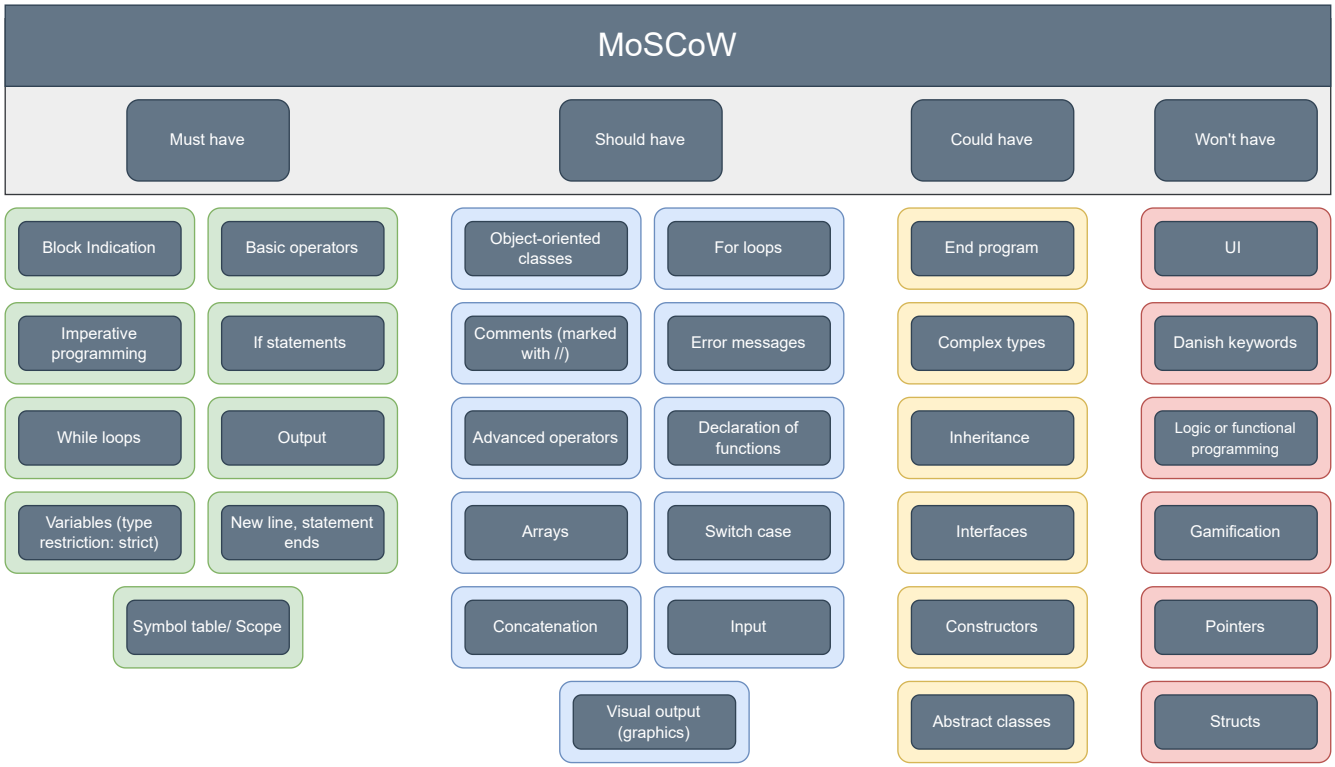


Figure 24: MoSCoW prioritization of the requirements

.

6.2.1 Must have requirements

Block indication:
The group choose to use the block indicates "then" and "do" for the beginnings of a block and "end" for the conclusion of a block. This was not the original choice since the group wanted to use indentation as the only indicator of blocks, but this feature was too difficult to implement and was therefore discarded in favor of the current solution.

Basic operators:
This requirement is fulfilled when the intended programming language contains basic operators. The operators meant to fit in this requirement are the addition (+), subtraction (-), multiplication (*), division (/), modulo (%), is equal (=), lower (<), higher(>), lower or equal to (<=), higher or equal to (>=), and not (not). The group has decided that the equal sign should be similar to how the equal sign is used in standard mathematics, instead of programming languages that use "==". When something is not equal, the user of the language should write "not", because the group found out through questioning people with no prior experience in programming, that most do not understand the meaning of "!=".

Imperative programming:
Imperative programming is a programming paradigm written in a top-down manner, which is the order the program executes it. The group has decided based on the interview results and their own experience that the paradigm the language should focus on is the imperative programming language paradigm.

If statements:
Based on the interviews, the survey, and the groups own knowledge, the group has decided that "if statements" are essential statements in programming and are often used commonly by beginners since the statement is easy to understand. It has therefore been decided that it is a must-have requirement that the new programming language supports the use of if statements.

While loops:
While loops are an essential part of coding, this importance, the group's own experience, and the results from the interview, resulted in While loops being placed in the "must have" requirements.

Output:
Based on the results from the collection of qualitative data, the commands to print output should be written in the following manner: print "(text)".

Variables (type restriction: strict):
According to the interviews type restriction is preferred, since it teaches a valuable programming habit early on. Having fewer keywords is another main point from the questionnaire and interviews, therefore a compromise was made that still allows for a fundamental understanding of type restriction, even though the number of variable types was lowered to Float and String. This requirement describes that when the user declares a variable they have to write the corresponding variable type and value.

New line, statement end:
Based on the results from the different interviews it has been decided that when ending a statement in the project's programming language, it should be done simply by a new line. This could also have been done by using a semicolon (;) at the end of each statement the same way as the language C does, however, it has been decided that it is a must-have requirement that the end of a statement is shown by a newline.

Symbol table/ Scope:
Throughout the project, the group discovered that a symbol table is a "must have" for the compiler and project to be successful.

Table 2: Table over the must-have requirements and their origins.

		Must Have		
No.	**Requirement**	**Description**	**Functionality**	**Origin**
1.	Block indication	Using then, and do to begin a block and end to end a block.	Nonfunctional	Brainstorm/ Interview
2.	Basic operators	The basic operators such as addition (+), subtraction (-), multiplication (*), division (/), modulo (%), is equal (=), lower (<), higher(>), not (!) should complete this requirement.	Functional	Brainstorm/ Interview
3.	Imperative programming	The language must execute code in an imperative style.	Nonfunctional	Interview/ Survey/ Brainstorm
4.	If statements	The new programming language must contain and support the use of if statements.	Functional	Interview/ Survey
5.	While loops	The programming language must be able to do a loop of some kind. The while loop is the most basic loop, which is why it is required.	Functional	Brainstorm Interview
6.	Output	The programming language must have an output function implemented as a method for the user to use, it has been decided that this function should be called with "print"	Functional	Interview/Brainstorm

7.	Variables (Type restriction)	The programming language should have strict type restriction. Meaning that the type of the variables should always be manually declared.	Functional	Interview
8.	New line, statement end	A newline indicates the end of a statement.	Functional	Interview/ Brainstorm
9.	Symbol table/ scope	A symbol table is a must to handle scope and types in the language	Functional	Brainstorm

6.2.2 Should-have requirements

Object-oriented classes:
This requirement primarily describes that the structure of the language should have characteristics of an object-oriented language. This means that the program allows imperative programming through a hidden main class, but also has the ability to further create its own classes.

For loops:
Just as while loops are an essential part of coding so are for loops. However, for loops have been placed as a "should have" requirement due to the ability to create for loops through while loops, but while loops can not be created through for loops. Therefore while loop has been placed as a "must have" requirement and for loop has been placed as a "should have" requirement.

Comments (marked with // or / ... */):*
Based on the results from the interviews regarding which comment indicator symbol they found worked best with new beginners, it has been decided that the symbol indicating a comment in the new program should be a double backslash (//). This requirement however is a should-have requirement and not a must-have due to it not being a core element to the function of the programming language itself. It should also be possible to make multiple-line comments by using a slash asterisk to start and end with an asterisk slash (/* ... */).

Error messages:
Error messages are an essential tool that can be used to learn from errors in coding and further prevent them. Based on the survey and interview results, the group has decided they want to put some focus on creating good and helpful error messages. However, it is not an essential element of the product and has therefore been placed as a "should have" requirement.

Advanced operators:
These operators were deemed more advanced and more time-consuming to develop than the basic operators in the "must have" requirements and therefore have been placed as "should have" requirements. The advanced operators contain operators such as: square root of, power of, log(x), limit, ceiling, and floor.

Declaration of functions:
A classic tool in a programming language is to declare functions. That way it is a lot easier to run the same code from multiple places. As the program is able to run and work without functions by rewriting the code at each position, the group decided not to include it in the "must have". Though rewriting the code at each position is both inefficient to write and confusing to read, the group prioritized it to be a "should have".

Arrays:
An array is a type of variable collection that is commonly used when programming, it is typically one of the first collection types beginners learn to use and therefore would be relevant to implement in the group's language.

However, it is not an essential element or focus of the language and has therefore been placed as a "should have" requirement.

Switch case:
The switch case is a statement type often used when programming and can be a great statement to learn and understand for beginners. It is not an essential statement when coding as the same functionality can be created through if statements. The implementation of the switch case has therefore been placed as a "should have" requirement.

Concatenation:
The group decided to take the rules of concatenation into consideration when creating their language. They wanted the concatenation to be similar to Java's, where the concatenation of two strings is written as: (text) + (text). As most programs can still be written without Concatenation, the group has then chosen to place it in "should have".

Input:
The group decided to focus on the output more than the language being able to input through the terminal. The group has decided the command for reading an input should be: read (text).

Visual output (Graphics):
This requirement describes the idea of the language outputting some graphics or visual aspects of the code written. This requirement is based on the interview results, seen in section 4.2.3, it is also inspired by The processing library seen in section 5.3.

Table 3: Table of should have requirements and where they originate from.

Should Have				
No.	**Requirement**	**Description**	**Functionality**	**Origin**
10.	Object-oriented classes	The programming language should be able to have classes in it but is able to work without.	Functional	Brainstorm
11.	For loops	The prioritization of implementing for loops as its own type of loop.	Functional	Brainstorm/ Interview
12.	Comments (marked with // or /* ... */)	The comment indicator in the new programming language should be the symbol //.	Functional	Interview
13.	Error messages	The group has decided that putting the focus on creating good error messages, when they occur should be a priority in the project	Nonfunctional.	Brainstorm/ Survey/ Interview
14.	Declaration of functions	The declaration of functions is a great tool when writing and reading a program, but not essential.	Functional	Brainstorm/ Survey
15.	Advanced operators	Advanced and time-consuming operators such as the square root of, power of, log(x), limit, ceiling, floor.	Functional	Brainstorm/ Interview

16.	Arrays	An array is a collection type that typically is implemented and supported in coding, however it is not an absolute essential when learning the basics of programming, but it could still add to the new language.	Functional	Brainstorm
17.	Switch case	The implementation and support of the switch statement in the new programming language.	Functional	Brainstorm
18.	Concatenation	The way the language should concatenate two strings together should be similar to Javas.	Functional	Brainstorm
19.	Input	Input is not a must-have based on the group's decision to focus on output first. If an input function is to be added it should be called with "read" in the terminal.	Functional	Interview/ Brainstorm
20.	Visual output (Graphics)	Research shows that a visual output helps understand the functionality of the code.	Functional	SOTA/ Interview/ Questionnaire

6.2.3 Could-have requirements

End program:
An end program is a simple method that ends the program when called. This functionality was deemed to be a low priority as it is not a necessary concept to understand to learn programming.

Interfaces:
Interfaces are a type of syntax or structure that allows certain properties to be enforced on objects (classes). Interfaces are primarily used in the context of OOP (object-oriented programming) which has been marked as a "should have" requirement. Interfaces are prioritized as a "could have" requirement, since they are not essential to learning object-oriented programming but are a nice feature that could be implemented.

Complex types:
Complex types include collections such as different lists, queues, and sets. These complex types are not necessary for beginners to know and understand and have therefore been placed as a "could have" requirement in the language.

Constructors:
Constructors are a great concept and tool to use and learn. However, the group has decided that constructors are more advanced than complete beginners need to have. They have therefore been placed as a "could have" requirement and are therefore not a primary priority.

Inheritance:
The group has based on their own experience deemed inheritance as too difficult and an unnecessary concept for beginners to learn. Inheritance is a concept that can be used in context with OOP (object-oriented programming), OOP has been placed as a "should have" requirement in the MoSCoW figure and not as a "must have". This also indicates that Inheritance is not a priority and only a "could have" for the language of this project.

Abstract classes:
Abstract classes from object-oriented programming are very practical variations of a class. What makes Abstract classes differ from standard classes in object-oriented programming is that an Abstract class can not be instantiated in the program, rather their fields, and methods have to be inherited by another class. Therefore Abstract classes have to be implemented after Inheritance and have been prioritized the same as Inheritance.

Table 4: Table of our "could have" requirements and where they originate from.

Could Have				
No.	**Requirement**	**Description**	**Functionality**	**Origin**
21.	End program	An "End program" method could add to the experience for the user.	Functional	Brainstorm
22.	Interfaces	Some of the most classic object-oriented programming patterns require the use of Interfaces to implement.	Functional	Brainstorm
23.	Complex types	Complex types would add to the flexibility and write-ability of the language.	Functional	Brainstorm
24.	Constructors	The construct method for object-oriented classes.	Functional	Brainstorm
25.	Inheritance	The language could have inheritance implemented to handle classes, however, this is not essential for beginners to learn in the language of the project.	Functional	Brainstorm
26.	Abstract Classes	This requirement is tightly associated with the Inheritance requirement and therefore has been placed as a "could have" requirement as well.	Functional	Brainstorm

6.2.4 Won't have requirements

UI:
The User interface (UI) of the programming language is not in the scope of this project and has therefore been placed as a wont-have requirement.

Danish keywords:
When looking into the results of the survey, it can be seen some of the participants, especially the younger ones marked that they had some struggles using the English language when coding. Based on this the group discussed if it would benefit if the language was to be written in Danish with Danish keywords. However, the group decided not to do this as the difference between Spark and other languages would be too hard and unnecessary to learn and understand. This means the new language, Spark, will only be with English keywords.

Logic and Functional programming:
The Logic paradigm and the Functional paradigm are both very different ways of programming compared to both each other and the two paradigms the groups have chosen. The opinion of all the interviewees was that both Logic programming and Functional programming are not suited for beginning programmers, and are therefore in the "won't have".

Gamification:
The group considered and discussed if the new programming language should be "gamified" or made look visually more fun or like a game in the same way as Scratch. However, the group decided against it and it has therefore been placed as a "won't have" requirement.

Pointers:
Pointers are a variable that holds the memory address of another variable. Based on the group members' own experience and on the interviews, it can be concluded to being very difficult to learn and understand. The group has decided not to implement pointers in their new language since they are not relevant to the core of the project and would be way too time-consuming.

Structs:
A struct can be a bit different depending on the language and the paradigm of said language [42] [43]. Struct is short for structure and is a value that is able to encapsulate data [43]. It is placed into the "won't have" category since a class can essentially do the same thing, which the group has prioritized over the struct.

Table 5: Table of our will-not-have requirements and where they originate from.

Will Not Have				
No.	**Requirement**	**Description**	**Functionality**	**Origin**
27.	UI	A visual aspect or help for the user when coding.	Nonfunctional	Survey/ Brainstorm
28.	Danish keywords	The language of the keywords the user would be writing when coding in the new language.	Functional	Survey/ Brainstorm
29.	Logic and Functional programming	Logic and Functional programming is a completely different programming paradigm, where research shows that it is more difficult paradigm to learn.	Nonfunctional	Brainstorm/ Interview
30.	Gamification	The visual aspect of the code could be showcased in a more game looking way.	Functional	Brainstorm
31.	Pointers	Implementing and supporting the use of pointers in the new language.	Functional	Brainstorm/ Interview
32.	Structs	As the program will be able to create and handle objects, which is able to have the same functionality as a struct, will structs not be included in the final product.	Functional	Brainstorm

7 Syntax and semantics specification

Now that the project has a specified problem statement and specific requirements for what the product should contain, it is time to shape the product which is a programming language and a compiler for the language. The group has named the coding language they are designing for this project "Spark". In this section, the definition and grammar rules of Spark will be presented. Firstly the syntax of the language will be seen in section 7.1 followed by a semantics section 7.2.

7.1 Syntax

This section contains the syntax and syntax rules of the Spark language, which was created for this project. This section starts with an informal definition of the language followed by a formal definition, then the syntax choices of the language will be justified.

To clear confusion the group looked up the definition of syntax: "The syntax of a programming language is the form of its expressions, statements, and program units. Its semantics is the meaning of those expressions, statements, and program units" according to Sebesta [44].

7.1.1 Informal definition

An informal definition of a programming language describes what keywords, data types, and variables exist in that programming language [45].

Keywords
A list of all the reserved keywords possible to use in the language can be seen below in figure 25:

- if
- while
- else
- break
- continue
- number
- text
- print
- printLine
- do
- then
- end
- else
- on
- start
- eachFrame
- color
- background
- circle
- square
- triangle
- darkRed
- red
- lightRed
- darkGreen
- green
- lightGreen
- darkBlue
- blue
- lightBlue
- black
- white
- darkGrey
- grey
- lightGrey

Figure 25: List of all the keywords

It is worth mentioning that the keywords "print" and "printLine" have different but similar functions. The "print" keyword, prints the output on the same line, while the "printLine" keyword creates a new line before printing the output.

