Make Me A Gear

- A High-Level Programming Language for 3D-printing of Gears -

Project Report

Software 4-03

Aalborg University
Dept. of Computer Science

Dept. of Computer Science
Aalborg University
http://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Make Me A Gear

Theme:
Design, Definition, and Implementation of Programming Languages

Project Period:
Spring Semester 2022

Project Group:
Software 4-03

Participant(s):
Andreas H. Paludan
Emilie S. Steinmann
Ibrahim A. Abu Rached
Julie H. Bengtsson
Patrick Laursen
William Brinck

Supervisor(s):
Christian Schilling

Page Numbers: 106

Date of Completion: February 19, 2026

Abstract:

With the increased accessibility of 3D printers, more people can print spare part gears for toys and home appliances. However, most computer-aided design programs are difficult to use and expensive for private users. The purpose of this project is to bridge this gap by designing an open source, easy-to-use programming language for 3D printing of gears, taking into consideration the many gear parameters that must be calculated correct.
The proposed solution is a language called MMAG in which a user can design a gear model. The associated compiler is implemented in C# with use of the ANTLR4 parser generator. It translates the MMAG program into a set of commands that can be executed on a 3D printer. In the language, flexibility is traded for simplicity to ensure that most users can use the language. The user only has to take a few measurements and assign them to the corresponding attributes in a MMAG program, based on which the printer instructions are generated.
This report documents the development of the MMAG language and its compiler that with a few lines of simple code can generate a full set of printer commands for printing a gear.

Contents

Chapter 1

Motivation

3D printers are becoming more available to the public, and private usage has increased over the last couple of years. In 2018 around 10% of the leading uses of 3D printing was in the field of personal interest, and in 2020 this had increased to around 32% [5]. Entry level printers can be bought for around $200 [7], making it accessible to almost everybody. A printer at this price point is fine for printing smaller objects, but it lacks the quality and detail that more expensive printers can produce. However, if the model requires more detail and better materials it is possible to buy printing-time at libraries or makerspaces [54, 37] at affordable prices. The objects to be printed can be created using Computer Aided Design (CAD) software such as Blender [16], and Ultimaker Cura [59] can be used to translate the 3D model to so-called G-code which is a list of instructions that tells the 3D printer how to build the 3D model. Both programs are really high quality despite being free to use, thus making the process of 3D printing accessible for private consumers.

A result of the increased accessibility of 3D printers is that private consumers for instance can replace broken parts in toys and household appliances with 3D printed plastic components that they design themselves. These parts could for example be custom gears for remote-controlled cars or for garage door openers [28, 55]. In general, gears are a popular item to print as seen on the online 3D print library Thingiverse [56]. In the childhood of the authors of this report, one would either end up discarding the product or ordering a spare part, potentially with high delivery times. With the option of 3D printing parts locally, it is assumed that the frustration with broken mechanical plastic parts can be diminished and the likelihood of people fixing the product instead of discarding it is increased. This also supports the 12th UN Sustainable Development Goal about sustainable consumption and production, where consumers are encouraged to create less waste, e.g., by fixing broken items [31, 61].

To print the desired parts, a set of commands for the 3D printer must be generated. One way to do this is by writing G-code directly, which is however a very complicated process with a big boiler

plate overhead. It also requires profound understanding of the G-code commands, which is tiresome to achieve as the commands are named with a letter and a number, which are difficult to remember. Furthermore G-code is not specifically made for 3D printing, but it is a language used to program Computer Numerical Control machines, commonly known as CNC-machines, in general. This means that G-code has a lot of functionalities, since it needs to satisfy all the different movements the different CNC-machines can perform. Identifying the relevant commands for 3D printing is then another barrier for the consumer [49].

To avoid the process of having to write G-code commands for printing a 3D model, CAD software can be used to design the models. CAD software makes it possible to digitally create and visualize 3D models. There are many different CAD programs available varying in price and serving different purposes. The more professional CAD software like AutoCAD can be quite expensive [34, 13]. For private use, different free CAD programs are available, one of them being Blender [16]. To run Blender on a computer, it is recommended that the computer among others has a 2560×1440 display, 32 GB RAM, and a dedicated graphics card with 8 GB RAM [15]. 90% of the people in Denmark (2018) has a laptop [20], so it is assumed that they will be designing their models on such a computer. Laptops for standard use, like streaming and internet surfing, are most likely to have 4-8 GB RAM, a 1600x900 or 1920x1080 display, and an integrated graphics card [43], thus the hardware recommendations exclude many private customers from using the software. Another disadvantage to the software is that it requires extensive knowledge on how to use the software, which can be time-consuming to achieve [12].

As mentioned in the beginning of this chapter, custom gears are a practical item to be able to print at home. However, the existing free CAD software appears to have limited capabilities when it comes to quickly designing gears of different dimensions and models for inexperienced users. One of the free CAD software that can be used to show this is Blender. Blender does indeed have the capabilities to create gears through one of its context menus, however the user first has to go into the preferences of Blender and enable the add-on, before the gear mesh option will appear [21]. Users who are not familiar with CAD software, might not know where to find these settings. Furthermore, the gear tool does not provide a complete, ready-to-print gear, meaning that a deeper, general understanding of CAD software is also required [21]. Since gears are complex, it would require to get all technicalities of the gear correct with a reasonable precision for it to work properly, making it difficult to create gears without a tool for this specific purpose [35].

In conclusion, it is time-consuming to become familiarized with both CAD software and fundamental G-code programming. To counter this, different solutions have been proposed. One solution to modify 3D models is to use the Mazatrol programming interface, which is however only available on Mazak CNC-machines and better suited for modifying models, rather than creating them from scratch [39, 4]. Another proposed solution is the Mecode library for the Python programming language [50]. Mecode provides understandable names for common G-code instructions as well as methods to generate simple structures like rectangles. However, it does not have the option of easily

making specific shapes like gears, why it can be concluded that there are no practical solutions available for designing gears easily when trying to avoid the use of CAD software, as well as avoiding having to write G-code manually. Therefore, this project aims to develop an easy way for consumers to create the necessary 3D printer instruction set for printing custom-made gears.

Chapter 2

Problem Analysis

Before a specific problem regarding easy design of 3D printed gears can be formulated, a deeper analysis of the general problem field must be made. Therefore, this chapter examines and describes relevant areas related to 3D printing, gears, and programming languages, in order to shed light on the central issue described in chapter 1.

2.1 3D Printers

3D printers are used for 3D printing, which is a type of additive manufacturing. 3D printing refers to the process of creating three dimensional objects from a digital 3D model [1, 62]. There are different types of 3D printer technologies, where the type of printing material differs among the technologies, making them suitable for different purposes. The most popular technology is called fused deposition modeling (FDM) or fused filament fabrication (FFF) [6], which will be the focus in this report. The technology can be referred to as both FDM and FFF, however this report will refer to it as FDM. The FDM technology uses plastic filament from a spool to create the 3D print. The filament will be added to an extrusion nozzle, which will then be heated for the filament to melt. The melted filament will be extruded layer for layer, while the nozzle moves around the printer, in order to create the 3D print [1].

3D prints are made by first creating a digital 3D model using, e.g., a CAD program, as mentioned in chapter 1. Typically, the 3D design will then be saved as an .STL file or an .OBJ file. The printer will not be able to receive instructions from files in those formats, so before being able to print it, the model must be converted into G-code using a slicing program. Slicing refers to the process of splitting up the 3D model into different layers, that the printer will then be able to print out. This is described in further detail in section 2.6. As a result of using a slicing program, the model is

converted to G-code, which can then be sent to the printer and printed out [1, 3].

Even though there are a lot of benefits to 3D printers, they still have some general limitations. One limitation is that it is not necessarily possible to print out everything, as the size of the print bed, on which the object is printed, together with the height of the printer, decides the maximum dimensions of the 3D objects that can be printed, and if bigger objects are wanted, these need to be printed as separate parts. Another limitation is that a 3D printer only supports one or maybe a couple of different materials for printing, e.g., FDM 3D printers are limited to certain types of plastic, which also implies some limitations in regards to the strength of the printed models [23].

To properly take these limitations into account when generating the G-code instruction set, it is important to have a good understanding of the function of the object that is being printed, as well as the stress, it will be put under. To achieve this, the following section will investigate the construction of and requirements for gears.

2.2 Gears

As mentioned in chapter 1, gears are a practical item to be able to 3D print at home. While they might be easy to prototype, optimal gears heavily depend on some key parameters being correct [17]. Understanding how gears are constructed and how the key parameters are calculated is essential when creating the required set of printer instructions to make functioning gears. The set of relevant key parameters, as well as the formulas for calculating their values, depend heavily on the specific gear type, why a brief introduction is first made to the most common gear types. Hereafter, some terminology is introduced, and relevant key parameters are identified, as these will play a central role in the development of a language for gear design.

Gear Types

When speaking of gears, it is important to mention that there exist a variety of gear types that are used throughout all kinds of mechanical devices. Different types of gears are used for different purposes. The most common gear type is the spur gear depicted in Figure 2.1a. Spur gears are recognized by the straight design of the teeth, and they are commonly used in gear reductions. This is the type of gear you will find mostly in plastic toys etc. [44].

Helical gears, as seen in Figure 2.1b, consist of angled teeth instead of straight teeth. This makes the contact between the gear teeth more gradual, and outputs a smoother connection. Due to the gradual engagement, helical gears also make less noise than a spur gear. This type of gear is very commonly

(a) Spur gear [44].

(b) Helical gear [44].

Figure 2.1: Diagrams of spur and helical gears.

found in cars, specifically in the car transmission [44].

In Figure 2.2, diagrams of bevel, worm, and rack and pinion gears are provided. The bevel gear in Figure 2.2a is a gear type that is used to change the direction of the power. It is commonly mounted on shafts that are 90 degrees apart, which will then result in a direction change. The teeth on a bevel gear can be straight like on spur gears, but they can also be spiral or hypoid. Bevel gears with hypoid teeth are commonly used in car differentials, due to how it engages the axes in different planes [44]. Hypoid gears will not be explained in further details, as their relevance to the project is low and their complexity is rather high.

(a) Bevel gear [44].

(b) Worm gear [44].

(c) Rack and pinion gear [44].

Figure 2.2: Diagrams of bevel, worm, and rack and pinion gears.

Worm gears, as shown in Figure 2.2b, are mainly used when a large gear reduction is required. A common worm gear will have a reduction of 20:1 or alike, with the possibility of reductions up to 300:1 or greater. A special feature of the worm gear, is how the worm gear can turn a spur gear without the spur gear being able to turn the worm gear. The worm gear can be found in the Torsen differential used in some high-performance cars [44].

The last gear type in Figure 2.2 is the rack and pinion gear, seen in Figure 2.2c. Rack and pinion gears are used to convert rotation into linear motion. A good example of the use of a rack and pinion gear is for the steering wheel in a car. Its rotation is converted into linear movement that adjusts the

angle of the wheels compared to the center line of the vehicle [44].

While these are only some of the gear types that exist, not all of these gears will be relevant in this project. This is mainly due to the focus being on private users wanting to replace, e.g., plastic gears in toys, where the spur gear is the most common of the different gears mentioned. Therefore, the focus for this project will primarily be on spur gears, and secondarily on bevel gears which are also used, e.g., in toy cars [55].

Gear Terminology

Gears are defined by a lot of properties, which all has their own specific term. Examples of these are among others the *pitch circle* which is an imaginary circle that is formed around the gear in a certain position, around the center of a tooth. This position is between the *dedendum circle* and the *addendum circle*. The addendum circle is the circle drawn through the top of the teeth, and the dedendum circle is the circle drawn through the bottom of the teeth. The three different circles are visualised in Figure 2.3. These are only some of the many terms that are used to describe gears, numbering up to a total of 20+ terms [36].

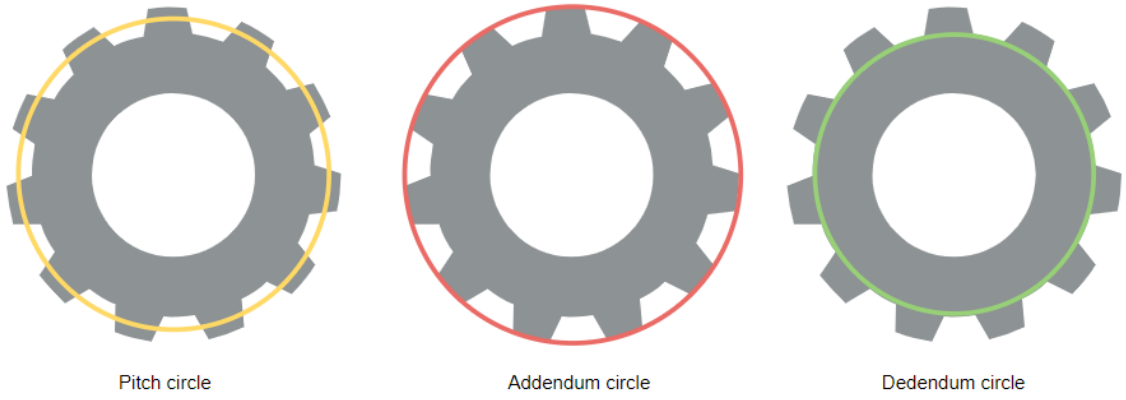

Figure 2.3: Visualisation of the pitch circle, addendum circle, and dedendum circle of a gear.

While gears are essentially mechanical marvels requiring a decent amount of mathematical calculations to create perfect gears, these very detailed calculations are not necessarily required for the project, as the gears that the users are expected to make are more simple. Toy gears are not expected to handle greater amounts of torque as their metal equivalent used in cars and other mechanical constructs.

Users can only gather a few simple measurements from a broken gear, these being the diameter of both the dedendum circle and the addendum circle, along with the number of teeth, and the height of the gear. The diameter of the dedendum circle will be referred to as the outer diameter, and the diameter of the addendum circle will be referred to as the internal diameter. From these measurements

it is at least possible to calculate the shape of a spur gear [33, 22], which can then be converted into G-code.

2.3 G-code

G-code instructions can be used to program CNC-machines, such as 3D-printers, milling machines, and lathes. Going into more detail, G-code is an assembly level language, which is explained in section 2.4, used for controlling CNC-machines. In its core, it is a list of fields, which act as a type of instructions, separated by line endings or white space. A field consists of a letter, which is typically followed by a number, where the number can be either a decimal or an integer. A field can for instance be either a command or a parameter for a command. Parameters are just written after the command, which they are passed to. Table 2.1 gives an overview of the different fields in G-code and their meaning. The "nnn" in the letter column is a placeholder for a number [49].

Letter	**Meaning**
Gnnn	Standard G-code command
Mnnn	Firmware specific command
Tnnn	Select tool nnn
Snnn	Command parameter
Pnnn	Command parameter
Xnnn	Axis X coordinate
Ynnn	Axis Y coordinate
Znnn	Axis Z coordinate
Innn	Parameter X offset
Jnnn	Parameter Y offset
Dnnn	Parameter used for diameter
Hnnn	Parameter used for heater
Fnnn	Feed rate in mm per minute (speed of nozzle)
Rnnn	Parameter used for temperature
Ennn	Length of filament to extrude
Nnnn	Line number

Table 2.1: Different letters and their meaning across flavors of G-code. “nnn” is a placeholder for numbers [49].

Since G-code is an extensive language that covers many different CNC-machines, there are many different developers from CNC-machine manufacturers who add features according to their needs in the firmware. This has resulted in many different machine-specific sub-flavors, especially in the 3D printing industry. Some of these sub-flavors are Marlin [38] and Griffin [60]. Marlin is an open source firmware made to support most mainboards for 3D-printers, whereas Griffin is a firmware

developed and used for a specific series of the Ultimaker printers. This also means that the Griffin flavor is not ensured to run on other mainbords than the ones used in the Ultimaker 3 and S line machines [60]. When wanting to write G-code and print it on a 3D printer, you first have to identify which firmware the 3D printer is running to interpret the G-code, then one can choose the correct flavor for writing G-code to the 3D printer.

Overall, G-code contains two types of commands, these being a set of G-commands and a set of M-commands. The G-commands are used for controlling the CNC-machine's tools. Many of the G-commands are generic movements that nearly every CNC-machine can perform. M-commands, on the other hand, covers different miscellaneous commands that can be different from machine to machine. These M-commands are typically used to control machine functions like `M85<seconds>`, which is an inactivity timer that halts or shuts down the printer, if it is inactive for a specified amount of time in seconds. G-code produced by a slicer can be found in .g, .gco or .gcode files, and could look something like the instruction set seen in Listing 2.1.

Listing 2.1: Common example of G-code produced by a slicer.

```
G92 E0
G28
G1 X2.0 Y2.0 F3000
G1 X3.0 Y3.0
```

From Listing 2.1 the syntax of G-code is clear: It is a set of single letters, each followed by a number. The command that is executed first in this example is `G92 E0`, which resets the position of the nozzle, i.e., sets the amount of filament that has been extruded to zero. Command `G28` then resets the position of the nozzle, by making the printer head move back to the origin of the specific printer. The next command is `G1 X2.0 Y2.0 F3000`, which is an example of a command with three parameters. This command will set the position of the nozzle to 2.0 mm on the x-axis, and the position on the y-axis to 2.0 mm, and set the feed rate to 3000 mm pr. minute. The feed rate refers to the speed at which the nozzle moves. The last command in the example is `G1 X3.0 Y3.0`, which will set the position of the nozzle to x = 3 mm and y = 3 mm. The printer moves in a straight line between the coordinates.

There always has to be a collection of start G-code commands and end G-code commands when 3D printing, making sure that the printer heats up and cools down correctly. The example of G-code commands in Listing 2.1 are similar to the some of the commands that might by used before and after 3D printing, where the `G28` command for resetting the position of the nozzle is done both before and after 3D printing. Besides the start and end G-code commands, the most common commands used when printing a 3D model are `G1` and `G0`, for linear moves with or without extrusion (if the E parameter is specified), respectively [49].

In conclusion, the G-code commands are divided into G-commands and M-commands, which are used to control different CNC-machines such as 3D printers. There are a lot of different G-code commands, and a full list of these can be found in [49]. However, the most relevant commands for this project are `G0` and `G1`, as well as the commands for heating up and cooling down the printer.

2.4 Programming Languages

As mentioned in the previous section, G-code is an assembly level programming language, also called a low-level language, which is one out of three main categories for programming languages. The other two categories are high-level languages and machine languages [14]. Figure 2.4 illustrates the three types in a hierarchy, where a higher level indicates a more abstract language. At the top, languages like C, FORTRAN, and Pascal are found, which are examples of high-level programming languages. At the level below them comes assembly languages, such as G-code. The lower the language is placed in the figure, the closer it is to the hardware [14].

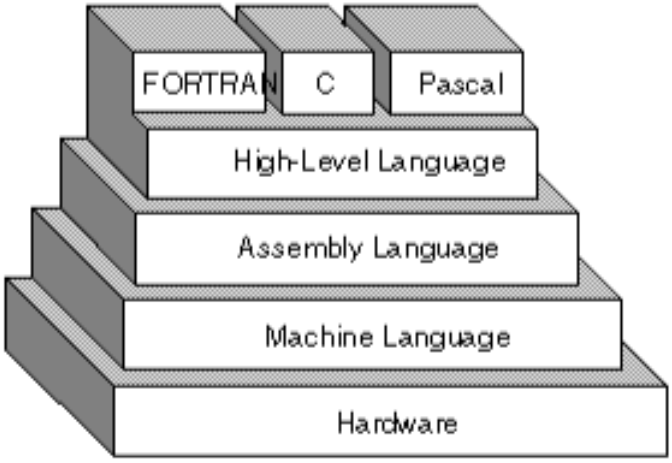

Figure 2.4: Programming languages hierarchy [14].

A high-level programming language is easier to read, write and maintain compared to a low-level programming language, as seen in Table 2.2, and it enables a programmer to write programs that are less dependent of a particular type of computer or machine in general. However, programs written in a high- or assembly level language must be translated into machine language before they can be executed, which is done by a compiler or interpreter, as hardware cannot execute high-level language commands. In terms of speed, programs written in a low-level language can be much faster, since programmers have more control over data storage, memory, and computer hardware. In general, the low-level languages are closer to the hardware than high-level languages, which are in return closer to human languages, i.e., more similar to the English language and its syntax. Table 2.2 provides a schematic overview of the main differences between high-level languages and low-level languages

compared to each other.

High-level language	**Low-level language**
Programmer friendly	Machine friendly
Less memory efficient	Highly memory efficient
Easy to understand for programmers	Tough to understand for programmers
Simple to debug	Complex to debug
Simple to maintain	Complex to maintain
Portable	Non-portable
Can run on multiple platforms	Machine dependent
Needs compiler or interpreter for translation	Needs assembler for translation
Widely used for programming	Not commonly used in programming

Table 2.2: High-level language vs Low-level language [14].

Since G-code is an assembly language, it is not very programmer friendly, and therefore it can be tough to write, debug and maintain etc., as mentioned before and illustrated in Table 2.2. However, G-code is essential for 3D printing, since 3D printers, as explained in section 2.1, can only execute commands written in G-code. This means that a high-level language that would be translated into G-code instructions would make it much easier for programmers to understand and write code for 3D printers, without having to consider each of the machine instructions, as required when writing programs in the G-code language.

2.5 Conversion of High-Level Code to Low-Level Code

As mentioned in section 2.4, programs that are written in a high-level language must be translated into a machine language before they can be executed, as hardware cannot execute high-level language commands. In the case of 3D printers, the printer firmware can convert G-code into machine code, meaning that a high-level program only has to be converted to low-level code, before it is transferred to the printer. There are two different ways to perform this translation, either by using a compiler or by using an interpreter. The purpose of both a compiler and an interpreter is to make a translation from a higher level language into a lower level language, however their translation approaches are different [24, 19].

A compiler or interpreter will take a high-level language as its source and translate it into a specific target language. The compiler takes a source program and translates the entire program into the target language, and if errors occur during the compilation, the compiler will terminate, and display the error messages. As a compiler translates the entire program, it is the most efficient solution, if the program is expected to be run multiple times. An interpreter however translates one statement at a time into a target language, while the program is running, instead of translating the entire program

ahead of its execution. This means that if errors occur, the interpreter has information about the position of the error in the source code, making interpreters suitable for debugging [24]. However, the execution of a program made with a compiler is 10-100 times faster than the same program executed using an interpreter [53].

3D printers can execute G-code, which means that any code written in another language must be translated into G-code for the printer to understand it. In most cases, when a digital 3D model made in a CAD program is converted into printer-executable code, it is done using a slicer. The slicer works as a more advanced translator, that will divide the 3D model into layers and then convert it to G-code commands. This means that a translator for the language that is being developed in this project is needed, and is this case a compiler will be the most suitable option. This is among others due to the facts that the entire program is run every time, and one might want to 3D print one program several times. As the users are expected to make quite small programs in the language, debugging is less of an issue, thus a compiler would therefore be the most optimal choice. A compiler with the same functionality as that of a slicer will therefore be developed to translate programs written in the language into G-code commands.

2.6 Common Responsibilities of Software Generating G-code

The most common group of programs used to generate G-code for printing of 3D models is the slicing software, often just referred to as *slicers*. Slicers generally perform the following operations in this order [58]:

- Error handling
- Slicing
- Generate support
- Generate infill
- G-Code generation

Each operation will be explained in the following sections. This project aims to implement most of these. However, support generation might be irrelevant depending on which gear types are supported in the final program.

Error handling

First, the slicer analyzes the 3D model and checks for errors, like if the model fits within the printer and whether the geometry of the 3D model contains encapsulated surfaces. Some slicing software, like CuraEngine [58], which powers Ultimaker Cura, can even fix minor errors such as gaps in the 3D model. This step also orients the model for the best printing quality.

Slicing

Next up, the model will be sliced into 2D layers, as visualised in Figure 2.5. The height of each layer is adjustable and is a trade-off between quality and printing time. Usually, the result of the slicing step is 2D polygons that define the outlines of the layer. This is the main functionality of a slicer, hence the name, but a good slicer will also generate support and various types of infill [59].

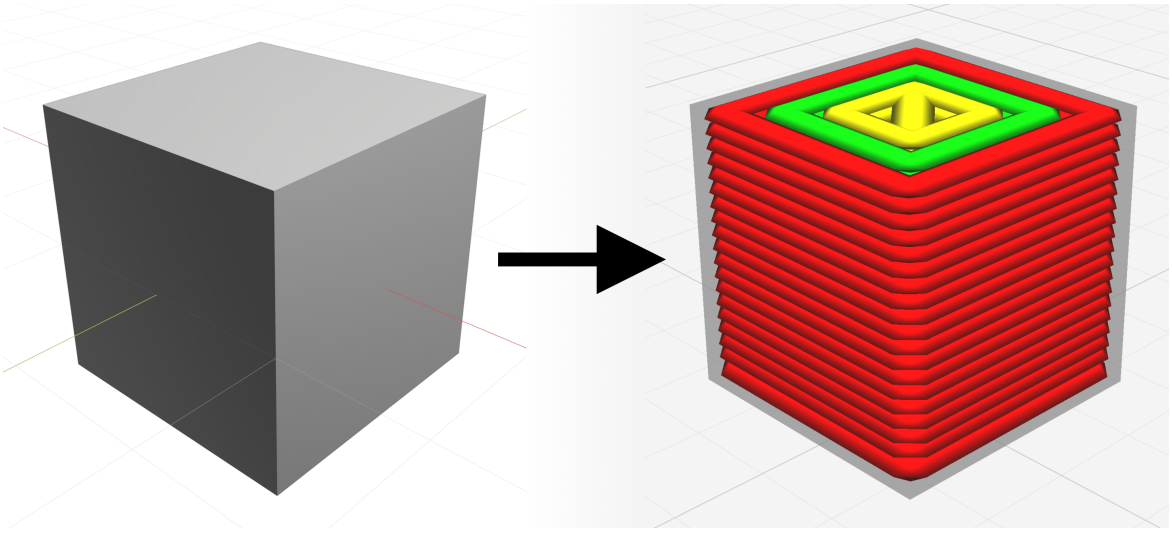

Figure 2.5: The slicing process: A cube is sliced into 2D layers. The left image is a screenshot from Blender [16], and the right image is the same cube after being sliced in Ultimaker Cura [59]

Generate support

Because 3D printers work by stacking layers on top of each other, it must always have a surface to stack the next layer on. So, when printing 3D models with an overhang or a bridge, the slicer must generate some kind of temporary support structure that can be removed after printing. Figure 2.6 shows how Ultimaker Cura generates support for a model with an overhang. As can be seen, support structures waste a lot of material and printing time, so it is favorable to only generate support where it is actually needed. The general rule is that if the angle of the overhang is less than 45 degrees from the vertical axis, it should be possible to print without supporting structures [18]. The angle depends

on the 3D printer, its settings, and which filament is in use. Using this information, it is possible to improve the support structure by creating a tree-like structure that only has a small footprint on the base layer and then branches out to support the whole structure. This reduces printing time and material usage with the only downside being that it takes slightly longer to slice the model [25].

