MuLE
A Multi-Paradigm Language for Education

Von der Universität Bayreuth
zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Abhandlung

von
Nikita Dümmel
aus Karaganda

1. Gutachter: Prof. Dr. Bernhard Westfechtel
2. Gutachter: Prof. Dr. Wolfram Haupt

Tag der Einreichung: 15. Dezember 2021
Tag des Kolloquiums: 23. Mai 2022

Acknowledgements

First and foremost, I want to thank my doctoral supervisor Prof. Dr. Bernhard Westfechtel for giving me this opportunity, mentoring me throughout the entire process and helping me make decisions which had a long lasting effect both on this work and on my personal attitude.

Speaking of opportunity, initially the work on this thesis was financed by the project *Qualitätsoffensive Lehrerbildung* of the *Federal Ministry of Education and Research* of Germany before I was employed directly by Prof. Dr. Westfechtel. This project has made it possible to start working on this thesis.

I would also like to thank Prof. Dr. Wolfram Haupt for his assessment of this thesis, and Dr. Matthias Ehmann, who has helped designing the implemented language and the programming course where it was used, as well as for the help with the subsequent statistical analysis of the gathered results. Additionally, my thanks goes to Dr. Michael Ebert who worked on another federal project called *EVELIN* and whose tool we have used in the initial iterations of our programming course.

Of course, I also want to thank my colleagues Dr. Thomas Buchmann, Monika Glaser, Sandra Greiner, Bernd Schlesier, and Johannes Schröpfer, as well as my former colleague Dr. Felix Schwägerl for helping me organize my work, providing helpful tips and technical support, finding an acronym for the implemented language which turned out to be more difficult than initially imagined, as well as for interesting discussions and conversations both related and unrelated to work.

My thanks goes also to the students, who have directly contributed to this work by providing implementation of certain modules or testing ideas, as well as all the students who have participated in the programming course and especially those who had the patience to give feedback.

And last but not least, I would like to thank all the people close to me for their support, but also for reminding me that there are other important things in life and shaping me into the person I am.

You have my sincerest gratitude!

Zusammenfassung

Über eine lange Zeit wurde in der Lehre die Programmiersprache Pascal eingesetzt, die speziell dafür konzipiert wurde. Sie wurde jedoch durch professionelle objektorientierte Sprachen wie Java abgelöst. Diese Entscheidung ist insofern verständlich, als Studierende in der beruflichen Praxis mit professionellen Sprachen konfrontiert sein werden. Die Komplexität dieser Sprachen kann jedoch für Progammieranfänger überwältigend sein. Daher wird im Kontext der Lehre für Programmieranfänger eine Sprache benötigt, die spezielle für diesen Zweck entworfen wurde.

Ein großer Teil dieser Arbeit besteht aus einer Analyse der Programmiersprachen, die aktuell in der Lehre eingesetzt werden. Dabei werden sowohl speziell für diesen Kontext, sowie für den industriellen Einsatz, entwickelte Sprachen betrachtet. Die gewonnenen Erkenntnisse werden dazu verwendet, die Anforderungen an eine neue Sprache für die Lehre zu formulieren, sowie diese Sprache zu entwerfen und zu implementieren. Die daraus entstandene multiparadigmatische Sprache für die Lehre MuLE (engl.: a *Multi-paradigm Language for Education*) unterstützt folgende Programmierparadigmen: prozedural, objektorientiert und funktional, wobei der Fokus der Sprache auf dem prozeduralen Programmierparadigma liegt. Die Sprache wird aktuell durch die Entwicklungsumgebung Eclipse und deren Werkzeugen unterstützt.

MuLE wurde im Rahmen eines Vorkurses für die Erstsemestervorlesung „Konzepte der Programmierung“, die auf Java basiert, an der Universität Bayreuth eingesetzt. Durch das Feedback der Studierenden wurde die Sprache und die darauf basierende Programmierumgebung kontinuierlich verbessert. Zur Evaluation wurden die Klausurergebnisse der Studierenden, die am Vorkurs teilgenommen bzw. nicht teilgenommen haben, miteinander verglichen. Abgesehen von der ersten Iteration des Kurses, waren die Leistungen der teilnehmenden Studierenden besser. Dies lässt vermuten, dass MuLE effektiv in der Programmierlehre eingesetzt werden kann.

Abstract

Since the emergence of object-oriented programming, the educational facilities have stopped using programming languages specifically designed for the purpose of education and are relying on professionally used languages instead. Even though it is understandable why this is the case, after all, sooner or later the students will be confronted with these languages, we are arguing that using complex professional languages in introductory programming education can be detrimental. These languages are designed with different priorities in mind than what is really important at the start of education. Furthermore, their complexity can act overwhelming for beginner programmers and the tendency towards implicit behaviour among some of these languages can be a cause for unnecessary misconceptions. Thus, it is our opinion that a language specifically designed with education in mind is required when teaching programming.

In this thesis, we analyse existing professional and educational languages and tools and use the gained information to summarize requirements, design, specify and implement an educational language that we have called MuLE – a *Multi-paradigm Language for Education*. As the name suggests, the language supports several programming paradigms, i.e. procedural, object-oriented and functional. We have chosen procedural programming as a platform upon which the other paradigms are implemented. The language is currently supported by the Eclipse IDE with its range of powerful tools.

MuLE was utilized for its intended purpose in a preliminary programming course, which is performed prior to the start of the CS1 lecture at the University of Bayreuth, where Java is used as the programming language. We have used the course to gather feedback from the target audience to improve the language. Additionally, we have compared the performance in the final exam of the CS1 lecture of those who have participated in the course to the performance of those who have not. Except for the very first iteration of the course, the performance of course participants was better than the performance of non-participants which leads to the assumption that MuLE can be effectively used to teach programming concepts to beginner students.

Table of Contents

1. Introduction

Learning and teaching programming is not an easy undertaking. These words, or at least semantically close variations, were often encountered when researching for this thesis, leading to the rising suspicion that it has become a sort of tradition to begin a paper, an article or a book focusing on programming education with these words. And why not, after all, during the learning process the students are overwhelmed by a set of new and unfamiliar concepts, conventions, algorithms and problems which have their roots in everyday life, however, follow slightly different rules and require them first to adapt to the new world of programming. Teaching these students is, as already said, not easy either. Every student is different and may show better results if a different approach is used. However, time constraints, the number of students, limitations of used tools and a plethora of various other reasons require the designer of a programming course to find an optimal solution, which would most likely not suit every student.

At the same time, the demand for programmers and software-engineers on the market is continuously rising. In the middle of the 20th century, computers were large machines located in specialized rooms and operated by teams of scientists used for scientific and military purposes. In the 1970s, many people working in the information industry assumed that at large, people are not and never will be interested in personal computers. However, by the turn of the millennium due to the triumphant success of the internet, emergence of new medial platforms and overall increasing reliance on computers by broad sections of the population and in the industry, these assumptions have been proved wrong.

Today, everyone who owns a smartphone carries a personal computer in her pocket, a very complex device which looks like something out of a science fiction movie by the standards of the previous century. Many people own a computer at home and have to operate one at work. Companies are continuously transferring to digitally assisted work to raise their productivity, starting with assembly line robots and diagnosis software and ending with computer assisted graphic design and accounting. Certain events, such as the ongoing Covid-19 pandemic, have forced the humanity to even further accelerate the process of digitalization.

1.1. Background and Motivation

Without software, a computer is just an electronic device lacking its intended practical functionality. Starting with the operating system, which keeps the device running, and ending with various applications, every piece of software must be implemented by a programmer. The ever increasing demand, rising complexity and continuous evolution of software systems create an ever present need for competent programmers, which in turn leads us back to the problem of their education.

The rising interest for computer scientists on the market has led to an increasing amount of students enrolling in the corresponding subjects. A large amount of them do this not out of a year long passion for computer science related disciplines and hobbies, but simply hoping for a better chance at future employment. Usually, these students have no prior programming experience and lack an intrinsic understanding of how algorithms work. Exactly these people are the target audience of this thesis. When starting to learn programming, students are often confronted by several unfamiliar concepts and problems at once. At the core of each program lies an algorithm, the logic of this program which has to be translated using a programming language into a piece of software, which will run on a computer. Depending on the chosen programming language and tools, the problems encountered by the students may range in their severity and include:

- Bridging the gap between *abstraction* and concrete implementation. Abstraction has multiple meanings in this case, i.e. problems are often formulated using natural language, which is abstract by its nature. Furthermore, abstraction also plays an integral role in computer science, be it different abstraction levels of programming languages, data abstraction, control abstraction, etc.

- *Problem solving*, i.e. understanding what has to be done and designing an appropriate algorithm. Studies [1][2] as well as own observations prove that problem solving is usually the more difficult task for beginners. Understanding how a loop works is less difficult than realizing that an iteration is required to solve a specific problem, which is usually just a small part of a much bigger problem.

- Application of the correct *language constructs and programming concepts* for the problem at hand, e.g. a loop or recursion if a certain process is repeated. This is aggravated when closely related yet still slightly different constructs can be used in certain cases.

- Knowing specific *hardware and software limitations* which may lead to unstable behaviour of the program. For example, languages without tail recursion optimization are less suitable for algorithms which rely on this feature and may lead to the stack overflowing at runtime.

- Knowing how to operate the *programming environment*. Although a simple text editor and a compiler are sufficient to start programming, it is arguably not the best choice of tools in teaching programming. An IDE with syntax highlighting and debugging support can be very helpful, however, more complex professional development environments may require more time for the students to get acquainted with.

- And, finally, how to use the *computer* itself as well as its *operating system*. Although basic computer skills are pretty much guaranteed today, as we have mentioned almost everyone owns some sort of a personal computer, same can not be said about the operating systems. The most commonly used OS in the computer labs and by the instructors of the University of Bayreuth is currently *Windows 10*. However, some students are using other operating systems such as *macOS*, which follows slightly different conventions regarding the display and the placement of specific menus, which has led to time consuming issues in several cases in the preliminary programming course, which we will discuss in chapter 9.

The less experience a student has in any or all of these fields, the more problems are encountered in the process which may accumulate to a really frustrating experience leading to students failing and quitting as a result. However, matters can be made even worse by using a complex programming language. Therefore, a comparably simple language must be used when teaching programming to absolute beginners. Simplicity does not, however, imply a lack of support for complex programming concepts, after all, this language is meant to prepare the students for their inevitable confrontation with a complex professionally used language.

1.2. A Multi-Paradigm Language for Education

Initially, the language MuLE was developed as a part of the *Qualitätsoffensive Lehrerbildung* [3] program financed by the *Federal Ministry of Education and Research* of Germany. The goal of this program is to improve the education of teacher students, thus the universities around Germany were given specific fields to focus on. Our topic was the knowledge heterogeneity and cultural diversity among the students. In the Computer Science department, we have chosen to focus on the knowledge heterogeneity, i.e. differences in the background knowledge of our first semester students. After less than a year, we have split from the federal project but have still continued working on MuLE. After all, the reasons why we have started working on it in the first place have not disappeared.

MuLE (*Multi-Paradigm Language* for *Education*)[1] is a multi-paradigm language that is designed to be both beginner friendly and at the same time as an effective educational tool. Finding the acronym was a lengthy task by itself, initially the language was called *Athena*, an arguably uninspired name with at least several other software systems sharing the same label. In its current state the language supports the three currently most important programming paradigms: procedural, object-oriented and functional.

To reduce the amount of concepts initially confronted by beginner students, MuLE provides a certain level of abstraction. Manual memory management is in our opinion not a required skill for beginner programmers and is rather an obstacle for beginner students at the initial state of programming education.

[1]http://www.ai1.uni-bayreuth.de/de/projects/MuLE/index.html

Therefore, MuLE offers implicit memory management and garbage collection. It provides a minimal set of orthogonal language constructs which are expressive enough to avoid obscure syntax. Speaking of which, MuLE offers a clear syntax which is easy to read, understand and learn. The orthogonality of provided language constructs should mitigate doubts at deciding which construct to use when confronted with a specific situation, i.e. by offering a single type of a loop the students will only have to decide between this loop and recursion instead of additional variations of a loop. Although not every problem which we have described in the previous subsection can be solved by such a language directly, using it alleviates some of these problems thus helping students to cope with other issues facilitating the entire process of learning how to program.

The language is designed in a way, that its constructs can be easily introduced incrementally in a programming course. It should be possible to use MuLE at both secondary and tertiary educational institutions. We have tested the language in a course designed around the "procedural-first" approach. Nevertheless, an "objects-first" approach should also be feasible.

1.3. Overview

We will start by analysing programming as a process itself as well as various programming paradigms in chapter 2. We will use various programming examples to demonstrate different approaches at implementations of the same algorithm according to the specific paradigms and the corresponding languages adhering to the these paradigms. Chapter 3 concentrates on the purpose of programming education and gives an overview over a selection of professional programming languages currently used in education as well as specifically designed educational languages and tools. Based on the knowledge gained from these chapters, the requirements that we have formulated for MuLE and the consequent design decisions are summarized in chapter 4. Subsequently, chapter 5 contains the specification of MuLE including the namespaces, its grammar, type system, definitions and semantics of its language constructs, etc. Chapter 6 gives a brief overview of the standard libraries distributed with MuLE and gives short examples of their usage. In its current implementation, MuLE is supported by the Eclipse IDE, the range of the offered tools is discussed in chapter 7. The implementation of MuLE itself as well as that of its tool support is presented in chapter 8, including a description of used tools. Finally, chapter 9 presents an evaluation of MuLE based mostly on its practical application in a non-mandatory preliminary programming course, the subsequent feedback by the students, as well as a comparison to other languages currently used in education. The appendix contains the installation instructions, APIs of the standard libraries, the complete Xtext grammar of the language as well as the implementation of the Universal Turing machine as a proof of Turing completeness of the language.

2. Programs and Programming Paradigms

This chapter is meant to clarify our understanding of programming in general, as well as look deeper into a selection of programming practices called paradigms. Therefore, in the first section we start by giving a brief explanation of algorithms and how they transfer into programs and general programming concepts. The subsequent sections focus on the specific programming paradigms, i.e. procedural, object-oriented, functional, and logic programming. We are using a dictionary algorithm as a running example over the course of this entire chapter.

2.1. From Algorithms to Programs

Every day people are confronted with problems they have to deal with. A lot of them are rather trivial and we are capable to solve them without really having to think about HOW we solve them. The problems in question are not the kind of finding a cure for a decease but, for example, how to tie shoes or cook a meal. Yet even these problems are not as trivial as we might think. As a child we have to learn not only how to tie shoes but also which shoe has to be put onto which foot. And our parents or guardians have to patiently teach us these procedures. When we have to write a text in a foreign language, we often have to look up words in a dictionary, a skill we usually acquire during our primary or secondary education.

The procedures that we use to solve specific problems are called *algorithms*. So what is an algorithm? It can be described as *a general approach to solve a group of related problems* [4]. Once we learn how to tie shoes, we can tie any existing shoe, it does not matter if those are sports or winter shoes. Since different non-related problems, e.g. cooking or looking up a word in a dictionary, require different sets of skills, there is no single algorithm to solve all problems. Technological evolution, while striving to make such problems easier to solve or offer new opportunities, brings a whole set of new problems which require new skills in order to be solved. As an example, we now have web based dictionaries in addition to classic ones in book format, meaning that different algorithms can exist to solve seemingly the same problem depending on available tools, skills, preferences, etc.

This leads us to computers, the tools that help us to solve immensely complex mathematical problems which, for example, allowed us to send humans into space. When we have to explain a solution of a problem to another human being we use natural language, hand gestures and additional resources like texts, images, models, etc. A computer is like a child, so screaming and hitting it will not lead to a positive outcome. Instead, we must use a *programming language* to explain how an algorithm works. An algorithm written down in a programming language is a *program*. Programs can be used to teach the computer how to interpret numbers, texts, images, models and all the other resources that a machine is capable to work with to provide the required results.

Today the programmers use higher level programming languages, which are easy to read (for a programmer) and are very expressive, but are not directly "spoken" by a computer. So first we actually have to teach it how to understand such languages. Depending on the architecture of the processor, each computer has its own machine language, which is not human readable as it consists of only ones and zeroes [5]. Above that are the assembly languages which are also dependent on the architecture, but are at least human readable. Assembly languages support basic operations, like storing a value in a register or adding two values, thus programming in such languages is quite tedious, even if sometimes necessary if the program has be optimised for a specific architecture. Therefore, most programmers mostly use higher level programming languages to write programs, which then have to be translated into machine code or other languages by specific programs called *compilers*. The other possibility is to execute whole programs or even separate statements and expressions in an *interpreter*. A hybrid solution may exist in case of some languages, where the program is compiled and executed on a virtual machine [6].

Now that we have created a rough overview over algorithms and programs, let us take a look at an example of paper based dictionaries. In the following scenario we formulate an algorithm on how to find a translation of a word. To achieve this we could follow these steps:

1. Look at the first letter of the word that needs to be translated.
2. Scroll through the pages to find the words starting with that letter.
3. Search though these words until the required word is found.
4. The result is the translation of this word.

This is a rather abstract definition of this algorithm, which is enough for a human being, but lacks depth to be used by a machine. Let us create a more in-depth version of the same algorithm:

1. The input is a word, that needs to be translated (we will call it the WORD).
2. Get the first letter of the WORD (the LETTER).
3. Navigate to the section of the dictionary containing words starting with that LETTER.
4. Get the first word which starts with the LETTER and compare it with our WORD.
5. If both words are equal, go to step 8.
6. If there is no next word starting with the LETTER, go to step 9.

7. Get the next word that starts with the LETTER, go to step 5.
8. The result is the translation of that word.
9. The result is the fact that the dictionary does not contain the WORD.

This is a more detailed algorithm and most steps can be easily translated into a program by using a higher level programming language. Step 3 requires us to navigate to a certain section in the dictionary but does not specify how. This action can be further described in detail as another algorithm with its own input and output. In fact, even such actions as *looking up the next word* and *comparing two words with each other* can be further specified as their own algorithms. If the programmer is using a lower level programming language, an in-depth definition of these actions is most certainly unavoidable. We could also remove step 3 entirely and just look at each word until we find the word that we are looking for. This would make our algorithm simpler, but also potentially require more time to get a result.

This brings us to the next topic: *abstraction.* This is a very important concept in computer science as it allows to simplify rather complex problems and lead to more elegant solutions. As already mentioned, computers “speak” machine language which is not easily understood by humans. Assembly languages require the programmers to explicitly describe each step. And while we need professionals who can work with such languages for specific tasks, the majority of programmers will work with higher level languages which allow to write very complex programs by using far less instructions compared to a lower level language, leading to a faster and less error prone development process. Such languages work at a higher level of abstraction, for example, most contemporary high level programming languages do not offer manual memory management mechanisms reducing the amount of work that has to be done by the programmer but also taking away some level of control. We will encounter other examples of abstraction, e.g. control abstraction and data abstraction [6], in programming languages in subsequent sections. In summary abstraction means simplification by reducing the amount of details.

Since the goal of this work is to implement a higher level educational language, we will not be further discussing low level languages. Instead we will analyse which program elements are unavoidable if we want to implement specific algorithms. This information will be imperative in chapter 5, where we will explain the design decisions behind MuLE.

Since programs are algorithms written down with a programming language, all such languages are required to support specific concepts. Writing a program is an evolutionary process that can be summarized into following steps [4]:

1. Formulate a functioning algorithm.
2. Refine the algorithm under consideration of the capacities of a specific machine.
3. Optimize the algorithm.

Usually, when we execute an algorithm, we have some resources that we want to work with and we expect some sort of result. In our dictionary lookup algorithm the starting resources are a word and the dictionary and the result is either the translation of the word or the knowledge, that this word is not included in that specific dictionary.

Thus, the very first important concept that we have to consider is the concept of data, i.e. the input of our program, any information required, modified or produced during the execution of the program as well as the output. In our dictionary lookup algorithm we have marked the words WORD and LETTER because these words represent memorized pieces of information which play an important role in our algorithm. Furthermore, the WORD is compared with other words, other pieces of information which are structured in a specific way, which allows us to navigate through them with relative ease. So not only do we need to understand what *data* is, we also need a way to *store and access* it.

So to define and use data, a programming language must support following concepts:

- *Data types* – specific operations may be performed only with specific data, for example we may divide one number by another but we cannot perform the same operation with two words. Data types allow us to distinguish between various kinds of information. Furthermore, since, in case of programming, we are working with hardware and its limitations, e.g. limited memory, data types have also the function of deciding the range of a value and therefore how much memory such a value will require when needed.
- *Data representation* – the machine representation of values is binary, which is not easily readable by humans. Thus, a high-level programming language must include a way to represent data in human readable form, e.g. decimal representation of numbers, words consisting of alphabet characters, etc.
- *Data storage and referencing mechanism* – in order to work with data, a language needs a mechanism to store and access it. Most high-level programming languages have automatic memory management, i.e. the user is not required to manually allocate enough memory for specific data before using it, and deallocate it to free up memory space after it is no longer needed. Data may be stored in different locations in memory, i.e. stack and heap, which vary in space and the speed of access. Programming languages rely on the concept of variables, which may represent a value (in

mathematical sense) or a named container which may contain a value or a reference to a specific value in memory, this depends on the paradigm of a specific language.

- *Data structures* – more often than not, information is stored in a specific way that allows quick access to data or a meaningful structure. In case of our dictionary example, rather than storing each word of a dictionary in its own variable, it is far simpler to store the collection of all words and access it via a single variable.

Such actions as comparing two words with each other and returning the corresponding translation are operations with data, i.e. we can evaluate and manipulate existing data or produce new pieces of information by executing such operations. These operations can be written down as elementary instructions. For example *get the first LETTER of a WORD* is such an instruction, which takes a string as a parameter and returns the first character which can then be stored in a variable.

Such instructions are executed in a specific sequence, usually one after the other. However, there are specific instructions which evaluate a condition and may skip some steps or return to a previous step. This allows to implement conditional branching and repetition contributing to the control flow of an algorithm or a program. For example step five of our algorithm evaluates the condition *two words must be equal*, and if this condition is true, jumps to step eight which ultimately leads to a successful termination of our algorithm. Step seven instructs the algorithm to jump to a previous step leading to a repetition. Modern high level languages lack a jump (`goto`) instruction for good reasons [6] and usually offer various language constructs which simulate the following three control structures, which are enough to solve any computable task [7]:

- *Sequence* – the steps in an algorithm and, therefore, the instructions in a program are executed one after the other, unless told otherwise by specific control structures.

- *Conditional branching* – depending on the condition, specific parts of a program may be executed while other parts may be skipped.

- *Repetition* – a set of instructions may be executed several (a potentially infinite number of) times, until a specific condition is met.

These concepts are more or less common amongst existing high level programming languages, nevertheless, these languages may exhibit significant differences in the implementation of these concepts. Different languages are built with different purpose and different programming paradigms in mind. A programming paradigm is: *"A way of approaching a programming problem. A way of restricting the solution set"* [8]. As a result, a single algorithm may be implemented differently with languages which adhere to different programming paradigms, however implementations of varying algorithms using the same paradigm will display many similarities in their general approach. In the following section we will see how our previously defined dictionary algorithm can be implemented in various programming languages which represent four different paradigms.

The first high-level programming languages that emerged in the 1950s were built around two different ideas of programming. Imperative programming relies upon state altering instructions that tell the machine *exactly how to proceed* whereas the declarative programming is more focused around telling the program *what the outcome should be.* Over time, due to constant evolution of hardware, change of standards and requirements, as well as the emergence of new challenges, these initial paradigms have evolved spawning new sub-paradigms and as a result thousands of new programming languages and dialects [5]. This is also one of the reasons, why there is no single universal programming language. The number of programming paradigms is a lot smaller, the authors of [9] mention 27 different paradigms which are used in praxis in their article. All of them offer some unique concepts but also have a lot in common, therefore, we will not discuss all of them but will focus on the four most important ones instead: *procedural*, *object-oriented*, *functional* and *logic* [10].

2.2. Procedural

As already mentioned, *imperative programming* was one of the two major approaches which emerged with the first high level programming languages in 1950s, i.e. FORTRAN [11], COBOL [12] and ALGOL [13]. These languages were developed with a specific purpose in mind, e.g. FORTRAN (Formula Translator) was meant to assist with complex numerical computations, which for example would occur in simulations of nuclear reactions, while COBOL (Common Business Oriented Language) was designed to deal with huge datasets that are prevalent in banking as well as big governmental and business organisations. Finally ALGOL (Algorithmic Language) was developed to represent algorithms as close as possible. This language was far less successful than the other two and was mostly used in the field of research and academia. Nevertheless, it was far ahead of its time, e.g. it included block delimiters, `for` and `while` loops, etc., and has greatly influenced the designers of both Pascal and C, two very successful procedural languages [5].

Concepts of Procedural Programming

The general idea of *imperative programming* (and therefore of *procedural programming*) is to work closely with the principles of the functionality of a computer, i.e. the von Neumann architecture, and not to be based on the mathematical model of computation. For instance, a variable in mathematics represents a value, which does not change during the process of computation. However, variables changing their values is a usual practice used in *imperative* and *procedural programming*. The general principle of this paradigm is the **change of state** controlled by basic instructions called **statements**.

In the previous section we have discussed concepts of data, operations with data and control flow. In our dictionary lookup algorithm we see several steps with jumps to other steps. This behaviour was initially supported in imperative languages (and is still available in assembly languages) by the `goto` statement, which allowed not only to jump to another statement in a subroutine (for example simulating a loop) but also to exit a subroutine and jump to an outer context. This was later criticized [14] which has led to *structured programming*, a sub-paradigm which evolved from *imperative programming*.

Structured languages include language constructs which offer almost the same level of control as the `goto` statement but also enforce a clear and **structured style** of writing a program. Such languages include loops as well as statements which allow to exit a loop (`exit`, `continue`) or a subroutine at any place (`return`). Some languages allow to *throw exceptions* which is another way to modify control flow.

The **subroutines**, that we have already mentioned several times, represent the core concept of *procedural programming*, another sub-paradigm of *imperative programming*. The step three of our dictionary lookup algorithm tells us to navigate to a specific section in a dictionary which contains all words beginning with a specific letter. This by itself is a complex operation which requires an input and produces some sort of result. Instead of specifying each step in that sub-algorithm we have simply invoked it implying that it is already implemented somewhere else. Such operations are called subroutines and act as *control abstraction* [6]. The caller may not even be aware of the implementation details of a subroutine, which is usually the case when we use subroutines provided by a library. Here we see another advantage of subroutines, once implemented, they can be distributed via libraries to be reused, greatly reducing the amount of effort put into implementing new complex software.

Subroutines are usually parameterized, i.e. they can accept value parameters which alter the behaviour of the subroutine. The previously mentioned sub-algorithm to navigate to a specific section in a dictionary may, for example, accept the fist letter of a word and the dictionary (or alternatively a pointer or a reference to it) as a parameter. Subroutines that return a value are called *functions*, e.g. our sub-algorithm could return a subset of the dictionary with only the specific words or a pointer to the first word in the required section. A *procedure* is a subroutine which does not return a value, e.g. if our sub-algorithm accepts a pointer to the first word in the dictionary, this pointer could be set to the first word in the specific section of the dictionary, thus altering the state of the pointer in the outer context without actually returning a value. These subroutines are eponymous to the whole *procedural paradigm*.

Example of a Procedural Program

Now let us finally translate our algorithm into a procedural program. For this task we will use the programming language Pascal [15], which was developed as an evolution of ALGOL and was used as the primary educational programming language around the world until the rise of object-oriented programming. The source code of the program is given in listings 1, 2 and 3.

Before we can start implementing our dictionary lookup algorithm, we must first define what a dictionary is. Real world dictionaries are collections of words which are arranged alphabetically and the purpose of a dictionary is, for example, to give a definition or a translation of a word. Therefore, we define a type `Dictionary` as a record – a composite type – which contains the purpose (simply as a string) and an array of `Sections`. A `Section` is a also a record, which contains a string (which should be the first letter of contained word, sadly our implementation lacks any validation mechanisms) and an array of `WordPairs`, again a record consisting of two strings: a word and its translation. Then we declare the global variables for our dictionary and three sections, this implementation is just an example and is not meant to be used as a fully functional dictionary program. These global variables are exemplary for the *imperative style of programming*, they represent the state of our dictionary which is shared between all subroutines as well as the main body of the program. A change performed in one subroutine on such variables will have side effects on the entire program.

```
program DictionaryProceduralExample;
uses sysutils;

type WordPair = record
   word : string;
   translation : string;
end;

type Section = record
   letter : string;
   words : array [1..100] of WordPair;
end;

type Dictionary = record
   purpose : string;
   sections : array [1..100] of Section;
end;

var
   sectionA, sectionB, sectionC : Section;
   dict : Dictionary;
```

Listing 1: Declarations of types and global variables in our procedural program.

Now that we have mentioned them, let us talk about subroutines of our program (listing 2). These subroutines represent the actual implementation of the behaviour defined by our dictionary lookup algorithm.

The procedure `lookupTranslation` accepts a formal string parameter named `word`, which is mapped to an actual parameter when this procedure is called (examples are given in listing 3). Afterwards, we declare a number of local variables which we will use in our procedure, such as the first letter of our word or the section in which we will look for our word. The keyword `begin` marks the beginning of the body of our subroutine, in Pascal the keyword pair `begin-end` is generally used to denote blocks of statements, a central characteristic of *structured programming*.

```
function getTargetSection(letter : string) : Section;
var
   counter : integer;
begin
   for counter := 1 to 100 do
      if AnsiCompareText(dict.sections[counter].letter, letter) = 0 then
      begin
         getTargetSection := dict.sections[counter];
         exit;
      end;
end;

procedure lookupTranslation(word : string);
var
   lookupSection : Section;
   letter : string;
   counter : integer;
begin
   letter := LeftStr(word, 1);
   lookupSection := getTargetSection(letter);
   matchFound := false;
   for counter := 1 to 100 do
   begin
      if AnsiCompareText (lookupSection.words[counter].word, word) = 0 then
      begin
         writeln (word + ' : ' + lookupSection.words[counter].translation);
         exit;
      end;
   end;
   writeln (word + ' : no matches found');
end;
```

Listing 2: Subroutines implementing the dictionary lookup algorithm.

The body of the procedure contains the actual functionality, i.e. the implementation of our algorithm. Here is a recollection of the algorithm as well as the explanation how it was implemented:

1. **The input is a word, that needs to be translated (we will call it the WORD)** – this is represented by the string parameter `word` in the procedure `lookupTranslation`.

2. **Get the first letter of the word (the LETTER)** – first we need a variable where we store the LETTER, the variable `letter` is declared in the variable declaration section of the procedure. In order to obtain the first letter of our word, we call the function `LeftStr` from the built in library `sysutils` for this task. The returned value, i.e. the first letter of our word, is then assigned to the variable `letter`.

3. **Navigate to the section of the dictionary containing words starting with that LETTER** – here we invoke the other subroutine that we have implemented in this program: the function `getTargetSection` which accepts a string parameter `letter` and returns a value of type `Section`. Since our dictionary is a collection of sections, we simply iterate with a *for loop* over this collection until we find the section with the respective letter. In that case, the section is assigned as the result of our function, meaning that it will be returned after the execution, which is terminated with the `exit` statement. Pascal lacks an explicit `return` statement.

4. **Get the first word which starts with the LETTER and compare it with our WORD** – this step as well as the steps six and seven are connected with each other and are represented by a *for loop* in our implementation. The loop is repeated the same number of times as the maximal amount of words in a section.

5. **If both words are equal, go to step 8** – we use another library subroutine `AnsiCompareText`, which returns a 0 if two strings are equal (case insensitive), to compare the parameter `word` with the current `word` in the section. This step is represented by the head of the `if`-statement. The body of the `if`-statement, which is executed if the condition is true, represents the step eight.

6. **If there is no next word starting with the LETTER, go to step 9** – the loop has run to its end.

7. **Get the next word that starts with the LETTER, go to step 5** – the loop is not yet finished.

8. **The result is the translation of that word** – this step is executed if the condition in the step five is true. This is represented by the body of the `if`-statement. Here we print the word as well as its translation and exit the procedure.

9. **The result is the fact that the dictionary does not contain the WORD** – if the loop ran all the way through without terminating the procedure, meaning that the word we are looking for is not in the section, the message *no matches found* is printed next to that word on the console and the procedure is terminated normally since there are no other statements.

This is only one of the many possible implementations of our dictionary lookup algorithm. For example, we could have used a linked list instead of an array as our data structure for storing word pairs and sections. This would require the use of pointers and `while`-loops instead of `for`-loops and it would make the implementation more dynamic.

```
begin
    sectionA.letter := 'A'; sectionB.letter := 'B'; sectionC.letter := 'C';

    sectionA.words[1].word := 'Alphabet';   sectionA.words[1].translation := 'alphabet';
    sectionA.words[2].word := 'Anweisung';  sectionA.words[2].translation := 'statement';
    sectionA.words[3].word := 'Ausdruck';   sectionA.words[3].translation := 'expression';
    sectionB.words[1].word := 'bauen';      sectionB.words[1].translation := 'build';
    sectionB.words[2].word := 'Baum';       sectionB.words[2].translation := 'tree';
    sectionB.words[3].word := 'Bedingung';  sectionB.words[3].translation := 'condition';
    sectionC.words[1].word := 'Chip';       sectionC.words[1].translation := 'chip';
    sectionC.words[2].word := 'Code';       sectionC.words[2].translation := 'code';
    sectionC.words[3].word := 'Computer';   sectionC.words[3].translation := 'computer';

    dict.purpose := 'German - English';
    dict.sections[1] := sectionA;
    dict.sections[2] := sectionB;
    dict.sections[3] := sectionC;

    lookupTranslation('Alphabet');
    lookupTranslation('Anweisung');
    lookupTranslation('Ausdruck');
    lookupTranslation('bauen');
    lookupTranslation('Baum');
    lookupTranslation('Bedingung');
    lookupTranslation('Chip');
    lookupTranslation('Code');
    lookupTranslation('Computer');

    lookupTranslation('Algorithmus');
    lookupTranslation('Programm');
    lookupTranslation('Sprache');
end.
```

Listing 3: Main body of our procedural program.

Finally, listing 3 shows the main block of our program. This is the part of every Pascal program (and many other languages, such as C or Java), which is executed first, meaning that this is where we are initially calling our subroutines in order to get the desired results. But first, we have to initialize our dictionary. Since it consists of sections, we need to initialize them too. Thus, we fill them with a couple of words, assign them their respective letters, and then fill our dictionary with the newly initialized sections.

Since we now have a dictionary that we can use for a test case, we start calling our `lookupTranslation` procedure. We use both words that are contained in our dictionary and words that are missing in order to test both successful and failed runs of our program.

The produced output is as follows:

```
Alphabet : alphabet
Anweisung : statement
Ausdruck : expression
bauen : build
Baum : tree
Bedingung : condition
Chip : chip
Code : code
Computer : computer
Algorithmus : no matches found
Programm : no matches found
Sprache : no matches found
```

Summary

Based on the structure of the Pascal program above, we can give a rough example of a procedural program structure:

1. Program name.
2. Import or include instructions.
3. Type declarations, which may include records, enumerations (`type RGB = (red, green, blue)`), range types (`type Number = 1 ... 100`) and user defined basic types (`type Age = integer`). Other languages may have other variants of type declarations.
4. Declarations of constants and global variables.
5. Subroutine declarations.
6. Main block.

Let us finally summarize the hallmarks of procedural programming, including those inherited from structured and imperative programming, as well as language constructs (not based on any specific language) that support them:

- A **statement** is the most basic language construct which may cause side effects. The *assignment statement* is the most prominent example here.
- Statements are **structured in blocks**, which may be parts of other statements, subroutines and other language constructs. Blocks are delimited by pairs of keywords (ALGOL, Pascal, Ada), brackets (C) or indentations (Python). Blocks can be nested, e.g. a subroutine may include loops and conditional statements. Blocks also represent namespaces for identifiers, more details in section 5.1.
- **Subroutines** are essentially blocks of statements with an identifier, which can be used to call the subroutine from different places in a program. Subroutines may accept parameters and may return values.
- **Control structures** decide in which sequence the statements are executed. We differentiate between three types of control flow:
  1. *Sequence* – is the basic way of executing statements if no other control structure is involved.
  2. *Conditional branching* – blocks of statements may be skipped or executed only if specific conditions are met. Examples of statements that support such behaviour are `if ... then ... else` and `switch ... case` statements.
  3. *Repetition* – allows to execute blocks of statements several times. We differentiate between infinite (`while ... do`, `do ... while`) and finite (`for ... do`, `foreach ... do`) repetition. Another way to implement repetition is via *recursion*, in which case a subroutine calls itself.
- The behaviour of **control flow may be further altered** by specific statements, that may terminate the entire loop (`break`), terminate the current iteration of the loop and begin instantly with the next iteration (`continue`) and terminate the execution of a subroutine (`return`, `exit` in Pascal).
- Procedural languages offer `records` or `structs` as a mechanism to define **composite datatypes**, which may combine data items with different types.
- Furthermore, such languages usually include **type homogeneous data structures** like arrays and sets.
- Dynamic and recursive data structures and relations can be implemented with **pointers or references**. Pointer variables store an adress to a location in the memory instead of a value.

2.3. Object-Oriented

In the previous section we have discussed how *procedural programming* has evolved from *imperative* by introducing subroutines, which are essentially meant to perform actions on data, i.e. subroutines are the active element in a procedural program while data plays a more passive role. Now what if those roles were reversed? What if the data would perform the active role and would provide its own behaviour? In *object-oriented programming* this role is taken over by the objects. Moreover, in pure object-oriented languages everything is an object, even elementary values such as integer numbers [16].

SIMULA (Simulation Language) was developed as an extension of ALGOL, a procedural language we have mentioned in the previous section. The first version of this language, SIMULA 1 [17], was originally intended to simulate quasi-parallel execution of several processes contained in a single activity. A process can be a full fledged ALGOL program or merely include passive data structures [18]. Each process is given a local time variable which simulates the passing of time when the processes are executed. It was, however, the next iteration of this language called SIMULA 67 [19] which marked the birth of object-oriented programming by introducing many of the concepts we know today, such as *objects*, *classes*, *inheritance*, *virtual procedures* (i.e. abstract methods) and a safe referencing mechanism.

Whereas SIMULA has evolved from ALGOL and thus contains both object-oriented and procedural concepts, Smalltalk [20] was developed to be a pure object-oriented language. As already mentioned, everything is an object in such a language. A programmer can send *messages* to objects in order to achieve some sort of behaviour. Let us take a look at the following example:

```
(x>0) ifTrue:[ x:=x+1. ] ifFalse:[ x:=0 ].
```

This line of code is Smalltalk's way of writing an `if`-statement. Here we send the messages `ifTrue` and `ifFalse`, each with a corresponding code block, to the object `(x>0)`. Unlike in procedural languages, where an `if ... then ... else` statement consists of keywords which are part of the grammar of the language, `ifTrue` and `ifFalse` are actions, that can be executed by Boolean objects. Such actions, which can be performed by objects and are therefore defined in a class, are called *methods* in more modern object-oriented languages like C++ and Java.

These two languages are not considered as pure object-oriented languages. C++ [21] was developed as an evolution of C, a procedural language, and thus includes most of its language constructs and allows to program in an entirely procedural way. Java [22], although heavily inspired by the syntax of C, is however not related to this language. Unlike Smalltalk, both languages offer statements as the most basic independent unit in a program, similar to procedural languages. In contrast to C++, it is not possible to program in a purely procedural way in Java. Many modern languages support object-oriented programming due to the fact that this paradigm is very efficient at modelling complex systems leading to implementations which are easer to understand and maintain.

Concepts of Object-Oriented Programming

The following three concepts are integral for the object-oriented programming paradigm [23][6].

As previously mentioned, unlike in procedural programming, data plays a more active role in object-oriented programming by providing its own behaviour. We no longer have separate subroutines that perform actions with chunks of data that are defined as global variables or were passed to them as parameters. Now our data objects can actively execute such operations by themselves, potentially changing their internal state in the process. To achieve this, the declaration of the type and the operations on the instances of that type should be contained in the same syntactic unit. Thus the first major concept of object-oriented programming is **encapsulation**, i.e. grouping of data and operations in a single syntactic unit. This leads to the emergence of the *class* concept, a blueprint for an *object* which defines the attributes and functionality of that object and acts as a type. The data and subroutines are often called *fields* or *data members* and *methods* or *member functions* respectively. Some languages like Java allow to define nested types, i.e. classes within classes, further encapsulating and hiding implementation details, that are not relevant outside of a specific class.

An abstraction of various entities is a generalized view with the most significant attributes of these entities. Thus, in regards to object-oriented programming we often talk about **data abstraction** and abstract data types. An abstract data type should allow the possibility to declare variables with that type but hide the underlying implementation details. In fact, an abstract data type may completely lack any implementation details and serve only as a sort of a blueprint for inheriting non-abstract types enforcing them to implement specific behaviour. This brings us to the concept of **inheritance** which allows to define subtypes of an existing type. A subtype inherits the implementation details of its super type and can add new features as well as redefine the inherited operations. An instance of a subtype is automatically an instance of its super types. A super type may be either abstract or non-abstract, both kinds of types can be used to declare variables. However, only non-abstract types may be used to instantiate objects since abstract types may lack implementation details.

There are two distinct approaches to inheritance: single and multiple inheritance. Single inheritance is less complex and easier to explain. It is however more restricted compared to multiple inheritance since it always results in a tree like hierarchy. Therefore it is not possible to implement a class inheriting the functionality of two independent classes without introducing some redundancy to the code by using single inheritance. Multiple inheritance, on the other hand, allows to design more natural solutions. However, it comes with a range of problems which require their own workarounds [24].

While most object-oriented languages offer abstract classes, some languages like Java or C# [25] include an additional construct called *interface*. Both constructs represent abstract types, i.e. you may declare a variable of such types, but you have to use an inheriting non-abstract type to instantiate that variable. The difference lies in their intended goal, i.e. an *abstract class* is a generalization of several classes while an *interface* can provide several super types which may act as interfaces to a single system for various clients with different access rules. Additionally interfaces offer a way to implement multiple inheritance in Java, albeit somewhat restricted. Some languages like Java allow only single inheritance on classes, while others, for example C++, allow multiple inheritance and implement specific mechanisms to counter the issues that come with it.

Other than by using abstract types to hide implementation details, many languages offer additional mechanisms for information hiding such as **visibility modifiers**. For example, there is no need for abstract types for a specific class, yet it still includes some internal operations which should not be called from outside. Such methods could be hidden by setting their visibility to `private` or other alternatives offered by a specific language. Fields are usually declared as `private` and specific access methods are implemented if needed.

Having several related yet still different types in a single inheritance hierarchy, and especially abstract types which can be used in variable declarations but not for initialization of the same variables, means that we now have variables which may have two different types at once as long as we are talking about languages with static typing. The static type is given at the time of the declaration and is used for compile time checks while the dynamic type is the one with which the variable is initialized. This way we may have collections of related objects with different implementations of the same method. For example if a class `Car` redefines a method `drive()` which was inherited from the class `Wheeled`, instances of `Car` will execute the redefining method regardless of the static type of the variable. The concept of **dynamic binding** is responsible for calls of the corresponding methods on objects at runtime depending on its dynamic type.

Another important aspect of object-oriented programming is the referencing mechanism. While an object is technically a chunk of data in the memory and can therefore be considered a value of a variable, semantically it makes no sense to use objects as values the same way they are used in procedural programming, e.g. by copying the value in assignments or `return` statements. Therefore most of the object-oriented languages, that were originally designed as object-oriented languages (unlike C++), offer either only reference types (e.g. Python) or a mix of value and reference types, whereby value semantics are reserved for primitive types only (e.g. Java).

Example of an Object-Oriented Program

In the following example we are going to implement the dictionary lookup algorithm in an object-oriented way by using Java. Unlike in the procedural implementation of the same algorithm in the previous section, in this example the implementation is separated into three compilation units: the interface (listing 4), its implementation (listing 5) and the test file (listing 6).

```java
package dictionary;

public interface Dictionary {
   String getPurpose();
   void lookupTranslation(String word);
}
```

Listing 4: The interface to our implementation.

First we have to specify what our dictionary does, i.e. we can look up the translation of a word and we can ask our dictionary what its purpose is. We use an interface for this task, which is one of the two ways to define abstract types in Java, the other would be by using abstract classes. The interface in question is called `Dictionary` and is displayed in listing 4. It includes merely the method heads for our operations that we intend to perform on our dictionary instances as a user. The methods are not implemented, in fact Java interfaces were not meant to provide any implemented methods until Java 8 was released. This is not a problem, even though we can declare a `Dictionary` variable we are not allowed to instantiate a `Dictionary` object. To be able to do this, we must first implement this interface by a non-abstract class which will force us to implement these methods.

The class `DictionaryImpl` in listing 5 implements our `Dictionary` interface. `DictionaryImpl` is a non-abstract subtype of `Dictionary`, which means that we can now declare `Dictionary` variables and initialize them as `DictionaryImpl` objects. Specific design patterns like the factory pattern [26] rely on this behaviour by internally creating objects and returning them only by their interface type effectively hiding any internal details of these objects from the user.

```java
package dictionary;

import java.util.LinkedList;
import java.util.List;

public class DictionaryImpl implements Dictionary {
   private class WordPair {
      public String word;
      public String translation;
   }

```

```java
    private class Section {
        public char letter;
        public List<WordPair> wordPairs = new LinkedList<WordPair>();
    }

    private String purpose;

    private List<Section> sections = new LinkedList<DictionaryImpl.Section>();
    public DictionaryImpl(String purpose, String[][] wordPairs) {
        this.purpose = purpose;
        for (String[] wordPair : wordPairs) {
            WordPair wp = new WordPair();
            wp.word = wordPair[0];
            wp.translation = wordPair[1];
            char letter = wp.word.charAt(0);
            Section section = getTargetSection(letter);
            if (section == null) {
                section = new Section();
                section.letter = letter;
                section.wordPairs.add(wp);
                sections.add(section);
            } else {
                section.wordPairs.add(wp);
            }
        }
    }

    private Section getTargetSection(char letter) {
        for (Section s : sections)
            if (s.letter == letter)
                return s;
        return null;
    }

    @Override
    public void lookupTranslation(String word) {
        char letter = word.charAt(0);
        Section section = getTargetSection(letter);
        if (section != null)
            for (WordPair wp : section.wordPairs)
                if (wp.word.equals(word)) {
                    System.out.println(word + " : " + wp.translation);
                    return;
                }
        System.out.println(word + " : no matches found");
    }

    @Override
    public String getPurpose() {
        return purpose;
    }
}
```

Listing 5: The object oriented implementation of the dictionary lookup algorithm.

Our implementation is mostly similar to the procedural example in the previous section, however there are some key differences. First and foremost, the types `WordPair` and `Section` are no longer separate from our `Dictionary`, but are now contained within our `DictionaryImpl` class and are both set as `private`. These types represent data structures which are relevant for the internal functioning of our dictionary and are not of any interest for the user.

Our dictionaries have a `purpose` and are implemented as a list of `sections`, both are private fields in our class. After the field declarations we see the constructor of the class, a specific language construct which is meant to initialize the fields of our objects. Constructors are called once when we instantiate an object basically providing means to initialize private fields, which are meant to be initialized once and it is otherwise not possible to change their state from the outside. Therefore, the constructor is set to public, otherwise we could not access it and it would not be possible to create instances of this class. The constructor relies on the method `getTargetSection(char letter)`, which is not relevant to the user and is therefore set to `private` in this example .

Finally, we have two `public` methods `lookupTranslation(String word)` and `getPurpose()` which provide implementation to each inherited method from the interface and therefore must be declared as `public`. The latter method basically provides read only access to the field `purpose` of our dictionary since Java Strings are immutable [27].

```java
package dictionary;

public class Main {
    public static void main(String[] args) {
        Dictionary dict = new DictionaryImpl(
            "German - English", new String[][] {
                {"Alphabet", "alphabet"},
                {"Anweisung", "statement"},
                {"Ausdruck", "expression"},
                {"bauen", "build"},
                {"Baum", "tree"},
                {"Bedingung", "condition"},
                {"Chip", "chip"},
                {"Code", "code"},
                {"Computer", "computer"}
            }
        );
```

```
            dict.lookupTranslation("Alphabet");
            dict.lookupTranslation("Anweisung");
            dict.lookupTranslation("Ausdruck");
            dict.lookupTranslation("bauen");
            dict.lookupTranslation("Baum");
            dict.lookupTranslation("Bedingung");
            dict.lookupTranslation("Chip");
            dict.lookupTranslation("Code");
            dict.lookupTranslation("Computer");
            dict.lookupTranslation("Algorithmus");
            dict.lookupTranslation("Programm");
            dict.lookupTranslation("Sprache");
        }
    }
```

Listing 6: The main class, which tests the implementation of the algorithm.

Let us finally take a look at the `Main` class in listing 6. This class contains the `main` method which has similar function to the main body of a Pascal program in listing 3, i.e. it is the first operation that is executed when we run our program, therefore, this is where we initialize and test our dictionary.

The dictionary variable is declared as `Dictionary` and initialized as `DictionaryImpl` with the corresponding parameters in the constructor call. Unlike in our procedural example, we can no longer change the internal state of our dictionary variable. We can neither add new sections or word pairs, nor can we change the values of the existing ones. We are only able to access the methods which are defined in our interface `Dictionary`.

Summary

So let us give a compact overview of the important features of object-oriented programming:

- The state is carried by objects, i.e. an **object saves information** similar to a record, however **it can also execute actions** called methods, i.e. subroutines reserved to specific objects. Most contemporary object-oriented languages rely on statements to implement behaviour of the methods.
- **Classes** represent non-primitive data types that define objects.
- Many object-oriented languages support primitive data types common to procedural and functional languages.
- **Abstraction and information hiding** are important, implementation details are often hidden by specific language constructs.

- **Inheritance** allows to specify related classes, which may still show differences among each other. Commonalities are specified in super types while differences are kept within each specific subtype.
- Some languages support multiple inheritance, while other offer only single inheritance. There are also languages that offer a mix of both versions, e.g. Java with single inheritance on classes and a possibility to implement multiple interfaces.
- **Abstract types** allow to specify abstract methods enforcing non-abstract subtypes to implement these methods. Such types are not instantiable.
- **Referencing mechanisms** are used to reference objects since value copy semantics of procedural languages make no sense in object-oriented context.

Therefore, a program written with this style would usually consist of class declarations, which reference each other and contain all the information about the functionality of the program. The internal structure of a class is quite similar to that of a procedural or a functional program, i.e. it may include nested type declarations, global variables (fields) and subroutines (methods).

2.4. Functional

Previously we have discussed the *procedural* and *object-oriented programming* paradigms and their evolution based on *imperative* way of thinking. The main idea of imperative programming is to tell the machine what to do exactly by using statements, i.e. state altering instructions. The alternative to this kind of programming is represented by the declarative programming paradigms, e.g. *functional* and *logic*, wherein instead of instructing the machine what to do step by step, the programmer writes down what the result of a program should be and most of the heavy lifting is then taken over by the implementation of the chosen programming language (this is especially apparent in logic programming with Prolog). Compared to imperative languages, such languages work at a higher level of abstraction and the programs tend to be shorter and more elegant.

In this section we will focus on *functional* programming, which has emerged in parallel to *imperative* programming in the late 1950s. The first functional programming language was Lisp (List processing) [28] which originally was specified as a mathematical notation for computer programs based on the lambda calculus by Alonso Church [29]. Many dialects of Lisp were developed since its introduction, some Lisp dialects like Common Lisp [30], Scheme [31] and Clojure [32] are still used today both in education and in industry. According to [30] Common Lisp is one of two most used programming languages in AI development (the other one being Prolog, more in the next section) due to its flexibility and powerful libraries. Meanwhile Scheme, which offers only a minimal set of powerful features, is rather used in education. Functional programming languages offer elegant solutions to deal with large data sets [33]. In general, functional programs tend to be shorter and more elegant compared to programs written in an imperative way and are therefore easier to read and maintain. Furthermore, known mathematical definitions can be more easily translated into a program by using a functional language, instead of requiring to find an algorithmic solution when using an imperative one, thus reducing the amount of work that has to be done writing and verifying a program in such a case.

Before we dive into details, let us first outline what contributes to the functional programming paradigm. According to [34] *"Functional programming is a style of programming that emphasizes the evaluation of expressions, rather than execution of commands. The expressions in these languages are formed by using functions to combine basic values. A functional language is a language that supports and encourages programming in a functional style."* The authors continue by giving an example calculating the sum of integers from 1 to 10 written in an imperative way:

```
total = 0;
for (i=1; i<=10; ++i)
    total += i;
```

In this example we have a variable `total` intended to store the result. The result is calculated via a `for` loop, which iterates over the range of integers using an internal variable whose value is added to the variable `total`. When programming in an imperative style, we are using statements to change the state of our variables.

The authors follow up with an example of the same program written with Haskell [35], a pure functional programming language.

```
sum [1..10]
```

In this example the expression `[1..10]` is evaluated to a list of integers defined by the given range, and `sum` is a library function which adds all values of a list and returns the result. However, not every functional language offers such extensive libraries as Haskell or language constructs to define lists by integer ranges. Here is the same example written with Scheme, a minimalistic dialect of Lisp:

```
(define sum
    (lambda (from total)
        (if (= 0 from)
            total
            (sum (- from 1) (+ total from)))))
(sum 10 0)
```

In this example, we define a function `sum` as a lambda expression which accepts two parameters: `from` and `total`. The function returns the value of `total` if `from` equals 0, otherwise the function is called recursively with a value of `from` reduced by 1 and the value of `total` aggregated by the current value of `from`. The values of the arguments are not changed during the execution, however, each recursive call is initiated with new values until the required conditions are met. The whole program is a list of computations which is ultimately evaluated to a single value. Such behaviour is typical for functional programming, but is not exclusive to functional languages since it can be reproduced in procedural and object-oriented languages.

Concepts of Functional Programming

We have already mentioned the lambda calculus and the lambda expressions but before we discuss them, let us first take a look at functions. *"A function is a rule of correspondence by which when anything is given (as argument) another thing (the value of the function for that argument) may be obtained"* [29]. It is further specified that a function may not necessarily be applied to any kinds of arguments, i.e. in case of programming a specific function may be applied to specific types and their subtypes. A function is basically a subroutine, which accepts one or more arguments and returns a value depending on those arguments. Since functional programming languages strive to be **free of any side effects**, meaning that the use of state altering constructs like mutable variables or pointers to memory are not desirable (and not available in pure functional languages), any function is always guaranteed to return exactly the same value if called with the same arguments.

Lambda calculus was introduced by Alonso Church in 1932 as an attempt to formalize every mathematical construct by using functions [33]. A mathematical function usually has a name, e.g. $f(x) = x + x$ is a function called f. When everything is supposed to be a function, having named functions everywhere, especially for functions that are only used once, might be impractical. So a specific notation was introduced to represent functions without a name. The previously defined function f is written as $(\lambda x.x + x)$ using this notation. Applying this function to the value 5 will give us a 10 as a result.

The interesting part about lambda calculus however, is that it allows us to pass other functions as an argument. Let us take a look at the following lambda expression: $(\lambda op.\lambda x.(op\ x\ x))$.

It defines a function which accepts two parameters *op* and x, wherein *op* is another function and x is a value. The body of the function is the application of *op* on two occurrences of x, so basically *op* acts as a binary operator. We can now use this function, for example, to apply the (+) operator on any applicable value: $(\lambda op.\lambda x.(op\ x\ x))\ (+)\ 5 = (\lambda x.((+)\ x\ x))\ 5 = (+)\ 5\ 5 = 10$. We can now use any other binary function as a parameter and get a different result, depending on the used function. Moreover a function can return not only values, but other functions as well. This concept is called **higher order functions**, an integral concept to functional programming languages. In such languages, functions are often represented by lambda expressions and handled as data, i.e. a function is a value in itself and the lambda expression is the lexical representation of that value.

The next important concept is that functional programming languages usually work with **immutable values and data structures** instead of mutable variables that are common in imperative programming languages. Functional programs are not written as a sequence of statements the way an imperative program is, but rather as a list of definitions and expressions, or even a single nested expression which evaluates to a single value. So instead of, for example, accessing data in memory and altering it via pointers or references, a functional program computes and returns a value. Therefore, there is no need for variables which change their state, just as there is no need to change the contents of a list. In a functional program a list is "altered" by creating a new list containing values of the old one plus some modifications and returning it. Same concept applies to composite types.

As previously said, functional programs are not a sequence of statements, but are rather a list of definitions and expressions which use these definitions. We will see this in our functional implementation of the dictionary lookup algorithm in listings 7 and 8. This also means that, unlike in imperative languages, **sequence is not used as a control structure in functional programming**. Functional programming languages can be usually executed in a REPL (Read-Evaluate-Print-Loop) environment, in fact Lisp was the first language to introduce such an environment. The machine reads user input, evaluates it, prints the result on console and loops to the initial state, i.e. waits for further user input. The user may write down the whole program as a single expression or define separate functions which are then invoked in an expression later on. Of course, such programs can also be written down in a text file and then compiled or interpreted similar to other languages [23].

Like with any other language constructs in functional programming, **conditional branching is represented by expressions**. In the previous Scheme example in which we added all integer values in a specific range, we have an `if` expression which either evaluates to the value of `total` if the value of `from` equals 0 or leads to a recursive call of the containing function, which will ultimately evaluate to a value.

Since a functional program is usually a list of definitions and an expression which uses them, or even just a single expression, there is no need for a loop-like control structure. Such programs usually **rely heavily on recursion to repeat specific computations**, which is also the way it is in mathematical functions. This does not mean that functional languages lack a loop construct, Lisp and its dialects for example include loop constructs, which however break away from a purely functional way of thinking. For example `(loop (print "Hello, world!"))` is an endless loop which prints `"Hello, world!"`. Haskell, on the other hand, does not offer such constructs and is considered a pure functional language. Some form of repetition without recursion can be reached by using list iteration functions, which evaluate to new lists, like the `takeWhile` function, which accepts a predicate and a list and returns list elements starting with the first one as a new list as long as that predicate is true.