It is also worth noticing that the keywords "number" and "text" are also the two data types available in the language. However, the words are also listed as reserved keywords due to these words being used to declare and initialize the variables of those types.

Data types
This language contains two data types, a list and description of these can be seen in the table below:

Table 6: Data types

Data types	Description
number	A placeholder for at float or integer
text	A placeholder for a char or string

Variables
In the language Spark developed through this project, there have been created some specific rules for declaring and assigning variables, these can be seen in section 7.1.2. When declaring a variable, which can be of the type "text" or "number", the variable has to be initialized correctly. If a type is initialized with a non-matching value the compiler will produce an error and not compile. However, if a variable is declared with the right value, it can then be reassigned whenever after the declaration.

7.1.2 Formal definition

A formal definition of a language uses the context-free-grammar (CFG) to showcase and specify the language's syntax and regular expressions [45]. Further in this section, the grammar of the project's language Spark can be seen through BNF syntax specifications. Firstly the CFG can be seen for each of these regular expressions in the language: Statement, lines, assignment, declaration, control structures, and operators.

The BNF consists of several rules which the group has chosen for the language. These rules define the syntax and which keywords can be used in the programming language and their relation to the code surrounding them. Each of these rules is built on what is called non-terminals and terminals [45]. A non-terminal defines the relations between rules and is replaced by another rule to further be built upon [45]. Terminals define the keywords and are typically the final leaf in each branch in the abstract syntax tree [45]. The set of all terminals in the context-free grammar, in this case, the BNF, makes up the context-free-language [45]. Spark's complete uncut BNF can be seen in *Appendix D*.

Statement and Line

The syntax rules for statements and lines can be seen in Listing 1. The language Spark starts with rule "s". Which can either be "extraLines line" or "extraLines globalStatement StartFunction (drawFunction | empty);". By adding a "extraLines" rule at the start, an error could be prevented where if the first line was empty in the input, then the input would not be able to be parsed at all.

```
s: extraLines line | extraLines globalStatement startFunction (drawFunction | empty);

startFunction: 'on' 'start' block forcedLineChange globalStatement;

drawFunction: 'on' 'eachFrame' block forcedLineChange globalStatement;

line: statement forcedLineChange line | variableStatement forcedLineChange line | visualStatement
    forcedLineChange line | empty;

statement: 'if' condition block extraLines elseToken | 'while' condition loopBlock | 'print'
    stringValue | 'printLine' stringValue | 'break' | 'continue';

globalStatement: declaration forcedLineChange globalStatement | empty;

extraLines: ('\n')*;

block: 'then' forcedLineChange line 'end';

forcedLineChange: ('\n')+;
```

Listing 1: Statement and Line in the language Spark.

Assignment and declaration

Assignments and declarations within the language Spark can be seen in Listing 2. The grammar handles these statements under one larger statement called "variableStatement". Then according to the production rules of the grammar, it can be decided whether it is a declaration or an assignment based on the scanned sequence of tokens.

```
variableStatement: declaration | assignment;

declaration: 'number' ID '=' equation | 'text' ID '=' stringValue;

assignment: ID '=' value;

value: ID extraValue | expression extraValue | stringCheckRule extraValue | '(' value ')'
    extraValue;

stringCheckRule: String;

extraValue: '+' value | operator value | empty;

stringValue: ID extraStringValue | String extraStringValue | num extraStringValue;

extraStringValue: '+' stringValue | empty;

equation: '(' equation ')' extraEquation | ID extraEquation | expression extraEquation;

extraEquation: operator equation | empty;

expression: '-' num | num;

num: Digits decimal;

decimal: '.' Digits | empty;
```

Listing 2: Assignment and declaration in the language Spark.

Control structures

The programming language contains both iterative and selective control structures:

- Iterative structures are used to repeatedly execute code until a certain condition is met. Spark's only iterative structure is *while* statements.
- Selective structures are used to execute code when a specific condition is met, allowing flexibility, since the code can branch in different directions depending on the conditions. Spark contains *if*, *else* statements, and *else if* statements.

```
elseToken: 'else if' condition block extraLines elseToken | 'else' block | empty;

condition: '(' condition ')' extraCondition | 'not' notCondition extraCondition | singleCondition
    extraCondition;

extraCondition: 'and' condition | 'or' condition | empty;

notCondition: '(' condition ')' | singleCondition;

singleCondition: equation comparator equation | String '=' String;

block: 'then' forcedLineChange line 'end';

loopBlock: 'do' forcedLineChange line 'end';
```

Listing 3: Control structures in the language Spark.

Visual features

Visual features are all the code that creates some form of visual output. The language is set up to automatically implement the right library and construct the output file to use the visual features if the user implements it in their program.

```
visualStatement: figure | colorPick;

colorPick: 'color' (colorText | parameter parameter parameter) | 'background' (colorText |
    parameter parameter parameter);

colorText: 'darkRed' | 'red' | 'lightRed' | 'darkGreen' | 'green' | 'lightGreen' | 'darkBlue' | '
    blue' | 'lightBlue' | 'black' | 'white' | 'darkGrey' | 'grey' | 'lightGrey';

figure: 'circle' parameter parameter parameter parameter | 'square' parameter parameter parameter
    parameter | 'triangle' parameter parameter parameter parameter parameter parameter;

parameter: expression | ID;
```

Listing 4: Control structures in the language Spark.

Operators and Lexemes

The operators in the language are +,-,*,/ and %, and the comparators are <, >, =, <=, and >=, since they are quite necessary to make simple equations, etc.

It is possible to write comments in the Spark language, they have to be written with // or /**/. This rule can be seen in the BNF "LINE_COMMENT" rule. In the BNF rule "WS", it can be seen that empty spaces and indentations are skipped, it can therefore be stated that Spark is more lenient on writability. Additionally, new lines are not included since they are used as separators.

```
operator: '+' | '-' | '%' | '/' | '*';

comparator: '<' | '>' | '=' | '<=' | '>=';

empty: ;

String: '"'  (~'"')*  '"';

ID: ([a-z]|[A-Z]) ([a-z]+|[A-Z]+|[0-9]+)*;

Digits: [0-9]+;

WS: (' '|'\t' | '\r')+ -> skip;

COMMENT: '/*' (~'*')* '*/' -> skip;

LINE_COMMENT:   '//' (~[\r\n])* -> skip;

ErrorSymbol: (~('"' | [a-z]| [A-Z] | [0-9]|' '|'\n'|'\r'))*;
```

Listing 5: Operators in the language Spark.

Token specification

In this part of the formal definition of the language Spark, a fragment token specification table will be presented. This table can be seen below.

The fragments used in Spark are the symbols the user is allowed to write. In Spark, there are only two types of fragments, letters in "ID" and digits in the fragment "Digits".

Table 7: Fragments used

Fragments	
ID	'([a-z]|[A-Z]) ([a-z]+|[A-Z]+|[0-9]+)*'
Digit	'[0-9]+'

An example of the usage of tokens in Spark can be seen in the coding example below and the parse tree generated according to the code.

```
number a = 10
number b = a - 2
```

Listing 6: Coding example in the language Spark

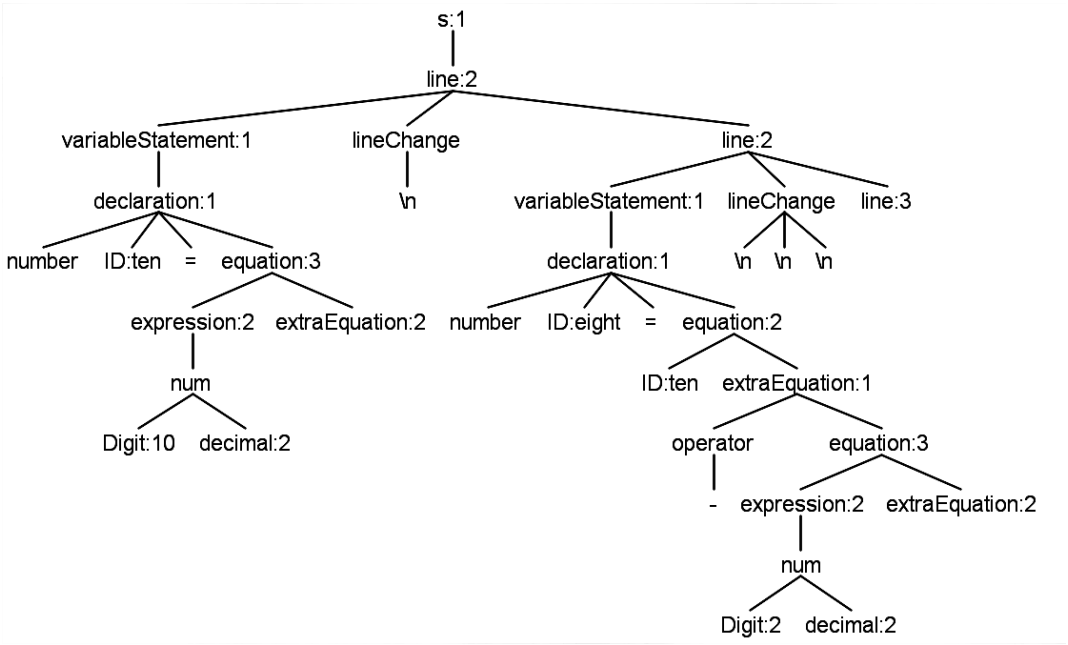


Figure 26: Example of a parse tree

When looking at the generated parse tree, it is important to understand how to read it properly. When looking at the first childNode of the start rule "s" which can be seen in the example in figure 26, the childNode is "line:2". This node refers to the rule "line" which is being used in this example. The ":2" in the rule "line:2" indicates it is option 2 in the line rule which has been taken to use here. The listing 1 shows how in the "line" there are multiple options within the "line" rule. These options are "statement lineChange line" OR "variableStatement lineChange line" OR it being empty. Therefore when looking back at the parse tree, when the childNode of the start rule "s" is "line:2" it refers to the rule "variableStatement lineChange line" option within the "line" rule.

The code example is a very short example of two declarations. Both of a number type, while the second uses the previously declared number. The tokens which have been used in line 1 in this example can be seen to be: "number" + "ID" + "=" + "Digit" + "decimal".

The "number" token indicates the variable type being declared, which is "number".
The "ID" token indicated the ID for the variable being declared, which is "a".
Then the token "=" is used to assign the variable a value.
The "Digit" token contains an integer value, which in this case is 10, and is assigned to the variable.
Lastly the token "decimal" is used to check and handle if the value is an integer or a float. This has to be done each time a number is declared in Spark.
All of these can be seen in the code and the parse tree above.

7.1.3 Error Handling

A topic worth covering during compiler design is error handling, which is especially important since error messages are one of the main project aspects. As mentioned in 4.3.2 one of the common barriers for beginner programmers is that the error messages are hard to understand and not very specific/useful. A proper construction of error handling can prove beneficial in helping users to write code according to the programming languages grammar. For error messages to be beneficial to the user, they need to be presented in a manner that provides relevant information, therefore the error message should consist of the location, the cause, and a possible solution to the error.

Different types of errors can occur corresponding to the compiler phases. These are called compile-time errors which are split into three errors: lexical errors, syntactic errors, and semantic errors [46]. Lexical errors are when "a sequence of characters that does not match the pattern of any token" [46]. Syntactic errors are when the production rules of the grammar are voided [46]. And semantic errors are caused due to incorrect logic [46].

7.1.4 Syntax Justification

Spark as a language is created to be an easy and simple language focused on making coding easier to learn and use for complete beginners. Due to this goal, the group has made some decisions to push the syntax of Spark in that direction.

Block rule
Through the interviews and questionnaire, it was clear that indentation was the preferred way to indicate blocks. Though after some research, it was clear that using and implementing indentations is more advanced than thought at first. Unfortunately there were not enough resources and time for the group to tackle that problem, which caused them to look for different options to solve the problem. A second choice was to take a page from Lua's book, and use words like "then", "do", and "end". Through research, interviews, and the questionnaire, it was clear that these words were intuitive, easy to understand, and easy to use as well, while also being within the capabilities of the group.

Collecting number values into one single datatype
Through the interviews the group realized that an obstacle some beginners encounter when first learning is the difference between the float and integer type normally used in coding. Based on this, the group deliberately chose that it should not be possible to use different data types when dealing with numbers. Instead, there should only be one data type in Spark that handles numbers. The data type chosen for the language is "number" which makes the user able to give the input of a number with or without decimals and it will be saved in the "number" type whether the input is a whole number (integer) or a decimal number (float).

Collecting character values into one single datatype
The group encountered the same thoughts with the types char and String as they did with float and integer. The group chose that it should only be possible to make Strings and not characters because characters do not contribute any different functionalities compared to Strings. Therefore a new data type to handle both char and Strings was created, in Spark that data type is called "text".

7.2 Semantics

This section contains and showcases the semantics and expressions of the language. When looking at operational semantics the group has chosen to use big-step semantics rather than small-step semantics. Big-step semantics describes the entire computation through one single transition. Big-step semantics for each of the expressions available in the language Spark can be seen in chapter 7.2.1. Finally, a section on the justification of the semantics choices is provided in section 7.2.2.

7.2.1 Expressions

Operator expressions:
Big Step semantics of the operators available in Spark can be seen below:

$$\begin{aligned} &\xrightarrow{a} \ is\ operator\ relation\ transition \\ &\xrightarrow{b} \ is\ comparator\ relation\ transition \\ &s = States, V = Value \rightarrow \mathbb{R} \end{aligned} \tag{1}$$

$$[ADDITION_{BS}]\frac{s \vdash a1 \xrightarrow{a} v1,\ s \vdash a2 \xrightarrow{a} v2}{a1 + a2 \xrightarrow{a} v}, v = v1 + v2 \tag{2}$$

$$[SUBTRACTION_{BS}]\frac{s \vdash a1 \xrightarrow{a} v1,\ s \vdash a2 \xrightarrow{a} v2}{a1 - a2 \xrightarrow{a} v}, v = v1 - v2 \tag{3}$$

$$[MULTIPLICATION_{BS}]\frac{s \vdash a1 \xrightarrow{a} v1,\ s \vdash a2 \xrightarrow{a} v2}{a1 * a2 \xrightarrow{a} v}, v = v1 * v2 \tag{4}$$

$$[DIVISION_{BS}]\frac{s \vdash a1 \xrightarrow{a} v1,\ s \vdash a2 \xrightarrow{a} v2}{a1\ /\ a2 \xrightarrow{a} v}, v = v1\ /\ v2 \tag{5}$$

$$[MODULO_{BS}]\frac{s \vdash a1 \xrightarrow{a} v1, s \vdash a2 \xrightarrow{a} v2}{a1\ \%\ a2 \xrightarrow{a} v}, v = v1\ \%\ v2 \tag{6}$$

Comparator expression:
Below Big Step semantics for each of the comparators available in the language Spark can be seen:

$$[Equal\_TRUE_{BS}]\frac{s \vdash a1 \xrightarrow{a} v1,\ s \vdash a2 \xrightarrow{a} v2}{a1 = a2 \xrightarrow{b} tt}, if v1 = v2 \tag{7}$$

$$[Equal\_FALSE_{BS}]\frac{s \vdash a1 \xrightarrow{a} v1,\ s \vdash a2 \xrightarrow{a} v2}{a1 = a2 \xrightarrow{b} ff}, if v1 \neq v2 \tag{8}$$

$$[NotEqual\_TRUE_{BS}]\frac{s \vdash a1 \xrightarrow{a} v1,\ s \vdash a2 \xrightarrow{a} v2}{not\ \ a1 = a2 \xrightarrow{b} tt}, if v1 \neq v2 \tag{9}$$

$$[NotEqual\_FALSE_{BS}]\frac{s \vdash a1 \xrightarrow{a} v1,\ s \vdash a2 \xrightarrow{a} v2}{not\ \ a1 = a2 \xrightarrow{b} ff}, if v1 = v2 \tag{10}$$

$$[LesserThan\_TRUE_{BS}]\frac{s \vdash a1 \xrightarrow{a} v1,\ s \vdash a2 \xrightarrow{a} v2}{a1 < a2 \xrightarrow{b} tt}, if v1 < v2 \tag{11}$$

$$[LesserThan\_FALSE_{BS}]\frac{s \vdash a1 \xrightarrow{a} v1,\ s \vdash a2 \xrightarrow{a} v2}{a1 < a2 \xrightarrow{b} ff}, if v1 \not< v2 \tag{12}$$

$$[LesserThanOrEqual\_TRUE_{BS}]\frac{s \vdash a1 \xrightarrow{a} v1,\ s \vdash a2 \xrightarrow{a} v2}{not\ \ a1 <= a2 \xrightarrow{b} tt}, if v1 \leq v2 \tag{13}$$

$$[LesserThanOrEqual\_FALSE_{BS}]\frac{s \vdash a1 \xrightarrow{a} v1,\ s \vdash a2 \xrightarrow{a} v2}{not\ \ a1 <= a2 \xrightarrow{b} ff}, if v1 \not\leq v2 \tag{14}$$

$$[GreaterThan\_TRUE_{BS}]\frac{s \vdash a1 \xrightarrow{a} v1,\ s \vdash a2 \xrightarrow{a} v2}{a1 > a2 \xrightarrow{b} tt}, if v1 > v2 \tag{15}$$

$$[GreaterThan\_FALSE_{BS}]\frac{s \vdash a1 \xrightarrow{a} v1,\ s \vdash a2 \xrightarrow{a} v2}{a1 > a2 \xrightarrow{b} ff}, if v1 \not> v2 \tag{16}$$

$$[GreaterThanOrEqual\_TRUE_{BS}]\frac{s \vdash a1 \xrightarrow{a} v1,\ s \vdash a2 \xrightarrow{a} v2}{a1 >= a2 \xrightarrow{b} tt}, if v1 \geq v2 \tag{17}$$

$$[GreaterThanOrEqual\_FALSE_{BS}]\frac{s \vdash a1 \xrightarrow{a} v1,\ s \vdash a2 \xrightarrow{a} v2}{a1 >= a2 \xrightarrow{b} ff}, if v1 \not\geq v2 \tag{18}$$

$$[AND\_TRUE_{BS}]\frac{s \vdash b1 \xrightarrow{b} v1,\ s \vdash b2 \xrightarrow{b} v2}{b1\ \ and\ \ b2 \xrightarrow{b} tt}, if v1 \wedge v2 = tt \tag{19}$$

$$[AND\_FALSE_{BS}]\frac{s \vdash b1 \xrightarrow{b} v1,\ s \vdash b2 \xrightarrow{b} v2}{b1\ \ and\ \ b2 \xrightarrow{b} ff}, if v1 \wedge v2 = ff \tag{20}$$

$$[OR\_TRUE_{BS}]\frac{s \vdash b1 \xrightarrow{b} v1,\ s \vdash b2 \xrightarrow{b} v2}{b1\ \ or\ \ b2 \xrightarrow{b} tt}, if v1 \vee v2 = tt \tag{21}$$

$$[OR\_FALSE_{BS}]\frac{s \vdash b1 \xrightarrow{b} v1,\ s \vdash b2 \xrightarrow{b} v2}{b1\ \ or\ \ b2 \xrightarrow{b} ff}, if v1 \vee v2 = ff \tag{22}$$