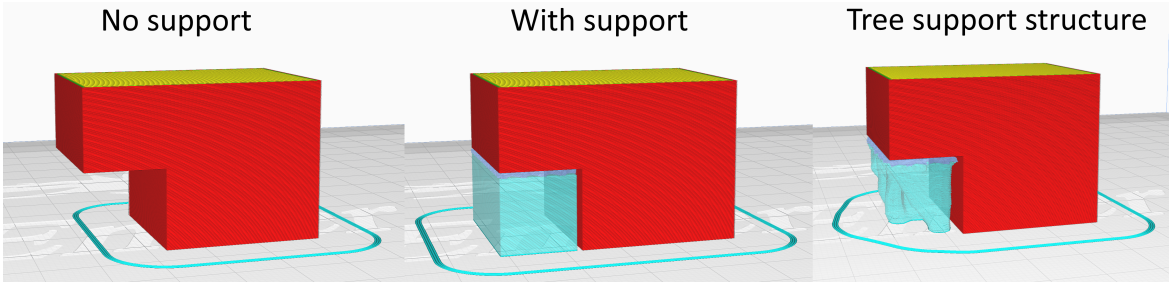

Figure 2.6: Various types of support generated in Ultimaker Cura [59].

Generate infill

Most 3D models only define the borders or shell but leave the inside of 3D objects hollow. Infill refers to a structure on the inside of the 3D model, which is responsible for supporting the outer shell. Without it, the print will be weak and fragile. Common parameters to adjust in most slicers are infill pattern and density. The infill pattern is the structure of the infill, and each pattern has advantages and disadvantages. While one pattern might improve the strength, it will probably also increase printing time and material usage [59]. Some common infill patterns can be seen on Figure 2.7.

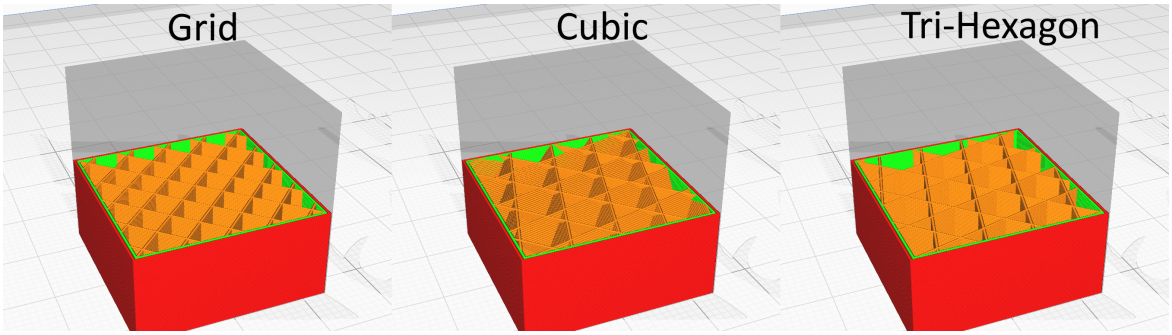

Figure 2.7: Various types of infill patterns with 20% density. Generated in Ultimaker Cura [59].

Infill density is usually defined as a percentage where 0% is hollow and 100% is completely solid. A higher density results in a stronger part, but it will also increase material usage and printing time [59].

G-code generation

Finally, the infill, the support, and the main 3D model are converted to G-code. Because printers are different, each G-code file must be specifically made for the 3D printer that will be used, with

respect to its G-code flavor and specific configurations. Especially the start and end of the G-code file are important to adjust to the specific printer configuration, because these parts contain important settings, such as bed and printing temperatures. Different filaments require different printing temperatures, so many slicers also include settings to change which filament is used, see for instance the Ultimaker Cura slicer [59].

This concludes the presentation of the responsibilities of software generating G-code, meaning that the analysis of the problem field is now complete. In total, the problem analysis provides a solid foundation for the formulation of a problem statement, which will be given in the next chapter.

Chapter 3

Problem Statement

As mentioned in chapter 1, 3D printers are becoming more available to private users, who could use them, e.g., to replace broken gears in toys and tools. However, it is still complicated as a private person to design gears using CAD programs. Another option would be for the users to write G-code commands themselves, but from chapter 2 it becomes clear that it is difficult for humans to write and read programs written in assembly languages, which G-code is. On the other hand, programs written in high-level languages are much more readable and understandable for humans, but they require a translation to G-code, before they can be executed on a 3D printer. The translation can be done using a compiler, which should take into consideration the limitations of 3D printers as well as program validation, design error handling, and infill and G-code generation.

As there are currently no high-level programming language for creating .gcode files for custom gears to be 3D printed, the following problem statement has been set:

> *How can a high-level programming language be developed to provide easy generation of 3D printer instructions for custom gears for inexperienced users?*

Here, gears primarily refer to simple spur gears, and secondarily also to bevel gears. The target audience are people with no or very little experience with working with CAD programs and 3D printing in general, and with the development of a programming language, follows – of course – also the development of a compiler, which can generate a G-code file from a valid source program.

In the following chapters, the requirements for the solution will be given as well as a specification of the programming language, which will lead to a solution for the present problem statement. As the language will be designed for gear design, it will be called “The MMAG Language”, where MMAG is short for “Make Me A Gear”.

Chapter 4

Language Definition

Three different angles have been applied, in order to define the language: First, the requirements for the language have been identified to ensure that the MMAG language will indeed solve the problem it is set to solve. Next, the syntax of the language is defined with respect to the desired characteristics of the language, and finally, the interpretation – the semantics – of the language has been formalised, such that no doubt can be made about the meaning of a syntactically correct source program written in the MMAG language.

4.1 Requirements

To develop a solution for the problem statement given in chapter 3, a set of requirements have been defined. These are mainly based on the problem statement, but they also take inspiration from Sebesta's Language Evaluation Criteria [53] as seen in Table 4.1. The criteria are normally used for evaluation of programming languages, as they provide a foundation for assessing the readability, writability, and reliability of programs written in a specific language by evaluating different characteristics of the language, e.g., the simplicity and expressivity of the language. To ensure high quality in the MMAG language, these criteria are taken into consideration beforehand, by selecting three key characteristics, which will be assigned a higher priority already from the beginning of the design phase. Finally, the requirements are analysed and structured using the MoSCoW method.

For this language, readability and writability are in focus, as the aim is to develop a language which is easy to learn and use compared to a CAD program. Therefore, simplicity will naturally be of high priority. Simplicity can be improved by only having a few ways of achieving a specific goal, i.e., avoiding feature multiplicity, or by assigning only one meaning to a specific symbol or string (avoiding operator overloading) [53]. Simplicity is a subjective matter though, and therefore it can

	Criteria		
Characteristic	Readability	Writability	Reliability
Simplicity	*	*	*
Orthogonality	*	*	*
Data Types	*	*	*
Syntax Design	*	*	*
Support for abstraction		*	*
Expressivity		*	*
Type Checking			*
Exception Handling			*
Restricted Aliasing			*

Table 4.1: Sebesta's Language Evaluation Criteria [53].

be difficult to achieve a language, which is objectively simple.

Another important characteristic for this language is orthogonality, as this covers the ability of interaction between program parts. When certain program parts cannot be applied in the same manner as others, it increases the number of rules, a user must know. This way, simplicity is also closely related to orthogonality [53].

Finally, exception handling is of high priority in the design of the compiler for this project. Normally, exception handling refers to the user's ability of catching run-time errors in the program, they are developing, but in this case exception handling is used to describe the exceptions that the compiler handles, as it will improve the overall user experience, e.g., by enabling the compiler to run even if minor syntactical mistakes are made in the source program. Furthermore, proper exception handling with informative messages for the user, will also support the familiarisation of the user with the language, if suggestions are made to how the user can solve the problem.

In conclusion, the three most important characteristics of the MMAG programming language are simplicity, orthogonality, and exception handling. These are taken into consideration when the formal requirements for the language are established.

The requirements for the MMAG language and compiler can be seen in Table 4.2, and they have been prioritised using the MoSCoW method. When prioritising requirements using MoSCoW, the requirements will be divided into four different categories, these being: *must have*, *should have*, *could have*, and *will not have*, as represented by the four capital letters in the name of the method. The *must have* requirements are requirements that have to be fulfilled, as these are essential for the program to function. The *should have* requirements add significant functionalities, however the program can still function without them. Requirements that are prioritised as *could have* are requirements that are nice to have, and the requirements prioritised as *will not have*, are the ones that

will not be included in this development iteration [47]. The *must have* and *should have* requirements are prioritised the highest, and for the solution that is being developed, these should be fulfilled as a minimum.

ID	Requirement
M1	Easy design of simple spur gears using the MMAG programming language.
M2	Compile a program accepted by the language into a G-code file that is accepted by a 3D printer.
M3	Informative error messages, if a program is not accepted by the language.
M4	One way to assign a value to a gear attribute in the language.
M5	Is a Command Line Interface .
S1	Error messages with information on missing assignments of attributes, if such occur.
S2	Error message, if the design results in an object with dimensions that exceed the printer limitations.
S3	One purpose for each keyword.
S4	Loosely typed variables, which will be interpreted as either integer or double variables with implicit conversion.
S5	Similar rules of usage for variables, gear attributes, and numerals.
S6	Automatically generate 20% infill for all structures.
S7	Allow the use of arithmetic expressions.
C1	Easy design of simple bevel gears.
C2	User-defined infill percentage.
C3	Assumptions on what the program was intended to do, when syntactical errors are encountered. Notify the user about the assumptions.
C4	An embedded resource with start- and end code for the most commonly used 3D printers.
W1	Automatically generate support when needed.
W2	Create multiple gears in one file so that they are printed in the same session.
W3	Take filament width into consideration.
W4	Have a graphical user interface that visualises the designed gear.

Table 4.2: Requirements prioritized using the MoSCoW method.

Together, the requirements in Table 4.2 constitute a foundation for the latter work on the project. The first section of requirements (from M1 to M5) defines the minimum requirements for a solution. When they are fulfilled, it will be possible to design simple spur gears in the MMAG language (M1). The simple version of a spur gear is a solid structure, where the outer edge of the gear is geometrically correct. As stated in M2, the designed gear must be converted into a G-code file, such that it can be printed on a 3D printer. The compiler responsible for the conversion from MMAG to G-code, should have a command line interface (M5). Of course, it would be nice for the user to have a graphical user interface capable of visualising the designed gear, but that is out of the scope for this project

and will not be included (W4). Furthermore, the compiler must provide informative error messages in case the MMAG program is not accepted (M3). Finally, for simplicity, there should only be one way to assign measured values to gear attributes when writing the program (M4). Together, these requirements form up a very basic, yet functional solution for the problem.

The next block of requirements (from S1 to S6) are the *should have* requirements. If they are implemented, the solution will be expanded with additional error messages (S1, S2) and the possibility of writing arithmetic expressions (S7). It will also be possible to save values (integer or decimal numerals) in variables, which will be loosely typed and implicitly converted (S4). To enhance the simplicity of the language, the variables, numerals, and gear attributes should be used in the same ways (S5), and each keyword in the language should also only have one meaning (S3). Finally, 3D prints are normally not completely solid structures, but are filled in with a printed structure as shown on Figure 2.7, thus requirement S6 states that by default, 20% infill should be generated for the 3D print. The value of 20% has been chosen, as this is the standard value for most slicers [2].

The third section of requirements in Table 4.2 are the *could have* requirements. If they are fulfilled, the user will be able to print simple bevel gears as well as spur gears (C1). The user will also be able to set the infill percentage themself (C2), which will be helpful for testing the dimensions of the printed gear, by quickly printing a gear with an infill percentage of 0, or the user might be designing a gear that will be used to transmit a very strong force and requires a 100% infill percentage. User-defined infill percentage allows the user to customize the gear to exactly their needs. Requirement C3 will allow the program to run, even if minor mistakes are made in the source program, and C4 will ensure that the generated G-code file is ready to be printed, without further actions from the user. If C4 is not implemented, the user will have to add the required start- and end code manually.

Finally, the fourth section contains the *will not have* requirements. For now, it is not expected that it will be possible to print multiple gears in the same printing session (W2), though multiple gears in one G-code file would allow the user to start the printer once to print multiple gears. Also, the compiler will not add any support (W1), which is not necessary either when printing the simple gears that can be produced using the MMAG language. Finally, the MMAG language will not take the width of the filament into consideration when generating the tool paths (W3). This means that the user might have to print a couple of gears, before the precise measurements are achieved, but as gears are relatively small with low printing times in most cases, this is considered an acceptable compromise. As mentioned, the compiler will not provide a graphical user interface either (W4).

Together, this section specifies the requirements for the MMAG language and compiler. In the next couple of sections, a language is developed and defined in a way that ensures that the requirements can be met.

4.2 Syntax

The syntax for a programming language describes the valid structure of programs written in the language, and this is formally specified with the use of a context-free grammar [19]. In this section, a specification of the tokens - special characters, keywords, and input - used in and accepted by the language will first be provided. Then a context-free grammar will be described, followed by a classification of the grammar. The syntax is developed based on several example programs similar to the one in Listing 4.1. The language allows assignments of values to keywords that are used to create a gear for a specific printer model. It is also possible to declare new variables using the keyword `Var` and assigning a value using an arithmetic expression.

The development of the MMAG language started by writing several example programs based on the combined intuition of the authors, while bearing in mind the purpose and expectations for the usage and users of the language. Throughout several iterations, the language was refined until it took the shape seen in Listing 4.1. A broader selection of example programs can be seen in Appendix A.

Listing 4.1: Example program.

```
1  /* settings */
2  PrinterModel = Ultimaker2Plus;
3  Type = SpurGear;
4
5  /* remember to subtract filament width */
6  Var diameter = 50 - 2 * 0.2;
7
8  /* model */
9  InfillPercentage = 0.3;
10 OuterDiameter = diameter;
11 InternalDiameter = OuterDiameter - 10;
12 NumberOfTeeth = 5 * 2;
13 Height = 10.5;
```

In the example program in Listing 4.1, the user first selects the printer model on line 2, which in this case is an Ultimaker 2+. The user chooses to make a spur gear on line 3. Then, on line 6 the user makes a variable for the diameter. As can be seen in the comment in line 5, the user subtracts the width of the filament from the diameter to counter this. Then the gear is designed from line 9 to line 13, where values are assigned to the relevant keywords. This can be considered the core part of the program, as the values of these keywords are used to generate the .gcode file for a gear, when compiling the program.

4.2.1 Token Specification

A program written in MMAG can be interpreted as a stream of tokens, where each word in the program complies with the rules for a valid token. The set of these tokens have therefore been defined to describe which tokens are valid in a MMAG program. The tokens have been identified based on several example programs similar to the one in Listing 4.1. In a valid program, the tokens will be written as some characters, and for each token there is a corresponding regular expression that describes which characters a specific token can be written as [19, 53]. The tokens and their corresponding regular expression constitute a token specification, and the token specification for this language can be seen in Table 4.3.

Terminal	**Regular Expression**	**Interpretation**
eofSymbol	EOF	End-of-file symbol.
endofstmtSymbol	;	Semicolon
assignSymbol	=	Equals sign
plusSymbol	+	Plus sign
minusSymbol	-	Minus sign
timesSymbol	*	Asterisk
divideSymbol	/	Slash
lparenSymbol	(	Left parenthesis
rparenSymbol	)	Right parenthesis
printerAttributeKeyword	PrinterModel	Exact string
vardclKeyword	Var	Exact string
typeAttributeKeyword	Type	Exact string
gearTypeKeyword	SpurGear	Exact string
numeralString	-? [0-9]+ (. [0-9]+)?	Numeral, decimal or integer, potentially negative, and with a dot as decimal separator
idString	[a-zA-Z]+	Letters, minimum length of one character
printerNameString	[a-zA-Z0-9]+	Letters and numbers, minimum length of one character
commentstartSymbol	/*	Slash followed by an asterisk
string	.*?	Any characters, can be empty. As short as possible
commentendSymbol	*/	Asterisk followed by a slash

Table 4.3: Token specification for the language.

In the regular expressions in Table 4.3, the symbols '?', '|', and '()' respectively expresses optionality, selection, and grouping, and the '+' expresses one or more repetitions. The regular expressions describe the possible ways a token can be written in a program. This means that in a valid program for MMAG, e.g., the assignSymbol token has to be written as a '='. Looking into the regular expression

for the numeralString token, it is a bit more complex than the regular expressions for the other tokens. It says that for a numeralString it is optional to have a '-' in the beginning, meaning that the numeralString can be either a negative or positive number. The number can be either an integer or a decimal numeral, depending on whether or not a '.' is present. In the regular expression, this is described as '[0-9]+ (. [0-9]+)?', resulting in an integer numeral, if the optional part is omitted, and a decimal numeral when the optional part is included.

Regarding the token for comments, a lazy regular expression is used for the string, as indicated by the question mark after the asterisk, such that the commentString ends the first time a commentendSymbol occurs [27]. This means that the language does not accept nested comments.

4.2.2 Context-Free Grammar

Based on the example programs, the grammar has been developed and re-written until a set of 30 legal test-programs were accepted by the grammar, and 10 illegal test-programs were rejected, to ensure that any valid program is accepted, but invalid programs are rejected.

The grammar is described as a set of production rules in extended Backus-Naur Form (EBNF), where the derivations of each non-terminal, i.e., class of syntax structure, are given on the right-hand side as a string of non-terminals and terminals. Terminals will be referred to as tokens and they can not be derived any further. Through a series of derivations, it should be possible to derive the non-terminals of the grammar into a sequence of non-terminals, or tokens, which cannot be further derived [19, 53]. Writing the grammar using EBNF makes it possible to syntactically describe whether or not specific expressions can be repeated or if they are optional by using symbols in a similar way as with regular expressions. In Table 4.4, the symbols used to describe the grammar in EBNF are explained [19].

Symbol	**Explanation**	**Example**
< >	Indicates a non-terminal	<stmt>
|	Expresses or	plusSymbol | minusSymbol
*	Expresses zero or more repetitions	(<stmt> | <comment>)*
?	Expresses optionality	(assignSymbol <aexp>)?
( )	Expresses grouping	(plusSymbol | minusSymbol)

Table 4.4: Different symbols used in EBNF [53, 19].

The full grammar for the developed language can be seen in Table 4.5. The naming of the different terminals in the grammar, the ones not enclosed by angle-brackets, are characterised by different suffixes, these being either "String", "Symbol" or "Keyword". The suffix "Symbol" indicates that the associated tokens are predefined symbols, and "Keyword" indicates that the tokens are predefined words. "String" indicates that the tokens are not predefined words or symbols, but that the user has

to insert a string or number of their own choice.

<program>	→	<printerSelection> <gearTypeSelection> (<stmt> | <comment>)* eofSymbol
<printerSelection>	→	printerAttributeKeyword assignSymbol printerNameString endofstmtSymbol
<gearTypeSelection>	→	typeAttributeKeyword assignSymbol gearTypeKeyword endofstmtSymbol
<comment>	→	commentstartSymbol string commentendSymbol
<stmt>	→	(<dcl> | <assignment>) endofstmtSymbol
<dcl>	→	vardclKeyword <id> (assignSymbol <aexp>)?
<assignment>	→	<id> assignSymbol <aexp>
<aexp>	→	(<aexp> (plusSymbol | minusSymbol))? <f>
<f>	→	(<f> (timesSymbol | divideSymbol))? <p>
<p>	→	<id> | <n> | lparenSymbol <aexp> rparenSymbol
<n>	→	numeralString
<id>	→	idString

Table 4.5: Context-Free Grammar written in EBNF. The production rules related to arithmetic expressions have been highlighted.

From the derivation of <program>, it can be seen that each program must first contain a specification of the printer, on which the gear should be printed, followed by the type of gear that is to be created. The selection of printer and gear type is followed by a sequence of statements (<stmt>) and comments (<comment>), and the program terminates when it reaches the end of the source file (eofSymbol).

The sequence of statements and comments is generated by the content within the parenthesis followed by a "*", i.e., (<stmt> | <comment>)*. Here, the parentheses and the asterisk indicate that the derivation of <program> will contain zero-to-many comments and/or statements, which can be in a mixed order. The derivation of a comment is simple, as it consists only of a start symbol, a text string, and an end symbol. The derivation of the statement, however, can give several different results, these being: a declaration or initialization (<dcl>), or an assignment (<assignment>). The <dcl> non-terminal can be derived into a variable declaration or initialization, depending on whether the optional part in the parenthesis followed by a question mark is included or not. When <dcl> is derived into "vardclKeyword <id>" it is a variable declaration, and if it is derived into "vardclKeyword <id> assignSymbol <aexp>", it is an initialization, thus these two variants fall under the same non-terminal when examining a source program. The simple assignment is rather straight forward, providing the option of assigning the evaluation of an arithmetic expression (<aexp>) to one of the defined variables, including a set of predefined gear attributes which will be introduced in subsection 4.3.3.

The derivations of <aexp>, <f>, and <p> are less intuitive, which is a direct consequence of a desire

to implement the typical precedence of operators, i.e., that multiplication and division has higher precedence than addition and subtraction, while ensuring left associativity. This leads to several production rules, as seen in the highlighted rows in Table 4.5. As the grammar is now, multiplications and divisions are placed lower in the parse tree than additions and subtractions, and expressions encapsulated in parentheses are placed even lower than that - meaning that expressions in parentheses will be evaluated first, then multiplications and divisions, and lastly additions and subtractions. Inspiration to this was found in [53]. As this is a very common problem, there might be tools which can solve this in a less complicated way - more about this in section 5.1.

4.2.3 Grammar Classification

Many different parsing algorithms can be used to parse a source program, depending on the properties of the grammar. Therefore, in order to choose a suitable method for parsing source programs, the qualities of the proposed grammar have been examined.

Some of the simplest grammars to parse, are those which can be classified as LL(1) grammars, which means that source programs can be parsed with a left-to-right scan generating leftmost derivations [53]. This can be performed with a lookahead of only one token, meaning that one token is sufficient for the parser to determine which production rule has been applied [19].

To examine whether a grammar is LL(1), the `PREDICT` set must be considered. The `PREDICT` set of a non-terminal A contains the first terminals that each derivation of A can lead to. To identify these terminals, the `FIRST` set and potentially also the `FOLLOW` set of A must be found. Here, the `FIRST` set refers to the set of terminals that immediately follow from the derivations of A, as described in Equation (4.1) [19]. In general, non-terminals are represented by capital letters, and terminals are represented by lowercase letters.

$$\texttt{FIRST}(A) = \{a \mid A \Rightarrow^* aB\} \tag{4.1}$$

The `FOLLOW` set contains the `FIRST` sets of the non-terminals following A, and, in a recursive manner, also a selection of the `FOLLOW` sets of the non-terminals following A, if the non-terminals can be derived into the empty string λ. This is described in Equation (4.2) [19], which states that if a non-terminal S can be derived into a sequence of terminals and non-terminals including the non-terminal A, then the terminal following A is in the `FOLLOW` set of A. In the equation, α and β represent arbitrary combinations of terminals and non-terminals.

$$\texttt{FOLLOW}(A) = \{b \mid S \Rightarrow^+ \alpha Ab\beta\} \tag{4.2}$$

Based on the FIRST and FOLLOW sets, the PREDICT set of A can be identified, either as being equal to the FIRST set, or, in the case where A can be derived into λ, the union of the two sets, i.e., $\texttt{PREDICT}(A) = \texttt{FIRST}(A) \cup \texttt{FOLLOW}(A)$. Finally, to assess whether a grammar is LL(1), the non-terminals in the grammar must be evaluated separately. The pairwise intersections of the PREDICT sets of the derivations for all non-terminals in the grammar must be empty. This is equivalent to the set described in Equation (4.3) containing a maximum of one member, as Equation (4.3) contains the set of derivations p for a non-terminal A, for which the terminal a is a part of the predict set [19].

$$P(a) = \{p \in \mathit{Derivations}(A) \mid a \in \texttt{PREDICT}(p)\} \tag{4.3}$$

If only one derivation of A leads to a derivation with some terminal a in the PREDICT set, then the grammar is LL(1) [19]. If, on the other hand, more than one derivation of A has a in the PREDICT set, then the grammar is not LL(1), and another, more powerful, classification must be found.

Returning again to the grammar proposed in this report, it can be seen from Table B.2 in Appendix B that the MMAG grammar does not satisfy the requirement of having distinct PREDICT sets for each derivation of the different non-terminals, as the derivations of the non-terminals <aexp> and <f> have identical PREDICT sets for each of their derivations. Therefore, a lookahead of one token is not enough to determine, which derivation of <aexp> or <f> has been applied. This problem occurs due to the left-recursive nature of the arithmetic expression, which, as described in subsection 4.2.2, is required in order to ensure left-associativity in the evaluation of the arithmetic expressions.

Using an online tool, it can be seen that the grammar is instead LALR(1) [48]. Along with the fact that the language is LL(1) except for two cases of simple left-recursion, this classification will be taken into consideration when choosing a parsing scheme.