Data structures like lists are usually defined recursively in functional languages, leading to potentially endless data structures, which leads to the necessity of **delayed or lazy evaluation**. That means that even if we define an endless list, for example we can simply write `lst = [1..]` in Haskell, it should not be evaluated right away which would lead to its initialization, hence we would never get any further and instead would get stuck in an endless initialization process. However, by means of delayed or lazy evaluation we still have got an endless list declared and bound to an identifier `lst`, we can now use this list, for example, to get first 10 integer numbers (`take 10 lst` in Haskell) or something in between. Of course, we can still call a function that iterates over the entire list, leading to an endless repetition, e.g. we could call `filter even lst` in Haskell which would endlessly print even numbers on the terminal until the program is terminated by other means, yet so can we do the same with an endless loop in an imperative language.

Another way to optimize such languages is by implementing tail call optimization, meaning that when the last expression that is evaluated in a function call is the recursive call of that function, the previous stack frame is replaced by the new one instead of adding the new stack frame on top of the previous, thus mitigating the stack overflow problem.

Example of a Functional Program

Now that we know what the ideas and concepts of functional programming are, let us implement our dictionary lookup algorithm in a functional way by using Haskell. Listing 7 contains the actual implementation of the algorithm, the test code is shown in listing 8, both listings are actually contained in a single compilation unit file.

The first thing we see in listing 7 is the explicit import of the module `Data.Char` which is required to use the function `toUpper` later in the code. Some modules are implicitly imported while others are not. After the import, we define the data types which represent our dictionary. A `Dictionary` has a `purpose` and consists of a list of `Sections`, each `Section` has a corresponding `letter` and consists of a list of `WordPairs`, each `WordPair` consists of a `word` and its `translation`. Each data type derives from the type class `Show`, meaning it can be printed on the terminal. These type declarations include named functions which can be applied to a value of that data type, e.g. if `wp` is a value of type `WordPair`, then applying (word wp) will return the String value stored as word in wp.

```haskell
import Data.Char

data WordPair = WordPair { word :: String, translation :: String } deriving (Show)
data Section = Section { letter :: Char, wordPairs :: [WordPair] } deriving (Show)
data Dictionary = Dictionary { purpose :: String, sections :: [Section] } deriving (Show)

head2 :: [a] -> Maybe a
head2 [] = Nothing
head2 (x:xs) = Just x

getTargetSection :: Dictionary -> Char -> Maybe Section
getTargetSection dict letter2 =
   let dictSections = sections dict
   in head2 (filter (\sec -> letter sec == letter2) dictSections)

lookUpTranslation :: Dictionary -> String -> String
lookUpTranslation dict word2 =
   let targetSection = getTargetSection dict (toUpper (head word2))
   in case targetSection of
      Nothing -> word2 ++ " : no matches found"
      Just targetSection ->
         let maybePair = head2 [x | x <- wordPairs targetSection, word x == word2]
         in case maybePair of
            Nothing -> word2 ++ " : no matches found"
            Just maybePair -> word2 ++ " : " ++ translation maybePair
```

Listing 7: Haskell implementation of the dictionary lookup algorithm.

Before we can continue with the implementation of our algorithm, we must take a look at our own implementation of the head function. We have already mentioned that lists are defined recursively in functional programming languages, usually a list is defined as a head and a tail, with the head being the first element and the tail is a list of the remaining elements. Thus the function head, which is meant to return the first element of a list, is implemented like this:

```haskell
head :: [a] -> a
head [] = error "empty list"
head (x:_) = x
```

So if we call Haskell's own function head on an empty list, a runtime error will occur. Since runtime errors are harder to fix, we define our own function head2 (in listing 7), which returns a Maybe a out of a list of as instead of an a. The a stands for a type parameter, which can be anything in this case, since we are not restricting it to a specific type class. Maybe is a data type which can act as a wrapper for functions which may go wrong and not return anything. Thus a value of type Maybe a represents either the value of type a (wrapped as Just a) or `Nothing`. So our `head2` function returns `Nothing` if applied to an empty list, or `Just` the first element if applied to a non-empty list. Unlike when using the original function `head`, using this function forces us to differentiate between the two possible outcomes in the source code, the program will not compile otherwise. In both cases we have two function definitions for the same declaration, which definition is called depends on the passed value. This is called *pattern matching* in functional languages.

Now let us examine the function `getTargetSection`. It accepts two parameters, a `Dictionary` and a `Char`, and returns a `Maybe Section` value. In the function we first define a value `dictSections` by calling the function `sections` on the passed `Dictionary` parameter `dict`, which returns us the list of all sections of that dictionary. We then filter this list for all sections whose assigned `letter` is equal to the passed `Char` parameter `letter2`, this should return a list with only one section, assuming the dictionary was initialized correctly and does not contain several sections with the same letter. Since we want a `Maybe Section` and not a list of `Sections` we call our function `head2` on that list, which will give us a `Just Section` if a section with a corresponding letter was found or `Nothing` if not.

Let us take a closer look at the application of the `filter` function in this example. The `filter` function accepts a predicate and a list as arguments, and returns a list of elements for which this predicate is true, in our case those are all sections which have the corresponding `letter`. The predicate here is passed as a lambda expression, an anonymous function `(sec -> letter sec == letter2)`. This function accepts a parameter `sec`, which is a section since we use this predicate together with a list of sections, and returns the result of an equality check between the `letter` of that section with the `Char letter2`, which is an argument of the function `getTargetSection`. This is an example how a lambda expression or an anonymous function can be used in a functional programming language.

The function `lookUpTranslation` is the one that actually implements our algorithm. It accepts a `Dictionary dict` and a `String word2` as parameters and returns a `String` as a result. Unlike in previous functions, there is no `Maybe` this time, meaning that this function is guaranteed to return something. First, we navigate to our target section by applying the `getTargetSection` function to the passed `Dictionary` argument and the first character of the passed `String` argument. To get the first character, we apply the Haskell's own `head` function to keep it short, since there shouldn't be any empty words in a dictionary anyway, and then apply `toUpper` from the imported module `Data.Char` to ensure that it is an upper case character. Now we have two cases, either there is no section which begins with the first letter of the passed String argument, in that case we return the passed word combined with the message that no match could be found, or the target section was found. In that case we search through the `wordPairs` of the `targetSection` and filter all words which are equal to our passed `word2` and get the first element of that list by using our function `head2`. This time we use Haskell's mechanism of list comprehension, however, we could have used the `filter` function just as well. If there is no such word in the list of `wordPairs`, then there is no such word in the entire dictionary. In this case we return the passed `word2` with the message that no match could be found. Otherwise we return the word with its translation as a `String`.

Listing 8 shows the test code of the program. Our dictionary is defined as a separate value which is then used in the main function of our program which returns an IO action. Lazy evaluation of Haskell means that the dictionary definition is not evaluated right away, but only when it is required to perform other evaluations, i.e. each time the `lookUpTranslation` function is called in the main program of our example.

IO actions are somewhat of a departure from the "no side effects" ideology of Haskell, since reading from and printing on a terminal are side effects, however, we wouldn't be able to get programs with which we can interact otherwise. The output is the same as that of the previous implementations of that algorithm.

```haskell
initTestDictionary :: Dictionary
initTestDictionary =
   Dictionary {
      purpose = "German-English", sections = [
         Section {
            letter = 'A', wordPairs = [
               WordPair { word = "Alphabet", translation = "alphabet"},
               WordPair { word = "Anweisung", translation = "statement"},
               WordPair { word = "Ausdruck", translation = "expression"}
            ]
         },
         Section {
            letter = 'B', wordPairs = [
               WordPair { word = "bauen", translation = "build"},
               WordPair { word = "Baum", translation = "tree"},
               WordPair { word = "Bedingung", translation = "condition"}
            ]
         },
         Section {
            letter = 'C', wordPairs = [
               WordPair { word = "Chip", translation = "chip"},
               WordPair { word = "Code", translation = "code"},
               WordPair { word = "Computer", translation = "computer"}
            ]
         }
      ]
   }


```

```
main :: IO()
main =
   putStrLn (
      (lookUpTranslation d "Alphabet") ++ "\n" ++
      (lookUpTranslation d "Anweisung") ++ "\n" ++
      (lookUpTranslation d "Ausdruck") ++ "\n" ++
      (lookUpTranslation d "bauen") ++ "\n" ++
      (lookUpTranslation d "Baum") ++ "\n" ++
      (lookUpTranslation d "Bedingung") ++ "\n" ++
      (lookUpTranslation d "Chip") ++ "\n" ++
      (lookUpTranslation d "Code") ++ "\n" ++
      (lookUpTranslation d "Computer") ++ "\n" ++
      (lookUpTranslation d "Algorithmus") ++ "\n" ++
      (lookUpTranslation d "Programm") ++ "\n" ++
      (lookUpTranslation d "Sprache") ++ "\n"
   )
   where d = initTestDictionary
```

Listing 8: The test code of our dictionary implementation in listing 7.

Summary

We conclude this section with a brief summary of the important aspects of functional programming:

- The **expression** is the main language construct which is evaluated **without causing side effects**. The program consists of a list of definitions and an expression which uses those definitions and results in a value. Some programs may be written entirely in a single expression.

- **Functions** specify the functionality of the expressions, some are distributed with the languages (arithmetic, list functions, etc.) others can be defined by the user. A function accepts arguments and returns a value, a function without arguments is just a value.

- **Lambda expressions** represent anonymous functions, and can be passed as arguments or returned as a result of another function.

- **Control structures** play a different role in functional programming languages:
  1. *Sequence* – is not present. We may have a list of defined functions, but they are not executed in the sequential order the way statements are executed in imperative languages.
  2. *Conditional branching* – functions and conditional expressions are evaluated to a certain value depending on specific conditions.

3. *Repetition* – is usually implemented via *recursion*, however, some functional languages offer loop-like language constructs.

- **Primitive data types** are the same as in imperative languages. Additionally **functions are handled as data**, so the return type of a function may be another function.
- **Lists** and other dynamic data structures are commonly used instead of arrays. **Composite types** similar to records are available in some languages.
- **Values and data structures are immutable**, i.e. values cannot be reassigned, they are basically constants, and contents of data structures cannot be altered.
- Functional languages do not offer explicit memory management mechanisms, since that would contradict the **"no side effects" policy** of such languages.
- Functional languages prefer **lazy or delayed evaluation**, meaning that expressions are not evaluated right away, but only when it is absolutely necessary to do so. This allows to define potentially endless data structures, segments of which can later be used to perform specific calculations.

Other than that, the structure of a functional program is similar to that of a procedural program. A functional program consists of import instructions, type and function definitions as well as a main function or an expression which acts as one.

2.5. Logic

While the three previously discussed programming paradigms are indeed quite distinct from each other, there are still some overlapping concepts, e.g. all three paradigms offer some sort of subroutines and the general structure of the programs is similar among the paradigms, i.e. we usually have a program name, import statements, type declarations, subroutines and some sort of a main subroutine. This does not apply to the paradigm which stands in the focus of this section: logic programming, which is sometimes also called non-procedural programming since there are no user defined subroutines in this style of programming [36].

Prolog [37] (French: programmation en logique) is the most used logic programming language today. That being said, the entire paradigm is not that widely utilised compared to previously discussed paradigms. It is mostly used in very specific fields, for example AI development, expert systems, relational databases and natural-language processing [23]. The declarative nature of Prolog allows to define a set of facts and rules in a very concise and readable way, which are then used to get an answer. That answer might be a `true` or `false`, or a set of values which fulfil specific constraints.

Concepts of Logic Programming

The previous three paradigms use boolean logic which is based around three basic operators `AND`, `OR` and `NOT` and two values `TRUE` and `FALSE`. The entire logic programming paradigm is based on the *first-order predicate calculus* with some limitations [6], similar to how functional programming is based on the *lambda calculus*. However instead of using functions to represent mathematical relations, logic programming relies on logical terms.

Let us take a look at a simple set of such terms written using the programming language Prolog in listing 9. What we have here is a set of **atomic propositions**, i.e. terms that consist of two parts, a functor which gives a name to a relation (e.g. `mother` or `man`) and an ordered list of atoms, which represent the objects in that relation. For example the term `mother(alice, charlie)` states that there are two atoms `alice` and `charlie` and `mother` is their relation, it does not explicitly state that `alice` is the `mother` of `charlie`, it could also mean that `charlie` is the `mother` of `alice` or something else entirely, the actual semantics should be known to the user. These terms represent a **set of facts** which we can now use to get some sort of information. For example, we can ask if a relation `woman(alice)` exists and the answer will be `true`. We can also ask for atoms which fulfil specific relations by replacing them with variables, for example querying `man(X)` would list `bob`, `charlie` and `david` as valid solutions for the variable `X`. The value of the variable is inferred from facts and other rules, meaning that sufficient information must be present, otherwise the value of a variable cannot be inferred. In a similar way we can ask who is the mother of `charlie` (assuming that this is the meaning of this relation) with the query `mother(X, charlie)` which will result in `X = alice`. This way, we can use atomic propositions to both state facts, and check them.

```prolog
woman(alice).
man(bob).
man(charlie).
man(david).
mother(alice, charlie).
father(bob, charlie).
father(david, bob).
```

Listing 9: Prolog facts of the family relations example.

Working with facts only is not feasible in the long term. As we see, `david` is the grandfather of `charlie`. We could add this as a fact, however, we can also infer this information from already existing facts that `david` is the father of `bob` and `bob` is the father of `charlie`. Using rules instead of facts whenever possible reduces the overall amount of definitions and therefore results in a shorter program. So let us add some rules to the previously stated facts:

```
child(X, Y) :- mother(Y, X).
child(X, Y) :- father(Y, X).
parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).
grandparent(X, Z) :- parent(X, Y), parent(Y, Z).
grandfather(X, Y) :- grandparent(X, Y), man(X).
grandmother(X, Y) :- grandparent(X, Y), woman(X).
```

Listing 10: Prolog rules of the family relations example.

To define rules we may combine several atomic propositions into compound propositions by using logical operators. Using all operators of the predicate calculus allows to define propositions in many different ways [23], which is not desirable for a programming language. The more complex our relations and, consequently, rules become, the more variations of such rules appear. Most logic programming languages rely therefore on the Horn clauses, which are capable to represent almost any logical statement, as the standard notation for rules [6]. A Horn clause is composed of a head and a body of terms linked with a logical AND operator: the clause $H \leftarrow B_1, B_2, ..., B_n$ (in Prolog the $\leftarrow$ is replaced by `:-` and a `.` is placed after each fact, rule or query) means that H is true if all propositions on the right side are true. Since the AND operator is the only one allowed, there is no need to explicitly write it out and it is replaced by a comma instead. Horn clauses without a head are called headless Horn clauses and can be used to state facts (in case of an atomic proposition) or as queries (both atomic and compound propositions). All facts written in listing 9 are headless Horn clauses.

For example, the rules `parent(X, Y) :- mother(X, Y)` and `parent(X, Y) :- father(X, Y)` mean that `X` is a parent of `Y` if `X` is either a mother or a father of `Y`. By using these rules we can now check whether `alice` is a parent of `charlie` by querying `parent(alice, charlie)` and the answer will be `true`. We can also ask *"Who are the parents of Charlie?"* with `parent(X, charlie)` resulting in `alice` and `bob`. The rule `child(X, Y)` works in a similar way, however we swap the variables `X` and `Y` when we check the queries `mother` and `father`. Querying `child(X, alice)` or `child(X, bob)` will yield `charlie` as a result while the query `child(charlie, X)` will return both `alice` and `bob`.

Another example is the definition of the `grandparent(X, Z)` rule as a conjunction of `parent(X, Y)` and `parent(Y, Z)`, meaning that `X` is a grandparent of `Z` if

X is a parent of Y and Y is a parent of Z. We can now specify what a grandfather or a grandmother is by combining the grandparent rule with either the man or the woman relation using the logical AND operator. The query grandfather(X, charlie) will now tell us that david is the grandfather of charlie, however grandmother(X, charlie) will not give us any positive results since we haven't got any objects in our facts database that fulfil all required relations.

The process of inferring of a proposition from other given propositions is called **resolution**. Let's say P $\leftarrow$ Q and Q $\leftarrow$ R, we can resolve this to P $\leftarrow$ R. In other words, if we assume that C_1 and C_2 are Horn clauses and the head of C_1 matches one of the terms in C_2, then we can replace the matched term in C_2 with the body of C_1.

There are two distinct approaches to resolve queries in logic programming. Forward chaining attempts to find a sequence of matches starting with the facts and rules and leading to the query. Prolog follows the alternative approach – backward-chaining – wherein the execution environment attempts to find a chain of propositions starting with the original query and leading to the facts and rules in the database. Furthermore, Prolog follows the depth-first search pattern when analysing compound propositions, i.e. it tries to prove the first proposition completely before moving to the next one, thus if the current proposition cannot be proven, there is no need to prove the propositions that follow it. The final feature of Prolog that should be mentioned is backtracking. When a compound proposition is analysed and the system fails to prove a term it abandons it and attempts to find alternative solutions for the previous term, if there is one. Once an alternative solution for the previous term was found, the system attempts to prove the formerly abandoned term again. This continues until either the whole query is proven, leading to some sort of a positive answer, or all possibilities are exhausted and the process fails, meaning that there are no facts and rules in the database that can prove that query.

Let us take a look at how this process works by resolving the query *"Who is the grandfather of Charlie?"*, i.e. grandfather(X, charlie):

1. The query grandfather(X, charlie) is associated with the head of the rule grandfather(X, Y) :- grandparent(X, Y), man(X). The variable Y is instantiated as charlie, X is not instantiated. Basically, we now have the following query:
   grandfather(X, charlie) :- grandparent(X, charlie), man(X).

2. grandparent(X, charlie) is the next sub-query that has to be proven. To do so, we have to find a match for parent(X, Y), parent(Y, charlie).

3. The first attempt is made with mother(X, Y), which is associated with the fact mother(alice, charlie).

4. We have found a match and return to the previous query, which currently looks like that: `grandparent(alice, charlie) :- true, parent(charlie, charlie)`, meaning that we now have to prove that `charlie` is a parent of `charlie` which will fail. Once it does so, the system tracks back to the previous sub-query, which was `parent(X, Y)` and tries to find an alternate solution.

5. The next attempt will be made with the match `father(bob, charlie)` which will also lead to `parent(charlie, charlie)`, so again that query has to be re-evaluated.

6. Once the sub-query `parent(X, Y)` is matched to `father(david, bob)`, `X` is instantiated as `david` and `Y` as `bob` in the context of the current query. The next sub-query becomes therefore `parent(bob, charlie)` which is true, thus proving the entire query `grandparent(X, charlie)` instantiating the variable `X` as `david` in the context of the query `grandfather(X, charlie) :- grandparent(X, charlie), man(X)`.

7. Which means that we now have to prove `man(david)` which is true proving the entire original query and giving us the answer `X = david`.

Example of a Logic Program

Since we now have gained some insight into logic programming and the programming language Prolog, let us implement our dictionary lookup algorithm using this language. The word algorithm might be a bit misleading in this case, since, as we have already seen, we are not implementing algorithms per se in logic programming. We do not perform any arithmetic calculations or searches through collections directly, instead we state facts, define rules and try to prove specific queries. All calculations, searches and pattern matching is performed in the background by the implementation of the language.

First and foremost, we have to define what is a dictionary. In the previous examples we would define data structures for the dictionary first, then the functionality as subroutines and only at the end feed our program with data and test it. Here we define our dictionary as a fact right away, as demonstrated in listing 11. To achieve this, we use the Prolog data structure which is ironically called *dict*. A Prolog *dict* is a map like data structure which stores key-value pairs, values can be accessed by their keys later on. In a similar way to our previous implementations, we define a *dict* `germanEnglishDictionary` with its purpose and a list of sections, which store a letter and a list of `wordpairs`, and state a fact that it is a `dictionary`.

```
dictionary(germanEnglishDictionary{
    purpose:"German - English", sections: [
        section{
            letter:"A",
            wordPairs:[
                wordPair("Alphabet", "alphabet"),
                wordPair("Anweisung", "statement"),
                wordPair("Ausdruck", "expression")
            ]
        },
        section{
            letter:"B",
            wordPairs:[
                wordPair("bauen", "build"),
                wordPair("Baum", "tree"),
                wordPair("Bedingung", "condition")
            ]
        },
        section{
            letter:"C",
            wordPairs:[
                wordPair("Chip", "chip"),
                wordPair("Code", "code"),
                wordPair("Computer", "computer")
            ]
        }
    ]
}).
```

Listing 11: Facts database of our prolog dictionary example.

Now that we know what a dictionary is, we must define rules to get data from that dictionary (see listing 12). Our goal is to get a translation of a word, to do that, we must first get the section in which the corresponding word pair is located. In order to get this section, we first need the first letter of the word that we want to translate. Words are stored as strings which can be converted to lists, which is the approach that we have used in the rule `firstLetterInWord(Letter, Word)`. Here we map the passed `Word` to a corresponding list `X` by using the internal rule `string_codes`. Furthermore we state that the list `X` is equal to `[H|_]` binding the variable `H` to the first element in the list `X`. The _ in `[H|_]` just tells that the tail of the list is of no interest to us. By using `string_codes` again we transform that element back into a string and map it to `Y`. Finally, we map `Y` to `Letter` as an upper case string. This way we can find out the first letter in a given word, for example, if we type the query `firstLetterInWord(X, "bauen").` the answer will be `X = "B"`. We can also use this rule to check if a letter is in fact the first letter in a word by typing `firstLetterInWord("B", "bauen").`, which is true in this case.

The next rule – `sectionInDictionary(Section, Letter, Dictionary)` – proves that a `Section` with a given `Letter` is in fact in the `Dictionary`, so when we know the `Letter` and the `Dictionary`, we can infer the corresponding `Section`. To achieve this, we first map the variable `Sections` to the list of `sections` in our `Dictionary`. Furthermore, we state that `Section` must be a member of that list and finally that the `letter` attribute of that `Section` must be equal to the passed `Letter` of the rule. If we now enter the query `dictionary(X), sectionInDictionary(Y, "A", X).`, we will get all sections which have the letter `"A"` in all dictionaries that we know about. In this case `X` will be our German-English dictionary and `Y` will be its section `"A"`.

```
firstLetterInWord(Letter, Word) :-
        string_codes(Word, X),
        X = [H|_],
        string_codes(Y, [H]),
        string_upper(Y, Letter).

sectionInDictionary(Section, Letter, Dictionary) :-
        Sections = Dictionary.get(sections),
        member(Section, Sections),
        Section.get(letter) = Letter.

translation(Word, Translation, TypeOfDictionary) :-
        dictionary(Dictionary),
        Dictionary.get(purpose) = TypeOfDictionary,
        firstLetterInWord(Letter, Word),
        sectionInDictionary(Section, Letter, Dictionary),
        member(X, Section.get(wordPairs)),
        X = wordPair(Word, Translation).
```

Listing 12: Translation lookup rules of our prolog dictionary example.

The rule `translation(Word, Translation, TypeOfDictionary)` is meant to prove that `Translation` is a translation of `Word` in a specific `dictionary`. First we introduce the variable `Dictionary`, ensure that it is in fact a `dictionary` and that its `purpose` is the one that we have given in our rule. Then we get the first letter of the given `Word` mapping it to the variable `Letter`, which is then used to get the correct section of our `Dictionary`. Then we tell that the variable `X` is one of the `wordPairs` of that `Section` and that both `Word` and `Translation` are included in that `wordPair`. Therefore, if we leave the variable `Translation` uninitialized when calling the rule, it will be initialized now. Calling `translation("bauen", X, "German - English").` will yield `X = "build"` as a result. Again, we may, for example, use this rule to prove that `"build"` is the translation of `"bauen"` by calling `translation("bauen", "build", "German - English").` which is true.

```
printTranslation(Word, Translation, TypeOfDictionary) :-
        translation(Word, Translation, TypeOfDictionary),
        write(Word), write(" : "), write(Translation), write("\n").

printTranslation(Word, Translation, TypeOfDictionary) :-
        not(translation(Word, Translation, TypeOfDictionary)),
        write(Word), write(" : no matches found\n").

test() :-
        printTranslation("Alphabet", _, "German - English"),
        printTranslation("Anweisung", _, "German - English"),
        printTranslation("Ausdruck", _, "German - English"),
        printTranslation("bauen", _, "German - English"),
        printTranslation("Baum", _, "German - English"),
        printTranslation("Bedingung", _, "German - English"),
        printTranslation("Chip", _, "German - English"),
        printTranslation("Code", _, "German - English"),
        printTranslation("Computer", _, "German - English"),
        printTranslation("Algorithmus", _, "German - English"),
        printTranslation("Programm", _, "German - English"),
        printTranslation("Sprache", _, "German - English").
```

Listing 13: Print rules and test code of our prolog dictionary example.

Listing 13 contains the print rules and the test code necessary to achieve the same output as with previous examples. The print rules cover two cases, either the translation is successfuly found and the results are printed, or the lookup fails for some reason (no word in the dictionary, wrong dictionary, etc.) and the message `no matches found` is printed after the word that we have tried to translate. The test rule ist calling the print rules leaving the variable `Translation` uninitialized as an anonymous variable, since we do not need a named variable in this rule. The resulting output is basically the same as with the previous examples with one slight difference:

```
?- test().
Alphabet : alphabet
Anweisung : statement
Ausdruck : expression
bauen : build
Baum : tree
Bedingung : condition
Chip : chip
Code : code
Computer : computer
Algorithmus : no matches found
Programm : no matches found
```

```
Sprache : no matches found
true ;
false.
```

We see `true; false.` at the end, which means that we have successfully found the translations, i.e. the rules were proven on the first run, but additional tries failed. This shows how Prolog may try to satisfy the given goals after they have been already proven once, after all, maybe we could have gotten other results with a bigger database. In our case, we have a pretty simple relation: one word has only one translation. But if we add more translations for certain words, for example as a new `wordPair("Ausdruck", "printout")` in the section `"A"` in our dictionary in addition to the already existing ones, we now have two matches for the same word. So if we call the `test()` rule now, we will get the following (shortened) output:

```
?- test().
Alphabet : alphabet
Anweisung : statement
Ausdruck : expression
bauen : build
...
Sprache : no matches found
true ;
Ausdruck : print
bauen : build
...
Sprache : no matches found
true ;
false.
```

So basically, the initial attempt succeeds with the same output as before, but now we also have a successful second attempt to validate this rule with a different translation for the word `"Ausdruck"`. Furthermore, the print rules prior to the rule that produced `"Ausdruck"` are not printed again, meaning that Prolog has backtracked to the rule with different possible outcomes and re-evaluated it and all the following rules. And again, we can see a `false` at the end meaning that no other possibilities exist.

Of course, we could have implemented this example in a different way, for example just as a set of facts `wordPair("bauen", "build")`, `wordPair("Baum", "tree")`, etc. This would make our rules less complex, a mere `translation(X) :- wordPair(X, Y), write(Y).` would be enough, or we could just enter the query `wordPair("build", X).` and get the results. However we have tried to keep the implementations of this example consistent among the discussed paradigms while still trying to show their features. In fact we could have provided simpler implementations for all of these paradigms, for example by relying on data structures internally supported by the languages such as maps. This would, however, result in some rather boring examples.

Summary

As we can see, logic programming follows an **even more abstract approach** compared to functional programming. Therefore, programs written with such languages tend to be even more declarative and less algorithmic in nature requiring a good understanding of the execution semantics of the language. Such programs consist of a set of facts and logical rules which are required to solve specific problems. Users can enter queries which are then attempted to be validated by the program based on rules and facts. While other paradigms share quite a lot of similar concepts, logic programming stands aside from the others in many ways.

To sum it up:

- Logic programs consist of **facts, rules and queries**.
- **Horn clauses** are the standard notation to write them. A Horn clause consists of an atomic proposition on the left side and an atomic or a compound proposition on the right side, which corresponds to a rule in a logic program. Headless horn clause do not have a left side and are used as queries or to state facts.
- The propositions are **combined with a logical operator** `AND`, meaning that if all propositions are true, the entire query is true.
- Data in represented by **constants and variables**.
- Constants may not have a specific value bound to them the way it is in the previously discussed paradigms. **They are used to represent relations between objects**, e.g. `mother(alice, charlie)` where `alice` and `charlie` are constants and `mother` is their relation. Actual values are also constants, such as the string words in our dictionary example as well as the dictionary itself.
- **Variables are instantiated when a rule is evaluated**, either directly as a parameter, or via mapping this variable to values and constants stated in facts or inferred from other rules. This way, entering the query `mother(X, charlie)` will instantiate `X` to `alice` assuming `mother(alice, charlie)` is stated as a fact or can be inferred by other means.
- `X = Y` means that `X` is bound with `Y`, or in other words that they are equal, so `X = X + 1` does not make sense. It is, however, possible to manually assign variables in Prolog with the `is` operator, e.g. `X is X + 1`.

2.6. Conclusion – Multi-Paradigm Languages

In this chapter we have discussed what programming is and what are the four main programming paradigms. We have seen the implementation of the same algorithm with four different programming languages, each representing their respective paradigm. All of them are general purpose languages, meaning that they should be capable to implement any program which can be executed by a machine. So if all of them, as well as many other existing languages, are capable to solve any problem, why do we have so many? Hundreds of programming languages exist today, some are no longer used, others are used and are still getting active support. If we count all dialects, esoteric and domain specific programming languages, we may easily get numbers reaching several thousands [5]. A lot of these languages are used for a short time in small communities for some very specific tasks. A comparatively small number of general purpose languages is, however, widely employed both in the industry and education around the world. So if there are languages capable to do everything, why don't we agree to use just one of them.

As we have seen in the previous sections, even if these languages are capable to do (almost) everything, each language excels at some specific tasks while being overshadowed by other languages in other fields of application. C for example is a procedural language that is capable to address memory directly, allowing the programmers to write potentially more efficient programs as long as they know what they are doing. Other languages are, on the other hand, capable to provide more elegant solutions for tasks, where C would either produce unnecessary complex code or even reach its limits.

Prolog offers an elegant way to implement decision making programs, which would require more complex code in a language like Java. At the same time, the Prolog user has much less control over the execution of the program [23] compared to less abstract languages, meaning that decision making programs written with such languages might be far more complex, but potentially more efficient.

Ultimately, there is no such language which would excel at any problem. While some languages provide more readable programs for specific tasks, others might be able to implement slightly more efficient solutions at the cost of code which is very hard to read and maintain. Therefore, software systems today may be implemented with different programming languages glued together by yet another programming language.

Procedural programming allows to easily translate algorithms into programs which can be then executed step by step and are therefore easy to debug. Functional programming is suited well to work with big data sets. Both functional and logic styles of programming usually result in programs that are easy to read due to their declarative nature. Object-oriented programming excels at representing real world entities and their relations among each other, furthermore it is well suited to implement graphical user interfaces. However, most currently employed object-oriented languages rely on procedural and functional concepts to provide functionality for the objects. Therefore, we can observe a certain tendency amongst currently employed languages which were originally centered around a specific paradigm to adopt styles and language constructs typical to other paradigms. C++ was developed as an evolution of C and borrows most of its language constructs to describe functionality. Java developers have implemented lambda expressions in Java version 8 introducing a language construct rather common among functional languages. Similarly, arrow functions, which is another name for lambda expressions, were introduced in JavaScript about 20 years after its initial release.

Languages that *"support two or more conventional programming paradigms"* [8] or represent *"a linguistic framework which does not force the programmer into thinking or working in only one model"* [38] are called multi-paradigm languages. Usually, these languages support a range of procedural, object-oriented and functional concepts. Still, such languages might have a bigger focus on a specific paradigm or have certain limitations which do not allow to program in a specific paradigm with its full potential. Logic programming can be implemented by using constructs from other paradigms in addition to providing libraries with necessary semantics and data types to a certain degree [39], however it will most certainly not result in Prolog like code. Multi-paradigm general purpose languages are capable to effectively perform a wide range of tasks. Nevertheless, specialized languages are usually better suited to solve problems they were specifically designed for. Finally, a paradigm is first and foremost a way of thinking or solving a problem and not a set of language constructs of a language. Even if a language does not officially support a specific paradigm, it does not mean that it is not possible to program in that way with this language. But it is certainly easier to do so with a language that supports it.

The goal of this chapter was to give a brief overview over programming itself as well as four different styles of programming. In the next chapter we will take a look at a selection of programming languages and tools used in the context of education.

3. Programming Languages in Education

In the previous chapter we have discussed what defines programming. We have observed, that there are many different ways to write programs, resulting from the various approaches and requirements. We finally have come to conclusion, that many programming languages used today are multi-paradigm languages, meaning that they can support different styles of programming which should enable the language to be used more broadly. Still, there is no such thing as a universal programming language.

According to the TIOBE index [40], which tracks the popularity of the programming languages by evaluating search statistics of various search engines, C Python and Java make up the top three most popular programming languages worldwide in the year 2021. Search queries for the languages and corresponding tutorials, book purchases, job requirements, available courses and so on can contribute to the rating of a language in that index.

Language	*2021*	*2016*	*2011*	*2006*	*2001*	*1996*	*1991*	*1986*
C	1	2	2	2	1	1	1	1
Java	2	1	1	1	3	26	-	-
Python	3	5	6	8	27	19	-	-
C++	4	3	3	3	2	2	2	8
C#	5	4	5	7	13	-	-	-
Visual Basic	6	13	-	-	-	-	-	-
JavaScript	7	8	10	9	10	32	-	-
PHP	8	6	4	4	11	-	-	-
SQL	9	-	-	-	-	-	-	-
R	10	17	31	-	-	-	-	-
Lisp	34	27	13	14	17	7	4	2
Ada	36	28	17	16	20	8	5	3
Basic + dialects	-	-	7	6	4	3	3	5

Table 1: Long term TIOBE index [40].

Table 1 displays a long term summary of that index over the course of several years. Until 2010 the authors have tracked the rating of Basic and all of its dialects as a single entity, therefore the separation into two categories *Basic + dialects* and *Visual Basic* in the table. Furthermore, the authors have only recently started to track SQL, although the language is widely used and has been so for a very long time.

Most of the listed languages are multi-paradigm general-purpose languages (GPL). Exceptions are C and Basic, both are procedural GPLs, Lisp, a functional GPL, and domain-specific languages R and SQL, the first being used for statistical analysis while the latter is designed to work with relational databases. The

GPLs in the table, while capable of solving any algorithmic problem, are either restricted by specific environments (JRE, Microsoft .NET) or by their originally envisioned fields of use (e.g. web development). Therefore, the choice of a language depends on many factors, amongst other things are project requirements, platform limitations and company policies.

The intentions of a potential programming student can be inferred from looking at such statistics. Most students want to acquire a specific set of skills which will guarantee their employment. In case of computer science this set encompasses programming languages, especially those in high demand. As we see, a lot of different languages are employed in the industry and while some are built around very specific concepts, e.g. R and SQL, justifying the need to teach them separately, others share numerous commonalities meaning that teaching one of such languages to the students should be enough to convey the most important programming concepts which is realistically possible in a beginner programming course. Therefore, while a student is rather interested in a language heavily used in the industry [41], educators might look for a language better suited to teach specific concepts [42]. In the next section we will discuss what is the general intention of introductory programming education.

3.1. The Purpose of Programming Education

What is programming education? In general, teaching programming is an integral part of the much wider computer science education. Computer science theory, programming language theory, algorithms, robotics, artificial intelligence, computer graphics, data base managements systems, object-oriented modelling, etc. are just a few topics of computer science education and programming as a process is a necessary part of most of them [43][44]. Therefore, programming courses are usually offered at the beginning of a computer science study [45][46][47]. Traditionally, these courses are designated CS1 and CS2, however, as written in [48], there is no universal standard on what contents these courses should include, especially considering that several decades have passed since the introduction of these designations and the world of computer science being a subject of constant evolution during that time. Usually, CS1 covers basic programming concepts while CS2 focuses on data structures and algorithms based on them. Knowledge gained in these courses is required in most other, more specialized, computer science lectures, which, however, might rely on other programming languages more suited to the contents of the respective courses.

As discussed in the previous chapter, programming is a process of finding an algorithm to solve an abstract problem and translating it into a program by using a programming language. Thus, a programming language is merely a tool, albeit a very important one [49], to achieve a goal in the same way as pencil and paper are used, for example, to solve a mathematical equation or write an essay. Therefore, one of the goals of a CS1 course is to teach students how to use a programming language, a tool that they will be using over the course of their entire studies as well as in their future careers. Since there are many different programming languages, the language used in such a course should be of a general nature in order to facilitate switching to other languages used in other courses or in the industry. Ultimately, while teaching how to use a programming language is necessary in a CS1 course, the focus is put into teaching the underlying concepts, which should be more or less universal to all programming languages, meaning that the language employed in such a course should be capable to demonstrate these concepts. To put it simply, the goal of the introductory programming education is to teach students how to program, not to teach them a specific language.

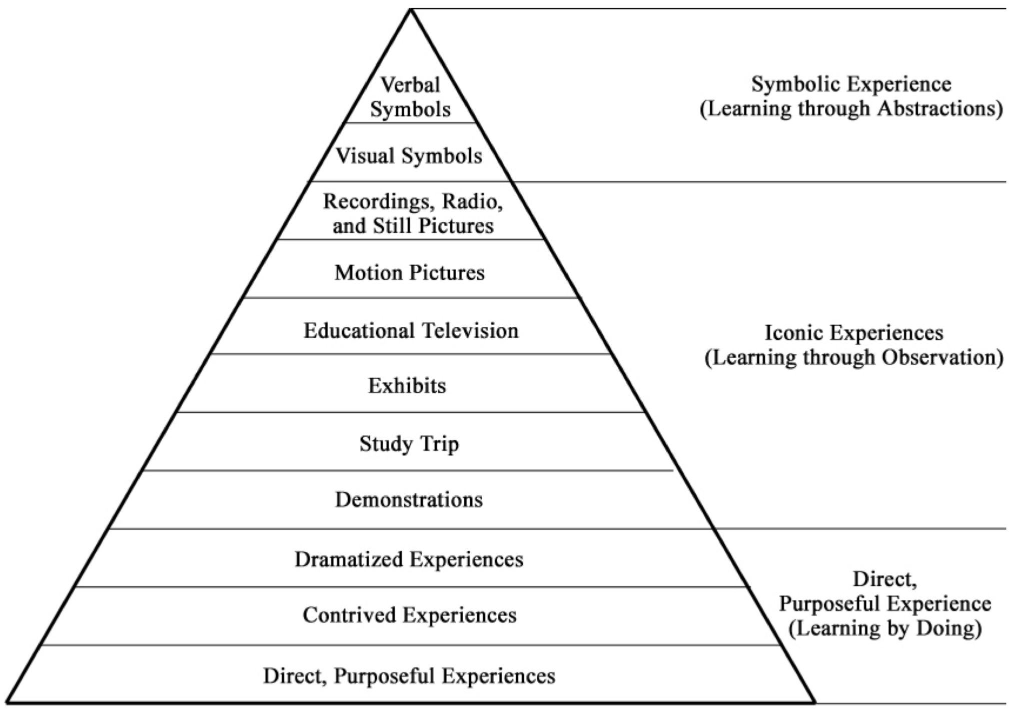

Verbal Symbols
Visual Symbols
Recordings, Radio, and Still Pictures
Motion Pictures
Educational Television
Exhibits
Study Trip
Demonstrations
Dramatized Experiences
Contrived Experiences
Direct, Purposeful Experiences
Symbolic Experience (Learning through Abstractions)
Iconic Experiences (Learning through Observation)
Direct, Purposeful Experience (Learning by Doing)

Figure 1: Dale's Cone of Experience [50].

A CS1 course is usually separated into a lecture and a practical part. If we look at the top and the middle sections of the Cone of Experience (figure 1), these are the contents covered in the lecture. Abstract concepts are explained to the students supported by code examples and demonstrations. The practical part of a CS1 course covers the bottom part of the cone, i.e. learning by doing. Both parts are important, without the practical part the students will never learn how to program, even if they perfectly understand the concepts taught in the lecture. Without the lecture, the students might learn how to program by imitating the actions of the instructor, however, they will not truly understand why they are doing something the way they are doing. This way, students may tend to copy solutions from other resources and modify them until the program compiles and does more or less what is asked and will have difficulties when confronted with a different programming language.

The Cone of Experience was designed to demonstrate the importance of media in the process of teaching and learning. The Berlin model of education [44] also recognises this importance. According to this model, educators have to meet specific decisions based on specific conditions when designing a course. One of such decisions is: *"How is the information carried to the students?"*. To be precise, the decisional factors of the Berlin model are:

1. Intention – what is the objective of a course? A CS1 course should teach the students fundamental concepts of programming required to understand contents of other lectures.

2. Contents – what information is taught in the course? The main contents of a CS1 course are algorithms, data types, control flow, static and dynamic data structures, recursion, object-oriented concepts, concepts of functional programming. As we have already mentioned, there is no universal standard for the structure of a CS1 course, therefore, the contents may vary among educational institutions.

3. Methods – how are these contents conveyed? As previously explained, a CS1 course is usually separated into two parts, a lecture and a practical part.

4. Media – what media is used to present the information to the students? Presentation slides, diagrams, example programs, etc. can be used for this task. A programming language can be put into this category, since it is a tool to write programs, i.e. it is a medium used to carry information. Social economic factors may play an important role here.

5. Consequences – which socio-cultural as well as anthropological and psychological effects will be the result from participating in a course? In case of a CS1 course, students should be capable to use the gained knowledge when confronted with programming languages previously unknown to them.

As we can see, the choice of a programming language plays an important role in an introductory programming course, some would even argue this choice is crucial to the students' subsequent performance in computer science studies [41]. In the following section, we will take a look at notable languages employed currently or in the past in the field of education.

3.2. Choosing a Language

In some areas of computer science education, the selection of the programming languages is rather limited and the choice is simple [51]. For example, SQL is a traditionally used domain-specific language in the courses teaching relational database management systems. HTML, CSS and JavaScript are the trio of common languages used in conjunction in web development [52]. However, choosing a specific general-purpose programming language is not a simple task, neither in education nor in any other field of application and many different factors have to be considered when making such a decision. In the industry that choice may be decided upon factors that are irrelevant in the context of education, such as successful marketing campaigns or company policies. The practical use of a language in the industry has, however, a substantial impact on the choice of the languages in education, i.e. languages are often selected not by their suitability as an educational language, but based on the demand in the labour market [53], even if those languages are not entirely suitable for the purpose of teaching programming [42].

Generally speaking, the deployed language should be capable to demonstrate the contents of the course in a clear and non-confusing way. The students should not struggle understanding the language which would distract them from focusing on the actual contents of a course [54]. Furthermore, the chosen language should keep the students motivated and interested in the practice of programming in general [49][53]. The syntax and semantics of the language should not be too different from other commonly employed languages to facilitate students switching to other languages in other courses or in their professional occupation. A language that is too different in nature to other "popular" languages might cause aversion amongst students directed to that language and, consequently, to the course in which it is employed leading to a higher drop-out rate. Finally, the language should provide sufficient tool support [51]. We will give a more in-depth summary of criteria that have to be considered when choosing an introductory programming language and, therefore, in the design of our own educational language in chapter 4, but before that, let us take a look at the programming languages used for the purpose of education.

3.2.1. Professional Languages

The first programming languages used in education were, not surprisingly, assembly languages. They were succeeded by the first higher level languages FORTRAN and COBOL, which were both developed for specific tasks that were already discussed in section 2.2. The rise of structured programming has ultimately led to their replacement by Pascal, a language specifically designed for education, until the emergence of object-oriented programming and the subsequent retirement of Pascal. Since then, C/C++ and Java were the prominent languages in education. [41]

C was initially developed for system programming and thus supports an explicit memory management mechanism to facilitate resource efficient programming. This, however, is a rather difficult concept for beginner programmers, who are confronted already with two different problems at the same time: designing an algorithm for an abstract problem and finding the right language constructs to translate it into a program. Throwing manual memory management into the mix might confuse them even more. When handled poorly, programs may lead to unexpected behaviour and a beginner student will have a hard time to trace its origin. Listing 14 demonstrates an example of a dangling pointer problem in C. Function `foo` returns an address of a local variable, which is deallocated as soon as the function is terminated, resulting in a program which seemingly does nothing when it is executed. This also demonstrates another issue of C and C++: cryptic error messages or complete lack thereof.

```
#include <stdio.h>

int *foo() {
    int a = 42;
    return &a;
}

int main(void) {
    int *dp = foo();
    printf("%d", *dp);
    return 0;
}
```

Listing 14: An example of a dangling pointer in C.

C++ [21] was developed as an extension of C by introducing object-oriented constructs. Due to its popularity, the language has grown over time introducing language constructs which can be used interchangeably, which also might lead to confusion amongst beginner programmers. On the other hand, both languages are supported by a huge amount of literature and documentation, reliable development environments (which may not exactly be easily approachable by beginners) and have the popularity factor. C++ can be used to teach both procedural and object-oriented programming and, since version C++11 and the inclusion of lambda expressions, it can be used to teach functional programming as well. However, the problem shown in listing 14 can happen in C++ as well. C and C++, while being more complex compared to Java or Python and thus harder to understand, especially at the beginning, are better suited than those languages to teach lower level concepts, while still being high-level general-purpose languages. Therefore, they are better suited for advanced specialized courses rather than for CS1 and CS2 [55].

At the time of its introduction, Java [56] was a smaller and a more manageable language compared to C++. Combined with the increasing popularity of object-oriented programming and a successful marketing campaign by Sun, Java quickly became the most prominent programming language in programming education [57][58]. It follows a more abstract approach compared to C/C++ dropping pointers and explicit memory management. Furthermore, it is platform independent and is utilized in a wide range of domains, e.g. web and mobile app development. Due to its success, users can rely on a wide range of documentation and powerful libraries.

However, being a professionally used language, it has grown tremendously since its introduction an can no longer be considered a “simple” language. Furthermore, although it supports procedural and functional programming to a certain extent, it is centered around the object-oriented paradigm making it impossible to program in other styles without constantly relying on object-oriented constructs. The example shown in listing 15 demonstrates a `"Hello, world!"` program written with Java. When choosing a procedural first approach, which goes hand in hand with explaining how algorithms work at the beginning of a course, topics like classes, objects and visibility modifiers should not be covered in order to reduce the amount of information presented to the students at once. Thus, when using Java, a course instructor is confronted by a dilemma: either handle these concepts as a black box, which may lead to some students believing that methods are always declared as `static` as we can observe every year at the final exam of our CS1 course, or explain everything at the beginning overwhelming the students with the amount of information which most of them will struggle to process.

```java
class HelloWorldExample {
    public static void main (String[] args) {
        System.out.println("Hello, world!");
    }
}
```

Listing 15: A *“Hello, world!”* program written with Java.

Since version 8, Java supports functional programming, however, again centered around object-oriented concepts. Lambda expressions are mapped to types defined by functional interfaces. Some of the generic interfaces included in the standard library are:

- `Function<T, R>` – represents a function which accepts one parameter and returns a value.
- `BiFunction<T,U,R>` – a function with two parameters.
- `Predicate<T>` – is a function which accepts one parameter and return a boolean value. A `BiPredicate` variant is provided too.
- `Supplier<T>` – returns a value.

- `Consumer<T>` – represents a procedure which accepts one parameter, a `BiConsumer` is also provided.

The standard library also includes non-generic variants of the above interfaces. The number of provided interfaces and the slight differences between them will not be easy to understand for beginner programmer students, especially the fact that these types, which they previously have learned are abstract types to be used as blueprints for objects, are now representing subroutines. Furthermore, the standard library lacks functional interfaces for subroutines with more than two parameters. Should the necessity for such a function arise, a functional interface has to be defined first, as demonstrated in listing 16. In this example, we attempt to implement a simple weather messenger using lambda expressions, but before we can do that, we have to define a functional interface which specifies a function with three parameters. We would have to define additional interfaces for a different number of parameters. Specifying a custom interface each time we need a function with more that two arguments is, in our opinion, a rather inflexible solution.

```java
public class CustomFunctionalInterfaceExample {
    public static interface TriFunction<T1, T2, T3, R> {
        R apply(T1 t1, T2 t2, T3 t3);
    }
    public static void main(String[] args) {
        TriFunction<String, Integer, Boolean, String>
            weatherMessenger = (location, temperature, celsius) -> {
                if (celsius)
                    return "Temperature in " + location + ": " + temperature + "°C";
                else
                    return "Temperature in " + location + ": " + temperature + "°F";
        };
        System.out.println(weatherMessenger.apply("Bayreuth", 14, true));
        System.out.println(weatherMessenger.apply("New York City", 68, false));
    }
}
```

Listing 16: An example of a custom functional interface in Java.

An alternative solution to that is to use currying, a common technique in functional programming, as demonstrated in listing 17. In this example, we use the already mentioned Java interface `Function`, which accepts only one parameter. However, if we return a function as a result, we can pass another parameter to the returned function, and so on. This way we can chain several functions and pass as many parameters as we like. However, this concept could be hard to understand for beginner students, especially if they are used to passing several arguments to a single subroutine at once.

```java
import java.util.function.*;

public class CurryingExample {
    public static void main(String[] args) {
        Function<String, Function<Integer, Function<Boolean, String>>>
            weatherMessenger = location -> temperature -> celsius -> {
                if (celsius)
                    return "Temperature in " + location + ": " + temperature + "°C";
                else
                    return "Temperature in " + location + ": " + temperature + "°F";
        };
        System.out.println(weatherMessenger.apply("Bayreuth").apply(14).apply(true));
        System.out.println(weatherMessenger.apply("New York City").apply(68).apply(false));
    }
}
```

Listing 17: Function currying in Java.

Yet again, both examples demonstrate the object-centric approach of Java. A functional interface may contain only one single abstract method [56] so that a lambda expression can be mapped to a variable of that type overriding the abstract method, i.e. functions are objects. In our opinion, Java is not a suitable language to teach functional programming and may even lead to false understanding of the core concepts of this paradigm. The output of both examples is:

```
Temperature in Bayreuth: 14°C
Temperature in New York City: 68°F
```

Since the establishment of Java, newer truly multi-paradigm general-purpose languages were developed like Scala and C#. They, however, have some of the same problems as Java, i.e. they are complex industrial grade programming languages with a large number of language constructs which can be overwhelming for beginner students. Furthermore, they allow to implement very different solutions for one problem which can impede the post assignment discussion if the explained solution and students' solutions are all different from each other.

In recent years, Python seems to be taking over as the leading educational programming language both in schools and at universities [59] due to its simplicity and attractiveness to students. Compared to the Java `"Hello, world!"` example in listing 15, the same program written in Python 3 looks like that:

```python
print("Hello, world!")
```

Studies [60] have shown, that students in general tend to make less errors (syntax, run-time and logical) using Python compared to Java. Python was designed with readability in mind [61] enforcing students to write well structured programs, which is not always the case with beginner programs written with C/C++

or Java. However, this may also lead to logical errors due to false indentation as demonstrated in an example in listing 18, an error often made by beginner students which we have observed first-hand in an introductory programming course performed with Python at our university [62].

```python
i = 0
while(i <= 5):
    print(i)
i = i + 1
```

Listing 18: An indentation error in Python.

Another issue of this language is the use of dynamic typing. While some consider static typing as redundant [63], it does help preventing type related runtime errors and demonstrates the importance of type systems, an important concept of an introductory programming course, on a lexical level. Furthermore, the authors of [64] have made negative observations regarding the dynamic behaviour of Ruby, a dynamically typed scripting language, when used as an introductory programming language. In our opinion dynamic typing is not suited for teaching programming. Not only can it lead to an increase of difficulty to find type related runtime errors, it could also cause students to underestimate the importance of type systems.

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

p1 = Person("Alice", 20)
p1.aeg = 25
print(p1.age)
```

Listing 19: Semantic field error caused by typographical mistake in Python.

Listing 19 demonstrates another source of errors that can be caused by the dynamic behaviour of Python and similar languages. In this example, we define a class Person with fields `name` and `age`. We follow on with initializing a `Person` object `p1` with the `name` Alice and `age` 20. Then, we want to change the attribute `age` to 25, however, we make a typographical error and write `aeg` instead of `age`. This adds a new field `aeg` with the value 25 to the object `p1`, the value of `age` remains 20. Such dynamic behaviour might be practical for experienced users, but beginners will struggle to understand why their program behaves not as intended due to such errors which are rather hard to find for inexperienced programmers.

In its core, Python is an object-oriented language with reference semantics. Listing 20 shows a small Python program with a simple two dimensional list. The list is initialized and one of its values is then set to `1`. The produced output is `[[1, 0], [1, 0]]` resulting from the underlying reference copy semantics. An inexperienced programmer would most likely expect `[[1, 0], [0, 0]]` in this case, which can be achieved by using Python's list comprehension mechanism.