Statement expressions:
Below big-step semantics for each of the statements available in the language Spark can be seen:

$$[IF\_TRUE_{BS}]\frac{S1, s \rightarrow s'}{if\ b\ then\ S1\ end\ else\ S2\ end,\ s \rightarrow s'}, if s \vdash b \rightarrow tt \tag{23}$$

$$[IF\_FALSE_{BS}]\frac{S2, s \rightarrow s'}{if\ b\ then\ S1\ end\ else\ then\ S2\ end,\ s\ \rightarrow s'}, if s \vdash b \rightarrow ff \tag{24}$$

$$[WHILE\_TRUE_{BS}]\frac{S\ , s \rightarrow s"\ ,\ while\ b\ do\ S\ end,\ s"}{while\ b\ do\ S\ end,\ s \rightarrow s'}, if s \vdash b \rightarrow tt \tag{25}$$

$$[WHILE\_FALSE_{BS}] while\ b\ do\ S\ end,\ s", if s \vdash b \rightarrow ff \tag{26}$$

Visual expressions
Below big-step semantics for each of the visuals available in the language Spark can be seen:

$$[Triangle_{BS}]\ \frac{s \vdash x_1 \rightarrow v_1, s \vdash y_1 \rightarrow v_2, s \vdash x_2 \rightarrow v_3, s \vdash y_2 \rightarrow v_4, s \vdash x_3 \rightarrow v_5, s \vdash y_3 \rightarrow v_6}{triangle\,(x_1, y_1)\,,(x_2, y_2)\,,(x_3, y_3) \rightarrow \triangle} \tag{27}$$

$$[Rectangle_{BS}]\ \frac{s \vdash x_1 \rightarrow v_1, s \vdash y_1 \rightarrow v_2, s \vdash h \rightarrow v_3, s \vdash w \rightarrow v_4}{square\,(x_1, y_1)\,, h, w \rightarrow \square} \tag{28}$$

$$[Ellipse_{BS}] \frac{s \vdash x_1 \to v_1, s \vdash y_1 \to v_2, s \vdash h \to v_3, s \vdash w \to v_4}{circle\,(x_1, y_1)\,, h, w \to \bigcirc} \tag{29}$$

7.2.2 Semantics Justification

The operator expressions in Spark, seen in 7.2.1, are based on expressions known from mathematics, and therefore the operators for addition (+), subtracting (-), multiplication (*), division (/) and modulo(%) are as you would expect, since it would not give any value to change these. This goes for the comparators:<, >, <= and >= as well. As seen in the comparator expressions 7.2.1, when two variables are equal, you simply insert a single "=" between the two variables. If, on the other hand, the two variables are not equal to each other, the keyword "not" must be written. When using "and", "or" and "not" in the language, it is also decided that you must write the words in full. In statement expressions 7.2.1, it is shown how if statements and while loops work. In an if-else statement, you must write "if", "then", "end" followed by "else", "then" and "end", and in a while loop you must write "while", "do" and "end". All these decisions are made based on the collected data in the questionnaire and the interviews, which is explained further in the discussion section 11.

The semantic rules are structured in roughly the same way, so only some selected rules will be explained below.

As seen in rule 10 in the semantics, it will return false if the two values are equal, as it checks if they are not equal. The conclusion can be read in what equates to the denominator of the semantics, where it shows the two variables being compared and return false. The numerator shows a1 being set to the value v1, and a2 being set to the value v2. These variables and values are arbitrary and just represent any variable and value. The rules on the right, outside the denominator and numerator, are conditions that have to be met for the rule to apply.

There are three rules, in particular, the group wants to put into focus. The three rules regarding visuals. Rule 27 showcases the semantics of how a triangle is made in Spark. The six variables: x_1, y_1, x_2, y_2, x_3, and y_3 are all assigned a numeric value that corresponds to a point in the x or y axis to create a coordinate. This can be seen in the denominator where each of the three coordinates is responsible for a point in the triangle. The rules 28 and 29, rectangle and ellipse respectively, are quite similar but instead of six variables they only have four: x_1, y_1, h, and w. The x_1 corresponds to a point on the x-axis, and y_1 to a point on the y-axis. Instead of having multiple coordinates, they instead use the last two variables on the height and width of the figure.

8 Design of the compiler

This section contains descriptions of what each of the compiler's phases does and how each of them acts in the process of compiling a language into another language. The phases the compiler will be going through is: the syntax analysis, the semantics analysis, and the code generation.

8.1 Syntax analysis

The final goal of the syntax analysis phase is to generate an AST (Abstract Syntax Tree). This phase of the compiler consists of two underlying phases, the lexical analysis phase, and the parsing phase. The lexical analyzer focuses on regular expressions and scans the input and turns it into tokens. The parsing phase checks if the source code follows the required syntactical structure and rules. This is done by creating a parse tree and an abstract syntax tree [47].

8.1.1 Parsing methods

When parsing the input into a parse tree, it is paramount to be able to easily identify all the non-terminals of the context-free-grammar, in other words, easily be able to identify the context-free-language from the input [45].

This can be done through LL(k) and LR(k), which represent how the parser reads and handles the input. LL(k) and LR(k) are the most common to use, but far from the only ones as will be seen later when the group goes into detail on how they set up their parser. The first letter represents which direction the parser reads the input, where L is left to right, and R is right to left [48] [49] [45]. The second letter is either a leftmost derivation or a rightmost derivation [48] [49] [45]. That means that when the parse is confronted with a rule from the CFG with the rightmost derivation, it will resolve the rightmost terminal/non-terminal in the given rule. For example, given the rules:

$$\text{start} -> \text{line statement}$$
$$\text{statement} -> \text{if condition block}$$

will the parser handle it like this:

Rightmost derivation:
start
= line ***statement***
= line if condition ***block***
= line if ***condition*** block

Leftmost derivation:
start
= ***line*** statement
= line ***statement***
= line ***if*** condition block

and so on. In these cases, when the parser hits a non-terminal, i.e. something that has no rule attached, it will then go to the next node in the sequence and continue. Lastly, the k inside the parenthesis is a constant and is the lookahead, which means the number of input symbols the parser will use for each step it takes [48] [49] [45].

8.1.2 Ambiguous grammar

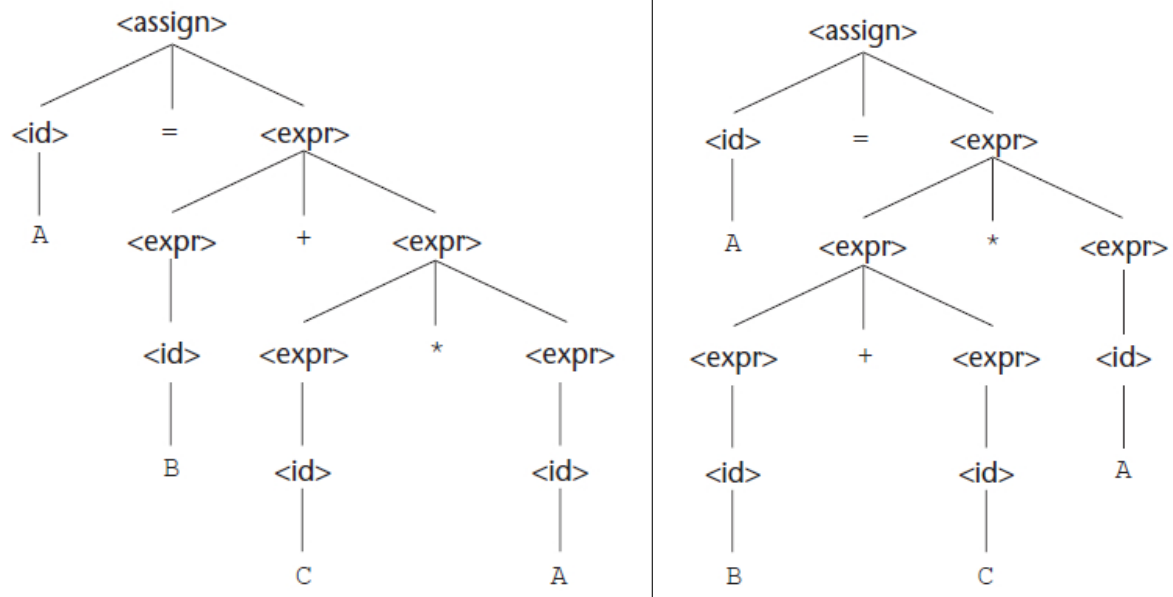
A grammar in which, according to the production rules, it is allowed for more than one possible interpretation or parse tree to exist from a given sentence, is called an ambiguous grammar. Oppositely if the production rules of a grammar allow for only one possible interpretation or parse tree to exist from a given sentence, it's the case of an unambiguous grammar [50].

An example of ambiguous grammar with its possible parse tree derivations will be showcased below:

```
<assign> -> <id> = <expr>
<id> -> A | B | C
<expr> -> <expr> + <expr>
        | <expr> * <expr>
        | ( <expr>)
        | <id>
```

Listing 7: An Ambiguous Grammar for Simple Assignment Statements [45].

Given the following sentence: "$A = B + C * A$", the production rules of the ambiguous grammar from Listing 7, allow for two possible parse trees to exist, which can be seen in figure 27.



(a) Left parsing

(b) Right parsing

Figure 27: Two distinct parse trees for the same sentence "$A = B + C * A$" [45].

According to the BNF of Spark, it can be said that the language itself is ambiguous. As seen previously with LL(k) and LR(k) parsers, there are different ways to scan and handle input, therefore the parser choice plays an important role in getting the language to be parsed in a specific way to achieve the desired result. Achieving the desired result through the choice of Sparks parser will be discussed in the upcoming section regarding parser choice.

8.1.3 ANTLR as a lexer and parser

ANTLR, also known as "ANother Tool for Language Recognition", is a lexer and a parser generator used to build languages, tools, and frameworks. ANTLR can create a parser that can build parse trees, representing how a grammar matches input [51].

In the process of ANTLR working as a lexer, ANTLR scans the input stream from the source code, turns it into tokens, and turns these into a decorated parse tree [52].

When designing a BNF, knowing how the chosen parser works with its input is important. As seen in section 8.1.1, LL(k) and LR(k) are some of the common ones. While, according to Fischer, LL(1) is what most programming languages use [45]. In this case, ANTLR is LL(*), which means that ANTLR uses an algorithm to find which constant (k) the CFG needs to parse the input.

In the case of the group's BNF, it is important that the parser is not LL(1) as some of the rules in the BNF are not decisive with just one lookahead. This is because of the three rules in the groups BNF shown below 8. The notCondition can either be "(condition)" which includes multiple conditions surrounded by parenthesis before the boolean is inverted, note that it starts with a "(", or the singleCondition which can be exchanged with an equation

that also has a rule that starts with a "(". This means the line "if not (a < 10) then" is not decisive with LL(1). But as the LL(*) will have a higher lookahead, it will be able to figure out which rule to use.

```
notCondition: '(' condition ')' | singleCondition;

singleCondition: equation comparator equation | String '=' String;

equation: '(' equation ')' extraEquation | ID extraEquation | expression extraEquation;
```

Listing 8: Small selected section of Spark's BNF.

8.2 Semantics analysis

The semantics analysis phase takes the generated AST (abstract syntax tree) and decorates it through the given type checking and scope checking [53]. The phase of scope checking is described in section 8.2.1, followed by a description of the type checking phase in section 8.2.2.

8.2.1 Scope checking through symbol table

A symbol table stores information about the state of variables, this information is gotten through the AST which was created through parsing in the compiler's syntax analysis phase. The symbol table keeps track of the scope, name, and value of the variables declared. The compiler is responsible for creating and maintaining the symbol table [54]. The implementation of the group's symbol table and its scope checking can be seen in section 9.4.

There are multiple ways to implement a symbol table, it can be done through a binary search tree, hash table, or a linked list to mention a few. The group decided to implement it as a stack of HashMaps due to its ability to quickly search [54].

8.2.2 Type checking

Following the scope checking through the symbol table, the AST generated in the syntax analysis phase will be type checked. This can be done through an appropriate Visitor that walks through each node in the AST [45], the Visitor then decorates the AST tree with type information.

The compiler for Spark type checks the AST through the assignmentVisitor which the group implemented. The group decided to type check through the declarationVisitor when a variable is declared and through the assignmentVisitor when a variable is reassigned to a new value. This was decided so the types get checked continuously rather than on their own afterward. The implementation and use of type checking through the assignmentVisitor will be presented in section 9.3.1.

8.3 Code generation

The code generation phase is the final phase of the compiler. In this phase, the compiler takes the decorated AST formed from the semantic analysis phase and generates the code in another language [55].

The Spark compiler translates code written in the language Spark into the language Java, this is done through visitor patterns. The reasoning and decision for choosing Java as the target language for the compiler is presented in section 9.1.

9 Implementation

This section contains how the code and language have been implemented through the language Java and through ANTLR. Firstly, section 9.1 will present the group's thoughts and considerations of what language the compiler should translate the source code into. Secondly, the implementation and use of ANTLR are described in section 9.2. Following that is a section on which visitor patterns were chosen to use and how they were implemented, this can be seen in section 9.3. Then the implementation and handling of errors and error messages are presented in section 9.5, followed by the use and implementation of the processing library seen in section 9.6. And lastly, the implementation of the code generator class and phase can be seen in section 9.7.

9.1 Choosing a target language

The group had different considerations for potential target languages, which the source program Spark could be translated into in the code generation phase. The chosen language is Java and the reason for this will be explained in the following text.

First of all, the group was already familiar with Java and could therefore make better and more well-informed decisions during the code generation. In addition, Java was chosen due to it making it possible to write imperative code. However, it should also be possible to make object-oriented code in Spark. The decision was made after assessing the outcome of the interview. If the target language was chosen to be e.g. C, the group would have had to make a custom-made implementation of object-oriented programming, which would be much more time-consuming. Furthermore, many of the features of Java's semantics fit well with those that were desired in Spark. For example, the group chose that the language should have strict typing, although it is a bit less strict compared to Java because the types are limited to only two types; numbers and text. However, Spark is on the other hand a bit more strict when it comes to declaring a variable because you have to give the variable a value as soon as it is declared.

One of the ways to visualize the transition from one language to another is through a tombstone diagram [56]. Tombstone diagrams are referred to as having a resemblance of puzzles, they are made up of different pieces with different characteristics, additionally, the bordering pieces must be matching [57]. To get an overview of how programs in Spark language are compiled, a tombstone diagram is created that can be seen in figure 28.

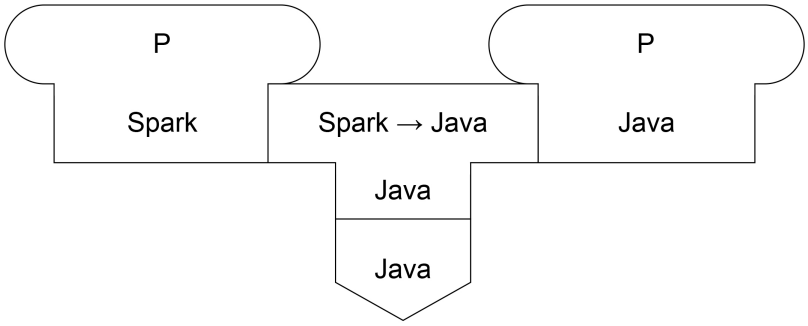


Figure 28: Sparks Tombstone Diagram

The tombstone diagram of Spark, in figure 28 is described as: a program written in the source language Spark, has a compiler that takes Spark to the target language Java, where the compiler itself is in language Java, and results in a program compiled to the language Java.