Throughout the last couple of sections, the syntax for the valid programs has been established and described through a context-free grammar describing the structure of valid programs, and a token specification describing the valid characters for the different tokens in the language. This means that it is now possible to write legal programs in the proposed language simply by adhering to these rules. The next section will establish the meaning, as well as additional rules, for valid programs, which can not be described with the context-free grammar.

4.3 Semantics

While the syntax describes the structure of the programs written in the language, the semantics define the meaning of them. There are some limitations to the context free grammar as it only describes

the syntax, and the semantics are therefore needed in order to describe how the program should behave. Specifically, the context free grammar is not powerful enough to describe transition rules, scope rules, or type rules, as these need more explanation than the grammar can give. The transition rules are required to describe the meaning of a valid program, and the scope rules and type rules are needed to decide whether a program is valid for a specific language or not [19], as it might be invalid, even if it is syntactically correct. The transition rules, scope rules, and type rules will be described in this section based on a simplification of the syntax called the *abstract syntax*. The two non-terminals representing the printer name and the gear type are not included in the semantics, as special rules apply to these. The values of their associated attributes can only be of type *string*, and they cannot be used in arithmetic expressions.

Overall, the semantics describe how the language can be used to make arithmetic expressions, declarations and assignments with the two types *double* and *int*.

4.3.1 Abstract Syntax

To describe the semantics for a language, the abstract syntax notion can first be used to describe the essential structure of a program by defining different syntactic categories and describing them further. This is possible, as the semantics of a language are not directly dependent on its syntax [29]. The syntax is therefore divided into different syntactic categories, and their associated formation rules are given. The formation rules define the different ways that a specific category can be formatted.

The syntactic categories for this language are numerals, variables, arithmetic expressions, and statements. The different syntactic categories describe the different fundamental parts of this language, and they can be seen in Figure 4.1.

$$
\begin{aligned}
&n \in Num - Numerals \\
&v \in Var - Variables \\
&a \in Aexp - Arithmetic\ Expressions \\
&S \in Stmt - Statements \\
&C \in Cmnt - Comments
\end{aligned}
$$

Figure 4.1: The syntactic categories for this language.

The structure for the arithmetic expressions and the statements are given as a set of formation rules, which are derived from the production rules described in subsection 4.2.2 [19, 29]. The formation rules can be seen in Figure 4.2. The formation rules for numerals and variables have not been described, as these are expected to be written accordingly with the regular expressions for *numeral-*

String and *idString* from the token specification seen in Table 4.3. The same applies to comments, which furthermore do not affect the state of the system.

$$a ::= n \mid v \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1 / a_2 \mid (a_1)$$
$$S ::= \text{"}Var\ v;\text{"} \mid \text{"}Var\ v = a;\text{"} \mid \text{"}v = a;\text{"} \mid S_1 S_2$$

Figure 4.2: The formation rules for the two syntactic categories *Arithmetic Expressions* and *Statements*.

Based on the syntactic categories and their corresponding formation rules, the transition rules can now be defined.

4.3.2 Transition Rules

To define the semantics for the MMAG language, its transition rules are first described. The transition rules form up the structural operational semantics, which describe the meaning of a valid program [19, 29]. In this section, the structural operational semantics will simply be referred to as semantics.

The semantics can be given in two different ways, either as big-step semantics or small-step semantics, which describe the computations in two different ways. The big-step semantics describe the entire computation with no intermediate steps, whereas the small-step semantics also describe the intermediate steps in a computation. The semantics for this language will be described using big-step semantics (BS) defining the meaning of each of the formation rules for the arithmetic expressions and the statements. This results in a set of transition rules, where each transition corresponds to an entire evaluation of either an arithmetic expression or a statement. The transition rules are written as shown in Equation 4.4:

$$\frac{s \vdash a_1 \rightarrow x_1 \; s \vdash a_2 \rightarrow x_2}{s \vdash a_1 + a_2 \rightarrow x}, \; x = x_1 + x_2 \tag{4.4}$$

As seen here, the transition rules are characterized by having a conclusion below a division line with one or more premises above it. If there is no premise, the division line will be omitted. Some transition rules might also have a side condition, which will be written on the right side of the division line. The example in Equation 4.4 describes a state s, in which there are the arithmetic expressions a_1 and a_2 that can be evaluated to the values x_1 and x_2, respectively. Then it can be concluded that in the same state, s, the expression $a_1 + a_2$ can be evaluated to x, where $x = x_1 + x_2$ [19, 29]. The semantics for the arithmetic expressions can be seen in Figure 4.3, and the semantics for the statements can be seen in Figure 4.4.

$$[Num_{BS}]\ s \vdash n \to x, x = \mathcal{N}[n]$$

$$[Var_{BS}]\ s \vdash v \to x, s(v) = x, \text{ and } x \text{ is not "undeclared"}$$

$$[Plus_{BS}]\ \frac{s \vdash a_1 \to x_1\ s \vdash a_2 \to x_2}{s \vdash a_1 + a_2 \to x},\ x = x_1 + x_2$$

$$[Minus_{BS}]\ \frac{s \vdash a_1 \to x_1\ s \vdash a_2 \to x_2}{s \vdash a_1 - a_2 \to x},\ x = x_1 - x_2$$

$$[Mult_{BS}]\ \frac{s \vdash a_1 \to x_1\ s \vdash a_2 \to x_2}{s \vdash a_1 * a_2 \to x},\ x = x_1 * x_2$$

$$[Div_{BS}]\ \frac{s \vdash a_1 \to x_1\ s \vdash a_2 \to x_2}{s \vdash \frac{a_1}{a_2} \to x}, x = \frac{a_1}{a_2} and\ a_2 \neq 0$$

$$[Parent_{BS}]\ \frac{s \vdash a \to x}{s \vdash (a) \to x}$$

Figure 4.3: The big-step semantics for the arithmetic expressions.

The semantics of the arithmetic expressions in Figure 4.3 describe the meaning of the formation rules for arithmetic expressions. They show in rule $[Num_{BS}]$ that a numeral n maps to the value x, if there exists a rational number that corresponds to n. Here, it is assumed that there exists a function $\mathcal{N} : n \to \mathbb{R}$, where n is a numeral that the function maps to a real number. In rule $[Var_{BS}]$, it is stated that a variable v maps to its value in the current state, though if the variable is not declared yet, it will return "undeclared" which will indicate that the program is invalid. The rest of the rules describes how basic mathematical operations are carried out in the language. These follow the most common applications of the operators.

The semantics for the statements can be seen in Figure 4.4. In rule $[Dcl_{BS}]$ it is stated that a variable can be declared by going into a state containing the identifier with the value *null*. This is only allowed, if the variable v is not already declared. The declaration of a variable does not specify the type of the variable. This is specified dynamically, when a value is assigned to the variable. When an initialization is made, it follows from rule $[Init_{BS}]$ that the variable v is contained in the resulting state after the execution of the statement, and that the value of v will be x, given that a maps to x, and – again – that v has not been declared earlier. The $[Assign_{BS}]$ rule shows how an assignment is executed, and the $[Comp_{BS}]$ rule show that several statements can be executed after each other.

Together, the transition rules describe what the different parts of a program mean.

4.3.3 Scope Rules

The scope rules for a language describe where the declared variables within a program can be used. Variables are local if they are declared within a scope, and non-local if they are visible within a

$$[Dcl_{BS}] < Var\ v;, s > \rightarrow s[v \mapsto null],\ s(v) \text{ is "undeclared"}$$

$$[Init_{BS}] < Var\ v = a;, s > \rightarrow s[v \mapsto x],\ s \vdash a \rightarrow x \text{ and } s(v) \text{ is "undeclared"}$$

$$[Assign_{BS}] < v = a;, s > \rightarrow s[v \mapsto x],\ s \vdash a \rightarrow x$$

$$[Comp_{BS}] \frac{< S_1, s > \rightarrow s'' \ < S_2, s'' > \rightarrow s'}{< S_1 S_2, s > \rightarrow s'}$$

Figure 4.4: The big-step semantics for the statements.

scope, but not declared there. How the scope structure is defined depends on the language. The scope structure for the MMAG language follows a so-called monolithic block structure. If a language has a monolithic structure it only has one scope block for the entire program. This means that all the variables are local, and that they can be seen and used throughout the entire program [19]. In this language a variable has to be declared before it can be used, and a variable may only be declared once throughout the program, as described in rule $[Var_{BS}]$ in Figure 4.3 and rules $[Dcl_{BS}]$ and $[Init_{BS}]$ in Figure 4.4. If these conditions are not satisfied, the program is invalid.

For most programming languages, it is also important to state whether the language uses static or dynamic scope rules, the difference here being whether variable values are referred to at the time of declaration or invocation of a method. If the scope is static, a variable will refer to its value at its place of declaration, and if the scope is dynamic it will instead refer to the actual value when the method is invoked [19]. As this language is monolithic, and it is not possible to declare functions in the language, it is not relevant to differentiate between the two, as the output will be the same.

To monitor the state of the variables within the scope of the program, a *symbol table* will be used. A symbol table can also be called a *variable environment*, as it stores references to the values of each identifier [29]. The values themselves must, in addition to their numerical value, carry information about their type, as this information is essential when applying the *type rules* described in the next section.

4.3.4 Type Rules

The type rules for a language describe which variables can be related to each other, and they help describe which operations are valid in a program. Types can be either primitive or composed depending on their complexity. The primitive types are the types that can not be further decomposed to other more simple types, whereas the composed types are a set of objects composed from objects of another type [19, 53]. The different possible types depend on the specific language, and in MMAG there are no methods or functions, and the variables can have numerical values, of either type double or type integer, which are primitive data types. This is formalized in Figure 4.5.

$$T ::= double \mid int$$

Figure 4.5: Types in the MMAG language.

It can be described formally which operations can be performed in a valid program by introducing the environment-store model to describe the type rules for the language [29]. The environment, in the shape of a symbol table as mentioned in subsection 4.3.3, provides references to the values of each variable, which contain a numerical value and the type of the value. It is then assumed that $E(v)$ returns the current type of a variable v in the symbol table. The type must be either a double or an integer in accordance with Figure 4.5, however, if a variable has not been declared or initialized yet, $E(v)$ will return information that the variable has no value, and therefore no type, or that the variable v has not been declared. A selection of the rules for arithmetic expressions are provided in Figure 4.6.

$$[Num1_{EXP}]\ E \vdash n : double,\ n \text{ contains a decimal separator}$$
$$[Num2_{EXP}]\ E \vdash n : int,\ n \text{ does not contain a decimal separator}$$
$$[Var_{EXP}]\ \frac{E(v) = T}{E \vdash v : T}$$
$$[Plus1_{EXP}]\ \frac{E \vdash a_1 : T\ E \vdash a_2 : T}{E \vdash a_1 + a_2 : T}$$
$$[Plus2_{EXP}]\ \frac{E \vdash a_1 : double\ E \vdash a_2 : int}{E \vdash a_1 + a_2 : double}$$
$$[Plus3_{EXP}]\ \frac{E \vdash a_1 : int\ E \vdash a_2 : double}{E \vdash a_1 + a_2 : double}$$

Figure 4.6: Type rules for a part of the arithmetic expressions, namely the numerals, variables, and addition expression.

The type rules in Figure 4.6 describe what the types of different expressions will be. Rules $[Num1_{EXP}]$ and $[Num2_{EXP}]$ show that if a numeral n contains a decimal separator, '.', it will be considered a double, else an integer. The type of a variable will be established by a lookup in the symbol table, as described in $[VAR_{EXP}]$. For addition, the type will remain the same, if the types of the two operands are the same (rule $[Plus1_{EXP}]$), and if they are different, the result will be of type double, as in rules $[Plus2_{EXP}]$ and $[Plus3_{EXP}]$. The same logic applies to all other arithmetic expressions, so for simplicity the formalization of the remaining arithmetic expressions have not been included in this section.

For the statements, it makes no sense to provide type rules for the variable declaration and initialization, as variables do not get a value at declaration, and at initialization any legal type in T can be assigned to the variable. The type rules for assignments and composite statements are shown in Figure 4.7. The rules for statements do not show the type of an output, but are simply indicators of whether a statement is well-typed. “ok” is used to indicate a well-typed statement [29].

$$[Assign1_{STM}]\ \frac{E \vdash v : \text{"}uninitialized\text{"}\ E \vdash a : T}{E \vdash \text{"}v = a;\text{"} : ok}$$

$$[Assign2_{STM}]\ \frac{E \vdash v : T\ E \vdash a : T}{E \vdash \text{"}v = a;\text{"} : ok}$$

$$[Assign3_{STM}]\ \frac{E \vdash v : double\ E \vdash a : int}{E \vdash \text{"}v = a;\text{"} : ok}$$

$$[Assign4_{STM}]\ \frac{E \vdash v : int\ E \vdash a : double}{E \vdash \text{"}v = a;\text{"} : ok}$$

$$[Assign5_{STM}]\ \frac{E \vdash v : int\ E \vdash a : int}{E \vdash \text{"}v = a;\text{"} : ok},\ v \text{ must be an integer}$$

$$[Comp_{STM}]\ \frac{E \vdash S_1 : ok\ E \vdash S_2 : ok}{E \vdash S_1 S_2 : ok}$$

Figure 4.7: Type rules for statements.

In general terms, what can be seen in Figure 4.7 is that in almost all cases, both double and integer values can be assigned to variables no matter their current type. This design of the type rules has been made to remove the type aspect from the view of the user who writes programs in the language. Specifically, it can be seen in rule $[Assign1_{STM}]$ that if a variable v has only been declared, but not initialized, any integer or double value can be assigned to the variable. When a value a of a specific type T is assigned to a variable v of the same type, the assignment is also well-typed ($[Assign2_{STM}]$). When an integer is assigned to a double ($[Assign3_{STM}]$), the statement is well-typed as well, and the type of the variable v remains the same, which is in contrast to when a double is assigned to an integer ($[Assign4_{STM}]$). In this case, the type of v is implicitly changed to a double. This means that if a variable has once been a double, it cannot become an integer by assigning an integer to it, but an integer variable can become a double variable, by assigning a double to it. This design choice has been made to ensure type safety in the system, as there are some special cases, in which the type of v must remain an integer ($[Assign5_{STM}]$). This is the case for some gear attributes, e.g., the number of teeth on the gear. As the gear attributes are handled as variables, this rule is important to include.

Finally, rule $[Comp_{STM}]$ states that if two statements are well-typed, the sequence consisting of the two statements is also well-typed.

Having the type rules in place now concludes the formalization of the semantics of the MMAG language. In union, the transition rules, scope rules, and type rules assign meaning and additional requirements to programs in the MMAG language, and they provide a solid foundation for the implementation of a compiler that can understand and translate programs written in the language.

Chapter 5

Compiler Implementation

In general, the purpose of the developed compiler is to take a source code file, written in the developed programming language called MMAG, short for Make Me A Gear, and translate it into G-code that 3D printers understand. The compiler itself is written in the C# language, as can be seen on Figure 5.1, using a tool to implement some of the phases in the compiler. The tombstone diagram also shows that the compiler translates MMAG code to G-code.

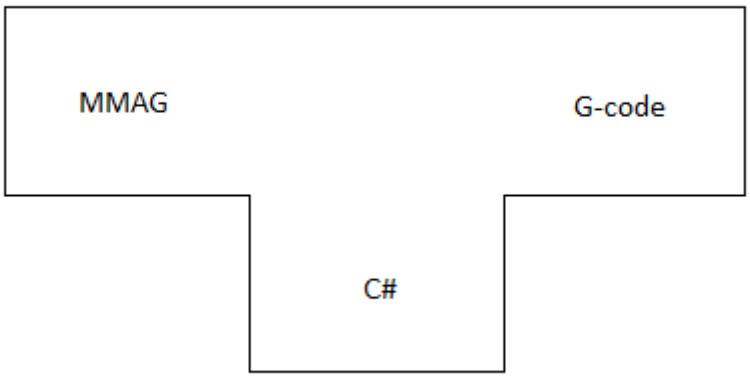

Figure 5.1: Tombstone diagram for MMAG compiler.

The control flow is summarized in Figure 5.2 and the implementation of the compiler, as well as the different phases will be described in the following sections.

In the implementation of the compiler for the programming language MMAG, spur gears are the only gear type supported, and some of the phases in the compiler can therefore only be used to create spur gears.

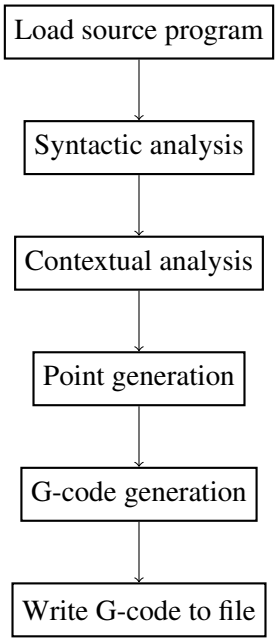

Figure 5.2: The general control flow of the compiler.

5.1 Syntactic Analysis

The first phase of the compiler is the syntactic analysis, which can be divided into two parts: the scanner and the parser. The main goal of the syntactic analysis is to analyze the syntax of a given source program and report if there are any syntactical errors that do not cohere with the context-free grammar described in subsection 4.2.2.

The scanner is the first part of the syntactic analysis. It takes the input that comes as a high-level language, in this case the MMAG language, and reads the characters from the source program. It groups them together into lexemes, "words", for example keywords, operators, identifiers, literals, etc. Each lexeme corresponds to an instance of one of the tokens in Table 4.3, which are defined by a regular expression that the scanner can understand. The regular expressions tell the scanner what to accept for different types of tokens, e.g., if it is acceptable to write with both lowercase and uppercase letters, or if it is allowed to use only integer numerals, only decimal numerals or both, etc. The scanner also removes certain elements like white space, newlines, and comments, which it skips, so they will not be considered by the next step, which is the parser.

The parser then takes the tokens one by one and uses the context-free grammar to construct a syntax tree. It happens by making a derivation that follows the production rules from the MMAG-grammar, which are described in subsection 4.2.2. The syntax tree that is constructed by the parser as the output from the syntactic analysis is usually a concrete syntax tree (CST), that contains a larger amount of details than the abstract syntax tree (AST). As some of the details in the CST might be unnecessary,

it is often preferred to have an AST as output instead of the CST.

5.1.1 Implementation Tools

For this project, the scanner and parser have been generated using a tool, as this has saved time that was instead spent on the development of the later phases of the compiler. There exist many different tools for generating scanners and parsers, and to make an informed decision on which to use, a comparison between some of the most common tools have been made. One of the criteria for the tool was that the generated scanner and parser should be written in C#, as the authors of this report have experience working in this language, and it would therefore make the development process more efficient.

The tools that have been selected for further investigation are the ones that follow in the list below:

- JavaCC
- SableCC
- ANTLR

JavaCC

JavaCC (Java Compiler Compiler) is a tool that generates scanners and parsers. The parsers it generates are top down LL(k) parsers. Defining the grammar is done in a Java-like language. It can only generate parsers and scanners in Java or C/C++.

SableCC

SableCC also generates scanners and parsers. The parser that will be generated is a bottom up LALR(1) parser. Defining the grammar is done using an EBNF-like notation. It can generate parsers and scanners in many languages, C# included. SableCC can be downloaded or used in the Eclipse development environment [51].

ANTLR

ANTLR (ANother Tool for Language Recognition) is a scanner and parser generator. The parser that ANTLR generates is an adaptive top down LL(*) parser. The grammar is defined using an EBNF-like notation. ANTLR can generate the scanner and parser in many languages, C# included. ANTLR can easily be set up, as an ANTLR4 plug-in can be used in different integrated development environments (IDE's) such as Intellij, Eclipse, Visual Studio Code, and Visual Studio. Additionally, ANTLR also supports visual representation of the generated parse trees [11].

Choice of tool

For this project, ANTLR is chosen as this seems to be the most suitable tool. The results of the grammar classification analysis in subsection 4.2.3 showed, that the MMAG language is LALR(1), meaning that the SableCC tool seems like the obvious choice – however, it would only add an unnecessary level of complexity to the scanner and parser, as LALR(1) scanners and parsers are more complex than LL(*) parsers, and the adaptive nature of ANTLR is able to handle the simple left-recursion that caused the grammar to not be LL(1) [10]. Other reasons behind choosing ANTLR is that the generated output files (scanner, parser, and other required classes) are easy to read, also because they are written in C#, and that ANTLR can easily be used in an IDE such as Visual Studio, which the authors are familiar with using. The ability to visualize the parse trees was also one of the important reasons why ANTLR was chosen, as this makes it easier to test the grammar and see, if the source program is being parsed as intended.

5.1.2 Scanner and Parser Generation using ANTLR

As mentioned before, ANTLR is used for generating the scanner and parser for the compiler for the MMAG language. ANTLR generates the scanner and parser from a .g4 file, which is a file that contains the token specifications for the scanner, as well as the CFG of MMAG. The content of the `MMAG.g4` file can be seen in Listing 5.1.

Listing 5.1: MMAG.g4 file. This file defines the grammar and tokens of the language and is used by ANTLR to generate scanner and parser.

```
1 // Grammar
2 program              : printerSettings gearTypeSelection (stmt | comment)* EOF ;
3 printerSettings      : PrinterSettingsKeyword '=' PrinterString ';' ;
4 gearTypeSelection    : TypeAttributeKeyword  '='  GearTypeKeyword  ';' ;
5 comment              : Mycomment ;
6 stmt                 : (dcl | assignment ) ';' ;
```

```
7  dcl                  : VardclKeyword id ('=' aexp)? ;
8  assignment           : id '=' aexp ;
9  aexp                 : '(' aexp ')'                          # parensExpr
10                      | op=('+'|'-') aexp                     # unaryExpr
11                      | left=aexp op=('*'|'/') right=aexp     # infixExpr
12                      | left=aexp op=('+'|'-') right=aexp     # infixExpr
13                      | name=id                               # VariableExpr
14                      | number=NumeralString                  # numberExpr
15                      ;
16 id                   : IdString ;
17
18 // Token specification
19 fragment LOWERCASE   : [a-z] ;
20 fragment UPPERCASE   : [A-Z] ;
21 fragment NUMBER      : [0-9] ;
22 fragment NUMBERDOT   : [.0-9] ;
23 EndofstmtSymbol      : ';' ;
24 AssignSymbol         : '=' ;
25 PlusSymbol           : '+' ;
26 MinusSymbol          : '-' ;
27 TimesSymbol          : '*' ;
28 DivideSymbol         : '/' ;
29 LparenSymbol         : '(' ;
30 RparenSymbol         : ')' ;
31 PrinterSettingsKeyword : 'PrinterModel' ;
32 VardclKeyword        : 'Var' ;
33 TypeAttributeKeyword: 'Type' ;
34 GearTypeKeyword      : 'SpurGear';
35 Mycomment            : '/*' .*? '*/' -> skip ;
36 WhiteSpace           : (' '|'\t'|'\r'|'\n')+ -> skip ;
37 NumeralString        : ( NUMBER+ | ( NUMBER+ '.' NUMBER+ )) ;
38 IdString             : ( LOWERCASE | UPPERCASE | '_' )+ ;
39 PrinterString        : ( LOWERCASE | UPPERCASE | NUMBER )+ ;
```

Listing 5.1 shows the different ANTLR rules for the CFG for the parser and the token specifications for the scanner. Each ANTLR rule consists of the name of the non-terminal followed by a colon and then its derivations. All rules are terminated by a semicolon. As can be seen from line 2 to 16, the rules from the CFG are written in EBNF as shown before in Table 4.5, and from line 19 until 39, the token specification is inputted. Token names for the scanner can either start with a capital letter or they can be written directly inside two single quotation marks, like ’=’. The parser rule names always start with a lowercase letter. As mentioned before certain elements, like white spaces, newlines, and comments, must be removed. This is implemented by letting the scanner know that it should skip them, which can be seen in line 35 and 36. A couple of fragments are defined on line 19 to 22. A fragment is only used to improve the readability of the grammar and can only be called in the different scanner rules. It can not be a token [8].

5.1.3 Precedence

In the MMAG programming language, a statement can include an arithmetic expression, including multiplication, division, addition, and subtraction. Because of that, it is very important that the evaluated result of the expression is correct. To ensure correctness, there should be some precedence rules that states the different precedence levels for the operators, such that multiplication and division have higher precedence than addition and subtraction, and parenthesis have the highest precedence. Having the correct precedence hierarchy ensures that the parser evaluates arithmetic expressions correctly with left associativity. Luckily, ANTLR does handle this in favor of the derivation rule given first, thereby allowing the specification of operator precedence [10]. This is implemented from line 9 to 15 in Listing 5.1 in the parser rule for the non-terminal `aexp`, where the derivation rule that comes first has a higher precedence than the one that follows it below, etc.

5.1.4 Errors report

One of the most important jobs of the syntactic analysis phase, other than generating an AST, is to inform the user if syntax errors have occurred. It could be errors such as missing a semicolon etc. A new class has therefore been defined for this purpose. This class is called `ThrowingErrorListener` and it inherits from the `BaseErrorListener` class and implements the `IAntlrErrorListener` interface that is generated by ANTLR. The `BaseErrorListener` class provides a default implementation of the virtual method `IANTLRErrorListener.SyntaxError()`. A new implementation for the `SyntaxError()` method is given in the `ThrowingErrorListener` class. It notifies the user, when a syntax error has occurred. As described on antlr.org, "This listener's job is simply to emit a computed message, though it has enough information to create its own message in many cases" [9]. The `SyntaxError()` method contains parameters that can indicate the position, where the error has occurred, and the message that needs to be emitted. Figure 5.3 below illustrates an example of an error message for a syntax error, given by the `SyntaxError()` method. This error occurs for the source program example seen in Listing 5.2, where a semicolon is missing on line 2, as the error message describes.

```
Error: line 2 missing ';' at '<EOF>'
```

Figure 5.3: Error message generated by the `ThrowingErrorListener` class.

After implementing a new version of the error listener, the `parser.removeErrorListeners()` and `lexer.removeErrorListeners()` can be called, to remove the default listener that usually writes to the console. and then call `parser.addErrorListener()` and `lexer.addErrorListener()` to add a specialised listener, to notify the user of any errors in a customised way.