```
lst = [[0] * 2] * 2
lst[0][0] = 1
print(lst)
```

Listing 20: Example of Python reference semantics.

Python is a simple language that supports procedural, object-oriented and functional programming [55], hence its raise in popularity in educational institutions. However, its dynamic type system, use of indentations to define blocks and reliance on implicit behaviour may lead to unnecessary errors and hide important concepts under a layer of implicity. Therefore, Python is in our opinion not that well suited as an educational language.

All of these languages are primarily based on the imperative style of programming and are therefore more suited to teach procedural and object-oriented paradigms. Recursion is one of the more difficult topics for the students, especially when it is used to implement repetition via loops. A lot of difficulties experienced by programming students can be attributed to a paradigm shift during a course [65], i.e. students have to abandon previously established mental models and have to learn alternative solutions. Some even consider teaching loops as harmful and rely on recursion instead, pointing out that students develop the ability to write code that can be automatically used in parallel infrastructures [66]. Thus the alternative approach to the usual *procedural-first* or *objects-first* is to teach *functional-first*.

Lisp, or rather its minimalistic dialects, e.g. Scheme or Racket, are usually used in programming courses utilizing this approach. Another reason to teach a functional language is the fact that a CS1 course is usually visited by a vastly heterogeneous group of students with a wide range of background knowledge, i.e. while some students have absolutely no programming experience, some have learned to program at school or as their hobby. These students are mostly familiar with imperative oriented languages like C and Java, meaning that using a functional programming language may slightly mitigate the discrepancy among the knowledge of the participating students.

As we can see, professionally used programming languages have some or all of the following limitations in the context of education:

- They were not developed with education in mind, some choices regarding the syntax and semantics of the language might be confusing to students.
- They include a large amount of non-orthogonal language constructs overwhelming the students.
- Some languages are designed for writing programs quickly with short but obscure syntax.

- They have evolved over time leading to language constructs with non-consistent syntax and deprecated features.
- Some languages may include cryptic error messages.
- Even though they are general-purpose languages, some of them were initially designed for rather specific tasks, limiting their usability as an educational language meant to teach a broad range of basic abstract concepts.

On the other hand, there are several points, which may or may not be considered as advantages when talking about these languages in the context of education:

- Students may be rather motivated to visit a programming course if the used language is relevant outside of that course [41].
- A professionally used language is usually supported by powerful IDEs, which are usually equipped with such helpful tools like debugging and syntax highlighting. However, professionally used IDEs are, just as the languages, not designed for the task of education and tend to be rather complex with an overwhelming amount of features. Furthermore, not all IDEs are provided free of charge, which is another important factor in the context of education.
- They are supported by a wide range of professionally written literature and documentation. A motivated student will have a good amount of information to tap into. A less motivated student will be most likely frightened by the amount of information and will instead rely purely on the filtered data presented in the programming course.
- Users may rely on a vast amount of standard and third-party open-source libraries. This increases the range of possible assignment scenarios: i.e. a language supported by a graphical library allows to design tasks concerned with the implementation of graphical user interfaces. This is however limited by the scope of the introductory course, which is usually centered on basic algorithmic problems which can be solved with available language constructs and a minimal amount of operations provided by standard libraries.
- Finally, such languages usually have a big follower community. Therefore, students can easily find a large amount of community created content in the context of a specific language: e.g. open-source projects and libraries, programming tutorials, blogs, forums, etc. However, by relying too much on the help from community, students may start to develop counter-productive habits like copying assignment solutions from the internet.

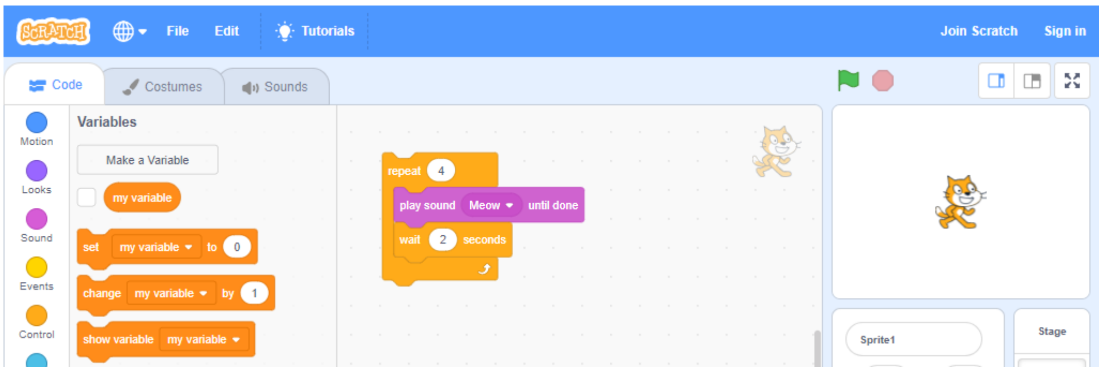


Figure 2: Scratch programming environment with a sample program.

3.2.2. Educational Languages

Whereas professionally used languages covered in the previous section are all text based languages, we will cover both visual and text based language in this section, since both types are heavily represented in the field of programming education.

Visual Languages

Scratch [67] and similar block based visual programming languages, e.g. Alice [68] and Blockly [69], are widely used in programming education at schools. As demonstrated in figure 2, Scratch is supported by an online platform, which not only consists of the execution environment, but also contains countless sample projects, tutorials and guides targeted both at programmers and teachers. The language offers the possibility to quickly assemble simple programs without syntactic errors, which is in our opinion one of its problems. An educational language should be capable to demonstrate common mistakes and ways to fix them to the students and, therefore, prepare them to some realities of working with industrial grade general-purpose languages. Scratch and similar languages are rather limited to be used as an effective tool at university level computer science education. Scratch is primarily focused on building algorithms from blocks, i.e. while it does certainly teach algorithmic thinking and specific programming patterns, it will not prepare students for advanced courses utilizing regular text based languages. Furthermore, these languages are not well suited to teach functional and object-oriented programming. Finally, it is impossible to write syntactically incorrect programs using these environments, however, error handling is an important part of programming education. Nevertheless, they can be effectively used at earlier stages of education. Furthermore, they may draw interest from students, who would otherwise find programming a rather boring practice.

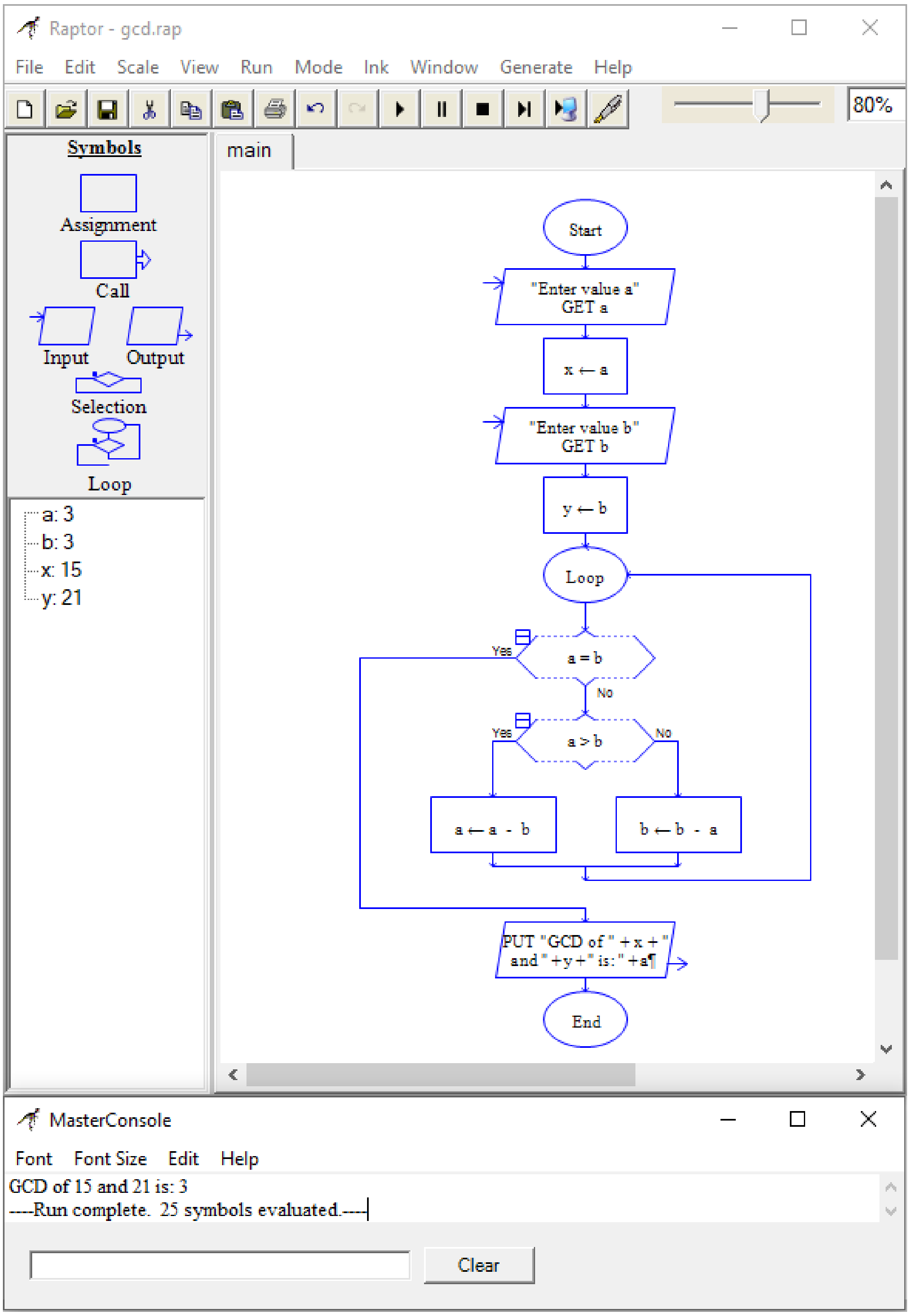
A less colourful but perhaps a more useful tool in the context of a CS1 course is RAPTOR [70], a visual programming environment which uses executable flow charts. Figure 3 demonstrates a GCD program written with this tool. It should be noted, that RAPTOR's flow charts do not fully represent hand drawn control flow diagrams, i.e. the constructs in RAPTOR flowcharts have a rigid structure and strongly resemble the blocks of block based languages. The environment highlights each node in the chart and tracks the values of the variables during the execution giving immediate visual feedback. Furthermore, it allows to pause and execute the program step by step. This tool can be used at the beginning of a CS1 course when introducing algorithms which would create a more visible link between abstract algorithms and actual programs.

This tool also offers an object-oriented mode which relies on UML class diagrams [71] to represent object-oriented constructs and flow charts to implement method functionality, an example is given in figure 4. However, this mode requires knowledge of Java or C# syntax. Furthermore, while RAPTOR's flow charts rely on type inference, users are required to enter data types for fields, methods and parameters in the UML class diagrams resulting in a mix of static and dynamic typing. Similarly to previously mentioned block based teaching environments, this environment does not prepare students to work with text based programming languages. Nevertheless, it can be used as an entry tool to demonstrate specific concepts before moving on to a text based language or as a complementary tool in a course which is mainly using Java or C#.

There are studies [72][73] which prove that visual languages improve performance of weaker students and keep them motivated. However, the focus on these languages lies on procedural programming, support for object-oriented and functional programming is rather limited. While block based and other languages with a strong focus on visual representation may not be feasible in a CS1 course due to certain limitations, they offer interesting concepts which may be implemented in form of libraries in a classic textual based programming language. These languages are better reserved to schools or as introductory tools at the universities, however not as the main language in a CS1 lecture.

Text Based Languages

As already mentioned, Pascal was one of the first programming languages intentionally developed for education and also one of the most successful. One of its main advantages is that it is a simple language specifically designed to teach structured programming [41], its syntax was designed to be intuitively clear [55]. Supported by well documented development environments at the time it had a massive success not only in educational, but also as a professionally used language, which led to a development of complex dialects, since the base language was not sufficient for this task. However, the emergence of object-oriented programming has led to the decline of Pascal's popularity and its subsequent retirement [41]. Standard version of Pascal was purely procedural, its dialects could not compete with C++ and Java and the supporting tools were becoming obsolete. Section 2.2 includes an example program written with Pascal.



```
a: 3
b: 3
x: 15
y: 21
```

```
GCD of 15 and 21 is: 3
----Run complete.  25 symbols evaluated.----
```

Figure 3: RAPTOR programming environment with a sample procedural program.

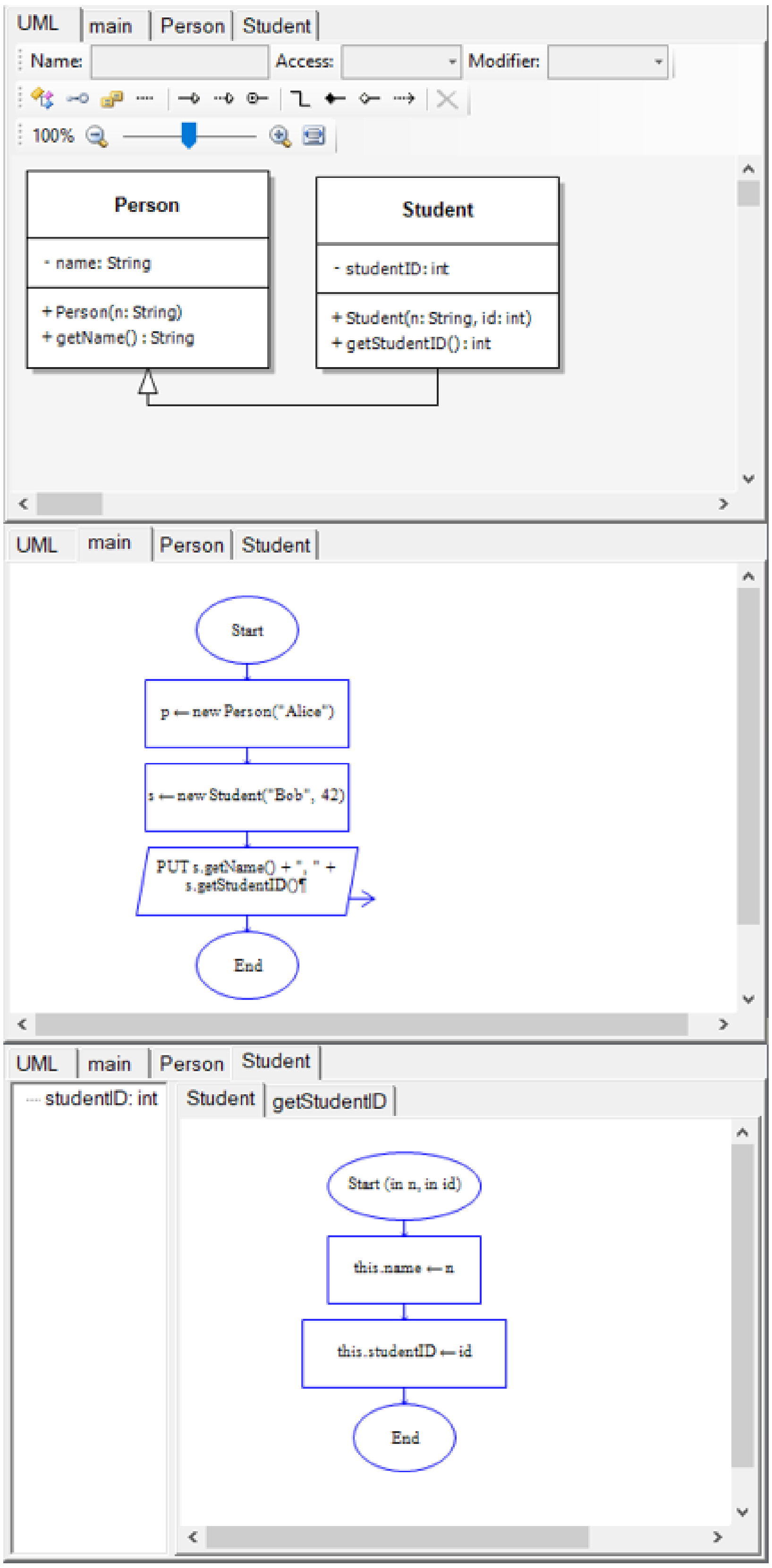
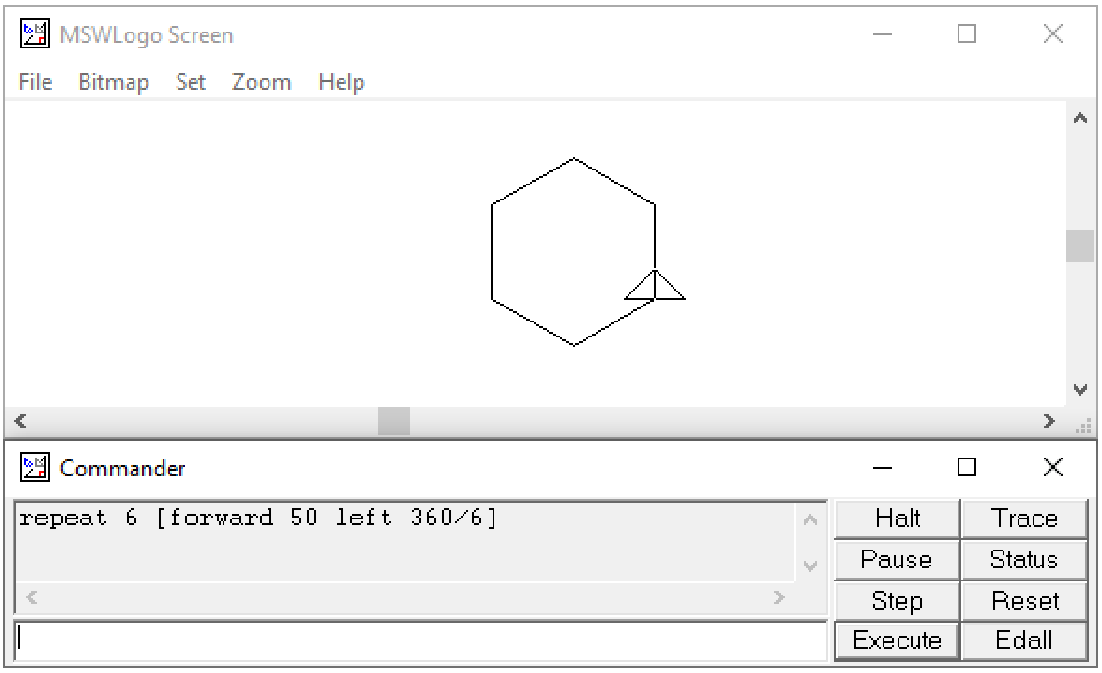


Figure 4: Example of a RAPTOR object-oriented program.



```
repeat 6 [forward 50 left 360/6]
```

Figure 5: Logo programming environment with a sample program.

Logo [74] is another educational programming language, which has actually appeared three years prior to Pascal, but is based on Lisp and is centered around Turtle Graphics, which has been successfully used in education [75] and has been since ported to other languages, e.g. Python and Java. In our opinion, it is well suited to teach topics like control flow, subroutines and recursion while keeping the students motivated at the same time. In a sense, Logo is not merely a language, but rather an educational environment. It consists of a command view, where the user is supposed to write the program, and a drawing view with the graphical representation of Turtle's movements (see figure 5). One of the design goals of Logo were informative error messages which should reduce the frustration amongst beginner programmers when confronted with programming errors. However, the strong focus on the Turtle Graphics system and the more abstract and declarative nature compared to Pascal may have been the reasons, why this language has not gained as much popularity at the time. Similarly to block based educational languages it is too limited to be effectively used as the main programming language throughout an entire CS1 course.

One of the more recently developed educational programming languages is Grace [76]. The principles of this language [58] are similar to the requirements which we have defined for our own language in many aspects (see chapter 4). For instance, this language is using garbage collection and is designed to be platform independent. It is meant to offer clear and easy-to-understand syntax and semantics, its constructs can be introduced incrementally according to the current progression in the corresponding course. The language supports the three main programming paradigms: procedural, functional, and object-oriented. Grace, however, is object-oriented first and foremost, i.e. everything is an object. For example, `true.not` is a valid expression which is evaluated to false, same as `! true`. The language offers both static and dynamic typing. Grace uses the operator `=` in declarations to initialize variables, `:=` in assignment statements and `==` in equality check expressions.

Listing 21 demonstrates Grace methods, which represent subroutines in this language whether they are contained in a class or not. As we can see, no explicit return statement is required, furthermore, the language allows rather unorthodox subroutine declarations, e.g. the method `divide(a) by(b)`, which might be misleading, since the call of this operation looks like two subsequent method calls to a person with prior programming experience in other languages.

```
def x = multiply(5,7)
print "{x}"
def y = divide(48) by(6)
print "{y}"
print "{pi}"

// declared with multiple parameters
method multiply(a, b) {
    a * b
}

// declared with multiple names
method divide(a) by(b) {
    a / b
}

// declared with no parameters
method pi {
    3.141593
}
```

Listing 21: An example of a Grace program taken from [77].

While the language does have less constructs compared to Java or C++, some of them are not orthogonal to each other, e.g. the language offers normal `if` and `match ... case` statements, as well as `while` and `for` loops. Mixing static with dynamic type systems will produce inconsistent solutions amongst students. If it is not absolutely necessary to do something, e.g. specify types in variable declarations in a mixed static/dynamic type system, students will not do it and therefore will not understand its meaning. Finally, using brackets to specify blocks is not optimal in our opinion, missing brackets is a common beginner mistake [78] which requires a time consuming *"bracket counting procedure"* to fix it.

Quorum [79] is another language specifically developed to be easily understandable for beginners and to be used in education. One of the authors' goals was to design a programming language which could also be used by visually impaired students. Therefore, the syntax of the language is oriented on naturally spoken languages which should make it easier to understand compared to C-style syntax. The loop construct, for example, can be formulated as `repeat while CONDITION`, `repeat until CONDITION` or `repeat N times`. The authors propagate the use of static typing based on knowledge gained from empiric studies [80], however, their language offers both static and dynamic typing. The authors have chosen `=` as both equality check and assignment operator as demonstrated in listing 22, which is a rather questionable design decision in our opinion. Quorum supports procedural and object-oriented programming paradigms. The language can be executed both in an online browser based runtime environment as well as in a specifically developed desktop IDE.

Allowing to use both static and dynamic typing, using operators with different meanings depending on their context, such as `=` for assignments and equality checks and `+` for numeric additions and string concatenations, as well as lack of support for functional programming are, in our opinion, problem of this languages when talking about programming education.

```
integer i = -1
repeat while i < 2
   if i = 0
      output "zero"
   elseif i < 0
      output "less than zero"
   else
      output "greater than zero"
   end
   i = i + 1
end
```

Listing 22: An example of a Quorum program.

Pyret [81] is a declarative educational programming language still in development, but already in a quite advanced state. An online interpreter facilitates accessibility to this language. The syntax of the language is partly inspired by Python, e.g. for functions and lists, however, blocks are terminated with a keyword `end`, which should eliminate errors due to wrong indentation while keeping the overall readability of Python. Another interesting decision is the different use of operators for variable initialization, equality checks and assignments. Listing 23 demonstrates the use of these operators in a `check` block, another interesting feature of this language. When a variable is declared, it is assigned a value with an operator `=`, however, `:=` must be used for later assignments. The `is` statement is used to test assertions in a `check` block, i.e. Pyret natively supports a testing environment which could be used to teach testing practices of software development right from the beginning. The authors even encourage to write tests along with the actual code. At the time of writing, this language offers dynamic typing only, however, authors are working on the implementation of a static type checking system. Pyret does not differentiate between integer and floating point numerical representations and uses a single type `Number` instead.

Testing the program while writing it is a good programming practice, and one that has to be encouraged. But in our opinion, testing behaviour can be implemented simply via `if`-statements or as a separate library if a more powerful solution is required. Including entire language constructs with several keywords reserved to testing code introduces complexity to the language which could otherwise be avoided. Furthermore, it introduces language constructs that are not present in other common languages, which might lead to problems in the future when transitioning to these languages. Dynamic type system is not well suited for beginner programming education, however, according to the authors static typing is in development. Finally, Pyret is more functional in nature and is, in our opinion, less suited to teach object-oriented programming.

```
check:
    var x = 10
    x is 10
    x := 15
    x is 15
end
```

Listing 23: Pyret check blocks and variables.

Another language specifically developed for education is RESOLVE [82]. One of its interesting features is the built in verifier which should improve software reliability. It allows to define pre- and postconditions when declaring operations, as demonstrated in listing 24, wherein the operation `Increment` is declared with a precondition which states that the passed parameter must be less than the maximum value supported by the type `Integer` and the postcondition which states that the value of `i` after the execution of the operation is its initial value plus 1. The language provides various parameter modes, e.g. the keyword `reassigns` in listing 24 implies that the parameter will be reassigned to a new value. However, the differences between parameter modes are not entirely clear, e.g. the modes `reassigns`, `alters`, `replaces` and `updates` essentially all mean that the value of the parameter will be changed in one way or another. Similarly, the modes `evaluates` and `preserves` indicate that the values of the parameters are not changed after the execution. The language allows to use shorter versions of keywords, e.g. `Oper` instead of `Operation`.

```
Operation Increment(reassigns i: Integer);
    requires i + 1 <= max_int;
    ensures i = #i + 1;
```

Listing 24: An operation declaration in RESOLVE [82].

An example of a `Main` procedure in shown in listing 25. Here we can see that the language is relying on keyword pairs to delimit blocks similarly to most other educational languages presented in this section. The `while` loop includes various clauses in addition to its condition: i.e. the `maintaining` clause specifies an additional condition while the `decreasing` clause decrements the value of the variable `temp`.

```
Facility Example4;
    uses Std_Integer_Fac;

    Operation Main();
    Procedure
        Var temp, temp2, temp3: Integer;
        temp2 := 2;
        temp3 := temp2;
        temp := 5;
        While(temp > 0)
            maintaining temp3 = temp2;
            decreasing temp;
        do
            Write_Line(temp3);
            Write_Line(temp2);
            Write_Line(temp);
        end;
    end Main;
end Example4;
```

Listing 25: An example of a Main-procedure in RESOLVE [82].

This language is using a static type system. It is translated to Java and can be executed in the Eclipse IDE or in the command line, in the latter case the user has to manually translate the RESOLVE program first. The language can be used to teach procedural and object-oriented programming, but lacks support for functional programming. However, the biggest issue of this language, is in our opinion its verification mechanism and additional constructs similar to those demonstrated in the `While` loop in listing 25. For example `maintaining` defines an additional condition, which could just as well be put into the condition of the loop, while the `decreasing` clause is normally implemented via a manual decrement of a variable. It should also be noted, that the language does not specify a complementary `increasing` clause, which further adds to the overall confusion.

As we can see, several attempts have been made to implement an educational programming language. Let us summarize their advantages and commonalities:

- These languages offer less language constructs compared to professionally used languages.

- Some languages are supported by easily accessible and simplistic execution environments and tutorials.
- Most text based languages use keyword pairs (`if ... end`) instead of curly brackets (`if { ... }`) common in languages inspired by the C syntax.
- The assignment operator is usually `:=` instead of `=`.
- Using such languages may bridge the gap between students with prior programming experience and absolute beginners.

However, since these languages, except for maybe Pascal and Scratch, are far less popular around the world, the amount of provided literature, examples and tutorials is rather limited compared to professional languages. Furthermore, using an unknown language may deter some students from participating in a course.

3.3. Tools

When choosing a programming language, course designers have to consider whether this language is sufficiently supported by tools that are qualified to be used in education, the chosen programming environment can either facilitate or impede students' learning process [44]. In the previous section we have already mentioned visual programming environments like Scratch, which are oriented towards beginner programmers. These environments as well as the languages are, however, rather limited to be effectively used in a CS1 or CS2 course and will not be discussed any further. The focus of this section lies on tools designed to complement text based programming languages.

When teaching procedural or functional programming a simple text editor and a compiler or interpreter may suffice. Some text editors, e.g. Notepad++ [83], may even have rudimentary programming support tools like syntax highlighting and automatic formatting. Web based execution environments offer the best availability, they do not require an installation and are easy to use, however, they are rather limited in regards to the provided tools similarly to text editors. Therefore, when dealing with object-oriented programming, multiple source files or when additional tools like a debugger are required, an IDE (integrated development environment) becomes a necessity [84].

The programming environment used in education must fulfil following criteria [84][55]:

- **Easy to use** – a difficult environment takes more time to explain prior to starting programming as well as during exercises when problems caused by its complexity will inevitably arise. The beginner students are already confronted with the two problems of finding an algorithm and translating it into a program. Adding an environment which is difficult to understand

and use may further discourage them from continuing their programming training. A difficult environment may be a professionally used IDE with an overwhelming amount of functionality or a simple combination of a text editor and a command line compiler which, although it needs the bare minimum of tools and therefore installation and configuration steps, requires students to perform manual compilation steps each time they need to test their program as well as an input of a more complex execution command compared to the activation of a single "run" button.

- **Available** – proprietary environments must be provided by the educational institutions, otherwise most of the students may not be able to effectively take part in a programming course. Free IDEs have therefore a huge advantage from this point of view, especially if they can be installed on the majority of operating systems.

- **Integrated tools** – this criteria eliminates the use of a text editor combined with command line compilers or interpreters as well as web based execution environments. At the very least, an educational IDE should include a text editor with syntax highlighting, helpful error messages, a console interface, a debugging tool, and a project manager. The user should be able to run and debug the program directly in the environment. Additional tools like automatic formatting, refactoring, shortcuts and quick-fix providers for compiler errors may be helpful but not necessary. It is important to keep the IDE simple while providing all the necessary functions.

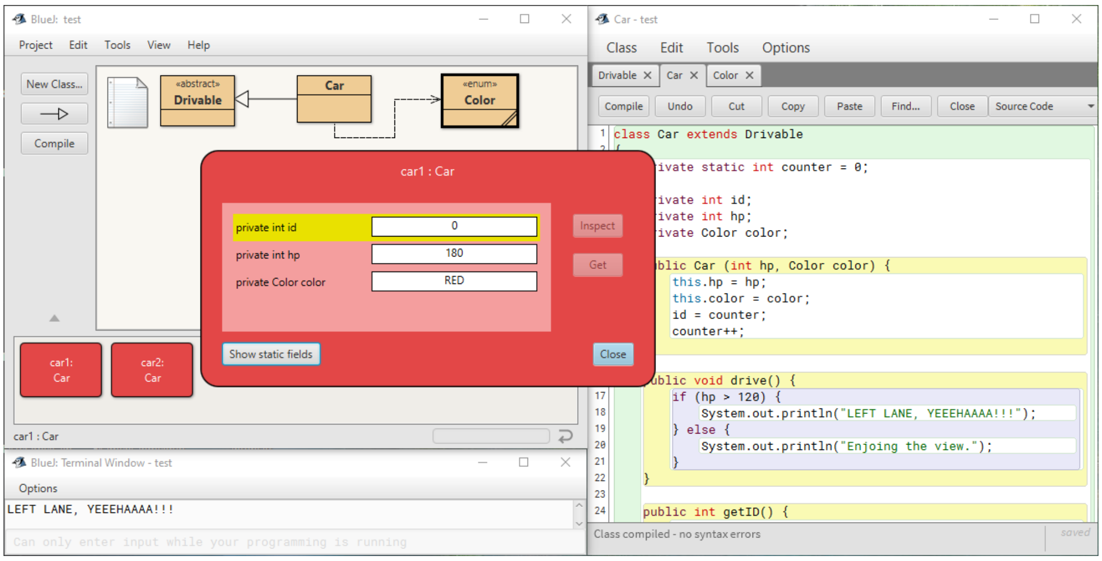
- **Code reuse** – the environment should provide a browser for standard libraries as well as libraries written by the user to encourage users to use libraries and reuse their own code.

- **Learning support** – the IDE should support specific learning techniques, e.g. stepwise execution of code or allow interaction with objects. Additional visualization of code by means other than text (e.g. diagrams, tree-like program overview providers) may enhance understanding of a program.

- **Object-support** – objects should be represented as independent entities within an IDE enabling the users to interact with them unlike in traditional programming environments where interaction with objects is reserved entirely to the source code of the program.

The authors of [84] mention an additional criteria: **group-support** which should allow groups of students to work on separate parts of a single project which is, in our opinion, irrelevant in the context of CS1 and CS2 courses, therefore, we do not consider this criteria as essential in our case.

The programming environment BlueJ [85] which is specifically designed to teach object-oriented programming with Java, was developed in concordance to



private int id	0
private int hp	180
private Color color	RED

```
class Car extends Drivable
{
ivate static int counter = 0;

ivate int id;
ivate int hp;
ivate Color color;

blic Car (int hp, Color color) {
    this.hp = hp;
    this.color = color;
    id = counter;
    counter++;

ublic void drive() {
    if (hp > 120) {
        System.out.println("LEFT LANE, YEEEHAAAA!!!");
    } else {
        System.out.println("Enjoing the view.");
    }
}

public int getID() {
```

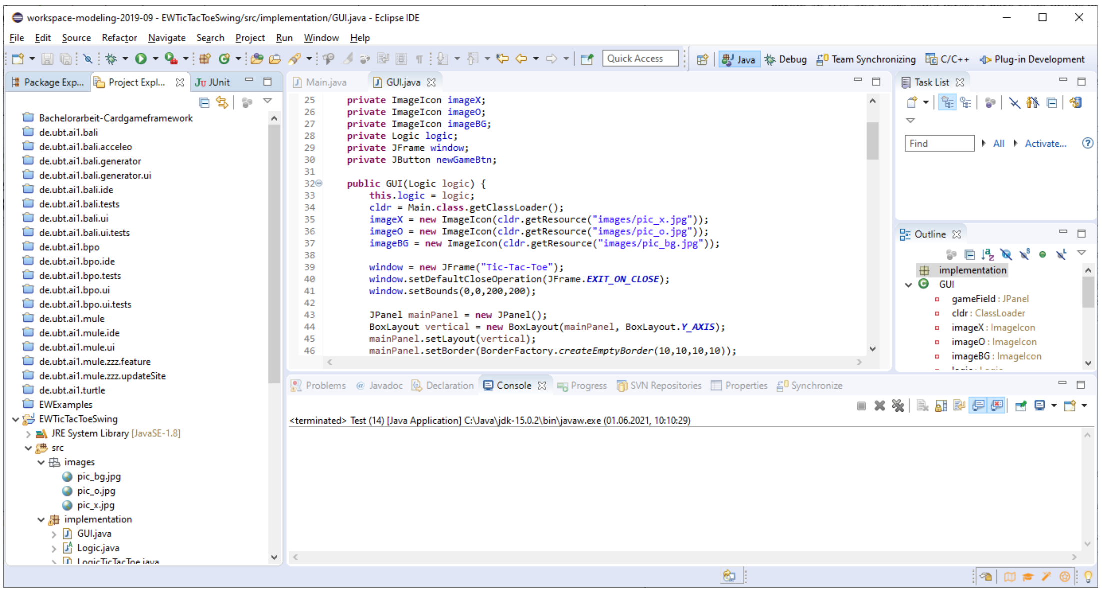
```
LEFT LANE, YEEEHAAAA!!!
Can only enter input while your programming is running
```

Figure 6: BlueJ educational IDE.

these criteria. This tool offers a very simple, intuitive, and at the same time informative graphical user interface shown in figure 6. The project view shows classes and other user defined types as a simplified class diagram. The classes are edited in the text editor, users do not actually see the corresponding Java source files unless they open the project folder in the file system browser. The blocks are marked with different colours in the editor helping to locate potential parenthesis errors. Static methods can be executed by right clicking a class in the diagram view. Objects can be created in the same way. Users may inspect created objects to check on their current state as well as execute non-static methods directly interacting with the objects. The IDE also offers tools not demonstrated in figure 6, for example a debugger, a content assist provider and an automatic formatter.

Professionally used IDEs can also be employed in education, depending on the language the only choice may sometimes be between an industry grade IDE or text editors, in which case an IDE is preferable. Such environments have mostly the same advantages and disadvantages as professionally used programming languages, i.e. they are rather complex and have an overwhelming amount of features but at the same time are well documented and have a large community built around them.

Figure 7 shows the Java perspective of the Eclipse IDE [86], a free open-source development environment with a large amount of plug-ins enabling support for other programming languages and tools. Compared to BlueJ, the user interface of Eclipse is filled with a much bigger number of menus, tabs and buttons, whose functionality is not understandable on the first glance. This may lead to students being overwhelmed by the sheer amount of advanced features and options. This IDE does not offer an interactive mode like the one provided by BlueJ since



```
    private ImageIcon imageX;
    private ImageIcon imageO;
    private ImageIcon imageBG;
    private Logic logic;
    private JFrame window;
    private JButton newGameBtn;

    public GUI(Logic logic) {
        this.logic = logic;
        cldr = Main.class.getClassLoader();
        imageX = new ImageIcon(cldr.getResource("images/pic_x.jpg"));
        imageO = new ImageIcon(cldr.getResource("images/pic_o.jpg"));
        imageBG = new ImageIcon(cldr.getResource("images/pic_bg.jpg"));

        window = new JFrame("Tic-Tac-Toe");
        window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        window.setBounds(0,0,200,200);

        JPanel mainPanel = new JPanel();
        BoxLayout vertical = new BoxLayout(mainPanel, BoxLayout.Y_AXIS);
        mainPanel.setLayout(vertical);
        mainPanel.setBorder(BorderFactory.createEmptyBorder(10,10,10,10));
```

```
<terminated> Test (14) [Java Application] C:\Java\jdk-15.0.2\bin\javaw.exe (01.06.2021, 10:10:29)
```

Bachelorarbeit-Cardgameframework
de.ubt.ai1.bali
de.ubt.ai1.bali.acceleo
de.ubt.ai1.bali.generator
de.ubt.ai1.bali.generator.ui
de.ubt.ai1.bali.ide
de.ubt.ai1.bali.tests
de.ubt.ai1.bali.ui
de.ubt.ai1.bali.ui.tests
de.ubt.ai1.bpo
de.ubt.ai1.bpo.ide
de.ubt.ai1.bpo.tests
de.ubt.ai1.bpo.ui
de.ubt.ai1.bpo.ui.tests
de.ubt.ai1.mule
de.ubt.ai1.mule.ide
de.ubt.ai1.mule.ui
de.ubt.ai1.mule.zzz.feature
de.ubt.ai1.mule.zzz.updateSite
de.ubt.ai1.turtle
EWExamples
EWTicTacToeSwing
JRE System Library [JavaSE-1.8]
src
images
pic_bg.jpg
pic_o.jpg
pic_x.jpg
implementation
GUI.java
Logic.java

implementation
GUI
gameField : JPanel
cldr : ClassLoader
imageX : ImageIcon
imageO : ImageIcon
imageBG : ImageIcon

Figure 7: Eclipse IDE with the Java perspective.

programs written with Java (as well as in most other languages) do require a `main` subroutine which acts as an entry point into the program. In this specific case, Eclipse actually enforces students to learn the correct way of writing programs, which can then be executed on the Java virtual machine outside of the Eclipse environment. Furthermore, using a professionally used IDE may raise students' confidence and sense of purpose [87]. Eclipse is highly customizable, it allows to design custom perspectives which contain only the views and buttons that are necessary in an introductory course reducing the amount of features presented to the students. While it is not the optimal solution, we have decided to use Eclipse as the IDE meant to support MuLE for the time being. It allowed us to concentrate more on the implementation of the language and quickly deliver a working prototype which we could test with the target audience and reuse Eclipse's infrastructure to provide tool support for MuLE.

Other professionally used IDEs, such as Visual Studio, have similar advantages and disadvantages as Eclipse. However, in case of proprietary IDEs the question of costs and licensing must also be considered when choosing a programming environment for an introductory course.

The information discussed in this section will be further relevant in chapters 4 and 7.

3.4. Summary

The choice of an introductory programming language depends on a lot of different factors. The intention of the programming course is definitely one of the most important ones: if the course is built around algorithmic or mathematical concepts, then a language of choice should be suited to teach procedural or functional paradigm respectively. Object-oriented programming should be taught using an object-oriented language. Multi-paradigm languages make this choice much easier, however, even in their case some are better suited for specific tasks than others. Some languages are more complex (C++ or Java) than others (Python). Using easier languages may help students to learn basic programming concepts, however, such languages may hide important concepts under a layer of abstraction. Using a language specifically designed for education may facilitate learning process, however, the use of such languages is mostly limited to CS1 and CS2 [45]. Finally, the language should have sufficient tool support.

In this chapter we have given an overview over the purpose of introductory programming education as well as programming languages and tools used in this context, their advantages and shortcomings. We use this information to define the requirements as well as the resulting design decisions for MuLE in the subsequent chapter. Additionally, a comparison of the text based languages discussed in this chapter to MuLE according to these requirements is presented in section 9.2.

4. Requirements and Design Decisions

In the previous chapters we have discussed what programming is and which limitations are present in the programming languages used in education. Based on this information, we will summarize the requirements for MuLE in the next section. Section 4.2 presents the design decisions that were derived from these requirements and demonstrates program snippets concentrating on specific language constructs and small programming examples based on specific paradigms. Finally, section 4.3 demonstrates multi-paradigm programming with MuLE based on the implementation of a dictionary lookup algorithm that was already prevalent in section 2.

4.1. Requirements

This section presents the general requirements for MuLE [88].

Requirement 1 *[Easy to learn] Students without prior programming experience should have no difficulties understanding and learning how to use MuLE.*

The focus of the language is on teaching programming [54], i.e. it should be capable to convey important programming concepts in a clear and comprehensible way without relying on unnecessary information. Important concepts, e.g. type system or referencing mechanism, should be clearly represented in a non-implicit way. At the same time, the language should avoid low-level concepts that represent direct interactions with the machine such as explicit manual memory-management [55][65]. Finally, the language should not be difficult to use, students should develop a feeling of competence and liking towards programming [49].

Requirement 2 *[Non-cryptic syntax] The syntax of the language should be clear and easy to read.*

The focus of the language should not be on short syntax with convenient shortcuts, the ability to write programs quickly or on performance [54]. The statements and expressions should be structured in such a way, which would facilitate verbalizing them [42], i.e. the pronounced form of the language construct should not deviate far from the written form making it easier to build a mental connection between the information spoken by the instructor and the program code seen by the students. The structure and the syntax of the language should not be too different from modern high-level GPLs to facilitate future transitioning to these languages [55][53].

Requirement 3 *[Minimal number of language constructs] MuLE should offer only the minimal number of necessary language constructs.*

The smaller number of language constructs implies less effort to learn how to handle the language. This includes built in data types, constructs for type and subroutine declarations, control flow, statements, expressions, operators, etc.

Requirement 4 *[Orthogonality] The language constructs should be orthogonal [89] to each other.*

Orthogonal constructs mean that the students should know exactly when to use which construct. At the same time, the language should not suffer from a lack of constructs in which case users may have to rely on complex workarounds, i.e. there should be a balance between orthogonality and a reasonable supply of functionality. Reducing the number of constructs to a bare minimum will result in programs that are hard to read and understand and will require a larger number of standard library functions [90].

Redundant or synonymous constructs such as various kinds of loops or conditional statements can be confusing to novices [91]. Feature multiplicity, for example several different syntactical ways to increment a variable, should be avoided in order to prevent confusion amongst students caused by examples with same semantics but different syntax [23]. Syntactic homonyms, e.g. operators and keywords with different meanings depending on the context or overloading of operators and subroutines in general, should also be avoided [90].

Requirement 5 *[Incremental introduction] It should be possible to introduce the programming concepts incrementally [55].*

Basic language constructs should not require knowledge of the more advanced concepts. This way constructs can be introduced one by one without risking to overwhelm students with an unnecessary large amount of information.

Requirement 6 *[Build upon present knowledge] MuLE should build upon already present knowledge of programmer novices [91], such as known mathematical notations and operations.*

Any knowledge that can be reused does not require students to process and learn new information allowing to concentrate on new concepts [54]. Moreover, language constructs should not contradict already present knowledge which would require students to unlearn it before assimilating new facts [90].

Requirement 7 *[Multi-paradigm language] MuLE should include the three main programming paradigms: procedural, functional and object-oriented [92].*

All main programming concepts of these paradigms (see chapter 2) should be represented in the language. Logic programming should not be included, at least not directly, due to its very different, non-procedural, nature compared to the other three paradigms. It's constructs will not interact well with the constructs of the other paradigms resulting in a language built within another language which may as well be simulated by a library.

Requirement 8 *[No forced object-orientation] It should be possible to write procedural or functional programs without relying on concepts of object-oriented programming.*

Java represents a negative example for this requirement, as demonstrated in section 3.2.1. Furthermore, relying on complex object-oriented concepts to demonstrate more basic procedural or functional concepts contradicts the requirement of incremental introduction.

Requirement 9 *[Clear execution semantics] The language should have clear logic and semantics.*

The students should know exactly what will happen when a specific statement is executed or an expression is evaluated. Implicit behaviour and silent state alterations should be avoided. Error messages should be easy to understand and not rely on technical or scientific jargon [54]. When possible, error messages should offer advices or solutions to the problem [91].

Requirement 10 *[Explicit static type system] The language should rely on explicit static typing to prevent type related runtime errors.*

Runtime errors are more difficult to fix than compile time errors. Furthermore, dynamic type systems can lead to unexpected behaviour, e.g. semantic errors resulting from implicit type conversions or from missing explicit type checks, that may not be perceived by novices during the execution of a program.

Requirement 11 *[Data abstraction] MuLE should include mechanisms of data abstraction.*

The language should allow to abstract certain implementation details, e.g. internal functionality of a library or a specific type. This may range from abstracted representations of basic primitive types, e.g. decimal representation of numbers (which is standard in high-level programming languages anyway), to explicit abstract data types acting as super types in object-oriented programming and explicit visibility modifiers.

Requirement 12 *[Polymorphism] Polymorphic operations can be applied to data with varying types.*

While this concept is inherently present in dynamically typed languages, specific mechanisms must be used to apply this concept in statically typed programming languages. There are two distinct approaches to polymorphism. Subtype-polymorphism allows an operation to be used with a specific type and any of its subtypes. Another approach is parametric polymorphism wherein a type or an operation declaration is parameterized by an unspecified formal type which can be substituted by any actual type at a later stage.

Requirement 13 *[Modularity] MuLE should offer mechanisms to separate programs into reusable units [51][91].*

Users should be able to write and import their own libraries. It should be therefore possible to separate programs into several compilation units.

Requirement 14 *[Standard libraries] The language should provide standard libraries with a manageable number of necessary operations.*

Operations such as input and output are required from the very beginning, for example in a simple `"Hello, world!"` program. The intention and the semantics of the provided operations should be easily understandable by novice programmers.

Requirement 15 *[Tool support] The language should be assisted by adequate tool support [91].*

An IDE with a debugger which allows stepwise execution of the program can facilitate explanation and understanding of imperative concepts. The programming environments should be appropriate for the target audience, i.e. beginner programmers who have no experience with such tools. Both the language and the environment should be easy to install and be platform independent [53]. More on the requirements for the provided tools in section 3.3.

4.2. Design Decisions

In the previous section we have laid out the general requirements for our language. This section presents the concrete design decisions we have made when implementing MuLE, as well as short code examples to demonstrate these decisions. The actual rules and implementation of these design decisions are given in chapter 5.

4.2.1. General Decisions

Before we focus on paradigm specific constructs we have to discuss the fundamental design of MuLE.

Language Architecture

Procedural programming was chosen as the base platform for MuLE, upon which the language was then expanded with object-oriented and functional language constructs which also allows incremental introduction of these concepts in a programming course based on *procedural-first* approach (requirement **5 – Incremental introduction**). This has allowed us to test this language fairly early in the development in a live environment with the target audience. Since the language has to offer a minimal number of orthogonal constructs (requirement **4**), this also means that procedural constructs are reused when programming in an object-oriented or functional way (see figure 8). It is, therefore, not possible to write pure object-oriented or functional programs the way Smalltalk or Haskell do.

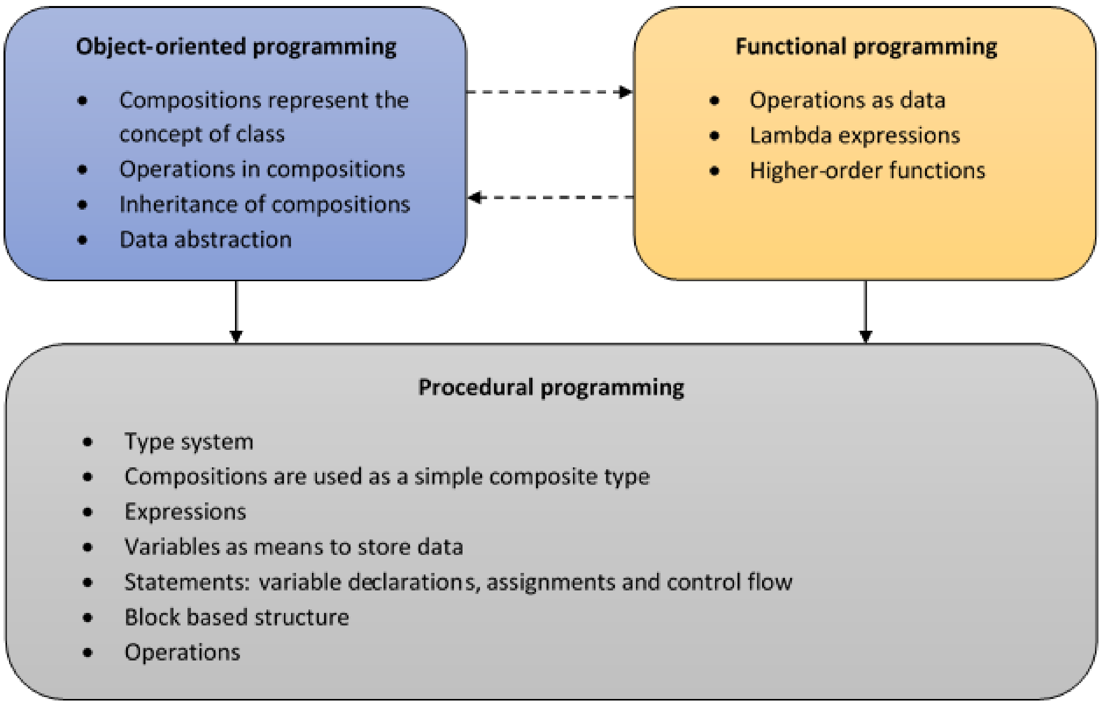

Object-oriented programming
Compositions represent the concept of class
Operations in compositions
Inheritance of compositions
Data abstraction
Functional programming
Operations as data
Lambda expressions
Higher-order functions
Procedural programming
Type system
Compositions are used as a simple composite type
Expressions
Variables as means to store data
Statements: variable declarations, assignments and control flow
Block based structure
Operations

Figure 8: Abstract representation of MuLE's architecture.

In a nutshell, functional and object-oriented language constructs are relying on procedural constructs. Functional and object-oriented constructs can be used in conjunction with each other facilitating a multi-paradigm style of programming. An example of a multi-paradigm program implemented with MuLE is presented in section 4.3. An explanation of the constructs mentioned in figure 8 will be given in the following subsections.

Program Structure

A simple program can be contained in a single compilation unit stored in a file. However, larger programs can be separated into several compilation units with one of them being the main one. The main unit must contain the entry point while the others must be explicitly imported in the main unit so that their contents can be accessed. Therefore, an import instruction is required and the imported units, let us call them libraries, have to be identifiable by their name. For the sake of consistency, all units, both the main unit and the libraries, are given explicit keywords (`program` and `library` respectively) followed by identifiers in the source code. The identifiers must be equal to the file name in order to minimize the effort to locate and open the necessary file in the programming environment.

Implicit imports are not allowed. By default, only those program elements can be used, that are defined within the same compilation unit. As already mentioned, users have to explicitly import libraries with an import instruction. Imported elements can then be accessed via their qualified names, e.g. `IO.writeString-("Hello, world!")` represents invocation of the operation `writeString(arg : string)` from the library `IO`. For the sake of consistency, we have allowed to use qualified names to refer to program elements from within the same compilation unit, e.g. we can refer to a type declaration in a program unit from the main procedure via its qualified name, which is optional. Finally, declarations of types and operations can be hidden in a library by using the explicit visibility modifier `private` allowing to prohibit user access to elements that are only relevant for the internal functionality of a library. This is again optional, declarations without the modifier private are exported by default.

The entry point is a structured block of statements, i.e. a main procedure, meaning that loose statements interspersed between subroutines and type declarations are not allowed. Arguments which could be passed to a main procedure are rarely used in the context of introductory programming, moreover, they are often confusing to the beginner programmers since they do not understand where the values should come from, therefore, we have decided not to include them. As already mentioned, libraries are not allowed to have a main procedure.

Separating a program into several importable compilation units fulfils the requirement **13 – Modularity** and allows to fulfil the requirement **14 – Standard libraries** (more on that in chapter 6). Explicit import mechanism of standard and custom libraries as well as having a single entry point in a program contribute to the requirements **9 – Clear execution semantics** and **1 – Easy to learn**. Preventing export of specific type and operation declarations via an explicit visibility modifier allows to abstract implementation details of a library (requirement **11 – Data abstraction**), furthermore, the optionality of that modifier allows to introduce the concept of information hiding at a later stage (requirement **5 – Incremental introduction**).

Syntax

MuLE's syntax should be readable, able to clearly convey the intended meaning and at the same time not deviate far from the syntax of other contemporary programming languages (requirement **2 – Non-cryptic syntax**). We have decided to introduce each declaration with a corresponding keyword: `program`, `library`, `type` (further specified as `composition` or `enumeration`), `operation`, `attribute`, `parameter`, `variable`, and finally `main`. MuLE has three separate declaration types for data containers, since they are representing three different concepts. There are no shortened versions of keywords. This might sound impractical, but that is not the focus of this language. In a similar fashion, primitive types are written as: `integer`, `rational`, `boolean` and `string`.

In earlier imperative programming languages such as COBOL a semicolon is used to separate statements, i.e. a semicolon has be to placed between statements but it is not required after the last statement. C and some languages based on its syntax rely on the semicolon to terminate each statement, i.e. a semicolon is required after each statement [93]. More modern programming and scripting languages, such as Scala, Python or JavaScript, offer an optional use of a semicolon. For example, a semicolon is required to separate statements written in the same line, however, a line break is also used to terminate a statement, which leads to inconsistent syntax [94] and to confusion among beginner students, meaning that an optional semicolon should be avoided at all costs. Semicolons are a common source of mistakes among beginner programmers [95][78], therefore, we have decided not to use semicolons to separate statements at all. Several statements can be written in a single line and are correctly identified by the compiler as long as white spaces separate keywords.

Similarly to C-based languages, commas are used to separate enumerated elements, such as value literals in enumeration types, parameters in operation declarations and invocations, and elements in value constructors for lists and composite types.

All kinds of brackets have a very narrow field of use (requirement **4 – Orthogonality**). Parentheses (`()`) are used in their usual role to manipulate operator precedence in expressions and to pass parameters to operations (both in declarations and invocations). Curly brackets ({}) are used to represent values of composite types (`Point2D{x = 2, y = 3}`). Square brackets (`[]`) are used to represent list values (`[1, 2, 3, 4]`) and indexed access to a list (`myList[0]`). Angle brackets (`<>`) are used to define type parameters of user defined composite types (`type MyList<Type> : composition`) and to pass actual type parameters to parameterized types (`reference<MyList<integer>>`).

Type System

According to requirement **10**, MuLE supports only **explicit strict static typing**. Data containers and functions are assigned a type upon their declaration. In concordance with the requirement **3 – Minimal number of language constructs**, MuLE should also offer a minimal number of data types. This does not imply that MuLE should follow the approach of Lisp, which offers lists as its single data type. Instead, MuLE should offer a range of orthogonal types which would cover the different kinds of data encountered in introductory programming courses.

In general, the language offers a limited number of types: e.g. two kinds of numeric types for integers and floating point numbers, a boolean type instead of relying on other types to simulate boolean values [23], a type to represent strings, a single flexible collection type, an enumeration type, a parameterizable composite type, a type to represent functions as data and the possibility to use each of these types as a reference type.

We have decided against a single numeric type to be able to demonstrate different numeric representation systems as well as their features and limitations. Rational numbers are a familiar concept to beginner students unlike their floating point representation on the hardware, therefore we have decided to name this type rational according to the requirement **6 – Build upon present knowledge**. Nevertheless, it should be noted that not every mathematically computable number can be represented by this type as these numbers can only be approximated on the machine.

Lists, composite types, references and function types are parameterized types, i.e. they accept type parameters that define which values can be stored by these wrapper types. For example, a list of integers may only store integer values, a reference to string can reference string values, a user defined composition can be used to implement generic data structures, and operation types define which types of arguments can be passed to lambda expressions as well as their return type.

A composite type can be declared as a subtype of another composite type inheriting all features of the super type (more on the topic of inheritance in section 4.2.3). Subtyping in MuLE is reflexive, meaning that any type is a subtype of itself, and transitive, e.g. if T_1, T_2 and T_3 are composite types and T_3 is a subtype of T_2 and T_2 is a subtype of T_1 then T_3 is a subtype of T_1. Formal type parameters of parameterized composition declarations can be restricted to a specific composite type, allowing only the subtypes of this type to be used as actual type parameters. Regarding the basic types, integer type is effectively a subtype of the floating point type.

Implicit type conversions are allowed only in case of a subtype being converted into a super type or an integer into a floating point number. Specific operations are provided to convert values of other types into strings. These semantics adhere to the requirement **9 – clear execution semantics**.

Since MuLE has strong static typing, type checking is performed on each expression at compile time. The context of the expression provides the expected type and the expression itself the actual type. The actual type must be compatible with the expected type, i.e. it must be either equal to or be a subtype of the expected type in most cases. In case of parameterized types, the corresponding type parameters must also be compatible. The program will not compile otherwise. More details are given in section 5.5.12.

In case of non-numeric primitive types, i.e. boolean, string and enumeration types, two values are equal when they have the same type and content. Numeric primitive types are equal if they represent the same numeric value, e.g. `42.0` is equal to `42` and `1.0E2` is equal to `100`. Two lists are equal, if they have the same number of elements, the order of the elements is equal in both lists and each respective element is equal. Two references are equal if the stored addresses are equal. Two compositions are equal if the value on one side of the equation expression is a reflexive or a transitive subtype of the value on the other side and the values of the shared attributes are equal.

Basic Types

As mentioned previously, MuLE offers a small set of primitive types, which can be represented by a single value literal. Here is the list of these types:

- `integer` – Integer values are represented by the 32bit two's complement system. Any arithmetic and comparative operator can be applied.
  `0, 42, 1337`

- `rational` – This data type represents floating point numbers by using the 64bit IEEE 754 standard [96]. Scientific notation is supported. Any arithmetic and comparative operator can be applied.
  `0.0, 3.14, 1.25E10`

- `string` – The only type to represent strings in MuLE, a character type is not included. MuLE offers a standard library with operations specific to this type. Equality check and string concatenation operators can be applied.
  `"", "Hello, world!"`

- `boolean` – This type represents logical Boolean values, thus Boolean operators and equality checks can be used in conjunction with this type.
  `true, false`

- `enumeration` – Users may declare enumerations, which define a limited range of value literals. The literals are accessed by their qualified names. Only equality check operators can be applied.
  `RGB.RED, RGB.GREEN, RGB.BLUE`
  The corresponding type declaration is:

  ```
  type RGB : enumeration
     RED, GREEN, BLUE
  endtype
  ```

Type Constructors

In addition to the primitive types discussed in the previous section, MuLE offers several types which are constructed by combining values of primitive types:

- `list<Type>` – Represents a list of values with the same (super) type. The list is ordered, the entries are accessed via their indices beginning at 0. A built-in standard library offers a range of functions working with this type. Multidimensional lists can be simulated as lists of lists.
  `[], [3, 1, 6], [1..10], [[1, 2], [3, 4]], [8**[8**"-"]]`

- `reference<Type>` – References are required to implement procedures with side effects, data structures and object relations in object-oriented programming. The reference expression stores a value in memory and returns the address of that value, which is then stored in a variable. In order to access the referenced value, the referencing variable must be dereferenced with the corresponding operator. Examples are given in sections 4.2.3 and 4.3.
  `null, reference 42, reference Point{}`

- `composition` – Compositions are another user defined type which represents structures in procedural programming and classes in object-oriented programming. Compositions may accept type parameters allowing to define custom parameterized types. Examples are given in sections 4.2.3 and 4.3.
  `Point{}, Point{x = 2, y = 3}`
  `MyList{}, MyList{head = reference 42, tail = null}`
  The corresponding type declarations are:

```
type Point : composition
   attribute x : integer
   attribute y : integer
endtype

type MyList<Type> : composition
   attribute head : reference<Type>
   attribute tail : reference<MyList<Type>>
endtype
```

- `operation(Type, ..., Type) : Type` – This is the data type representative for named and anonymous operations (i.e. lambda expressions). Types enclosed in parentheses define the data types of operation parameters, while the type after the colon is the return type of the operation. Both parameter types and the return type are optional. The lambda expression is the lexical representation of the value with such a data type. A variable with an operation type as well as the corresponding lambda expression are displayed in the example below.