The syntax of Spark has some similarities with Java, but also some terminals that differ. The comparison can be seen in the Token Specification table below. In the table, all the possible terminal nodes of Spark have been translated into equivalent Java terminals. Though

Spark terminals	Java terminals
if	if
while	while
else	else
break	break
continue	continue
do	{
then	{
end	}
number	float
text	String
and	&&
or	||
not	!
+	+
-	-
<	<
>	>
=	==
>=	>=
<=	<=
%	%
/	/

Spark terminals	Java terminals
ID	ID
print	System.out.print( )
printLine	System.out.println( )
color	color
background	background
circle	ellipse
square	rect
triangle	triangle
darkRed	120, 0, 0
red	200, 0, 0
lightRed	255, 0, 0
darkGreen	0, 120, 0
green	0, 200, 0
lightGreen	0, 255, 0
darkBlue	0, 0, 120
blue	0, 0, 200
lightBlue	0, 0, 255
black	0, 0, 0
darkGrey	50, 50, 50
grey	120, 120, 120
lightGrey	200, 200, 200

Figure 29: Token specification

A feature that differs a lot from Java is the implementation of the library Processing. It was proven to be beneficial to have visual elements in a language for beginners according to the various interviews. The implementation of Processing will be introduced in section 9.6.

9.2 Implementation of ANTLR and the Run class

In this section, the implementation of the Run class will be explained. The Run class is responsible for the syntax and contextual analysis of the spark code. When the code analysis phase has been completed without any errors it should then call all the visitors that are needed for code generation.

9.2.1 Implementation of the Run class constructor

```
1  public class Run {
2       private static CodeGenerator codeGenerator =new CodeGenerator();;
3
4      //CharStream is an array of all symbols recognized by the BNF.
5      private CharStream codePointCharStream = null;
6      private bnfLexer lexer;
7      private bnfParser parser;
8
9      private bnfParser.SContext start;
10
11
12     public Run(String inputName) {
13
14
15         try {
16             //fills the CharStream array with the symbols gotten from the file code.txt in the
                   written path
17             codePointCharStream = CharStreams.fromPath(FileSystems.getDefault().getPath("Input/" +
                    inputName));
18         } catch (IOException e) { //it does not work, IOexception
19             throw new RuntimeException(e); //console writes error
20         }
21
22         //LEXER
23         //Instantiates a ANTLR gen bnfLexer object, with the CharStream array as input
24         lexer = new bnfLexer(codePointCharStream);
25
26         CommonTokenStream tokens = new CommonTokenStream(lexer);
27
28
29         //PARSER
30         //Instantiates a ANTLR gen bngParser object,
31         // with a CommonTokenStream as input made from the CharStream array from the lexer obj
32         parser = new bnfParser(tokens);
33
34         // List of children which is of the "s" rule
35         start = parser.s();
36     }
```

Listing 9: Implementation of the run class

At the beginning of the class, some global variables are declared. On line 2 occurs an instantiation of a codeGenerator object, which ensures that the Spark code is written to the output file, more details of how the class works can be seen in 9.7. As well as the codeGenerator class, the variables lexer, parser, and start also get declared.

The lexer starts with a ”try” on line 15. On this line, the program tries to generate a char stream (which is similar to an array) with a number of chars, from the code that the user has typed, which is generated after the code is compiled. However, if an error occurs and the code can not do this, an exception, on line 19, is thrown to handle the error. On line 24 a lexer is instantiated and the char stream is given as an input. On line 26 the code is setup in such a way that it makes ANTLR convert the chars into tokens. Then we have the parsing part of the code where tokens are built into a parse tree on line 32. And on line 35, the context of the first s in the parse tree is found and it is assigned to the variable start.

9.2.2 Implementation of the main function

```
1  public static void main(String[] args) {
2
3          //Instantiating Run object
4          Run codeLauncher = new Run("code.txt");
5
6          //Instantiating SparkToJavaVisitor and TopVisitor object
7          TopVisitor topVisitor = new TopVisitor(codeGenerator);
8          SparkToJavaVisitor sparkToJavaVisitor = new SparkToJavaVisitor(codeGenerator, codeLauncher
               );
9
10         try {
11             codeLauncher.start.accept(topVisitor);
12             codeLauncher.start.accept(sparkToJavaVisitor);
13
14             //closes the class in the output file
15             codeGenerator.indentation--;
16             codeGenerator.appendStrToFile("}\n");
17
18         }  catch (RuntimeException error){
19             System.out.println(error.getMessage());
20         }
21     }
22 }
```

Listing 10: Implementation of the main function

In the beginning of the main function, from lines 4 to 8, some objects are instantiated. First, a new Run object named codeLauncher is created, and given the string "code.txt" as input. The Run class is described above in section 9.2. Then a TopVisitor object, which gets the codeGenerator as an input and a SparkToJavaVisitor object, which gets the codeGenerator and the codeLauncher as input, is instantiated. On line 10 a "try" is entered and in line 11 the topVisitor is accepted by the start object. The sparkToJavaVisitor is then accepted by the start object on line 12. On line 15, the class of the output file is closed, by first changing the indentation by removing two spaces and then adding a curly bracket and a new line. If the topVisitor or the sparkToJavaVisitor throws an error, the program will stop and enter the catch on line 18. It then prints the message that it retrieves from the exception.

9.3 Implementation of visitors

This section will present the visitors that have been implemented as a part of the project and how they have been implemented. In section 9.3.1 visitor patterns will be described as a concept and how they are used, the section also presents a variety of the visitors which has been implemented in the project and why. Then follows section 9.3.2 which presents the top visitor, this visitor has a more special role which will be discussed in the section.

9.3.1 Visitor patterns

ANTLR can use both visitors and listeners to walk the parse trees generated by ANTLR. The group choose to use visitors for all the translations of the Spark source code to Java as this granted greater control over the flow of the code generation. In the implementation of code generation, the group utilized a recursive interpretation of the generated parse trees, because thorough error checks and ease of implementation were determined to be worth the longer compilation time [58].

Start Visitor

```
1  //This class runs through the different tokens in the TokenStream, to see which input has been
       giving.
2  //Then we are able to run functions based on what token it was, and modify/run code accordingly.
3  //s: extraLines line | extraLines globalStatement startFunction (drawFunction | empty)
4  public RuleNode visitS (bnfParser.SContext ctx) throws RuntimeException {
5      symbolTable.CreateScope();
6
7      //If the code uses visual features
8      if (ctx.startFunction() != null){
9
10         if(ctx.globalStatement().empty() == null) {
11             ctx.globalStatement().accept(this);
12         }
13
14         ctx.startFunction().accept(this);
15
16         if(ctx.drawFunction() != null) {
17             ctx.drawFunction().accept(this);
18         }
19
20     //if the code doesn't use visual features
21     } else {
22         //Increases the indentation in the output file, to put the code inside the constructor
23         codeGenerator.indentation++;
24
25         ctx.line().accept(this);
26
27         codeGenerator.indentation--;
28         codeGenerator.appendStrToFile("}\n");
29     }
30
31     symbolTable.RemoveScope(); // closes the scope such that it is no longer accessible
32     return ctx;
33 }
```

Listing 11: Implementation of the S rule visitor

The s rule is the top rule in the BNF and its visitor is responsible for opening the global scope and visiting its children. Spark has two ways of being written and this has to be taken into consideration when making the s visitor. The first way is if the user wants to have access to the graphical features of the language, the second way is if the user just wants to write a program that only makes use of the console. The function starts on line 4 with opening the global scope as this has to happen for both methods of writing Spark code. The condition on line 8 is where the visitor differentiates between the two methods of writing Spark code, it does this by checking if the s rule has a startFunction child as the startFunction is necessary for access to the graphical features. Line 8 to 19 is only run if the visitor has determined that the graphical features are needed, the first condition in this part is on line 8 where it is checked if the s nodes globalStatement child does not have an emptyNode child. This means that the globalStatement child has a child that should be visited, if this is the case line 10 will run and the globalStatement is visited. On line 14 the startFunction child is visited, and on line 16 it checks if the s visitor has a drawFunction child, and if this is true the drawFunction child is visited as well, this is seen on line 17. If the condition on line 6 is not true, the code from line 19 to 25 is run, which starts by increasing the indentation variable in the codeGenerator object, which is seen in section 9.7. Line 25 calls the line visitor on the S node's line child, and when this is complete the indentation variable in codeGenerator is decreased and a ";" and a new line is appended to the output file by calling the appendStrToFile function in the codeGenerator object. Lastly, on line 31 the global scope is closed by calling the RemoveScope function from the symbolTable object in the run object.

Assignment Visitor

```
1  String assignmentTypeCheck = "Number";
2  String assignmentName = "";
3
4  //assignment: ID '=' equation | ID '=' stringValue;
5  public RuleNode visitAssignment(bnfParser.AssignmentContext ctx) throws RuntimeException {
6
7
8      if (!symbolTable.VariableIsInScope(ctx.ID().getText())){
9          throw new variableNotInScopeException((RuleNode) ctx);
10     }
11
12
13
14     //We keep track of the key value, to see if the assignment type is wrong in the value visitor.
15     assignmentType = symbolTable.FindVariableValue(ctx.ID().getText()).getType();
16     assignmentName = ctx.ID().getText();
17
18     codeGenerator.appendStrToFile(ctx.ID().getText());
19     codeGenerator.appendStrToFile(" = ");
20     ctx.value().accept(this);
21     codeGenerator.appendStrToFile(";\n");
22
23     return ctx;
24
25 }
```

Listing 12: Implementation of the Assignment rule visitor

Assignment Visitors is responsible for translating the assignment command in Spark code into a Java equivalent assignment command, and error handling if the variable is not accessible from the current scope. The visitor starts off on line 8 by calling the VariableIsInScope, seen in section 9.4, function in the symbolTable object. This returns true if the variable is in the scope and if the variable that the Spark code is trying to declare is not found in the scope, the exception variableNotInScopeException is called, as seen on line 9. Afterward, on line 15, the function assigns the value to assignmentType by using the FindVariableValue, seen in section 9.4, in the symbolTable. The function finds the VarRef 9.4 object associated with the ID from the assignment, and with the VarRef object the value of the type variable is assigned to assignmentType. On line 15 The variable assignmentName gets assigned the value of the ID child of the assignment. On line 18 and 19 the ID child of the assignment and a "=" is added to the Output file with the appendStrToFile function, seen in Listing 24. On line 20 the value that the variable is assigned, is found by visiting the value child of the assignment, which is responsible for all error handling of the value and generating the code for the output file. Lastly on line 21, when the values have been visited a ";" and a new line is added to the output file.

Value Visitor and Extra Value Visitor

```
//value: ID extraValue | expression extraValue | stringCheckRule extraValue | '(' value ')'
    extraValue
public RuleNode visitValue(bnfParser.ValueContext ctx) throws RuntimeException {

    //Checks if the ID is type 'number' and assigning a String to that type.
    if(assignmentTypeCheck.equals("number") && ctx.getChild(0).getClass() == bnfParser.
        StringCheckRuleContext.class){
        throw new TypeConflictException(assignmentName,"number", "text");
    }

    //if the original ID is set to another ID, it checks if they align
    if(ctx.ID() != null){

        //Checks if the ID exist
        if (!symbolTable.VariableIsInScope(ctx.ID().getText())){
            throw new variableNotInScopeException(ctx.ID().getText());
        }
        //Checks if the ID is a number
        if(symbolTable.FindVariableValue(ctx.ID().getText()).declarationNode.getChild(0).getText()
             == "text" && assignmentTypeCheck == "number"){
            throw new TypeConflictException(assignmentName,"number", "text");
        }

        codeGenerator.appendStrToFile(ctx.ID().getText());

    } else if(ctx.expression() != null) {

        ctx.expression().accept(this);

    } else if(ctx.stringCheckRule() != null){

        codeGenerator.appendStrToFile(ctx.getChild(0).getText());

    } else {

        codeGenerator.appendStrToFile("(");
        ctx.value().accept(this);
        codeGenerator.appendStrToFile(")");

    }

    if(ctx.extraValue().empty() == null) {
        ctx.extraValue().accept(this);
    }
    return ctx;
}

//extraValue: '+' value | operator value | empty
public RuleNode visitExtraValue(bnfParser.ExtraValueContext ctx) throws RuntimeException {

    //Checks if the ID is type 'text' and using a wrong operator for that type.
    if(assignmentType.equals("text") && ctx.getChild(0).getClass() == bnfParser.OperatorContext.
        class){
        throw new TypeConflictException(assignmentName,"text", "number");
    }

    codeGenerator.appendStrToFile(" " + ctx.getChild(0).getText() + " ");

    ctx.value().accept(this);

    return ctx;
}
```

Listing 13: Implementation of the value and extraValue rule visitor

```
assignment: ID '=' value;

value: ID extraValue | expression extraValue | stringCheckRule extraValue | '(' value ')'
    extraValue;

extraValue: '+' value | operator value | empty;
```

Listing 14: A chosen selection of the BNF

The Value and ExstraValue visitors are a bit special in their goal compared to the other visitors. When the user wants to assign a variable to a new value, the BNF does not know what type the variable is, as that was declared outside of that rule's scope. That means that the BNF needs to be written in a way that is uncaring of what the assigned type will be, so the compiler can be in charge of type checking. This can be seen in listing 14 when the group uses the value rule in the assignment which can represent both numbers and text, or both at the same time. When the compiler then runs through the tree, it will know which type the variable is that is being reassigned from the symbol table, and can then check for type errors.

To go into a bit more depth, there are two errors that can happen: if a number is assigned any kind of text, or if a mathematical operator other than '+' is used on a text. This is because a text can be assigned a number, as that would just make it a String containing that number. So the compiler has to figure out which options in the Value and ExtraValue rule can cause any of the two type errors. The group figured out that there are three points in the BNF that can potentially cause an error.
1. The Value rule option three: where the variable being reassigned is a number and the value it is being assigned is a text.
2. The Value rule option one: same as the problem before, but instead of straight text, it is through a variable.
3. The ExtraValue rule option two: where the variable being reassigned is a text and the operator is a mathematical operator other than '+'.

With all that in mind, it is time to go through the two visitors in listing 13. The first thing that the Value visitor checks on line 5 are problem point 1, which throws an error if the condition is met. Next is line 10 which checks if it is the first option in the Value rule in the parse tree, where the value is a variable. It then runs through two checks to see if the variable is in the symbol table and does not match problem point 2. If any of the checks conditions are met then an error is thrown, otherwise it appends the variable to the output file. Lines 23 to 37 just run through and check if it is any of the other three options for the Value rule in the parse tree. Lastly, the Value visitor checks if there are any other values attached through an operator to the assignment and if there are it calls the ExtraValue visitor. The ExtraValue visitor which starts on line 46 in listing 13 immediately checks for problem point 3 on line 49 and throws an error if the condition is met. After that it is quite simple, it just appends the operator used and visits the Value visitor again.

StartFunction Visitor

```
//startFunction: 'on' 'start' block forcedLineChange globalStatement
public RuleNode visitStartFunction(bnfParser.StartFunctionContext ctx) throws RuntimeException {

    codeGenerator.appendStrToFile("public void setup() {\n");

    symbolTable.CreateScope(); // creates a new scope
    codeGenerator.indentation++;

    codeGenerator.appendStrToFile("frameRate(60); //Sets the framerate.\n");

    ctx.block().line().accept(this);

    codeGenerator.indentation--;
    codeGenerator.appendStrToFile("}\n\n");
    symbolTable.RemoveScope(); // closes the scope such that it is no longer accessible

    if(ctx.globalStatement().empty() == null) {
        ctx.globalStatement().accept(this);
    }

    return ctx;
}
```

Listing 15: Implementation of the startFunction rule visitor

The StartFunction rule visitor represents the "on start" function. It is used when using visual elements, and anything coded into the "on start" function is run only once at the launch of the program. The "on start" function has its own scope, the first thing the visitor will do is append a function declaration to the output file and create a scope for the function. The library used to create the window and the visual elements have a prerequisite of setting the frame rate of the said window in the start function, also called setup in the library, which is why line 9 is necessary. Afterward, the visitor uses the block rule visitor to append all the code inside the "on start" function, and lastly, the visitor closes the scope.

To have the option of creating global variables after "on start" function, it needs to have lines 17 to 19. These lines check if there are any GlobalStatement after the "on start" function, and then visit the GlobalStatement visitor if there are.