Listing 5.2: A source program with a missing semicolon.

```
1 PrinterModel= Ultimaker2Plus;
2 Type = SpurGear
```

Other error messages can be given, depending on where and which error occurs. An example of another error message can be seen in Figure 5.4. Here, the source program in Listing 5.3 causes an error, where it is expecting the Type keyword instead of the Var keyword.

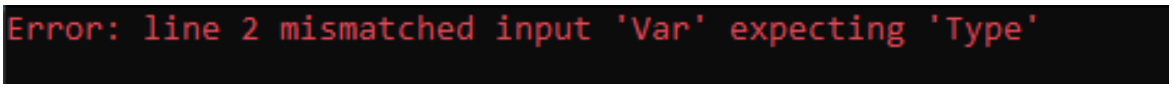

Figure 5.4: Error.

Listing 5.3: A source program with syntax errors.

```
1 PrinterModel= Ultimaker2Plus;
2 Var = 2;
```

5.1.5 Abstract Syntax Tree & Visitor Pattern

The parser outputs a CST, which contains redundant information, such as all the different tokens in a program. Only a smaller part of this information is important for the following phases, and by converting the CST to an AST, the redundant information can be removed. ANTLR generated the CST nodes, but now they must be converted to AST nodes. Therefore, each type of CST node has a corresponding class that is used in the AST. Shown on Listing 5.4 is the root of the AST, a `ProgramNode` that has a `PrinterSettingsNode`, a `GearTypeSelectionNode`, and a list of `StmtNodes` as its properties. `PrinterSettingsNode` and `GearTypeSelectionNode` contains information about the target 3D printer and gear type respectively. Each statement (`StmtNode`) can be either a declaration/initialization or an assignment.

Listing 5.4: The class called `ProgramNode` which inherits from the `Node` class.

```
1 public class ProgramNode : Node
2 {
3     public PrinterSettingsNode PrinterSettings { get; set; }
4     public GearTypeSelectionNode GearTypeSelection { get; set; }
5     public List<StmtNode> Statements { get; set; }
6 }
```

`DclNode` and `AssignNode` are very similar except for their implementation of `AddToSymbolTable()`, which will be described in section 5.2. As shown on Listing 5.5, `DclNode` contains an Id, which is the name of the variable, and an optional arithmetic expression of type `EvaluateNode` that is an abstract class that declares a method `Evaluate` that all derived classes must implement. This will be useful later when building the symbol table in section 5.2. A wide variety of nodes inherits from EvaluateNode such as `AdditionNode`, `MultiplicationNode`, `NumberNode` and `IdNode`. Together they can be combined to build all kinds of different arithmetic expressions.

Listing 5.5: The `DclNode` class.

```
public class DclNode : StmtNode
{
    public string Id { get; init; }
    public EvaluateNode Expression { get; set; }

    public override void AddToSymbolTable(SymbolTable symbolTable)
    {
        ...
    }
}
```

ANTLR provides a class, `MMAGBaseVisitor`, which can be extended to create a visitor that builds an AST. The class `MMAGBaseVisitor` is an empty implementation of the ANTLR generated interface `IMMAGVisitor` which defines a generic visitor for the CST produced by the parser. Now the base class methods can be overridden to implement a visitor that produces an AST. The first method to be called is `VisitProgram()` which is shown on Listing 5.6. It takes a parameter of type `ProgramContext`. This is ANTLRs name for the CST node that corresponds to this compiler's `ProgramNode`.

Listing 5.6: The `VisitProgram()` method. Recursively traverses a CST and returns an AST.

```
public override Node VisitProgram([NotNull] MMAGParser.ProgramContext context)
{
    ProgramNode node = new() { Statements = new List<StmtNode>() };

    foreach (var statement in context.stmt())
    {
        if (statement.dcl() is not null)
        {
            var dcl = (DclNode)Visit(statement.dcl());
            node.Statements.Add(dcl);
        }
        else if (statement.assignment() is not null)
        {
            var assign = (AssignNode)Visit(statement.assignment());
            node.Statements.Add(assign);
        }
```

```
17      }
18      node.PrinterSettings = (PrinterSettingsNode)Visit(context.printerSettings());
19      node.GearTypeSelection = (GearTypeSelectionNode)Visit(context.gearTypeSelection());
20
21      return node;
22  }
```

On line 3 in Listing 5.6, a new instance of `ProgramNode` is created. Then, for each statement in the CST, it is determined whether the statement is a declaration or an assignment. If, for example, the statement is a declaration of a new variable, `Visit` is called with the parameter `statement.dcl()` that returns a `DclContext`. `Visit` is a method ANTLR has generated and it leads to `VisitDcl()` being called with the `DclContext` as parameter. Then on line 3 in Listing 5.7 a new instance of `DclNode` is created and if the `DclContext` does not contain an arithmetic expression, it is returned. Otherwise, line 8 is reached where the result of `Visit(context.aexp())` is assigned to `node.Expression`. Like mentioned before, `Visit()` is generated by ANTLR and it calls the appropriate method for converting that specific type of arithmetic expression to an AST node. For example, if the expression is an addition it will call `VisitInfixExpr()` that will create a `AdditionNode` and then call `Visit()` on both operands. These operands can be another expression, a number, an Id or something else, and `Visit()` will be called recursively until it reaches a leaf node.

Listing 5.7: `VisitDcl()` that returns a `DclNode`.

```
1  public override Node VisitDcl([NotNull] MMAGParser.DclContext context)
2  {
3      DclNode node = new() {
4          Id = context.id().GetText()
5      };
6      if (context.aexp() is null)
7          return node;
8      node.Expression = (EvaluateNode)Visit(context.aexp());
9      return node;
10 }
```

Each type of CST node has its own unique method for converting to an AST node but not all of them are included in the report, since they are similar to the ones shown here and most of the same logic applies. Finally, when every node in the CST has been visited, `VisitProgram()` will return the root of the AST and all of its descendants.

5.2 Contextual Analysis

After the syntactic analysis phase, the next phase in the compiler is the contextual analysis, which is responsible for performing a scope and type check to make sure that the program is valid, and

for evaluating arithmetic expressions. The contextual analysis phase uses the semantics from the programming language specification described in section 4.3 and checks that the program follows these rules. The outcome of this phase is a dictionary-style data structure called a symbol table, in which the variables, meaning the id's and their associated values or attributes, will be saved, making it easy for the following phases to access these values [53, 19]. Using the output from the previous phases, namely the AST, this will be traversed with the scope rules and type rules in mind. If the program follows the rules, the variables and their associated values will be inserted in the symbol table.

The symbol table for MMAG has been implemented as a class inheriting from the dictionary data structure, where the key is a string and the value is dynamically typed. Different operations are needed in order to manage the symbol table, and these operations includes adding id's to the symbol table, as well as adding values to specific id's already in the symbol table. The values in the symbol table can be strings or numbers, of either type double or integer, meaning that if there is an expression in the program, this will be evaluated before being inserted to the symbol table as either a double or an integer. For some of the different types of nodes in the AST, there is a method for adding a value or an id. The nodes that implement a `AddToSymbolTable()` method are the node for the type of gear, the one for the printer model, the one for statements, the one for declarations and the one for assignments. For the nodes that can hold an expression, there is an evaluate method that ensures that the expression is evaluated correctly as an integer or double value. This evaluation follows the transition rules described in Figure 4.3. Whether the nodes contain a `AddToSymbolTable()` method, or if they can hold an arithmetic expression, depends on their type. On the class diagram in Figure 5.5, it can be seen, what properties and methods the different nodes hold.

Figure 5.5 shows that an empty, abstract base class called `Node` is used to ensure polymorphism, as all node classes directly or indirectly inherit from this class. Three different classes are derived directly from the `Node` class, as they do not have properties in common with other classes. These are the `ProgramNode`, `GearTypeSelectionNode`, and `PrinterSettingsNode`. Apart from these, two abstract classes are derived from the base class, which are the `StmtNode` class and the `EvaluateNode`. From the `StmtNode`, two types of nodes are derived, which correspond to the two derivations of a statement in the grammar. All nodes used for arithmetic expressions inherit from the abstract `EvaluateNode` class, which declares a `Evaluate()` method. The derived, abstract class `ExpressionNode` is the base class for the binary expressions which have both a right hand side and a left hand side in the expression, as reflected in the properties of the `ExpressionNode`.

When creating an instance of the symbol table class, a method for initialising the symbol table is called to add different keywords as id's, as these have to be specified in a valid MMAG program. The keywords not specific to a certain gear type are `PrinterModel`, and `Type` which are added without values, and `InfillPercentage` which is added with a default value of 0.2. For a spur gear the gear attributes that are added without a value are: `NumberOfTeeth`, `Height`, `OuterDiameter`, and `InternalDiameter`, as mentioned in section 2.2. The symbol table is then created as seen in

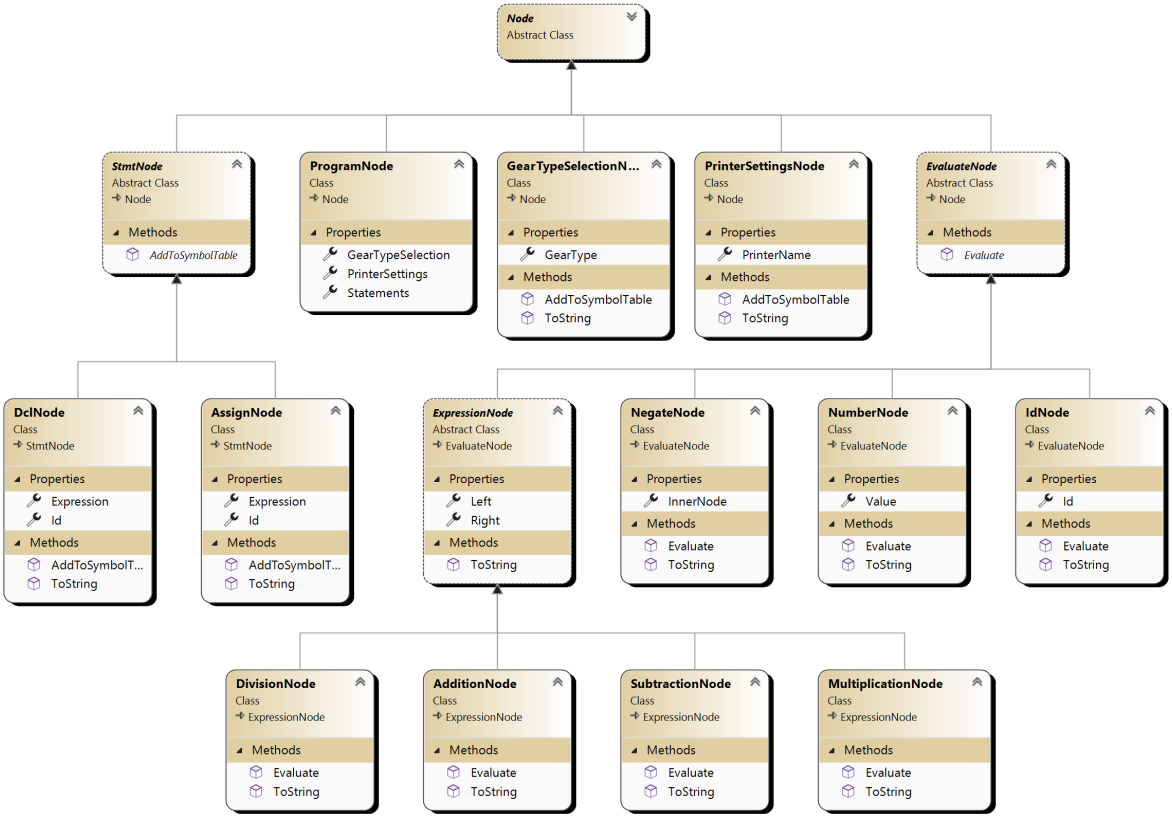

Figure 5.5: Class diagram of the class `Node` and all derived classes.

Listing 5.8 based on the AST, by using the different `AddToSymbolTable()` methods associated with the different types of nodes within the AST. From Listing 5.8 it can be seen that a new instance of the symbol table class is first created on line 3, and then on line 5-6 the values from the AST associated with the `PrinterSettings` and `GearTypeSelection` nodes are added to the symbol table. These nodes hold information about the type of printer that will be used, as well as the type of gear that will be printed, and their values are therefore of type string and not double or integer. As described in subsection 4.2.2, a MMAG program can contain zero to many statements, and on line 11 these are added to the symbol table. However if a statement can not be added to the symbol table, meaning that an exception has been thrown within an `AddToSymbolTable()` method, the compiler will terminate with an error message (see line 13-17). The scope and type checking is done within the `AddToSymbolTable()` methods for the different nodes and will be further described.

Listing 5.8: The `BuilSymbolTable()` method for creating a symbol table.

```
1  public SymbolTable BuildSymbolTable()
2  {
3      SymbolTable symbolTable = new();
4
5      Root.PrinterSettings.AddToSymbolTable(symbolTable);
6      Root.GearTypeSelection.AddToSymbolTable(symbolTable);
7      foreach (var stmt in Root.Statements)
8      {
9          try
10         {
11             stmt.AddToSymbolTable(symbolTable);
12         }
13         catch (Exception e)
14         {
15             Console.WriteLine(e.Message);
16             Environment.Exit(2);
17         }
18     }
19     return symbolTable;
20 }
```

5.2.1 Scope Checking

The scope block structure for a language helps determine the amount of symbol tables needed to keep track of the variables in a program. As described in the language specification for MMAG in subsection 4.3.3, the scope block structure for MMAG follows a monolithic block structure. This means that only one symbol table is necessary, as there is only one scope in a program and the variables will therefore all be in the same symbol table. The scope checking is done when adding new values to the symbol table, meaning values from a declaration node. Before inserting a new id, a check has to be made to ensure that the id is not already present in the symbol table - if this is the

case, the variable has already been declared which violates the scope rules mentioned in section 4.3, and an exception will therefore be thrown. Similarly when assigning values to id's, a check has to be made to ensure that the variable has been declared before a value can be assigned to it. The same applies when a variable is used in an arithmetic expression – it must then have been initialised before use.

5.2.2 Type Checking

The type checking is, as the scope checking, done when adding values to the symbol table. As mentioned, the only types that can be inserted to the symbol table are doubles or integers, apart from the two strings that are assigned to the `PrinterModel` and `Type`. If a value has the type int, a value of either type int or double can be assigned to it, however if the value is assigned to a double, it is not possible to convert it back to an integer. For this to be complied with, a type check must be done. This type check is done in a simple if-else case (see Listing 5.9) within the `AddToSymbolTable()` method for an assignment node. On line 1 the if-case checks if the id is in the symbol table and checks if its value is a double. If this is evaluated to true, the value for the specific id is of type double, and the new value that is to be added, has to be converted to a double to comply with the type rules for MMAG. This happens on line 3, where the conversion is done as a typecast. However if the if-condition is evaluated to false, the else-case on line 5 will be entered, and the value is simply inserted as the type it has been evaluated to.

Listing 5.9: The type checking part of the contextual analysis.

```
1 if (symbolTable[Id] is not null && symbolTable[Id].GetType().Equals(typeof(double)))
2 {
3     symbolTable[Id] = (double)evaluatedExpression;
4 }
5 else
6 {
7     symbolTable[Id] = evaluatedExpression;
8 }
```

If something in the program does not comply with the scope and type rules, an exception is thrown, and the compiler will terminate with an error message, as shown in section 5.1. However if everything in the program is valid, the symbol table will be returned, and the next phase, namely the code generation phase, will be able to retrieve the values from it.

5.3 Code Generation

The code generation phase is the phase where the contents of a valid source program are translated into a target language. A source program written in MMAG will in the code generation phase be translated into G-code commands. The code generation in MMAG is divided into two sub-phases, namely a phase for calculating coordinate points for the gear specified in the MMAG program, and a phase for generating the G-code commands. The code generation phase will use the symbol table as its starting point and first calculate the coordinate points needed before translating these into G-code commands that can be used to print the gear.

5.3.1 Gear Point Generation

The gear point generation phase is responsible for generating the coordinates of the points that constitute the outline of a gear. The points are calculated based on data from the symbol table. The data that is required for the generator, is the number of teeth on the gear along with the internal and outer diameter.

The points that must be calculated to draw a gear can be seen in Figure 5.6, and are for each tooth:

- Two points at the dedendum circle, at the root of the tooth.
- Two points at the pitch circle, where the tooth is angled (pitch points).
- Two points at the addendum circle, at the tip of the tooth.

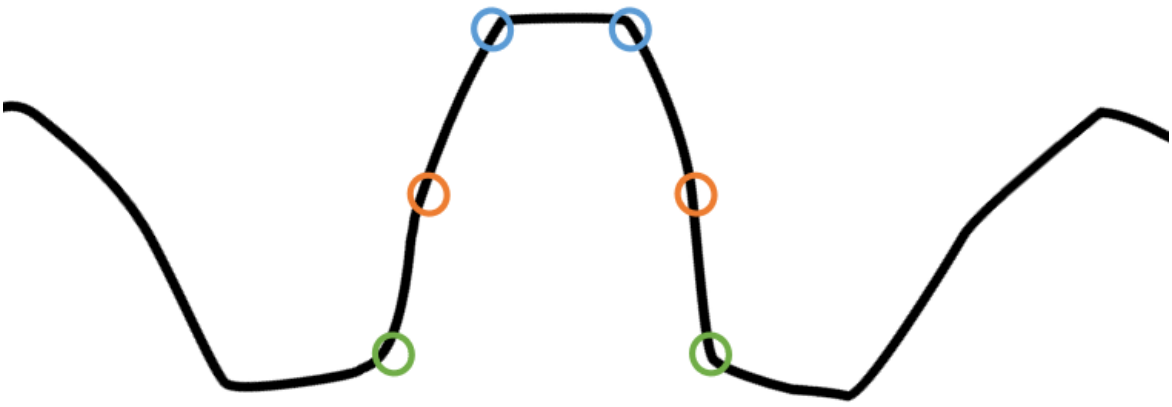

Figure 5.6: Visualisation of the different points calculated for each tooth in a gear. The two blue points are the points at the addendum circle. The two orange points are the points at the pitch circle, and the two green points are at the dedendum circle.

The sine and cosine functions have been used to calculate the points with the radius of either the dedendum circle (half the *internal diameter*), the pitch circle (half the *pitch diameter*), and the addendum circle (half the *outer diameter*). The diameter of the dedendum and addendum circles are found in the symbol table, and the diameter of the pitch circle has been calculated using the following equation [33]:

$$PD = \frac{OD \times N}{N+2} \tag{5.1}$$

In the equation, PD is the pitch diameter, OD is the outer diameter, and N is the number of teeth, which is also taken from the symbol table.

The sine and cosine functions take an angle as an input, thus the complex part about calculating the points, is to get the angle to each of them. For the points at the dedendum circle and pitch circle, the angle to the next points can be calculated by adjusting the angle of the current point with the angle, a tooth takes up. As all teeth and the space in between them are equal in size, this is simply calculated by dividing the degrees of the circle into $2N$ equally sized parts, where N is equal to the number of teeth. This is performed in the constructor on line 6 in Listing 5.10.

Calculating the angle to the points on the addendum circle is more complex, as the angle in the pitch point (see *PP* in Figure 5.7) must be taken into consideration.

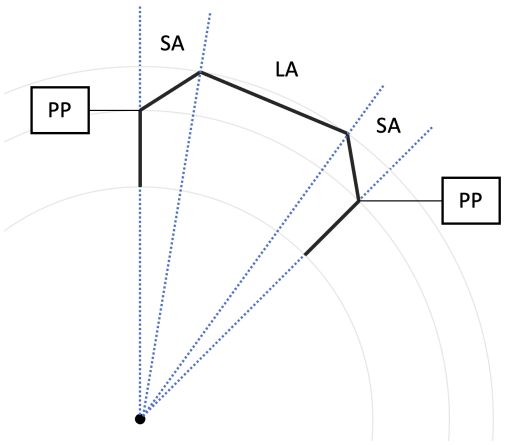

Figure 5.7: Illustration of short angle (SA) and long angle (LA) on a gear tooth. To express the point clearly, the tooth is overdimensioned.

The angle in the pitch points is 20°, and this is also called the pressure angle [22]. To get the angles for the points on the tip of the tooth, the difference in the angles of the root point and the tip point must be adjusted for. This is illustrated in Figure 5.7, where it can be seen that the angle of the root point and pitch point, which is easily calculated, must be adjusted with *SA* to get the angle for the first tip point, and then further adjusted with *LA* to get the angle for the second tip point. *SA* is calculated

by first determining the intersection between the line that goes from the pitch point through the tip point, and the addendum circle. Then the angle between the vectors from origo to the tip point and from origo the pitch point is calculated, which corresponds to the angle of SA. LA is then calculated by isolating it in Eq. (5.2), which must intuitively hold true, as the $2SA + LA$ must be within the angle of one tooth.

$$DegreesPerTooth = SA + LA + SA = 2SA + LA \tag{5.2}$$

The constructor for the `GearPointGenerator` class can be seen in Listing 5.10. Here, the data from the symbol table is first retrieved and saved to different properties in the class, see line 3 to 5. On line 6, the number of degrees each tooth takes up around the gear is calculated and saved in the property called `DegreesPerTooth`. This is simply calculated by dividing the degrees of the circle into n equally sized parts, where n is equal to the number of teeth times two, as there also has to be space in between the teeth.

Listing 5.10: The constructor for the `GearPointGenerator` class.

```
1  public GearPointGenerator(SymbolTable symbolTable)
2  {
3      NumberOfTeeth = InitializeFromSymbolTable<int>(symbolTable, "NumberOfTeeth");
4      InternalDiameter = InitializeFromSymbolTable<double>(symbolTable, "InternalDiameter");
5      OuterDiameter = InitializeFromSymbolTable<double>(symbolTable, "OuterDiameter");
6      DegreesPerTooth = (double)_degreesInCircle / (NumberOfTeeth * 2);
7
8      if (InternalDiameter >= CalculatePitchCircleDiameter())
9      {
10         throw new InvalidInternalDiameterException($"Internal diameter =
               '{InternalDiameter}' is too big!");
11     }
12
13     if (NumberOfTeeth is not (>= 10 and <= 150))
14     {
15         throw new InvalidNumberOfTeethException(
16             $"Number of teeth = '{symbolTable["NumberOfTeeth"]}' is not valid!");
17     }
18
19     if (symbolTable["InfillPercentage"] > 1 || symbolTable["InfillPercentage"] < 0)
20     {
21         throw new InvalidInfillPercentageException("Error: Infill percentage must be
               between 0 and 1");
22     }
23 }
```

From line 8 to 11 it is checked whether the internal diameter is greater than the pitch circle diameter, and an exception is thrown if so. Then from line 13 to 17 it is checked that the number of teeth on the gear is within a accepted range, and an exception is thrown if the number is outside the range.

Lastly, from line 19 to 22 a check is made to ensure that the infill percentage is within the accepted range, and again an exception is thrown, if it is outside of this.

The method that retrieves the data from the symbol table and performs input validation, is the `InitializeFromSymbolTable` method shown in Listing 5.11. This method has been made as a generic method, where the return type is of type `T`, to improve its usability. It first checks if the keyword given matches anything in the symbol table (line 3), and if there are no matching results, an exception is thrown to let the user know what problem occurred (line 5). An if condition on line 8 then checks if the value has the type `T`, which is specified when the method is called, and the if body returns the value on line 11, if true. If the required type is a double and the type of the value stored in the symbol table is an integer, the if statement on line 14-18 will be entered, and the value from the symbol table will then be converted and returned. On line 20 to 23 a special error is thrown, if no legal value is found for the `NumberOfTeeth` keyword, and on line 25 an exception is thrown if none of the above mentioned if statements are entered.

Listing 5.11: `InitializeFromSymbolTable()` checks whether the value assigned to a keyword is a legal value.

```
1  private static T InitializeFromSymbolTable<T>(SymbolTable symbolTable, string keyword)
2  {
3      if (symbolTable[keyword] is null)
4      {
5          throw new ValueNotInSymbolTableException("No value for " + keyword + " was found.
               Please assign a value to this keyword.");
6      }
7
8      if (symbolTable[keyword].GetType().Equals(typeof(T)))
9      {
10         // If the value is of the same type as the property that it will be written to:
11         return symbolTable[keyword];
12     }
13
14     if (typeof(T).Equals(typeof(double)) &&
           symbolTable[keyword].GetType().Equals(typeof(int)))
15     {
16         // If the value is an int that is to be written to a property of the type; double:
17         return (T)symbolTable[keyword];
18     }
19
20     if (keyword.Equals("NumberOfTeeth"))
21     {
22         throw new ValueNotInSymbolTableException("No legal value of " + keyword + " was
               found. Please assign an integer value.");
23     }
24
25     throw new ValueNotInSymbolTableException("No legal value of " + keyword + " was found.
           Please assign a number value.");
26 }
```

The `Start` method in Listing 5.12 starts the gear point generator and begins to calculate the points

for the gear. All calculations are called from this method, which generates the final set of gear points. Once all the points are generated, they will be sorted in the correct order for the next phase to use them.