```
variable f : operation(integer) :  integer
f := operation(parameter x : integer) : integer
        return x * 2
     endoperation
```

Default Values

Variables and attributes of compositions are automatically assigned a default value upon the declaration of a variable in order to prevent issues caused by uninitialized variables. Default values for specific types are:

- `integer` – `0`
- `rational` – `0.0`
- `boolean` – `false`
- `string` – `""` which represents an empty string value.
- `enumeration` – first listed literal of the enumeration type declaration.
- `reference` – `null`
- `list` – `[]` which represents an empty list value.
- `composition` – all attributes of the composition are initialized with their respective default values depending on their types.
- `operation` – a lambda expression with the expected parameter profile and a default return value based on the return type of the operation type. If the operation type has no return type, the lambda expression has no return value.

Expressions

The operators should be easily accessible from the keyboard, students should have no difficulty to find the required keys in order to type an operator, meaning that Unicode shortcuts for specific characters like $\leftarrow$ for assignment or $\times$ for multiplication should be avoided. Therefore, we have decided to use `:=` (*"is defined as"*) and `*` respectively, other arithmetical operators follow the same pattern and are similar to those used in most programming languages. Exceptions are the *exponential* (C++ and Java do not have such operator, Python uses `**`), *modulo* (the commonly used `%` is associated with percentage calculation) and *integer division* (the operator `/` is reserved for rational division) operators, for which we have used the keywords `exp`, `mod` and `div`.

We have chosen textual form for boolean operators, since this is mostly a new concept for beginner programmers and keywords `not`, `and` and `or` are more readable and meaningful compared to, for example, Java operators `!`, `&&` and `||` (requirement **2 – Non-cryptic syntax**).

The operator `=` implies equality and is therefore used for equality check operations in MuLE. We have decided to use the operator `&` for string concatenations instead of the commonly used + operator according to the requirement **4 – Orthogonality** which, amongst other things, states that syntactic homonyms and, therefore, operator overloading should be avoided.

Requirement **6** states that MuLE should build upon already present knowledge, therefore, infix notation is used for arithmetical operators. The operator precedence follows the same rules as school mathematics, expressions are evaluated from left to right. Section 5.3.5 includes a complete overview of operators as well as their precedence and associativity. In most cases, an expression is evaluated left to right, the types on both sides of binary expressions must be compatible with each other

Value Copying Semantics

Value copying semantics, i.e. creating and passing a copy of a value each time when a value is passed on, are easier to understand for beginner programmers compared to referencing semantics (requirement **1 – Easy to learn**). Moreover, value copying is a more intuitive concept to students based in their prior mathematical knowledge (requirement **6 – Build upon present knowledge**). In order to understand how references work in a programming language, the processes of memory usage have to be explained first, which is a difficult topic [97] that may lead to unnecessary information overload when introduced right at the beginning.

However, references are required to implement specific concepts, e.g. relations between entities in object-oriented programming. Furthermore, references are used in other programming languages. Therefore, it is still necessary to explain how this concept works in an introductory programming course, for example at a later stage prior to the discussion of topics like object-oriented programming or dynamic data structures (requirement **7 – Multi-paradigm language**).

Therefore, we have decided to offer value copying semantics as the standard mechanism when evaluating expressions and implement an additional explicit reference type, which can be used as a wrapper type for any other type, as explained in the previous section. The value of the reference type is the address to the value of the referenced type stored in heap. The reference type and the corresponding semantics and language constructs can be explained after a solid understanding of the basic value semantics is already present contributing to the requirements **5 – Incremental introduction** and **9 – Clear execution semantics**.

Any value is copied when passed to a data container (via an assignment or as an operation parameter). If a reference type is used, the address to the stored value is copied leading to several data containers referencing the same value in the memory. Inherently, this means that the language supports only call-by-value semantics for operation parameters. When a value is copied, a shallow copy is created, i.e. in case of a list or a composite type the contained values are copied over, however, the referenced values are not, only their references are copied. The language design excludes dangling references, garbage collection is used for automatic memory management (**1 – Easy to learn**). References must be explicitly dereferenced in order to access the value (**9 – Clear execution semantics**), `null` references cannot be dereferenced.

```
 1  program Values
 2
 3  operation foo(parameter a : integer)
 4      a := 5
 5  endoperation
 6
 7  main
 8      variable a : integer
 9      variable b : integer
10
11      a := 42
12      b := a
13      b := 10
14      foo(a)
15  endmain
```

Listing 26: Example program for value copying semantics with basic types. The corresponding memory states are shown in figure 9.

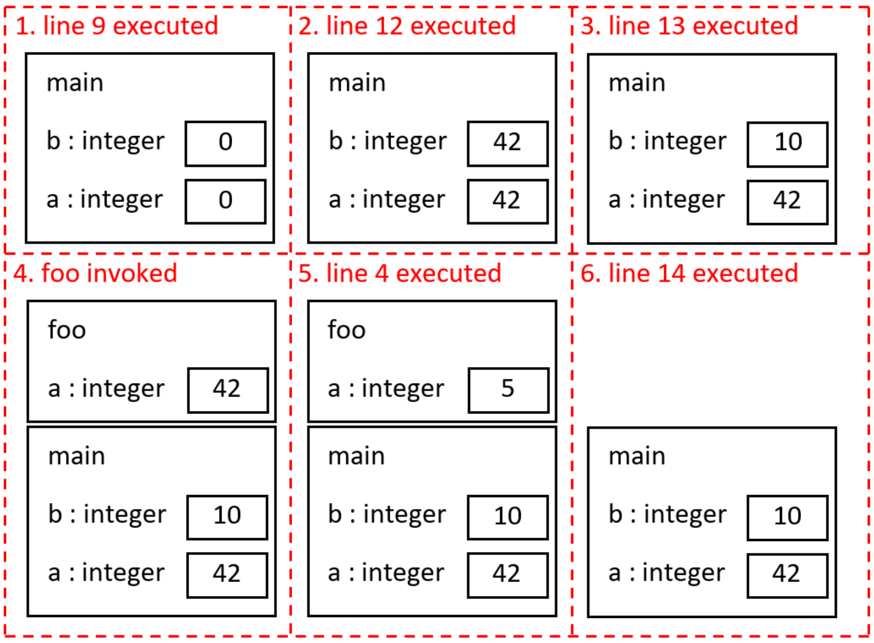


Figure 9: Depiction of the memory at various times during the execution of the program in listing 26.

Listing 26 and figure 9 show a simple program and the corresponding memory states at various times during the execution of the program. As we can see, the variables are initialized with their default values, in this case zeroes, directly after their declaration (*state 1*). When we assign the value of `a` to the variable `b` in line 12, the value of `a` is copied and the copy is stored in `b` (*state 2*). Both variables are still two separate data containers and are not aliases for the same value even if their values are equal. This means that subsequent altering of one of these variables will not have any effect on the other variable (*state 3*). Same applies to passing parameter values on operation calls, as demonstrated in line 14. The value of the variable `a` is copied and stored in parameter `a` of the operation `foo` (*state 4*). Changing the value of the parameter `a` inside the operation `foo` has no effect on the value stored in the variable `a` in the `main` procedure (*states 5 and 6*).

```
1   program References
2
3   operation foo(parameter a : reference<integer>)
4       a@ := 5
5   endoperation
6
7   main
8       variable a : reference<integer>
9       variable b : reference<integer>
10
11      a := reference 42
12      b := a
13      b@ := 10
14      foo(a)
15      b := reference a@
16      b@ := 42
17  endmain
```

Listing 27: Example program for value copying semantics with reference types. The corresponding memory states are shown in figure 10.

Listing 27 and figure 10 show a similar example program, however, this time we use reference types instead. After their declaration the variables `a` and `b` are initialized with their default values as usual, in case of reference types this value is always `null` (*state 1*). We use a reference creation expression to initialize the variable `a` in line 11, the referenced value is stored in the memory and the address to this value is then stored in the variable `a`. We then assign the value of `a` to the variable `b`, i.e. the stored address is copied and stored in `b`, meaning that both variables are now referencing the same stored value in the memory (*state 2*). We can alter this value by manually dereferencing one of the variables, e.g. `b` in line 13. Invoking an operation with a reference as a parameter will lead to the parameter referencing the same value (*state 3*). Altering the referenced value in the operation affects the state of the entire program in this case (*state 4*). If we, however, dereference one of the variables first, and then use the reference expression to assign a new value to the other variable (line 15), the dereferenced value is copied, stored in a separate place in the memory and the new address is stored in the variable (*state 5*). Even though the referenced values are equal for both variables, the equality check `a = b` will yield `false` since the stored addresses are different. Altering the stored value of one variable no longer has any effect on the other variable (*state 6*).

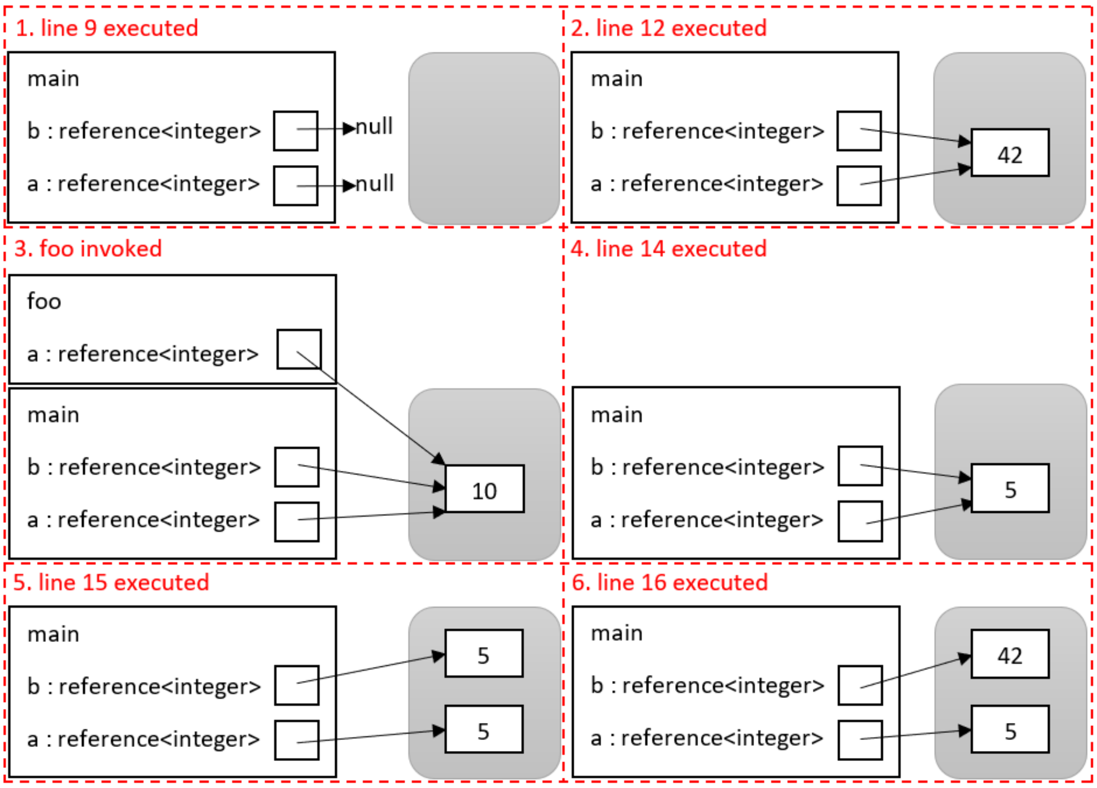


Figure 10: Depiction of the memory at various times during the execution of the program in listing 27.

Memory Management

As displayed in figures 9 and 10, the memory is separated into two parts, the call stack and the heap. The memory on the stack is allocated whenever a procedure or a function is invoked, starting with the `main`-procedure. The last invoked procedure, i.e. the one that is currently executed, is located at the top of the stack, as seen in states 4 and 5 in figure 9 and state 3 in figure 10. Memory for values of operation parameters and variables are therefore also allocated on the stack as a part of the encompassing operation. In both examples, the variables `a` and `b` were allocated in the context of the `main`-procedure while the memory for the parameter `a` was allocated when the operation `foo` was invoked. In the example in figure 9 we are using non-reference types, the integer values are stored directly on the stack. However, in the second example in figure 10 reference types are used, meaning that the values that are stored on the stack are addresses to the referenced values, which are allocated on the heap. Whenever a `reference` expression is executed (lines 11 and 15 in listing 27), the value after the keyword `reference` is copied onto the heap and its address is returned and can now be stored in a variable with a reference type. Therefore, when using non-reference types, memory is allocated on the stack only, however when references are used, the referenced values are stored on the heap and their addresses are stored on the stack. Values on the heap can not only be referenced by variables and parameters from the stack, but also by other values from the heap.

When an operation finishes its execution and is terminated, all memory reserved by that operation on the stack is automatically deallocated and the operation that was previously underneath the last operation becomes active. In both examples, the operation `foo` is invoked by the `main`-procedure, which becomes active again as soon as `foo` finishes its execution. The memory reserved on the heap by referenced values is required as long as the values are actually referenced. In the example figure 10 in state 3 the value 10 is referenced by three data containers from two different procedures, when the procedure `foo` is terminated, the parameter `a` no longer exists and is thus not referencing the value, which is however still referenced from the `main`-procedure and thus will be deallocated when the `main`-procedure, and thus the entire program, is terminated. Let us assume, that the variables `a` and `b` in the `main`-procedure do not exist. This would mean that as soon as the operation `foo` is terminated, the referenced value (which is changed to 5 in `foo`) is no longer referenced anywhere. In this case, the memory reserved by that value is deallocated by the garbage collector.

To sum it up, on the stack the memory is allocated for variables (on variable declaration) and operation parameters (on operation invocation) in the context of their respective procedures and functions including the `main`-procedure, and deallocated when the operation is terminated. On the heap, the memory is allocated when the `reference` expression is executed, and deallocated when the reserved memory is no longer referenced.

4.2.2. Procedural Programming

As mentioned previously, procedural programming was chosen as the core paradigm of MuLE which was then to be expanded by other paradigms [98]. This means that procedural constructs discussed in this section are also reused in other paradigms for the sake of orthogonality and keeping the number of language constructs at a minimum.

Statements are the central construct of imperative and, therefore, procedural programming. A statement is an instruction, that can change the state of the program or manipulate the execution order of other statements. It should be possible to easily identify the purpose of a statement based on its syntax and differentiate it from other statements (requirement **2 – Non cryptic syntax**).

Variables and Initialization

A *variable declaration* statement declares a named data container with a specific data type. Data of this data type or its subtypes can then be stored in the variable. The name of the variable is used to identify it and must therefore be unique within the given scope. The statement is introduced with a specific keyword – `variable` – followed by the name of the variable and its type, for example:

```
variable x : integer
```

When explaining the statement in the source code it can be pronounced as *"We declare a variable with the name x of type integer"*. Upon declaration, each variable is automatically assigned a specific default value in order to prevent unexpected behaviour caused by uninitialized variables (requirement **9 – Clear execution semantics**).

To keep the concept of declaration orthogonal to assignment, the corresponding statements are separated, i.e. manual assignment is not allowed in a declaration statement. The question arises, whether the variable declarations should be kept separate from other statements in a specific block similar to Pascal. Since variable declarations are statements just like any other statement, we have decided to allow users to declare variables at any place in the encompassing language construct.

Constants are similar in their functionality to variables, the only difference is that their values can only be assigned once which can be simulated by using variables just as well. They are thus not required.

Assignment

An *assignment statement* is necessary to assign values to data containers such as variables, parameters, attributes of composite types and entries in a list. The symbol `:=` is used as the assignment operator. The left hand side of the statement references the identifier while the right hand side consists of an expression which represents the assigned value. The type of the value must be compatible with the type of the data container. The statement can be pronounced as *"x is defined as 42"*.

```
x := 42
```

Control Structures

Blocks of statements are representative of structured programming and are, therefore, a mandatory requirement in procedural and object-oriented programming. To clarify, we are not talking about visual block based languages like

Scratch. A block can be a part of a main procedure, a subroutine or a specific statement, e.g. a loop or a conditional statement. Statements in a block are executed sequentially from top to bottom. Blocks are also used to specify the scope of variables. There are three distinct approaches to specify blocks:

- Use indentations to define blocks, which enforces to write well structured and readable code resulting in syntax similar to pseudo code. On the other hand, this may lead to semantic errors caused by wrong indentation which are harder to find.

- User brackets similar to languages like C and Java. This mitigates the issue of wrong indentations and reduces the number of required keywords at the cost of accumulated use of brackets. Missing opening or closing brackets is one of the most common mistakes of beginner programmers [95][78], however, such errors are recognised by the compiler and are therefore easier to fix. One remaining issue of this approach is that the brackets do not carry the information of the context, i.e. the language construct they belong to, by themselves.

- Use pairs of keywords that can be associated with the context of a block, for example `if ... end`, `if ... fi`, `if ... endif`. The first variant results in less keywords in the grammar of the language, however, this has the same issue of a lack of information regarding the context of the block as when using brackets. One of the latter two variants is thus more preferable, we have decided to use the variant `if ... endif` since its appearance is more intuitive. Surveys performed at the end of our preliminary programming course (see chapter 9) indicate that students also tend to prefer this approach to brackets.

An *if statement* is required to implement conditional branching. Such a statement could be pronounced as *"If a condition is true then execute these statements, otherwise do something else."* Switch-case statements have a similar function, i.e. they are not orthogonal to an `if`-statement, and can not be as easily expressed via natural language. They can be simulated by `if`-statements with multiple conditional branches and are, therefore, not included in the language.

```
if x = 42 then
   IO.writeString("The answer is correct!")
else
   IO.writeString("Sorry, try again.")
endif
```

A *loop statement* is necessary when several statements have to be repeated until a specific condition is met. Many different kinds of loops exist in various languages, examples are: `while CONDITION do ...`, `do ... while CONDITION`, `for each ELEMENT in a COLLECTION do ...`, etc. In addition, many languages include statements that allow to alter the behaviour of these loops such as the `break` and `continue` statements in Java.

To keep it simple, the language includes a single endless `loop` construct, which can be terminated with a single `exit` statement at any place in the loop if a specific condition is met, which offers the highest flexibility since the terminating condition can be placed anywhere in the loop. This comes, however, at the cost of requiring at least one `if`-statement in each loop to check the termination conditions. On the other hand, this behaviour does make sense for the sake of orthogonality and from a semantic point of view (specific to the intended behaviour of the respective constructs and their combinations).

A loop can be pronounced this way: *"We execute these statements, here we check if a specific condition is met and exit the loop if it is, if not, we continue executing statements in the loop until none are left and start from the beginning of the loop."*

```
loop
    IO.writeInteger(x)
    if x >= 42 then
        exit
    endif
    x := x + 1
endloop
```

An *exit statement* is an absolute requirement if the loop construct is representing an endless loop by itself. This statement is used to terminate the containing `loop` at any place of execution. A `continue` statement, which terminates the current iteration and initiates the next one is not required and can be simulated by placing the statements that would be skipped in an `if`-statement.

Operations and Operation Calls

Subroutines represent the main concept of procedural programming. We have decided not to separate between procedures, functions and methods on the lexical level, since it would imply three non-orthogonal language constructs with slight differences. Instead, our subroutines are simply called *operations*, this name implies functionality of some sort and is more familiar to novices than the actual word *subroutine* (requirements **4 – Orthogonality** and **6 – Build upon present knowledge**). Subroutines may accept value parameters and may return one value. Call-by-value is the sole parameter passing mode in MuLE since it is easier to understand as previously explained in this section.

Operations without a return type are procedures while those with a return type are functions. Functions must have a return statement for each of its control flow paths. Listing 28 demonstrates an example of a procedural MuLE program, operation `gcd` is a function while `lcm` is a procedure.

```
program gcdAndLcm
import IO

operation gcd(parameter a : integer, parameter b : integer) : integer
   loop
      if a = b then
         return b
      elseif a < b then
         b := b - a
      else
         a := a - b
      endif
   endloop
endoperation

operation lcm(parameter a : integer, parameter b : integer)
   IO.writeInteger(a * b div gcd(a, b))
endoperation

main
   IO.writeInteger(gcd(15, 21))
   IO.writeLine()
   lcm(15, 21)
endmain
```

Listing 28: Example of a procedural MuLE program.

Operation invocation – is necessary to call subroutines from the main procedure or other subroutines. Functions can be invoked as a statement, however, this way it will not be possible to use the returned value. Therefore, they are usually invoked as a part of an expression. Procedures must be invoked as statements since they do not return a value. The following example shows the invocation of the standard output procedure.

```
IO.writeString("What is the answer to the ultimate question of life, the universe,
    and everything?")
```

A *return statement* – is required to return values from a function or to terminate a procedure with a specific condition before it is fully executed. In the latter case the `return` statement is written simply as the keyword `return` without a specific value.

```
operation foo() : integer
   return 42
endoperation
```

4.2.3. Object-Oriented Programming

We have decided not to introduce a separate *class declaration* construct, but to expand the already present composite type (requirement **3 – Minimal number of language constructs**) [99]. In its basic procedural form, the composite type is similar to records or structs in other languages, i.e. it is a composition of named values with potentially different types.

One of the principle concepts of object-oriented programming is encapsulation, i.e. grouping of data and operations in a single syntactic unit. Therefore, we have decided to allow to declare operations within compositions, meaning that a composition becomes semantically a class with fields and methods in such a case. This approach also offers another advantage when following the *procedural-first* methodology of teaching. The biggest issue with the *procedural-first, objects later* approach is students struggling to accept new concepts due to a paradigm shift [65]. By using a known language construct at a later stage in an introductory course and slowly enhancing it with new features we can facilitate a slow gradual change from the procedural to the object-oriented way of programming (requirement **5 – Incremental introduction**).

The next logical step was to introduce inheritance relations by allowing a composition to extend another composition. Inheritance allows to reuse already implemented algorithms and types by declaring subtypes and is a fundamental concept of object-oriented programming. Both single and multiple inheritance have been taken into consideration. Single inheritance is less complex and easier to explain. It is however more restricted compared to multiple inheritance since it always results in a tree hierarchy. Therefore, it is not possible to implement a class inheriting the functionality of two independent classes without introducing some redundancy to the code by using single inheritance. Multiple inheritance on the other hand allows to design more natural solutions, comes, however, with a range of problems which require their own workarounds [91][24]. We have decided to implement single inheritance exactly for the reason of it being less complex than multiple inheritance (requirement **1 – Easy to learn**).

Some languages, while not naturally supporting multiple inheritance, make it possible to simulate it by using some of its language constructs. For example, it is possible to achieve some level of multiple inheritance in Java by using interfaces. Interfaces have however another very important function, they act as an interface to a complex system by making only specific functions of an implementing class visible to the client. For example, this allows users to implement different access rules for different user groups (guest, registered user, administrator, etc.) at a single system. We have decided not to offer an interface construct since its semantics can be represented by abstract compositions and visibility modifiers to a certain extent and it allows us to keep the number of language constructs at a minimum (requirement **3 – Minimal number of language constructs**).

In listing 29 we show an example of an object-oriented program which also demonstrates the use of references in MuLE, the summary of that program is shown in figure 11. In this example we have a composition `Person` which has a name and a reference to a `Vehicle`. On its own, this composition is not any different from a simple structure and shows an example of how a composite type is used in a purely procedural context.

```
program vehicles
import IO
import Lists

type Person : composition
   attribute name : string
   attribute myVehicle : reference<Vehicle>
endtype

abstract type Vehicle : composition
   protected attribute manufacturer : string
   protected attribute mileage : integer

   operation drive(parameter distance : integer)
      mileage := mileage + distance
   endoperation

   operation printData()
      IO.writeString(manufacturer)
      IO.writeLine()
      IO.writeString("Mileage: ")
      IO.writeInteger(mileage)
      IO.writeLine()
   endoperation
endtype

type Car : composition extends Vehicle endtype

type Truck : composition extends Vehicle
   private attribute weight : rational
   override operation printData()
      super.printData()
      IO.writeString("Total weight: ")
      IO.writeRational(weight)
      IO.writeLine()
   endoperation
endtype
```

Listing 29: Example of an object-oriented MuLE program.

The type `Vehicle`, on the other hand, is declared as `abstract` and features `protected` attributes and `public` operations. Abstract types can include `abstract` operations which lack an operation body, meaning that such types can not be instantiated. We have decided to use explicit visibility modifiers only if we want to restrict visibility, meaning that they are not required in procedural programs (requirements **8 – No forced object-orientation** and **5 – Incremental introduction**). Visibility modifiers can still be used in procedural context, e.g. to hide type declarations and operations in libraries. The types `Car` and `Truck` are both subtypes of `Vehicle`, whereby `Truck` includes an additional `private` attribute and redefines the inherited operation `printData()`.

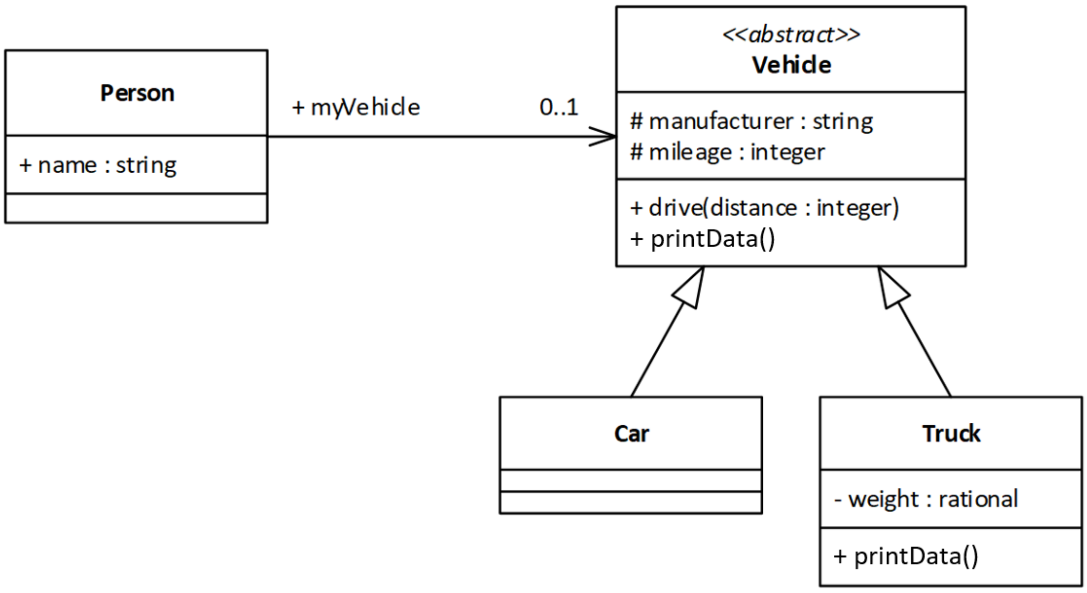


Figure 11: Class diagram representing the example in listing 29.

The main procedure of this program is displayed in listing 30. Here we declare two vehicles and instantiate them by using non abstract subtypes and their corresponding composite type value constructors. Since the visibility of the attributes is restricted, after the instantiation their values can only be printed to the terminal via the `printData()` operation. All attributes are visible in the value constructors, meaning that their values can be set regardless whether their visibility is restricted or not. Otherwise it would be necessary to implement interface or factory design patterns to demonstrate object initialization with read-only access to its attributes. We have decided to provide two variants of a value constructor for composite types, an empty constructor and a variant where all attributes including inherited ones are listed. It should be noted, that these value constructors are not comparable to constructors of languages like Java and C++, whose constructors are implemented as special methods which create an object and offer the full flexibility of a subroutine. Continuing with our example, we declare and initialize two persons and finally call the `printData()` operation on the public attribute `myVehicle` of both `Person` instances.

```
main
    variable v1 : reference<Vehicle>
    v1 := reference Car{manufacturer = "BMW", mileage = 0}
    variable v2 : reference<Vehicle>
    v2 := reference Truck{manufacturer = "Daimler", mileage = 0, weight = 18}

    variable alice : reference<Person>
    alice := reference Person{name = "Alice", myVehicle = v1}
    IO.writeString("Alice's vehicle:\n")
    alice@.myVehicle@.printData()   IO.writeLine()
    variable bob : reference<Person>
    bob := reference Person{name = "Bob", myVehicle = v2}
    IO.writeString("Bob's vehicle:\n")
    bob@.myVehicle@.printData()     IO.writeLine()

    variable vehicles : list<reference<Vehicle>>
    vehicles := [v1, v2]
    variable i : integer
    loop
        if i >= Lists.lengthOf(vehicles) then exit endif
        let variable truck : reference<Truck> be vehicles[i] do
            IO.writeString("It's a truck!\n")
        endlet
        i := i + 1
    endloop
endmain
```

Listing 30: Main procedure of the program in listing 29.

The produced output is:

```
Alice's vehicle:
BMW
Mileage: 0

Bob's vehicle:
Daimler
Mileage: 0
Total weight: 18.0

It's a truck!
```

As we see, we have a different output in case of Bob's vehicle, the executed operation depends on the dynamic type of the object, which is `Truck` in this case.

The language also offers a `let`-statement which is also shown in listing 30 where we iterate over a list of `Vehicles` containing instances of each a `Car` and a `Truck` and use the `let`-statement to execute code only when the corresponding list value is a `Truck`. As we see in the output of the program, the message `It's a truck!` was printed only once. The `let`-statement performs an instance check and a type conversion. The head of a `let`-statement includes a variable declaration and an expression, if the type of the expression corresponds to the type of the variable, i.e. it is equal or is a subtype, the value of the expression is stored in the variable and the block of the `let`-statement, where this variable is visible, is executed. This allows us to perform runtime-safe type conversions within an inheritance hierarchy thus removing a source of type conversion related mistakes (requirement **1 – Easy to learn**).

Finally, as already mentioned in section 4.2.1, composition declarations can accept type parameters allowing us to implement generic data structures.

4.2.4. Functional Programming

We have already discussed what functional programming is in section 2.4. Its main idea is to use functions as data represented by lambda expressions, i.e. functions may accept other functions as parameters or return them as values. Furthermore, functional programming encourages stateless calculations, ideally, a functional program is a single expression which returns a single value. State altering constructs, like variables and assignments are not present in purely functional languages. However, since MuLE is a multi-paradigm language that supports procedural and object-oriented programming, this cannot be enforced. Keeping the number of constructs small and orthogonal (requirements **3 – Minimal number of language constructs** and **4 – Orthogonality**) we have decided not to introduce constants as data containers or specific modifiers that prevent more than one assignment to variables such as `final` in Java.

Functional programming languages tend to have a shorter syntax compared to imperative languages. Functional constructs in multi-paradigm languages follow a similar pattern, for example the Java lambda expression `(x, y) -> x + y;` would require far more keywords (return type, parameter types, return keyword, brackets, etc.) if implemented as a common method while keeping the same semantics. It may be a more elegant way to write programs, however, this will confuse students. Using different syntax for rather familiar concepts, e.g. lambda expressions and common subroutines, will not facilitate recognition of the similarities between these concepts. Therefore, we have decided not to use shortened syntax for lambda expressions (requirements **1 – Easy to learn**, **3 – Minimal number of language constructs**, and **5 – Incremental introduction**).

A MuLE lambda expression represents an anonymous operation, its syntactical form is the same as that of a named operation except for the lack of an identifier. It can accept an arbitrary number of arguments and may return a value, i.e. it can represent both functions and procedures. It can be argued, that this is not the originally intended way of using lambda expressions, i.e. as anonymous *functions*, however, MuLE is not a pure functional language and this approach enables a more flexible use of this construct. Moreover, it makes it easier to explain the concept of lambda expressions if subroutines were already explained in a *procedural-first* approach (requirement **5 – Incremental introduction**). Thus, the aforementioned lambda expression `(x, y) -> x + y;` has to be written the following way in MuLE:

```
operation(parameter x : integer, parameter y : integer) : integer
   return x + y
endoperation
```

Listing 31: Example of a lambda expression in MuLE.

```
program filterExample
import IO
import Lists
import Strings

operation filter(parameter l : list<integer>,
                 parameter f : operation(integer) :  boolean) : list<integer>
   return filterHelper(l, f, [], 0)
endoperation

operation filterHelper(parameter l : list<integer>,
                       parameter f : operation(integer) :  boolean,
                       parameter lFiltered : list<integer>,
                       parameter i : integer) : list<integer>
   if i = Lists.lengthOf(l) then
      return lFiltered
   elseif f(l[i]) then
      return filterHelper(l, f, Lists.append(lFiltered, l[i]), i + 1)
   else
      return filterHelper(l, f, lFiltered, i + 1)
   endif
endoperation

main
   IO.writeString(Strings.genericToString(
      filter([1 .. 10],
         operation(parameter x : integer) : boolean
            if x mod 2 = 0 then
               return true
            else
               return false
            endif
         endoperation
      )
   ) & "\n")
endmain
```

Listing 32: Example of higher order function in MuLE.

Since an expression represents a value, it must have a type. We have already mentioned the operation type in section 4.2.1. The type of the lambda expression in listing 31 is `operation(integer, integer) : integer` which closely represents the signature of the operation. Lambda expressions and named operations (merely referenced by their names without the parameters) can be assigned to data containers with an operation type allowing to implement higher order functions. Listing 32 demonstrates such an example, the operation `filter` accepts a list `l` and an operation `f` which accepts an integer parameter and returns a boolean value. This operation is meant as an interface for the operation `filterHelper` which performs the actual filtering recursively and requires additional parameters which represent the filtered list and the current position in the original list to do this. This operation appends values from the original list to the filtered list if the index is within the list boundaries and the passed predicate is `true` for the current value in the list, and returns the filtered list. We invoke the operation `filter` in the main procedure with a list containing all integer numbers between `1` and `10` and a lambda expression which returns `true` if the passed integer parameter is even. The output of the program is `[2, 4, 6, 8, 10]`.

Currying is also possible, to achieve this, a function must return another function, an example is shown in listing 33. The variable `sum` is declared as an operation which accepts an integer and returns another operation, which also accepts an integer and returns an integer. The corresponding lambda expression returns another lambda expression with matching parameter and return types. Therefore, we pass two parameters subsequently when invoking the operation `sum`, the second parameter `(3)` is passed to the operation returned from invoking `sum(2)`. The output of the program is `5`.

```
program curryingExample
import IO

main
   variable sum : operation(integer) :  operation(integer) :  integer
   sum := operation(parameter x : integer) : operation(integer) :  integer
      return operation(parameter y : integer) : integer
         return x + y
      endoperation
   endoperation
   IO.writeInteger(sum(2)(3))
endmain
```

Listing 33: Example of currying in MuLE.

MuLE does not offer lazy evaluation which is, for example, present in Haskell. The built in list data structure does not allow to define potentially endless lists, since it would lead to an endless initialization when assigning such a list to a variable. Nevertheless, the standard library with list specific operations provides a set of typical functions associated with this paradigm such as `head`, `tail`, `filter` and `forEach`. Finally, users can implement recursive data structures as well as related operations similar to those used in functional programming languages using the reference type.

4.3. Multi-Paradigm Programming with MuLE

In the previous section we have seen short examples of procedural, object-oriented and functional programs in MuLE. Let us now demonstrate all previously discussed concepts in a single bigger example by implementing the dictionary lookup algorithm that we have previously discussed and implemented using various languages and paradigms in chapter 2.

The program is separated into two compilation units, the library `dictionary` which contains all dictionary related implementation details and the program `test` with the main procedure. Listing 34 shows the first part of the library with the necessary standard library import instructions and two private composite types `WordPair` and `Section`, meaning that these types are not exported and are meant for internal use. Both types contain only attributes and are therefore not meant to represent objects. Finally, it contains the abstract type `Dictionary` which lacks a visibility modifier, i.e. this is the type that is meant to be used by the users of this library. It contains only two abstract operations `printPurpose()` and `printTranslations(word : string)`.

```
library dictionary
import IO
import Strings
import Lists

private type WordPair : composition
    attribute word : string
    attribute translation : string
endtype

private type Section : composition
    attribute letter : string
    attribute wordPairs : list<WordPair>
endtype

abstract type Dictionary : composition
    abstract operation printPurpose()
    abstract operation printTranslations(parameter word : string)
endtype
```

Listing 34: First part of the dictionary library.

The `Dictionary` itself is abstract, it cannot be instantiated and acts merely as an interface. Listing 35 shows the continuation of the `dictionary` library unit. To be precise, it contains the composite type DictionaryImpl which extends the abstract type Dictionary. Since it is not abstract, it has to implement both inherited abstract operations. Additionally, it implements the operation getTargetSection(letter : string), which is required for the internal working of our dictionary implementation. As a reminder, when looking for a translation our dictionary algorithm first searches for the section where the word is stored, and then through the word pairs in the section. Thus, the operation printTranslations(word : string) must first invoke the operation getTargetSection(letter : string), letter being the first character of word transformed into its upper case variant. If no section with the given letter is found, a null reference is returned and a corresponding message is printed. Otherwise, the operation searches for a matching word pair in the sections and either prints the translation or a corresponding message if no match is found. These lookup operations are implemented by using the function filter from the standard library Lists. This functions returns a new list with copies of the entries of the original list, which fulfil a specific condition that is defined in a lambda expression. The iteration over the list of word pairs is performed in a similar functional way with the Lists operation forEach, which accepts a list and a lambda expression, which is subsequently applied to all entries in the list. This type and, therefore, all its implementation details are hidden from the user by the modifier private.

```
private type DictionaryImpl : composition extends Dictionary
    attribute purpose : string
    attribute sections : list<reference<Section>>

    operation getTargetSection(parameter letter : string) : reference<Section>
        variable targetSections : list<reference<Section>>
        targetSections := Lists.filter(sections,
            operation(parameter s : reference<Section>) : boolean
                if s@.letter = letter then
                    return true
                else
                    return false
                endif
            endoperation
        )
        if Lists.lengthOf(targetSections) > 0 then
            return targetSections[0]
        endif
        return null
    endoperation

    override operation printPurpose()
        IO.writeString(purpose) IO.writeLine()
    endoperation
```

```
        override operation printTranslations(parameter word : string)
            variable targetSection : reference<Section>
            targetSection :=
                    getTargetSection(Strings.toUpperCase(Strings.subString(word, 0, 0)))
            if targetSection = null then
                IO.writeString(word & " : no matches found")
                IO.writeLine()
            else
                variable legitWordPairs : list<WordPair>
                legitWordPairs := Lists.filter(targetSection@.wordPairs,
                    operation(parameter wp : WordPair) : boolean
                        if wp.word = word then
                            return true
                        else
                            return false
                        endif
                    endoperation
                )
                if Lists.isEmpty(legitWordPairs) then
                    IO.writeString(word & " : no matches found")
                    IO.writeLine()
                endif
                Lists.forEach(legitWordPairs,
                    operation(parameter wordPair : WordPair)
                        if (wordPair.word = word) then
                            IO.writeString(word & " : " & wordPair.translation)
                            IO.writeLine()
                        endif
                    endoperation
                )
            endif
        endoperation
    endtype
```

Listing 35: Second part of the dictionary library.

In the final part of the dictionary library (listing 36) we see an operation `createDictionary`, which accepts a string and a two dimensional list of strings as parameters. The string is meant to be the purpose of the soon-to-be dictionary while the list is meant to contain the words and their corresponding translations. These parameters are used to initialize the dictionary as a `reference<DictionaryImpl>`, which is then returned at the end of the operation as `reference<Dictionary>`, i.e. the visible abstract type. Since the actually working type `DictionaryImpl` is hidden, users can only interact with the the library using the abstract type `Dictionary` and this creation operation.

```
operation createDictionary(parameter p : string,
                        parameter wPs : list<list<string>>) : reference<Dictionary>
    variable dict : reference<DictionaryImpl>
    dict := reference DictionaryImpl{purpose = p, sections = []}
    variable i : integer
    loop
        if i >= Lists.lengthOf(wPs) then exit endif
        variable word2 : string
        word2 := wPs[i][0]
        variable translation2 : string
        translation2 := wPs[i][1]
        variable letter2 : string
        letter2 := Strings.toUpperCase(Strings.subString(word2, 0, 0))
        variable section : reference<Section>
        section := dict@.getTargetSection(letter2)
        if section = null then
            dict@.sections := Lists.append(dict@.sections,
                                reference Section{letter = letter2, wordPairs = []})
        endif
        dict@.getTargetSection(letter2)@.wordPairs := Lists.append(
            dict@.getTargetSection(letter2)@.wordPairs,
            WordPair{word = word2, translation = translation2}
        )
        i := i + 1
    endloop
    return dict
endoperation
```

Listing 36: Third part of the dictionary library.

The library `dictionary` does not contain a main procedure and can not be executed by itself. It is meant to be imported and used somewhere else, for example in the `test` program in listing 37. Here we declare a variable `dict` as a `reference<dictionary.Dictionary>`. We must use a reference type, otherwise we wouldn't be able to create a default value (which is `null` for reference types) since `Dictionary` is an abstract type. Furthermore it makes sense to use a reference type since we are handling objects. The variable is then initialized by using the operation `createDictionary`, with two different meanings for the word *Ausdruck*. Finally we test it by calling the operation `printTranslations` with various words, the produced output is:

```
German - English
Ausdruck : expression
Ausdruck : printout
bauen : build
Algorithmus : no matches found
Sprache : no matches found
```

The resulting program relies on all three supported paradigms (requirement **7 – Multi-paradigm language**). The library `dictionary` provides the necessary types and the functionality. It relies on object-oriented and functional programming to implement the dictionary itself and on procedural programming to provide the operation that is tasked with the creation of `Dictionary` instances. Implementation details that are not relevant from the perspective of a user are abstracted (requirement **11 – Data abstraction**).

```
program test
import dictionary

main
   variable dict : reference<dictionary.Dictionary>
   dict := dictionary.createDictionary("German - English", [
      ["Alphabet","alphabet"],
      ["Anweisung","Anweisung"],
      ["Ausdruck","expression"],
      ["Ausdruck","printout"],
      ["bauen","build"],
      ["Baum","tree"],
      ["Bedingung","condition"],
      ["Chip","chip"],
      ["Code","code"],
      ["Computer","computer"]
   ])
   dict@.printPurpose()
   dict@.printTranslations("Ausdruck")
   dict@.printTranslations("bauen")
   dict@.printTranslations("Algorithmus")
   dict@.printTranslations("Sprache")
endmain
```

Listing 37: Test program for the MuLE dictionary example.

4.4. Discussion

The decision to use procedural programming as a platform and implement other paradigms as expansions while keeping the number of language constructs at a minimum has a strong influence on the implementation of these paradigms, which is especially evident in the support of functional programming. The type system is based on procedural programming with explicit strict static typing and data types typical for this paradigm. Meanwhile, functional programming relies usually on type inference which allows to write shorter and more elegant programs. State alteration is an integral concept of procedural programming which stands in contrast to the side-effect free approach of functional programming and is thus mostly absent from these languages. MuLE supports both styles but does not enforce any of them. Other concepts typical for functional programming such as lazy evaluation and recursive data structures can be simulated, but are not natively integrated into the language. Keeping the textual representation of lambda expressions in MuLE consistent to the syntax of named operations results in a quite lengthy notation compared to lambda expressions in other functional languages.

Object-oriented programming is supported via extensions to the composition type thus allowing both stack and heap allocated objects, with only the latter making overall sense in the context of object-oriented programming. This means that users have to manually wrap each composition type as a reference type and use the dereferencing operator to access the stored values again resulting in less elegant code compared to the one written in Java or Python where this operations are handled implicitly. Furthermore, this also means that MuLE lacks a `this` reference which would lead to issues with dangling references when using stack allocated objects. Finally, constructors as they are used in object-oriented programming are not present in MuLE, the provided value constructors for compositions merely allow to initialize attribute values, further constructor functionality has to be simulated by operations.

As we can see, these decisions result in less elegant code compared to other languages. However, as previously stated it is not our intention to implement a language with a short syntax. MuLE is meant to teach programming concepts, which is easier achieved when these concepts are not obscured by short syntax or hidden behind a layer of implicit behaviour. Even though the number of language constructs is kept small, the language is Turing complete, which it must be in order to be a capable educational tool. The proof of the Turing completeness is presented in appendix J.

In this chapter we have discussed the requirements that we have defined for MuLE and the resulting design decisions. Furthermore, we have presented various programming examples utilizing our language focusing on each supported paradigm as well as a larger example which utilizes language constructs and programming concepts of all three paradigms. In the following chapter we will specify concrete rules for MuLE which are based on the design decisions explained in this chapter.

5. Specification of MuLE

By now we have defined the requirements for MuLE, explained our design decisions and have already seen it in action as demonstrated in several programming examples. This chapter will provide the specification of our language offering a more precise definition of implemented language constructs based on grammar, scoping and validation rules. Not following any of these rules will result in compilation errors. Since this chapter contains the specification of MuLE based on the design decisions explained in the previous chapter, there will be some overlapping content between the two chapters.

5.1. Identifiers, Namespaces and Scoping

The topic of this section are identifiable program elements, namespaces and scoping rules. The lexical composition of identifiers is presented later in section 5.3.2. An identifier is used to reference a program element in language constructs, i.e. it is a name or a label given to a program element. For example, an expression `a + 2` is referencing a data container by its name `a`. Following program elements can be referenced by their identifiers:

- Compilation units, i.e. MuLE programs and libraries.
- Declared types, i.e. enumerations, compositions and type parameters of compositions.
- Literals of enumeration types.
- Attributes of compositions.
- Operations.
- Value parameters of operations and lambda expressions.
- Variables.

5.1.1. Namespaces and Scope Rules

A *namespace* is a section of a program in which a named element was declared. Named program elements in the same namespace must have unique identifiers in order to prevent ambiguity. Namespaces can be nested, identifiers declared in outer namespaces are visible and can be referenced in inner namespaces. Shadowing of identifiers from an outer namespace is not permitted, meaning that the identifiers must be unique not only in the same namespace, but also in all enclosing namespaces. There are three types of namespaces in a single compilation unit:

1. The namespace defined by the compilation unit itself represents the outer most namespace. It includes itself, import instructions, type declarations and operations. This means the identifiers that are visible in this namespace are the name of the containing compilation itself, the names of imported compilations units, as well as all types and operations declared directly in the compilation unit.

2. The namespaces defined by enumerations and compositions. The namespace defined by an enumeration includes all enumeration literals while the composition namespace may include identifiers of type parameters, nested compositions, attributes and operations.

3. The namespaces of language constructs with a block of statements, such as the `main`-procedure, operations and control flow defining statements (`if`, `loop` and `let`). Similar to the previous type, these namespaces can be nested. The difference is however is that since these namespaces are mainly defined by blocks of statements, only the previously defined elements can be referenced in the same encompassing namespace. This disallows referencing variables which are declared at a later stage in the source code and are thus not yet initialized. These namespaces include identifiers of variable declarations (both as a variable declaration statement and as a part of a `let`-statement clause) and operation parameters. Conditional `if`- and `let`-statements, which may have several clauses with separate blocks, define separate namespaces for each clause.

A *scope* is the region of the code where a named element can be referenced by its name. Scope rules define which identifiers are allowed to be accessed within a namespace from a concrete textual region in a program. Scopes are structured hierarchically in a compilation unit. The lowest tier is the area containing the reference to an identifier. The container of this area represents the immediate containing scope area. The upper most scope area is the namespace of the compilation unit where other compilation units are imported and the types and operations are declared.

Whenever a named element is referenced in the source code, e.g. when a declared type is used to declare a data container of a variable is used in an expression, scoping rules check whether this named element is actually accessible from the current context. We will use the term *reference* as the reference to a named element from a specific context in the scoping rules in this section. The left-hand side of the rule describes the context of the reference while the right-hand side summarizes elements visible from this context. Scoping rules merely compute elements visible from a specific context, whether these elements are actually usable is checked by validation and type consistency rules (see further sections of this chapter). The rules are:

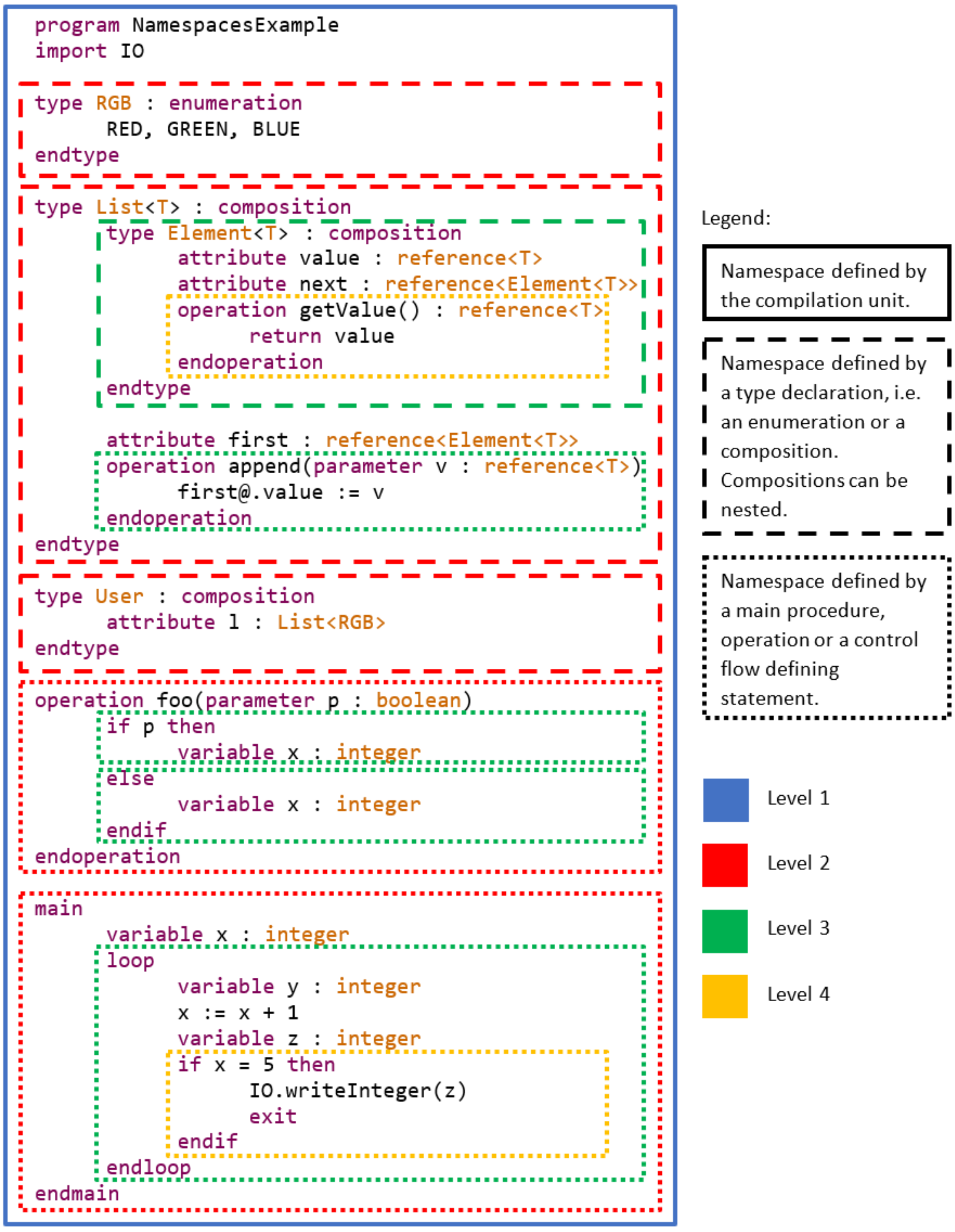
Simple name – expressions are contained in statements, which are contained in blocks. All kinds of previously mentioned named elements can be referenced by an expression. If the reference is not a member call, i.e. its context is not part of a qualified name, then the accessible named elements are as follows:

- All variables declared prior to the context statement in the same namespace and all namespaces defined by encompassing block.
- If the containing block is assigned to a clause in a `let`-statement, then the variable defined by this statement is included.
- If the containing block is assigned to an operation, then all parameters of this operation are included. If the operation is included in a composition, then the named elements defined in the namespace of the composition are included.
- All named elements defined in the namespace of the encompassing compilation unit are included.

Qualified name – the reference is part of a qualified name, e.g. `a.b.reference`, wherein `b` is the context of the `reference`, i.e. the named elements accessible by the reference are those visible in this context. Visibility modifiers `private` and `protected` may further restrict the visibility depending on the context. Naturally, the elements `a` and `b` must be visible too. More details on qualified names and visibility modifiers are given in sections 5.1.2 and 5.1.3 respectively.

- **Compilation unit** – accessible named elements are those defined in the namespace of the unit. If the reference is not contained within the same compilation unit, then visibility modifiers are checked as well. Elements marked as private are not visible.
- **Declared enumeration type** – accessible named elements are the enumeration literals.
- **Declared composition type** – accessible named elements are the nested compositions. Visibility modifiers are checked as well.
- **Instance of a composition type** – accessible named elements are attributes and operations defined in the namespace of the composition as well as those that are inherited. Visibility modifiers are considered as well.

Figure 12 shows an example of a program with different namespaces, i.e. the namespace defined by the unit itself, the namespaces defined by the declared types and finally the namespaces defined by block based language constructs. The figure also shows the different nesting levels coded by colours, blue representing



```
program NamespacesExample
import IO

type RGB : enumeration
        RED, GREEN, BLUE
endtype

type List<T> : composition
        type Element<T> : composition
                attribute value : reference<T>
                attribute next : reference<Element<T>>
                operation getValue() : reference<T>
                        return value
                endoperation
        endtype

        attribute first : reference<Element<T>>
        operation append(parameter v : reference<T>)
                first@.value := v
        endoperation
endtype

type User : composition
        attribute l : List<RGB>
endtype

operation foo(parameter p : boolean)
        if p then
                variable x : integer
        else
                variable x : integer
        endif
endoperation

main
        variable x : integer
        loop
                variable y : integer
                x := x + 1
                variable z : integer
                if x = 5 then
                        IO.writeInteger(z)
                        exit
                endif
        endloop
endmain
```

Legend:

Namespace defined by the compilation unit.

Namespace defined by a type declaration, i.e. an enumeration or a composition. Compositions can be nested.

Namespace defined by a main procedure, operation or a control flow defining statement.

🟦	Level 1
🟥	Level 2
🟩	Level 3
🟨	Level 4

Figure 12: An example demonstrating namespaces.

the highest level of the compilation unit and orange is the lowest level in this example.

The `main`-procedure represents the namespace for the variable `x` while the `loop`-

statement acts as the namespace for the variables `y` and `z`. The `loop`-statement also represents a nested inner namespace of the outer `main` namespace which means that variable `x` exists within the loop and no other named element can be declared in the loop using the identifier `x`. The `main`-procedure is located in the compilation unit `NamespacesExample` which acts as the outer most namespace, which contains the identifiers of itself, the imported library `IO`, the declared types `RGB`, `List` and `User` as well as the operation `foo`. Finally, the `if`-statement does not include any variable declarations meaning that its namespace is empty, however since it is nested in the loop namespace, all declarations from this namespace and its outer namespaces are visible in the `if`-statement. To summarize it, the namespaces related to the `main`-procedure and their corresponding identifiers are structured in the following order:

```
compilationUnit[NamespacesExample, IO, RGB, List, User, foo]
    main[x]
        loop[y, z]
            if[]
```

The assignment references the variable `x` which means that this identifier, which is located in an outer namespace, has to be visible, otherwise the program would not compile. The actually visible identifiers for this assignment defined by scope rules are following:

```
compilationUnit[NamespacesExample, IO, RGB, List, User, foo]
    main[x]
        loop[y]
```

The difference is that the variable `z` is not visible in the assignment since forward references are only allowed in the outer most namespace and namespaces defined by compositions. Types and operations defined in the compilation unit can reference each other whereas in a local namespace, e.g. inside an operation, an expression can reference only those variables that are declared prior to the statement containing the referencing expression. Similarly the namespace of the `if`-statement is not listed, since it is defined after the assignment.

As previously mentioned, all declarations must have unique identifiers in the same namespace, otherwise the program is ambiguous and cannot be compiled. The operation `foo` in figure 12 demonstrates an example for two variable declarations which are located in separate namespaces. The `if`-statement includes a `then` block, which is obligatory, as well as an optional `else` block. Only one of these blocks can be executed at the same time, therefore only one of these variables can exist at runtime resulting in a correct and unambiguous program. The example in listing 38 on the other hand shows two program elements with the same identifier within overlapping namespaces. The operation `a` is declared within the outer namespace defined by the compilation unit. The variable `a` is declared in the local namespace of the `main` procedure and is therefore nested in the namespace of the compilation unit. This leads to a compile time error message and the program can be neither compiled nor executed.

```
program NamespacesExample3

operation a()
endoperation

main
    variable a : integer
endmain
```

Listing 38: Two elements with the same name in overlapping namespaces.

Continuing with the example in figure 12, as already mentioned, the identifiers defined in the namespace of the compilation unit are: `NamespacesExample`, `IO`, `RGB`, `List`, `User` and `foo`. All of them can reference each other, even those that are declared later in the code, for example the composition `List` could reference the composition `User` as its super type and invoke the operation `foo` in one of its own operations. The named elements declared within the composition `List` are `T`, `Element`, `first` and `append`. Yet again, all of them can reference each other, for example we could invoke the operation `append` from within the composition `Element` where it is visible.

5.1.2. Qualified Names

Import statements allow to reuse operations and types implemented in other compilation units. It should be noted, that only compilation units marked as libraries which lack a main procedure can be imported, otherwise a program would have several entry points.

```
program QualifiedNames
import IO

operation writeString(parameter str : string)
    IO.writeString("myWriteString: " & str)
endoperation

main
    IO.writeString("Hello\n")
    writeString("Hello, world!\n")
    QualifiedNames.writeString("Hello, world!\n")
endmain
```

Listing 39: Referencing library operations by using qualified names.

In order to be used, an element contained in an imported library must be referenced by its qualified name. Example given in listing 39 shows the use of the `writeString` operation which is declared in the MuLE standard library `IO`. The corresponding qualified name consists of the name of the containing library and the name of the operation separated by a dot, i.e. the resulting qualified name is `IO.writeString`.

The program in listing 39 also contains an operation with an identifier `writeString`. Both operations are contained within separate namespaces defined by their corresponding compilation units, meaning that no naming conflict exists and the program can be compiled. While the imported operation has to be accessed via its qualified name, the operation `writeString` in the program `QualifiedNames` can be accessed within the program either directly by its simple name (as can be seen in line 10) or by its qualified name (line 11). Both variants lead to the same result, the output of the program is:

```
Hello
myWriteString: Hello, world!
myWriteString: Hello, world!
```

Qualified names must also be used to access types, attributes and operations defined in compositions as well as literals of enumeration types. An enumeration `RGB` and a composition `Color` are defined in the program in listing 40. In the main procedure of the program a variable `color` of type `Color` is defined and its attributes are then accessed and initialized via their qualified names. In this case the variable represents the data container for the compositions with the corresponding attribute values, therefore, the identifier of the variable `color` is used in the qualified name instead of the type `Color` followed by the name of an attribute separated by a dot, for example `color.rgbValue`. Enumeration literals are accessed in a similar way, an example is shown in line 14, the qualified name for the value `GREEN` contained in the enumeration `RGB` is `RGB.GREEN`.