GlobalStatement Visitor

```
1  //globalStatement: declaration forcedLineChange globalStatement | empty
2  public RuleNode visitGlobalStatement(bnfParser.GlobalStatementContext ctx) throws RuntimeException
       {
3
4      //As the scope has already been open in visitS,
5      //and as that is the outmost scope, as in the global scope,
6      //will we just use that scope instead of opening a new one.
7      ctx.declaration().accept(this);
8
9      //As there can be multiple global statements after each other,
10     //it is then possible for globalStatement to call itself.
11     //Though it first checks if the child is empty before, so we don't get at null error.
12     if(ctx.globalStatement().empty() == null) {
13         ctx.globalStatement().accept(this);
14     }
15
16     return ctx;
17 }
```

Listing 16: Implementation of the globalStatement rule visitor

The visitor for the GlobalStatement rule, which can be seen in Listing 16 is one of the more simple visitors, but it is presented as it is only used when the visual aspect of Spark is used. The user has to use the functions "on start" and "on eachFrame" to use the visual elements. This means that to have a working variable in both functions, it has to be declared outside of each function. In other words, it has to be a global variable. This is not required when the user is not using the "on start" or "on eachFrame" as the code will be written within a single function, i.e. the same outmost scope.

The GlobalStatement rule only has two functionalities, which are to call declaration and check to see if there are more GlobalStatements.

Figure Visitor

```
1  //figure: 'circle' parameter parameter parameter parameter | 'square' parameter parameter
       parameter parameter | 'triangle' parameter parameter parameter parameter parameter parameter
2  public RuleNode visitFigure(bnfParser.FigureContext ctx) {
3
4      switch (ctx.getChild(0).getText()) {
5          case "circle" -> codeGenerator.appendStrToFile("ellipse(");
6          case "square" -> codeGenerator.appendStrToFile("rect(");
7          case "triangle" -> codeGenerator.appendStrToFile("triangle(");
8          default -> System.out.println("NULL" + ctx.getChild(0).getText());
9      }
10
11     //loops through all the parameters.
12     for (int i = 1; i < ctx.getChildCount(); i++){
13         ctx.getChild(i).accept(this);
14         if(i != ctx.getChildCount() - 1) {
15             codeGenerator.appendStrToFile(", ");
16         }
17     }
18
19     codeGenerator.appendStrToFile(");\n");
20
21     return ctx;
22 }
```

Listing 17: Implementation of the figure rule visitor

The figure visitor is also quite simple compared to some other visitors. The reason why the group chose to present this visitor, is because it handles some of the keywords the group implemented from the processing library. It starts on line 4 in listing 17 with a switch case to determine which option is used in the figure rule and append the right keyword to the output file. As all of these options need some parameters, but not necessarily the same amount, which means the next step is then to loop through each of these parameters and visit their visitor, which happens on line 12. Lastly, on line 19 it appends the end of the command to the output file.

9.3.2 The topVisitor class

The topVisitor class works in much the same way as the primary visitor class (SparkToJavaVisitor). The difference is that the topVisitor is run first and goes through the parse tree looking for specific things, whereas the main visitor runs through everything to compile it. In this case, the topVisitor checks for two things before the main visitor is run. One of the things is common errors the user could have made, that way the program could throw the error and stop the compiling, so the user does not have to wait for the primary visitor class to finish before editing the code. The second thing the topVisitor checks is if the user has implemented any kind of visual features into their code, that way the output file can be generated with the right libraries and code structure.

```
1  //s: extraLines line | extraLines globalStatement startFunction (drawFunction | empty)
2  public Object visitS (bnfParser.SContext ctx){
3
4      //If the code has an 'on start' function
5      if(ctx.startFunction() != null){
6          ctx.startFunction().accept(this);
7
8          //As the 'on eachFrame' function isn't required, it doesn't have to run
9          if(ctx.drawFunction() != null) {
10             ctx.drawFunction().accept(this);
11         }
12         codeGenerator.createFile(false); //false indicates processing is used, and it should not
               create a standard file
13
14     //If the code doesn't have an 'on start' function
15     } else {
16         ctx.line().accept(this);
17         codeGenerator.createFile(true); //true indicates that a standard file should be created
18     }
19
20     //In case any visual features has been used, but the 'on start' function isn't implemented
21     if((visualStatement && !startFunction) || (drawFunction && !startFunction)){
22         throw new ProcessingConflictException(ctx);
23     }
24     return this;
25 }
```

Listing 18: Implementation of the s rule visitor in the top visitor

The s visitor shown in the code above is the function that is initially run by the topVisitor class. The first thing it does in line 5 in listing 18 is to check if there is an 'on start' function, as that will determine which option in the s rule it will take, as well as how the output file's structure is going to be. This can be seen on line 12 and line 17, where the function that generates the file is run. The boolean parameter determines if the output file is to generate a standard Java file or to include extra libraries. The three '.accept' lines on line 6, 10, and 17 runs through the parse tree for any node that has a matching visitor in this class. One of those visitors is the errorNode visitor, which handles some of the mistakes the user can make, which the group thought are prevalent. Through the '.accept' line can the program check if the user uses any visual features, and through the check on line 21 it can throw an exception if the 'on start' function is not present as well.

```
1  public Object visitErrorNode(ErrorNode node){
2
3      // 59 refers to the Errorsymbol terminal in the bnf
4      // if the node is not an Errorsymbol
5      if(node.getSymbol().getType() != 59){
6          return this;
7      }
8
9      //Switch case for which error it is
10     switch (node.getParent().getClass().getName()){
11         case "Main.AntlrGenerated.bnfParser$DeclarationContext":
12             throw new unexpectedDeclarationException(node);
13         case "Main.AntlrGenerated.bnfParser$ConditionContext":
14             System.out.println("Needs to throw error, has not been implemented yet.");
15             break;
16         case "Main.AntlrGenerated.bnfParser$AssignmentContext":
17             System.out.println("Needs to throw error, has not been implemented yet.");
18             break;
19     }
20
21     return this;
22 }
```

Listing 19: Implementation of the Error rule visitor in the top visitor

The errorNode visitor shown above is quite simple in its functionality. It does two checks and throws an exception depending on those checks. The first check is on line 5 in listing 19, which checks if the error node contains any of the error symbols in the ErrorSymbol rule in the BNF in the listing 5. The number 59 is a way to check which

rule is used, as the ErrorSymbol rule is the 59th rule in the BNF. There are potentially better ways of checking this as this number changes if the BNF is changed, but unfortunately the group was not able to find it. If the error symbol is not one of the symbols in the ErrorSymbol, ANTLR will just throw its own error as the ANTLR does it by default, if the program does not override it. The switch case on line 10 checks what parent node the node has, as the program will then be able to throw the right exception, based on the context of the mistake the user made.

9.4 Implementation of the Symbol Table

The function of a symbol table in a compiler has previously been discussed in section 8.2.1, where it was concluded that the group wanted to implement the symbol table as a stack of HashMaps.

The group's implementation of the Symbol table can be seen in the code below:

```
1  public class SymbolTable {
2      private List<HashMap<String,VarRef>> ScopeStack; // A list of maps, which uses string as keys
           and a VarRefs as value
3
4      public void AddToScope(VarRef varRef){ // adds a scope to the list
5          if (VariableIsInScope(varRef.name)){
6              throw new IDAlreadyInScopeException(varRef, FindVariableValue(varRef.name)); // new
                   and old variable
7          }
8          ScopeStack.get(ScopeStack.size()-1).put(varRef.name, varRef);
9      }
10
11     public void RemoveScope(){ // removes a scope from the list
12         ScopeStack.remove(ScopeStack.size()-1);
13     }
14
15     public void CreateScope(){ // creates a new empty scope
16         ScopeStack.add(new HashMap<String,VarRef>());
17     }
18
19     public boolean VariableIsInScope(String ID){
20         for (int i = ScopeStack.size()-1; i >= 0; i--){ // begins at the top of the stack.
21             if (ScopeStack.get(i).containsKey(ID)){
22                 return true; // returns true if the variable is in the Scope
23             }
24         }
25         return  false;
26     }
27
28
29     public VarRef FindVariableValue(String ID){
30         for (int i = ScopeStack.size()-1; i >= 0; i--){ // begins at the top of the stack.
31             if (ScopeStack.get(i).containsKey(ID)){
32                 return ScopeStack.get(i).get(ID); // returns VarRef object associated with the ID
33             }
34         }
35         return  null;
36     }
37 }
```

Listing 20: Implementation of the symboltable

The symbol table handles scopes of variables through some different functions. A **CreateScope()** function can be seen on line 15, this function creates a new scope HashMap and adds it to the ScopeStack list. There is also a **RemoveScope()** function which can be seen on line 11, this function removes scopes from the list ScopeStack if necessary.

The scope functions are called in the different visitors. A scope is either removed when leaving a block or when all the code has been generated. It must be possible to use global variables in the whole program, and they are therefore handled in the start visitor.

The**AddToScope()** function, which can be seen on line 4, adds a newly created scope to the list ScopeStack. In the function, it is first checked whether the variable wanted to be added is already in the scope. This is done

through the VariableIsInScope() function. If that function returns true the code will throw an error and not add the variable to the scope. However, if it returns false the variable will be added to the correct scope.

The **VariableIsInScope()** function, which can be seen on line 19, returns a boolean based on if the variable input it gets matches up with an existing variable in the scope.

The function **FindVariableValue()** which can be seen on line 29, has the same functionality as the VariableIsInScope function. However, it returns the VarRef instead.

Implementation of the VarRef class
The group decided on creating a variable reference class that could be used to help maintain the symbol table. The implementation of the VarRef class can be seen in the listing below:

```
1  // variable reference to be used in the symbol table
2  public class VarRef {
3      String name;
4      String type;
5      bnfParser.DeclarationContext declarationNode;
6
7      public VarRef(String name, String type, bnfParser.DeclarationContext declarationNode) {
8          this.name = name;
9          this.type = type;
10         this.declarationNode = declarationNode;
11     }
12
13     public String getName() {
14         return name;
15     }
16
17     public String getType() {
18         return type;
19     }
20
21     public bnfParser.DeclarationContext getDeclarationNode() {
22         return declarationNode;
23     }
24 }
```

Listing 21: Implementation of VarRef class

The VarRef class embodies and returns the three values that the symbol table needs for a symbol table entry. The values are name, type, and declarationNode which the classes access through the three get-function: getName(), getType(), and getDeclarationNode().

9.5 Implementation of Errors and Error messages

The group has implemented a variety of exceptions to handle errors and deliver error messages. The decision to implement error messages was made to appeal to making the language helpful for beginners, which is described as a part of the problem statement in section 4.4.

One example of the Errors and custom Error messages the group has implemented for the language is a UnexpectedDeclarationException that can be seen below:

```
1  //Exception used to catch if the user is trying to declare a variable incorrectly
2  public class UnexpectedDeclarationException extends RuntimeException{
3
4      private ErrorNode node;
5
6      public unexpectedDeclarationException(ErrorNode node) {
7          this.node = node;
8      }
9
10     @Override
11     public String getMessage() {
12         return "On line: "+node.getSymbol().getLine() +" you tried to \"" +node.getParent().
               getText() +"\". "+ findAlternative();
13     }
14
15     public String findAlternative(){
16
17         switch (node.getSymbol().getText()){
18             case "is":
19             case "==":
20                 return "You should use one a single \"=\" to assign the value of the variable" ;
21             case "/=":
22             case "*=":
23             case "-=":
24             case "+=":
25             case "%=":
26                 return "This command is not possible, because it is a recursive definition";
27             default:
28                 return "The "+ node.getText() +" is not a know declaration";
29         }
30     }
31
32 }
```

Listing 22: A part of the UnexpectedDeclarationException:

The showcased part of the class UnexpectedDeclarationException shows the general structure of how each of the exceptions implemented is built. If the UnexpectedDeclarationException is run, there has been made a mistake when declaring a variable.

The point of these exception classes is that when the user makes an error, they will receive an error message that tells them what is wrong and how they can solve the error. This specific class therefore contains the constructor UnexpectedDeclarationException() which can be seen on line 6, this function is given an ErrorNode as an input for further use in this class. In other words, this function gets the declaration node that did not match the required value according to the BNF rules.

The function getMessage() returns an error message string. The string is the error message the user will get if their code throws an exception that refers to this error. The string tells the user which node (what they typed that was wrong), this is done through the node.getSymbol().getLine() function and also the node.getParent() function

The alternative solution is provided by the function findAlternative() which can be seen on line 15. This function returns a String based on a switch case, the switch case is determined by what symbol was gotten as the ErrorNode. In the switch case, the group has inserted typical mistakes a user could have tried to write to try declaring a variable. An example of the switch case is if the ErrorNode provided is the text "is", it will then go through the first case in the switch case which returns the message "You should use a single "=" to assign the value of the variable". This way, Spark directly tells whether the tried method can be used as a declaration or not.

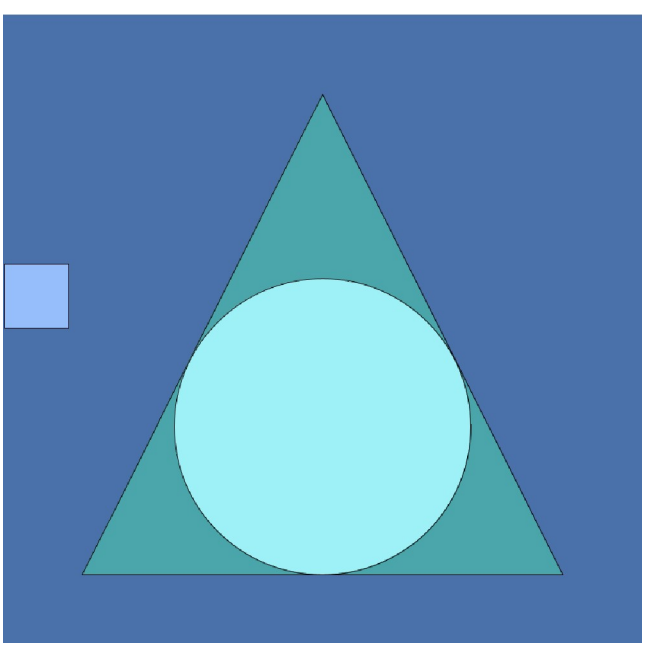
The other Exceptions in the code are:

- IDAIreadyInScopeException
- ProcessingConflictException
- TypeConflictException
- VariableNotInScopeException
- AssignmentException
- LoopConflictException

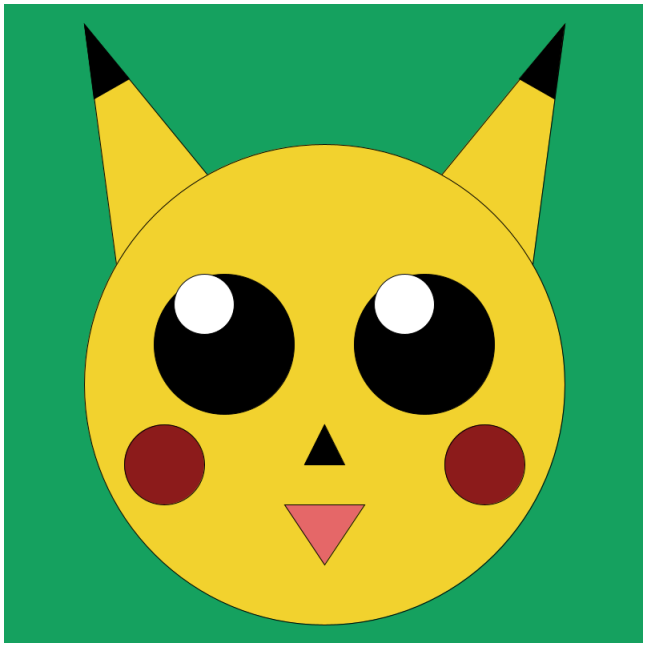
9.6 Implementation of Processing

One of the goals the group had with Spark is that it should have visual features to help with visualizing output. That way the user can set up a square to move with a variable or under certain conditions, and have visual feedback for that variable or when that condition is met.

There are multiple ways for this to be done, but one of the ways the group was familiar with and had already researched was processing, as seen in section 5.3. The group is able to import the processing library when it is needed and leave it out when not. That way the user does not have to think about the libraries and implement unnecessary ones.



(a) A square, triangle, and a circle in different colors.



(b) A drawing of Pikachu, made using Spark.

Figure 31: An example output from using the visual features.

Above is an example of two programs that can be made with Spark when it has access to the processing library. Which meets the goal in the problem statement and MoSCoW of having a visual output.

One issue the group encountered using processing, is that processing uses specific functions when running. For example, it has a 'draw' function that runs every frame of the window, to make it able to move and change the elements in the window. This issue was solved by creating two unique functions 'on start' and 'on eachFrame' which represent when the program just launched and for each frame update on the window respectively. The 'on start' function is the only one required to have when using the visual features and is not required at all when not. Where the 'on eachFrame' function is optional. There are different ways to deal with this problem of having unique functions with predetermined execution points, which the group discussed and experimented with, but eventually went with the solution mentioned. One of the discussed solutions was to only have the 'on eachFrame' function and let 'on start' be the default function. In other words, whatever the user wrote outside of the 'on eachFrame' function would automatically be put into 'on start' to make it easier. One of the problems with that was declared variables and scope, which the group decided was too big of a problem to continue with that solution.