Listing 5.12: The `Start()` method in `GearPointGenerator`.

```
1  private void Start()
2  {
3      var pitchCircleDiameter = CalculatePitchCircleDiameter();
4
5      var outerCircumference = CalculateOuterCircumference();
6      var outerShortArcAngle = CalculateOuterShortArcAngle(OuterDiameter / 2);
7      var outerShortArcLength = CalculateOuterShortArcLength(OuterDiameter / 2,
           outerShortArchAngle);
8      var outerLongArcAngle = CalculateOuterLongArcAngle(outerCircumference,
           outerShortArcLength);
9
10     var internalPointList = CalculateInnerCirclePoints(InternalDiameter / 2);
11     var pitchPointList = CalculateInnerCirclePoints(pitchCircleDiameter / 2);
12
13     var outerPointList = CalculateOuterCirclePoints(OuterDiameter / 2,
           outerShortArchAngle, outerLongArcAngle);
14     var printOrder = PrintOrderGenerator(internalPointList, pitchPointList,
           outerPointList, NumberOfTeeth);
15
16     foreach (var item in printOrder)
17     {
18         item.X = Math.Round(item.X, 5);
19         item.Y = Math.Round(item.Y, 5);
20     }
21
22     ConvertToJson(printOrder);
23 }
```

From line 3 to 8, the pitch circle diameter, outer circle circumference, the angle of the short and long arcs illustrated on Figure 5.7 are calculated, as these are needed for calculating the points for the gear. A list of all the points on the inner circle is then calculated on line 10 and saved to a list called `internalPointList`. On line 11, the pitch points are calculated, and then on line 13 the tip points are calculated, these being the points placed on the outer circle. Then the different lists of points are merged together in the order, in which they will be drawn by the printer, and saved to the `printOrder` list. From line 16 to line 20 the coordinate points are rounded to ensure consistency, and `printOrder` is then saved to a .json file.

The calculations for the gear points on the inner and outer circle are pretty similar, only that the outer points contain a slight addition to their functions, why the method for calculating the outer points are given as an example, see Listing 5.13. With the auxiliary data already calculated, the purpose of the `CalculateOuterCirclePoints()` method is to calculate the coordinate points for each point on the outer circle and add them to a list. For the points to be calculated at the right positions, a “ghost

pointer" named `currentDegree` is pushed around the circle at specific offsets to ensure the correct positions.

Listing 5.13: `CalculateOuterCirclePoints()`. Calculates and returns the list of points on the perimeter of the outer circle.

```
1  private List<Point> CalculateOuterCirclePoints(double radius, double shortArc, double
       longArc)
2  {
3      List<Point> outerPoints = new();
4
5      double currentDegree = 90;
6
7      for (var i = 0; i < NumberOfTeeth; i++)
8      {
9          currentDegree = CalcFirstOffset(currentDegree, shortArc);
10
11         var x1 = CalcX(currentDegree, radius);
12         var y1 = CalcY(currentDegree, radius);
13         var p1 = new Point(x1, y1);
14         outerPoints.Add(p1);
15
16         currentDegree = CalcSecondOffset(currentDegree, longArc);
17
18         var x2 = CalcX(currentDegree, radius);
19         var y2 = CalcY(currentDegree, radius);
20         var p2 = new Point(x2, y2);
21         outerPoints.Add(p2);
22
23         currentDegree -= shortArc + DegreesPerTooth;
24     }
25
26     return outerPoints;
27 }
```

In Listing 5.13, on line 3, a list of points is created to store every point of the outer circle. In a for loop from line 7 to 24, the points will be calculated and saved to a list. Each tooth has two points on the outer circle that define the width of the tip of the tooth. So to calculate the coordinates for each tooth, this method moves a pointer around the outer circle perimeter. The first move on line 9 pushes the pointer a distance equal to the short arc. That point is then added to the list before moving to the next point. The second movement on line 16 pushes the pointer a distance equal to the long arc and the pointer's current coordinates are added to the list. On line 23, the ghost pointer is moved to the next tooth's starting point. This process repeats, moving the pointer 360 degrees around the circle until all points have been calculated, and the completed list of points is returned.

Once all points are generated, the last method `PrintOrderGenerator` is called, which repeats a loop that merges the points from the three lists into one, where the points appear in the correct order. The method can be seen in Listing 5.14. The number of loop iterations is the number of teeth the

gear has. The method contains a counter that is used to access the correct point from each list. As the first point of each list only creates a half tooth, the counter has to be increased midway to acquire the next points to finish the tooth.

Listing 5.14: `PrintOrderGenerator()` creates a list of points sorted to create an optimized toolpath.

```
1  private List<Point> PrintOrderGenerator(List<Point> rootPoints, List<Point> pitchPoints,
2              List<Point> outerPoints)
3  {
4      List<Point> printOrder = new();
5
6      var counter = 0;
7
8      for (var i = 0; i < NumberOfTeeth; i++)
9      {
10         printOrder.Add(rootPoints[counter]);
11         printOrder.Add(pitchPoints[counter]);
12         printOrder.Add(outerPoints[counter]);
13
14         counter++;
15
16         printOrder.Add(outerPoints[counter]);
17         printOrder.Add(pitchPoints[counter]);
18         printOrder.Add(rootPoints[counter]);
19
20         counter++;
21     }
22
23     return printOrder;
24 }
```

First on line 4 a list of points is created, and on line 6 the integer called `counter` is initialised to 0. This `counter` will be used for accessing the correct point from each list. From line 8 to 21 a for loop is seen, in which the points added to the `printOrder` list, and on line 23 the list of points is returned.

5.3.2 G-code Generation

The G-code generation phase has to have some similarities to the usual slicers, see section 2.6, as different elements have to be added for the model to be able to be printed correctly. For MMAG this include walls, infill and a top and bottom layer.

Generation of the G-code takes place after the gear design's proportions have been calculated. The method for generating G-code can be seen in Listing 5.15. At first, printer settings are retrieved from the symbol table (line 5), this is done inside a try/catch so any exception that might occur during the G-code generation will be caught. On line 6, the private method `Start()` is called, which will

generate the G-code based on the symbol table inputted as a parameter.

Listing 5.15: `GenerateGCode()`. Returns a G-code file represented as a string.

```
1  public string GenerateGCode(SymbolTable symbolTable)
2  {
3      try
4      {
5          _printerSettings = new(symbolTable["PrinterModel"]);
6          var result = Start(symbolTable);
7          return result;
8      }
9      catch (Exception e)
10     {
11         Console.WriteLine(e.Message);
12         Environment.Exit(1);
13         return null;
14     }
15 }
```

Looking into the private method `Start` in Listing 5.16, an object of the `StringBuilder` class is first instantiated (line 3). The number of layers needed for the gear is then calculated based on the desired height, see line 6. Then on line 7, it is calculated how many of the total layers should be used for the top and bottom of the gear. The height of each layer is retrieved from the `printerSettings` object in line 10, and on line 11 the gear points from the previous phase are retrieved. From line 14 to line 17 the different generators are instantiated. These being the `bottomTopGenerator`, `wallGenerator` and `infillGenerator`, and at last the `codeStringsGenerator`. Since the difference in how the infill and how the surfaces are generated is so small, the same generator can be utilized for the two components. The only difference is that for the `infillGenerator`, the infill percentage from the symbol table is sent with as the `infillPercentage` parameter, which has a default value of 1 in the constructor, if no actual parameter is passed to it. Hence no value is passed for the `bottomTopGenerator`, which is responsible for generating the surface GCode.

Listing 5.16: `GCodeGenerator.Start()`.

```
1  private string Start(SymbolTable symbolTable)
2  {
3      var stringBuilder = new StringBuilder();
4
5      // variables
6      int totalLayerCount = CalculateTotalNumberOfLayers(symbolTable["Height"]);
7      var bottomTopLayerCount = CalculateNumberOfLayersBottomTop(totalLayerCount);
8      var infillLayerCount = totalLayerCount - 2 * bottomTopLayerCount;
9      var currentLayer = 0;
10     var layerHeight = _printerSettings.LayerHeight;
11     var gearPoints = GetGearPointsFromJsonFile();
12
```

```
13      // generators
14      var bottomTopGenerator = new SurfaceGenerator(symbolTable["OuterDiameter"],
            _printerSettings, gearPoints);
15      var wallGenerator = new WallGenerator(_printerSettings, gearPoints);
16      var infillGenerator = new SurfaceGenerator(symbolTable["OuterDiameter"],
            _printerSettings, gearPoints, symbolTable["InfillPercentage"]);
17      var codeStringGenerator = new GCodeStringsGenerator(_printerSettings);
```

Next, the G-code generation starts. It takes place in three different for loops, one for the bottom layer, one for the walls and infill, and one for the top surface. The construction of these loops are very similar, see Listing 5.17 for an example, but act as different phases of the print. Then a call to the method `DrawLayer()` is made in each iteration of the loops. This private method is responsible for generating a layer of the design in G-code.

Listing 5.17: For loop construction that is similar for the generation of both the top and bottom layer, and for the wall and infill.

```
1   for (var i = 0; i < bottomTopLayerCount; i++)
2       {
3           DrawLayer(i, CalculateZCoordinate(currentLayer, layerHeight), wallGenerator,
                bottomTopGenerator, stringBuilder);
4           currentLayer++;
5       }
```

The method `DrawLayer()`, seen in Listing 5.18, starts of by generating the wall, which is the vertical surface on the perimeter of the gear. This is done on line 4, by calling the method `wallGenerator.Start()`, which takes a z-coordinate as parameter. The rest of the layer consist of a surface which lines are printed in alternating directions depending on the current loop iteration (from line 7 to 15). This means that the printed lines will alternate between being parallel to the x-axis and the y-axis.

Listing 5.18: `DrawLayer()` method in the `GCodeGenerator`.

```
1   private static void DrawLayer(int i, double z, WallGenerator wallGenerator,
        SurfaceGenerator surfaceGenerator, StringBuilder stringBuilder)
2   {
3       // draw outer edge/wall
4       stringBuilder.AppendLine(wallGenerator.Start(z));
5
6       // fill in surface (alternating directions)
7       if (i % 2 == 0) // parallel with x-axis
8       {
9           stringBuilder.AppendLine(surfaceGenerator.GenerateSingleLayer(true, z));
10      }
11      else // parallel with y-axis
12      {
13          stringBuilder.AppendLine(surfaceGenerator.GenerateSingleLayer(false, z));
14      }
```

```
15 }
```

After the three loops, the G-code for the design is generated and the last thing needed to finalize the G-code is the start code and end code for the printer. These can be retrieved by calling `codeStringGenerator.GetPrefix()` and `codeStringGenerator.GetSuffix()` as shown in Listing 5.19, where the start code is inserted at the beginning of the string and the end code is appended at the end.

Listing 5.19: Code block of the code for retrieving the start and end G-code.

```
1 // Start code
2 stringBuilder.Insert(0, codeStringGenerator.GetPrefix());
3
4 // End Code
5 stringBuilder.AppendLine(codeStringGenerator.GetSuffix());
6
7 return stringBuilder.ToString();
```

Wall Generator

The `WallGenerator` class is responsible for generating the G-code string for the outer wall of the 3D print. It does so from a list of coordinates, which is obtained as a parameter on its constructor. The method for generating the wall related G-code commands is called `GenerateGCodeWall()` and can be seen in Listing 5.20.

Listing 5.20: `GenerateGCodeWall()` method for generating the G-code commands for the wall of the gear.

```
1  private string GenerateGCodeWall(double z)
2  {
3      var wallPrintOrder = "";
4      var moveToStart = _gStringGenerator.LinearMove(_cordList[0], z);
5      wallPrintOrder += moveToStart;
6
7      int j;
8      for (j = 0; j < _cordList.Count - 1; j++)
9      {
10         if (j == 0)
11         {
12             var printMove = _gStringGenerator.LinearMove(_cordList[j], _cordList[j + 1],
                   1800);
13             wallPrintOrder += printMove;
14         }
15         else
16         {
17             var printMove = _gStringGenerator.LinearMove(_cordList[j], _cordList[j + 1]);
```

```
18              wallPrintOrder += printMove;
19          }
20      }
21
22      var moveToEnd = _gStringGenerator.LinearMove(_cordList[j], _cordList[0]);
23      wallPrintOrder += moveToEnd;
24
25      return wallPrintOrder;
26  }
```

Here, an empty string called `wallPrintOrder` is first initialised on line 3, this will be string containing the commands for the wall. On line 4 to 5 a G-code command for a linear move to the start position, without printing, is added to the `wallPrintOrder` string. In a for loop (line 8-20), that loops from 0 and to the second last point in the list of points, more commands will be added to the string. On line 10 to 14 a G-code command for printing the first point is added. This is added separately as the feedrate of the nozzle has to be set to 1800, to make sure that it does not move too fast. After this the rest of the commands for printing the wall is added to the `wallPrintOrder` string, except for the very last. Then outside the loop on line 22 and 23 a command for printing the last point is added and on line 25 `wallPrintOrder` is returned as a string that contains all the G-code commands for printing the wall.

Surface Generator

The `SurfaceGenerator` class is the component responsible for generating the surface though while implementing the component, it was discovered that with little to no alternation the surface generator could be used to generate an infill structure much similar to Cura's grid infill seen in Figure 2.7. It will be further discussed later in this section how it also works as an infill generator.

The constructor for the `SurfaceGenerator` can be seen in Listing 5.21. One important detail to note about the constructor is the variable `infillPercentage`, which is an optional argument. The value of the parameter will default to 1, if no other value is parsed as a parameter when instantiated. This is exemplified in Listing 5.16, where the constructor is used with three arguments when generating the `topBottomGenerator`, which produces solid layers, whereas the `infillGenerator` is instantiated with four arguments for the constructor, such that the infill percentage is set based on its value in the symbol table.

Listing 5.21: `SurfaceGenerator` the constructor for the surface generator.

```
1  public SurfaceGenerator(double outerDiameter, PrinterSettings printerSettings, List<Point>
       points, double infillPercentage = 1)
2  {
3      _printerSettings = printerSettings;
```

```
4       _points = points;
5       _lineWidth = _printerSettings.NozzleSize / infillPercentage;
6       _maxRadius = outerDiameter / 2 + 1; // added a 1 mm buffer
7
8       InitializeListsOfLines();
9       InitializeGearLines();
10  }
```

The method `InitializeListsOfLines()` is called on line 8, and is responsible for initializing two lists of lines that make up a grid pattern. It does so by generating two sets of lines parallel with the x- and the y-axes, respectively. The lines are equally distributed along the axes, and cover the area of the gear, including a small buffer. These lines are used to generate the tool paths for the surface layers.

On line 9 `InitializeGearLines()` is called, and this method creates two lists of lines, based of the gear points generated in the earlier phase. The lists contain the lines defining the outer perimeter of the gear, and the difference between them is that in one of the lists, the lines are sorted by the x-value of their start coordinate, and in the other list, the lines are sorted by the y-value of their starting coordinate. These lists will facilitate the generation of the alternate print direction between layers.

The generation of one surface layer is performed in the `GenerateSingleLayer()` method, which is called from the G-code generator in the `DrawLayer()` method, as shown in Listing 5.18. The method `GenerateSingleLayer()` takes two parameters: a bool and a double. The bool is used to determine whether the printed lines of the surface should be parallel to the x-axis or the y-axis, and the double is used to identify the z-coordinate for the layer. The body of the method contains two method calls. First, the method `GetExtrusionLines()` is called, and secondly the `GenerateCommands()` is called on the output of the first method.

The first half of the `GetExtrusionLines()` method can be seen in Listing 5.22, where a foreach loop on line 5 iterates through the list `_xLines` or `_yLines`. The loop either iterates through the lines parallel to the x-axis or the lines parallel to the y-axis based on the value of the boolean `isParallelWithXaxis`. The loop on line 10-17 iterates though the gear lines and finds intersections between the grid lines and the outline of the gear. The intersection points are added to a list and used later in the same method for making the actual lines for extrusion, see Listing 5.23.

Listing 5.22: The first half of `SurfaceGenerator.GetExtrusionLines()`. The method is responsible for calculating the actual lines to be printed.

```
1   private List<Line> GetExtrusionLines(bool isParallelWithXaxis)
2   {
3       List<Line> result = new();
4
5       foreach (var line in isParallelWithXaxis ? _xLines : _yLines)
6       {
```

```
7           var intersections = new List<Point>();
8
9           // identify intersections
10          foreach (var gearLine in isParallelWithXaxis ? _gearLinesX : _gearLinesY)
11          {
12              var intersection = FindIntersection(line, gearLine, isParallelWithXaxis);
13              if (intersection is not null)
14              {
15                  intersections.Add(intersection);
16              }
17          }
```

Listing 5.23: The second half of `SurfaceGenerator.GetExtrusionLines()` that is responsible for calculating the actual lines to be printed

```
1   // check if number of intersections is even
2           if (intersections.Count != 0 && IsEvenNumber(intersections.Count))
3               // make lines between points, two and two, and add lines to result
4           {
5               foreach (var point in intersections)
6               {
7                   var index = intersections.IndexOf(point);
8
9                   if (IsEvenNumber(index))
10                  {
11                      var newline = new Line(point, intersections[index + 1]);
12                      if (IsValidLine(newline))
13                      {
14                          result.Add(newline);
15                      }
16                  }
17              }
18          }
19          else if (!IsEvenNumber(intersections.Count))
20          {
21              Console.WriteLine("Warning: There might be holes in the generated gear");
22          }
23      }
24
25      return result;
26  }
```

In the second half of the `GetExtrusionLines()` method, shown in Listing 5.23, lines are created between the intersection points of each grid line. The points are connected two and two, such that the created lines are all within the walls of the gear. These lines are what will make up the gear surface when the design is printed. The foreach loop on line 5 iterates though the intersection points and if the index of the point is even, a set of new lines are created. Alternatively, if the number of intersections is not even, there is a slight chance that holes will appear in the surface structure, so a warning is given if the number of intersections does not add up. Lastly, the result is returned to `GenerateSingleLayer()` and the G-code for the layer is generated in the second function that is

called from `GenerateSingleLayer()`, namely `GenerateCommands()`.

This concludes the description of the implementation of the MMAG compiler, and in the next chapter a validation of the compiler will be conducted.

Chapter 6

Test

To validate that the developed compiler for MMAG works as intended it is crucial to perform different tests. These are typically performed as three different types of tests, namely unit testing, integration testing, and acceptance testing. The two first types of tests will validate if the program parts works as intended both individually and together, and the last type of test is for determining whether or not the program fulfills the requirements set for the program as a whole [30]. The xUnit framework is used for testing and one can therefore see the attributes *[Fact]*, *[Theory]* and *[InlineData]* in the code. The *[Fact]* attribute is used to identify a test that takes no parameters, while *[Theory]* is used to identify a parameterized test i.e., a test that can be run multiple times with different parameters as provided in the *[InlineData]* attribute.

Tests often follow the AAA pattern, which consist of three parts: Arrange, Act, and Assert. In the arrange part, the data for the test is set, and in the act part the method being tested is called. At last comes the assert part, where the expected values are compared to the actual values obtained in the act part [40]. The tests for the compiler follow the AAA pattern.

6.1 Unit Test

The first type of tests that have been performed are unit tests, which are small tests that test the different units in a system, here being the developed compiler, with the purpose of validating that each single unit work as intended. A unit refers to the smallest component that is testable, and it is therefore not necessarily one single method, but it can involve several methods [30]. Methods that are private, meaning that they can only be accessed from within a specific class, will not be tested individually, but only through their invocations in public methods. It is common practice to test private methods only through the public methods calling them, so to keep these methods private in

the code for the developed compiler, they will not be tested directly [52]. Unit tests have been written for each part of the compiler shown on Figure 5.2 and will be described in the next section.

6.1.1 Syntactic Analysis

As mentioned in section 5.1, ANTLR is being used to generate the scanner and parser parts of the compiler. ANTLR is well-tested and reliable to use [57], and it has therefore been decided to only test the code blocks generated by ANTLR briefly. Some small unit tests have been made to check the correctness of the output from the scanner and parser. For the scanner, it has been checked that the generated token stream is correct, i.e., with correct token types and content. For the parser, a test has been conducted to check if the source program is being parsed correctly. This is done by comparing concrete syntax tree nodes with the expected string representation of it. The parser only works on the output of the scanner, thus the scanner has been used in the *Arrange* part of the unit test for the parser.

The source program seen in Listing 6.1 has been used as the input program for both the unit tests for the scanner, and the unit test for the parser.

Listing 6.1: Source program used to test scanner and parser.

```
string s =
PrinterModel = Ultimaker2Plus;
Type = SpurGear;
Var hello = 1;
NumberOfTeeth = 10;
```

Scanner

The first unit test conducted for the scanner can be seen in Listing 6.2. Here, the text contents of the tokens are evaluated.

Listing 6.2: Essential parts of the unit test of the scanner, checking if the correct text contents are stored in the tokens.

```
// Arrange
...
CommonTokenStream tokens = new(lexer);
MMAGParser parser = new(tokens);

//Act
parser.program();
```

```
8
9  string a = tokens.Get(0).Text;
10 string b = tokens.Get(1).Text;
11 string c = tokens.Get(2).Text;
12 string d = tokens.Get(3).Text;
13
14 string e = tokens.Get(4).Text;
15 string f = tokens.Get(6).Text;
16 string g = tokens.Get(8).Text;
17 string h = tokens.Get(13).Text;
18
19 //Assert
20 Assert.Equal("PrinterModel", a);
21 Assert.Equal("=", b);
22 Assert.Equal("Ultimaker2Plus", c);
23 Assert.Equal(";", d);
24
25 Assert.Equal("Type", e);
26 Assert.Equal("SpurGear", f);
27 Assert.Equal("Var", g);
28 Assert.Equal("NumberOfTeeth", h);
```

The `Get()` method used in the code example from line 9 to line 17 takes an index as its input parameter. The method is used to select the token placed at the specified index in a token stream, which in this case is called `tokens`. The `.Text` attribute is used to get the actual content from the specific token, i.e., the fraction of the source program that the specific token represents. The correct result is known, as the scanner starts reading from the top to the bottom, so it tokenizes the program in the same order as one would read it. After having chosen the token, e.g., at line 9, which is the token at index 0, the content is extracted using the `.Text` attribute and the value is saved in a string. This is done in the same way for the remaining tokens in line 10 to line 17. From line 20 to line 28, a check is done for each token to see that the text contents from the tokens generated by the scanner are equal to the words from the input program. In line 20, the `Assert.Equal()` method compares the string ‘‘`PrinterModel`’’ with the string saved in variable `a`, to make sure that they are identical. The rest of the test does the same, thereby checking if the source program is being scanned correctly by the scanner with respect to the text content of each token.

Another test of the scanner has been made, where it is tested that the type of the token is also correct, i.e., if the input is being tokenized correctly in terms of token type. This test can be seen in Listing 6.3. After running the grammar file (`MMAG.g4`) in ANTLR, a text file called `tokens` is generated, which contains an enumeration of all the token types. This file can be seen in Appendix C, and it is being used here to check whether the type is correct or not.

Listing 6.3: Important parts of the unit test of scanner, checking if the generated tokens are of the correct type.

```
1  // Arrange
2  ...
3  CommonTokenStream tokens = new(lexer);
4  MMAGParser parser = new(tokens);
5
6  //Act
7  parser.program();
8
9  int aa = tokens.Get(0).Type;
10 int bb = tokens.Get(1).Type;
11 int cc = tokens.Get(2).Type;
12 int dd = tokens.Get(3).Type;
13
14 int ee = tokens.Get(4).Type;
15 int ff = tokens.Get(6).Type;
16 int gg = tokens.Get(8).Type;
17 int hh = tokens.Get(13).Type;
18
19 //Assert
20 Assert.Equal(9, aa);
21 Assert.Equal(2, bb);
22 Assert.Equal(17, cc);
23 Assert.Equal(1, dd);
24
25 Assert.Equal(11, ee);
26 Assert.Equal(12, ff);
27 Assert.Equal(10, gg);
28 Assert.Equal(16, hh);
```

The test seen in Listing 6.3 is similar to the test seen in Listing 6.2, the only difference is that the type of the tokens are now checked instead of the contents of the tokens. Again, the `Get()` method is used to select tokens in the token stream, according to an index, and after the token is selected, the `.Type` attribute is used to check the type of the token. The `.Type` attribute returns an integer value, and this value corresponds to a certain type as seen in Appendix C. This is done for all tokens in the source program, and the types are saved in a variable, making it possible to compare it to the expected type. For instance, at line 20 a check is made to see if the actual value of the variable `aa` is equal to the integer value `9`, which corresponds to the token type called `PrinterSettingsKeyword`, and at line 21 it is checked whether or not the actual value of the variable `bb` is equal to the integer value `2` that corresponds to the token type `AssignSymbol`.

Combined with the other unit test for the scanner, both the type and text content of each token is checked to validate the output of the scanner.

Parser

The unit test conducted for the parser can be seen in Listing 6.4. Here, we match the printed parse trees of the direct child nodes of the `ProgramNode` to their expected output based on the source program.

Listing 6.4: Excerpt of the unit test of the parser.

```
1  // Arrange
2  ...
3  MMAGParser parser = Setup(s); //s : sourceprogram
4
5  // Act
6  IParseTree tree = parser.program();
7
8  string printerSettings = tree.GetChild(0).ToStringTree(parser);
9  string gearTypeSelection = tree.GetChild(1).ToStringTree(parser);
10 string stmtDcl = tree.GetChild(2).ToStringTree(parser);
11 string stmtAssignment = tree.GetChild(3).ToStringTree(parser);
12 string EndOfFile = tree.GetChild(4).ToStringTree(parser);
13
14 //Assert
15 Assert.Equal("(printerSettings PrinterModel = Ultimaker2Plus ;)", printerSettings);
16 Assert.Equal("(gearTypeSelection Type = SpurGear ;)", gearTypeSelection);
17 Assert.Equal("(stmt (dcl Var (id hello) = (aexp 1)) ;)", stmtDcl);
18 Assert.Equal("(stmt (assignment (id NumberOfTeeth) = (aexp 10)) ;)", stmtAssignment);
19 Assert.Equal("<EOF>", EndOfFile);
```

On line 3 in Listing 6.4 the method `Setup()` is being used, in which the scanning part is done, and the parser phase is set up. The `Setup()` method takes a string as input parameter, and again the source program in Listing 6.1 is used. A parser is returned, which will be assigned to the `parser` variable of type `MMAGParser`. On line 6, the `program()` method of the `parser` is called. This method creates a concrete syntax tree with the content from the source program, and this will be assigned to the variable called `tree` which is of type `IParseTree`. The `GetChild()` method and `ToStringTree()` method can then be used, as shown from line 8 to line 12. First, the `GetChild()` method is used to select a child node and its belonging subtree from the tree, e.g., at line 8, the child node that is placed at index 0 is being selected. Then the `ToStringTree()` method is called on the subtree, to display the contents of it. The `ToStringTree()` method takes a parser as a parameter and extracts the data from the subtree. Then from line 15, a check is made to test if the data that is being extracted from the subtrees, matches the expectations. In this case it was the production rule for `printerSettings`, and the output from `ToStringTree()` is formatted such that every non-terminal is placed in a new parenthesis. In this case, we only follow one production rule and that is the one for `printerSettings`. In other cases where there are more than one non-terminal in a subtree, a new level of parentheses are added for each non-terminal, e.g., the one at index 2, where it is a variable declaration and initialization. Following the production rule defined in Listing 5.1, this child node

is defined as a `stmt`, whose derivation contains the non-terminals `dcl`, `id`, and `aexp`, therefore the expected output should match the one given at line 17.