```
1   program QualifiedNames2
2
3   type RGB : enumeration
4      RED, GREEN, BLUE
5   endtype
6
7   type Color : composition
8      attribute rgbValue : RGB
9      attribute alphaValue : integer
10  endtype
11
12  main
13     variable color : Color
14     color.rgbValue := RGB.GREEN
15     color.alphaValue := 125
16  endmain
```

Listing 40: Referencing attributes and enumeration literals using their qualified names.

Let us assume that an enumeration RGB is contained in a library Colors which is imported in a program. In this case the qualified name for the value GREEN would be Colors.RGB.GREEN. Finally, if we once again take a look at our example in figure 12, we see that the composition List has a nested type Element which is not directly visible in the composition User. However, since its visibility is not restricted to its containing type, we can still access it via its qualified name. For example, we could define an attribute with the type NamespacesExample.-List.Element<TYPE> or simply List.Element<TYPE> in the composition User.

5.1.3. Visibility Modifiers

As mentioned in the previous section, qualified names are used to access types and operations declared in libraries as well as attributes and operations in compositions. By default, these elements have no visibility modifier and are thus accessible. However, users may restrict their accessibility by using explicit visibility modifiers private and protected.

Visibility of types and operations in libraries

Operations and type declarations within a library can be hidden by explicitly adding the private modifier. Listing 41 demonstrates a library with two operations count and countHelper.

```
library VisibilityModifiersLibrary
import Lists

operation count(parameter numbers : list<integer>,
   parameter number : integer) : integer
   return countHelper(numbers, number, 0)
endoperation

private operation countHelper(parameter numbers : list<integer>,
                              parameter number : integer,
                              parameter pos : integer) : integer
   if pos = Lists.lengthOf(numbers) then
      return 0
   else
      if numbers[pos] = number then
         return countHelper(numbers, number, pos + 1) + 1
      else
         return countHelper(numbers, number, pos + 1)
      endif
   endif
endoperation
```

Listing 41: Visibility modifiers in a library.

The second operation counts the occurrences of a specific integer value in a list of integers recursively by passing the current position in the list as a parameter. The first operation calls the second one and basically acts as the user interface for that operation by hiding the parameter `pos` from the user. Thus, the operation `countHelper` is hidden by the visibility modifier `private` and the only operation that is visible from the outside is `count`. The program in listing 42 imports the library `VisibilityModifiersLibrary` and calls the operation `count` from it resulting in the output 4.

```
program VisibilityModifiers1
import VisibiliryModifiersLibrary
import IO

main
    variable numbers : list<integer>
    numbers := [3, 4, 1, 42, 0, 345, 2, 42, 42, 67, 0, 42]
    IO.writeInteger(VisibilityModifiersLibrary.count(numbers, 42))
endmain
```

Listing 42: Program using the library in listing 41.

Should the user attempt to invoke the operation `countHelper` from the imported library in this program an error message “Couldn’t resolve reference to NamedElement ’countHelper’.” would be displayed. Although the called operation is contained in the namespace of the library and can theoretically be accessed by its qualified name, it is marked by the visibility modifier `private` and is thus ignored by the scoping mechanism.

Visibility of Composition Members

Accessibility of composition members can also be restricted by visibility modifiers. Since compositions can inherit from other compositions, an additional modifier `protected` allows to hide an element to the outside but still make it accessible within the inheritance hierarchy. Listing 43 shows an example of a composition `Point3D` inheriting from the composition `Point2D`. The attributes `x`, `y` and `z` are hidden and therefore cannot be accessed in the main procedure by using their qualified names. However `Point3D` is able to access the inherited attributes `x` and `y` since they are marked as `protected`. Should another composition inherit from `Point3D` it would still be able to access `x` and `y` but not `z` since it is `private` and is therefore only visible inside `Point3D`.

MuLE lacks an explicit object constructor similar to those in Java, it offers a simplified value constructors for compositions instead, where the user can assign specific values to all attributes of the composition, including inherited ones, regardless of their visibility (lines 26 and 30). This might contradict with the concept of data abstraction, but on the other hand this offers a much simpler solution while still allowing to implement single assignment semantics for attributes (more in section 5.5.8).

The output of the program in listing 43 is:

```
Point:
    x = 2
    y = 3
Point:
    x = 2
    y = 3
    z = 4
```

```
program VisibilityModifiers2
import IO

type Point2D : composition
   protected attribute x : integer
   protected attribute y : integer

   operation print()
      IO.writeLine() IO.writeString("Point:")
      IO.writeLine() IO.writeString("\tx = ") IO.writeInteger(x)
      IO.writeLine() IO.writeString("\ty = ") IO.writeInteger(y)
   endoperation
endtype

type Point3D : composition extends Point2D
   private attribute z : integer

   override operation print()
      super.print()
      IO.writeLine() IO.writeString("\tz = ") IO.writeInteger(z)
   endoperation
endtype

main
   variable p2d : Point2D
   p2d := Point2D{x = 2, y = 3}
   p2d.print()

   variable p3d : Point3D
   p3d := Point3D{x = 2, y = 3, z = 4}
   p3d.print()
endmain
```

Listing 43: Restricting visibility of composition members.

Validation Checks

Following error messages can appear in the context of visibility modifiers:

- The visibility modifier protected is not allowed in this context. You may use private if you wish to prevent operations and type declarations to be exported from the library.

- A visibility modifier is not allowed in this context. You must not restrict visibility of operations and type declarations within a program unit.

- An overriding operation must have the same visibility modifier as the overridden operation.

5.2. Grammar Notation

The grammar of the language is represented by a sequence of production rules similar to the EBNF notation. The chosen notation is a simplified version of the notation [100] used to define the grammar of a DSL in the Xtext framework (see section 8.2), which was used to implement this language. Each rule is composed of a non-terminal symbol on the left-hand side and a set of terminal and non-terminal symbols on the right-hand side. Listing 44 is an excerpt of the actual MuLE grammar which shows examples of grammar rules which demonstrate the entire range of rules and operators used in this notation.

```
CompilationUnit:
    (’program’|’library’) ID
    Import*
    ProgramElement*
    MainProgram?;

Import: ’import’ [CompilationUnit];

INT: (’0’..’9’)+;

STRING: ’"’ ( ’\\’ . | !(’\\’|’"’) )* ’"’ ;

ML_COMMENT: ’/*’ -> ’*/’;
```

Listing 44: Examples of grammar rules in the chosen notation used in this chapter.

Following rules apply to this grammar notation:

- Terminal and non-terminal symbols are written with `typewriter` styled font.

- Terminal symbols are additionally denoted by ’blue color with single quotation marks’.

- By convention, rules that represent tokens are written in uppercase characters.

- `:` separates the left-hand from the right-hand side in a production rule.
- `;` terminates a production rule.
- `()` are used for their usual parentheses function, e.g. if an operator has to be applied to a set of operands.
- `|` is used to enumerate alternative elements, i.e. out of the listed elements, exactly one must be present.
- `?` represents an optional occurrence of an element, i.e. none or one element must be present.
- `*` represents an optional multiple occurrence of an element, i.e. none to infinite number of elements.
- `+` represents a multiple occurrence of an element, i.e. one to infinite number of elements.
- `x..y` represents a range between `x` and `y`, which can be integer numbers as well as lower or upper case alphabetical characters. The `INT` rule in listing 44 is represented as a multiple occurrence of a character defined by the range `'0'..'9'`, i.e. an integer literal consists of 1 to n digits ranging from `0` to `9` (both borders are included).
- `[Element]` represents a cross-reference to an existing element, for example the `import` instruction is defined as a reference to an existing `CompilationUnit`.
- `.` represents a wildcard, i.e. any possible character. For example, in the `STRING` rule in listing 44, the wildcard is used to represent any character preceded by a double backslash character, i.e. this combination represents escaped characters in a string including an escaped double quotation mark which would otherwise terminate the string literal.
- `!` represents a negated character, i.e. any character that is not explicitly stated after this operator. In the `STRING` rule in listing 44, this operator is used to represent any character other than a double backslash (which is covered by a combination with a wildcard) and the closing quotation marks. Basically, this operator is used to represent non-escaped characters in a string literal.
- `->` means that every character between two tokens is consumed as a part of this rule. The rule `ML_COMMENT` is defined by this operator, wherein any character between the characters specified in the rule is part of the multi-line comment.

Subsequent sections will focus on specific language constructs and will therefore contain short excerpts of the entire grammar of MuLE, which is additionally located in appendix B in its full form. Additionally to the grammar, the language is defined by scoping and validation rules. Scoping rules are explained in section 5.1 while validation rules are mentioned and explained whenever necessary for each respective language construct.

5.3. Lexical Units

Semantically related groups of characters form lexical units, the so called tokens, in the source code. In the process of tokenizing the longest legitimate sequence of characters is evaluated to a lexical unit. Tokens represent the smallest independent syntactic unit whose purpose is defined by the rules of the language. Following tokens exist in MuLE:

- Identifiers
- Keywords
- Operators
- Separators
- Literals
- Comments

Identifiers and keywords are separated by white spaces. Space characters, tabulators, line breaks and comments are all evaluated as white spaces.

5.3.1. Comments

Comments are ignored during compilation and have therefore no effect on the functionality of a program. They are used to write comments explaining chunks of code for the sake of maintainability and to temporarily disable lines of code for various purposes. There are two types of comments:

- `//` (double forward slash) symbolizes a line comment. The entire content of the line between this symbol and the following line break is treated as a comment.
- `/*` (forward slash with a star) initiates a block comment which is closed by `*/`. The entire content between these symbols, which can contain several lines, is treated as a comment. It is not possible to nest block comments.

```
ML_COMMENT: '/*' -> '*/';
SL_COMMENT: '//' !('\n'|'\r')* ('\r'? '\n')?;
```

Listing 45: Grammar rules used to define single- and multi-line comments.

Comments which are contained in a string literal are evaluated as a part of that literal. Listing 46 demonstrates a program with examples for both types of comments. The value of the variable is an empty string due to its default initialization. The output of the program is: `/* NO COMMENT */, // NO COMMENT`.

```
program comments
import IO

/*
 * block comment
 */
main
   variable var : string // line comment
   //  var := "Hello, world!"
   IO.writeString(var & "/* NO COMMENT */, // NO COMMENT")
endmain
```

Listing 46: Examples for comments.

5.3.2. Identifiers

Identifiers are assigned to named elements in order to identify them in a namespace (section 5.1.1) . Identifiers may contain an arbitrary number of upper and lower case characters of the English alphabet as well as digits and underscores. There is however one restriction, identifiers can not begin with a digit. An identifier represents a simple name of a named element. Additionally, a named element can be referenced by its qualified name (see section 5.1.2), which must include at least one identifier and may consist of several identifiers separated by a dot (`.`) character. The last identifier in a qualified name is the simple name of the referenced element, while the preceding identifiers are the elements containing the referenced element (such as type declarations and compilation units).

```
ID : ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
QualifiedName : ID ('.' ID)*;
```

Listing 47: The grammar rules for identifiers and qualified names.

When referencing an identifier, one must pay attention to the use of upper and lower case characters. As an example `Foo`, `foo` and `FOO` would represent three different identifiers in a program.

Validation Checks

Different elements must have unique identifiers in a single namespace in order to prevent ambiguity. Furthermore, the name of the compilation unit must not be the same as the name of an imported library. Finally, there is a number of reserved keywords (section 5.3.3), which must not be used as identifiers. Reserved keywords and types of the language Java [56] are also not allowed to be used since MuLE is compiled into this language (see chapter 8 for implementation details). Following error messages can be triggered in the context of an identifier:

- An element with such name already exists. Use a different name for this element.
- Use of this name is not allowed. You are attempting to use a reserved word as an identifier.
- Naming conflict, the names of the importing and imported compilation units must not be equal.

Examples

Listing 46 contains following identifiers:

- `comments` is the name of the compilation unit. This name must be equal to the name of the file, which is followed by the file extension `.mule`.
- `IO` is the name of the imported standard library.
- `writeString` is the name of the operation which is used to print a string on the console. The operation is contained in the standard library `IO`.
- `var` is the name of the declared variable.

Following character strings are legitimate identifiers:

- `_variable`
- `var1`
- `MAX_VALUE`

Examples of illegal identifiers:

- `2variable` – starts with a digit.
- `Tür` – uses a character `ü` that is not allowed.
- `integer` – is a reserved word in MuLE.
- `int` – is a reserved word in Java.

5.3.3. Keywords

Keywords are reserved words that fulfil a specific semantic function in the language. They cannot be used as identifiers.

- **program** – specifies that the corresponding compilation unit is a program unit, i.e. it must have a `main` procedure. This keyword is followed by the name of the program, this name must be equal to the name of the containing file.
- **library** – specifies that the compilation unit is a library, i.e. it can be imported by other compilation units and must not include a `main` procedure. This keyword is followed by the name of the program, this name must be equal to the name of the containing file.
- **import** – initiates an import statement. Import statements must be contained at the top level in a compilation unit.
- **main** – start of the main procedure.
- **endmain** – end of the main procedure.
- **integer** – primitive data type for integer numbers.
- **rational** – primitive data type for floating-point numbers.
- **string** – primitive data type for strings.
- **boolean** – primitive data type for boolean values.
- **reference** – has two meanings depending on its context. In a declaration of a data container or an operation this represents the reference type and must be parameterized by the type of the referenced value. In an expression it is used as a prefix operator which creates a reference to a value.
- **list** – represents the built-in parameterized list data type.
- **type** – start of a type declaration.
- **endtype** – end of a type declaration.
- **enumeration** – the type declaration is an enumeration.
- **composition** – the type declaration is a composition.
- **extends** – the composition is extending another composition. Alternatively it is used to restrict a type parameter to a specific type and its sub types.
- **operation** – start of an operation or of an operation type signature.

- **endoperation** – end of an operation.
- **override** – an operation is redefining another operation with the same name inherited from an extended composition.
- **super** – access to an overridden operation in a super type.
- **abstract** – is used to denote abstract types and operations.
- **private** – is used to prevent export of types and operations in libraries and restrict the visibility of attributes and operations in composite types to its immediate containing type.
- **protected** – is used to restrict the visibility of attributes and operations in composite types to its immediate containing type and its subtypes.
- **attribute** – declaration of an attribute in a composition.
- **parameter** – declaration of a parameter of an operation.
- **variable** – declaration of a variable.
- **return** – terminates an operation, may return a value if an operation has a return type.
- **exit** – terminates the immediate loop in which it is contained.
- **loop** – start of a `loop`-statement.
- **endloop** – end of a `loop`-statement.
- **if** – start of an `if`-statement.
- **then** – denotes the beginning of the body of an `if` or an `elseif` clause.
- **elseif** – start of an `elseif` clause in an `if`-statement.
- **else** – start of an `else` clause in an `if`-statement or in a `let`-statement.
- **endif** – end of an `if`-statement.
- **let** – start of a `let`-statement.
- **be** – is used in a `let`-statement after a variable declaration and before an expression, which is assigned to the declared variable.
- **do** – denotes the beginning of the body of a `let` or an `elselet` clause.
- **elselet** – start of an `elselet` clause in a `let`-statement.

- **endlet** – end of a `let`-statement.
- **null** – represents null-references.
- **true** – one of the literals for boolean values.
- **false** – one of the literals for boolean values.

5.3.4. Separators

Statements are separated by white spaces, i.e. spaces, tabulators and line breaks. Commas (`,`) are used to separate value parameters in operations and value constructors for lists and compositions, as well as type parameters in parameterized types. Examples are:

- `operation foo(parameter x : integer, parameter y : integer)` – separation of parameters in an operation declaration. Same applies to anonymous operations.
- `operation(integer, integer)` – type signature of the previous operation.
- `foo(2, 3)` – separation of parameters in an operation invocation.
- `Pair<T, R>` – separation of formal type parameters in a composition declaration.
- `variable p : Pair<integer, string>` – separation of actual type parameters in the type signature used in a variable declaration.
- `p := Pair{first = reference 42, second = reference "hello"}` – separation of attribute initialization instructions in a composition value constructor.
- `[1, 2, 3]` – separation of values in the value constructor of the built-in list type.

5.3.5. Operators

Operators are reserved tokens which apply an operation on one or two operands in an expression. The assignment operator is used in assignment statements (section 5.8). In order to eliminate ambiguity when evaluating expressions, operators are executed in a strict order. Table 2 shows all available MuLE operators, as well as their precedence and associativity with level 1 representing the highest priority.

Level	Operator	Description	Associativity
1	@	dereferencing	left to right
	[]	indexed access to list values	
	.	member access	
	()	parentheses	
2	not	negation of boolean expressions	none
	+	unary plus	
	-	unary minus	
3	reference	creating a reference value	none
4	exp	exponentiation	left to right
5	*	multiplication	left to right
	/	rational division	
	div	integer division	
	mod	modulo	
6	+	addition	left to right
	-	subtraction	
	&	string concatenation	
7	<	less than	none
	<=	less than or equal	
	>	greater than	
	>=	greater than or equal	
8	=	equal	left to right
	/=	not equal	
9	and	logical AND	left to right
10	or	logical OR	left to right
12	**	repetition operator, used in a list	none
	..	range operator, used in a list	
13	:=	assignment operator	none

Table 2: Operator precedence and associativity

5.3.6. Brackets

MuLE uses all four common types of brackets, each type is mostly reserved for specific language constructs (as explained in section 5).

- **Round brackets** `()` – are used as parentheses to manipulate operator precedence in expressions (for example `2 * (3 + 4)`) and to pass parameters to operations. This is the only type of brackets used for two semantically different purposes.

- **Square brackets** `[]` – are reserved for the built-in list type. They are used both to access stored values via their indices (e.g. list1D[0], list2D[1][4]) and in value constructors used to create values of this type.
- **Curly brackets** {} – are reserved solely for the value constructors of compositions.
- **Angle brackets** `<>` – are used to pass type parameters to parameterized types. It should be noted, that a use of a single `<` or `>` character is seen as comparison operators and not brackets, which are to be used in pairs.

5.3.7. Value Literals

A literal is the textual representation of a value of predefined basic types, the user defined enumeration type as well as the `null`-literal representing null-references (see chapter 5.5).

```
AtomicExpression:
    StringConstant | IntegerConstant | RationalConstant |
    BooleanConstant | Null | ... ;

StringConstant: STRING;
IntegerConstant: INTEGER;
RationalConstant: RATIONAL;
BooleanConstant: 'true' | 'false';
Null: 'null'

INT: ('0' .. '9')+;
INTEGER: INT;
RATIONAL: INT '.' INT ('E' ('+' | '-')? INT)?;
STRING: '"' ( '\\' . | !('\\'|'"') )* '"';
```

Listing 48: Simplified excerpt from the `AtomicExpression` rule demonstrating rules for value literals.

Integer Literals

Integer literals are represented using the decimal system, meaning that digits in the range [0 .. 9] can be used. Additional validation checks ensure that literals with more than one digit can not start with a zero. Additional value range restrictions apply to integer literals as a result of the semantic representation of the corresponding type (see section 5.5.3).

Valid `integer` literals: `0`, `2147483647`, `-2147483648`.
Invalid `integer` literals: `007`, `2147483648`, `-2147483649`.

Following error messages can be triggered in the context of an `integer` literal:

- Invalid literal, an integer literal with multiple digits must not start with a zero.
- The value is out of the supported range (-2147483648..+2147483647).

Floating Point Number Literals

Floating point number literals are also represented by the decimal system. The dot (`.`) acts as the separator between the integer and fractional parts. Scientific notation can be used to express large numbers, e.g. the literal `2.5E10` is equivalent to 2.5 * 10^{10}. The integer part of the literal must not begin with a zero if it contains more than one digit. Similar to integer literals, rational literals are restricted by the underlying data type representation (section 5.5.4).

Valid `rational` literals: `0.0`, `3.14`, `-0.25`, `1.5E23`, `0.5E-2`.
Invalid `rational` literals: `00.2`, `.23`, `1.`, `1.0E`, `1E100`.

Following error messages can be triggered in the context of a `rational` literal:

- Invalid literal, the integer part of a rational literal must not start with a zero when having multiple digits.
- The value is out of the supported range.

Boolean Literals

Literals representing boolean values are `true` and `false`.

String Literals

String literals are denoted by double quotation marks (`"`). Any symbol can be used between these marks, the backslash (\) symbol is used to represent escape sequences. The symbols between the quotation marks are encoded by the CP1252 standard [101]. Keywords, operators and delimiters used within a string literal do not have their reserved functionality and are thus recognised as a part of that string literal.

Escape sequences are used in order to correctly depict special symbols (quotation marks, backslash, line break, tabulator) within a string literal. Examples for escape sequences are shown in listing 49, the output of the program is:

```
Hello, world!
"Hello",    \world\
program stringLiterals
import IO

main
   IO.writeString("Hello, world!\n")
   IO.writeString("\"Hello\",\t\\world\\")
endmain
```

Listing 49: Examples of `string` literals and escape sequences.

Valid `string` literals: `""`, `"Hello, world!"`, `"a string with a line break\n"`. Invalid `string` literals: `"`, `"unfinished string\"`, `"wr\ong e\scape"`.

Literals of Enumeration Types

Enumerations are the only user defined primitive types in MuLE. Enumeration literals are identifiers and thus follow the same rules that are defined in section 5.3.2. They must begin with a lower or an upper case character of the English alphabet or alternatively with an underscore. Non-starting characters may additionally be a digit from 0 till 9.

Literal for Null-References

The value of a null-reference is represented by the literal `null`. Other reference values cannot be expressed by simple literals and must be represented by the corresponding expressions (see section 5.5.10).

5.4. Compilation Unit

A MuLE compilation unit can be either a `program` or a `library`. Programs must have a `main` procedure while libraries are not allowed to have one but can be imported via an `import` statement. Import statements are then written in the source code, the imported libraries are referenced by the name given to them. Type and operation declarations are placed after the import statements. The main procedure is always the last textual element in a MuLE program.

```
CompilationUnit:
   ('program' | 'library') ID
   Import*
   ProgramElement*
   MainProcedure?;

Import: 'import' [CompilationUnit];

ProgramElement: TypeDeclaration | Operation;

MainProcedure: 'main' Block 'endmain';
```

Listing 50: Composition of a MuLE compilation unit.

Example

Listing 51 demonstrates an example of a simple compilation unit declared as a `program` unit with the identifier `HelloWorld`. The unit imports the standard library `IO`, declares the operation `sayHello` and includes a `main` procedure.

```
program HelloWorld
import IO

operation sayHello()
   IO.writeString("Hello, world!")
endoperation

main
   sayHello()
endmain
```

Listing 51: Example of a *"Hello, World!"* programm.

Validation Checks

Following error messages can be triggered in the context of a compilation unit:

- Program name must be equal to the file name.
- Naming conflict, the names of the importing and imported compilation units must not be equal.
- A program must have a main operation.
- A library must not have a main operation.
- You can not import programs, only libraries are allowed to be imported.
- A visibility modifier is not allowed in this context. You must not restrict visibility of operations and type declarations within a program unit.
- The visibility modifier protected is not allowed in this context. You may use private if you wish to prevent operation and type declarations to be exported from the library.

5.5. Type System and Values

This section focuses on the underlying type system in our language. MuLE supports **strict static explicit** typing. Data containers and operations which return a value are explicitly given a type upon their declaration. Language constructs are checked for type validity at compile time. The textual representation of value literals for basic types is explained in section 5.3. This section includes textual representation for value constructors for types, which cannot be represented by single value literals.

As explained in section 4.2, each data type has a corresponding default value which is assigned to variables and uninitialized attributes after their declaration. For the sake of completeness of this section, we will mention the default value of each type in the respective subsections.

5.5.1. Typed Elements

Any value has a type, therefore any language construct which stores (i.e. data container) or otherwise yields (i.e. an expression or an operation) a value must have the corresponding type. Data containers and operations which return a value when invoked, are given types upon their declaration. It should be noted, that each operation has an operation type, which describes its signature and allows to pass operations as data. Literals for specific types are associated with their corresponding types, e.g. a `"Hello, world!"` is types as a `string` while `42` is an `integer`. Expressions (section 5.6) can combine several literals and references to data containers and operations connected by operators. If the combination of types and operators is legitimate, the expression returns a value, otherwise the program cannot be compiled. A type provider mechanism computes the type of an expression, that type is then used in compile time validation checks.

Following elements are considered typed elements:

- **Expressions** – in general, every expression is evaluated and a value is produced as a result. An expression can be a simple value (the type of the value is the type of the expression) or a chain of computations including value literals and constructors, data container references and operation invocations with various operators in between. In such cases, the operators are the decisive factor for the resulting type of the entire expression. The types of operands must be compatible to the types expected by the operators.

- **Value literals** – are lexical representations of values with basic types, i.e. integer, rational, boolean, string and enumerations. The literal `null` is a specific case for reference types representing the null-reference. Value literals represent the simplest possible expressions.

- **Value constructors** – are specific expressions that produce values for more complex types: compositions, lists, references and operations.

- **Parameterized data types** – parameterized types are basically are types which accepts other data types as parameters, e.g. a `reference<integer>` type is a reference with the type `integer`. Same applies to lists and parameterized compositions.

- **Types with super types** – Compositions and their type parameters can have a super type. A formal type parameter with a super type represents a restriction, i.e. only a type which is a subtype of the specified super type can be used as an actual type parameter.
- **Operations** – any named and anonymous operation has an `operation` type by itself which represents its signature, i.e. they can be passed as parameters or returned as values in higher-order functions. Operations with a return type produce a value when executed, meaning that they can be used as a part of an arithmetic or logical expression or in other cases when a non-`operation` type value is required.
- **Data containers** – such as variables, parameters and attributes are named elements meant to store data. They can be referenced in expressions in order to retrieve the previously stored values.

5.5.2. Definition of MuLE's Data Types

As defined in listing 52, MuLE offers a set of four predefined basic types (integer, rational, boolean and string), three user defined types (enumerations, compositions and type parameters of parameterized compositions), a parameterized list type, a parameterized reference type, and an operation type which represents operations and lambda expressions.

```
DataType: BasicType | DeclaredType | ReferenceType | ListType | OperationType;

BasicType: 'integer' | 'rational' | 'string' | 'boolean';

DeclaredType: [TypeDeclaration] ('<' DataType (',' DataType)* '>')?;

ReferenceType: 'reference' '<' DataType '>';

ListType: 'list' '<' DataType '>';

OperationType: 'operation' '(' (DataType (',' DataType)*)? ')' (':' DataType)?;

TypeDeclaration: Composition | Enumeration | TypeParameter;
```

Listing 52: Grammar rules for MuLE's data types.

Each type will be discussed in the following sections. Value literals for basic types were already mentioned in section 5.3.7.

5.5.3. Integer

Integer is one of the two basic numeric types in MuLE, its values are represented by the 32bit two's complement format allowing us to represent any integer between -2^{31} and $2^{31} - 1$. Literals that are not in that range are invalidated at compile time resulting in a compiler error. An overflow may still occur at runtime. The default value is 0.

Compatible Operators

Following operators can be used with this type:

- Binary (and unary in some cases) arithmetic operators `+`, `-`, `*`, `/`, `exp`, `mod`. It should be noted that expressions using the operators `/` and `exp` always result in a value with the type `rational`.
- Comparative operators `<`, `<=`, `>`, `>=`.
- Equality check operators `=`, `/=`.

Type Specific Operations

Furthermore, following operations are offered by the standard library `Mathematics` (section 6) specifically for this data type:

- `getMaxIntegerValue() : integer` – returns the biggest supported value ($2^{31} - 1$).
- `getMinIntegerValue() : integer` – returns the smallest supported value (-2^{31}).

5.5.4. Rational

The values of this type are represented by the 64bit IEEE 754 standard [96]. The smallest supported number is $4.9E - 324$ whereas $1.7976931348623157E308$ is the biggest. If a number is divided by zero, the resulting value is `Infinity`, same applies in case of overflow. In some cases, for example when dividing zero by zero or multiplying `Infinity` by zero, the result is `NaN` (Not a Number). In case of an underflow the resulting value is 0.0, which is also the default value for this type.

Compatible Operators

Following operators can be used with the values of this type:

- Binary (and in some cases unary) arithmetic operators `+`, `-`, `*`, `/`, `exp`, `mod`.
- Comparative operators `<`, `<=`, `>`, `>=`.
- Equality check operators `=`, `/=`.

Type Specific Operations

Furthermore following operations are offered by the standard library `Mathematics` (section 6) specifically for this data type:

- `getMaxRationalValue() : rational` – returns the biggest supported floating point number (1.7976931348623157E308).
- `getMinRationalValue() : rational` – returns the smallest supported floating point number (4.9E-324).

5.5.5. String

The default value is an empty string. Following operators can be used with values of this type:

- Equality check operators `=`, `/=`.
- string concatenation `&`.

The standard library `Strings` (section 6) offers a set of operations which allow values of other basic types to be transformed into strings (section 5.5.12).

Listing 53 demonstrates the use of string concatenations, string comparisons and a conversion of a boolean values to string. It should be noted, that the values 42 and 5 are compared as strings in this example and are not equal. Using comparative operators like `<` and `>=` would result in a compile time error. The output of the program is: `42 = 42 and 42 /= 5 is true`.

```
program stringsAndBooleans

import IO
import Strings

main
    IO.writeString("42 = 42 and 42 /= 5 is "
        & Strings.booleanToString("42" = "42" and "42" /= "5"))
endmain
```

Listing 53: Examples for operations with strings and boolean values.

5.5.6. Boolean

Logical truth values are represented by the data type `boolean`. These values can be either `true` or `false`. The default value is `false`.

Following operators can be used with this type:

- Equality check operators `=`, `/=`.
- Logical operators `and`, `or`, `not`.

5.5.7. Enumeration Type

Enumerations are user defined primitive types which define a limited set of of literals as eligible values of this type. User defined types must be declared within a program or an imported library. The keyword `type` is used to initiate a type declaration followed by the type's name and its kind, i.e. `enumeration` in this case. The body of the declaration contains the literals separated by commas. At least one literal must be present in the body of the declaration.

```
Enumeration:
   'type' ID ':' 'enumeration'
      EnumerationValue (',' EnumerationValue)*
   'endtype';

EnumerationValue: ID;
```

Listing 54: Grammar rules for enumerations and its values.

```
program Enumerations
import IO

type Figure : enumeration
   PAWN, ROOK, KNIGHT, BISHOP, QUEEN, KING
endtype

main
   variable figure : Figure
   figure := Figure.KNIGHT
   IO.writeBoolean(figure = Figure.KNIGHT)
   IO.writeLine()
   IO.writeBoolean(figure = Figure.PAWN)
endmain
```

Listing 55: Example of enumeration type.

An example is given in listing 55. When accessing an enumeration value users must use its qualified name. Values of enumeration types can be compared using `=` and `/=` with each other. The output of the program is:

```
true
false
```

The default value of an enumeration is the first literal in the corresponding type declaration, in case of the program in listing 55, the default value is `Figure.PAWN`.

Validation Checks

Following error messages can be triggered in the context of an enumeration:

- An element with such name already exists. Use a different name for this element.
- The visibility modifier protected is not allowed in this context. You may use private if you wish to prevent operations and type declarations to be exported from the library.
- A visibility modifier is not allowed in this context. You must not restrict visibility of operations and type declarations within a program unit.

5.5.8. Compositions

Compositions are user defined types similar to enumerations (see 5.5.7). In its basic form, compositions represent a structured collection of named values with potentially different types called `attributes`. Additionally, compositions may be used for object-oriented programming by including operations within a composition and supporting further concepts of this paradigm. The default value of a composition is defined by the default values of its attributes depending on their types.

```
Composition:
    VisibilityModifier? ('abstract')? 'type' ID
    ('<' TypeParameter (',' TypeParameter)* '>')? ':' 'composition'
    ('extends' [Composition] ('<' TypeParameter (',' TypeParameter)* '>')? )?
        TypeDeclaration*
        Attribute*
        Operation*
    'endtype';

TypeParameter: ID ('extends' QualifiedName)?;

Attribute: VisibilityModifier? 'attribute' ID ':' DataType;
```

Listing 56: Grammar rules for compositions, attributes and type parameters.

As shown in the grammar rule for compositions in listing 56, every composition has an identifier and is marked by keywords `type` and `composition`. A visibility modifier can be assigned to a composition declaration. A composition may be abstract and it may extend other compositions. Formal type parameters can be passed to compositions, every formal type parameter may be declared as a subtype of another composition.

Composition members are nested type declarations (compositions and enumerations), attributes and operations. By default, every member can be accessed from the outside via its qualified name. Visibility modifiers `private` and `protected` can restrict access to composition members strictly to the encompassing type or additionally to its subtypes respectively.

Value Constructors and String Representation

Unlike values of primitive types, values of compositions cannot be represented by a single literal. The grammar rule for value constructors for this type is shown in listing 57. The value constructor is part of the SymbolReference rule. The SymbolRefAccessModifier? part of this rule is of no interest to us in this section, it is used for indexed access to list values, operation invocations and dereferencing, validation rules prevent its usage in the context of composition value constructors. SymbolReference is an expression which is used whenever a named element must be referenced. In this case, a composition type is referenced as a part of the corresponding value constructor, which consists of the type identifier followed by an optional number of attribute initializations enclosed in curly brackets. An attribute initialization references an existing attribute by its identifier and initializes it with a value produced by the corresponding expression.

```
AtomicExpression: SymbolReference | ... ;

SymbolReference:
    [NamedElement] (SymbolRefCompositionInit)? SymbolRefAccessModifier?
        ('.' SymbolReference)?;

SymbolRefCompositionInit:
    '{' (SymbolRefCompositionAttribute
        (',' SymbolRefCompositionAttribute)*)? '}';

SymbolRefCompositionAttribute: [Attribute] '=' Expression;
```

Listing 57: Grammar rules for value constructors of compositions.

All attributes are visible in the value constructor, regardless of their visibility modifiers, in order to allow correct initialization of hidden attributes. For each composition with at least one attribute, two value constructors are allowed: an empty constructor without attribute initialization instructions, and a full constructor wherein every attribute of this type including the inherited attributes is initialized. Listing 58 shows an example of a simple composition with two attributes x and y. The allowed signatures for value constructors are in this case: Vector2D{} and Vector2D{x = EXPRESSION, y = EXPRESSION}. The empty constructor initializes the attributes with the default values for each respective type, visible attributes may be manually accessed and their values changed at a later stage. In the example in listing 58, we have used a non-empty constructor Vector2D{x = 1, y = 5}. We have enhanced this example with object-oriented constructs in listing 59, wherein we have added an additional composition Vector3D which acts as a subtype of Vector2D by adding an additional dimension. The attributes in both types are now hidden. The corresponding value constructor signatures remain the same for Vector2D, while for Vector3D they are Vector3D{} and Vector3D{x = EXPRESSION, y = EXPRESSION, z =

EXPRESSION}.

Composition values can be transformed into string values by using the `Strings` library operation `genericToString(EXPRESSION)`. The string representation is equivalent to the notation used in non-empty compositions value constructors. Same representation is seen when inspecting variable values of composition types in the debugger. The string representation for the default value of a `Vector2D` type is `Vector2D{x = 0.0, y = 0.0}`, if we initialize the attributes `x` and `y` with 2 and 3 respectively, the representation becomes `Vector2D{x = 2.0, y = 3.0}`.

Compositions in Procedural Programming

Listing 58 shows an example of a composition type used in a procedural way. A composition `Vector2D` with two attributes `x` and `y` as well as an operation which accepts values of this type and prints the data in a readable way are defined in the compilation unit. A variable with the type `Vector2D` is declared in the main procedure an initialized by using the value constructor for compositions, i.e. `Vector2D{x = 1, y = 5}` in this case.

```
program CompositionExample
import IO

type Vector2D : composition
   attribute x : rational
   attribute y : rational
endtype

operation print(parameter v : Vector2D)
   IO.writeString("\nVector:")
   IO.writeString("\n   x = ") IO.writeRational(v.x)
   IO.writeString("\n   y = ") IO.writeRational(v.y)
endoperation

main
   variable v : Vector2D
   v := Vector2D{x = 1, y = 5}
   print(v)
   v.x := 2
   v.y := 3
   print(v)
endmain
```

Listing 58: An example of a composition type used in a procedural program.

After printing the value of the variable by using the previously defined procedure, the values of the attributes are changed via their qualified names and the print procedure is called again.

The output of the program is:

```
Vector:
    x = 1.0
    y = 5.0
Vector:
    x = 2.0
    y = 3.0
```

Compositions in Object-Oriented Programming

Following rules apply to compositions in accordance with the concepts of object-oriented programming:

- A composition can include operations.
- A composition can include nested type declarations, i.e. other compositions and enumerations.
- A composition can be abstract, in which case it is marked by the keyword `abstract` and may include abstract operations.
- Abstract compositions cannot be instantiated and must be wrapped as a reference type when used to declare variables and attributes.
- Compositions may inherit from another composition, only single inheritance is allowed.
- Inherited operations may be overridden, in which case the redefining operation must be denoted by the keyword `override`.
- The keyword `super` can be used to access overridden methods from the immediate super type.
- An operation can be abstract, in which case it is marked by the keyword `abstract` and is not allowed to have an operation body. Such an operation must be overridden in a non abstract subtype.
- Composition members, i.e. inner type declarations, attributes and operations, are per default visible to the outside. Their visibility can be restricted by visibility modifiers (see section 5.1.3) `private` (the element is only visible within the composition, e.g. the attribute `z` in `Vector3D`) and `protected` (the element is visible within the composition and its subtypes, e.g. the attributes `x` and `y` in `Vector2D`).
- Compositions may accept type parameters which are substituted with actual types at runtime enabling parametric polymorphism.

```
program Vectors
import IO

abstract type Vector : composition
   abstract operation getLength() : rational
   abstract operation print()
endtype

type Vector2D : composition extends Vector
   protected attribute x : rational
   protected attribute y : rational

   override operation getLength() : rational
      return (x exp 2 + y exp 2) exp 0.5
   endoperation

   override operation print()
      IO.writeString("\nVector:")
      IO.writeString("\n   length = ")   IO.writeRational(getLength())
      IO.writeString("\n   x = ")        IO.writeRational(x)
      IO.writeString("\n   y = ")        IO.writeRational(y)
   endoperation
endtype

type Vector3D : composition extends Vector2D
   private attribute z : rational

   override operation getLength() : rational
      return (x exp 2 + y exp 2 + z exp 2) exp 0.5
   endoperation

   override operation print()
      super.print()
      IO.writeString("\n   z = ")        IO.writeRational(z)
   endoperation
endtype

type Client : composition
   operation test(parameter v : reference<Vector>)
      v@.print()
   endoperation
endtype

main
   variable v2d : reference<Vector2D>   v2d := reference Vector2D{x = 2, y = 3}
   variable v3d : reference<Vector3D>   v3d := reference Vector3D{x = 2, y = 3, z = 4}

   variable client : Client
   client.test(v2d)     client.test(v3d)
endmain
```

Listing 59: Vector example with additional OO concepts.

Listing 59 demonstrates how object-oriented concepts can be implemented in MuLE by using compositions. The output of the program is:

```
Vector:
    length = 3.605551275463989
    x = 2.0
    y = 3.0
Vector:
    length = 5.385164807134504
    x = 2.0
    y = 3.0
    z = 4.0
```

Validation Checks

Placeholders in error messages represented by `[...]` are replaced by context specific information at runtime. Following error messages can be triggered in the context of a composition, its members and type parameters:

- An element with such name already exists. Use a different name for this element.
- The visibility modifier protected is not allowed in this context.
  You may use private if you wish to prevent operations and type declarations to be exported from the library.
- A visibility modifier is not allowed in this context.
  You must not restrict visibility of operations and type declarations within a program unit.
- Cyclic inheritance is not permitted.
  Check the following type declarations and remove the cycle in the inheritance: `[TYPES CONTRIBUTING TO CYCLIC INHERITANCE]`.
- Composition inherits unimplemented abstract operations: `[UNIMPLEMENTED OPERATIONS]`.
- Composition declaration must include all inherited type parameters.
  Inherited type parameters are: `[INHERITED TYPE PARAMETERS]`.
  Owned type parameters are: `[OWNED TYPE PARAMETERS]`.
- Declaration of a nested composition must include all type parameters of its outer composition.
- An attribute must not have the same type as the containing composition.
  You can use a reference type instead.

- Cyclic referencing of compositions detected, use reference types.
- An abstract type is not allowed to be used directly in a feature declaration. Either use a non-abstract type or wrap the abstract type as a reference type.
- A type parameter is not allowed to be used directly in a feature declaration. Either use a non-abstract type or wrap the type parameter as a reference type.
- Incompatible type was used for type parameter.
  Expected type: `[EXPECTED TYPES]`
  Actual type: `[ACTUAL TYPE]`

5.5.9. Lists

MuLE offers a built in parameterized collection type called `list`. Both lexical constructs and library operations are included with the language to facilitate a flexible use of this data type. Upon declaration, a data container with a list type must be given an actual type parameter which determines which elements are allowed to be stored in the list. Passing another list as an actual type parameter allows to create multi dimensional lists, i.e. lists of lists (example in listing 61, line 21). The number of dimensions is not limited by the language.

```
ListType: 'list' '<' DataType '>';

AtomicExpression: SymbolReference | ListInit | ... ;

SymbolReference:
    [NamedElement] (SymbolRefCompositionInit)? SymbolRefAccessModifier?
        ('.' SymbolReference)?;

SymbolRefAccessModifier: ListAccess | ... ;

ListAccess: '[' Expression ']' SymbolRefAccessModifier?

ListInit: "[" (Expression (ListInitFunction | ListInitElements))? "]";

ListInitFunction: ("**" | "..") Expression;

ListInitElements: ("," Expression)*;
```

Listing 60: Grammar rules related to the list type and the corresponding expressions.

Stored values are accessed via their indices. The starting index is 0, the last element has the index $list_{length} - 1$. When accessing an entry in the list, the copy of that entry is returned. It should be noted that it is not possible to access a list entry via its index directly on a value constructor notation for lists, that means that `[1 .. 9][0]` is not a legal expression and would lead to a compile error instead of returning a 1 as a result. The default value of this type is an empty list.

Value Constructors and String Representation

Three different value constructor notations can be used to initialize a list. Various notations can be mixed when initializing multidimensional lists.

- **List of elements** – the corresponding elements are separated with commas. An empty list represents a special case.
- **Range of integer values** – fills the list with the numbers in the range of `[min .. max]`, both `min` and `max` are included and must be integer values, the increment is 1. If the value of `max` is lesser than that of `min`, this expression results in an empty list.
- **Repetition** – has following syntax: `[number ** element]`. The list is filled with the given `number` of elements, the `number` must be an integer value. If the value of `number` is lesser than 1, this expression results in an empty list.

String representation of list values displayed in the debug view or by converting list values to string resembles the list of elements notation, i.e. all elements are listed separated by commas and enclosed in square brackets. For example, a list initialized as `[1 .. 4]` is represented by the string `[1, 2, 3, 4]`.

Validation Checks

Placeholders in error messages represented by `[...]` are replaced by context specific information at runtime. Most validation checks performed on list related language constructs are concerned with type correspondence, i.e. error messages are displayed if the type of an expression does not correspond to the expected type of its context. Additionally, an error message may be displayed if the user attempts to use indexed access in the wrong context.

- Expected type is: `[DATA TYPE]`. Actual type is: `[DATA TYPE]`.
- Invalid access modifier, you must not use list access on a non-list type.

Example Program

Listing 61 shows examples for list declarations, standard notations and expressions using list values. List values can be checked for equality, two lists are equal if both lists have the same number of entries, all entries have the same type, the corresponding entries have the same value and their order in the corresponding lists is the same.

```
program lists
import IO
import Strings

main
    variable empty : list<string>
    empty := []
    variable elements : list<string>
    elements := ["a", "b", "c"]                 // ["a", "b", "c"]
    elements[1] := "x"                          // ["a", "x", "c"]
    variable repetition : list<rational>
    repetition := [3 ** 3.14]                   // [3.14, 3.14, 3.14]

    variable list1 : list<integer>
    list1 := [0 .. 2]                           // [0, 1, 2]
    variable list2 : list<integer>
    list2 := [3, 4, 5]                          // [3, 4, 5]
    variable lists : list<list<integer>>
    lists := [list1, list2]                     // [[0, 1, 2], [3, 4, 5]]
    lists := [[0, 1, 2], [2 ** 3], [4 .. 6]]    // [[0, 1, 2], [3, 3], [4, 5, 6]]

    IO.writeBoolean(list1 /= list2)
    IO.writeLine()
    IO.writeBoolean(list1 = lists[0])
    IO.writeLine()
    IO.writeString(Strings.genericToString(lists))
endmain
```

Listing 61: Examples for MuLE lists.

The output of the program is:

```
true
true
[[0, 1, 2], [3, 3], [4, 5, 6]]
```

5.5.10. References

MuLE reference types are parameterized types which allow to encapsulate any value of any type as a reference. A variable with the type `reference<integer>` will only accept references to integer values. Data containers with reference types store the address of a value stored in the memory. A reference which does not reference anything is a null-reference, which is also the default value for this type.

```
ReferenceType: 'reference' '<' DataType '>';

AtomicExpression: SymbolReference | Null | Reference | ... ;

SymbolReference:
   [NamedElement] (SymbolRefCompositionInit)? SymbolRefAccessModifier?
      ('.' SymbolReference)?;

SymbolRefAccessModifier: Dereference | ... ;

Dereference: '@' SymbolRefAccessModifier?

Reference: 'reference' AtomicExpression;

Null: 'null';
```

Listing 62: Grammar rules related to the reference type and the corresponding expressions.

Value Constructors and String Representation

Null-references are represented by the literal `null`. Reference values are explicitly created with the reference creation expressions consisting of the keyword `reference` combined with an expression. The value of the expression is stored in the memory, the value of the reference creation expression is the address of the stored value. To access this value, the data container with the stored reference must be explicitly dereferenced with the postfix operator `@`. One must be aware of null-references when dereferencing otherwise a runtime error may occur.

The string representation of a reference type consists of the stored value followed by the identity hash code [102], which is used to uniquely identify elements stored on the heap. An example representation of a value created by the expression `reference 42` can be `42@404b9385`.

Example Programs

An example of using reference types is given in listing 63. The variables `a` and `b` are declared and initialized in the `main` procedure. Their values are then swapped in the operation `swap`. This change is visible after the execution of the `swap` operation in the `main` procedure. The output of the program is: `5 42`.

```
program references
import IO

operation swap(parameter x : reference<integer>, parameter y : reference<integer>)
   variable z : integer
   z := x@    x@ := y@      y@ := z
endoperation
```

```
main
    variable a : reference<integer>
    variable b : reference<integer>
    a := reference 42
    b := reference 5
    swap(a, b)
    IO.writeInteger(a@)  IO.writeString(" ")
    IO.writeInteger(b@)  IO.writeLine()
endmain
```

Listing 63: Example of a procedure using reference types.

Listing 64 demonstrates the behaviour for equality checks of references. The output of the program is:

```
false
true
true
```

The values of references are addresses, which are compared in the equality check in line 10 explaining why `false` is the result for the first case. Both references address different memory objects whose values are, however, the same, which is why `ref1@ = ref2@` results in `true`. After reassigning `ref2` to `ref1`, meaning that both references now point to the same memory object, the expression `ref1 = ref2` is evaluated as `true`. Additional details are presented in section 5.5.13.

```
1   program references2
2   import IO
3
4   main
5       variable ref1 : reference<integer>
6       variable ref2 : reference<integer>
7
8       ref1 := reference 42
9       ref2 := reference 42
10      IO.writeBoolean(ref1 = ref2)    IO.writeLine()
11      IO.writeBoolean(ref1@ = ref2@)   IO.writeLine()
12
13      ref2 := ref1
14      IO.writeBoolean(ref1 = ref2)    IO.writeLine()
15  endmain
```

Listing 64: Equality checks with references.

Listing 65 shows the copy semantics when creating references. The `integer` variable `num` is assigned the value 42. When creating a reference, this value is copied first and the copy is then referenced by `ref`. Therefore changing the value of `num` does not have any side effects on the value referenced by `ref`. The output of the program is `42`.

```
program references3
import IO

main
    variable ref : reference<integer>
    variable num : integer

    num := 42
    ref := reference num
    num := 21

    IO.writeInteger(ref@)
endmain
```

Listing 65: Copy semantics with references.

Validation Checks

Placeholders in error messages represented by [...] are replaced by context specific information at runtime. Validation checks ensure that correct value is passed to the reference creation expression depending on the expected type. Following error messages may be displayed in the context of reference types and the corresponding expressions.

- Expected type is: [DATA TYPE]. Actual type is: [DATA TYPE].
- Cannot assign a null-reference to a non-reference type.
- Cannot compare a non-reference type to a null-reference.
- Invalid access modifier, you must not dereference a non-reference type.

5.5.11. Operations

Named and anonymous operations are represented by the operation type, which resembles the operation signature, i.e. its parameters and the return type. This type allows to handle operations as data in the context of functional programming, meaning that operations can be stored as normal values in data containers. The "value" of an operation is its semantics.

```
OperationType: 'operation' '(' (DataType (',' DataType)*)? ')' (':' DataType)?;

Operation:
    VisibilityModifier? ('abstract')? ('override')? 'operation' ID
        '(' (Parameter (',' Parameter)*)? ')' (':' DataType)?
            (Block 'endoperation')?;

AtomicExpression: SymbolReference | LambdaExpression | ... ;
```

```
LambdaExpression:
    ’operation’ ’(’ (Parameter (’,’ Parameter)*)? ’)’ (’:’ DataType)?
       Block ’endoperation’;

SymbolReference:
    [NamedElement] (SymbolRefCompositionInit)? SymbolRefAccessModifier?
       (’.’ SymbolReference)?;

SymbolRefAccessModifier: OperationInvocation | ... ;

OperationInvocation:
    ’(’ (Expression (’,’ Expression)*)? ’)’ SymbolRefAccessModifier?
```

Listing 66: Grammar rules related to the operation type and the corresponding language constructs.

Example Program

Listing 67 shows a simple example of using the operation type. The operation `foo` and the variable `bar` have both the operation type `operation(integer) : integer`, hence why `foo` can be assigned to `bar`, making `bar` effectively an alias of `foo`. This allows us to pass named operations as parameters to higher-order functions. The lambda expression assigned to `bar` in line 15 also has the same type. The output of the program is: `4 4 4`.

The operation type should not be confused with the return type of an operation. For example, the type of the expression `foo` is `operation(integer) : integer` while the type of `foo(2)` is `integer`.

```
1  program OperationTypes
2  import IO
3
4  operation foo(parameter x : integer) : integer
5     return x * x
6  endoperation
7
8  main
9     IO.writeInteger(foo(2)) IO.writeString(" ")
10
11    variable bar : operation(integer) :  integer
12    bar := foo
13    IO.writeInteger(bar(2)) IO.writeString(" ")
14
15    bar := operation(parameter x : integer) : integer
16       return x * x
17    endoperation
18    IO.writeInteger(bar(2))
19 endmain
```

Listing 67: Operation types allow to use named and anonymous operations as data.

Validation Checks

The operation type is used to ensure that operations with correct signatures are passed to data containers, therefore, the usual error message “Expected type is: `[DATA TYPE]`. Actual type is: `[DATA TYPE]`.” is displayed whenever incompatible types are used in the same context. Other than that, section 5.9 covers operations as a language construct, thus the corresponding validation messages will be discussed there.

5.5.12. Type Compatibility and Conversion Rules

This section covers rules concerned with the type compatibility and type conversion.

Type Compatibility Rules

Whenever an expression is evaluated or a value is passed to a data container, the type of the value has to be determined and then compared to the expected type of the context in which it is used. For example, in the context of an assignment statement, the type of the assigned value must be compatible with the type of the data container. In an arithmetic expression, values on both sides of the expression must have numerical types, etc. When the actual type is compatible to the expected type, the program can be executed as long as no other rules are broken. Otherwise, if the actual type of the value is not compatible with the expected type of the context, the following compile time error message is displayed and the program will not compile:

“Expected type is: `[DATA TYPE]`. Actual type is: `[DATA TYPE]`.”

Let us assume T_1 is the expected type and T_2 is the actual type. In general, T_2 is *compatible* to T_1 if it is a reflexive or transitive subtype of T_1. Following type compatibility rules apply, left-hand side shows the expected type while the right-hand side lists compatible types and additional constraints:

- `boolean` – `boolean`.
- `string` – `string`.
- `integer` – `integer`.
- `rational` – `rational`, `integer`.
- `enumeration` – exact the same `enumeration` type contained in the same compilation unit. Two enumeration types with the same name and same literals but contained in different compilation units are not compatible.

- `list<TYPE1>` – `list<TYPE2>`, whereby `TYPE1` and `TYPE2` must be compatible.

- `reference<TYPE1>` – `reference<TYPE2>`, whereby `TYPE1` and `TYPE2` must be compatible, unless they are list types, in which case they must be equal (explanation is given in section 5.5.13).

- `composition1` – `composition2`, whereby `composition2` is a transitive or reflexive subtype of `composition1`. Let us assume that `Point3D` is a subtype of `Point2D`, `p2d` is a variable of type `Point2D` and `p3d` is a variable of type `Point3D`. This means that the assignment `p2d := Point3D{x = 1, y = 2, z = 3}` is a valid statement (more in section 5.5.13). However, the assignment `p3d := Point2D{x = 1, y = 2}` will result in a compile time error message since the expected type is `Point3D` which is inferred from the left side. Since `Point2D` is not a subtype of `Point3D`, it is not a compatible type in this case.

- `formal type parameter` – another `formal type parameter` with the same name if used within the body of the same composition, where the expected `formal type parameter` was declared. Outside of the body of that composition, formal type parameters are replaced by actual type parameters, which have to follow the rules described above.

In case of equality check expressions, types on both sides must be compatible, however, it does not matter whether the subtype is written on the right-hand or on the left-hand side. Fir example, both `Point2D{x = 1, y = 2} = Point3D{x = 1, y = 2, z = 3}` and `Point3D{x = 1, y = 2, z = 3} = Point2D{x = 1, y = 2}` as well as `42 = 42.0` and `42.0 = 42` are valid expressions which will all evaluate to `true`.

Type Conversion Rules

MuLE's mechanisms for type conversions are designed to prevent illegal type conversions and resulting runtime errors. Subtypes may be implicitly converted to super types. Rational is handled as a super type of integer, furthermore compositions may have a super type via inheritance relations. Explicit type conversions are supported by library operations which convert values of other types to string values and rational values to integers.

Following library (section 6) operations are used for explicit type conversions:

- `Strings` – `integerToString(integer) : string` – converts an integer value to a string value.

- `Strings` – `rationalToString(rational) : string` – converts a rational value to a string value.

- `Strings` – `booleanToString(boolean) : string` – converts a boolean value to a string value.
- `Strings` – `genericToString(Expression) : string` – converts values of any type to a string value, which allows to conveniently convert values of compositions and lists into their textual representations. Example is shown in listing 68.
- `Mathematics` – `floor(rational) : integer` – converts a rational value to integer by dropping the decimal expansion of the number.
- `Mathematics` – `round(rational) : integer` – converts a rational value to integer by rounding it.

Furthermore, MuLE offers `let`-statements (section 5.8.6) which check if a value is an instance of a specific type in an inheritance hierarchy and execute the corresponding code if this is the case. These statements are meant to be used in the context of object-oriented programming and reference types when the static type of a declared value is not enough and dynamic type checks must be performed by the user.

```
program LetStatements
import IO
import Strings

type Point2D : composition
    attribute x : integer
    attribute y : integer
endtype

type Point3D : composition extends Point2D
    attribute z : integer
endtype

main
    variable v1 : reference<Point2D>    v1 := reference Point2D{}
    variable v2 : reference<Point2D>    v2 := reference Point3D{}

    let variable v : reference<Point3D> be v1 do
        IO.writeString(Strings.genericToString(v@) & "\n")
    endlet

    let variable v : reference<Point3D> be v2 do
        IO.writeString(Strings.genericToString(v@) & "\n")
    endlet
endmain
```

Listing 68: Explicit conversion with let-statements

Listing 68 demonstrates examples of explicit type conversion, `let`-statements are used to check whether the dynamic types of the variables `v1` and `v2` are compatible to the type of the variable `v` which is declared in the scope of the respective `let`-statements. Both variables have the static type `reference<Point2D>` but different dynamic types. In both statements the variable `v` is declared as `reference<Point3D>`. In the first case, the dynamic type of `v1` is the super type of the expected type, meaning that we would need to convert the stored value into a subtype with further attributes. This is not possible, therefore the body of the first `let`-statement is not executed. However, even though the static type of `v2` is `reference<Point2D>`, the dynamic type matches the expected type in the second `let`-statement, which is then executed.

The output of the program is: `Point3D{x = 0, y = 0, z = 0}`.

5.5.13. Value Copying Semantics and Equality Rules

We have already given an overview over value copying semantics in section 4.2 by giving two examples when passing integer values and references to integer values. As a reminder, whenever a value is passed to a data container, it is copied and the copy is then stored. Therefore, all data containers are always storing different data chunks, even if their values might be equal. For example, if we declare two integer variables `a` and `b`, initialize `a` as `42` and then assign value of `a` to `b`, the `42` is copied and the copy is stored in `b`. Assigning a different value to one variable will not alter the state of the other one. When we use reference types, a value is stored in the memory and then the address to that value is copied and passed over to data containers. In this case, we may have two variables referencing the same value in the memory allowing us to change the state of one variable by altering the other one.

In short, whenever data is passed to a data container, a shallow copy of the data object is created and passed on, i.e. normal values are copied while referenced values are not.

Value Copying Rules

Following rules apply when passing a value depending on its type:

- The actual type of the value must be compatible to the expected type. Otherwise, the program will not compile. This rule applies to all types.

- Values of basic types, i.e. integer, rational, boolean, string and enumerations, are copied as they are. In case of integer values, if the expected type is rational, then the integer value is converted to rational without changing its actual numerical value (i.e. a `42` becomes a `42.0`).

- Values of compositions can be copied, if the type of the copied value is a reflexive or transitive subtype of the data container. During the copying process, first an empty value of the target type is created, then the values of attributes of the actual value are copied recursively into the new value according to the copying rules respective to the types of each value. If the target type has less attributes than the source type (e.g. if a `Point3D` is copied to a `Point2D`), then only the shared attributes are transferred.

- Values of reference types are addresses to actual values stored in the memory. A reference may be copied if the type of the source value is compatible to the referenced target type. For example, a `reference<Point3D>` may be copied to a `reference<Point2D>`. If a reference is dereferenced, than the source type is the type of the actual value in the memory, i.e. normal value copying semantics apply respective for that type. A null-reference does not point to a data object in the memory. A null-reference can be assigned to any reference, if no other reference addresses the previously referenced value, the value is removed from the memory by the garbage collector.
- Values of the built-in list type are copied according to the type of listed values. First, an empty list with the same type as that of the source list is created, then it is filled with recursive copies of the values from the source list.
- Values of operation types are semantically different from usual values, after all, operations represent behaviour and not data. Thus, a copy of an operation can be described as a lambda expression with the exact same signature and behaviour as the source operation.

Examples for Value Copying Semantics

For the sake of all code examples in this section, we will use the declarations of `Point2D` and its subtype `Point3D` in listing 68. The first example (listing 69 and figure 13) demonstrates a similar scenario to the one explained above, which was a repetition of the example shown in section 4.2. However, this time we are using the aforementioned compositions `Point2D` (variable `v1`) and `Point3D` (variable `v2`).

Once declared, both variables are initialized with their default values based on their types (state 1). We then initialize `v2` using the value constructor for composition types (state 2), and assign the now new value of `v2` to `v1` (state 3). Since the type of the value of `v2` is a subtype of the expected type in the assignment statement in line 6, this is a legitimate assignment. However the entire value of `v2` cannot be stored under `v1`, therefore, a value of type `Point2D` is created under `v1` and the values of the shared attributes are copied over. Finally, if we alter the internal state of one of the variables (state 4), the state of the other variable remains unchanged. Same would happen if the types of both variables were equal, with the difference that values of all attributes would be carried over.