9.7 Implementation of the code generator

The code generator class consists of two functions: createFile() and appendStrToFile().

createFile()

```
1  public void createFile(Boolean standardFile) {
2      file = new File("Output/" + FileName + ".java");
3      boolean NewFileCreated = false;
4      String programIntroText;
5      try {
6
7          // create a new file with name specified
8          // by the file object
9          NewFileCreated = file.createNewFile();
10         if (NewFileCreated) {
11             System.out.println("New Java File is created.");
12         } else {
13             System.out.println("The file already exists.");
14         }
15     } catch (Exception e) {
16         e.getStackTrace();
17     }
18
19     if (standardFile) {
20         programIntroText ="public class " + FileName + " { \n" +
21                 "\tpublic static void main(String[] args) { \n" +
22                 "\t\t" + FileName + " runFile = new " + FileName + "();\n" +
23                 "\t}\n" +
24                 "\t\n" +
25                 "\tpublic " + FileName + "(){\n";
26      }
27     else {
28         programIntroText = "import processing.core.PApplet;\n\n" +
29                   "public class " + FileName + " extends PApplet { \n" +
30
31                   "\tpublic static void main(String[] args) { \n" +
32                   "\t\tString[] processingArgs = {\"MySketch\"};\n" +
33                   "\t\t" + FileName + " javaFile = new " + FileName + "();\n" +
34                   "\t\tPApplet.runSketch(processingArgs, javaFile);\n" +
35                   "\t}\n" +
36
37                   "\tpublic void settings() {\n" +
38                   "\t\tsize(800, 800); //Sets the window size.\n" +
39                   "\t}\n\n";
40     }
41     try {
42         // Creates a Writer using FileWriter
43         FileWriter output = new FileWriter("Output/" + FileName + ".java", false);
44
45         // Writes the programIntroText to file
46         output.write(programIntroText);
47         System.out.println("Data is written to the file.");
48
49         // Closes the writer
50         output.close();
51     }
52     catch (Exception e) {
53         e.getStackTrace();
54     }
55 }
```

Listing 23: Implementation of the function createfile in the code generator class

The purpose of the createFile() function, which can be seen in listing 23 is to check whether there already exists a Java file ready for the output or if there has to be created one. The function keeps track of this by using the Boolean "NewFileCreated", this Boolean is by default false but gets assigned true or false based on if there already exists an output Java file, it then prints the result to the terminal. This can be seen on lines 9 to 14 in listing 23.

When the Java file is ready there has to be printed some intro text to be put into the Java file, this has to be done manually since the Spark language does not take libraries or declarations of functions into consideration. If the user writes keywords in Spark linked to visual elements the Java file has to contain the processing library and additional setup to be able to run the code as intended in Spark. However this library is only needed if the input

code contains visual elements, therefore there are two options for the intro text to the Java file, one including the processing library and the additional setup and one without it. These different intro text strings and their handling of them can be seen on lines 19 to 40 in listing 23. It can also be seen that these decisions on which of these are to be used are determined by the Boolean "standardFile". The value of this Boolean is given through the TopVisitor class when it runs the visitS function.

Lastly, the function creates a FileWriter object to the Java output file and then writes in the chosen intro text. This can be seen on lines 42 to 50 in listing 23.

appendStrToFile()

```
1  private boolean needIndentation = true;
2  public int indentation = 1;
3
4  public void appendStrToFile(String str) {
5      // Try block to check for exceptions
6      try {
7
8          // Open given file in append mode by creating an
9          // object of BufferedWriter class
10         FileWriter out = new FileWriter("Output/" + FileName + ".java", true);
11
12         if(needIndentation){
13             for (int i = 0; i < indentation; i++) {
14                 out.write("\t");
15             }
16             needIndentation = false;
17         }
18
19         // Writing on output stream
20         out.write(str);
21
22         //If the next line add indentation next line
23         if(str.endsWith("\n")) {
24             needIndentation = true;
25         }
26
27         // Closing the connection
28         out.close();
29
30     } catch (Exception e) {
31         System.out.println("error");
32     }
33 }
```

Listing 24: Implementation of the function appendStrToFile in the code generator class

The purpose of the appendStrToFile() function, which can be seen above in listing 24, is to append all the strings given by the SparkToJavaVisitor class to the output file correctly. Firstly a FileWriter object is created which then is attached to the Java output file, this can be seen on line 10 in listing 24. Then The function takes the need and use of indentation into consideration through an if statement which can be seen in lines 12 to 17 in listing 24, if an indentation is needed the FileWriter prints an indentation to the output file. The Indentation is set to true for the next line if the string given ends with a new line, this can be seen on line 23 in the if statement.

On line 20 in the listing 24, the Filewriter writes the given string to the Java output file, the string is given by the SparkToJavaVisitor which is a string translated from Spark to Java. Lastly, the FileWriter is closed on line 28.

10 Testing

When entering this section of the report, the product of this project is on the edge of being finalized. The first iteration of the implementation of the Spark compiler has been finalized and now it is time to evaluate and test the current version of the compiler. Through testing, errors within the product can be found, solved, and prevented for future users.

This section contains testing of the compiler in section 10, the tests conducted on the Spark compiler consists of unit tests 10.1, integration tests 10.2 and acceptance tests 10.3.

10.1 Unit testing

A unit test is used to focus on a single unit such as one class or function to test within the software [59]. The group has decided to showcase a couple of unit tests done in the Visitor class and in the codeGenerator class.

10.1.1 Testing the visitor

Before the test of the single unit can be done, there has to be setup up a parser and input for a TokenStream. The setup done before each unit test can be seen in listing 25. Firstly a SparkToJavaVisitor is instantiated. Then the method findParserThroughString() creates a bnfLexer with the CharStream for the given input string, the given input string is done through the specific test methods.

The group has tested the methods in the SparkToJavaVisitor class by using the library *import org.junit.jupiter.api.Test;*, this library gives access to the method assertEquals() which is used throughout the testing of the SparkToJavaVisitor class.

```
1 @BeforeEach
2 public void setup() throws IOException {
3     myVistor = new MyVistor(codeGenerator, run);
4 }
5 public bnfParser findParserThroughString(String inputString) throws IOException {
6     bnfLexer lexer = new bnfLexer((CharStreams.fromString(inputString)));
7     CommonTokenStream tokens = new CommonTokenStream((lexer));
8     return new bnfParser(tokens);
9 }
```

Listing 25: Setup for unit testing

```
1  @Test
2  void visitLine() throws IOException {
3
4      bnfParser parser = findParserThroughString("number a = 2\n");
5
6      run.symbolTable.CreateScope();
7      assertEquals("LineContext", myVistor.visitLine(parser.line()).getClass().getSimpleName());
8      run.symbolTable.RemoveScope();
9
10 }
```

Listing 26: Test of visitLine

The first unit test is testing the visitLine method, this is done by giving the bnfParser the input string it has to test. This can be seen on line 4 in the listing 26, where the parser is given the string "number a = 2 $\backslash n$".

The visitLine method requires a scope in the symbolTable to be created for the input string, this is done on line 6 and then the scope is removed again on line 8 after the test is done. To test if the parser actually visits the Line an assertEquals method is called, where the expected output is "LineContext" and the actual output is gotten through the visitLine method in the SparkToJavaVisitor class. If the assertEquals reports an error then the visitLine method is not visiting the correct Node from the input.

```
@Test
void visitDeclaration() throws IOException {

    bnfParser parser = findParserThroughString("number a = 2\n");

    run.symbolTable.CreateScope();
    assertEquals("DeclarationContext", myVistor.visitDeclaration(parser.declaration()).getClass().
        getSimpleName());
    run.symbolTable.RemoveScope();

}
```

Listing 27: Test of visitDeclaration

The group created another unit test for the visitDeclaration method in the SparkToJavaVisitor class. The setup and course of action are exactly the same as shown in the visitLine test method seen in listing 26. The difference in the visitDeclaration test method is that we expect the output to be "DeclarationContext" since we are visiting the declaration node in the inputs parse tree. The visit method can be seen in listing 27.

10.1.2 Testing the codeGenerator

The codeGenerator class is an important part of the compiler since it has the responsibility to generate the Java output file and append the correct strings based on the input in the language Spark.

```
@Test
void appendStrToFile() throws IOException{

    String testString = "This is a test";

    codeGenerator.appendStrToFile(testString);
    Scanner scanner = new Scanner(new File("Output/JavaFile.java"));

    assertEquals(true, scanner.nextLine().endsWith(testString));

}
```

Listing 28: Test of visitDeclaration

The appendStrToFile method has been tested which can be seen in listing 28, the codeGenerator is given a string to append to the output file, then a simple Scanner is instantiated to read the output file. Then the assertEquals method is called to compare if the data the scanner got from the output file matches the string appended through the codeGenerator. However since the appendStrToFile method handles indentation which we want to ignore in this case through assertEquals, the testString is called through the endsWith() method in the codeGenerator which ignores the indentation prior to the string.

10.1.3 Testing of error handling

```
@Test
void getMessage() {

    String expectedString = "The variable "+"\""+"IDtest"+"\""+" was declared of type "+"\""+"
        number"+"\""+" and was assigned to type "+"\""+"text"+"\""+" This is not possible.";

    typeConflictException = new TypeConflictException("IDtest", "number", "text");
    assertEquals(expectedString, typeConflictException.getMessage());
}
```

Listing 29: Test of TypeConflictException getMessage

A unit test of error handling was also made, the group decided to create a unit test on the class TypeConflictException, specifically the method getMessage is interesting to test in this class. This test was like the other unit tests conducted through the assertEquals call which can be seen in the listing 29.

10.2 Integration testing

An integration test is where multiple components or units are combined and tested together. An integration test's main purpose is therefore to test interactions between multiple integrated components [60]. The group performed the integration test by giving the compiler certain input in the language Spark, and then the output is compared with the expected result based on knowledge of the Java language.

Below in figure 32 is an integration test of the while loop statement in Spark can be seen. Listing 30 shows the input for a simple while loop written in the Spark language. In listing 31 the output can be seen after the compiler has compiled the input from Spark to Java, which is the target language of the compiler. When comparing the input and the Java code compiled in the output, it can be concluded that the output looks correct and the Java code written is correct. The compiler reads the while loop in the input on line 4 in listing 30, which is written correctly according to the BNF rules, then translates the while loop to the Java syntax and outputs it correctly on line 9 in listing 31 in the output file. The integration test for the while loop statement is deemed successful.

```
1 number a = 2
2 number b = 4
3
4 while a < b do
5     a = a + 1
6     break
7 end
```

Listing (30) Integration test of while loop Input

```
1  public class JavaFile {
2      public static void main(String[]
           args) {
3          JavaFile runFile = new JavaFile
               ();
4      }
5
6      public JavaFile(){
7          float a = 2;
8          float b = 4;
9          while(a < b) {
10             a = a + 1;
11             break;
12         }
13     }
14 }
```

Listing (31) Integration test of while loop Output

Figure 32: Integration test of the while loop statement

Another integration test was made to test a variety of the visual elements implemented in Spark's language. This integration test which can be seen below in figure 33, was performed the same way as the integration test for while loop statements seen in figure 32.

The input given, which can be seen in listing 32, in this test, it had to create a square with a given color and position, a text variable is also declared and printed. The output compiled to the target language Java seen in listing 33 looks correct and matches what was expected from the input written in Spark.

```
text squareText = "The square is blue"

on start then
    color 0 0 255
    square 0 0 80 80
    print squareText
end
```

Listing (32) Integration test of visual elements Input

```
import processing.core.PApplet;

public class JavaFile extends PApplet {
    public static void main(String[]
        args) {
        String[] processingArgs = {"
            MySketch"};
        JavaFile javaFile = new
            JavaFile();
        PApplet.runSketch(
            processingArgs, javaFile);
    }
    public void settings() {
        size(800, 800); //Sets the
            window size.
    }

    String squareText = "The square is
        blue";
    public void setup() {
        frameRate(60); //Sets the
            framerate.
        fill(0, 0, 255);
        rect(0, 0, 80, 80);
        System.out.print(squareText);
    }

}
```

Listing (33) Integration test of visual elements Output

Figure 33: Integration test of the statements linkted to visual elements

10.3 Acceptance testing

An acceptance test is the last phase of testing that the compiler can go through before being declared complete [61]. The purpose of an acceptance test is to test the compiler as a whole. The group has chosen to do this by simply giving the compiler an input file that should cover multiple if not all of the functionalities that Spark contains. Since Spark can produce visual elements it has been decided that two acceptance tests should be made, one focusing on testing visual elements and one focusing on more basic keywords and functions.

10.3.1 Acceptance test of visual elements in Spark

The input file for the first acceptance test of the visual elements can be seen in *Appendix E* figure 43. The source code given to the compiler is written in the language Spark which translates and compiles that to Java which can be seen below in listing 34. When looking at the compiled Java code seen in listing 34, the input code has been compiled correctly because the output is correct according to the Java syntax and use of language. When compiling the new Java file, the code runs correctly and does what is intended through the source code. Therefore this acceptance test has been deemed complete and successful.

```
import processing.core.PApplet;

public class JavaFile extends PApplet {
        public static void main(String[] args) {
                String[] processingArgs = {"MySketch"};
                JavaFile javaFile = new JavaFile();
                PApplet.runSketch(processingArgs, javaFile);
        }
        public void settings() {
                size(800, 800); //Sets the window size.
        }

        public void setup() {
                frameRate(60); //Sets the framerate.
        }

        float top = 0;
        float left = 0;
        float d = 0;
        float e = 0;
        float size = 716;
        float speed = 3;
        public void draw() {
                if((e <= size && left == 0 && top == 0) || e <= 0) {
                        e = e + speed;
                        left = 0;
                        if(e >= size) {
                                left = 1;
                        }
                        System.out.println("right");
                }
                else if ((d <= size && top == 0 && left == 1) || (d <= 0)) {
                        d = d + speed;
                        top = 0;
                        if(d >= size) {
                                top = 1;
                        }
                        System.out.println("down");
                }
                else if ((e > 0 && left == 1 && top == 1)) {
                        e = e - speed;
                        left = 1;
                        System.out.println("left");
                }
                else if ((d > 0 && left == 0 && top == 1)) {
                        d = d - speed;
                        top = 1;
                        System.out.println("up");
                }
                background(0, 150, 136);
                fill(96, 125, 139);
                rect(e, d, 80, 80);
                fill(76, 175, 80);
                triangle(400, 100, 100, 700, 700, 700);
                fill(139, 195, 74);
                ellipse(400, 515, 370, 370);
        }
}
```

Listing 34: Output from the input file "ColoredTriangle.txt"

10.3.2 Acceptance test of the language Spark

As mentioned prior the second acceptance test should be focused on testing that the if statements are working and translated correctly. This test is conducted the same way as the previous acceptance test, simply by giving the compiler an input file and then the Java output file is checked. The input file given in this test can be seen in *Appendix E* figure 44. The output gotten from the source code can be seen below in listing 35.

Through the group's knowledge of Java it has been concluded that this acceptance test is successful due to the output seen in listing 35 is correctly translated to Java from the input file written in Spark. The output can also run without any errors occurring and prints the correct output in the terminal.

```
1  public class JavaFile {
2          public static void main(String[] args) {
3                  JavaFile runFile = new JavaFile();
4          }
5
6          public JavaFile(){
7                  float i = 0;
8                  float input = 20;
9                  while(i < input) {
10                         if(i % 3 == 0 && i % 5 == 0) {
11                                 System.out.println("FizzBuzz");
12                         }
13                         else if (i % 3 == 0) {
14                                 System.out.println("Fizz");
15                         }
16                         else if (i % 5 == 0) {
17                                 System.out.println("Buzz");
18                         }
19                         else {
20                                 System.out.println(i);
21                         }
22                         i = i + 1;
23                 }
24         }
25 }
```

Listing 35: Output from the input file "FizzBuzz.txt"

11 Discussion

This section contains the discussion of the thoughts and work progress that resulted in the final product. Firstly the products fulfillment of the Moscow requirements created throughout the project will be discussed in section 11.1. Afterwards a discussion debating how the syntax and keywords have been made to become more intuitive and memorable will be seen in section 11.2. Then the use of visual elements to enhance learning in the product will be discussed in section 11.3. And lastly, there will be a discussion in section 11.4 presenting what a further development plan for the product could look like.

11.1 Fulfillment of the MoSCoW requirements

The requirements for the Spark programming language have been presented in the MoSCoW seen in section 6. Throughout the project, the "must have" have been prioritized to be completed first and they have all been fulfilled. The "must have" requirement for the project is: "Block indication", "Basic operators", "Imperative programming", "If statements", "While loops", "Output", "Variables (type restriction: strict)", "New line, statement ends", and "Symbol table/scope". A description of these can be read in section 6.2.1 in table 2.

The requirement of "Block indication" has been fulfilled as Spark does not use curly brackets as block indicators but uses "then" and "end", or "do" and "end". However, the best solution mostly recommended by the interviewees was forced block indentation, this will be discussed further in the upcoming section 11.4.

"Basic operators" has been fulfilled due to the Spark language containing basic operators. This means that the operators which are defined under the requirement "Basic operators" in section 6.2.1 has been implemented in Spark.

The "Imperative programming" requirement is fulfilled due to the Spark language following the Imperative programming paradigm and executing code in a top-down manner.

Spark can write and handle if statements and while loops and therefore the requirements "If statements" and "While loops" are fulfilled.

The "Output" requirement is fulfilled because Spark can print to the terminal through the command print.

The "Variables (type restriction: strict)" is fulfilled due to the Spark language containing two variable types "text" and "number". This requirement also describes that when declaring a variable the user has to write the type followed by the variable name.

The "New line, statement ends" requirement describes that a statement should end when entering a new line, this has been implemented in Spark and therefore this requirement is fulfilled.