6.1.2 Contextual Analysis

To ensure that the contextual analysis is performed correctly, 11 different tests of the implementations of the transition, type, and scope rules within the AST nodes have been conducted. A unit test for the type check part, and a unit test for the scope check part of the contextual analysis have been selected for further description to provide examples of how these tests are implemented.

The selected example of a unit test for ensuring that the scope checking is done correctly can be seen in Listing 6.5. The test checks that the correct exception is thrown if the user tries to declare the same variable twice. First in the arrange part, on line 6 the variable `testvar` is added to the symbol table. On line 7-11 a declaration node is created with the same id, namely `testvar`. Then in the act and assert part on line 13 `Assert.Throws()` checks if the `AlreadyInSymbolTableException` is thrown when trying to add the `dclNode` to the symbol table.

Listing 6.5: Unit test for checking that the `AlreadyInSymbolTableException` is thrown when a variable has not been declared.

```
1  [Fact]
2  public void DclNodeIdAlreadyDeclared()
3  {
4      //Arrange
5      SymbolTable symbolTable = new();
6      symbolTable.Add("testvar", 5);
7      DclNode dclNode = new()
8      {
9          Id = "testvar",
10         Expression = null
11     };
12     //Act/Assert
13     Assert.Throws<AlreadyInSymbolTableException>(() =>
           dclNode.AddToSymbolTable(symbolTable));
14 }
```

The example for the type checking test can be seen in Listing 6.6. This unit test has been conducted as a *Theory*, which makes it possible to have different test cases. Four test cases have been created to ensure that a variable with a value of type double can not be converted to an integer. This is done by first declaring a variable of a certain type, and then assigning a new value of either the same type or a different type, and then checking if the result is of the expected type. In the test from line 10 to line 15, a variable is first declared with a certain value and inserted to the symbol table. Then from line 17 to line 22 a new value is assigned to the variable. On line 24, `Assert.Equal()` will check if

the type of the variable is the same as the type expected. In the first test case on line 2, the variable is first of double and then an integer is assigned to it, and the type should therefore still be a double in the end. The second and third test case on line 3-4 checks that the type does not change if values of the same type is assigned to the variable. The fourth test case on line 5 first declares the variable as an integer and then a double is assigned to it, meaning that the variable is expected to be a double in the end.

Listing 6.6: Unit test to check if the type of a variable corresponds with the type rules.

```
1  [Theory]
2  [InlineData("1.5", "1", "Double")]
3  [InlineData("1", "1", "Int32")]
4  [InlineData("1.5", "1.5", "Double")]
5  [InlineData("1", "1.5", "Double")]
6  public void CorrectAssignTypeInSymbolTable(string first, string second, string
       expectedType)
7  {
8      // Arrange
9      SymbolTable symbolTable = new();
10     DclNode dclNode = new()
11     {
12         Id = "testvar",
13         Expression = new NumberNode { Value = first}
14     };
15     dclNode.AddToSymbolTable(symbolTable);
16     // Act
17     AssignNode assignNode = new()
18     {
19         Id = "testvar",
20         Expression = new NumberNode { Value = second }
21     };
22     assignNode.AddToSymbolTable(symbolTable);
23     // Assert
24     Assert.Equal(expectedType, symbolTable["testvar"].GetType().Name);
25 }
```

Overall, 11 unit tests for the contextual analysis were conducted, and the result was that the contextual analysis, including the type and scope checking, performs as intended.

6.1.3 Gear Point Generation

The testable units in the gear point generation class are the constructor and the method for generating a .json file containing the calculated gear coordinate points. The constructor retrieves the values of the gear attributes from the symbol table and performs input validation of them, i.e., checks if these values can be used to generate coordinate points for a gear. There are three criteria that the values have to fulfill, and if they are not fulfilled, an exception will be thrown. Three

unit tests have therefore been conducted to account for all three criteria. An example of one of these can be seen in Listing 6.7. Here, it has been tested that the constructor throws an exception if the entered internal diameter is greater than what the pitch circle diameter of the gear is calculated to be. This unit test has been conducted as a *Theory* with three different test cases. The arrange part of the test can be seen in Listing 6.7 from line 8 to line 11. Here, a symbol table is initialised with the *InlineData*, and on line 14 the act and assert parts are conducted. The act and assert parts happen at once, and the `Assert.Throws()` method checks that the exception called `InvalidInternalDiameterException` is thrown when the constructor for the `GearPointGenerator` class is called with the values from the different *InlineData* as input.

Listing 6.7: Unit test of the input validation in `GearPointGenerator` constructor with an invalid value of the `InternalDiatmeter` attribute in the symbol table.

```
1  [Theory]
2  [InlineData(20.5, 440, 12)]
3  [InlineData(30, 30, 40)]
4  [InlineData(40.8, 40, 70)]
5  public void ConstructorInvalidInternalDiameterTest(double outerDiameter, double
       internalDiameter, int numberOfTeeth)
6  {
7      // Arrange
8      SymbolTable symbolTable = new();
9      symbolTable["NumberOfTeeth"] = numberOfTeeth;
10     symbolTable["OuterDiameter"] = outerDiameter;
11     symbolTable["InternalDiameter"] = internalDiameter;
12
13     // Act/Assert
14     Assert.Throws<InvalidInternalDiameterException>(() => new
           GearPointGenerator(symbolTable));
15 }
```

The purpose for the gear point generation class is to calculate the coordinate points for a gear, and it is therefore crucial that this part works as it is supposed to. A unit test has been conducted for the public method called `GenerateJsonPointList()` in which all the private methods needed for the calculations are called. The unit test for this can be seen in Listing 6.8. Three test cases have been created, with different values for the symbol table, as well as different files with the expected result. First from line 8 to line 11, a symbol table is created with the input parameters. Then on line 14, the `GenerateJsonPointList()` method is called. This method creates a .json file with x and y coordinates of each corner of the gear, and this file is then compared to the expected file in the `Assert.True()` method on line 17. This comparison is done by using a helper method called `AreFileContentsEqual()` which goes through the contents of both files while comparing them.

Listing 6.8: Unit test for testing that the gear points are correctly calculated.

```
1  [Theory]
2  [InlineData(20.5, 10, 12,
       @"..\..\..\GearPointGeneratorTests\testCasesGearPointgen\testCase1.json")]
3  [InlineData(30, 20, 40,
       @"..\..\..\GearPointGeneratorTests\testCasesGearPointgen\testCase2.json")]
4  [InlineData(40.8, 30, 70,
       @"..\..\..\GearPointGeneratorTests\testCasesGearPointgen\testCase3.json")]
5  public void GenerateJsonPointListTest(double outerDiameter, double internalDiameter, int
       numberOfTeeth, string expectedFilepath)
6  {
7      // Arrange
8      SymbolTable symboltable = new();
9      symboltable["NumberOfTeeth"] = numberOfTeeth;
10     symboltable["OuterDiameter"] = outerDiameter;
11     symboltable["InternalDiameter"] = internalDiameter;
12     GearPointGenerator actualPointsGenerator = new GearPointGenerator(symboltable);
13     // Act
14     actualPointsGenerator.GenerateJsonPointList();
15     var actualPointsFilePath = "PrintOrder.json";
16     // Assert
17     Assert.True(AreFileContentsEqual(new FileInfo(actualPointsFilePath), new
           FileInfo(expectedFilepath)));
18 }
```

In conclusion, three tests for the different types of exceptions in the constructor for the gear point generation class were conducted, and they all worked as intended. Moreover, a test for generating the gear coordinate points was conducted, and this also worked as intended.

6.1.4 G-code Generation

To ensure that the generation of the G-code would be correct, some test data had been gathered that could be compared to, and a set of tests that would use this data was created. For the first set of tests that runs through the methods that generate the movement, some specific test data was given to test if it would catch some out-of-bounds exceptions, but also if all data is correct, it would output the correct string of G-code. Multiple methods that generate movement are used in the project, and since they look much alike, one of them will be explained. This method is `LinearMove4Test()` which can be seen in Listing 6.9. First on line 1-6 the different test cases are set. Then in the arrange part on line 10 and 11 new points are created with the test data. On line 12 and 13 an instance of the `PrinterSettings` and `GCodeStringsGenerator` class are created, in which the last mentioned handles the generation of the G-code. To make sure that the extrusion is calculated correctly the property `E` on the `GCodeStringsGenerator` class is set to 0. In the act and assert part from line 17 to 20 a check is made to see if the expected string is empty, and to ensure that an exception is thrown. This check is made using the `Assert.Throws()` method. From line 21 to 25 a string of G-code is generated using the created instance of the `GCodeStringsGenerator` class, and this is saved to the

string called `actual`. The `Assert.Equal()` method then compares the expected data from the test cases to the actual data.

Listing 6.9: Unit test of the G-code string generation, where single commands are generated.

```
1  [Theory]
2  [InlineData("Ultimaker2Plus", 0, 0, 200, 200, 7200, "")]
3  [InlineData("Ultimaker2Plus", 0, 0, -200, 0, 7200, "")]
4  [InlineData("Ultimaker2Plus", 0, 0, 0, -200, 7200, "")]
5  [InlineData("Ultimaker2Plus", 0, 0, -200, -200, 7200, "")]
6  [InlineData("Ultimaker2Plus", 0, 6.867, 0, 7.014, 1800, "G1 X111.5 Y118.514 E0.01467
       F1800\n")]
7  public void LinearMove4Test(string printerName, double xCurrent, double yCurrent, double
       xTarget, double yTarget, int f, string expected)
8  {
9      //Arrange
10     var pointCurrent = new Point(xCurrent, yCurrent);
11     var pointTarget = new Point(xTarget, yTarget);
12     var printerSettings = new PrinterSettings(printerName);
13     var gCodeStringsGenerator = new GCodeStringsGenerator(printerSettings);
14     GCodeStringsGenerator.E = 0;
15
16     //Act & Assert
17     if (expected.Equals(string.Empty))
18     {
19         Assert.Throws<OutsideBedException>(() =>
               gCodeStringsGenerator.LinearMove(pointCurrent, pointTarget, f));
20     }
21     else
22     {
23         var actual = gCodeStringsGenerator.LinearMove(pointCurrent, pointTarget, f);
24         Assert.Equal(expected, actual);
25     }
26 }
```

The next test is on the generator that creates the surface layers, being the top and bottom layers. Here a file containing the expected result has been provided to the test, along with some test parameters. As there is only a need to test a single layer instead of the five that is normally generated, only the data for a single layer is provided. The test can be seen in Listing 6.10, where the test data is first set on line 2. In the arrange part on line 6 the `printersettings` object is created with the test data as input. On line 8 to 10 the list of points needed in the test is loaded to a string and then converted to a list of points. An explicit recalculation of the outer diameter is done to match the used diameter in the provided test data (see line 12). On line 13 an instance of the `SurfaceGenerator` class is created, which handles the creation of a surface layer. The expected data is loaded to a string on line 15, and on line 17 a specific character is removed from the string to avoid errors in the test. The current extrusion is on line 18 set to 0. In the act part on line 21 a string that consists of a single surface layer is generated and saved to the variable `actual`. The `expected` string is compared to the `actual` string to ensure that these are equal.

Listing 6.10: Unit test of the method to generate a single, horizontal filament layer described in G-code commands.

```
[Theory]
[InlineData("Ultimaker2Plus", 0.27, 50.5, 1, true)]
public void GenerateSingleLayerTest(string printerName, double z, double outerDiam, double
    infill, bool parallelWithXaxis)
{
    //Arrange
    var printersettings = new PrinterSettings(printerName);

    var printOrder = File.ReadAllText(
    @"..\..\..\GCodeGenerationTests\TestFiles\PrintOrder.json");
    var pointList = JsonConvert.DeserializeObject<List<Point>>(printOrder);

    outerDiam /= 3; //Explicit calculation
    var surf = new SurfaceGenerator(outerDiam, printersettings, pointList, infill);

    string expected = File.ReadAllText(
    @"..\..\..\GCodeGenerationTests\TestFiles\SurfaceTest3.txt");
    expected = expected.Replace("\r", "");
    GCodeStringsGenerator.E = 0;

    //Act
    string actual = surf.GenerateSingleLayer(parallelWithXaxis, z);

    //Assert
    Assert.Equal(expected, actual);
}
```

The final test for the G-code generation is a test for the wall generator. The wall generator will generate one layer of the gear's wall, being the sides of the gear, each time it is called. For this test, another file with test data has been provided that consists of a single layer of wall. The test can be seen in Listing 6.11. In the arrange part on line 6-8 the string containing the expected data is loaded to the string called `expected`. The list of points is then on line 10 loaded to the string called `cordlist` and converted to a list of points on line 12. The `printerSettings` objects is then on line 14 created with the test data and an instance of the `WallGenerator` class is then created on line 15, this will handle the creation of the wall layer. In the act part on line 19 a string that consists of a single wall layer is generated and saved to the variable `acutal`. On line 22 the test is run by comparing `actual` to `expected`.

Listing 6.11: Unit test of the method that generates the walls on the perimeter of the gear.

```
[Theory]
[InlineData("Ultimaker2Plus", 0.27)]
public void WallGeneratorStartTest(string printerName, double z)
{
    //Arrange
    string expected = File.ReadAllText(
    @"..\..\..\GCodeGenerationTests\TestFiles\WallOrderTest2.txt");
    expected = expected.Replace("\r", "");
```

```
9
10      var cordlist = File.ReadAllText(
11      @"..\..\..\GCodeGenerationTests\TestFiles\PrintOrder.json");
12      var pointList = JsonConvert.DeserializeObject<List<Point>>(cordlist);
13
14      var printersettings = new PrinterSettings(printerName);
15      var wall = new WallGenerator(printersettings, pointList);
16      GCodeStringsGenerator.E = 0;
17
18      //Act
19      string actual = wall.Start(z);
20
21      //Assert
22      Assert.Equal(expected, actual);
23  }
```

All tests conducted within this area passes once run. In total there is four tests that runs the linear movement methods, one test that runs the surface generator method, and lastly one test that runs the wall generator method. All tests worked as intended and the tests passed.

6.2 Integration Test

Integration tests are tests that are performed on two or more units. The purpose of an integration test is to validate that the units will work together as intended [30]. For MMAG an integration test is performed of the syntactic and contextual analyses as a whole and of the code generation as a whole, to make sure that these units also work as intended when they are integrated together. Finally, an integration test has been made of the entire compiler to make sure that the compiler as a whole does what it is intended to do.

6.2.1 Syntactic and Contextual Analysis

The scanner, parser and contextual analysis phases ensure that the source program is being interpreted correctly. To conclude that this is the case, integration tests have been conducted, involving the three aforementioned phases. An example of such test can be seen in Listing 6.12, where an expected symbol table is compared to a symbol table created by the contextual analysis phase, which is based on the output from the scanner and parser phases. The integration test is, like the unit tests, divided in an arrange, an act and an assert part. In the arrange part on line 5-13 a source program is first assigned to a string called `source`. The expected symbol table is then created on line 14-25 and assigned to the variable called `expected`. The act part from line 27 to line 28 first uses the scanner and parser class to generate an abstract syntax tree with the content from `source`. Then the abstract syntax tree is used to create the symbol table which is then assigned to the variable called `actual`. The assert part

happens on line 31 where a helper method called `AreSymboltablesEqual()` is used to compare the content within the actual symbol table to the content within the expected symbol table. The `AreSymboltablesEqual()` method is used as input to the `Assert.True()` method, which expects the two symbol tables to be identical.

Listing 6.12: Test of syntactical and contextual analysis of source program, resulting in a symbol table.

```
1  [Fact]
2  public void ScannerParserToSymbolTableTest()
3  {
4      // Arrange
5      string source = @"PrinterModel = Ender5;
6                        Type = SpurGear;
7                        NumberOfTeeth = 10;
8                        InternalDiameter = 40;
9                        OuterDiameter = 50;
10                       Height = 5;
11                       InfillPercentage = 0.5;
12                       Var one = 0.5 * 2;
13                       Var two = -200 / 10 + 5;";
14     SymbolTable expected = new()
15     {
16         ["PrinterModel"] = "Ender5",
17         ["Type"] = "SpurGear",
18         ["NumberOfTeeth"] = 10,
19         ["InternalDiameter"] = 40,
20         ["OuterDiameter"] = 50,
21         ["Height"] = 5,
22         ["InfillPercentage"] = 0.5,
23         ["one"] = 1,
24         ["two"] = -15
25     };
26
27     // Act
28     AST ast = ScannerParser.ScannerParser.GenerateASTfromSourceProgram(source);
29     SymbolTable actual = ast.BuildSymbolTable();
30
31     // Assert
32     Assert.True(AreSymboltablesEqual(expected, actual));
33 }
```

Overall two integration tests were conducted to ensure that the scanner, parser and contextual analysis phase works together as intended, and both tests passed.

6.2.2 Code Generation

Having ensured that the `GearPointGeneration` project, and the `GcodeGeneration` project each perform as intended when used separately, an integration test has been conducted to ensure that they

also work correctly together. Together the two units constitute the code generation phase, and it is therefore crucial for the entire compiler, that these perform as intended when integrated together. The integration test of `GearPointGenerator` and `GcodeGeneration` can be seen in Listing 6.13. In the test two strings containing G-code commands are compared to check that they are identical. The arrange part of the test can be seen from line 5 to 16, in which a symbol table is first initialised with test values, and the needed objects for generating the points and the G-code are instantiated. On line 15-16 the file with the expected result is loaded to a string called `expected` for it to be used in the assert part. In the act part on line 19, the coordinate points are first generated. Then the actual G-code is generated on line 20, and saved to the string called `actual`, with the `GenerateGCode()` method. The assert part is on line 24 where the `Assert.Equal()` method checks if the two strings are identical.

Listing 6.13: Integration test of G-code generation based on a symbol table.

```
1  [Fact]
2  public void PointsAndCodeIntegration()
3  {
4      //Arrange
5      SymbolTable symboltab = new();
6      symboltab["PrinterModel"] = "Ultimaker2Plus";
7      symboltab["NumberOfTeeth"] = 10;
8      symboltab["OuterDiameter"] = 50.5 / 3;
9      symboltab["InternalDiameter"] = 41.2 / 3;
10     symboltab["Height"] = 3;
11     symboltab["InfillPercentage"] = 1;
12     GearPointGenerator actualPointsGenerator = new GearPointGenerator(symboltab);
13     var gCodeGenerator = new GCodeGenerator();
14
15     var expected = File.ReadAllText(@"../../../IntegrationTestFiles/GcodeFullGear.gcode");
16     expected = expected.Replace("\r", "");
17
18     //Act
19     actualPointsGenerator.GenerateJsonPointList();
20     var actual = gCodeGenerator.GenerateGCode(symboltab);
21     actual = actual.Replace("\r", "");
22
23     //Assert
24     Assert.Equal(expected, actual);
25 }
```

The integration test for the code generation passed, as the actual string generated in the test was equal to the expected string. It is therefore concluded that the code generation part performs correctly.

6.2.3 Complete Compiler

To be able to conclude that the compiler works as intended, an integration test of the complete compiler had to be conducted. The integration test of the complete compiler involves all the different phases in the compiler, meaning that it begins with a `.mmag` source program and ends with a `.gcode` file generated by the compiler based on the source program. In the arrange part of the integration test, the file containing the expected G-code commands are loaded to a string called `expected`. This happens on line 5. Like previous tests, the carriage return characters are removed on line 6. The string called `args` on line 7 contains the file path for the source program that is to be compiled. When using the compiler, the user would normally specify the source program directly in the console, and this is also what happens on line 11-12, where the source program is loaded to the compiler. The first phase in the compiler is the syntactical analysis, see line 15, where an abstract syntax tree is generated based on the content in the source program. On line 18 the contextual analysis phase will create the symbol table based on the abstract syntax tree from the previous phase. The next phase is the gear point generation phase, which can be seen from line 21 to line 24. First on line 21 an object of the `GearPointGenerator` class is instantiated with the generated symbol table as input to its constructor, and then on line 22 the `GenerateJsonPointList()` method generates the gear coordinate points. On line 24 the static property `E` on the `GCodeStringsGenerator` class has to be set to zero to make sure that the value of `E` will be calculated correctly in the G-code generation phase, which can be seen on line 27 to 28. Here the G-code is generated and on line 31 and 32 the G-code commands from the `gCodeProgram` string is saved to a file with the same name as the source program, but with a different file extension. Similar to how the expected file was loaded to a string, the actual file in which the G-code commands were saved, is also loaded to a string and the carriage return characters are again removed.

Listing 6.14: An integration test that checks if an entire program is correctly compiled.

```
1  [Fact]
2  public void ProgramIntegrationTestSuccess()
3  {
4      //Arrange
5      var expected = File.ReadAllText(@"../../../IntegrationTestFiles/GcodeFullGear.gcode");
6      expected = expected.Replace("\r", "");
7      var args = new[] { @"../../../IntegrationTestFiles/sourceprogram.mmag" };
8
9      //Act
10     // load source program from file
11     FileInfo sourceFile = new(args[0].Trim());
12     var sourceProgram = Program.LoadSourceProgram(sourceFile);
13
14     // syntactical analysis
15     var AST = LexerParser.LexerParser.GenerateASTfromSourceProgram(sourceProgram);
16
17     // contextual analysis
18     var symbolTable = AST.BuildSymbolTable();
```

```
19
20      // point generation
21      GearPointGenerator gear = new(symbolTable);
22      gear.GenerateJsonPointList();
23      // reset E
24      GCodeStringsGenerator.E = 0;
25
26      // g-code generation
27      GCodeGenerator gCodeGen = new();
28      var gCodeProgram = gCodeGen.GenerateGCode(symbolTable);
29
30      // save to file
31      var outputFileName = sourceFile.Name.Replace(".mmag", ".gcode");
32      File.WriteAllText($"{sourceFile.DirectoryName}\\{outputFileName}", gCodeProgram);
33      // load file
34      var actual = File.ReadAllText(@"../../../IntegrationTestFiles/GcodeFullGear.gcode");
35      actual = actual.Replace("\r", "");
36
37      // Assert
38      Assert.Equal(expected, actual);
39  }
```

At last the two strings containing the expected G-code commands and the actual generated G-code commands are compared in the `Assert.Equal()` method. This test passed, meaning that the complete compiler does indeed perform as intended.

6.3 Code Coverage

Overall, 74 tests have been conducted, these include both the unit tests and the integration tests. Code coverage describes the percentage of code that has been covered in the tests and can be a indicator of whether enough tests have been conducted. When testing code the aim should be to cover a larger percentage of the code to ensure that the program works as intended [41]. However it is more crucial to test the most important parts of the code correctly, rather than aiming only for a high code coverage percentage, but usually it is reasonable to aim for a code coverage percentage at around 80 [46]. Different frameworks can be used to analyze and visualize the code coverage for a project and for the developed compiler, ReSharper Ultimate has been used [32].

The code coverage for the developed compiler can be seen in Figure 6.1. The figure shows an overview of the total code coverage, as well as the code coverage for the different projects. From Figure 6.1 it can be seen that 77.80% of the code blocks in all the different projects have been covered in the tests.

Looking into the code coverage of the projects individually, it can be seen that both `GearPointGeneration` and `GCodeGeneration` have a code coverage just above 90%, meaning that almost every part of the

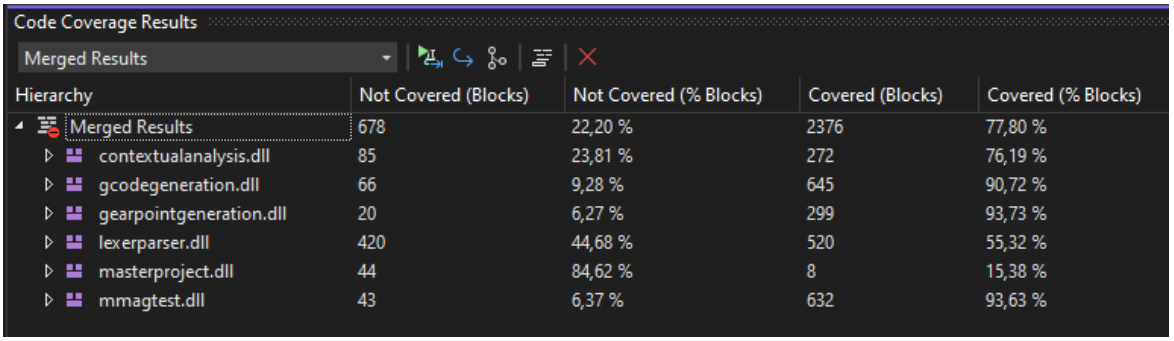

Figure 6.1: Overview of the code coverage in the different code projects for the developed compiler.

code in these two project have been tested. The code coverage for the `ContextualAnalysis` project is 76.19%, whereas the code coverage for the `LexerParser` project is only 55.38%. This is due to most of the code blocks in the `LexerParser` being generated by ANTLR, and the focus when testing has been to test the developed code rather than the code generated by ANTLR, as this is assumed to perform as it should. When testing the compiler, the code blocks generated by ANTLR has been involved, however not tested individually, why the code coverage for the `LexerParser` is lower than the for the other projects.