```
1  main
2      variable v1 : Point2D
3      variable v2 : Point3D
4
5      v2 := Point3D{x = 1, y = 2, z = 3}
6      v1 := v2
7      v2.x := 5
8  endmain
```

Listing 69: Value copying semantics with composition types

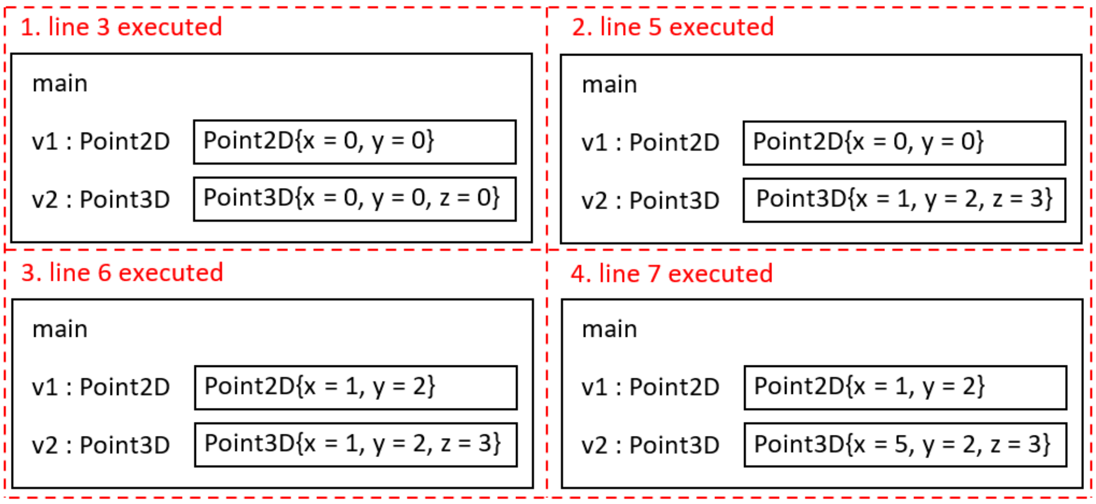


Figure 13: State of memory at different times of the execution of the program in listing 69.

Listing 70 and figure 14 demonstrate an example using reference types. Immediately after their declaration, both variables `v1` and `v2` are initialized with their standard values, which is a null-reference in both cases (state 1). We continue by assigning these two variables values based on their types using the `reference` expression, the values are stored in the memory and their referencing addresses are stored in the variables (state 2). Since `Point2D` is a super type of `Point3D`, `reference<Point3D>` is compatible to `reference<Point2D>`, meaning that we can assign the value of `v2` to `v1` (state 3). Both variables are now addressing the same value in the memory, i.e. `v1` and `v2` are equal. The static type of `v1` is still `reference<Point2D>` while its dynamic type is now `reference<Point3D>`, meaning that we cannot access the attribute `z` via the variable `v1` although it does exist in the referenced value. The value `Point2D{x = 1, y = 2}` is no longer referenced, the reserved memory will be freed by garbage collector. Altering the state of the referenced value via one of the variables will have an effect for both variables (state 4). After the program will finish its execution, none of the remaining values will be referenced any longer and their reserved memory will be freed by the garbage collector.

```
main
    variable v1 : reference<Point2D>
    variable v2 : reference<Point3D>

    v1 := reference Point2D{x = 1, y = 2}
    v2 := reference Point3D{x = 1, y = 2, z = 3}

    v1 := v2
    v2@.x := 5
endmain
```

Listing 70: Value copying semantics with reference types and compositions

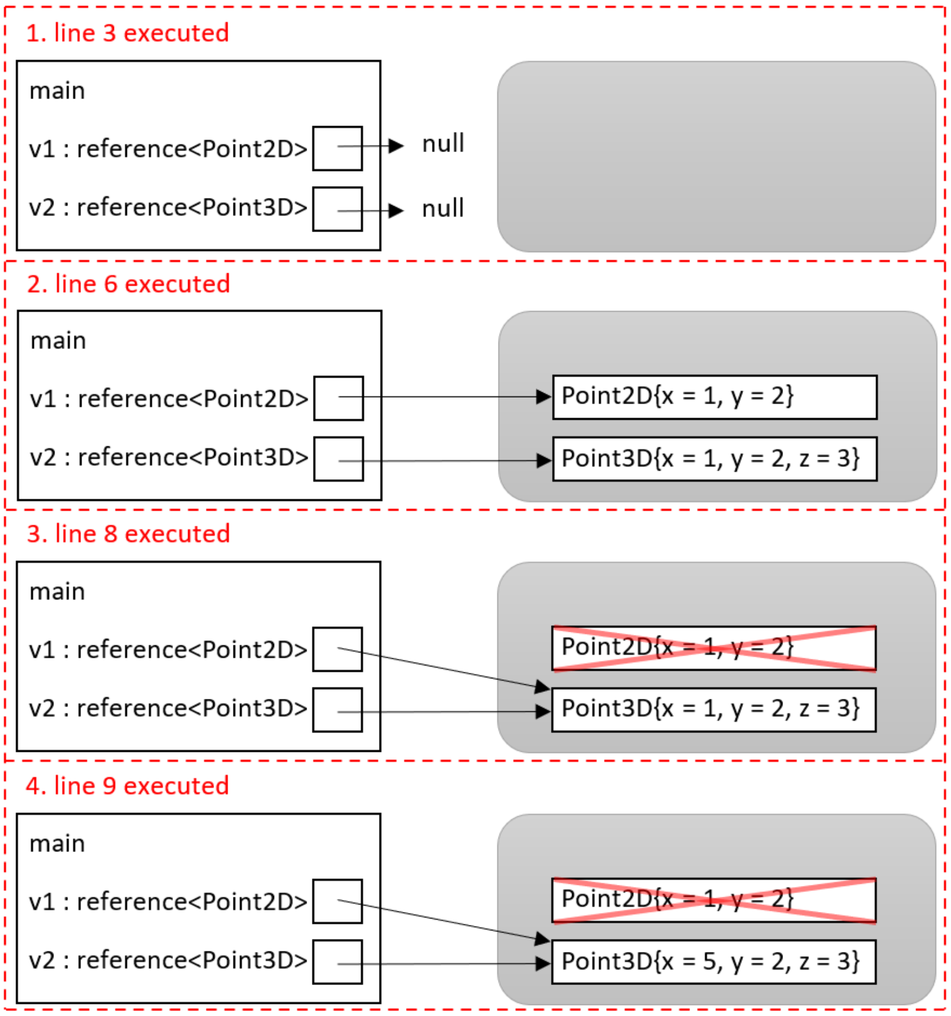


Figure 14: State of memory at different times of the execution of the program in listing 70.

```
1  main
2      variable l1 : list<reference<Point2D>>
3      variable l2 : list<reference<Point2D>>
4
5      l1 := [reference Point2D{}, reference Point2D{}]
6      l2 := l1
7      l2[0]@.x := 3
8      l1[1] := null
9  endmain
```

Listing 71: Example of a shallow copy.

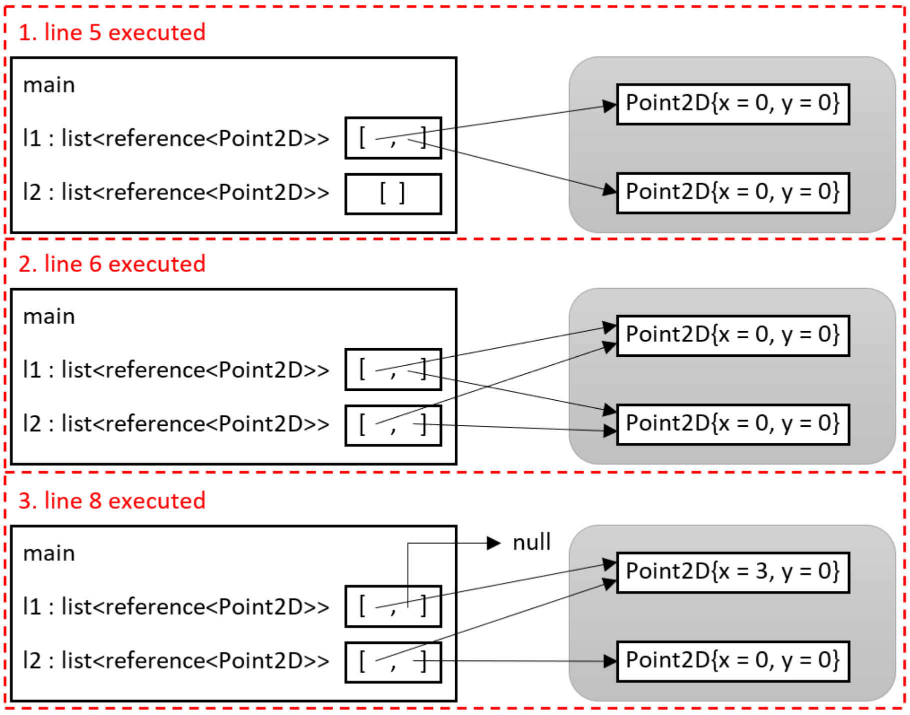


Figure 15: State of memory at different times of the execution of the program in listing 71.

Listing 71 and figure 15 show a concrete example of a creation of a shallow copy when passing a composition or a list value which contains references to other values, in this example we have lists of references. Both variables l1 and l2 are declared as `list<reference<Point2D>>`, `l1` is initialized with a concrete value while l2 keeps its default value, i.e. an empty list (state 1). We then assign the value of `l1` to `l2`, the list stored in `l1` is copied as well as the references which are stored in the list, however, the referenced values in the memory are not copied (state 2). We now have two equal list values which store references to the same values in the memory. If we now access one of the references from one list, dereference it and manipulate the value, as seen in line 7, the effect is visible in the other list (state 3). However, changing one of the stored references directly, for example by accessing a stored reference and assigning it a null-reference is line 8, will not affect the other list (state 3).

```
main
    variable v1 : reference<list<Point2D>>
    variable v2 : reference<list<Point3D>>

    v2 := reference [Point3D{x = 1, y = 1, z = 1}, Point3D{x = 2, y = 2, z = 2}]
    v1 := v2   // error:      Expected type is: reference<list<Point2D>>
               //             Actual type is: reference<list<Point3D>>
    v1@[0] := Point2D{x = 1, y = 1}
    IO.writeInteger(v2@[0].z)
endmain
```

Listing 72: Type incompatibility in case of references to lists of compositions.

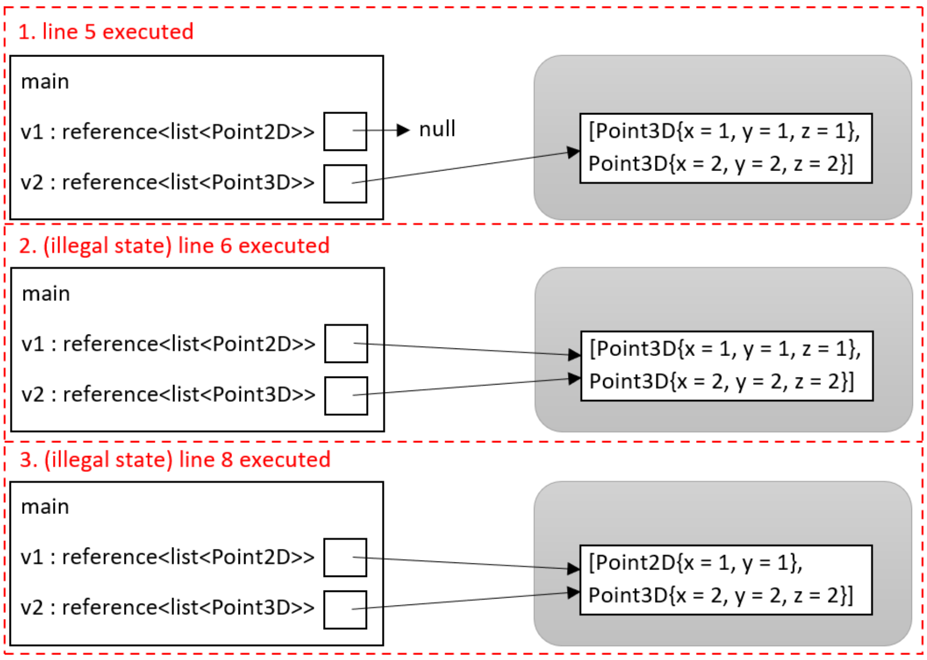

1. line 5 executed
main
v1 : reference<list<Point2D>>
null
v2 : reference<list<Point3D>>
[Point3D{x = 1, y = 1, z = 1},
Point3D{x = 2, y = 2, z = 2}]
2. (illegal state) line 6 executed
main
v1 : reference<list<Point2D>>
v2 : reference<list<Point3D>>
[Point3D{x = 1, y = 1, z = 1},
Point3D{x = 2, y = 2, z = 2}]
3. (illegal state) line 8 executed
main
v1 : reference<list<Point2D>>
v2 : reference<list<Point3D>>
[Point2D{x = 1, y = 1},
Point3D{x = 2, y = 2, z = 2}]

Figure 16: Assumed state of memory at different times of the theoretical execution of the program in listing 72.

A rather different example is shown in listing 72 and figure 16. Even though the types in line 6 are compatible at the first glance, the program does not compile. As stated in the type compatibility rules previously in section 5.5.12, if the expected type is `reference<list<T1>>` and the actual type is `reference<list<T2>>`, then `T1` and `T2` must be equal. The reason for this is shown in figure 16, which demonstrates the state of the memory if the program in listing 72 would actually be executed. The variable `v2` is initialized as a reference to a list filled with instances of `Point3D` (state 1). If it would be possible to assign `v2` to `v1`, we would create two aliases with different static types to the same list (state 2). We would now be able to store values of `Point2D` in the list via the variable `v1` (state 3). This leads to an issue when accessing these values via the variable v2, which offers access attributes which are no longer present in some of the values (line 8 in listing 72).

Value Equality Rules

Following rules apply when performing an equality check on two values:

- The types on both sides of the equality check expression must be compatible with each other. Otherwise, the program will not compile. This rule applies to all types.
- Rules for boolean values are simple: `true = true`, `false = false`.
- Two string values are equal, if both values have the same number of characters, the characters' order of appearance is equal, and each character is equal.
- Two integer or rational values are equal, if they represent the same numerical value. For example, both expressions `42 = 42.0` and `10 = 1.0E1` yield true.
- Two enumeration values are equal, if both literals are equal.
- Two references are equal, if both references address the same value in the memory. A null-reference is equal to the null-literal.
- Two list values are equal, if both lists have the same number of values, the order of the values is the same and each corresponding value is equal.
- Two compositions are equal, if the values of shared attributes are equal. Since the types on both sides must be compatible but not equal to each other, the equality check `Point2D{x = 1, y = 2} = Point3D{x = 1, y = 2, z = 3}` will be evaluated as `true`.

5.6. Expressions

Expressions are syntactic units which represent the simplest possible instructions in MuLE. Expressions are evaluated and return a value but do not change the state of a program on their own, unless the expression is a call to a memory altering operation. Furthermore expressions can be contained in other expressions but the root expression has to be contained in a statement (section 5.8).

The Expression Grammar

Listing 73 contains the entire expression grammar of MuLE. Excerpts and shortened versions of this grammar rules were already presented in previous sections. Literals for the various primitive types as well as the null literal are not included, the corresponding rules are presented in chapter 5.3.

The expression grammar can be separated into two parts, the first one includes binary expressions which yield a value by applying a binary operator to two values. The second one contains atomic expressions, which represent value literals, value constructors, apply a unary operator to a value, as well as the parenthesized expression.

```
Expression:
    OrExpression;

OrExpression:
    AndExpression ('or' AndExpression)*;

AndExpression:
    EqualityExpression ('and' EqualityExpression)*;

EqualityExpression:
    ComparisonExpression (('=' | '/=') ComparisonExpression)*;

ComparisonExpression:
    AdditiveExpression (('<' | '<=' | '>' | '>=') AdditiveExpression)*;

AdditiveExpression:
    MultiplicativeExpression (('+' | '-' | '&') MultiplicativeExpression)*;

MultiplicativeExpression:
    ExponentExpression (('*' | '/' | 'div' | 'mod') ExponentExpression)*;

ExponentExpression:
    AtomicExpression (('exp') AtomicExpression)*;

AtomicExpression:
        SymbolReference | SuperExpression | StringConstant | IntegerConstant |
        RationalConstant | BooleanConstant | Null | Unary | Reference |
        ParenthesizedExpression | ListInit | LambdaExpression;

Unary:
    ('+'|'-'|'not') AtomicExpression;

Reference:
    'reference' AtomicExpression;
```

```
ParenthesizedExpression:
    '(' Expression ')';

LambdaExpression:
    'operation' '(' (Parameter (',' Parameter)*)? ')' (':' DataType)?
        Block 'endoperation';

SuperExpression:
    'super' '.' SymbolReference;

SymbolReference:
    [NamedElement] (SymbolRefCompositionInit)? SymbolRefAccessModifier?
        ('.' SymbolReference)?;

SymbolRefAccessModifier:
    OperationInvocation | ListAccess | Dereference;

OperationInvocation:
    '(' (Expression (',' Expression)*)? ')' SymbolRefAccessModifier?;

ListAccess:
    '[' Expression ']' SymbolRefAccessModifier?;

Dereference:
     '@' SymbolRefAccessModifier?;

SymbolRefCompositionInit:
    '{' (SymbolRefCompositionAttribute
        (',' SymbolRefCompositionAttribute)*)? '}';

SymbolRefCompositionAttribute:
    [Attribute] '=' Expression;

ListInit:
    "[" (Expression (ListInitFunction | ListInitElements))? "]";

ListInitFunction:
    ("**" | "..") Expression;

ListInitElements:
    ("," Expression)*;
```

Listing 73: The expression grammar.

Semantics and Examples of Expressions

Operators are an integral part of most expressions. They define the function of the corresponding expression, the order in which the expressions are evaluated as well as which operands can be accepted by validation checks. See section 5.3.5 for the table of operators, their function and precedence.

Following list presents a summary of all supported expressions, their functionality and requirements:

- **Logical OR** – boolean values are required as operands. The result is `false`, if both operands are `false`. Otherwise the result is always `true`.
  `a or b, true or false`

- **Logical AND** – boolean values are required as operands. The result is true if both operands are true and false in any other case.
  `a and b, true and false`

- **Logical not** – negates a boolean value.
  `not a, not false`

- **Equality checks** – are performed to check if two values are equal or not. Both operands must have compatible types. The result is a boolean value.
  `a = b, a /= b, 5 = 2, 5 /= 2`

- **Comparative checks** – are performed to evaluate how the value on the left-hand side of the expression compares to the value on the right-hand side. Both operands must be numerical values, i.e. either rational or integer. The result is a boolean value.
  `a < b, a <= b, a >= b, a > b`
  `2 < 4, 2 <= 4, 2 >= 4, 2 > 4`

- **Addition and subtraction** – are used to add or subtract two numerical values. If one of the operands is a rational number, the result also also rational. If both operands are integer values, the result is also integer.
  `a + b, a - b, 2 + 4, 2 - 4`

- **Unary signs** – are used to apply a unary `+` or `-` to a numerical value. Unary operators can be chained, whereby the innermost operator is applied first.
  `-2, +4, --2`

- **String concatenation** – both operands must be strings. The result is a string value composed of both original values.
  `a & b, "Hello " & "world!"`

- **Multiplication** – is used to multiply the left-hand value by the right-hand value. Both operands must be either integer or rational. The resulting value is an integer if both operands are integer. As soon as one of the operands has the type rational, the resulting value is also rational.
  `a * b, 2 * 4`

- **Rational division** – is used to divide the left-hand value by the right-hand value. Both operands must be either integer or rational. The resulting value is always rational, e.g. `5 / 2` will result in `2.5`.
  ```
  a / b, 2 / 4
  ```
- **Integer division** – is used to divide the left-hand value by the right-hand value. Both operands must be integers. The resulting value is always integer, e.g. `5 div 2` will result in `2`.
  ```
  a div b, 2 div 4
  ```
- **Modulo** – is used to compute the remainder of the left-hand value divided by the right-hand value. Both operands must be numerical values. The result is an integer if both values are integers and rational in other cases.
  ```
  a mod b, 4 mod 2
  ```
- **Exponentiation** – computes the left-hand side to the power of the right-hand side. Both operands must be numerical values. The resulting value is always rational.
  ```
  a exp b, 4 exp 2
  ```
- **Parenteses** – can be used for their usual function in mathematical and boolean expressions.
  ```
  2 * (5 - 3), a and (b or c)
  ```
- **Named element call** – is used when a named element, e.g. a variable or an operation, has to be referenced in the source code. If the named element cannot be accessed directly, i.e. it is not in the immediate scope, its qualified name must be used.
  ```
  a, a.x
  ```
  Operation calls can be used both as a part of an expression or stand-alone as a statement (more details in section 5.8.3). Operations without a return value cannot be called in an expression.
  ```
  IO.writeString("Hello, world!"), 42 + sum(2, 5), foo()
  ```
- **Reference value constructor** – is used to create a reference type of a value. The value is copied and wrapped in a reference, meaning that the manipulations of the referenced value do not cause any effects on the original value.
  ```
  reference b, reference 42
  ```
- **Dereferencing** – is used to get the value of a reference which is stored in a data container, i.e. a variable, an attribute or a parameter.
  ```
  a@
  ```
- **List value constructor** – there are three distinct ways to express a list value in MuLE. The first one as a set of values separated by commas. The

second option is to create a repetition of values, wherein the value on the right-hand side (which can be of any type) is repeated by the amount on the left-hand side (integer). The ∗∗ operator is used in this case. And finally, a range of integers can be defined by using the .. operator with the corresponding borders represented by integer values. Square brackets are used in all cases.

```
[a, b, c], [a ** b], [a ..  b]
```

- **List access** – if a named element represents a list value, the stored values can be accessed via their index. In this case the index is written in square brackets after the identifier of the named element. In case of multi dimensional lists several list access modifiers can be written after the list name.

```
a[1], table[0][1]
```

- **Super expression** – is used if a redefined operation has to be invoked in a sub type.

```
super.foo()
```

- **Composition value constructor** – is used to create values of composition types. It consists of the identifier of the composition followed by attribute value assignments in curled brackets. The attributes are written in order of their declaration in the composition (including inherited attributes, which come first) separated by commas. Two variants exist for each composition representing an empty and a non-empty value constructor.

```
Vector2D{}, Vector2D{x = 2, y = 3}
```

- **Lambda expression** – represents anonymous operations which can be passed as data to data containers. The corresponding data container is then used to invoke the stored operation. The syntax of a lambda expression is very similar to that of an operation construct, same applies to the performed validation checks. Section 5.9 covers the specification of operations in detail, the corresponding operation type is discussed in section 5.5.11, while the rules of operation invocations are covered in section 5.8.3.

```
operation(parameter x : integer) : integer
    return x * x
endoperation
```

Validation Checks

The expected and provided types of MuLE's expressions are listed above. Each part of an expression is checked for type compatibility. Following error messages may appear in the context of an expression at compile time. As usual, text elements marked as `[...]` are placeholders which are filled with context specific information at runtime. Rules that apply to operations (see section 5.9) are also applied for lambda expressions, except for the duplicate name error, since lambda expressions do not have an identifier by themselves.

- Couldn't resolve reference to NamedElement `[IDENTIFIER]`.
- Expected type is: `[DATA TYPE]`. Actual type is: `[DATA TYPE]`.
- Cannot compare a non-reference type to a null-reference.
- You must not invoke an abstract operation in the context of a super expression.
- You must not use an abstract composition in order to create a value. Use one of its subtypes instead.
- The referred element is a composition, it requires a composition initialization notation. The correct notation is either: `[EMPTY VALUE CONSTRUCTOR]` or `[NON-EMPTY VALUE CONSTRUCTOR]`.
- Incorrect attribute sequence used in the composition value constructor. The correct notation is either: `[EMPTY VALUE CONSTRUCTOR]` or `[NON-EMPTY VALUE CONSTRUCTOR]`
- You must not use operation invocation notation on a composition value constructor.
- You must not use list index access notation on a composition value constructor.
- You must not dereference a composition value constructor.
- The referred element is not a composition, this notation is not allowed in this context.
- You must not pass an operation which is a member of a composition to a data container since it may rely on other composition members not accessible in this context.
- The referred element is an operation, either invoke it with a parameter profile or pass it without any access modifiers to a data container with an operation type.
- Invalid access modifier, you must not dereference a non-reference type.
- Invalid access modifier, you must not use list access on a non-list type.

- Invalid access modifier, you must not use operation invocation on a non-operation type.

- Incomplete expression, you have to access a type declaration or an operation from the imported library via its qualified name.

5.7. Blocks

A block is a group of statements (section 5.8) which is attached to another language construct, i.e. another statement, an operation or the main procedure. In MuLE, a block may contain only statements, therefore the body of a composition, which contains attribute declarations and may contain operations, is strictly speaking not a block, whereas it may be defined as a block in other programming languages.

```
Block:
    Statement*;

MainProcedure:
    'main' Block 'endmain';

LoopStatement:
    'loop' Block 'endloop';
```

Listing 74: Grammar rule for blocks and examples for its uses.

As an example, the statements between the keywords `loop` and `endloop` represent the block of statements attached to that `loop`-statement. The `main` procedure represents a special kind of an operation and also has a block of statements enclosed by the keywords `main` and `endmain`. Any number of statements may be included in a single block.

Following language constructs include one or more blocks of statements:

- The main procedure.

- Non-abstract named operations.

- Lambda expressions.

- Loop-statements.

- If-statements, each branch of an if-statement has its own block.

- Let-statements, each branch of a let-statement has its own block.

5.8. Statements

Statements are language constructs which represent single executable actions in a program. Statements express the behaviour of a program step by step. They define the flow and may change the state of the program. In MuLE, statements are structured in blocks (section 5.7).

```
Statement:
    VariableDeclaration | AssignmentOrOperationCall | IfStatement |
    LoopStatement | LetStatement | ReturnStatement | ExitStatement;

VariableDeclaration: 'variable' ID ':' DataType;

AssignmentOrOperationCall: (SymbolReference | SuperExpression) (':=' Expression)?;

LoopStatement: 'loop' Block 'endloop';

IfStatement:
    'if' Expression 'then' Block
    ('elseif' Expression 'then' Block)*
    ('else' Block)?
    'endif';

LetStatement:
    'let' VariableDeclaration 'be' Expression 'do' Block
    ('elselet' VariableDeclaration 'be' Expression 'do' Block)*
    ('else' Block)?
    'endlet';

Return: 'return' (Expression)?;

ExitStatement: 'exit';
```

Listing 75: Grammar rules for statements.

Actual computations and logical operations are defined by expressions (section 5.6) which are part of some statements. For instance, if a value has to be stored in a data container, an assignment statement can be used wherein an expression is first evaluated and the resulting value is then stored in the data container.

Grammar rules of supported statements are displayed in listing 75. Subsequent sections will offer more details on each type of statement. The rule `AssignmentOrOperationCall` is listed under statements since it is used for assignment statements (which reference a named element as a `SymbolReference` expression rule on the left-hand side of the assignment statement) and invocations of operations (referred by `SymbolReference` or `SuperExpression`) at statement level.

5.8.1. Variable Declaration Statement

Before a variable may be used, it must be declared via a variable declaration statement. The syntax of a variable declaration statement is similar to that of attribute and parameter declarations. The statement starts with the keyword `variable` followed by the identifier of the variable and its data type.

```
VariableDeclaration: 'variable' ID ':' DataType;
```

Listing 76: The grammar rule of the variable declaration statement.

When a variable is declared, a type dependent default value is automatically assigned to it. Listing 77 shows examples of variables declarations for various MuLE types. Default values of compositions, enumerations and operations are dependent on the type declaration or its signature. Abstract types and composition type parameters cannot be used to declare variables directly since no default values exist for such types. Instead, they must be wrapped as reference types (line 14). Finally, a variable declaration is also part of every `let`-statement, more in section 5.8.6.

```
 1  program VariableDeclarations
 2
 3  type RGB : enumeration
 4     RED, GREEN, BLUE
 5  endtype
 6
 7  type Point2D : composition
 8     attribute x : integer
 9     attribute y : integer
10  endtype
11
12  type Wrapper<T> : composition
13     operation foo()
14        variable v0 : reference<T>     // null
15     endoperation
16  endtype
17
18  main
19     variable v1 : integer               // 0
20     variable v2 : rational              // 0.0
21     variable v3 : boolean               // false
22     variable v4 : string                // empty string
23     variable v5 : list<integer>        // []
24     variable v6 : reference<integer>    // null
25     variable v7 : RGB                   // RED
26     variable v8 : Point2D                // Point2D{x = 0, y = 0}
27     variable v9 : operation() :  integer // returns 0 when invoked
28  endmain
```

Listing 77: Examples of variable declarations and their default values.

Validation Checks

Following error messages may appear in the context of a variable declaration statement at compile time. As usual, text elements marked as `[...]` are placeholders which are filled with context specific information at runtime.

- Use of this name is not allowed. You are attempting to use a reserved word as an identifier.
- An element with such name already exists. Use a different name for this element.
- Couldn't resolve reference to TypeDeclaration `[IDENTIFIER]`.
- An abstract type is not allowed to be used directly in a feature declaration. Either use a non-abstract type or wrap the abstract type as a reference type.
- A type parameter is not allowed to be used directly in a feature declaration. Either use a non-abstract type or wrap the type parameter as a reference type.
- Incompatible type was used for type parameter.
  Expected type: `[EXPECTED TYPES]`
  Actual type: `[ACTUAL TYPE]`

5.8.2. Assignment Statement

Assignment statements are used to change the value of a data container, i.e. a variable, a composition attribute or an operation parameter. The data container is located on the left side of an assignment statement, followed by an assignment operator `:=` and an expression on the right side.

```
AssignmentOrOperationCall: (SymbolReference | SuperExpression) (':=' Expression)?;
```

Listing 78: The grammar rule of the assignment statement.

According to the grammar rule, an applicable statement is evaluated as an assignment statement if an assignment operator with the corresponding expression is present, otherwise this rule is meant to represent operation invocation statements (see section 5.8.3). Validation checks invalidate any attempts to assign values to operation calls. Multiple assignments are not allowed within one statement. After the expression is evaluated, a shallow copy of the resulting value is created and stored in the referenced data container.

Listing 79 shows examples for assignments to variables, composition attributes and operation parameters. The type of the expression on the right side must be compatible to the type of the data container on the left side.

```
program Assignments

type Point2D : composition
   attribute x : integer
   attribute y : integer
   operation setX(parameter _x : integer)
      x := _x               // Assignment to an attribute
   endoperation
endtype

operation increment(parameter num : integer) : integer
   num := num + 1           // Assignment to a parameter
   return num
endoperation

main
   variable p : Point2D
   p := Point2D{x = 1, y = 1} // Assignment to a variable
   p.y := 3              // Assignment to an attribute via its qualified name
endmain
```

Listing 79: Examples of assignments.

Validation Checks

In addition to the usual error messages which can be caused by invalid expressions which are part of every assignment statement, following error messages may appear in the context of an assignment statement at compile time. As usual, text elements marked as `[...]` are placeholders which are filled with context specific information at runtime.

- Couldn't resolve reference to NamedElement `[IDENTIFIER]`.
- Expected type is: `[DATA TYPE]`. Actual type is: `[DATA TYPE]`.
- You are not allowed to assign a value to an operation invocation statement.
- You are not allowed to assign a null-reference to a non-reference type.

5.8.3. Operation Invocation

Referring to an operation simply by its name does not invoke the operation. Instead, it is handled as data, i.e. it can be passed to data containers. In order to actually execute an operation, it must be invoked by passing a list of parameters, even if this list is empty. To invoke an operation, the list of the passed parameters must have the same length as the list of parameters specified in the parameter profile of the operation declaration. Furthermore, the types of each passed parameters must correspond to the types in the same order as specified in the parameter profile. For example, if we have an operation with the signature `foo(integer, boolean)`, a correct invocation could be `foo(5, true)`, while `foo()`, `foo(true, 5)` and `foo("5", true)` are all incorrect. Finally, if the operation is a procedure, i.e. it has no return type, it can not be invoked in an expression, only as a statement. Operations with a return type can be invoked both as expressions and statements.

The syntax of operation invocation instructions is defined by the expression grammar rule `SymbolReference` (listing 73 in section 5.6, a shortened version is also displayed is listing 80) which allows to invoke functions, i.e. operations with a return value, as a part of other expressions. In order to invoke operations without a return value, this expression must be used as a statement, which is achieved by the rule `AssignmentOrOperationCall`, assignments begin as a symbol reference and are interpreted as assignments only if the symbol reference is followed by the assignment operator and an expression. Validation mechanisms invalidate the statement, if an assignment is performed on an operation invocation construct or if the symbol reference is neither followed by an operation invocation nor an assignment.

```
AssignmentOrOperationCall: (SymbolReference | SuperExpression) (':=' Expression)?;

SuperExpression: 'super' '.' SymbolReference;

SymbolReference:
    [NamedElement] (SymbolRefCompositionInit)? SymbolRefAccessModifier?
        ('.' SymbolReference)?;

SymbolRefAccessModifier: OperationInvocation | ... ;

OperationInvocation:
    '(' (Expression (',' Expression)*)? ')' SymbolRefAccessModifier?;
```

Listing 80: The grammar rules used to invoke operations.

Static and Dynamic Binding

When invoking operations in the context of a composition type value, one must consider that the operation may be redefined in an inheritance hierarchy, i.e. several specifications of an operation may be given. When using non-reference types, the static type, i.e. the one specified in the declaration of a data container, is checked when deciding which operation is executed. In case of non-reference types, the type of the value is always same as the static type.

However, when using reference types, the dynamic type of the referenced value may deviate from the static type, i.e. the dynamic type may be a subtype of the static type. In that case, the operations are bound dynamically, i.e. the type of the referenced object is relevant when deciding which operation will be executed. If the dynamic type specifies a concrete implementation of the invoked operation, i.e. it redefines an inherited operation, than this operation will be executed.

In both cases, if the type in question does not redefine an inherited operation and this operation is invoked, the inheritance hierarchy is traversed up until a definition of the required operation is found, which is then executed. An example demonstrating both cases is shown in the next subsection in listing 82.

Examples

Various examples of operation calls are demonstrated in listings 81, 82 and 83. The simplest example is the invocation of the operation `bar` in listings 81 (line 13). The operation expects a string parameter, which is passed in order to execute the operation. This example is representative of procedural programming.

Since the operation `bar` is declared within the scope of the invocation statement in line 13, it can be referred simply by its name. Alternatively we could also invoke it by its qualified name which is `OperationInvocationExample1.bar` in this case. However, if an operation was not defined within the scope of the invoking statement, which is the case when calling imported library operations, the qualified name must be used. An example is shown in line 9, where the operation `writeString` is invoked via its qualified name.

If an operation is referred without the parameter profile in an expression, it is handled as data, which is more common in functional programming. This is the case in line 5, where we return the operation `bar` from the operation `foo`. The operation `foo` is invoked in line 14, since the returned value is the operation `bar`, we need to pass another list of parameters in order to invoke the returned operation. The output of the program is: `Hello, world!`

```
1   program OperationInvocationExample1
2   import IO
3
4   operation foo() : operation(string)
5      return bar        // returning an operation as data
6   endoperation
7
8   operation bar(parameter str : string)
9      IO.writeString(str) // invocation of an imported library operation
10  endoperation
11
12  main
13     bar("Hello, ")    // invocation of an operation
14     foo()("world!")   // invocation of an operation returned from another operation
15  endmain
```

Listing 81: First example of operation calls.

```
 1 program OperationInvocationExample2
 2 import IO
 3 
 4 type A : composition
 5    operation baz()
 6       IO.writeString("A.baz invoked") // invocation of a library operation
 7    endoperation
 8 endtype
 9 
10 type B : composition extends A
11    override operation baz()
12       IO.writeString("B.baz invoked, ") // invocation of a library operation
13       super.baz()   // invocation of an overridden operation in a super type
14    endoperation
15 endtype
16 
17 main
18    variable a : A
19    a.baz() IO.writeLine() // invocation of an operation defined in a composition
20 
21    variable b : B
22    b.baz() IO.writeLine() // non-reference types are used for now
23 
24    a := b // the type does not change
25    a.baz() IO.writeLine()
26 
27    variable aRef : reference<A>
28    aRef := reference a // invocation on reference types, dynamic type is checked
29    aRef@.baz() IO.writeLine()
30    aRef := reference b // the dynamic type deviates from the static type
31    aRef@.baz() IO.writeLine()
32 endmain
```

Listing 82: Second example of operation calls.

Listing 82 shows examples of operation invocation instructions in the context of object-oriented programming. Here we have got two types: `A`, which specifies an operation `baz`, and its subtype `B`, which overrides the inherited operation `baz`. The overriding operation invokes the inherited operation by using the `super` expression to access it, otherwise we would have a recursive call of this operation.

Operations defined within compositions are either invoked directly within the defining type hierarchy, or on instances of these types. The latter case is demonstrated in line 19, where we invoke the operation `baz` on the variable `b`, which is an instance of type `B`. This means that we first invoke the operation `baz` as defined in type `B`, which then invokes the inherited operation `baz`.

The output of the program is:

```
A.baz invoked
B.baz invoked, A.baz invoked
A.baz invoked
A.baz invoked
B.baz invoked, A.baz invoked
```

The first three lines in the output are caused by invocations on non-reference types, i.e. the the operations are bound statically in this case. Even when we assign the value of `b` to `a`, the types remain the same. However, the last two lines are caused by invoking the operation on reference types. At first, the static and the dynamic types are equal, i.e. `reference<A>` in line 28, resulting in the output as specified in type `A`. In line 30 the static type remains `reference<A>`, we still have the same variable after all, while the dynamic type is now `reference<B>`. Since `B` provides a redefinition of the operation `baz`, the output is different, although we still invoke the operation on the variable `aRef`.

```
program OperationInvocationExample3
import IO

main
   variable f : operation(integer, operation(integer) :  integer) :  integer
   f := operation(parameter x : integer,
                  parameter g : operation(integer) :  integer) : integer
      return g(x)   // invocation of an operation which is passed as a parameter
   endoperation

   IO.writeInteger(  // invocation of an imported library operation
      f(3,           // invocation of an operation stored in a variable
         operation(parameter x : integer) : integer
            return 2 * x
         endoperation)
   )
endmain
```

Listing 83: Third example of operation calls.

Finally, our last example (listing 83) demonstrates invocation of operations defined by lambda expressions, which is usually the case in functional programming. The variable `f` is initialized as a lambda expression which accepts an integer value `x` and another function `g`. The function `g`, which accepts and returns an integer value, is applied to the parameter `x`, the resulting value is returned by the lambda expression, which is now mapped to the variable `f`.

We can now invoke this operation by passing a list of operation parameters to the variable `f`, an integer `3` and an operation which accepts an integer and returns this value multiplied by `2`. This operation is mapped to the parameter `g` during the invocation of the operation mapped to `f`. Basically, we invoke the procedure `writeInteger` first, which then invokes the function `f`, which ultimately invokes the function `g`. The output of the program is: `6`

Validation Checks

In addition to the usual error messages which can be caused by invalid expressions which represent the syntax of an operation invocation and are passed as parameters, following error messages may appear in the context of operation invocation statements and expressions at compile time. As usual, text elements marked as `[...]` are placeholders which are filled with context specific information at runtime.

- Couldn't resolve reference to NamedElement `[IDENTIFIER]`.
- Expected type is: `[DATA TYPE]`. Actual type is: `[DATA TYPE]`.
- Invalid number of arguments was passed. Expected types of arguments are: `[DATA TYPES]`.
- You are not allowed to assign a value to an operation invocation statement.
- The referred element is an operation, either invoke it with a parameter profile or pass it without any access modifiers to a data container with an operation type.
- The super expression is not allowed to be used in this context. This expression can only be used in operations defined in a composition.
- You must not invoke an abstract operation in the context of a super expression.

5.8.4. `if` Statement

An `if`-statement is one of the control flow defining statements in MuLE. This statement defines which part of code is executed depending on one or more conditions.

```
IfStatement:
    'if' Expression 'then' Block
    ('elseif' Expression 'then' Block)*
    ('else' Block)?
    'endif';
```

Listing 84: Grammar rule for `if`-statements.

Following keywords can be used in the `if`-statement:

- `if` - begins the `if`-statement and is followed by the condition. The condition is an expression of boolean type.
- `then` - comes after the condition and is followed by a block of statements, which are executed if the condition is `true`.
- `elseif` - introduces an additional condition, which is checked if the first one was `false`. The additional condition is followed by the keyword `then` with a corresponding block. An arbitrary number of `elseif`-clauses can be included in a single `if`-statement.
- `else` - the following code block is executed if all previous conditions were `false`. The `else` branch is optional.
- `endif` - terminates the `if`-statement. Also removes ambiguity in nested if statements thus eliminating the dangling else problem.

```
if a then
   // if block
elseif b < 5 then
   // elseif block
elseif b = 5 and not a then
   // elseif block
else
   // else block
endif
```

Listing 85: An example of an `if`-statement.

The `if`-clause and each `elseif`-clause are checked whether their respective conditional expression evaluates to a boolean type. If not, the message “Expected type is: boolean. Actual type is: [DATA TYPE].” is displayed at the invalid expressions. Other than that, no other validation checks are performed directly on an `if`-statement, however, each statement contained within an `if`-statement is checked against its own applicable rules.

5.8.5. `loop` and `exit` Statements

A repetition of a block of statements can be achieved by means of recursion or by using the `loop`-statement, which in its basic form represents an infinite loop. Both `return`- and `exit`-statements can be used in order to exit a loop. In the first case the containing operation is terminated, in the second case only the immediate containing loop. An `if`-statement has to be used in order to check the termination condition, if an endless loop is not required. An example is shown in listing 87. Any type of loops can be simulated with these constructs.

```
LoopStatement: 'loop' Block 'endloop';

ExitStatement: 'exit';
```

Listing 86: Grammar rule for `loop`- and `exit`-statements.

```
loop
   IO.writeString("Type the charachter \"N\" to exit the loop: ")
   if (IO.readString() = "N") then
      exit
   endif
endloop
```

Listing 87: An example of a loop terminated by an `exit`-statement.

No validation checks are performed directly on a `loop`-statement, however, each statement contained within a `loop`-statement is checked against its own applicable rules. The `exit`-statement must be used in the context of a `loop`-statement, i.e. it may be contained in several nested `if`-statements, however, at some point one of the `if`-statements must be a part of a block of a `loop`-statement. If this is not the case, following error message will appear: "An exit-statement must be included in the context of a loop-statement, e.g. loop if CONDITION then exit endif STATEMENTS endloop." It should be noted, that while an `exit`-statement must be used within a `loop`-statement, not every `loop`-statement is required to have an `exit`-statement.

5.8.6. `let` Statement

The purpose of the `let`-statement is to check whether a specific value is an instance of a certain type, i.e. if the dynamic type of the value is compatible to the type specified in the head of the `let`-statement. Therefore, it is meant to be used in the context of object-oriented programming when working with values with different dynamic types within the same type hierarchy.

```
LetStatement:
   'let' VariableDeclaration 'be' Expression 'do' Block
   ('elselet' VariableDeclaration 'be' Expression 'do' Block)*
   ('else' Block)?
   'endlet';
```

Listing 88: Grammar rule for `let`-statements.

This statement accepts a value as a parameter (can be any expression) and declares a variable. The type of the expression must be a reflexive or transitive subtype of the type of the declared variable. Both types must be reference types. If the type of the value is compatible to the type of the variable, the value is assigned to the variable and the block of the statement is executed. The declared variable acts as an alias inside of the corresponding block of the statement.

The statement allows to define additional `elselet`-clauses which function the same way, but are checked only if the preceding cases in the whole `let`-statement were not executed. Similar to the `if`-statement, a `let`-statement offers an optional `else` clause, which is executed if all previous clauses were not.

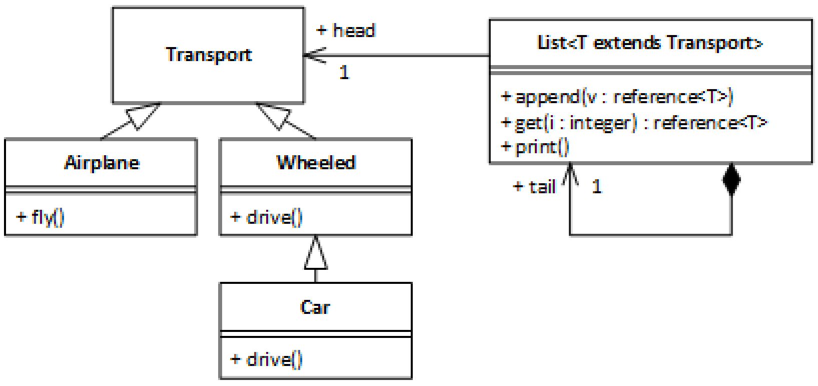

Transport
+ head
1
List<T extends Transport>
+ append(v : reference<T>)
+ get(i : integer) : reference<T>
+ print()
+ tail
1
Airplane
+ fly()
Wheeled
+ drive()
Car
+ drive()

Figure 17: Class diagram for the recursive list example in listing 89.

```
main
   variable lst : List<Transport>
   variable f1 : reference<Wheeled>
   f1 := reference Wheeled{}
   variable f2 : reference<Airplane>
   f2 := reference Airplane{}
   variable f3 : reference<Car>
   f3 := reference Car{}
   lst.append(f1)   lst.append(f2)   lst.append(f3)
   lst.print()   IO.writeLine()

   variable i : integer
   loop
      if lst.get(i) = null then
         exit
      endif
      let variable ff : reference<Airplane> be lst.get(i) do
         ff@.fly()
      elselet variable ff : reference<Wheeled> be lst.get(i) do
         ff@.drive()
      elselet variable ff : reference<Car> be lst.get(i) do
         ff@.drive()
      endlet
      i := i + 1
   endloop
endmain
```

Listing 89: An example demonstrating the use of `let`-statements.

Listing 89 shows an example of a `let`-statement. This is only a short outtake of a larger object-oriented program, whose type structure is summarized in the class diagram in figure 17. In this example we have a list of transports stored as references to actual objects. The used list type is not the built-in mule list type, but a custom one created for this example. In the `let`-statement we check the type of each value in the list. For example, if the value has the data type `reference<Airplance>` the value is passed to the variable `ff`, which is then dereferenced and the operation `fly()` is invoked. The loop iterates over the list and executes the `let`-statement for each value in the list. The `let`-statement checks the dynamic type of the value and executes the operation specific to that type. Since `Car` is a subtype of `Wheeled`, the `elselet`-clause reserved to the type `reference<Car>` can actually never be visited in this example. However, due to the dynamic binding of operations when using reference types, the program will execute the correct operation `drive` depending on the dynamic type of the value referenced by the variable `ff`. The output of the program is:

```
[Wheeled [Airplane [Car]]]
driving
flying
car is driving
```

Validation Checks

All statements inside of a `let`-statement are checked against their respective rules. Following error messages can be triggered directly in the context of a `let`-statement:

- An element with such name already exists. Use a different name for this element.
- The type of this variable must be a reference type.
- The type of this expression must be a reference type.

5.8.7. `return` Statement

A `return`-statement is required to return a value from a function or to terminate a procedure prior to its usual completion. The statement is initiated by the keyword `return` which may be followed by an expression. Each control flow path of operations with return types must end with a return statement with an expression, which is evaluated to a value whose type must be compatible to the return type of the operation. Operations without a return type may have empty return statements, which will terminate the operation.

```
ReturnStatement: 'return' (Expression)?;
```

Listing 90: Grammar rule for the `return`-statement.

Validation Checks

Following error messages can be triggered in the context of a `return`-statement:

- Expected type is: `[DATA TYPE]`. Actual type is: `[DATA TYPE]`.
- An empty return statement is not allowed in an operation with a return type.
- A non-empty return statement is not allowed in an operation without a return type

5.9. Operations and their Parameters

MuLE offers both named and anonymous operations, the former can be invoked or passed to a data container directly by their identifier while the latter are represented by lambda expressions, which must first be passed to a data container before they can be invoked.

Operations may have a return type, in this case at least one `return`-statement (see section 5.8.7) is expected to be included in the operation body. Operations without a return type may include empty return statements which will simply terminate the operation at a specified place. Finally, to be able to handle operations as data, every operation has an operation type (see section 5.5.11), which should not be confused with the return type. An operation type stores information about the operation parameter profile as well as its return type.

As previously mentioned, an operation can be passed to a data container with a compatible type. In this case, the operation is simply referenced by its identifier without specifying the operation parameters. The data container must have an operation type stated in its declaration. The types specified in the parameter profile of the passed operation must be compatible to those defined in the operation type of the data container. Same applies to the return type. Actively passing parameters to an operation referred by its identifier results in an invocation of that operation (see section 5.8.3), i.e. the operation will be executed.

5.9.1. Operation Parameters

An operation parameter is a data container meant to pass data to an operation from outer context. Operations contained within compositions can work with the corresponding attributes, i.e. they can provide meaningful functionality without parameters. Named operations in compilation units as well as anonymous operations (which cannot access data containers from outer context) require value parameters to perform any meaningful computations, unless they are meant to return or print constant values. Each operation may have an arbitrary number of parameters.

```
Parameter: 'parameter' ID ':' DataType;
```

Listing 91: Rule for operation parameters.

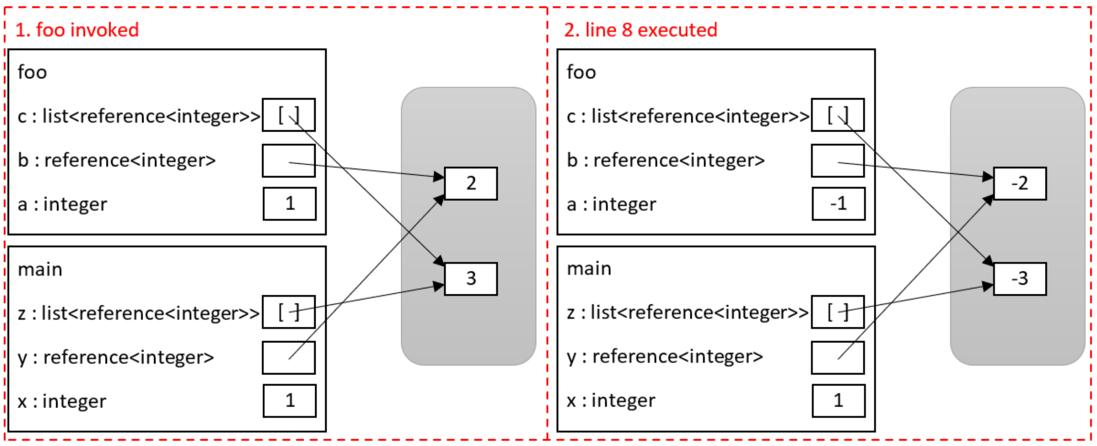


Figure 18: State of the memory during the execution of the program in listing 92.

```
program ParameterValues
import IO

operation foo(parameter a : integer, parameter b : reference<integer>,
              parameter c : list<reference<integer>>)
    a := -1
    b@ := -2
    c[0]@ := -3
endoperation

main
    variable x : integer
    variable y : reference<integer>
    variable z : list<reference<integer>>
    x := 1
    y := reference 2
    z := [reference 3]
    foo(x, y, z)
endmain
```

Listing 92: Value copying semantics to operation parameters in the context of operation invocation.

The value of an operation parameter is assigned when the operation is invoked. MuLE offers only *pass-by-value* semantics, meaning that parameter values are always copied when they are passed to an operation. In this case the passed reference is copied while still pointing to the same data object in memory after being passed to an operation.

Listing 92 and figure 18 show an example of value passing semantics. As we see, the parameters `a`, `b` and `c` are initialized with the values of variables `x`, `y` and `z` respectively. The referenced values are not copied, so even though the list and the reference contained within are copied when passed to the parameter `c`. Manipulating the referenced values in the operation `foo` will cause side effects outside of the operation. In named operations the parameter can be assigned a new value inside of the operation, in anonymous operations this is forbidden.

5.9.2. Named Operations

Named operations can be defined directly in a compilation unit (section 5.4) or in a composition (section 5.5.8). Following subsections will specify rules for using operations in the context of their respective container based on the used programming paradigm as well as provide various examples.

```
Operation:
    ('override')? VisibilityModifier? ('abstract')? 'operation' ID
        '(' (Parameter (',' Parameter)*)? ')' (':' DataType)?
            (Block 'endoperation')?;
```

Listing 93: Rule for named operations.

Named Operations in Procedural Programming

The visibility of library operations can be restricted by the visibility modifier `private`, such operations can only be accessed within the same `library` compilation unit. Listing 94 demonstrates examples of library operations. The operation `pi` does not accept any parameters and returns a constant value. The operation `pow` invokes the operation `powHelper` with its own parameters as well as an additional parameter required for the internal computation. The `private` operation `powHelper` can not be accessed outside of the library. All included operations are functions.

Listing 95 demonstrates an operation which accepts one parameter and invokes the procedure `writeString` from the standard library `IO`. Visible operations imported with the library `MyMath` (listing 94) are invoked in the `main` procedure of the program. Library operations are referred by their qualified names. Operations declared within `program` units can not be assigned a visibility modifier.

The output of the program is:

```
Hello, world!
3.14
8
```

```
library MyMath

operation pi() : rational
    return 3.14
endoperation

operation pow(parameter n : integer, parameter e : integer) : integer
    return powHelper(n, e, 0)
endoperation

private operation powHelper(parameter n : integer,
                            parameter e : integer,
                            parameter counter : integer) : integer
    if counter = e then
        return 1
    else
        return n * powHelper(n, e, counter + 1)
    endif
endoperation
```

Listing 94: Examples of named operations in a library.

```
program OperationsProcedural
import IO
import MyMath

operation sayHello(parameter str : string)
    IO.writeString("Hello, " & str & "!\n")
endoperation

main
    sayHello("world")
    IO.writeRational(MyMath.pi())
    IO.writeLine()
    IO.writeInteger(MyMath.pow(2, 3))
endmain
```

Listing 95: Examples of named operations in a procedural program.

Named Operations in Object-Oriented Programming

The visibility of operations in compositions can be restricted by visibility modifiers `private` and `protected`. Operations declared as `private` can only be accessed within the same composition. Operations with the modifier `protected` are visible within the containing composition and its subtypes. If an operation has the same name and same parameter profile as an existing operation in a super type in the same inheritance hierarchy, the redefining operation must be marked with the keyword `override`. An operation contained in an abstract composition can be marked as `abstract`, meaning that it must not have an operation body and must be overridden in one of the inheriting subtypes. When using reference types, operations are bound dynamically, i.e. it depends on the dynamic type of the referenced object on the heap which operation is executed at runtime.

The example in listing 96 demonstrates operations used in object-oriented context, i.e. declared as member operations of compositions. We have three type declarations:

- An abstract composition `Vector` which only specifies two abstract operations `printData` and `length`, the latter operation is additionally marked as `protected`.
- A composition `Vector2D`, which extends `Vector` and contains protected attributes `x` and `y`. Both inherited operations are overridden, otherwise the program would not compile since `Vector2D` is not abstract.
- A composition `Vector3D`, which extends `Vector2D` and contains an additional protected attribute `z`. Again, both inherited operations are overridden in order to take the additional coordinate into account. Since the inherited abstract operations from `Vector` were already overridden in `Vector2D`, it is not necessary to override them in `Vector3D` to compile the program.

The operation `printData` is the only accessible member outside of this type hierarchy, all other members are `protected`. Both vector variables are declared as references to the abstract `Vector` and typed as either `Vector2D` or `Vector3D`.

The output of the program is:

```
Vector:
    length: 3.605551275463989
    x: 2.0
    y: 3.0
Vector:
    length: 3.7416573867739413
    x: 2.0
    y: 3.0
    z: 1.0
```

```
program OperationsOO
import IO
import Strings

abstract type Vector : composition
   abstract operation printData()
   protected abstract operation length() : rational
endtype

type Vector2D : composition extends Vector
   protected attribute x : rational
   protected attribute y : rational

   override operation printData()
      IO.writeString("Vector:\n")
      IO.writeString("\tlength: " & Strings.rationalToString(length()) & "\n")
      IO.writeString("\tx: " & Strings.rationalToString(x) & "\n")
      IO.writeString("\ty: " & Strings.rationalToString(y) & "\n")
   endoperation

   protected override operation length() : rational
      return (x exp 2 + y exp 2) exp 0.5
   endoperation
endtype

type Vector3D : composition extends Vector2D
   protected attribute z : rational

   override operation printData()
      super.printData()
      IO.writeString("\tz: " & Strings.rationalToString(z) & "\n")
   endoperation

   protected override operation length() : rational
      return (x exp 2 + y exp 2 + z exp 2) exp 0.5
   endoperation
endtype

main
   variable v2d : reference<Vector>
   variable v3d : reference<Vector>
   v2d := reference Vector2D{x = 2, y = 3}
   v3d := reference Vector3D{x = 2, y = 3, z = 1}
   v2d@.printData()
   v3d@.printData()
endmain
```

Listing 96: Examples of named operations in an object-oriented program.

Named Operations in Functional Programming

Semantically, both named and anonymous operations are very similar, both types of operations can and need to be used in the context of functional programming.