The "Symbol table/scope" requirement has been fulfilled since a symbol table has been implemented in the Spark compiler to handle the scope and related information of the variables declared.

Following the completed implementation of the "must have" requirements, the group began implementing the "should have" requirements as they were the next priority following the "must have" requirements.

The "should have" requirements which were fulfilled and implemented were: "Comments (marked with //)", "Error messages", "Concatenation", and "Visual output(graphics)". These are described in section 6.2.2 in the table 3.

The requirements which improve quality of life, such as "Comments (marked with //)" and "Concatenation " was given the highest priority of the "should have" requirements and were completed first, as they were relatively fast and simple to implement.

Then "Error messages" and the possibility to get a visual output through the requirement "Visual output(graphics)" were fulfilled. These would contribute to making the language more beginner-friendly and easier to learn.

When looking at the problem statement, which can be seen in section 4.4, one of the sub-questions the group wanted to focus on throughout the project was *"How can error messages be helpful and understandable for programming beginners?"*. Therefore the group focused on making meaningful error messages as a "should have" requirement, as

seen in table 3. The error messages give suggestions on how to rewrite the code for selected problems. A problem with the error messages can be if the identification of the intention is missed. Thereby, the error messages might not always give a suggestion that addresses the problem faced by the user. An example of this could be if the user tried to make the comment "number a is a number" and forgot to indicate it with the symbol "//", this is recognized as an error and will call the UnexpectedDeclarationException class, which can be seen in section 9.5 in Listing 22. The UnexpectedDeclarationException will return an error and in this case the error message "On line 1 you tried to "number a is a number". You should use a single "=" to assign the value of the variable". However, the user might not have tried to make a declaration but simply forgot the comment indicator "//", therefore this error message will not be helpful for the user. This will also be discussed in the further development section.

There was not enough time to implement the requirements from the could have part of the MoSCoW model in the time frame of the project. And as expected the group did not implement any of the "won't have" requirements.

11.2 Making syntax and keywords intuitive and memorable

As part of this project, the group wanted to focus on making the syntax and keywords easy, intuitive, and more memorable for beginners to learn and use. This was expressed through the sub-questions: *How can the programming languages syntax be intuitive?* and *How can the keywords be intuitive and memorable?*

This problem was confirmed by the data gathered from the participants of the questionnaire. In question 10, 24,5% of the participants from the questionnaire marked their biggest problem when starting to learn to code was "All the commands to remember". The group's proposed solution to solving this problem that beginners encounter was to make the keywords in Spark more intuitive and therefore easier to remember. However, it is debatable whether the keywords chosen for the language are easier to remember than keywords from other languages. For example, a keyword such as "!" is something one might have to learn in the beginning, but later on, it might be as easy to remember as the keyword "not" from Spark. But the fact that Spark is mostly based on "everyday words", can be a help for someone who has never coded before. In addition, as discussed previously in section 11.1, Spark uses the block indicators "then" and "end", or "do" and "end", which are also everyday words. Spark is also based on mathematics, an example of this is that regardless of whether the operation is an assignment or a comparison only one "=" symbol should be used. This reduces the amount of unique syntax rules the users have to remember which makes it easier to learn. But the fact that only a single "=" is used, does not make it clear whether it is an assignment or a comparison, which might be problematic for the understanding of the logic. However, since the group has not had enough time to conduct user tests, it is impossible to conclude whether these discussed problems might occur for the user and if they would be major or minor problems for the user to overcome. Based on the collected data from the questionnaire in section 4.1, the group assumes these problems might not be as major as they could seem.

11.3 Use of visual elements to enhance learning

As part of this project, the group wanted to focus on implementing visual elements and making them accessible for users to use through the Spark language. This was expressed through the sub-questions: *How can the programming language use visual elements and leverage graphics to enhance learning?*

The implementation of a visual aspect in Spark has resulted in the possibility to write the code and seeing a visual change based on their code. Based on the interview results seen in section 4.2.2, it turned out that visual aspects is really helpful for beginners in coding. When making geometric shapes in Spark, it has also been decided to use common words used in everyday life, like in other parts of the syntax. If the programmer is familiar with standard mathematical notations the syntax would most likely make sense. Overall, these decisions seem useful in relation to the problem statement. However, the way to declare a square may not be completely intuitive. When currently declaring a square in Spark the user has to write "square positionX positionY width height". This is justified in section 7.2.2 which shows the semantics of Spark. An example of this can be seen in Figure 44 line 45 in *Appendix E*. The declaration of a square would be more intuitive if it followed the same format as a declaration of a triangle which is: "triangle pointx1 pointy1 pointx2 pointy2 pointx3 pointy3". In other words it would probably be more intuitive to indicate all four points of the square when creating it, but it would be cumbersome if you want to change the position of the figure because you would then have to change all the corners instead of just one single point. It is possible to add colors to the shapes and the group decided that it should be done using everyday words, while also having the ability to use RGB values. This makes it easier for beginners, but it also limits the programmer in

which colors can be used. If there should be more options they would have to be integrated into the language.

The group made several tests, but due to the lack of time, it was decided not to do any user tests, although it would have contributed to a better insight into whether the language is really useful for beginners. Another thing that would have positively impacted the language could be if there was more time to make iterations of the language. The user tests and iterations would together add great value to the program. But to do proper user testing, the group would need to find some testers who have not tried to do programming before.

11.4 Further development

This section will discuss various features and other additions that would benefit and improve different aspects of the Spark programming language if it was developed further in the future. Some of the features and additions will be based on the MoSCoW requirement that was not completed or could be improved. Beyond these, there are some desirable additions that were not considered in the MoSCoW requirement as they were determined to be outside of the limitation set by the University, but these additions will also be highlighted in this section.

Many of the "should haves", seen in section 6.2.2, that were not implemented in Spark are not strictly necessary and can be worked around with existing features in Spark, but they make it easier for the programmer to code. To be more specific the features are "For loops", "Arrays", and "Switch case", by adding these, Spark could be written more efficiently by the user.

Adding advanced mathematical operators, would further contribute to the group's general goal for Spark to have a syntax that is intuitive for users with a background in mathematics on A level, as seen in the interview results in section 4.2.2. More advanced mathematics operators would have two main benefits for the Spark programming language. Firstly, it would allow users of Spark to apply more of the existing knowledge of mathematics in their projects. Secondly, by adding advanced mathematical operators Spark could better be used in general math education, or in cross-disciplinary projects.
The "Input" requirement, seen in section 6.2.2, in the "should haves" from the MoSCoW, would allow the users to interact with their project. The input system described in the "should haves" refers to a system where input comes from the terminal, which is a solution, but there are many other ways where the user could give input to the program such as reading the state of the keyboards, mouse, and other sensors on the computer of the user. Adding multiple systems for input would allow the user to make more complex projects such as games, simulations, and other systems.
Another of the "should have" requirements, seen in section 6.2.2, that the group did not manage to implement, was "Object-oriented programming", which appeared to be quite an important point from the interviews, seen in section 4.2, as most of the interviewees ended their courses with a focus on concepts from object-oriented programming. With the implementation of the most important concepts from object-oriented programming, Spark could be used for more of their courses. Furthermore, object-oriented programming is a common programming paradigm, and to be able to understand and use it is a valuable skill for any programmer.
The last "should have" that was not implemented in the Spark programming language is the "Declarations of functions", which is seen in section 6.2.2, which would allow the user to make their own functions. This addition would greatly improve the relevance of Spark as a learning tool as functions are a common feature in many programming languages and a cross-paradigm concept, which means it will be relevant to learn for most new programmers.

The group also made a category of "could have" requirements, however, these requirements had the lowest priority of all the requirements made. The "could have" requirements were: "End program", "Complex types", "Inheritance", "Interfaces", "Constructors", and "Abstract classes" these are each described in section 6.2.3. None of the "could have" requirements were implemented into the final product, primarily due to the lack of time in the project, and since they had the lowest priority of all the requirements the group focused on the "must have" and "should have" requirements first. However, if there were to be made further development of this project it would be ideal to implement the rest of the "should have" requirements missing and all the "could have" requirements.

As most of the interviewees said, forced block indentation would be a great feature to have for beginners, as seen in the results of the interviews in section 4.2.2, as this makes the code more readable. The main benefit of this change is it makes it easier for new programmers to read and understand code examples, follow step-by-step guides, and use other documentation. Block indentation would also help teachers and instructors, read and get a better overall understanding of the student's code, which would help them give a better explanation or solution. Lastly,

the increase of readability from the block indentation would make collaboration between users easier as the time used for comprehending changes and code from other users would be decreased for all members of a group.

One possible visual feature that could be added to the language is a general shape solution, where it is possible to decide the shape for yourself and are not limited to triangles, rectangles, and circles. It would also be great to have the opportunity to change the window size which right now is fixed.

Outside of additional features to the Spark programming language, it would be important to develop more comprehensive documentation specifically designed for new programmers. This type of documentation would serve as a beneficial addition, enabling individuals to learn Spark without the need for a teacher or instructor. By providing a clear and accessible resource, new programmers would be empowered to explore Spark independently and engage in problem-solving within the language. Comprehensive documentation would offer detailed explanations of programming concepts, practical examples showcasing how commands can be used, and step-by-step guides for making simple programs, ensuring that learners can understand and utilize Spark's functionalities effectively. With such documentation in place, new programmers would have the opportunity to develop into Spark's intricacies, experiment with different approaches, and ultimately harness the full potential of the language for their specific needs. Furthermore, parts of the documentation could be added to the IntelliSense [62] of the user's IDE (Integrated development environment) to give useful suggestions, explanations, and links to further reading in the documentation, throughout the user's coding and interaction with Spark. In addition to using links to the documentation in IntelliSense, links could also be used in the error messages to link the user to a page with a more detailed description of the problem, a thorough description of possible solutions, and additional relevant information. These additions to IntelliSense and the error messages would enhance the troubleshooting experience and encourages new programmers to actively explore the documentation for a deeper understanding of Spark.

12 Conclusion

This section concludes whether the final product proposes a solution to the problems stated in the problem statement. The group chose to undertake a topic focusing on beginners learning to code. Through research was the target group delimited to "teenagers and young adult programming beginners" where the main focus and problem statement are:

How can a programming language be designed and implemented to address the most common barriers for teenagers and young adult programming beginners?

This is followed by some additional sub-question that can be seen in section 4.4.

The final product consists of the language Spark, which is designed as a beginner language with few keywords and easily understood error messages, as discussed in section 11. With the creation of the language Spark, a working compiler was similarly created that translates Spark into Java.

User testing has not yet been conducted, therefore it would prove difficult to provide an accurate evaluation of how effectively the developed product solves the problem statement. Although it can be pointed out that the product should still prove to be a viable solution nonetheless since it was built whilst keeping the problem statement, questionnaire, and interview data in mind. As well as resolving all of the sub-questions in the problem statement to a certain degree.

References

[1] J. E. Holgaard, T. Ryberg, N. Stegeager, D. Stentoft, and A. O. Thomassen, *Problembaseret læring og projektarbejde ved de videregående uddannelser*. 2020, vol. 2. ISBN: 9788759333969.

[2] Kanbanize, *What Is a Kanban Board and How to Use It? Basics Explained.* Mar. 2023. [Online]. Available: https://kanbanize.com/kanban-resources/getting-started/what-is-kanban-board (visited on 03/23/2023).

[3] Trello, *Manage Your Team's Projects From Anywhere — Trello*, Mar. 2023. [Online]. Available: https://trello.com/ (visited on 03/23/2023).

[4] VMware, *What is Configuration Management?* en-US. [Online]. Available: https://www.vmware.com/topics/glossary/content/configuration-management.html (visited on 05/18/2023).

[5] Github, *GitHub: Let's build from here*, en, May 2023. [Online]. Available: https://github.com/ (visited on 05/18/2023).

[6] JetBrains, *IntelliJ IDEA – the Leading Java and Kotlin IDE*, en, May 2023. [Online]. Available: https://www.jetbrains.com/idea/ (visited on 05/18/2023).

[7] A. Bhat, *Surveys: What They Are, Characteristics & Examples*, en-US, Oct. 2018. [Online]. Available: https://www.questionpro.com/blog/surveys/ (visited on 05/24/2023).

[8] *Likert scale — social science — Britannica*, en, Apr. 2023. [Online]. Available: https://www.britannica.com/topic/Likert-Scale (visited on 05/24/2023).

[9] mBlock, *mBlock*. [Online]. Available: https://mblock.makeblock.com/en-us/ (visited on 05/10/2023).

[10] Microsoft, *Visual Studio Code - Open Source ("Code - OSS")*, original-date: 2015-09-03T20:23:38Z, Mar. 2023. [Online]. Available: https://github.com/microsoft/vscode (visited on 03/08/2023).

[11] Microsoft, *Microsoft acquires GitHub*, en-US, Jun. 2018. [Online]. Available: https://news.microsoft.com/announcement/microsoft-acquires-github/ (visited on 03/13/2023).

[12] Openai, *Product*, en-US, Mar. 2023. [Online]. Available: https://openai.com/product (visited on 03/13/2023).

[13] A. Froehlich, *What is the Lisp (List Processing) Programming Language? – A Definition from TechTarget.com*, en. [Online]. Available: https://www.techtarget.com/whatis/definition/LISP-list-processing (visited on 05/24/2023).

[14] J. Moreno-León and G. Robles, *Automatic detection of bad programming habits in scratch: A preliminary study*, English, 2015. [Online]. Available: https://www.researchgate.net/publication/283041226_Automatic_detection_of_bad_programming_habits_in_scratch_A_preliminary_study.

[15] *Scratch - About*, da. [Online]. Available: https://scratch.mit.edu/ (visited on 02/27/2023).

[16] *Platform tutorial on Scratch*. [Online]. Available: https://scratch.mit.edu/projects/238763/ (visited on 03/15/2023).

[17] *History and License*. [Online]. Available: https://docs.python.org/3/license.html (visited on 03/15/2023).

[18] *Introduction to Python*, en-US. [Online]. Available: https://www.w3schools.com/python/python_intro.asp (visited on 03/15/2023).

[19] *Python Variables*, en-US. [Online]. Available: https://www.w3schools.com/python/python_variables.asp (visited on 03/15/2023).

[20] *General Python FAQ*. [Online]. Available: https://docs.python.org/3/faq/general.html (visited on 03/15/2023).

[21] Pontifical Catholic University of Rio de Janeiro, *Lua: About*, Oct. 2022. [Online]. Available: https://www.lua.org/about.html (visited on 02/23/2023).

[22] Roblox Corporation, *Introduction to Scripting*, en, Feb. 2023. [Online]. Available: https://create.roblox.com/docs (visited on 02/23/2023).

[23] Armak, *Comprehensive Beginner's Guide for WoW Addon Coding in Lua*, en, Nov. 2022. [Online]. Available: https://www.wowhead.com/guide/comprehensive-beginners-guide-for-wow-addon-coding-in-lua-5338 (visited on 02/23/2023).

[24] Nodecraft, *gLua 101 - An Introduction to Garry's Mod Lua coding - Nodecraft*, en, Feb. 2019. [Online]. Available: https://nodecraft.com/support/games/gmod/glua-101-an-introduction-to-garrys-mod-coding (visited on 02/23/2023).

[25] Wube Software, *Runtime Docs — Factorio*, Feb. 2023. [Online]. Available: https://lua-api.factorio.com/latest/index.html (visited on 02/23/2023).

[26] Pontifical Catholic University of Rio de Janeiro, *Lua 5.1 Reference Manual*, Aug. 2019. [Online]. Available: https://www.lua.org/manual/5.1/manual.html (visited on 02/27/2023).

[27] w3schools, *Python Keywords*, en-US, Feb. 2023. [Online]. Available: https://www.w3schools.com/python/python_ref_keywords.asp (visited on 02/27/2023).

[28] corob-msft, *C Keywords*, en-us, Sep. 2021. [Online]. Available: https://learn.microsoft.com/en-us/cpp/c-language/c-keywords (visited on 02/27/2023).

[29] T. Taylor, *Static vs. dynamic typing: The details and differences — TechTarget*, en, Jun. 2021. [Online]. Available: https://www.techtarget.com/searchapparchitecture/tip/Static-vs-dynamic-typing-The-details-and-differences (visited on 02/27/2023).

[30] R. Ierusalimschy, *Programming in Lua*, eng. Rio de Janeiro: Lua.org, 2003, ISBN: 9788590379812.

[31] *What is a Programming Library? A Beginner's Guide (2023)*, en-US, Running Time: 933 Section: Web Development, Jan. 2023. [Online]. Available: https://careerfoundry.com/en/blog/web-development/programming-library-guide/ (visited on 02/27/2023).

[32] *People*, en-US. [Online]. Available: https://processing.org//people (visited on 02/23/2023).

[33] *Welcome to Processing!* en-US. [Online]. Available: https://processing.org// (visited on 02/23/2023).

[34] I. Greenberg, *Processing: creative coding and computational art*, eng. Berkeley, CA New York: Friends of Ed Distributed to the Book trade worldwide by Springer-Verlag, 2007, ISBN: 978-1-4302-0310-0.

[35] *Processing Foundation*, en. [Online]. Available: https://processingfoundation.org/ (visited on 02/23/2023).