Looking at the code coverage for the `MasterProject`, which is where all the phases of the compiler are connected, it seems rather low, as only 15.38% of the code has been covered in the tests. Only the method for loading the source program in the `MasterProject` has been tested directly, as it is crucial that the source program is loaded correctly. The `Main()` method in `MasterProject` is where the other projects are called, meaning that this method does not hold any logic itself. Therefore it has not been relevant to focus on this project in the tests. The integration test seen in Listing 6.14 is however very similar to the actual `Main()` method, and the content of `Main()` method has therefore been tested through this integration test.

Overall it can be concluded that, even though not all of the code projects have a code coverage at 80% or above, the most important and crucial parts of the code has indeed been tested enough to ensure that the compiler as a whole, perform as intended.

6.4 Acceptance Test

The purpose of this section is to evaluate the compiler against the requirements setup in the MoSCoW Table 4.2, and the section will also include a visual evaluation of the gear generated based on a compiled source program written in MMAG language. There will only be mentioned fulfilled requirements, mainly those placed in the *must have* and *should have* categories of the MoSCoW, where all requirements have been met. Since it is an educational project, there are no customers to test

the product, which also means that all the acceptance tests are conducted without external review. Therefore, the compiler is tested based on the requirements from the MoSCoW.

6.4.1 Must Have Requirements

As mentioned before, the *must have* requirements are the minimum requirements for the compiler, therefore it is very important to have these requirements fulfilled.

M1	Easy design of simple spur gears using the MMAG
M2	Compile a program accepted by the language into a G-code file that is accepted by a 3D printer
M3	Informative error messages, if a program is not accepted by the language
M4	One way to assign a value to a gear attribute in the language.
M5	Is a console application.

Table 6.1: *Must have* requirements for the language and compiler.

According to the first *must have* requirement, M1, in Table 6.1, it should be easy to design a spur gear using the MMAG language. It is assumed that making a gear is easy because the user does not have to make all the calculations for the designed spur gear. The language provide several attributes that helps the user to customize the desired spur gear such as the number of teeth or the proportions of the gear. These attributes can also be independent settings such as the infill percentage. The language in its current state only supports spur gear generation, since there are different calculations needed for generating other types of gears. In the second *must have* requirement, M2, it is required that a source program can be compiled into G-code so that it can run on a 3D printer and create a spur gear. The following Listing 6.15, is a MMAG source program, which is used as an input to the compiler.

Listing 6.15: Source Program in MMAG language.

```
1 PrinterModel=Ultimaker2Plus;
2 Type= SpurGear;
3 Height = 2;
4 OuterDiameter = 16.8;
5 InternalDiameter = 13.7;
6 InfillPercentage = 0.5;
7 NumberOfTeeth = 12;
```

When the source program in Listing 6.15 is compiled, the output will result in a G-code file, since the G-code file is too large to be shown, Ultimaker Cura [59] has been used to simulate the output G-code and give a visual representation of what the G-code would look like, if it were to be run on

a 3D printer. Since the visual representation seen in Figure 6.2 shows a spur gear which has been accepted and printed by 3D printer, requirement M2 can be seen as fulfilled.

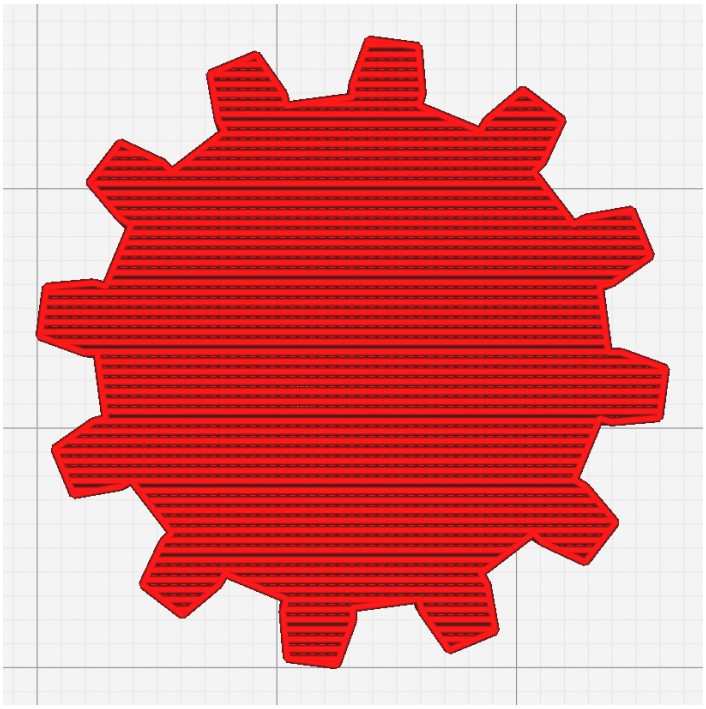

Figure 6.2: Spur gear visualisation made in Ultimaker Cura.

Requirement M3 states that when a source program is not accepted by the language, an informative error message should be thrown containing the necessary information to the user. Right now, when an error is detected through out the different phases of the compiler, an error message is shown helping the user to identify the problem. In addition to the exception types for syntactic errors that are part of the scanner generated by ANTLR, 13 customized exception types have been made for semantic and other miscellaneous errors, where the contents of the displayed error messages depend on the exception type, and often the error messages are dynamically generated based on the specific error. As for the fourth must have requirement the user should be able to assign values to different gear attributes using the MMAG language as seen in Listing 6.15, where some of these attribute's values are used in calculations for the spur gear. The fifth and last *must have* requirement M5 which states that the compiler has to run in a command-line interface is also fulfilled.

The result of this acceptance test leads to the compiler fulfilling all the essential requirements written under the *must have* category in the MoSCoW, which means the compiler can successfully read a source program in MMAG language and generate an output G-code file that can then be used to print the desired spur gear.

6.4.2 Should have Requirements

This section will dive in the *should have* requirements for the compiler, these requirements should be fulfilled when the compiler is in a working state.

S1	Error messages with information on missing assignments of attributes, if such occur.
S2	Error message, if the design results in an object with dimensions that exceed the printer limitations.
S3	One purpose for each keyword.
S4	Loosely typed variables, which will be interpreted as either integer or double variables with implicit conversion.
S5	Similar rules of usage for variables, gear attributes, and numerals.
S6	Automatically generate 20% infill for all structures.
S7	Allow the use of arithmetic expressions.

Table 6.2: *Should have* requirements for the language and compiler.

The first requirements in Table 6.2 state that the compiler should be able to detect and throw an error message when attributes have missing assignments. The only exception is the attribute infill percentage since the requirement S6 states that the infill percentage should automatically be set to 20% density as default. This requirement is seen as fulfilled since the compiler at the current state throws an error explaining what gear attribute is missing assignment. To make sure that the 3D printer will not be damaged by the generating G-code, some calculations are made to determine the size of the gear so it will not exceed the build limitations of the 3D printer, if so an appropriate error message will be thrown as S2 indicates. To make the language easy to use, there will be one purpose only for each keyword, which can add simplicity to the language, by that the S3 requirement is fulfilled. In the compiler's current state, it is possible to have a variable that can be assigned with either a double or integer value. If the value changes from integer to double, an implicit conversion is made, this also applies for gear attributes. This means that the fourth and fifth requirement S4, S5 are fulfilled. According the requirement S7, the user should be allowed to use arithmetic expressions. Right now it is possible to use arithmetic expressions in the assignment of attributes and variables, and these will be evaluated and then assigned.

In this acceptance test of the requirements in the *should have* category, it is concluded that the compiler fulfills all the requirements in this category.

6.4.3 Could Have Requirements

The requirements that are categorized as *could have* can be seen in Table 6.3. As they are *could have* requirements, they are all extra features and are not necessary for the compiler to work.

C1	Easy design of simple bevel gears.
C2	User-defined infill percentage.
C3	Assumptions on what the program was intended to do, when syntactical errors are encountered. Notify the user about the assumptions.
C4	An embedded resource with start- and end code for the most commonly used 3D printers.

Table 6.3: *Could have* requirements for the language and compiler.

As a default the infill percentage is set to 20%, but as the *could have* requirement C2 states, it should be possible for the user to change the infill percentage. Right now the infill can be changed by changing the value for the gear attribute *infill percentage*, and if no value is represented it reverts to the 20% default.

The user can choose from different printer models, therefore some different embedded resources are made in order to make it possible, and a start- and end code for the most commonly used 3D printers are added to those embedded resources. Right now the language supports four type of printers, these are CR10, Ender3, Ender5, and Ultimaker2Plus. The Ultimaker2Plus has been the only printer available for testing. However, theoretically, it should work on the other types of 3D printers, because it is only the start- and end code that differ. And by that, the fourth *could have* requirement, C4, is considered fulfilled.

Only some of the requirements prioritised as *could have* have been implemented, those are C2 and C4, where the user now can have a user-defined infill percentage and the ability in choosing from different type of 3D printers.

At the end it can be concluded that all the essential requirements are fulfilled, as all the requirements prioritised as *must have* and *should have* requirements have been implemented, and half of the *could have* requirements are implemented as well.

Chapter 7

Discussion

In this chapter, the solution to the problem stated in chapter 3 is discussed. First, a general overview of the solution is provided, followed by an evaluation of the solution. The evaluation is split into three parts, focusing on general advantages and disadvantages, the language design, and the architecture of the implemented compiler. Then an evaluation is made of the validation of the compiler's functionality, and suggestions are made as to how the compiler and the language could be further improved. Finally, we have documented our reflections on the project process and group collaboration in the last section of this chapter.

7.1 Problem and Solution

Generally speaking, the purpose of this project was to develop a new way to design gears with the possibility of converting the design into a .gcode file that could be executed on a 3D printer, in the end resulting in a physical gear. The proposed solution is a new programming language called MMAG, which is an abbreviation of *Make Me A Gear*. In the language, a model of a gear can be designed by assigning values to essential gear attributes. For a spur gear, which is the type of gear that is supported in the solution at this point, these essential attributes are the internal diameter, the outer diameter, the number of teeth, and the height of the gear. An additional attribute can be set to define the infill percentage inside the printed gear.

Based on the model, a .gcode file is generated as the output of the implemented translator. It is often a good idea to use an interpreter as a translator, as it eases the debugging process, but the nature of usage of this language means that a compiler is a better choice, as the G-code program that the compiler makes, is supposed to be executed on another platform than the one on which it is created. The compiler is used on a personal computer and it will generate a .gcode file that can be moved to

and executed on a 3D printer.

One important benefit of interpreters is that they compile one statement at a time and can show exactly where in the source code potential errors occur. Compilers are not able to do this, so to compensate for this disadvantage, our compiler has customized exception handling, and all displayed error messages contain information about the specific error that happened, often with suggestions as to how the user can solve the problem.

7.2 Solution Evaluation

In this section, we go into more detail with the discussion of the advantages and disadvantages of the solution, and an evaluation of the language and the implementation is made.

7.2.1 Advantages and Disadvantages of the Proposed Solution

The models of gears that a user can make in the MMAG language are rather simple compared to the ones created through the use of a CAD software. The focus has been more on simplicity, as the main focus was an easy-to-use language that would create a gear through simple code. With simplicity as a focus, there is a trade-off when it comes to flexibility. As the coding language had to be simple, the amount of options given to the user have been reduced, resulting in a limited amount of flexibility.

Another difference seen from the compiler's gears and the ones made through CAD software, is the number of wall layers and the printing process. For a gear that was made in Blender and sliced in Ultimaker Cura, the model would have a *skirt*, which is a very thin layer below the printed object that makes it stick better to the bed surface. The model would furthermore have three wall layers, where the outside layer would be printed slower to improve the visual quality. Calculating the points for inner wall layers was no easy task to do, so it ended up becoming a low priority feature, due to how complex and time-consuming it was to create. This could have figured in the MoSCoW prioritisation as a *will not have* requirement.

The compiler is essentially also a slicer, due to how it processes the given information, i.e., the model. The responsibilities of a slicer mentioned in section 2.6 also apply to our compiler, with a minor exception. Our compiler handles errors, should they arise, and also slices the model into 2D layers. The compiler generates layers of infill and convert the model to G-code. The only task our compiler cannot complete is generating support for the model. However, this is on purpose, as the models created by the compiler, these being the gears, do not require support to be printed. Additionally, most slicers usually feature a lot of settings the user can control, which this compiler

does not. Some of the common settings slicers usually feature, is allowing for the temperature of the bed or the nozzle to be set to a specific temperature. The compiler allows the user to change the infill percentage and the printer model as the only non-model related settings.

Due to some problems with the printer, we experienced a lot of *stringing*, while printing some of the early models. Stringing describe the phenomenon, when the printer leaves very thin strings of filament on moves where it was not supposed to extrude any filament. These strings end up as clutter on the model or between printing points on it, decreasing the quality of the printed models. This occurred primarily between the teeth of the gear, as the printer had to move over a gap, leaving strings of plastic on every move. This was solved in two ways; the first being a change in the printing temperature, and the second being a change of the tool path. Changing the temperature reduced the stringing overall, but some still remained. This was further fixed by moving the tool path to only move within the gear, fixing the problem with stringing over any gaps. This way, if any stringing should occur, it would happen inside of the gear along with the infill, which in turn just becomes a very small amount of extra infill. One of the downsides for this fix, is the printing time. The nozzle now returns to the center before every movement, which makes the printing time longer as the tool path in general becomes longer.

For this project, we only had access to one printer. Therefore, we were discussing within the group whether the stringing was a problem of the generated .gcode files, or if the problem was a hardware problem. This is always a challenge when working with hardware, and we could have tried to print the .gcode files on another printer to identify the cause of the problem. It might not have been necessary to make any adjustments to the tool path generated by our compiler.

7.2.2 Language Evaluation

In section 4.1, three of the characteristics in Table 4.1 were selected to be the focus for the design of the MMAG language and compiler, namely simplicity, orthogonality, and exception handling. These were selected to ensure good readability and writability. Programs written in MMAG are usually quite small and it is easy to figure out what the result will look like. A reason for this is that the language only has a small number of basic constructs because it is a very specialized language for one task only. Another reason is that there is no feature multiplicity so there is only one way to accomplish a particular operation. Although it can be useful to have some feature multiplicity, we decided that is was important to keep it simple so inexperienced users can learn the language quickly. The language is reasonably orthogonal since almost all types can be combined in arithmetic expressions. The only exception is `PrinterModel` and `Type` because these are keywords that the compiler can not treat as variables, since their values are of type *string*.

After finishing the development of the compiler, we found an error in the grammar of our language.

The problem is that the parser does not recognize the name of a printer if it does not contain a number. This means that printer models without numbers in the name are not interpreted correctly. This is caused by the fact that the parser matches strings without numbers to an IdString instead of a PrinterString. Unfortunately, all printers used for testing contained a number in their name, so we did not detect this problem until after the development, when we were adding an extra exception for unsupported printer types. A way to fix this would be to differentiate between IdStrings and other string by requiring the user to write strings inside quotes like many other programming languages also do. It is a relatively simple fix, however we did not have time to make changes to the compiler.

The grammar forces the user to write the printer model and gear type at the top of the source file. On one hand, it creates a nice header for the source file, but on the other hand some user might find it annoying that they are forced to write it like that. The reason why the gear type must come in the beginning of the source program is that if more gear types had been supported, it would have been important to know early in the contextual analysis, which type of gear the user wanted to make. In section 5.2 it is explained that when the symbol table is created, keys for each attribute are inserted. Because different gear types have different properties, the gear attributes vary depending on the gear type. For example, a helical gear would require the user the specify the helix angle, while a spur gear would not. Values of `StmtNode` are inserted into the symbol table while traversing and evaluating the nodes of the AST, so the properties for each gear type must be declared in the symbol table before that. Otherwise, the user has to declare the gear attribute as variables themself which is something we wanted to avoid. Another solution to this problem would be the traverse the AST twice. The purpose of the first pass would be to only search for a `GearTypeSelectionNode` and create a symbol table according to the selected gear type. Then the second pass could insert values into the symbol table as normal.

7.2.3 Evaluation of Implementation Architecture

In the implementation of the compiler, its responsibilities have been split into several projects and classes to ensure high cohesion within the classes. The syntactical analysis and the contextual analysis phases of the compiler have been split into one separate project each, and the final phase of the compiler – the code generation – has been split into two projects, where the classes of the first project are responsible for calculating the necessary points to define the outer perimeter of the gear shape, and the classes of the second project generate the actual G-code. A main project has been used as entry point, which has references to the four projects implementing the compiler phases in order to be able to invoke their methods in the correct order.

In general this has resulted in a good separation of concerns. An example of this, is that in the generation of the G-code, one class computes the start and end coordinates of the lines that must be drawn on the 3D printer, and another class is responsible for generating the actual G-code commands

based on the start and end points received from the first-mentioned class.

One area where improvements could still be made in terms of ensuring high cohesion in the classes is that several classes require some kind of geometric calculations, e.g., determining the intersection between two lines. The methods used to calculate these points are placed within the classes who needs them. As this solution implements one gear type, the methods are not duplicated, but if more gear types were to be implemented, we would potentially risk copying the code for these methods to other classes. Instead, the geometric methods should be gathered in a static geometry helper class in a separate project, which the other projects could have references to.

In terms of coupling, the relatively high cohesion within the projects and classes automatically lead to a relatively low coupling between projects and classes. The only interesting exception to this is how the node classes, which the abstract syntax tree is composed of, are used in both the syntactical analysis project, when the CST is converted into an AST, and also in the contextual analysis, when the tree traversal is performed. This is implemented by having the classes in the contextual analysis project, and letting the syntactical analysis project have a reference to the contextual analysis. This results in a relatively tight coupling between those two projects, however, that means that if changes are made to the AST nodes, they are instantly applied to both the syntactical and contextual analyses. The question that remains up for discussion is where the classes should ideally be located within the solution.

Overall, a good level of abstraction and separation of concerns have been achieved. We have had good use of this in the development phase, as the division of the solution also has allowed us to work simultaneously on the project. For a couple of weeks, all group members were working on the development of the compiler, and this was remarkably easily facilitated, due to the division of the solution into several projects and classes.

7.3 Evaluation of Tests

When developing a new programming language and an associated compiler, it is essential that everything performs correctly, and several tests were therefore conducted of the developed MMAG compiler to validate that this was the case.

The total code coverage of the code for the developed compiler was at 77.80%, which seems quite acceptable. However, the code coverage percentage does not express anything about the quality of the tests. The fact that the tests are of high enough quality to discover possible errors in the code is crucial for validating its correctness. In several tests for the compiler the *[Theory]* attribute was used to create different test cases for making sure that several cases were considered, as these might help discover different errors. That a test had different test cases helped ensuring that the method

worked correctly, even on inputs that we had not thought of when developing the code. In the end the methods that were tested using different cases are more likely to not have any undiscovered errors, as several inputs were considered. In other tests, either the *[Fact]* attribute was used, or only one test case was considered. In the methods that were tested without different cases, we can not assure that there are no errors in these, as only one input was considered. The input used might not be one where the method is thoroughly tested, as the method might work correctly on average inputs, but if a different input was given, the method might not be able to handle this. Overall, several methods were tested more thoroughly with different inputs, but not all, and the quality of the tests might have been increased if all methods had been tested with several inputs.

Apart from making sure that the methods are tested with different inputs, it is also important to make sure that the expected results of the tests are actually a valid indicator of the correctness of the method. The integration test of the complete compiler, as well as some of the unit tests for the methods for calculating points and generating G-code commands, are a bit questionable as to whether the tests are actually reliable. The files containing the expected results are generated by the methods that are being tested, and these tests therefore serve more as a stability test. Having tested that the generated code actually was printed correctly ensured that the points calculated and the G-code commands generated indeed were correct at that specific time. However, if changes were made to the code, e.g., an optimization of the tool path, which would change the entire structure of the G-code commands, the methods might not necessarily be wrong, even though the tests might fail, as there can be different results that would print the same correct gear. These tests were therefore better suited for ensuring that when making smaller changes, that did not affect the order in which the G-code commands were generated, the methods still gave the same output.

When we developed the code for the compiler, testing was first considered in the very end of the process. In the future, testing could be thought of earlier in the process, as this might help with discovering errors when developing, and not just at the end. After having made the tests, we experienced that when minor changes were made in the final weeks of the project work, it was easy to ensure that the code still functioned as it was supposed to, simply by running the tests. The tests can be run whenever changes are made, and if a test suddenly fails, it might be easier to discover what went wrong, and where the error is, instead of having to debug the entire code before discovering the error. Having the tests when developing the code can therefore help ensuring that the changes will not break the code.

7.4 Future Development

For future development, some improvements could be made to the existing features of the compiler, and some functionalities could be added to improve the user experience. As not all requirements from Table 4.2 were fulfilled for the compiler, due to the time frame of the project, these could be

added in the future to extend the compiler.

Currently, the compiler supports 3D printing of one gear type, namely spur gears, and to make the solution more general for gears of different types, the functionality for creating bevel gears could be added. It would also improve the user experience, if it was implemented that the compiler could notify the user with an assumption on what the program was intended to do, if syntactical errors were encountered, e.g., the compiler could add a semicolon, if a semicolon was missing on a specific line. Then the compiler could finish the compilation, even if the program had small syntactical errors.

Other requirements that currently have not been fulfilled are the four *will not have* requirements. Currently, there compiler can not generate support structures for the models, as this has not been necessary with the intended use of the language. However, in the future this could be implemented, as this is normally a job for a slicer, and it might be useful for more complex gear types. If the user needs several gears, one currently has to write several MMAG programs and print them individually, which might be quite a lengthy process, and the printer would have to heat up several times. This could be solved, if it was possible for the user to create multiple gears in the same program, which could then be printed in the same session. Another thing that would improve the user experience would be having a graphical user interface for visualising the designed gear before printing it, which would make it easy for the user to see if it looks like one wishes. Currently the user has to either copy the G-code to an G-code visualiser or print the entire gear before the design is visualised, and this functionality would therefore be good to implement. Finally, W3 could be implemented such that the width of the printed lines could be included in the code. This is especially important in regards to the width of the teeth on small gears, where the line width might be relatively big compared to the width of the teeth.

When it comes to the parts that are currently implemented, some of these could be optimised. For instance, each layer of the gear is calculated when it is used, but in reality many layers are similar to each other with the exception of the z-coordinate. As the z-coordinate only has to be set once for each layer, the G-code command sequence that generates, e.g., the bottom layer could also be used for the top layer by saving the command sequence, and then setting the z-coordinate independently before the commands for the full layer were added.

Another improvement that could be implemented in the future, which was not considered in MoSCoW analysis, is to make it possible to create the gears with a hole, e.g., a keyway, in the middle for mounting the gear on a shaft. There could therefore be added extra attributes for the user to be able to express the needed information.

7.5 Project Process

In order to complete this project as a group, some thoughts have been put into how a good project management, group collaboration, and supervisor collaboration could be achieved. In the following sections, the employed strategies are evaluated.

The problem for the project was identified by first choosing to make a programming language for programming of CNC-machines, which was one of the project suggestions presented at the first day of the semester. This decision was made based on the group members' personal interests. Through brain storming and reading through internet fora, we discovered that CAD programs sometimes have a weakness in terms of their abilities to design gears, so that was our initial problem: to design a language that could be used to easily generate gears. Based on this initial problem, we mapped a set of questions, we would have to answer, before the final problem statement could be made. These questions provided the foundation for the structure of the problem analysis, after they had been categorised and prioritised.

Project Management

Project management highly affects the productivity and structure of a group collaboration. How this project has been managed in terms of time and usage of tools, is described in this section.

Time Management

Time management has been performed on different levels. The day-to-day- and weekly planning has been done on morning meetings and Friday meetings, respectively, which are described in further detail under Group Collaboration. The overall planning of the entire project period was done in the beginning of the project in the shape of the Gantt diagram seen in Figure 7.1.

On the Gantt diagram in Figure 7.1, the broad, blue bars represent the original plan, whereas the thin, pink bars represent the actual progress in the project. In the original plan, the first four weeks had been dedicated to finding and analysing a problem for the project. Then seven weeks had been dedicated to the development of a solution, after which two weeks had been reserved for documentation and testing of the solution. Finally, one week had been dedicated to writing the final parts of the report, these being the discussion, the conclusion, and the abstract, and one full week had also been set aside for proofreading. In combination with the half week available in week 21, this gave a sufficient buffer at the end of the project period.

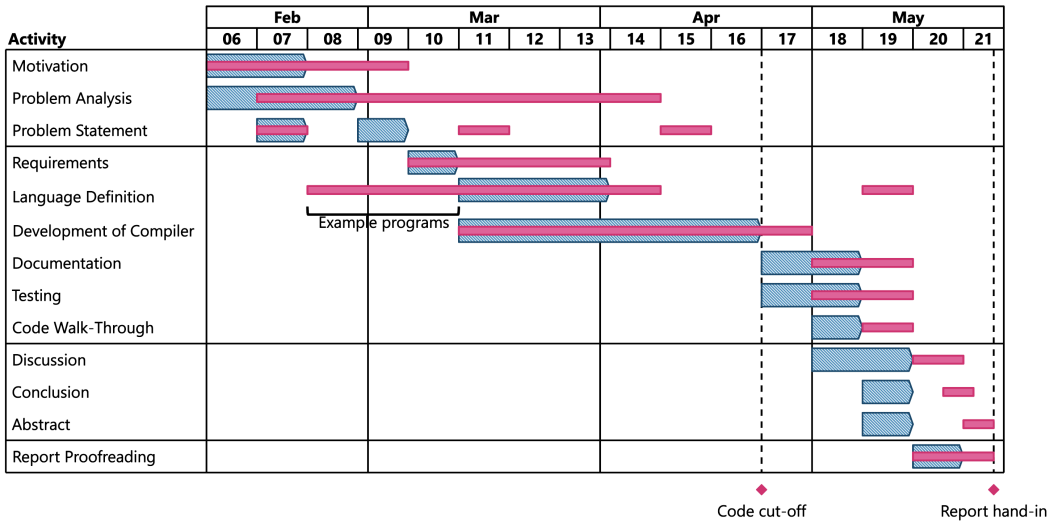

Figure 7.1: Gantt diagram of the project period with blue bars marking the original pan, and thin, pink bars marking the actual progress.