```
program OperationsFun
import IO

operation foo(parameter x : integer, parameter y : integer) : integer
   return x + y
endoperation

operation bar(parameter x : integer,
              parameter f : operation(integer, integer) : integer) : integer
   return f(x, x)
endoperation

operation baz() : operation(integer, integer) : integer
   return foo
endoperation

main
   IO.writeInteger(bar(2, foo))
   IO.writeString(", ")
   IO.writeInteger(bar(2, baz()))
endmain
```

Listing 97: Examples of named operations in a program written in a rather functional style.

Listing 97 demonstrates higher-order named operations `bar` and `baz` as well as a first-order operation `foo`. The operation `bar` accepts another operation as a parameter, maps it under the identifier `f`, and finally, invokes the passed function by its new name in order to return the expected integer value. The operation `baz`, on the other hand, simply returns another operation which is again the named operation `foo`.

In the `main` procedure, we invoke the operation `bar` twice, first by passing the named operation `foo` directly and then by invoking the operation `baz` which ultimately returns `foo`. In both cases the operation `foo` is executed in the operation `bar`. The produced output is: `4, 4`.

5.9.3. Anonymous Operations

Anonymous operations are represented by lambda expressions making them the integral language construct of functional programming. They lack an identifier, in order to be invoked they must be referenced by another applicable named element first, e.g. a data container or as a returned value of another operation. It is not allowed to access data containers declared outside of lambda expressions, since they can be passed to another context where these data containers might be missing. For example, if we declare an integer variable and a lambda expression in the context of a main procedure, allow the lambda expression to access the variable and then pass the lambda expression to another operation where this variable does not exist, the lambda expression would try to access a non-existing variable.

```
LambdaExpression:
    'operation' '(' (Parameter (',' Parameter)*)? ')' (':' DataType)?
        Block 'endoperation';
```

Listing 98: Rule for anonymous operations represented by lambda expressions.

Listing 99 contains examples of anonymous operations. All demonstrated lambda expressions are functions, however, procedures are possible as well. For the sake of simplicity, we will refer to these operations by the names of their respective variables. Operations `mult` and `sum` are both first-order functions.

The operation `provider` is a higher-order function which returns other operations depending on the value of the passed parameter. The returned operations are basically the same lambda expressions we have already stored in the variables mult and sum, however, we cannot simply return these variables since it is not allowed to refer to variables declared outside of the lambda expression.

Finally, we have the higher order functions `f1` and `f2`. The former function accepts and integer and another function, which is applied to the integer parameter and the produced value is then returned by `f1`. The latter function accepts an integer value as well as two other functions, which are both applied to the integer parameter and the sum of their results is then returned. The functions `f1` and `f2` are invoked with the functions `mult` and `sum` directly as well as with those returned by the function `provider`.

The output of the program is:

```
f1(3, mult) = 9
f1(3, sum) = 6
f2(sum, mult, 3) = 15
f2(provider("+"), provider("*"), 3) = 15
```

```
program AnonymousOperations
import IO

main
   variable mult : operation(integer) :  integer
   mult := operation(parameter x : integer) : integer
      return x * x
   endoperation

   variable sum : operation(integer) :  integer
   sum := operation(parameter x : integer) : integer
      return x + x
   endoperation

   variable provider : operation(string) :  operation(integer) :  integer
   provider := operation(parameter symbol : string)
                                    : operation(integer) :  integer
      if symbol = "+" then
         return operation(parameter x : integer) : integer
            return x + x
         endoperation
      else
         return operation(parameter x : integer) : integer
            return x * x
         endoperation
      endif
   endoperation

   variable f1 : operation(integer, operation(integer) :  integer) :  integer
   f1 := operation(parameter x : integer,
                parameter g : operation(integer) :  integer) : integer
      return g(x)
   endoperation

   variable f2 : operation(operation(integer) :  integer,
                    operation(integer) :  integer, integer) :  integer
   f2 := operation(parameter g1 : operation(integer) :  integer,
                parameter g2 : operation(integer) :  integer,
                parameter x : integer) : integer
      return g1(x) + g2(x)
   endoperation

   IO.writeString("\nf1(3, mult) = ") IO.writeInteger(f1(3, mult))
   IO.writeString("\nf1(3, sum) = ") IO.writeInteger(f1(3, sum))
   IO.writeString("\nf2(sum, mult, 3)) = ") IO.writeInteger(f2(sum, mult, 3))
   IO.writeString("\nf2(provider(\"+\"), provider(\"*\"), 3) = ")
      IO.writeInteger(f2(provider("+"), provider("*"), 3))
endmain
```

Listing 99: Examples of anonymous operations.

5.9.4. Returning References to Local Values

In section 3.2.1 we have shown an example of the dangling pointer issue in C, the same source code is presented on the left side in figure 19. As a reminder, the function `foo` returns the address of a locally allocated variable which is deallocated as soon as the function is terminated. The attempt to store the result of `foo` in a pointer variable `dp` and to print the stored value in the `main` procedure will fail.

Right side of figure 19 contains a comparable MuLE program. The operation `foo` returns a reference to a value stored in a local variable, however this time the local value is copied and stored in the memory when the expression in the `return`-statement is evaluated. Therefore, the returned reference can be successfully dereferenced in the main procedure to access the stored value, meaning that the issue of the dangling references is not present in MuLE.

```c
#include <stdio.h>


int *foo() {
    int a = 42;
    return &a;
}


int main(void) {
    int *dp = foo();
    printf("%d", *dp);
    return 0;
}
```

```
program ReturningReferences
import IO

operation foo() : reference<integer>
    variable a : integer
    a := 42
    return reference a
endoperation

main
    variable intRef : reference<integer>
    intRef := foo()
    IO.writeInteger(intRef@)
endmain
```

Figure 19: Example of a dangling reference in C and a comparative MuLE program.

5.9.5. Validation Checks in the Context of Operations

Each statement in the body of an operation is checked against its own rules. For error messaged displayed in context of an operation invocation see section 5.8.3. Following error messages can be triggered in the context of an `operation`:

- Use of this name is not allowed. You are attempting to use a reserved word as an identifier.
- An element with such name already exists. Use a different name for this element.
- An operation with a return type must have a return statement.

- An operation can not be abstract and override at the same time.
- Declaration of abstract operations is not allowed in non abstract types.
- Declaration of abstract operations is not allowed in compilation units.
- An abstract operation must not have an operation body.
- A non-abstract operation must have an operation body.
- This operation does not override any inherited operation. The override keyword is not allowed in this case.
- This operation overrides an inherited operation with the same name. Please use the override keyword.
- An overriding operation must have the same visibility as the overridden operation.
- The super type has an operation with the same name but with a different parameter profile. Both operations must have either different names or the same parameter profile.
- The overriding operation must have the same return type as the overridden operation.
- An operation declared in a compilation unit must not be marked with the modifier override.
- Unreachable code.

5.10. Main Procedure

The `main` procedure is a special operation which acts as an entry point into any MuLE program, which is why every program unit must include a `main` procedure and libraries are not allowed to. Unlike operations, the `main` procedure does not accept any parameters, nor does it return a value. Still, it may include empty return statements.

```
MainProcedure: 'main' Block 'endmain';
```

Listing 100: Grammar rule for the `main` procedure.

Listing 101 shows a simple example of a `main` procedure, which contains a variable declaration and a `loop`-statement. The procedure is terminated when the variable reaches the value 5. The output of the program is: `1, 2, 3, 4, 5.`

```
program MainProcedureExample
import IO

main
   variable x : integer
   loop
      x := x + 1
      IO.writeInteger(x)
      if x = 5 then
         IO.writeString(".")
         return
      endif
      IO.writeString(", ")
   endloop
endmain
```

Listing 101: Examples of a `main` procedure.

Each statement inside of a `main` procedure is checked against its own rules. The absence of a `main` procedure in a `program` unit or its presence in a `library` unit are reported by the respective errors in the context of the compilation unit. Statements that can not be executed due to early termination of the `main` procedure with a return statement are marked as “Unreachable code”.

5.11. Conclusion

In this chapter we have covered the detailed specification of our language based on the design decisions presented in the previous chapter. We have covered the scoping rules, explained the type system as well as all supported language constructs. For each construct we have shown the grammar rules, the validation rules which are defined by the type system, visibility rules as well as further constraints, and finally, programming examples which demonstrate how these constructs can be used. Next chapter will cover the standard libraries distributed with MuLE.

6. Libraries

The range of functionality provided by the grammar of the language, i.e. by its built-in types, operators and keywords, is rather limited by design. The grammar offers a minimal set of commonly used types and operators. More specific operations are reserved to various standard libraries, each containing thematically consistent type declarations and operations. The most basic example is the `IO` library which provides input and output operations allowing to print and read values to or from the console or files. Other libraries may include mathematical operations not covered by the operators, operations reserved to specific types or educational libraries. Another convenient aspect of libraries, in addition to reducing the number of required operators and keywords, is that it is far easier to edit existing libraries or add new ones compared to changing the grammar of the language.

MuLE comes with a set of standard libraries which offer additional functionality and must be explicitly imported just as any user defined library. The currently included libraries are the aforementioned `IO`, which provides basic input/output functionality, `Mathematics` with additional mathematical functions, `Strings` and `Lists`, both of them containing operations reserved for these specific types, the educational libraries `Turtle` and `UBTMicroworld` which provide an environment to learn programming while drawing shapes or navigating an agent through a maze, and finally `GUIFactory`, a simple library which allows to implement graphical user interfaces. Following sections offer an overview over the included standard libraries as well as examples of their uses, the APIs of these libraries are specified in the appendix of this thesis.

6.1. IO

The standard library for input/output operations. The specification of the API is provided in appendix C. It includes operations which print or read values of basic types on or from the console, create a line break as well as create files and read from them. Students are immediately confronted with this library when implementing their first *"Hello, world!"* program. Due to frequent use, the instruction `import IO` is included by default in newly created MuLE files.

Operations tasked with reading values of basic types from the console follow the following rules: the initial white spaces are ignored, i.e. if the user enters a line break first, the system will still wait for user input, however the next white space character after a character sequence will terminate the sequence evaluation, i.e. the sequences `Hello world!`, `true false`, or `42 15` will be evaluated as `Hello`, `true` and `42` respectively. If the evaluated character sequence cannot be parsed as a legitimate value of the expected return type, a default value is returned instead, for example if `readInteger()` was called and the user entered `hello`, the result will be `0`.

```
program InputOutputExamples
import IO
main
   IO.writeString("Enter a string: ")
   IO.writeString(IO.readString()) IO.writeLine()
   IO.writeString("Enter a boolean: ")
   IO.writeBoolean(IO.readBoolean()) IO.writeLine()
   IO.writeString("Enter an integer: ")
   IO.writeInteger(IO.readInteger()) IO.writeLine()
   IO.writeString("Enter a rational: ")
   IO.writeRational(IO.readRational()) IO.writeLine()
endmain
```

Listing 102: Examples of usage of `IO` library operations to read and write values of basic types.

Our first example of `IO` operations is demonstrated in listing 102. The contents of the console with both user input (green) and program output (black) are as follows:

```
Enter a string: Hello, world!
Hello,
Enter a boolean: true
true
Enter an integer: hello
0
Enter a rational: 42
42.0
```

The entered string is `Hello, world!`, however only `Hello,` is printed since the entered character sequence is evaluated up until the next white space. Furthermore, we have entered a `hello` when we were prompted to enter an integer value and since this sequence cannot be parsed as an integer, the printed result is `0`. Finally, we have entered the integer 42 when the program asked us to enter a rational value, which was promptly converted into a 42.0 and printed on the console.

Our next example in listing 103 shows how this library can be used to read and write contents of files. A file can be identified either by its absolute or by its relative path. The absolute path contains all the details needed to locate a file starting with the drive name. If one of the folders on the path does not exist, the file cannot be located. The relative path is, as the name suggests, relative to a specific location, in our case the project folder where our current program file is located. If we simply enter the file name `test.txt` as the path, the program will look for it in the root project folder, however if the entered path is `files\\test.txt`, the program will attempt to locate the folder `files` in the root folder of the project first and then, assuming that the folder does exist, look for the file there. If we want to read from a file and it does not exist, we either get a string `"file not found"` or a list `["file not found"]`, depending on which operation we have executed. If we want to write to a file and it does not exist, it will be created as long as all the folders on the path do exist. If the file already exists, its contents will be overwritten, therefore the user may want to check if this is the case by invoking the operation `fileExists` first.

```
1   program InputOutputExamples2
2   import IO
3   import Strings
4   main
5      variable path : string
6      path := "test.txt"
7      IO.writeString(IO.readFile(path)) IO.writeLine()
8      if IO.fileExists(path) then
9         IO.writeFile(path, IO.readFile(path) & "\n" & "world")
10     else
11        IO.writeFile(path, "hello")
12     endif
13     IO.writeString(Strings.genericToString(IO.readFileLines(path)))
14  endmain
```

Listing 103: Examples of usage of `IO` library operations to read and write contents of files.

We are using relative paths in the example in listing 103, meaning that the files will be located in the root folder of the project. Assuming that we execute the program for the first time after we have created the project, the required files do not yet exist, which means that the attempt to read from the file and print its contents in line 7 will fail, hence the printed `file not found` in the initial output of the program:

```
file not found
File path: C:\Users\UserName\WorkspaceFolder\ProjectFolder\test.txt
[hello]
```

The file is created when we invoke the operation `writeFile` and its path is printed on the console to help users locate the file more easily. We check explicitly if the file exists in the `if`-statement. If this is not the case yet, we invoke `writeFile` with the content `"hello"`, otherwise we add `"world"` to the existing content. Finally, we read the file with the operation `readFileLines` which returns us the list of lines in the file, which we then print on the console. For each subsequent execution of the program, an additional line containing `"world"` is added to the file. The output of the program after three runs including the initial execution is:

```
hello
world
File path: C:\Users\UserName\WorkspaceFolder\ProjectFolder\test.txt
[hello, world, world]
```

6.2. Mathematics

This library includes a range of mathematical functions that are not already covered by operators. Additionally, it contains functions which provide pseudorandom numbers, approximations of mathematical constants as well as the minimal and maximal supported values of numerical types. The specification of the API is provided in appendix D.

It should be noted, that due to the limitations of the representation of floating point numbers [103], the resulting values may differ slightly from the ideal mathematical value. This behaviour is demonstrated in the example in listing 104, where we first convert the angle of 45° to radians, apply the *sine* function, then apply the `arc sine` function and convert the result back to degrees. The produced result should be 45 again, the output of the program is however `44.99999999999999` which is a result of rounding errors and approximations in the floating point arithmetic.

```
program MathematicsExample
import IO
import Mathematics

main
    IO.writeRational(
        Mathematics.toDegrees(
            Mathematics.asin(
                Mathematics.sin(
                    Mathematics.toRadians(45)))))
endmain
```

Listing 104: Examples of `Mathematics` trigonometric library operations.

Listing 105 demonstrates the use of operations which generate pseudorandom integer and rational numbers. We have two lists which are initially filled with four zeroes each and are meant to be filled with the number of the generated numbers in range [0 .. 3] in case of integers and in steps of 0.25 in case of rational numbers. In the latter case, the list is filled with the number of occurrences of the generated numbers. The loop is executed 4000000 times, meaning that the expected ideal result in both cases would be `[1000000, 1000000, 1000000, 1000000]`. The actual output in one of the runs is:

```
lst1: [999963, 1000886, 1000559, 998592]
lst2: [1000469, 999406, 999542, 1000583]
```

The actual result is close to the expected result, which is good enough for the tasks in which students have to rely on random number generators in the introductory courses.

```
program MathematicsExample2
import IO
import Mathematics
import Strings
main
   variable count : integer
   variable lst1 : list<integer>
   lst1 := [4 ** 0]
   variable lst2 : list<integer>
   lst2 := [4 ** 0]
   loop
      variable rndInt : integer
      rndInt := Mathematics.randomInteger(0, 3)
      lst1[rndInt] := lst1[rndInt] + 1
      variable rndRat : rational
      rndRat := Mathematics.randomRational()
      if rndRat < 0.25 then
         lst2[0] := lst2[0] + 1
      elseif rndRat < 0.50 then
         lst2[1] := lst2[1] + 1
      elseif rndRat < 0.75 then
         lst2[2] := lst2[2] + 1
      else
         lst2[3] := lst2[3] + 1
      endif
      count := count + 1
      if count >= 4000000 then
         exit
      endif
   endloop
   IO.writeString(Strings.genericToString(lst1) & "\n")
   IO.writeString(Strings.genericToString(lst2) & "\n")
endmain
```

Listing 105: Examples of `Mathematics` random number generator library operations.

6.3. Strings

This library includes a range of functions meant to work with string values, e.g. provide a substring, the number of characters in a string, replace the first or all occurrences of a substring with another, split a string into a list of strings at specific substrings, convert alphabetical characters to upper or lower case, as well as convert values of other types to string values. The first character in a string value has the index 0. The specification of the API is provided in appendix E.

The first example in listing 106 demonstrates how values of other types can be converted to string values. This example is actually a shortened version of an early assignment, which has to be solved by the students in our preliminary programming course where MuLE is used as the programming language (more in chapter 9). The students have to implement a simple user interaction scenario, where the user is prompted to enter his or her age as an integer. They also have to compute the year of birth depending on the current year and the age again as an integer and use string concatenation to print the answer in a single instruction. Since string concatenation requires string values on both sides, students have to convert the year of birth to a string value first. The contents of the console with both user input (green) and program output (black) are as follows:

```
Please enter your age: 31
So, you were born in 1990
```

```
program UserInteraction
import IO
import Strings

main
   IO.writeString("Please enter your age: ")
   IO.writeString("So, you were born in "
                  & Strings.integerToString(2021 - IO.readInteger()) & ".")
endmain
```

Listing 106: Converting integer values to strings using the `Strings` library.

Similarly to our first example, the second example in listing 107 is one of the assignments in our preliminary programming course, albeit one of the last ones with recursion being the main topic. Students have to implement a recursive algorithm, which decides whether a string is a palindrome, i.e. it reads the same forward or backward, or not. The approach is to compare characters on the left and right edge of the string, if they are not equal, the string is not a palindrome. If they are, the remainder of the string must be checked if it is a palindrome. If the string consists of only one or zero characters, it is definitely a palindrome, which also serves as the termination condition. Students have to use the `Strings` operations `subString` to get the characters at the edges as well as the substring without those characters, and the `legthOf` operation in order to compute the position of the last character.

The output of the program is:

```
true
true
false
false
```

```
program Palindromes
import Strings
import IO

operation isPalindrome(parameter str : string) : boolean
   if Strings.lengthOf(str) <= 1 then
      return true
   else
      variable s1 : string
      variable s2 : string
      s1 := Strings.subString(str, 0, 0)
      s2 := Strings.subString(str,
               Strings.lengthOf(str) - 1, Strings.lengthOf(str) - 1)
      if s1 /= s2 then
         return false
      else
         variable newStr : string
         newStr := Strings.subString(str, 1, Strings.lengthOf(str) - 2)
         return isPalindrome(newStr)
      endif
   endif
endoperation

main
   IO.writeBoolean(isPalindrome("abba")) IO.writeLine()
   IO.writeBoolean(isPalindrome("abcba")) IO.writeLine()
   IO.writeBoolean(isPalindrome("abc")) IO.writeLine()
   IO.writeBoolean(isPalindrome("abca")) IO.writeLine()
endmain
```

Listing 107: Examples of `Strings` library operations `lengthOf` and `subString`.

6.4. Lists

This library includes functions designed to work with the built-in list type, such as checking if the list is empty, adding or removing new entries, returning specific entries or sublists, filtering, iterating, checking whether an entry is contained in the list or returning the index of an entry (if available). The index of the first entry in the list is 0. The specification of the API is provided in appendix F.

When adding or removing entries, the original list, which is passed as a parameter to the corresponding functions, is not changed and an altered copy of the original list is returned instead. This behaviour is demonstrated in listing 108, where we declare a variable `lst` as a list of integers and initialize it as `[1, 2, 3]`. When we simply invoke the **append** operation, which adds an element at the end of the list, the printed result is `[1, 2, 3, 4]`. However, when we print the value of `lst`, we notice that it has not changed. Its state is changed only after we repeat the invocation of `append` and explicitly assign the resulting list to `lst`. The output of the program is:

```
[1, 2, 3, 4]
[1, 2, 3]
[1, 2, 3, 4]
```

```
program ListsExample1
import IO
import Strings
import Lists

main
    variable lst : list<integer>
    lst := [1, 2, 3]
    IO.writeString(Strings.genericToString(Lists.append(lst, 4)))
    IO.writeLine()
    IO.writeString(Strings.genericToString(lst))
    lst := Lists.append(lst, 4)
    IO.writeLine()
    IO.writeString(Strings.genericToString(lst))
endmain
```

Listing 108: Examples of `Lists` library operations `forEach` and `filter`.

```
program ListsExample2
import IO
import Lists

main
    Lists.forEach(
        Lists.filter(
            [1, 2, 3, 4],
            operation(parameter entry : integer) : boolean
                if entry mod 2 = 0 then
                    return true
                else
                    return false
                endif
            endoperation
        ),
        operation(parameter entry : integer)
            IO.writeInteger(entry) IO.writeString(" ")
        endoperation
    )
endmain
```

Listing 109: Examples of `Lists` library operations `forEach` and `filter`.

An example of how we can use the operations `forEach` and `filter` are demonstrated in listing 109. In this example, we `filter` the list `[1, 2, 3, 4]` first by applying an operation which returns `true` for each even integer in the list, meaning that the resulting filtered list is `[2, 4]`. Then we use the `forEach` procedure to apply another procedure on the filtered list, which prints each entry of the list followed by a white space. The output of the program is: `2 4` .

6.5. Turtle

MuLE includes an implementation[2] of the Turtle library, which has been successfully used in programming education over decades [75]. The library is designed to be used in procedural way without relying on object-oriented programming. As soon as at least one invocation of a Turtle operation is included in the source code, the Turtle window will open when the program is executed. The upper left corner of the canvas has the coordinates *(0, 0)*, the standard canvas size is 600x400 pixels, the pen is initially placed at *(300, 200)* pixels. Users can resize the canvas and turn the coordinates grid on and off. The specification of the API is provided in appendix G.

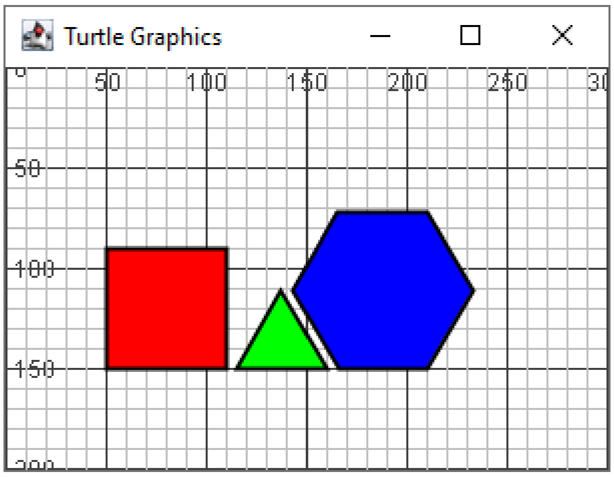


Figure 20: Result of the program in listing 110.

Listing 110 shows an example program using the Turtle library to draw filled polygons displayed in figure 20. This program is another assignment students have to solve in our preliminary programming course (see chapter 9.1). The operation `draw` contains the logic of drawing a polygon with the given number of edges (`n`), their length (`l`) and the fill colour. The operation `move` moves the cursor to the next position where the next shape has to be drawn, the move

[2]The Java implementation of this library was provided by Stefan Schill as a part of his Master of Science project [104] at the University of Bayreuth and subsequently included in MuLE.

direction depends on the orientation of the pen in this example. First, we set up our Turtle window by turning the coordinate grid on, setting the size of the canvas, the thickness of the drawn lines, etc. Then, we draw our three shapes with the help of the operation `draw` and move the cursor to the next position with the operation `move`. Finally, we hide the cursor after we have finished drawing.

```
program TurtleExample
import Turtle

operation draw(parameter n : integer,
          parameter l : integer, parameter color : Turtle.Colors)
   Turtle.startFilledPolygon(color)
   variable count : integer
   loop
      if count >= n then
         exit
      endif
      Turtle.forward(l)
      Turtle.left(360.0 / n)
      count := count + 1
   endloop
   Turtle.endFilledPolygon()
endoperation

operation move(parameter a : integer)
   Turtle.penUp()
   Turtle.forward(a)
   Turtle.penDown()
endoperation

main
   Turtle.showCoordinateSystem(true)
   Turtle.setFrameSize(300, 200)
   Turtle.setPosition(50, 150)
   Turtle.setOrientation(Turtle.Orientation.EAST)
   Turtle.setThickness(2)
   draw(4, 60, Turtle.Colors.RED)
   move(65)
   draw(3, 45, Turtle.Colors.GREEN)
   move(50)
   draw(6, 45, Turtle.Colors.BLUE)
   Turtle.showCursor(false)
endmain
```

Listing 110: Example program using the Turtle library to draw simple polygons, the result is shown in figure 20.

When drawing polygons, a line is drawn between the current position of the pen and the initial position when this operation was invoked, filling the space between the edges of the polygon with the given colour, figure 21 shows a square polygon in the process of drawing. The operation `endFilledPolygon` must be invoked at some point to correctly terminate the process of drawing a polygon.

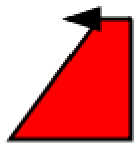


Figure 21: An unfinished coloured square during the drawing process. The initial position of the cursor was in the bottom-left corner of the square.

6.6. UBTMicroworld

MuLE includes a programming microworlds library[3] inspired by educational programming environments such as Scratch [67] and Karel [106]. These environments, their purpose as well as their advantages and shortcomings were discussed in section 3.2.2, thus the focus of this section will be purely on the library `UBTMicroworlds` and its functionality. The specification of the API is provided in appendix H.

The idea of this library is to offer a two dimensional environment with one or more agents which can be navigated in this environment either by instructions written in source code or by inputs from the keyboard. The environment is basically a board with tiles that can be entered and those that cannot be entered, starting tiles where agents are placed, target tiles which represent the goal of the agents in most of the scenarios, as well as optional objects, that can be placed on the board and must be gathered by the agents in order to achieve their goal. An agent moving from one tile to a neighbouring tile or turning left or right or actively doing nothing (by invoking the `doNothing` operation) counts as one step.

The library offers several predefined levels with predefined win conditions such as reach the target tile or gather all objects and then reach the target tile. Furthermore, users may define their own levels with their own win conditions making room for some creativity. This library can be used to playfully introduce beginner programmers to such topics like control flow, but also for the more advanced exercises such as implementation of navigational algorithms in a two dimensional maze. Several agents can be placed, with some of them following the programmed algorithm while one agent can be registered to key listeners.

Users initialize the level and implement the movements of the agents in the source code of the MuLE program. When they execute the program, a graphical

[3]The Java implementation of this library was provided by Marco Jantos as a part of his Bachelor of Science thesis [105] at the University of Bayreuth and subsequently included in MuLE.

user interface (see figure 23) displays the environment and shows the movements of the agents as defined in the program. After the level is finished, the replay mode is activated, wherein the user can retrace the movements for each agent.

Level Details

Each level is represented by the terrain map and optionally a map of objects, if objects are placed in the level. Win conditions define when a level is completed. There are three predefined win conditions which require all agents to reach a target tile, gather all objects or both, as well as a custom win condition, which requires the user to define the logic for successful of failed level completion on their own.

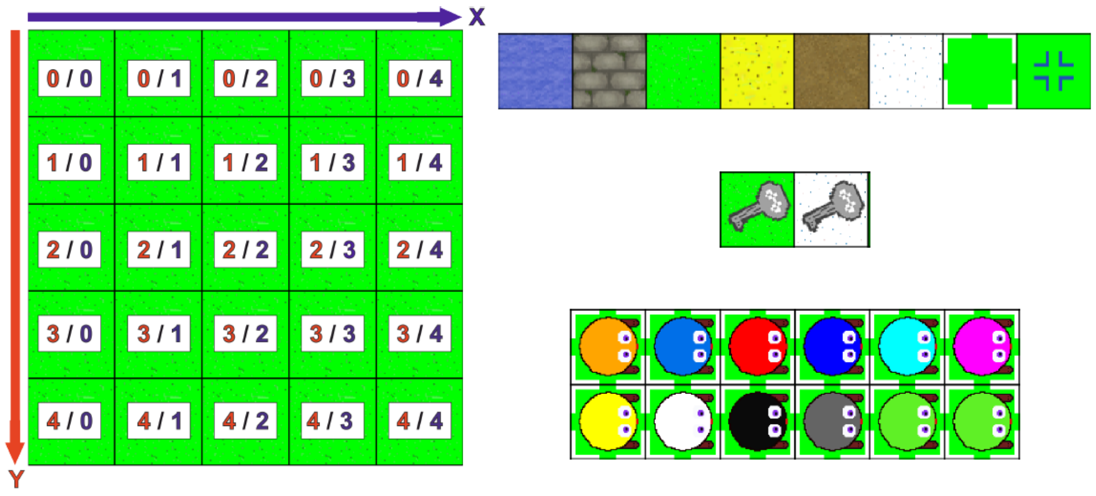


Figure 22: Tile coordinates, tile colouration based on terrain type, object placement and agent colours *(images taken from the B.Sc. thesis [105] of Marco Jantos).*

Figure 22 displays information regarding the tile coordinates, terrain types, object placement and agent colouring. The tile in the upper left corner has the coordinates *(0,0)*. The board size can range from 5x5 tiles up to 32x32 tiles. Each tile has a specific terrain type, each terrain type is additionally coded by an RGB value which can used to define custom levels by PNG images, wherein each pixel in the image is representative of one tile, meaning that a board with 32x32 tiles is represented by an image file with 32x32 pixels. There are two possibilities to define custom levels, either programmatically which can become tedious when specifying larger maps, or by loading the aforementioned PNG images with terrain types coded as RGB values, wherein each terrain type is represented by the corresponding colour. For example, WATER is represented by the colour blue with the RGB values (0, 0, 255), i.e. all pixels in the PNG file with this colour will mean that the tiles with the same coordinates in the map are displayed as WATER. Examples of initialising custom levels using both approaches are given in the next subsection. The terrain types are as follows:

- WATER – inaccessible, RGB code is (0, 0, 255).
- STONE – inaccessible, RGB code is (255, 0, 255).
- GRASS – accessible, RGB code is (0, 255, 0).
- SAND – accessible, RGB code is (255, 255, 0).
- SNOW – accessible, RGB code is (0, 255, 255).
- PATH – accessible, RGB code is (255, 0, 0).
- TARGET – accessible, RGB code is (0, 0, 0), required by some win validators.
- START – accessible, RGB code is (255, 255, 255), an agent is placed at each starting tile, i.e. the number of placed agents is equal to the number of starting tiles.

Objects can be placed on tiles and automatically collected by the agents if they visit the tile. Each agent tracks the number of collected objects. Some default win validators require that all placed objects have to be collected. When initializing a custom level via PNG images, an additional image that represents the map of the objects can be passed. The object map must have same dimensions as the terrain map, the objects are coded by the RGB value (0, 0, 0), any other colour will signalize a lack of an object at the corresponding tile.

As already mentioned, an agent is placed for each starting tile facing east. The first eleven agents have a unique colour as shown in figure 22, subsequent agents have the colour of the eleventh one.

Examples

Figure 23 shows a screenshot of the graphical user interface after the program in listing 111 was executed.

As we can see in the source code, the library is simply imported as any other library. The user then needs to initialize a game level, as we see in line 5. Executing this line alone is enough to show the interface displayed in the screenshot. The intended procedure is to initialize a level first, execute the program to display it, design and implement a solution and run the program again to test it.

In our example we have one agent placed initially in the upper left corner in the tile with the coordinates *(1, 1)* as defined by the default level. We get a reference to this agent from the library and implement a movement algorithm wherein our agent makes a forward move, a right turn, moves forward again followed by a left

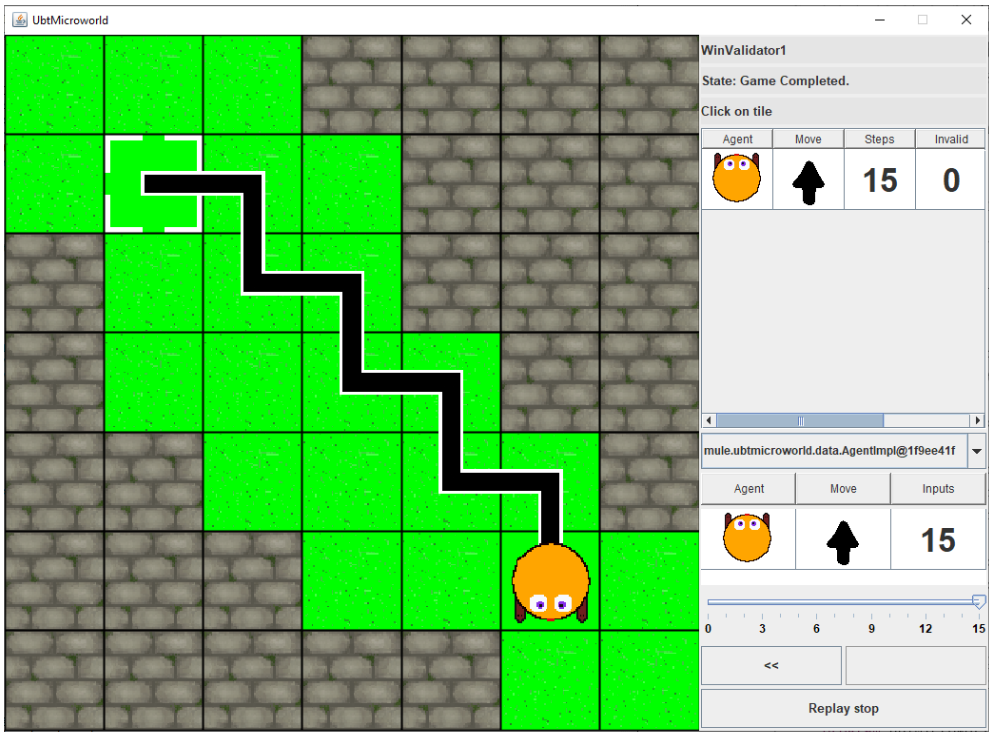


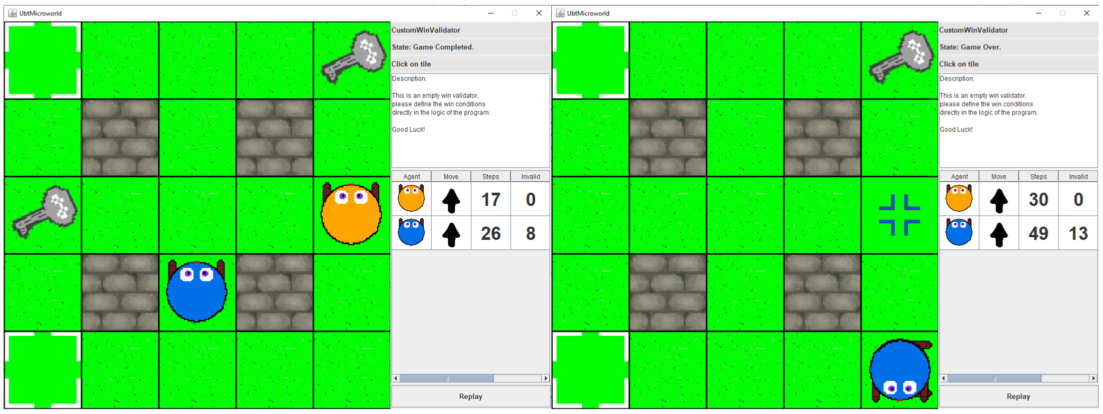
Figure 23: Example screenshot of the `UBTMicroworld` environment after executing the code in listing 111.

turn, and then repeats these movements until the target tile is reached. What we see in the screenshot in figure 23 is the replay mode, where we can trace each movement made by the agent after the level is finished, which helps to debug the implemented algorithm if necessary.

```
program UBTMicroworldExample1
import UBTMicroworld

main
    UBTMicroworld.initDefaultGame(UBTMicroworld.DefaultLevelType.DEFAULT_LEVEL_2)
    UBTMicroworld.setDelayTime(UBTMicroworld.DelayTime.SHORT_DELAY)
    variable agent : reference<UBTMicroworld.Agent>
    agent := UBTMicroworld.getAgentList()[0]
    loop
        agent@.moveForward()
        agent@.rotateRight()
        agent@.moveForward()
        agent@.rotateLeft()
        if UBTMicroworld.isGameRunning() = false then exit endif
    endloop
endmain
```

Listing 111: An example of a predefined level with 23.

Our next example in listing 112 and figure 24 demonstrates how custom levels can be defined by lists of terrain types with object placements and custom win conditions. In this scenario we implement a small game where a player agent controlled by arrow keys has to gather at least three keys and reach the target goal while not getting caught by the enemy agent controlled by the computer.



CustomWinValidator

State: Game Completed.

Click on tile

Description:

This is an empty win validator,
please define the win conditions
directly in the logic of the program.

Good Luck!

Agent	Move	Steps	Invalid
🟠	↑	17	0
🔵	↑	26	8

CustomWinValidator

State: Game Over.

Click on tile

Description:

This is an empty win validator,
please define the win conditions
directly in the logic of the program.

Good Luck!

Agent	Move	Steps	Invalid
🟠	↑	30	0
🔵	↑	49	13

Figure 24: Two states of the game after finishing the program in listing 112, a successful completion on the left and the game over state on the right.

```
program UBTMicroworldExampleCustom1
import UBTMicroworld
import Mathematics

main
    variable g : UBTMicroworld.TerrainType g := UBTMicroworld.TerrainType.GRASS
    variable s : UBTMicroworld.TerrainType s := UBTMicroworld.TerrainType.STONE
    variable O : UBTMicroworld.TerrainType O := UBTMicroworld.TerrainType.START
    variable T : UBTMicroworld.TerrainType T := UBTMicroworld.TerrainType.TARGET
    variable X : UBTMicroworld.ObjectType X := UBTMicroworld.ObjectType.KEY
    variable _ : UBTMicroworld.ObjectType _ := UBTMicroworld.ObjectType.NO_OBJECT
    UBTMicroworld.initCustomGame2(
            [   [O, g, g, g, g],
                [g, s, g, s, g],
                [g, g, g, g, T],
                [g, s, g, s, g],
                [O, g, g, g, g]    ],
            [   [_, _, X, _, X],
                [_, _, _, _, _],
                [X, _, X, _, _],
                [_, _, _, _, _],
                [_, _, X, _, X]    ],
            UBTMicroworld.WinValidatorType.CUSTOM
    )
    UBTMicroworld.setDelayTime(UBTMicroworld.DelayTime.NO_DELAY)
```

```
        variable player : reference<UBTMicroworld.Agent>
        player := UBTMicroworld.getAgentList()[0]
        variable enemy : reference<UBTMicroworld.Agent>
        enemy := UBTMicroworld.getAgentList()[1]
        UBTMicroworld.registerAgentForKeyListener(player)
        variable playersMoves : integer
        variable playersTurn : boolean
        loop
            if UBTMicroworld.isGameRunning() then
                if not playersTurn then
                    if Mathematics.randomInteger(1, 2) = 1 then
                        enemy@.rotateLeft()
                    else
                        enemy@.rotateRight()
                    endif
                    enemy@.moveForward()
                    playersTurn := true
                else
                    if player@.getNumberInputs() > playersMoves then
                        playersMoves := playersMoves + 1
                        playersTurn := false
                    endif
                endif
                if enemy@.getXPosition() = player@.getXPosition() and
                        enemy@.getYPosition() = player@.getYPosition() then
                    UBTMicroworld.setGameOver()
                elseif player@.getNumberOfCollectedObjects() >= 3 and
                        player@.getXPosition() = 4 and player@.getYPosition() = 2 then
                    UBTMicroworld.setGameFinished()
                endif
            else
                exit
            endif
        endloop
    endmain
```

Listing 112: Example showing custom levels, win conditions and agents contolled by different means.

First we define a set of variables for different terrain types and object placement for the sake of more clarity, which we promptly use to initialize our custom game with two lists representing the terrain map and the object placement map as well as a custom win validator, which basically means that we have to define our own win conditions in the code. We continue by removing the delay time between actions, since our program will be waiting on user input anyway.

We then get our references to both the player and the enemy agents and implement a loop which runs until the game is finished. If it is not the player's turn, we move our enemy agent by performing a random turn and a move forward. If it is the player's turn, the program waits for the user input by comparing the number of inputs of the `player` agent and the corresponding value of the variable `playersMoves`. If the player has not yet pressed a key in his turn, both values are equal, otherwise the value of `playersMoves` is increased and `playersTurn` is set to `false`. In any case, we check if both agents are placed in the same tile which sets the state to game over, and if this is not the case, we check whether the player agent has completed the goal which sets the state to game completed.

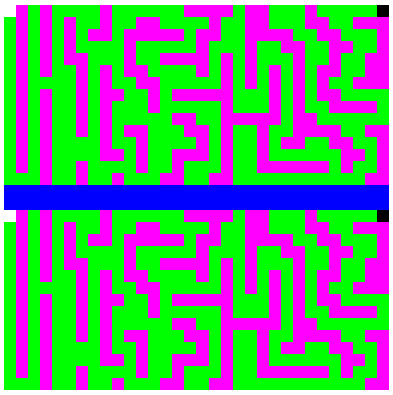
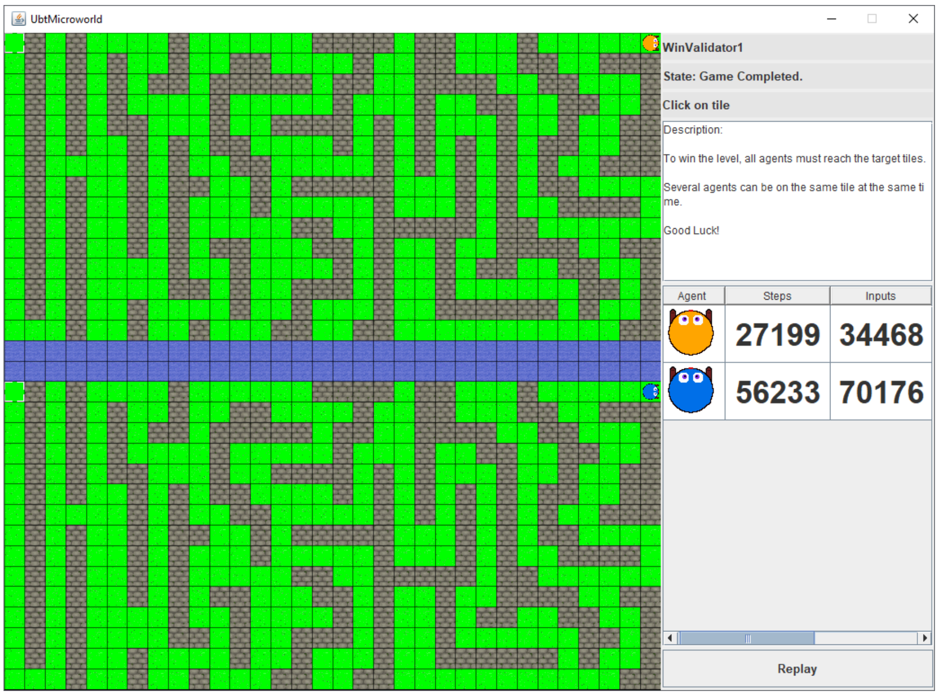


Figure 25: An enlarged terrain map used to initialize the custom level in the example in figure 26, the actual size of the image is 32x32 pixels.



Description:

To win the level, all agents must reach the target tiles.

Several agents can be on the same tile at the same ti me.

Good Luck!

Agent	Steps	Inputs
	27199	34468
	56233	70176

Figure 26: Screenshot of the level defined by the terrain map in figure 26 after it has been completed.

Defining a larger level which may contain 32x32 tiles can become quite tedious, therefore, there is an alternative way which allows to define a custom level by reading the terrain types and object placement from image files. Each pixel in the image file represents one tile, meaning that a level with 32x32 tiles is defined by an image with 32x32 pixels. The colour coding of the different terrain types was explained in the previous subsection. Figure 25 shows an enlarged version of the terrain map used to define the level in figure 26. The instruction used to initialize this custom level is:

```
UBTMicroworld.initCustomGame3("img/maps/map2.png",
                              UBTMicroworld.WinValidatorType.VALIDATOR_1)
```

In this particular example we have created two identical mazes, the goal of the level is that both agents are located at their target tiles. The agents move in the same random pattern as in the previous example until they reach a target tile. As we see in the screenshot, the blue agent has required approximately twice as many actions to reach the goal. Such exercises can be used to test implementations of different students or different navigational algorithms against each other.

6.7. GUIFactory

Graphical user interfaces usually rely on a wide range of different components and are implemented using advanced techniques and design patterns which are unknown to beginner programmers. In the context of programming education, professionally used GUI libraries like Swing [107] and JavaFX [108] suffer from the same issues as professionally used languages and IDEs, the amount of contents provided by these toolkits is overwhelming for programming beginners. Therefore, the implementation of `GUIFactory` is consistent with the general design principles of MuLE. This library[4] is meant to be used as an educational tool to teach basics of graphical user interfaces (GUI). Thus, it provides a minimal set of components which are sufficient for the intended task in the context of programming education. The specification of the API is provided in appendix I.

The relation between the elements of this library is quite simple as shown in figure 27, a `Window` includes a `Pane` (with a specific layout), which includes a list of `Components`. Several `Panes` can be nested, since a `Pane` is also a `Component`.

The entire library is split into five compilation units to keep them manageable. Most likely, users will have to import most or all of the library units in order to implement a working graphical user interface. As the name implies, this library was heavily inspired by the factory pattern. Therefore, the usual build up of a library unit consists of enumerations and abstract compositions, which are used

[4]The Java implementation of this library was provided by Johannes Glier as a part of his Bachelor of Science thesis [109] at the University of Bayreuth and subsequently included in MuLE.

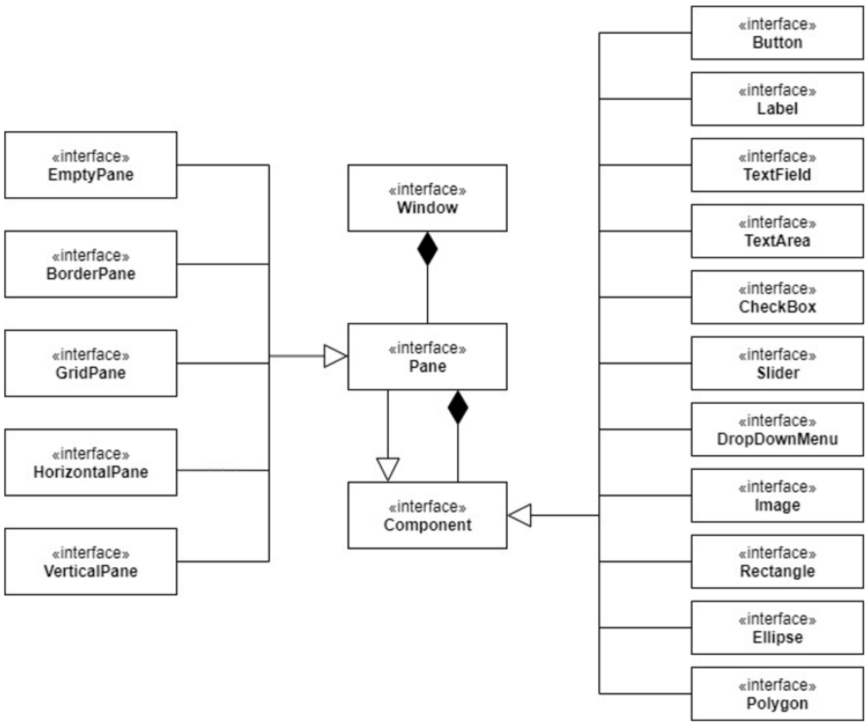


Figure 27: Overview over core elements of the `GUIFactory` library *(images taken from the B.Sc. thesis [109] of Johannes Glier.)*

to declare variables of these types and creation operations used to initialize these variables.

`GUIFactory` Unit

This is the main compilation unit, in addition several enumeration types, compositions and operations related with colours, font types and alignments, this unit contains the composition `Window`, which represents the root element of a GUI. As previously mentioned, the library is inspired by the factory pattern and since instances of `Window` must be created one way or another, this unit also provides the corresponding creation operation `createWindow`.

`GUIFactoryPanes` Unit

A pane is a special component of a GUI with a specific layout. It serves as a container for GUI components and defines how these components are arranged. Since a pane is a `Component` itself, it can contain other panes allowing users to create complex layouts with a small amount of predefined layouts defined by panes.

Users may set the border as well as add components via operations defined by these types. If the user does not specify the size of the components, they

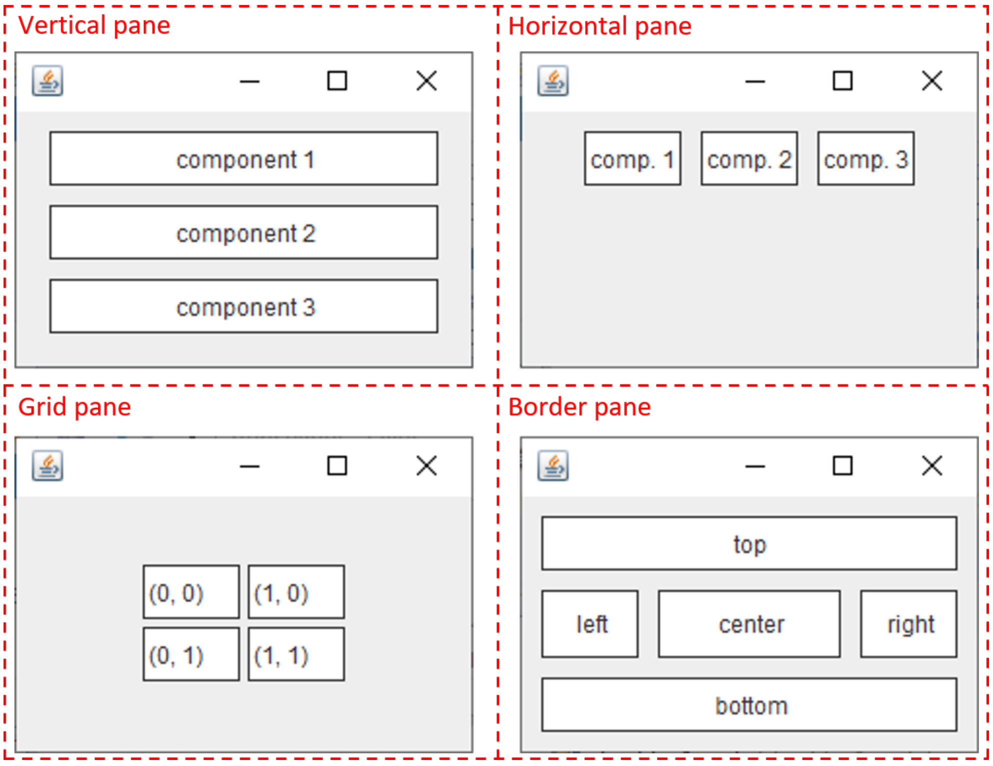


Figure 28: Examples of pane types and the corresponding layouts specified in `GUIFactoryPanes`.

will automatically take all available shared space. Components, whose specified size is bigger than the available space are scaled down. Users may specify the padding, i.e. the space between the borders of a pane and its contents, and the spacing between child components. There are four pane types specified in this library unit, as well as the corresponding creation operations (examples are given in figure 28):

- `VerticalPane` – components are arranged vertically when added as children to this pane. Children components are displayed in the same order as they are added via the `addComponent(c : reference<Component>)` operation to the pane, i.e. the first added child will be displayed at the top while the last at the bottom of the pane.

- `HorizontalPane` – components are arranged horizontally when added as children to this pane. Similarly to the `VerticalPane`, the children are displayed in the same order as they are added, starting from left to right.

- `GridPane` – components are arranged in a grid pattern, they have to be added at specific coordinates as children. The upper left corner has the coordinates *(0, 0)*. Unlike the previous panes, the `addComponent` operation, which adds a component to the pane, requires coordinates, where the added component is placed.

- `BorderPane` – this pane is separated into five sectors: top, bottom, center, left and right, thus representing the common arrangement inside of a GUI separated into a header, menu bar, content page, contributions bar, etc. Unlike with the previous panes, children components are not added via the simple `addComponent` operation, but via setter operations, e.g. `setTop` or `setCenter`. Furthermore, the size if the components contained directly in the border pane is overridden so that they take up all available space defined by the corresponding sector.

`GUIFactoryBorders` Unit

This library unit contains an abstract type `Border`, an enumeration `BorderType` with the values `RAISED` and `LOWERED`, three non-abstract border types which are subtypes of `Border` as well as the corresponding creation operations. There are three border types, which are displayed in figure 29. The `TitledBorder` has a title in the upper left corner of the border which is given upon the creation of the border. The `LineBorder` is a simple line border, the example border in figure 29 surrounds a text field with the entry `line border` and has a thickness of two pixels. And finally, the `BevelBorder` appears either as raised or lowered against the background by using different colours. The example in figure 29 surrounds a text field with the entry `lowered bevel border`, is a lowered border defined by colours black and light grey.

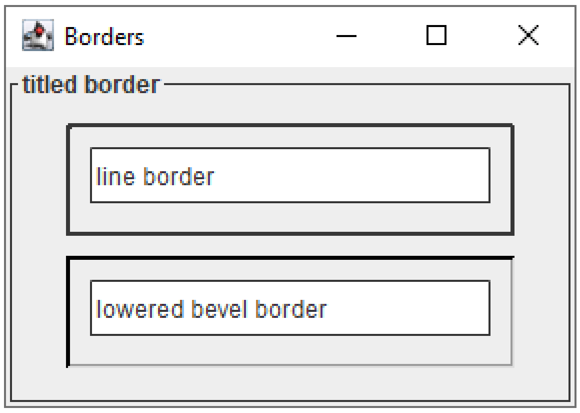


Figure 29: Examples of border types specified in `GUIFactoryBorders`.

`GUIFactoryComponents` Unit

This unit contains all components that are not `Panes`, which are summarized in a separate unit which we have already discussed earlier. As with all elements of this library, the components are abstract compositions. For each component, a creation operation is offered. In general, the components can be summarized into three groups: visual components, i.e. images and various kinds of shapes, textual components and control components, for example buttons.

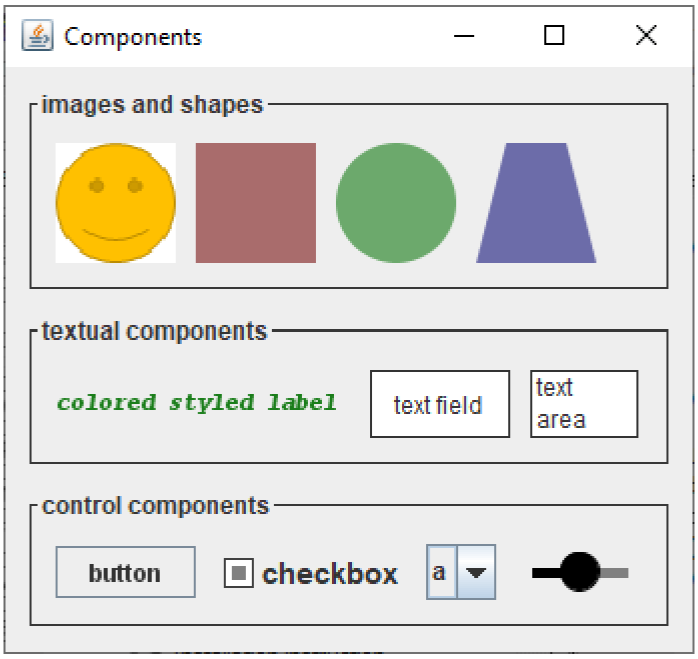


Figure 30: Examples of components in `GUIFactoryComponents`.

Images can be loaded by using both absolute and relative paths, in the latter case the path is relative to the project folder, i.e. an image with the path *img/smiley.png* is located in the folder *img*, which is located in the project folder. While the shapes `Rectangle` and `Ellipse` are self-explanatory, a `Polygon` is defined by a set of points. The polygon displayed in figure 30 is defined by four points with coordinates *(15, 0)*, *(45, 0)*, *(60, 60)* and *(0, 60)*. All shapes can be coloured, their size can be changed.

The group of textual components is made up of a `Label`, which represents simple text that can not be edited directly in the GUI without using setter operations, a `TextField`, which represents a single line text field, and a `TextArea`, wherein the text can be displayed over several lines. Users may change the font and the colour of the displayed text. The size of the `TextFields` and `TextAreas` can be altered, furthermore, the programmers may specify whether the users of the GUI are allowed to edit the text in these components.

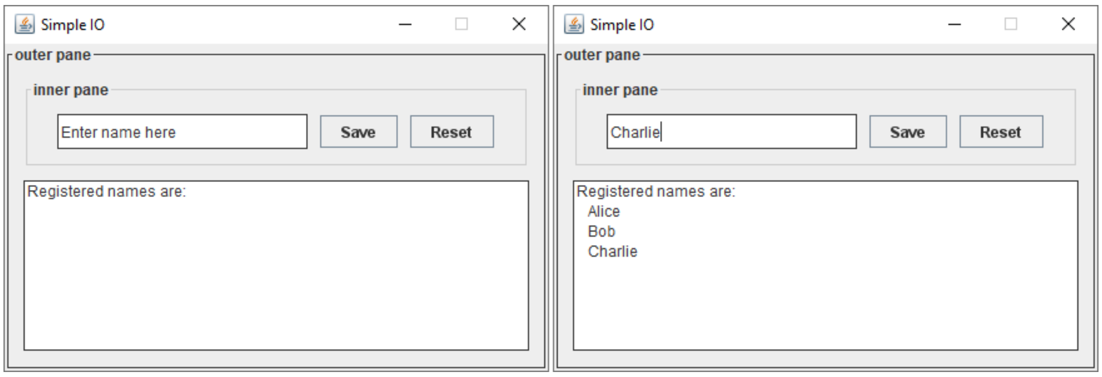
The final group consists entirely of interactive control components, including `Buttons`, `CheckBoxes`, `DropDownMenus` and `Sliders`. The button is the only control element that can perform actions when activated. The performed action is implemented in a redefined operation in a user defined subtype of `ActionTask` (see next subsection), which is passed to a button via the operation handle-ActionTask.

GUIFactoryTasks Unit

This unit includes the single abstract composition ActionTask which contains a single abstract operation actionPerformed. Furthermore, this unit does not contain a creation operation for ActionTask, users are meant to extend this type, provide their own implementation of the operation actionPerformed and pass instances of this type to Buttons. Additional tasks may be included in the future, such as tasks handling mouse or key events.

Example

In the following example we will implement a simple GUI which is displayed in figure 31. The initial state is displayed on the left, this state can be restored by pressing the **Reset** button. The right side shows the GUI after entering various names in the text field and pressing the **Save** button several times. The main layout of the GUI is defined by a vertical pane, marked by the border with the title **outer pane**. Its only two children are a horizontal pane (border with the title **inner pane**) and a text area. The contents of the horizontal pane are a text field and two buttons.