[36] *Home — p5.js*. [Online]. Available: https://p5js.org/ (visited on 02/23/2023).

[37] *Processing for Android*. [Online]. Available: https://android.processing.org/ (visited on 02/23/2023).

[38] *Overview*, en-US. [Online]. Available: https://processing.org//overview (visited on 02/23/2023).

[39] M. A. Hoy, *From point to pixel: a genealogy of digital aesthetics* (Interfaces. Studies in Visual Culture), eng. Hanover, New Hampshire: Dartmouth College Press, 2017, ISBN: 978-1-5126-0023-0 978-1-5126-0022-3 978-1-5126-0021-6.

[40] L. Olivieri and J. Northrop, *Cooper-Hewitt, National Design Museum Announces Winners and Finalists of the 12th Annual National Design Awards*, eng, May 2011. [Online]. Available: http://cdn.cooperhewitt.org/2011/05/26/Final%20-%20CHNDM%20NDA%202011%20Winner%20Release%205-26-11.pdf.

[41] ProductPlan, *What is MoSCoW prioritization? Overview of the MoSCoW Method*, Mar. 2023. [Online]. Available: https://www.productplan.com/glossary/moscow-prioritization/.

[42] *C Structures (structs)*, en-US. [Online]. Available: https://www.w3schools.com/c/c_structs.php (visited on 04/11/2023).

[43] BillWagner, *Structure types - C# reference*, en-us, Apr. 2023. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/struct (visited on 04/11/2023).

[44] R. W. Sebesta, *Concepts of Programming Languages*, Eleventh edition. Pearson.

[45] C. N. Fischer, R. K. Cytron, and R. J. LeBlanc, *Crafting a compiler* (Crafting a compiler with C). Boston: Addison-Wesley, 2010, OCLC: ocn268788318, ISBN: 978-0-13-606705-4.

[46] *Errors in Compiler Design - Coding Ninjas*. [Online]. Available: https://www.codingninjas.com/codestudio/library/errors-in-compiler-design (visited on 05/12/2023).

[47] *Introduction to Syntax Analysis in Compiler Design*, en-us, Section: Compiler Design, Sep. 2015. [Online]. Available: https://www.geeksforgeeks.org/introduction-to-syntax-analysis-in-compiler-design/ (visited on 05/02/2023).

[48] *LL(1) Grammars*. [Online]. Available: https://www.csd.uwo.ca/~mmorenom/CS447/Lectures/Syntax.html/node14.html (visited on 05/01/2023).

[49] *LR Parsers*. [Online]. Available: https://www.csd.uwo.ca/~mmorenom/CS447/Lectures/Syntax.html/node29.html (visited on 05/01/2023).

[50] R. W. Sebesta, *Concepts of programming languages* (Always learning), eng, Eleventh edition, global edition, S. Mukherjee and A. K. Bhattacharjee, Eds. Boston Munich: Pearson, 2016, ISBN: 9781292100555.

[51] ANTLR, *About The ANTLR Parser Generator*. [Online]. Available: https://www.antlr.org/about.html (visited on 04/20/2023).

[52] T. Parr, *ANTLR v4 with Terence Parr*, da-DK, Feb. 2013. [Online]. Available: https://www.youtube.com/watch?v=q8p1voEiu8Q (visited on 05/16/2023).

[53] *Semantic Analysis in Compiler Design*, en-us, Section: Compiler Design, Oct. 2019. [Online]. Available: https://www.geeksforgeeks.org/semantic-analysis-in-compiler-design/ (visited on 05/02/2023).

[54] GeeksForGeeks, *Symbol Table in Compiler*, en-us, Apr. 2018. [Online]. Available: https://www.geeksforgeeks.org/symbol-table-compiler/ (visited on 05/04/2023).

[55] *Phases of a Compiler*, en-us, Section: Compiler Design, Nov. 2017. [Online]. Available: https://www.geeksforgeeks.org/phases-of-a-compiler/ (visited on 05/02/2023).

[56] subhajitghosh1997, *What is Tombstone Diagram?* Dec. 2022. [Online]. Available: https://www.geeksforgeeks.org/what-is-tombstone-diagram/ (visited on 05/22/2023).

[57] B. Thomsen, *Course: Languages and Compilers (DAT4, AAL-SW4, KBH-SW4, CS-IT8)*. [Online]. Available: https://www.moodle.aau.dk/course/view.php?id=46654 (visited on 05/12/2023).

[58] B. Thomsen, *Lecture 14-2 - Programming Languages and Compilers*, May 2023.

[59] S. Jacob, *How to test a compiler*. [Online]. Available: https://circleci.com/blog/unit-testing-vs-integration-testing/ (visited on 05/11/2023).

[60] STF, *Integration Testing*, en-US, Mar. 2011. [Online]. Available: https://softwaretestingfundamentals.com/integration-testing/ (visited on 05/16/2023).

[61] UTOR, *What is Acceptance Testing? The Ins & Outs*, en-US, Aug. 2020. [Online]. Available: https://u-tor.com/topic/about-acceptance-testing (visited on 05/17/2023).

[62] microsoft, *IntelliSense in Visual Studio Code*, en, Mar. 2023. [Online]. Available: https://code.visualstudio.com/docs/editor/intellisense (visited on 05/21/2023).

Appendix A - Declaration of Consents

Aalborg Universitet CPH ***Marts-2023***

Samtykkeerklæring

Dette er en anmodning om dit samtykke til at behandle dine personoplysninger. Formålet med interviewet er at samle viden om hvordan undervisning og læring af programmering foregår.

Du giver samtykke til at behandle følgende oplysninger om dig: Navn, alder, køn, arbejdsposition, arbejdsform, evt. private oplysninger angåendes din stilling, undervisning metode, og jeres pensum. Samt at optage dine svar i interviewet.

Vi, Gruppe 6: Goran Kirovski, Karina Botes, Magne Frank Dinesen, Mikkel Skovlund, Pernille Knudsen, & Tobias Friese, er personligt dataansvarlig for dine oplysninger.

Dine oplysninger bliver opbevaret sikkert, og vi benytter dem udelukkende til ovenstående formål.

Du har altid ret til at trække dit samtykke tilbage. Ønsker du senere at trække dit samtykke tilbage, kan du kontakte gruppens mail på følgende mailadresser: Tubbe.Friese@gmail.com & Magne@skoven7.dk

Databeskyttelsesforordningen giver dig ret til at få en række oplysninger, som du finder i den e-mail du modtager fra os senere.

1 / 4

Figure 34: DOS - Page 1

☐ Jeg giver hermed samtykke til, at Gruppen må behandle mine oplysninger i henhold til ovenstående formål og oplysninger.

☐ Jeg giver hermed samtykke til, at Gruppen bruge lydoptagelse og/eller videooptagelse under interviewet.

Dato:

Navn:

Underskrift

Sådan behandler vi dine data

Dataansvarlige

Gruppe 4: Goran Kirovski, Karina Botes, Magne Frank Dinesen, Mikkel Skovlund, Pernille Knudsen, & Tobias Friese

Figure 35: DOS - Page 2

Formålet med at behandle dine oplysninger

Formålet med at indsamle dine oplysninger er at samle kvalitativ data til brug i vores rapportskrivning og projektudvikling her på Aalborg Universitet I København. Oplysningerne og informationerne fra interview o.l. vil blive analyseret på diverse måder og danne baggrund for vores projektskrivning.

Vi behandler disse personoplysninger

☑ Almindelige personoplysninger (jf. art. 6, stk. 1, litra a)

(Fx navn, adresse, e-mail, alder, selvoffentliggjorte data mv.)

☐ Følsomme personoplysninger (jf. art. 9, stk. 2, litra a)

(Fx helbredsoplysninger, race, politisk overbevisning mv.)

Sådan opbevarer vi dine oplysninger

Vi opbevarer dine personoplysninger, så længe det er nødvendigt i forhold til formålet med at indhente dit samtykke og i henhold til gældende lovgivning. Herefter sletter vi dine personoplysninger.

Dine rettigheder

Når vi behandler dine personoplysninger, har du ifølge databeskyttelsesforordningen flere rettigheder. Det betyder bl.a., at du har ret til sletning og dataportabilitet.

Figure 36: DOS - Page 3

I visse tilfælde har du ret til indsigt, berigtigelse, begrænsning og til at gøre indsigelse mod vores behandling af de omfattede personoplysninger.

Vær opmærksom på, at du ikke kan trække dit samtykke tilbage med tilbagevirkende kraft.

Vil du klage?

Mener du ikke, at vi lever op til mit ansvar, eller vi ikke behandler dine oplysninger efter reglerne, kan du klage til Datatilsynet på dt@datatilsynet.dk.

Vi opfordrer dig dog til også at kontakte os først, da vi vil gøre, hvad vi kan, for at imødekomme din klage.

Overdragelse til og fra tredjepart

Din data (eller dele af din data) kan blive overdraget til: Aalborg Universitet København.

Figure 37: DOS - Page 4

Appendix B - Interview Exercise Template

- **Statement ends**
  - ;
  - Ny linje
  - ,
- **Comments**
  - //
  - /**/
  - %
  - <!-- -->
  - !
  - --
  - *
- **Tvunget Indentation**
- **Blocks**
  - {}
  - Then og end
  - :- .
- **No Type restriction (fx let)**
- **Pointers**
- **Paradigmer**
  - Logisk programmering
  - Objekt orienteret programmering
  - Imperativ programmering
  - Funktionel programmering
  - Block programming (e.g Scratch)

Figure 38: Interview Exercise Template

Appendix C - Interview Exercise Answers

- **Statement ends**
  - ~~;~~
  - Ny linje
  - ,
- **Comments**
  - //
  - /**/
  - %
  - <!-- -->
  - !
  - --
  - *
  - #
- **Tvunget Indentation**
- **Blocks**
  - {}
  - Then og end
  - :- .
- **No Type restriction (fx let)**
- ~~**Pointers**~~
- **Paradigmer**
  - Logisk programmering
  - Objekt orienteret programmering
  - Imperativ programmering
  - Funktionel programmering
  - ~~Block programming (e.g Scratch)~~

HiP som Hap

Figure 39: Christian (Interviewee 1) - Interview Exercise

Louise

- **Statement ends**
  - ;
  - Ny linje
  - ,
- **Comments**
  - //
  - /**/
  - %
  - <!-- -->
  - !
  - --
  - *
- **Tvunget Indentation**

Ja!

- **Blocks**
  - {}
  - Then og end
  - :- .
- **No Type restriction (fx let)**
- **Pointers**
- **Paradigmer**
  - Logisk programmering
  - Objekt orienteret programmering
  - Imperativ programmering
  - Funktionel programmering
  - Block programming (e.g Scratch)

Figure 40: Louise (Interviewee 2) - Interview Exercise

Daniel

- **Statement ends**
  - ;
  - Ny linje
  - ,
- **Comments**
  - //
  - /**/
  - %
  - <!-- -->
  - !
  - --
  - *
- **Tvunget Indentation**
- **Blocks**
  - {}
  - Then og end
  - :- .
- **No Type restriction (fx let)** strict
- **Pointers** %
- **Paradigmer**
  - Logisk programmering
  - Objekt orienteret programmering
  - Imperativ programmering
  - Funktionel programmering
  - Block programming (e.g Scratch)

Figure 41: Daniel (Interviewee 3) - Interview Exercise

kristoffer

- **Statement ends**
  - ;
  - Ny linje
  - ,
- **Comments**
  - //
  - /**/
  - %
  - <!-- -->
  - !
  - --
  - *
- **Tvunget Indentation**
- **Blocks**
  - {}
  - Then og end
  - :-
- **No Type restriction (fx let)**
- ~~**Pointers**~~
- **Paradigmer**
  - Logisk programmering
  - Objekt orienteret programmering
  - Imperativ programmering
  - Funktionel programmering
  - Block programming (e.g Scratch)

Ivan Bosse

Figure 42: Kristoffer (Interviewee 4) - Interview Exercise

Appendix D - Spark's BNF

```
grammar bnf;

s: extraLines line | extraLines globalStatement startFunction (drawFunction | empty);

startFunction: 'on' 'start' block forcedLineChange globalStatement;

drawFunction: 'on' 'eachFrame' block forcedLineChange globalStatement;

line: statement forcedLineChange line | variableStatement forcedLineChange line | visualStatement forcedLineChange line | empty;

statement: 'if' condition block extraLines elseToken | 'while' condition loopBlock | 'print' stringValue | 'printLine' stringValue | 'break' | 'continue';

globalStatement: declaration forcedLineChange globalStatement | empty;

visualStatement: figure | colorPick;

colorPick: 'color' (colorText | parameter parameter parameter) | 'background' (colorText | parameter parameter parameter);

colorText: 'darkRed' | 'red' | 'lightRed' | 'darkGreen' | 'green' | 'lightGreen' | 'darkBlue' | 'blue' | 'lightBlue' | 'black' | 'white' | 'darkGrey' | 'grey' | 'lightGrey';

figure: 'circle' parameter parameter parameter parameter | 'square' parameter parameter parameter parameter | 'triangle' parameter parameter parameter parameter parameter parameter;

parameter: expression | ID;

elseToken: 'else if' condition block extraLines elseToken | 'else' block | empty;

condition: '(' condition ')' extraCondition | 'not' notCondition extraCondition | singleCondition extraCondition;

extraCondition: 'and' condition | 'or' condition | empty;

notCondition: '(' condition ')' | singleCondition;

singleCondition: equation comparator equation | String '=' String;

block: 'then' forcedLineChange line 'end';

loopBlock: 'do' forcedLineChange line 'end';

variableStatement: declaration | assignment;

declaration: 'number' ID '=' equation | 'text' ID '=' stringValue;

assignment: ID '=' value;

value: ID extraValue | expression extraValue | stringCheckRule extraValue | '(' value ')' extraValue;

stringCheckRule: String;

extraValue: '+' value | operator value | empty;

stringValue: ID extraStringValue | String extraStringValue | num extraStringValue;

extraStringValue: '+' stringValue | empty;

equation: '(' equation ')' extraEquation | ID extraEquation | expression extraEquation;

extraEquation: operator equation | empty;

expression: '-' num | num;

num: Digits decimal;

decimal: '.' Digits | empty;
```

```
64
65 forcedLineChange: ('\n')+;
66
67 extraLines: ('\n')*;
68
69 operator: '+' | '-' | '%' | '/' | '*';
70
71 comparator: '<' | '>' | '=' | '<=' | '>=';
72
73 empty: ;
74
75 String: '"'  (~'"')*  '"';
76
77 ID: ([a-z]|[A-Z]) ([a-z]+|[A-Z]+|[0-9]+)*;
78
79 Digits: [0-9]+;
80
81 WS: (' '|'\t' | '\r')+ -> skip;
82
83 COMMENT: '/*' (~'*')* '*/' -> skip;
84
85 LINE_COMMENT:   '//' (~[\r\n])* -> skip;
86
87 ErrorSymbol: (~('"' | [a-z]| [A-Z] | [0-9]|' '|'\n'|'\r'))*;
```

Listing 36: The full BNF of the parser

Appendix E - Acceptance tests input

```
1
2  on start then
3
4  end
5
6  number top = 0
7  number left = 0
8  number d = 0
9  number e = 0
10 number size = 716
11 number speed = 3
12
13 on eachFrame then
14
15     if (e <= size and left = 0 and top = 0) or e <= 0 then
16         e = e + speed
17         left = 0
18         if e >= size then
19             left = 1
20         end
21         printLine "right"
22     end
23     else if (d <= size and top = 0 and left = 1) or (d <= 0) then
24         d = d + speed
25         top = 0
26         if d >= size then
27             top = 1
28         end
29         printLine "down"
30     end
31     else if (e > 0 and left = 1 and top = 1) then
32         e = e - speed
33         left = 1
34         printLine "left"
35     end
36     else if (d > 0 and left = 0 and top = 1) then
37         d = d - speed
38         top = 1
39         printLine "up"
40     end
41
42     background 0 150 136
43
44     color 96 125 139
45     square e d 80 80
46
47     color 76 175 80
48     triangle 400 100 100 700 700 700
49
50     color 139 195 74
51     circle 400 515 370 370
52 end
```

Figure 43: File example1.txt, input for the first acceptance test

```
1
2  on start then
3
4  end
5
6  number top = 0
7  number left = 0
8  number d = 0
9  number e = 0
10 number size = 716
11 number speed = 3
12
13 on eachFrame then
14
15     if (e <= size and left = 0 and top = 0) or e <= 0 then
16         e = e + speed
17         left = 0
18         if e >= size then
19             left = 1
20         end
21         printLine "right"
22     end
23     else if (d <= size and top = 0 and left = 1) or (d <= 0) then
24         d = d + speed
25         top = 0
26         if d >= size then
27             top = 1
28         end
29         printLine "down"
30     end
31     else if (e > 0 and left = 1 and top = 1) then
32         e = e - speed
33         left = 1
34         printLine "left"
35     end
36     else if (d > 0 and left = 0 and top = 1) then
37         d = d - speed
38         top = 1
39         printLine "up"
40     end
41
42     background 0 150 136
43
44     color 96 125 139
45     square e d 80 80
46
47     color 76 175 80
48     triangle 400 100 100 700 700 700
49
50     color 139 195 74
51     circle 400 515 370 370
52 end
```

Figure 44: File example2.txt, input for the second acceptance test