In reality, several adjustments were made to the plan, as certain parts of the project required less or more time than expected. First of all, the project was not the only task at our hands in the beginning of the period, as the group members were also attending courses. Though we did take this into consideration, we probably underestimated the affect of this on the progress of the project. Secondly, we were encouraged to develop example programs very early in the process, which also meant that resources were moved from working on the problem analysis to developing example programs and designing the language. In weeks 11, 12, and 13, the main effort was thus concentrated around the language definition and the compiler development, and in week 14 we then finalized the problem analysis, meaning that we could also make the final adjustments to the problem statement in week 15.

For this project, we decided to set a cut-off date for when the last bit of code should be added to the compiler. The purpose of this was to ensure that there was enough time to test and document the compiler after its development. We did postpone the deadline for one week, but that was a minor adjustment. Overall, having the deadline worked well, and it ensured that we worked hard in weeks 16 and 17 with some long days, but we could go back to normal hours again in week 18, when we started testing and documenting the compiler. Having a code cut-off date is something we will bring with us into the next projects.

Version Control

For version control of the developed solution, we used git via the GitHub platform [26]. When adding new features, we created new branches, which had to be reviewed and approved by other group members, before they were merged into the solution. We also used GitHub's *issues* to administer,

what had to be done, and who was working on the different parts of the solution. This was done by using tags to mark importance, and by assigning group members to issues, such that it was evident, who was responsible for what. In general, this was less relevant in the beginning of the development phase, but in weeks 16 and 17, all group members were writing code for the compiler, and the GitHub tools came in very handy for this part.

Project Report Platform

For writing the report, we chose to use *Overleaf* [45], which we are all familiar with. When using overleaf, it is easy for multiple people to work on the same report at the same time, as the content can be split into multiple files that are put together in *main* files. Overleaf also ensures a nice and consistent look, when Overleaf is used with a good template.

File Sharing

For general file sharing and storing common files, we used *OneDrive* [42], which we have access to through the university. Here, we stored documents like the minutes of our Friday- and supervisor meetings, as well as the results of brainstorms, mainly from the beginning of the project.

Group Collaboration

In the beginning of the group collaboration, a group contract was developed in which the common group expectations were discussed and the foundations for the collaboration were set. The group contract was divided into different parts these being; meeting times, requirements for "the good report", project management and social events. The group quickly agreed on the meeting times and that most of the group work should be physical rather than online. If a group member would be more than five minutes delayed this should be informed to the rest of the group via Discord or Messenger. The individual thoughts on what "the good report" involved was discussed and the common thoughts were included in the group contract. When discussing the project management part, the group agreed that the group should work in SCRUM alike sprints, which was quite a new way of working for the group members, where different tasks were to be solved during the sprint to ensure that the different parts of the project were finished in time. This way of working in sprints ensured progress on the project every week, why this could indeed also be a good way to work in future project groups. Furthermore it was important to the group that there should be room for social events during the project process. Small group events were held during the project time and this had a good impact on the group as it shook the group closer together.

When working on the project as well as attending the courses the group preferred it to be physical at the university most of the time, however as the project progressed the group ended up working a lot from home as well. When working from home, a daily meeting in the morning was held, as well a status meeting by the end of the day. This was good for ensuring that every group member participated in the group work, however some group members preferred to work during different time slots, which made it a bit difficult to always see the progress. In the future this should be discussed beforehand and it should be considered when planning status meetings. In the beginning all group members worked a lot together to agree on a subject to work with, and to identify a problem that all group members would find interesting to work with. As the project process moved forward, the group work became more individual which not all group members preferred. However this was first discussed in the very end of the project and in further groups one should express this earlier as it was to late to change in the end.

A weekly meeting was held, usually on Fridays, to discuss the process of the recent sprint, the group collaboration, and to agree on the tasks and length of the upcoming sprint. The purpose of the weekly Friday meeting was firstly to plan ahead with regards to the project and to make sure that each group member knew what to work with for the next sprint, but also to discuss the status of the dynamic within the group and to prevent possible conflicts. Overall the group dynamic was in the beginning characterised by being well functioning both socially and professionally. The Friday meetings were to be used as a place to let the other group members know if one was dissatisfied with the collaboration or if one was stuck with a certain task. The group ended up by working a lot individually, and sometimes with tasks one did not find very interesting. Even though this was discussed on the Friday meeting, nothing was done to change it, and with hindsight this is something that should have been considered better during the project. In the end of the project the group dynamic became less professionally functioning as one group member did not participate as much as one had before, and without giving a reason for the sudden change in ones participation. The group first tried to solve this without help from the supervisor, but if this were to happen in the future, the supervisor could be informed earlier. In the future every group member should also be better to use possible Friday meetings to express ones feelings as it is essential for a group to function, that every group member participates equally, or at least expresses why one might have a difficult period so that this can be taken into account.

Supervisor Collaboration

Regarding the collaboration with the supervisor, weekly meetings have been arranged to discuss any technical questions, as well as the progress with the project work and the report. The supervisor has informed the group on how the work was going in a good and useful way, e.g., by letting the group know, if the supervisor felt like we were progressing too slowly. This encouraged the group to further prioritise their efforts and focus the work, and we will continue to ask the supervisor for this kind of

information in the next project.

Before each super visor meeting, the group has sent different materials for the supervisor to go through before the meeting. The supervisor has sent feedback on reading materials ahead of the meeting, such that the group could look through the comments beforehand. This was a good approach, as everyone was then updated on the comments for the reading materials, which were also discussed on the meeting. We will encourage supervisors for the next project to do the same. The group also sent an agenda for the meeting to the supervisor with additional questions and/or topics for discussion. Especially when developing the syntax and CFG, the supervisor facilitated a very good and critical discussion, which improved the CFG. Sometimes, the discussions in the supervisor meetings spun a little out of control, in the sense that they might have been less relevant for the supervisor, e.g., internal discussions between group members about timing of next meeting or suggestions to solutions to technical challenges that the group could discuss internally, before they were discussed with the supervisor. The supervisor did not seem annoyed with this, but we will encourage supervisors for the next projects to let students know when they should save a discussion for after the meeting and just take it internally in the group, as this would strengthen the students' abilities to read situations like this.

The supervisor also assisted the group in solving an internal conflict, when one group member stopped participating in the group work for approx. two weeks, without giving any information to the group. After trying to get in contact with the group member, the group reached out for the supervisor's advise, and in the end the conflict was solved, and the group member remained part of the group. For the next project, we will include the supervisor early, and we will consider how to follow up on the conflict in a proper way, which we did not have a plan for in this situation.

Overall, we have indeed enjoyed working on the project, which we all found to be quite interesting. Throughout this chapter, the advantages and disadvantages of the proposed solution, i.e., the MMAG language and its compiler, have been discussed together with any improvements that could be implemented, if we had had more time on our hands. In the next, closing chapter, a conclusion will be made as to whether and how the problem in chapter 3 has been solved.

Chapter 8

Conclusion

The purpose of this project was to develop a high-level programming language, in which models of gears could be designed and compiled into G-code, in the end resulting in a physical gear, if the G-code commands were executed on a 3D printer. This was an interesting problem on both a theoretical and a practical level. The language should be easy to read and write, as it was meant as an alternative to complicated and expensive CAD programs.

Throughout the project period, a thoroughly worked-out solution to this problem has been proposed. The solution is indeed a programming language based on simple assignments to a set of relevant gear attributes that together constitute the gear model. The language has a simple syntax, is monolithically scoped, and provides implicit type conversion, such that the user does not have to worry about these aspects of the program. The compiler has a comprehensive set of custom exceptions that are used to provide the user with valuable information and potential suggestions for corrections, in case a source program cannot be compiled. This supports the user's familiarisation with the language. The compile time of a program is less than 0.4 seconds for a gear with an outer diameter of 50.5 mm.

The compiler has been validated through several tests that ensure that the compiler generates .gcode files that can be run on a 3D printer and actually result in a printed gear. The tests cannot necessarily be used for further development due to their nature, but they show that the compiler at its current state generates .gcode files that can be printed successfully. The acceptance test show that the solution fulfills the essential requirements, as well as some additional requirements. Combined with the results of the other tests, it can be concluded that throughout the project, a solid solution with a low compile time has been developed, which indeed provides a good alternative to traditional CAD programs.

Bibliography

[1] 3D printing. *What is 3D Printing?* URL: https://3dprinting.com/what-is-3d-printing/. (Accessed on 02/21/2022).

[2] 3dsourced. *How to Use 3D Print Infill Settings – Increase Strength, Save Filament*. URL: https://www.3dsourced.com/rigid-ink/3d-print-optimal-infill-settings/. (Accessed on 12/05/2022).

[3] AdditiveX. *What is Slicing Software, and what does it do?* 2021. URL: https://www.additive-x.com/blog/what-is-slicing-software-and-what-does-it-do/. (Accessed on 02/25/2022).

[4] Steve Addy. *Addy Machinery - Mazak Mazatrol 3D Assist Programming*. 2018. URL: https://www.youtube.com/watch?v=QgSk7vvemTs. (Accessed on 09/03/2022).

[5] Thomas Alsop. *Leading uses of 3D printing from 2015 to 2020*. 2020. URL: https://www.statista.com/statistics/560271/worldwide-survey-3d-printing-uses/. (Accessed on 02/25/2022).

[6] Thomas Alsop. *Which 3D printing technologies do you use?* 2021. URL: https://www.statista.com/statistics/560304/worldwide-survey-3d-printing-top-technologies/. (Accessed on 02/25/2022).

[7] Amazon. *Creality Ender 3 - Amazon*. URL: https://www.amazon.com/Official-Creality-3D-Precision-220x220x250mm/dp/B09GLN3WRN/ref=sr_1_28?crid=34JUCN6RA172K&keywords=3d+printer&qid=1646387528&s=industrial&sprefix=3d+print\%2Cindustrial\%2C153&sr=1-28. (Accessed on 03/04/2022).

[8] ANTLR. *ANTLR Lexer Rules documentation*. 2021. URL: https://github.com/antlr/antlr4/blob/master/doc/lexer-rules.md. (Accessed on 12/05/2022).

[9] ANTLR. *Interface ANTLRErrorListener*. URL: https://www.antlr.org/api/Java/org/antlr/v4/runtime/ANTLRErrorListener.html#syntaxError(org.antlr.v4.runtime.Recognizer,java.lang.Object,int,int,java.lang.String,org.antlr.v4.runtime.RecognitionException). (Accessed on 10/05/2022).

[10] ANTLR. *Left-recursive rules*. 2016. URL: https://github.com/antlr/antlr4/blob/master/doc/left-recursion.md. (Accessed on 10/05/2022).

[11] antlr.org. *ANTLR Development Tools*. URL: https://www.antlr.org/tools.html. (Accessed on 14/05/2022).

[12] Arctervex. *Advantages And Disadvantages of Using Computer Aided Design (CAD)*. URL: https://www.arcvertex.com/article/advantages-and-disadvantages-of-using-computer-aided-design-cad/. (Accessed on 02/21/2022).

[13] AutoDesk. *AutoCAD*. URL: https://www.autodesk.dk/products/autocad/. (Accessed on 02/21/2022).

[14] Vangie Beal. *High level programming language*. 2022. URL: https://www.webopedia.com/definitions/high-level-language/. (Accessed on 19/04/2022).

[15] Blender. *Hardware Requirements*. URL: https://www.blender.org/download/requirements/. (Accessed on 02/21/2022).

[16] Blender Foundation. *blender.org - Home of the Blender project - Free and Open 3D Creation Software*. URL: https://www.blender.org/. (Accessed on 02/21/2022).

[17] Lucas Carolo and David Pechter. *3D Printed Gears: How to Make Them | All3DP*. 2021. URL: https://all3dp.com/2/3d-printed-gears-get-the-gear-that-fits-your-needs/. (Accessed on 03/02/2022).

[18] Dibya Chakravorty. *3D Printing Supports – The Ultimate Guide*. 2021. URL: https://all3dp.com/1/3d-printing-support-structures/. (Accessed on 09/03/2022).

[19] Charles N. Fischer, Ron K. Cytron and Richard J. LeBlanc Jr. *Crafting A Compiler*. Addison-Wesley, 2009.

[20] Danmarks Statistik. *Elektronik i hjemmet 2018*. 2018. URL: https://www.dst.dk/Site/Dst/Udgivelser/nyt/GetPdf.aspx?cid=26813. (Accessed on 02/21/2022).

[21] Dragon 3D. *Make Print in Place Gears In Blender! // 3d Printing & Blender*. 2021. URL: https://www.youtube.com/watch?v=zxta2myEKnQ. (Accessed on 04/03/2022).

[22] Engineers Edge. *Design Equations and Formula Circular Pitches and Equivalent Diametral Pitches Table*. URL: https://www.engineersedge.com/gear_formula.htm. (Accessed on 14/04/2022).

[23] Formlabs. *Guide to 3D Printing Materials: Types, Applications, and Properties*. URL: https://formlabs.com/blog/3d-printing-materials/. (Accessed on 02/28/2022).

[24] GeeksforGeeks. *Difference between Compiler and Interpreter*. 2021. URL: https://www.geeksforgeeks.org/difference-between-compiler-and-interpreter/. (Accessed on 04/12/2022).

[25] Pranav Gharge. *Cura Tree Support: All You Need to Know*. 2021. URL: https://all3dp.com/2/cura-tree-support/. (Accessed on 09/03/2022).

[26] GitHub, Inc. *GitHub*. 2022. URL: https://github.com. (Accessed on 17/05/2022).

[27] Chua Hock-Chuan. *Regular Expressions (Regex)*. 2018. URL: https://www3.ntu.edu.sg/home/ehchua/programming/howto/Regexe.html. (Accessed on 15/05/2022).

[28] Home 3d Prints. *Making Replacement Gears – Can You Print On a 3d Printer?* URL: https://home3dprints.com/making-replacement-gears-can-you-print-on-a-3d-printer/. (Accessed on 21/02/2022).

[29] Hans Hüttel. *Transitions and Trees - An Introduction to Structural Operational Semantics*. Cambridge University Press, 2010.

[30] IBM. *What is software testing?* URL: https://www.ibm.com/topics/software-testing. (Accessed on 05/04/2022).

[31] International Institute for Sustainable Development. *UN Urges Tackling Waste Management on World Habitat Day*. 2018. URL: https://sdg.iisd.org/news/un-urges-tackling-waste-management-on-world-habitat-day/. (Accessed on 04/03/2022).

[32] Jetbrains. *Unit testing*. 2021. URL: https://www.jetbrains.com/resharper/features/unit_testing.html. (Accessed on 11/05/2022).

[33] Krishna Gali. *Module gear data*. 2014. URL: https://www.slideshare.net/krishnachaitanyagali/module-gear-data. (Accessed on 14/04/2022).

[34] ana Larry Bernstein. *What is Computer-Aided Design (CAD) and Why It's Important*. 2020. URL: https://www.procore.com/jobsite/what-is-computer-aided-design-cad-and-why-its-important/. (Accessed on 02/21/2022).

[35] Lucas Carolo and David Pechter. *3D Printed Gears: How to Make Them*. 2021. URL: https://all3dp.com/2/3d-printed-gears-get-the-gear-that-fits-your-needs/. (Accessed on 14/03/22).

[36] Saif M. *List of 24 Gear Terminology & Formula [Diagrams & PDF]*. URL: https://www.theengineerspost.com/gear-terminology/. (Accessed on 03/21/2022).

[37] MakerSpace. *MakerSpace 9220*. URL: https://makerspace9220.dk/. (Accessed on 02/21/2022).

[38] Marlin. *Marlin main page*. 2011. URL: https://marlinfw.org/. (Accessed on 10/03/2022).

[39] Mazak. *CNC Technology*. 2022. URL: https://www.mazakusa.com/machines/technology/cnc-controls/. (Accessed on 09/03/2022).

[40] Microsoft. *Unit test basics*. 2022. URL: https://docs.microsoft.com/en-us/visualstudio/test/unit-test-basics?view=vs-2022. (Accessed on 09/05/2022).

[41] Microsoft. *Use code coverage to determine how much code is being tested*. 2022. URL: https://docs.microsoft.com/en-us/visualstudio/test/using-code-coverage-to-determine-how-much-code-is-being-tested?view=vs-2022. (Accessed on 10/05/2022).

[42] Microsoft 365. *OneDrive Personal Cloud Storage*. URL: https://www.microsoft.com/da-dk/microsoft-365/onedrive/online-cloud-storage. (Accessed on 17/05/2022).

[43] Nicolai Buch Mogensen. *Guide: Sådan vælger du den rigtige computer*. 2022. URL: https://www.refurb.eu/da-dk/blog/post/guide-sadan-valger-du-den-rigtige-computer. (Accessed on 04/03/2022).

[44] Karim Nice. *How Gears Work | HowStuffWorks*. (Accessed on 03/07/2022). 2021. URL: https://science.howstuffworks.com/transport/engines-equipment/gear.htm.

[45] Overleaf. *Overleaf*. 2022. URL: https://www.overleaf.com. (Accessed on 17/05/2022).

[46] Sten Pittet. *An introduction to code coverage*. URL: https://www.atlassian.com/continuous-delivery/software-testing/code-coverage. (Accessed on 10/05/2022).

[47] ProductPlan. *MoSCoW Prioritization*. URL: https://www.productplan.com/glossary/moscow-prioritization/. (Accessed on 05/03/2022).

[48] Programming Lagnuages Laboratory, University of Calgary. *The Context Free Grammar Checker*. URL: https://smlweb.cpsc.ucalgary.ca. (Accessed on 09/05/2022).

[49] RepRap. *G-code*. 2022. URL: https://reprap.org/wiki/G-code. (Accessed on 03/02/2022).

[50] RepRap. *Mecode*. 2015. URL: https://reprap.org/wiki/Mecode. (Accessed on 09/03/2022).

[51] sablecc.org. *SableCC Documentation*. URL: https://sablecc.org/documentation.html. (Accessed on 14/05/2022).

[52] David Sauvage. *Should I test private methods directly?* 2021. URL: https://medium.com/decathlontechnology/should-i-test-private-methods-direclty-c48f4fa7bb4d. (Accessed on 11/05/2022).

[53] Robert W. Sebesta. *Concepts of Programming Languages*. Pearson, 2016.

[54] Taarnby Kommunebiblioteker. *Vil du printe noget i 3D?* 2021. URL: https://taarnbybib.dk/nyheder/biblioteket-informerer/vil-du-printe-noget-i-3d. (Accessed on 02/25/2022).

[55] TheDIYGuy999. *How To Repair Your RC Car Gears Using 3D Printed Gears: WLtoys 20402*. 2018. URL: https://www.youtube.com/watch?v=A2Jj3CZOkWk. (Accessed on 21/02/2022).

[56] Thingiverse. *Gears*. 2022. URL: https://www.thingiverse.com/search?q=gear&type=things&sort=newest&page=1. (Accessed on 21/02/2022).

[57] Gabriele Tomassetti. *The only good thing of 2020: ANTLR 4.9*. URL: https://tomassetti.me/the-only-good-thing-of-2020-antlr-4-9/. (Accessed on 10/05/2022).

[58] Ultimaker. *CuraEngine - GitHub*. URL: https://github.com/Ultimaker/CuraEngine. (Accessed on 03/25/2022).

[59] Ultimaker. *Ultimaker Cura*. URL: https://ultimaker.com/software/ultimaker-cura. (Accessed on 02/21/2022).

[60] Ultimaker Support. *Ultimaker support*. 2020. URL: https://support.ultimaker.com/hc/en-us/articles/360012733080-What-is-g-code-. (Accessed on 10/03/22).

[61] UN. *Responsible Consumption & Production - Why It Matters*. 2020. URL: https://www.un.org/sustainabledevelopment/wp-content/uploads/2019/07/12_Why-It-Matters-2020.pdf. (Accessed on 04/03/2022).

[62] Chris Woodford. *3D printers*. 2021. URL: https://www.explainthatstuff.com/how-3d-printers-work.html. (Accessed on 02/21/2022).

Appendix A

Example Programs

Listing A.1: Example program 1.

```
/*
multi-line comment
*/
PrinterModel = Ender3;
Type = SpurGear;

Var diameter = 50 - 2 * 0.2;
diameter = 20;

/* model */
InfillPercentage = 0.3;
OuterDiameter = diameter;
InternalDiameter = OuterDiameter - 10;
Var teeth = 10;
NumberOfTeeth = teeth + 5 * 2;
Height = 10.5;
```

Listing A.2: Example program 2.

```
PrinterModel = Ender5;
Type = SpurGear;
NumberOfTeeth = 12;
Height = 5;
OuterDiameter = 50.5 / 2;
InternalDiameter = 41.2 / 2;
InfillPercentage = 0.1;
```

Listing A.3: Example program 3.

```
PrinterModel = CR10;
Type = SpurGear;

NumberOfTeeth = 12;
InternalDiameter = 41.2 / 2;
InfillPercentage = 0.1;
Height = 500 - 450;
OuterDiameter = 50.5 / 2 * 10;
```

Listing A.4: Example program 4.

```
PrinterModel = Ultimaker2Plus;
Type = SpurGear;

InfillPercentage = 0.1;
NumberOfTeeth = 50;
InternalDiameter = (41.2 / 2) * 1 - 5; /*This is an arithmetic expression*/
Height = 500 - 450 + 300;
OuterDiameter = 50.5 / 2 * -1 * -1;
```

Listing A.5: Example program 5.

```
PrinterModel = Ender3;
Type = SpurGear;
/* this is a comment */

InfillPercentage = 0.1;
NumberOfTeeth = 100;
Var variable = 12;
InternalDiameter = 12 + variable;
Height = 10;
OuterDiameter = 50.5 / 2 * -1 * -1;
/* this is another comment*/
```

Appendix B

Context-Free Grammar Analysis

This appendix displays first the proposed grammar in simple BNF (Table B.1), as this provides a more suitable base for the identification of the FIRST, FOLLOW, and PREDICT sets of the grammar. The properties of these three sets are described in subsection 4.2.2. Secondly, the three sets are listed in Table B.2 for each non-terminal *A*, as the PREDICT sets reveal certain properties of the grammar. For simplicity in Table B.2, abbreviations have been used for long terminal and non-terminal names. These are written in square brackets after the name of the item in Table B.1.

A	Derivation	FIRST	λ	FOLLOW	PREDICT
P	PS GTS S eSy	pAK	No	–	pAK
PS	pAK aSy fpSt esSy	pAK	No	–	pAK
GTS	tAK aSy gTK esSy	tAK	No	–	tAK
S	Stmt esSy S	vdK, idSt	No	–	vdK, idSt
	COM S	csSy	No	–	csSy
	λ	–	Yes	eSy	eSy
COM	csSy string ceSy	csSy	No	–	csSy
Stmt	Dcl	vdK	No	–	vdK
	ASS	idSt	No	–	idSt
Dcl	vdK Id C	vdK	No	–	vdK
C	aSy Aexp	aSy	No	–	aSy
	λ	–	Yes	esSy	esSy
ASS	Id aSy Aexp	idSt	No	–	idSt
Aexp	Aexp pSy F	idSt, nSt, lpSy	No	–	idSt, nSt, lpSy
	Aexp mSy F	idSt, nSt, lpSy	No	–	idSt, nSt, lpSy
	F	idSt, nSt, lpSy	No	–	idSt, nSt, lpSy
F	F tSy P	idSt, nSt, lpSy	No	–	idSt, nSt, lpSy
	F dSy P	idSt, nSt, lpSy	No	–	idSt, nSt, lpSy

	F	idSt, nSt, lpSy	No	–	idSt, nSt, lpSy
P	Id	idSt	No	–	idSt
	N	nSt	No	–	nSt
	lpSy Aexp rpSy	lpSy	No	–	lpSy
N	nSt	nSt	No	–	nSt
Id	idSt	idSt	No	–	idSt

Table B.2: First, Follow, and Predict sets.

Program[P]	→	PrinterSelection[PS] GearTypeSelection[GTS] Stmts[S] eofSymbol[eSy]
PrinterSelection[PS]	→	printerAttributeKeyword[pAK] assignSymbol[aSy] printerNameString[pNS] endofstmtSymbol[esSy]
GearTypeSelection[GTS]	→	typeAttributeKeyword[tAK] assignSymbol[aSy] gearTypeKeyword[gTK] endofstmtSymbol
Stmts[S]	→	Stmt endofstmtSymbol[esSy] Stmts[S] | Comment[COM] Stmts[S] | λ
Comment[COM]	→	commentstartSymbol[csSy] string commentendSymbol[ceSy]
Stmt	→	Dcl | Assignment[ASS]
Dcl	→	vardclKeyword[vdK] Id C
C	→	assignSymbol[aSy] Aexp | λ
Assignment[ASS]	→	Id assignSymbol[aSy] Aexp
Aexp	→	Aexp plusSymbol[pSy] F | Aexp minusSymbol[mSy] F | F
F	→	F timesSymbol[tSy] P | F divideSymbol[dSy] P | P
P	→	Id | N | lparenSymbol[lpSy] Aexp rparenSymbol[rpSy]
N	→	numeralString[nSt]
Id	→	idString[idSt]

Table B.1: Context-Free Grammar written in simple BNF.

Appendix C

MMAG.Tokens

Listing C.1: Tokens and their corresponding integer value generated by ANTLR

```
EndofstmtSymbol=1
AssignSymbol=2
PlusSymbol=3
MinusSymbol=4
TimesSymbol=5
DivideSymbol=6
LparenSymbol=7
RparenSymbol=8
PrinterSettingsKeyword=9
VardclKeyword=10
TypeAttributeKeyword=11
GearTypeKeyword=12
Mycomment=13
WhiteSpace=14
NumeralString=15
IdString=16
PrinterString=17
';'=1
'='=2
'+'=3
'-'=4
'*'=5
'/'=6
'('=7
')'=8
'PrinterModel'=9
'Var'=10
'Type'=11
```

```
29 'SpurGear'=12
```