```
Registered names are:
```

```
Registered names are:
  Alice
  Bob
  Charlie
```

Figure 31: A simple example of a user interface implemented with GUIFactory.

The functionality of both buttons is displayed in listing 113, which represents the first part of the implementation. This listing contains the import instructions, as we see, all GUIFactory units are imported, as well as two subtypes of ActionTask. Both tasks work with the text field and the text area of the GUI, therefore both of them contain attributes with the corresponding types, which are initialized when the tasks are instantiated. Furthermore, both tasks override the inherited operation actionPerformed, which is executed when a button is pressed. The task `SaveActionTask`, which is meant to be used with the **Save** button, defines this operation as reading the current content of the text field and appending the resulting string to the content of the text area. Meanwhile, the task `ResetActionTask`, which is used with the **Reset** button, sets the content of both textual widgets to the strings displayed on the left side in figure 31.

```
program simpleIO
import IO
import GUIFactory
import GUIFactoryBorders
import GUIFactoryPanes
import GUIFactoryComponents
import GUIFactoryTasks

type SaveActionTask : composition extends GUIFactoryTasks.ActionTask>
   private attribute _textField : reference<GUIFactoryComponents.TextField>
   private attribute _textArea : reference<GUIFactoryComponents.TextArea>

   override operation actionPerformed()
      variable text : string
      text := _textArea@.getText() & "\n " & _textField@.getText()
      IO.writeString(text)
      _textArea@.setText(text)
   endoperation
endtype

type ResetActionTask : composition extends GUIFactoryTasks.ActionTask>
   private attribute _textField : reference<GUIFactoryComponents.TextField>
   private attribute _textArea : reference<GUIFactoryComponents.TextArea>

   override operation actionPerformed()
      _textField@.setText(" Enter name here")
      _textArea@.setText(" Registered names are:")
   endoperation
endtype
```

Listing 113: Part one of the example depicting the GUI displayed in figure 31.

The second part of the implementation is displayed in listing 114. First, we create a non-resizable `Window` with a set size and title. We continue with creating panes and other components as described at the beginning of this subsection, i.e. a `VerticalPane` with a `TitledBorder` **outer pane** as the main pane of the window object, and a `HorizontalPane` and a `TextArea` as its children. The `HorizontalPane` is marked by the `TitledBorder` **inner pane**, its children are a `TextField`, a `Button` **Save** with the assigned task `SaveActionTask` and a `Button` **Reset** with the assigned task `ResetActionTask`.

```
main
   variable window : reference<GUIFactory.Window>
   window := GUIFactory.createWindow("Simple IO", 450, 300)
   window@.setResizable(false)

   variable mainPane : reference<GUIFactoryPanes.VerticalPane>
   mainPane := GUIFactoryPanes.createVerticalPane(GUIFactory.Alignment.TOP_CENTER, 10)
   mainPane@.setPadding(10, 10, 10, 10)
   window@.setPane(mainPane)

   variable outerBorder : reference<GUIFactoryBorders.TitledBorder>
   outerBorder := GUIFactoryBorders.createTitledBorder(
                  GUIFactory.createColour(55, 55, 55, 255), 1, "outer pane")
   mainPane@.setBorder(outerBorder)

   variable textArea : reference<GUIFactoryComponents.TextArea>
   textArea := GUIFactoryComponents.createTextArea(" Registered names are:")
   textArea@.setEditable(false)

   variable secondaryPane : reference<GUIFactoryPanes.HorizontalPane>
   secondaryPane := GUIFactoryPanes.createHorizontalPane(GUIFactory.Alignment.CENTER, 10)
   secondaryPane@.setPadding(10, 10, 10, 10)

   variable textField : reference<GUIFactoryComponents.TextField>
   textField := GUIFactoryComponents.createTextField(
                  GUIFactory.HorizontalAlignment.LEFT, " Enter name here")
   textField@.setSize(200, 28)

   variable button1 : reference<GUIFactoryComponents.Button>
   button1 := GUIFactoryComponents.createButton("Save")
   button1@.handleActionTask(
                  reference SaveActionTask{_textField = textField, _textArea = textArea})

   variable button2 : reference<GUIFactoryComponents.Button>
   button2 := GUIFactoryComponents.createButton("Reset")
   button2@.handleActionTask(
                  reference ResetActionTask{_textField = textField, _textArea = textArea})

   variable innerBorder : reference<GUIFactoryBorders.TitledBorder>
   innerBorder := GUIFactoryBorders.createTitledBorder(GUIFactory
                  .createColourFromPalette(GUIFactory.Palette.LIGHT_GREY), 1, "inner pane")

   secondaryPane@.addComponent(textField)
   secondaryPane@.addComponent(button1)
   secondaryPane@.addComponent(button2)
   secondaryPane@.setBorder(innerBorder)

   mainPane@.addComponent(secondaryPane)
   mainPane@.addComponent(textArea)

   window@.showWindow()
endmain
```

Listing 114: Part two of the example depicting the GUI displayed in figure 31.

6.8. Conclusion

As demonstrated in this section, MuLE is distributed with a small set of standard libraries which can be used in the context of programming education. The provided libraries are intended to have a small number of absolutely necessary types and operations in order to be kept manageable and not overwhelming for beginner programmers. Experienced programmers may find the range of provided operations lacking for more advanced tasks, however, this is not the intended scope of application of MuLE and its libraries. Therefore, the included libraries are implemented with the same ideology behind the general design of the language.

Additional libraries can be implemented in the future, to further expand the range of educational tasks which can be solved using MuLE. Such libraries could cover, among others, the following topics: logic programming, parallel computing, database management, microcontroller programming, etc.

7. Tool Support

An educational language itself is not sufficient in a programming course, it must be supported by proper tools, such as an integrated development environment (IDE). The requirements for tool support were already discussed in section 3.3, this section will cover the provided tools. As explained in the aforementioned section, Eclipse is currently used as the IDE. Therefore, a Java version 8 installation (or higher) and an Eclipse IDE are required to run MuLE on any common operating system.

The main section of this chapter covers Eclipse and its IDE functionality currently supported by MuLE. This section focuses on such topics as the project structure and MuLE files, text editor and its features, as well as various views such as the outline tree and the console. Additionally, it covers the views and further features used when debugging a program. The subsequent section shortly explains the two execution modes *Run* and *Debug*.

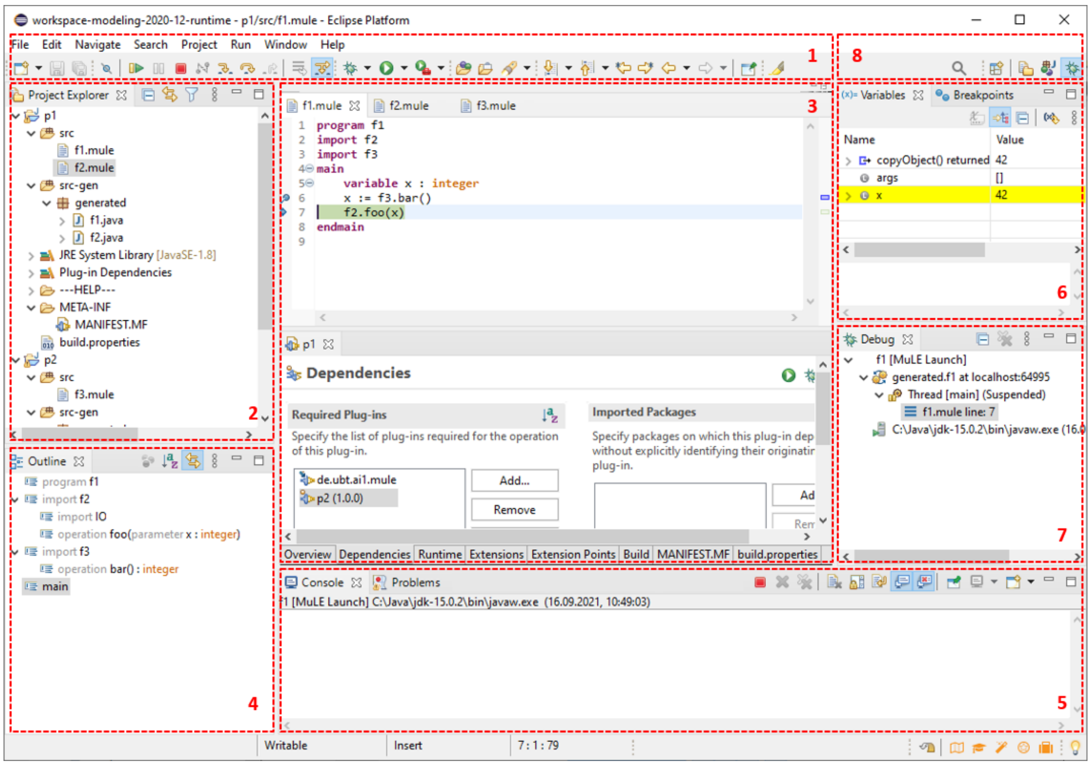
7.1. Eclipse IDE

Eclipse is a widely used IDE in the field of Java development [86]. In addition to standard tools expected from a programming environment, such as an editor, a project manager, a debugger, etc., the functionality of this IDE can be heavily expanded by plug-ins via the built-in marketplace. These plug-ins may provide support for version control systems, such as Git and SVN, additional programming languages, frameworks, etc. In fact, MuLE was developed as an Eclipse plug-in by using Eclipse with support of SVN and the Xtext framework (more in section 8), which allowed us to reuse the tools provided by this IDE to support MuLE. The installation instructions can be found in appendix A.

Figure 32 shows a screenshot of the Eclipse user interface with a MuLE program in the state of debugging. The user interface is built up as a combination of different views (e.g. the file editor view in the center and the console view at the bottom of the screenshot) combined in a perspective (e.g. the debug perspective displayed in the screenshot). The perspectives are by no means set in stone, they are merely a template which can be customized by the user by adding or removing views, changing their size and location, etc. We have separated the screenshot into eight sectors, which we will discuss in the following sections. The displayed arrangement of views is also the one used during the preliminary programming course (see chapter 9.1) by the instructor, the students are encouraged to do the same.

Sector 1: Menu Bar and Options

The first sector contains the menu bar at the top and a quick bar directly below it. The menu bar contains all options available in Eclipse separated into categories,


workspace-modeling-2020-12-runtime - p1/src/f1.mule - Eclipse Platform
File Edit Navigate Search Project Run Window Help
1
8
Project Explorer
Variables Breakpoints
Name Value
copyObject() returned 42
args []
x 42
6
Debug
f1 [MuLE Launch]
generated.f1 at localhost:64995
Thread [main] (Suspended)
f1.mule line: 7
C:\Java\jdk-15.0.2\bin\javaw.exe
7
2
3
f1.mule f2.mule f3.mule
p1
Dependencies
Required Plug-ins
Imported Packages
de.ubt.ai1.mule
p2 (1.0.0)
Add...
Remove
Overview Dependencies Runtime Extensions Extension Points Build MANIFEST.MF build.properties
Outline
4
Console Problems
5
Writable Insert 7 : 1 : 79

```
program f1
import f2
import f3
main
    variable x : integer
    x := f3.bar()
    f2.foo(x)
endmain
```

Specify the list of plug-ins required for the operation of this plug-in.

Specify packages on which this plug-in depends without explicitly identifying their originating plug-in.

```
f1 [MuLE Launch] C:\Java\jdk-15.0.2\bin\javaw.exe (16.09.2021, 10:49:03)
```

Figure 32: Screenshot of the Eclipse user interface.

e.g. the category *File* contains, among others, options which allow to create new files and projects, save and open files or import and export projects, whereas the category *Window* contains options which allow to open different views and perspectives as well as open the preferences window, which allows to change very specific settings of the IDE, such as the version of the Java compiler.

The quick access control bar contains the options which are often needed in the context of a specific perspective. The first three icons represent the options *New*, *Save* and *Save All*. Since we are in the Debug perspective, the next options represent actions reserved to debugging, such as skipping all breakpoints, resuming or terminating the program, and stepping modes. The green icons represent types of execution of the program, such as *debugging*, wherein the program is executed and halted at the first active breakpoint, and *executing*, wherein all breakpoints are ignored and the program is executed normally. The remaining icons represent buttons which allow to open a specific type declaration, search in the source file, jump between errors and annotations or last viewed locations in the editor, and other editor specific options, such as *Show Whitespace Characters*. As previously mentioned, the perspectives can be customized, i.e. these buttons can be removed or added when necessary.

The user also may right-click on almost any element in the user interface, such as a view or a textual element in the editor, to open a contextual menu which will display options available specifically for this element. For example, the user may execute a program by right-clicking on the file or in the text editor.

Sector 2: Project Explorer

This sector represents the project explorer view which acts as the current workspace. The workspace is a physical location on the drive of the machine, which contains projects. Projects are folders which contain designated source folders with source files stored within them, and non-source folders with other resources and meta-files which are meant to organize the project, e.g. specify the execution environment, the dependencies, etc. The workspace is selected when starting Eclipse, or can be changed via the *File* → *Switch Workspace* option, which will however force a restart of Eclipse.

As long as the MuLE plug-in is correctly installed in Eclipse, users can see the MuLE wizards for project and file creation when navigating to wizards via the *File* → *New* option. MuLE projects are realised as Eclipse plug-in projects, i.e. in addition to the files and folders directly related to MuLE and its implementation, the project folder includes files and folders of a plug-in project, e.g. the `MANIFEST.MF` file which specifies a dependency to the MuLE plug-in in each created MuLE project. In addition, this also allows to create functioning Eclipse plug-ins from MuLE projects.

In our example in the screenshot in figure 32, we have two projects `p1` and `p2` currently open in our workspace. The project `p1` contains MuLE files `f1` (program displayed in the editor in sector three) and `f2` (a library with a procedure `foo` which prints an integer on the console), while `p2` contains the file `f3` with a single function `bar` which returns `42`. Open projects can be referenced by other projects, e.g. as a dependency. A project can also be closed, i.e. its folder is located in the workspace folder but the project itself is inactive, meaning that it cannot be referenced by other projects. Open and closed projects are marked by different icons. Since both of our projects are open, they can reference each other, which we have made a use of by stating that `p2` is a dependency of `p1`, allowing us to import the compilation units contained in `p2` in the compilation units that are located in `p1`. This can be seen in sector three, which contains the file editor view and which we will explain soon. In the lower half of this sector we see the graphical editor of the file `MANIFEST.MF` where the plug-in `p2` is stated as the dependency in addition to the MuLE plug-in. If we remove `p2` from the list of the dependencies, we will not be able to import the library `f3` in the program `f1`, already present import instructions such as in the program `f1` (displayed in the upper half of sector three) will be reported as unresolved references.

If we look closely at the project `p1` in the *Project Explorer* view, we see two folders `src` and `src-gen` which are designated as the source folders of the project, meaning that our source files are located there. The former folder contains `.mule` files which are created and edited by the user, while the latter contains the generated `.java` files, which are executed in the background and should not be tampered with. Finally, each MuLE project contains a `---HELP---` folder with `.txt` files which contain the information concerning the standard libraries (basically a shortened version of the contents presented in chapter 6) and short programming examples demonstrating the various language constructs and the functionality of the more complex standard libraries.

Sector 3: File Editor

The central and the biggest part of the user interface is usually taken by the source file editor, displayed in the sector three. In our concrete example we have split this sector into two file editor views, to demonstrate that the editor may look differently depending on the format of the opened file and whether a specific editor is provided for this format. In the upper half of this sector we have the source file editor with the usual IDE functionality expected from such an editor, while in the lower half we have the graphical editor with project relevant settings, where we can edit the specifics of the `MANIFEST.MF` file, plug-in extensions, etc.

Usually, we would rather use a single view to maximize the available work area. Programmer beginners are not required to manipulate project relevant settings, and are even advised not to do so. As depicted in the upper half of this sector, several files can be opened at the same time marked by their respective tab, with only one of them visible in a single view.

As mentioned, programmer beginners should stick to the source file editor, which offers significantly more visual feedback than most common text editors, which are not designed with programming in mind. Following features are provided by the textual source file editor:

- *Syntax highlighting* is the most obvious feature of the text editor. By default, the keywords are coloured **mauve**, data types are **orange**, string literals are **blue**, comments are **green** and everything else is **black**. These settings can be overridden in the Eclipse preferences page.

- *Line numbers* are extremely helpful when discussing examples, assignments and their solutions with the students. Users may toggle the line numbers on and off or place breakpoints at specific lines, which are used to suspend the execution of the program at these lines during the debugging process. If parts of code are violating grammar or validation rules of the language, an error marker is placed close to the corresponding line number.

- *Folding* allows to collapse or expand specific text fragments, hiding or showing parts of source code to maintain overview, which may become helpful when writing larger programs. Type declarations and all block based language constructs, i.e. the main procedure, operations and control flow defining statements, can be folded.

- *Error messages* are displayed if a part of the source code does not comply to grammar or validation rules specified in chapter 5. The corresponding checks are performed at the same time the code is written providing immediate feedback. The program will not compile if at least one validation error is present in the code. The rule breaking text fragment is underlined in red. In addition, an error marker is placed at the corresponding line number and an error message is displayed when hovering with mouse cursor over the error marker or the offending text fragment.

- The *quick fix provider* offers proposals for automatic fixes for most validation error messages. For example, in the context of a compilation unit, the first error that can happen is when the identifier of the unit is not equal to the file name. The offered quick fix changes the identifier of the unit to the file name. Other types of errors may have more than one proposal, e.g. in case a program unit is lacking a main procedure, one of the proposals adds an empty main procedure while the other proposal changes the unit to a library. Due to their automated nature, quick fix proposals are not provided for every available error message, for example, in cases when a duplicate name is registered since providing meaningful names can not be achieved by automated mechanisms.

- The *automatic formatter* of the source code can be invoked by *right click* → *Source* → *Format* or by the corresponding shortcut *Shift+Ctrl+F*.

- The *content assist provider* offers source code proposals at the location of the textual cursor in the editor. The content proposal mechanism is invoked by *right click* → *Source* → *Content Assist* or by the corresponding shortcut *Ctrl+Space*. The proposals depend on the context, i.e. the proposed identifiers are the ones that are visible at the current position of the cursor as defined by the scoping rules (section 5.1), the proposed keywords are allowed in the context of the encompassing language construct, etc. Selecting a proposal will automatically insert the proposed keyword or an identifier. In case of operations, the proposal displays the entire signature of the operation, i.e. it includes parameter types and the return type. The generated operation invocation contains placeholders for expected parameters. For example, if content assist is invoked in the context of the library `IO` (section 6.1), i.e. when the cursor is placed after `IO.`, one of the proposals is displayed as `operation writeString(parameter arg : string)`. If this proposal is selected by the user, the generated source code is `writeString(___ARG__STRING___)`.

- *Find/Replace* – this function is present in any common text editor and can be accessed by pressing the key combination *Ctrl-F*.

Sector 4: Outline Tree

The outline tree provides a quick overview of the program currently displayed in the source file editor. Furthermore, it allows quick access to the elements displayed in the tree, e.g. clicking on the node `main` in the tree will highlight the keyword `main` in the text editor. The nodes displayed in the outline tree represent the program unit itself, import instructions, type declarations, operations and the main procedure. Import nodes contained directly in the program and type nodes, i.e. compositions and enumerations, can be folded and unfolded, which allows to inspect the contents of these elements in the outline tree. Operations and the main procedure nodes can not be unfolded.

Sector 5: Console and Problems Views

The active tab in this sector represents the *Console* view. User interactions via the console are supported by `write` and `read` operations of the library `IO` (see section 6.1). Runtime errors are also displayed in this view. Additionally, the *Problems* tab is visible in the sector. The problems view lists current error and warning messages, the corresponding resource (i.e. the file), the line of code, and allows to quickly access the location causing the problem by double clicking the list entry.

The control bar in the *Console* view, visible in the top-right corner of this sector, contains, among others, buttons which toggle, whether the console should be automatically displayed (if it is not already visible) when the output changes, clear the console contents, as well as the terminate button (red square icon, which is also present in the quick access control bar in sector 1).

Sector 6: Variables and Breakpoints View

The views displayed in this sector are relevant for the debugging process. The active view, *Variables*, displays the currently existing data containers and their values if the execution of the program is halted at a specific line during debugging. To halt the execution of the program, a breakpoint must be set up in the source file editor, for example by double clicking on a line number. The breakpoints are listed in the breakpoints view. Users can quickly access the location of the breakpoint in the source code, disable, enable or, ultimately, delete breakpoints.

In our screenshot, we see a breakpoint placed in line six in the text editor view in sector 3. The program is in the middle of the debugging process, meaning that it was executed by pressing the debug button and was halted as soon as it has reached the first breakpoint. We have then executed one more line by pressing the step-over button in the quick access bar (sector 1), as we can see, the program is currently suspended in line seven. The variable `x` is highlighted in the *Variables* view, meaning that its value has recently changed, this change happened when we have executed the previous line. This way, users have a good visual feedback over the current state of the program when debugging it. Since Java is executed in the background, we can also see some traces of the Java code, i.e. the `args` parameter of the `main` method and the return value of the `copyObject` method (see chapter 8).

Sector 7: Debug View

The debug view displays currently running processes, which can be suspended by halting at a breakpoint, waiting for user input, or running in an endless loop. In our example we have only one running program with the main-procedure which is currently suspended in line seven. As we can see, a Java program is actually executed in the background. However, the suspended line is that of the MuLE text editor, which is also displayed when the program is halted during the debug process. When several subroutines are invoked in a single program but have not yet finished their execution, they are displayed in a stack-like arrangement in this view, with the uppermost subroutine being the last executed one with the others waiting for it to finish its execution. Users may switch between these subroutines to observe their internal state as well as see in which line they were halted.

Similarly, several programs can be displayed in this view, for example when the user is debugging several programs at once or running several programs which are waiting for user input or a specific event, etc. The latter case is often observed when beginner students involuntarily implement an endless loop, and keep executing the program under the false assumption that nothing is happening. Running programs are marked by the green *play* icon (visible in the last entry in the debug view), which is replaced by the red *stop* icon on terminated programs. Each running process can be selected in this view and terminated manually. Users may also terminate all running programs at once. Terminated programs can be removed to clean up the view.

Sector 8: Perspectives Bar

We have called this sector the perspectives bar since it mostly contains buttons tasked with opening new or displaying previously opened perspectives, however the first icon in the bar represented by the magnifying glass allows to search for various commands in the Eclipse IDE. For example, if the user clicks on this icon and types *co* in the displayed pop-up view, the displayed actions will include, among others, options to open the console view and collapse all foldable regions. The next icon allows to open other perspectives. The following icons represent the active as well as previously opened perspectives. The active perspective, in our case the debug perspective, is highlighted.

7.2. Execution

Upon saving a MuLE compilation unit, Java code is generated (see chapter 8) as long as no compile time errors are present. The MuLE program can then be executed by pressing the *Run* or *Debug* button, or by right clicking in the editor frame or directly on a file in the project explorer view and selecting these options in the contextual menu. The generated Java code is executed in the background.

Breakpoints can be set up in order to initiate the debugging process with stepwise execution. Even though Java code is executed in the background, MuLE source code is displayed in the editor. The current line of execution as well as the state of the variables can also be seen during this process as explained in the previous section. Running programs can be manually terminated if necessary, for example if the user has executed a program with an endless loop.

7.3. Conclusion

Since Eclipse is a professionally used IDE, the amount of offered functionality may become quickly overwhelming for a beginner programmer. While the project structure, the text editor, the console and the outline view are more or less easy to understand, experience shows that beginner programmers are usually not using the debugger when trying to locate runtime or semantic errors in their code. Some of them will even ignore the compilation errors, which are marked directly in the source code. Therefore, a huge amount of offered functionality should not be discussed with the students, at least not at the beginning.

That being said, we have not provided a dedicated beginner friendly IDE specifically implemented to support MuLE, since this task would go beyond the scope of this dissertation. In future, it should be considered to provide a simpler IDE comparable to BlueJ [85] or similar programming environments designed specifically with education in mind and oriented towards beginner programmers. BlueJ is merely an example, the IDE should be compatible in its simplicity but still be implemented with the goal to support MuLE, its intended use as well as the chosen educational approach.

8. Implementation

While the previous chapters focused on the requirements, the design and the specification of MuLE, the topic of this chapter is the implementation of our language. Although the majority of our students are using Windows, some of them are using other operating systems. Therefore, it was necessary to provide either an implementation for each operating system that might be used by the target audience, or a platform independent implementation, which was one of the reasons why we have decided to use Java as the base of our implementation. Other reasons included the fact that it is a multi-paradigm general purpose language with a powerful library support, meaning it promised an easier implementation of the planned concepts (see chapter 4), and last but not least our own experience with this language. This allowed us to use the Xtext framework integrated into the Eclipse IDE, which in turn allowed us to integrate MuLE into Eclipse and reuse its IDE functionality for our language (see chapter 7).

Xtext [110] is a framework specifically designed to implement text-based domain-specific languages (DSLs). Usually, a DSL is far less flexible than a general purpose language, since it is designed with a very specific goal in mind, such as SQL and R which we have mentioned in chapter 3. Neither language can be used for tasks, that may be totally different in nature, like implementing a generic data structure, algorithms which will work with it and writing tests for the implementation with a single language. However, while a GPL can be used to implement any scenario that is computable on a *Turing Machine*, a DSL designed to solve specific problems will be a far more user-friendly tool when applied in their intended field. Thus, using a framework designed to implement DSLs might sound like a questionable decision at first. However, since both DSLs and GPLs are languages defined by their abstract and concrete syntax as well as their static and execution semantics [111], a tool capable to develop text-based DSLs should also be applicable to the development of text-based GPLs.

Xtext was developed as a part of the *Eclipse Modelling Project* [112], it relies on other tools and frameworks which contribute to this project, thus we should first briefly explain these tools as well as the underlying concepts. The first two sections of this chapter cover the Eclipse Modelling Project in general and the Xtext framework in particular. The third section offers an overview over the implemented MuLE plug-in projects, which are then integrated into an Eclipse installation to provide MuLE support. Section 8.4 focuses on the implementation of language specific modules defined by the specification of MuLE (see chapter 5), i.e. the grammar, code generation, type system, scoping and compile time validation. The final section covers the implementation of modules specific to tool support (see chapter 7), such as the debug support, the outline tree provider or the project and file creation wizards.

8.1. The Eclipse Modelling Project

By itself, *modelling* is not a new concept. For example, by drawing a blueprint, an architect creates a model which is then used to build a structure. A teacher who explains complex mechanisms in a simplified way so that the students could understand it based on their present capabilities, presents them an abstracted model of the corresponding system. Thus, we have two types of models: *prescriptive*, which are used to specify future projects, and *descriptive*, i.e. abstract representations of complex systems. Both are used in the context of software development of a software system, for example when designing modules or presenting these to the stakeholders in a less technical way, by using various diagrams defined by the *UML2* standard [71]. The entire discipline of *Model Driven Software Development* (MDSD) was born out of the idea of bridging the gap between the models and the implementation by generating code directly from these models, thus increasing productivity, quality and reusability [113]. The general approach in MDSD is to provide a model which defines a future system as well as generator templates with transformation rules that specify how this model will be translated into source code or another model. The model must correspond to the rules formally defined by its metamodel, which may define itself or be defined by its own meta-metamodel.

The *Eclipse Modelling Project* was developed as a powerful toolkit which supports MDSD. The core of this toolkit is represented by the *Eclipse Modelling Framework* (EMF) [114], which unifies three technologies: *Java*, *UML* and *XML Schema*, allowing to represent a model using one or all of them. EMF includes its own standard for models: the *Ecore metamodel*, which is a simplified subset of the UML standard. The Ecore metamodel defines itself, meaning that it acts as the meta-metamodel for itself as well as for all other models created using EMF.

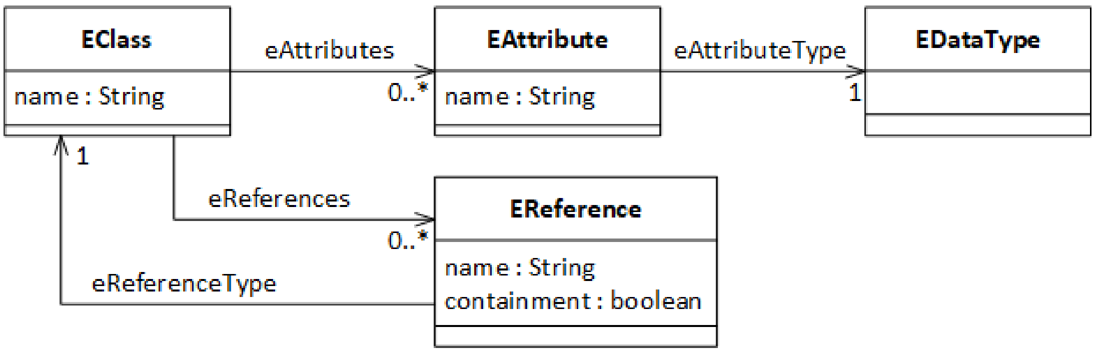


Figure 33: A simplified subset of the Ecore metamodel [114].

A subset of the Ecore metamodel is displayed in figure 33, it contains class definitions that specify Ecore classes `EClass`, `EAttribute`, `EDataType` and `EReference`. The last class demonstrates one of the bigger differences between Ecore and UML, while an association in UML can be bidirectional and can also be an aggregation or a composition, a reference in Ecore is unidirectional and can only simulate compositions if it is marked as a containment reference. As we can see, an `EClass` has a name and can have an arbitrary number or `EAttributes` and `EReferences`. While the type of the first ones (`EDataType`) covers both primitive and object types, the type of an `EReference` is always an `EClass`.

As we can see, the graphical representation of the subset of the Ecore metamodel in figure 33 looks similar to any other UML class diagram. Previously, we have mentioned that EMF is based on three technologies, and since we have already covered UML, we have yet to see which roles are played by XML Schema and Java in this framework. Ecore models are serialized using the *XML Metadata Interchange* standard [115], wherein model elements are defined by XML tags with the specification of their types and various attributes. A serialized shortened version of the subset of the Ecore metamodel is displayed in listing 115. This example is not an excerpt of the actual Ecore metamodel, but our own model representing the same structure as in listing 115. The root element of the model, `ecore:EPackage` represents packages and stores, in addition to other data, information required to load the model as a resource, for example via its `nsURI`. In it, classes are listed as `eClassifiers`, the super type of `EClass` in the actual full Ecore metatmodel, thus we can see our `ecore:EClass` with the name `EClass`, i.e. an `EClass` which defines other `EClasses`. It has an `EAttribute` with the identifier `name` typed `EString`. Furthermore, it has two `EReferences` referencing the classes `EAttribute` and `EReference` with their corresponding names. The tags for `EAttribute` and `EReference` are listed as `eStructuralFeatures` named after the super type of these classes in the full Ecore metamodel.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="ecoreMM"
           nsURI="http://www.example.org/ecoreMM" nsPrefix="ecoreMM">
   <eClassifiers xsi:type="ecore:EClass" name="EClass">
      <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
               eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
      <eStructuralFeatures xsi:type="ecore:EReference" name="eAttributes"
               upperBound="-1" eType="#//EAttribute"/>
      <eStructuralFeatures xsi:type="ecore:EReference" name="eReferences"
               upperBound="-1" eType="#//EReference"/>
   </eClassifiers>
   <!-- Further eClassifiers for EAttribute, EReference and EDataType -->
</ecore:EPackage>
```

Listing 115: A shortened version of our own implementation of the Ecore model in figure 33 serialized using XMI.

Now that we have a serialized model, we can start generating Java code from it, which is, after all, one of the main goals of MDSD. The model element `EClass` is mapped to a Java interface `EClass` (listing 116) and the corresponding implementation class `EClassImpl` (listing 117). The transformation rules from Ecore to Java are implemented in EMF, we do not need to provide them in this case. We have shortened the generated code in both listings, the functionality of the displayed methods is self explanatory.

```java
public interface EClass extends EObject {
    String getName();
    void setName(String value);
    EList<EAttribute> getEAttributes();
    EList<EReference> getEReferences();
}
```

Listing 116: The generated Java interface `EClass`.

```java
public class EClassImpl extends MinimalEObjectImpl.Container
                        implements EClass {
    protected static final String NAME_EDEFAULT = null;
    protected String name = NAME_EDEFAULT;
    protected EList<EAttribute> eAttributes;
    protected EList<EReference> eReferences;

    protected EClassImpl() {
        super();
    }
    public String getName() {
        return name;
    }
    public void setName(String newName) {
        String oldName = name;
        name = newName;
        if (eNotificationRequired())
            eNotify(new ENotificationImpl(this, Notification.SET,
                        EcoreMMPackage.ECLASS__NAME, oldName, name));
    }
    public EList<EAttribute> getEAttributes() {
        if (eAttributes == null) {
            eAttributes = new EObjectResolvingEList<EAttribute>(EAttribute.class,
                        this, EcoreMMPackage.ECLASS__EATTRIBUTES);
        }
        return eAttributes;
    }
    public EList<EReference> getEReferences() {
        // similar to getEAttributes
    }
    // further generated methods, which are unrelated to our example
}
```

Listing 117: The generated Java class `EClassImpl`.

As we can see, the generated code can be used out of the box. There is, however, one limitation to this, we can add `EOperations`, which are mapped to Java methods, to our `EClasses` in the model in order to define the functionality of our classes. However, there is no way to specify their semantics, the generated methods will throw an exception as long as the user does not provide an actual implementation manually.

As a final note, even though Java does not support direct multiple inheritance on classes, EMF supports this feature which we have made use of in our implementation. The multiple inheritance is simulated via Java interfaces, i.e. while a class may extend only one other class, it may implement multiple interfaces which are effectively super types of this class. Since EMF provides interfaces for all of its classes and a corresponding factory class, this simulation is even less noticeable.

8.2. The Xtext Framework

Now that we know what EMF is and how we can generate Java code from Ecore models, let us move on to the Xtext framework, which is built upon EMF and which we have used to implement MuLE. One of the defining factors of any language is its grammar. In chapter 5 we have specified language constructs of MuLE primarily based on its grammar. So let us again take a look at the definition of a MuLE compilation unit, which we see in listing 118.

```
CompilationUnit:
    ('program' | 'library') ID
    Import*
    ProgramElement*
    MainProcedure?;

Import: 'import' [CompilationUnit];

ProgramElement: TypeDeclaration | Operation;

MainProcedure: 'main' Block 'endmain';
```

Listing 118: Composition of a MuLE compilation unit.

Our compilation unit can be a program or a library, has an identifier, which can be represented as a string of characters (see section 5.3.2), a set of import instructions and program elements (i.e. type declarations and operations) and it can include a main procedure, which has a block of statements. In a sense, it is very similar to the definition of the `EClass` from the previous section, meaning that we can create a similar model, that would resemble our grammar. The resulting model is displayed in figure 34, to keep it simple, we have left out most of the structural features of the class `Operation` (except for the `block` reference), this is just an example after all. Although it bears a resemblance to a typical UML class diagram, it should be noted that we are using `EDataTypes` to specify the types of our attributes and do not use bidirectional associations, the reference `imports` is a containment `EReference` while `importedNamespace` is another `EReference` without the attribute `containment` set to `true`. This model is an instance of the Ecore metamodel, meaning that we could now start generating Java code from it.

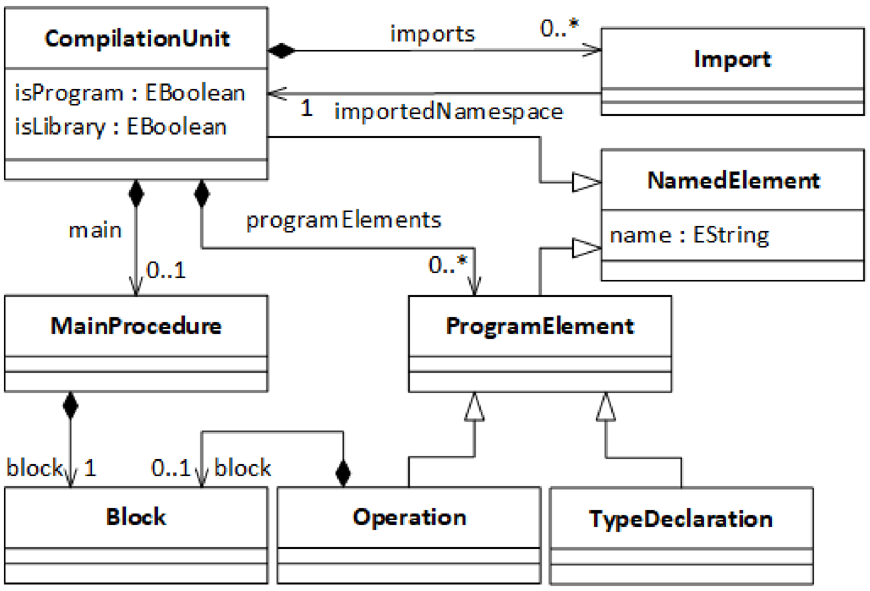


Figure 34: A class diagram representing the grammar in listing 118.

However, although we could now define the entire model of our language this way (which is also possible in Xtext), traditionally the context free grammar of a programming language is specified by using a textual notation such as the *Extended Backus-Naur Form* or its variant that we have used in this thesis and that is in fact used by Xtext, as we will see shortly. The grammar of the language acts as a model, upon which a *lexer* and a *parser* are implemented, the former attempts to separate the stream of characters into grammar confirming tokens (keywords, identifiers, operators, etc.) while the latter evaluates whether the sequence of tokens contributes to a valid language construct defined by the grammar and builds a parse tree, a data structure which closely represents the parsed syntax, while doing so. In addition to the parse tree, the abstract syntax tree (AST) can be created, which excludes terminal and non-terminal symbols and stores only the language constructs which carry all the necessary semantic information. The AST can then be analysed by further tools to ensure validity of the program that can not be defined by the grammar alone, such as type conformity, scoping and further validation checks. Therefore, the AST is basically the model of our program, while the grammar is its metamodel.

The cornerstone of an Xtext project is an `.xtext` file which includes the grammar of the language written in a textual form similar to the one in listing 118 but with some additions (see listing 121), which allow it to be translated into an Ecore model represented by the diagram in figure 34. For instance, we need to provide actual names for our `EAttributes` and `EReferences`, for example `name=ID` in our compilation unit is an attribute with the identifier `name` and type `ID`. We will not go further into implementation details yet, we have an entire section 8.4.1 dedicated to the grammar in this chapter, this example is just meant to demonstrate how Xtext maps the textual representation of the grammar to the corresponding Ecore model.

```
CompilationUnit:
    (isProgram?='program' | isLibrary?='library') name=ID
    imports+=Import*
    programElements+=ProgramElement*
    main=MainProcedure?;

Import: 'import' importedNamespace=[CompilationUnit];

ProgramElement: TypeDeclaration | Operation;

MainProcedure: 'main' block=Block 'endmain';
```

Listing 119: Composition of a MuLE compilation unit.

As explained earlier, both the lexer and the parser are entirely dependent on the grammar of the language, meaning that once the grammar is specified, these modules can be generated based on the grammar by using parser generator tools. One of such tools is ANTLR [116], which generates recursive descent parsers and is used by Xtext for exactly this specific task. While the current version 4 of ANTLR generates adaptive LL(*) parsers which are able to parse sequences defined by left-recursive grammars, Xtext uses the previous version 3 of ANTLR which does not offer this feature. This means, that the we have to take care when designing our grammar in order to avoid ambiguity. This is especially evident when writing the expression grammar (section 5.6), which is recursive by its nature and must be left factored in Xtext.

The initial step using Xtext is, therefore, to write a grammar using an EBNF-like notation, and then generate Xtext artefacts from this grammar. One of these artefacts is an EMF model, which represents the grammar, as well as the Java code generated from this model. Another generated artefact is the ANTLR lexer and parser. These modules are defined by the grammar and, usually, do not require any post-generation editing by the developer of a DSL. In addition to these modules, Xtext generates class stubs for code generation, scoping and validation, which must be manually implemented by the user. Finally, Xtext generates the infrastructure required to integrate the implemented language into Eclipse. The project structure of the MuLE plug-ins as well as the implementation of the various modules will be discussed in the subsequent sections, but before that we still have to cover some of the remaining features of Xtext.

Now that we have our grammar and the generated artefacts, we can actually write our very first program. When we write our program, the Xtext generated parser builds simultaneously both a parse tree and an AST, which is an instance of the generated EMF model. It should be noted, that the model which acts as the AST is not serialized by default in Xtext, but merely stored in memory for as long as necessary [110].

Let us take a look at a simple MuLE program (listing 120), which we can write based on the snippet of the grammar included in this section. The generated AST is displayed in the object diagram in figure 35, this AST is an instance of the generated EMF model represented as a class diagram in figure 34.

```
program f1

operation foo() endoperation

main endmain
```

Listing 120: A simple program example which we are about to parse.

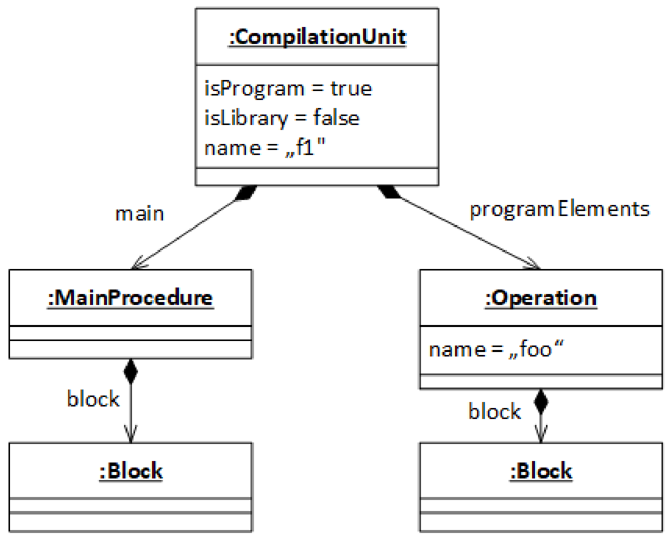


Figure 35: An object diagram representing the AST of the program in listing 120.

Earlier, we have mentioned that Xtext generates additional modules, for example for code generation or semantic checking. This is where the parsed AST becomes very important. Up until now, our written program did not actually carry any information, except for the fact that it is grammatically correct. However, we can now compile this program, for example by generating Java code by using the model-to-text template based transformation tool *Xpand* [112], which is integrated into Xtext. The generated Java code can then be executed, effectively allowing us to execute our program.

Additionally, Xtext allows to implement logic of various DSL modules, e.g. for the purpose of validation checking or as an additional help in the generator, by using *Xtend* [110][112], a programming language which is very similar to Java, it reuses its type system, allows to use all of its libraries and adds such features like type inference and more elegant lambda expressions which make the syntax more readable. By default, most of the generated module stubs use Xtend, however Java can be used just as well.

Finally, since a programming language can only benefit from being supported by a powerful IDE and Xtext is an Eclipse based framework, it generates the infrastructure necessary to integrate a DSL implemented with Xtext into Eclipse. The subsequent sections will present first the implementation of MuLE and then its Eclipse based UI support using the tools discussed in this section.

8.3. Plug-in Project Structure

Provided, Xtext is installed as an Eclipse plug-in, the corresponding plug-in projects for the development of a DSL can be created via the project creation wizards. The users must enter the project name and specify the extension of the DSL source files. In our case the project name is `de.ubt.ai1.mule` and the corresponding file extension is `mule`. Users may additionally specify, whether they want to generate projects intended to contain *JUnit* test cases for the language itself and its UI support, as well as plug-ins related with setting up the update site for the language plug-ins.

The three main plug-in projects, which are downloaded when installing MuLE, are therefore:

- `de.ubt.ai1.mule` – this is the main project, which contains the grammar definition as well as all modules related directly to the language, such as the parser, linker, code generator, validation, etc. The implementation of these modules is discussed in section 8.4. If the language is meant to be used separately from Eclipse, this is the only required plug-in.

- `de.ubt.ai1.mule.ide` – this project contains services for the content assist provider, which is implemented in the `ui` project. We have not performed any changes within this project, thus it will not be discussed any further.

- `de.ubt.ai1.mule.ui` – this project contains Eclipse related tool support functionality (discussed in chapter 7), such as the syntax highlighting in the text editor, debug support, wizards, etc. The implementation of these modules is discussed in section 8.5.

8.4. Language Specific Modules

This section focuses on the implementation of the language itself, thus we are going to discuss the modules included in the project `de.ubt.ai1.mule` here. As explained in section 8.2, the foundation of any Xtext project is the grammar of the language, which is used as a model from which the lexer and the parser are generated automatically. We have not altered the implementation of neither of these modules, therefore we will merely give a brief explanation of their core functionality before moving on to other artefacts.

The generated Xtext lexer is implemented as an extension of the ANTLR lexer. It is tasked with tokenizing the input, i.e. the entire program is passed to the lexer as a stream of characters upon its initialization, which then attempts to match the characters to keywords, operators, literals, comments, white spaces, etc., at the current position in the input. For example, one of the keywords in MuLE is `program`, thus the generated MuLE lexer includes a method, which attempts to match the input stream (which, for example, could start as `[p,r,o,g,r,a,m, ,f,1]`) against the string `"program"`. If the characters can be successfully matched, a corresponding token object is created and added to the stream of tokens (e.g. `[program, ,f1]`). The internal state of the lexer is updated and it continues to match the subsequent characters until the end of file is reached. If the characters cannot be matched, for example if the input starts with the character string `programm`, the token cannot be recognized, the lexer computes the position and the offset of the offending character string and reports it as an error.

Similarly to the lexer, the generated Xtext parser is implemented as an extension of the ANTLR parser. The parser accepts a token stream and checks whether the tokens contribute to a grammatically valid language construct. It contains methods, i.e. parser rules, which attempt to match actual tokens from the input against the expected tokens depending on the context. For example, a MuLE compilation unit starts either with the keyword `program` or `library`, thus the generated parser method tasked with parsing the entire compilation unit starts by matching the first token to either `program` or `library`, then continues to match the identifier and, if present, import instructions, type declarations, operations and finally the main procedure. All of these language constructs are checked by their own parser rules. The output of the parser is a parse tree, which contains the entire lexical information of the program based on tokens, including their position in the source text, as well as an abstract syntax tree, which is a semantic representation of the program realized as a non-serialized EMF model which is an instance of the metamodel generated from the grammar of the language. If the input cannot be parsed, for example either a token could not be recognized by the lexer (e.g. `programm` instead of `program`) or the token is misplaced (e.g. `main main` instead of `main endmain`), the parser generates a node with a syntax error in the parse tree. This node contains the error message and its location in the source code, which is then displayed to the user in the editor. Thus we have a parse tree, which is used to relay information specific to the text editor and the abstract syntax tree, which we can use for other tasks like validation checks and code generation.

8.4.1. Grammar and the Resulting Metamodel

As already mentioned, all other aspects of a language are dependent on its grammar, therefore it has to be specified first. Generally, Xtext users are advised to provide a less restricting grammar and then write validation rules which would prevent occurrence of specific cases at the same time providing more understandable messages. Xtext offers two approaches to define the grammar, as an extension of either the `Terminals` or the `Xbase` grammar [110]. *Xbase* is an expression language designed to reuse the Java type system, the language Xtend is built upon Xbase. By choosing the Xbase approach, users get a ready to use expression grammar which imitates Java, as well as code generation, scoping, validation and debugging support for the constructs defined by this grammar. They still have to implement mapping for user defined types specified by their own grammar to Java types. Nevertheless, choosing this approach promises quicker results. However, once we have tried to use this approach initially, we have noticed that a large amount of work has to be invested into redefining and reimplementing specific constructs which differ from Java syntax. Furthermore, although the debug support is automatically provided, the generated code includes intermediate variables which are not present in the DSL code, however are visible when debugging it, which may confuse beginner programmers.

Therefore, we have decided to use the approach with the `Terminals` grammar, which meant that we had to implement more from the ground, however, we also had more control over the implementation. The `Terminals` grammar (the definition is included in appendix B) specifies terminal symbols for identifiers, integer and string literals, comments and white spaces. We have redefined the rule for identifiers to disallow the placement of the symbol `^` at the start of an identifier as well the rule for string literals to allow to use only double quotation marks instead of additionally single quotation marks to denote string literals.

The entire grammar is located in appendix B, snippets of this grammar without Xtext related implementation details were discussed in section 5. In this subsection we will discuss the general approach we took when writing this grammar based on smaller examples taken out of the entire grammar. Since we have already used the MuLE compilation unit as a running example throughout this chapter, we will continue by looking at the first part of the grammar (see listing 121) which specifies this construct as well as other elements directly related to it.

The Compilation Unit and its Elements

As mentioned, we have decided to use the `Terminals` grammar as the base of our own grammar, which can be seen in the first line in listing 121. The next line specifies the name and the URI of the generated EMF model which then acts as the metamodel of our language.

```
grammar de.ubt.ai1.mule.MuLE with org.eclipse.xtext.common.Terminals
generate muLE "http://www.ubt.de/ai1/mule/MuLE"

CompilationUnit:
    (isProgram?='program' | isLibrary?='library') name=ID
    imports+=Import*
    programElements+=ProgramElement*
    main=MainProcedure?;

Import: 'import' importedNamespace=[CompilationUnit];

NamedElement: EnumerationValue | TypeDeclaration | Feature | CompilationUnit;

ProgramElement: TypeDeclaration | Operation;

Feature: Attribute | VariableDeclaration | Parameter | Operation ;

Operation:
    override?=('override')? visibility=VisibilityModifier? abstract?=('abstract')?
    'operation' name=ID '(' (params+=Parameter (',' params+=Parameter)*)? ')'
    (':' type=DataType)? (block=Block 'endoperation')?;

VisibilityModifier: 'private' | 'protected';

Parameter: 'parameter' name=ID ':' type=DataType;

MainProcedure: 'main' block=Block 'endmain';

Block: {Block} statements+=Statement*;
```

Listing 121: An excerpt of the MuLE grammar containing the definition of the compilation unit and its elements.

Let us now take a look at the rule `CompilationUnit`, which specifies that the initial keyword is either `program` or a `library` with a corresponding `isProgram` and `isLibrary` attribute. Both attributes are typed as `EBoolean`, which is denoted by the operator `?=` after the attribute name followed by the respective keyword. This means, that if the unit starts with the keyword `program`, the attribute `isProgram` in the instance of `CompilationUnit` in the AST is set to true and `isLibrary` to false, since a unit can not be a program and a library at the same time. Examples of a class diagram and the corresponding object diagram for the AST are displayed in figures 34 and 35 in section 8.2 of this chapter. Each unit has a single `name` attribute defined by `ID` rule, which is mapped to the EMF type `EString`. An attribute name followed by a simple `=` is mapped to a `0..1` relation in the metamodel of the language, parser checks ensure that an identifier is actually present ultimately resulting in cardinality `1` in this case.

The class diagram representing a part of the model covered by the grammar in listing 121 is displayed in figure 36. As already mentioned, this model acts as the metamodel for our AST, which can be used for code generation, for example.

Each compilation unit may have several import instructions or none at all, which is marked by the operator `+=` and a `*` symbol after the `Import` reference. Same behaviour is seen with the attribute `programElements` which results in a `0..*` relation to instances of type `ProgramElement`, which are also either instances of `TypeDeclaration` or `Operation`. Speaking of import instructions, the `[CompilationUnit]` denotes a cross reference, i.e. an existing compilation unit is referenced by `importedNamespace`.

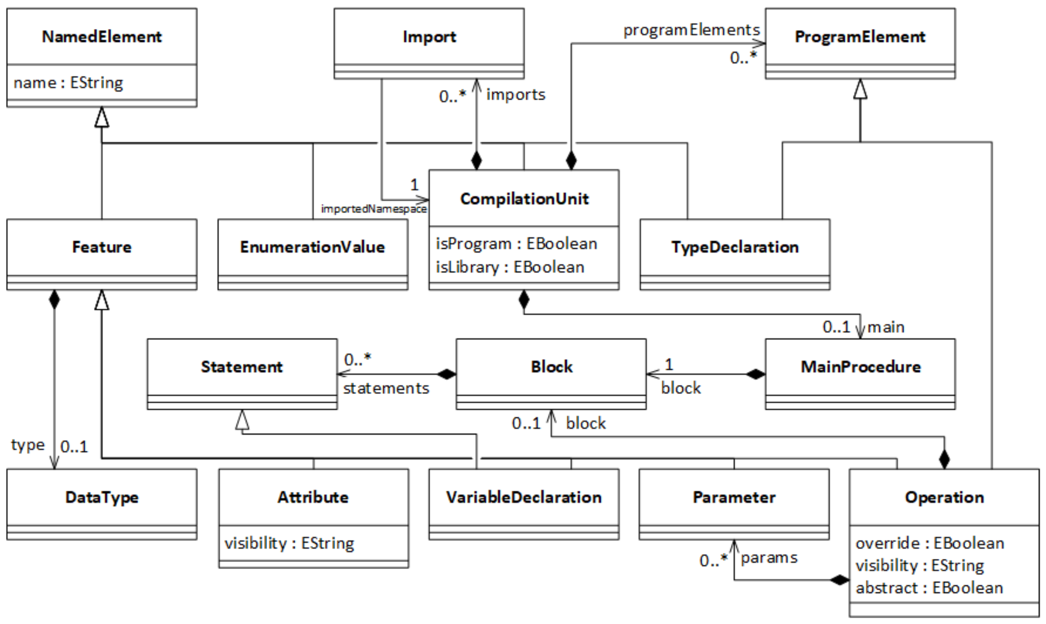


Figure 36: A class diagram representing an excerpt of the metamodel, which contains elements mentioned in the grammar in listing 121.

The reference `main` is typed `MainProcedure` and has a relation `0..1` to instances of this type, which is denoted by the symbol `?` after the `MainProcedure` reference. The reference `block` in the `MainProcedure` has the cardinality `1` to instances of type `Block`, marked by the absence of the question mark symbol. A `Block` represents a collection of `Statements`, {`Block`} at the beginning of the rule ensures that a `Block` object is in fact instantiated (even if the list of statements is empty) and not consumed by the encompassing rule, i.e. by `MainProcedure`, `Operation` or one of the statements.

As previously mentioned, each CompilationUnit can specify a list of ProgramElements, which are either TypeDeclarations or Operations. We have included only the Operations in the grammar excerpt in listing 121 and will later take a separate look at TypeDeclarations. Each operation may be marked by keywords override and abstract which are mapped to the corresponding boolean attributes in the class which represents the Operation in the generated metamodel. Validation rules ensure, that these attributes can not be true at the same time, we will talk about the implementation of such rules in section 8.4.4. In addition, an operation may have a VisibilityModifier, the corresponding attribute is called visibility (typed EString) and can be null if no modifier is present. Other than that, each Operation instance has a name attribute, a list of Parameters which may be empty, and may have a return type. Furthermore, it may have a block of statements, so, unlike in MainProcedures, the reference block has the cardinality 0..1 in case of Operations.

Data Types

The implementation of MuLE types is displayed in the class diagram in figure 37, which represents the part of the language metamodel generated from the grammar excerpt in listing 122.

A large number of language constructs are typed, i.e. they reference a DataType, such as data containers, named operations (all subtypes of Feature) and lambda expressions, but also parameterized types, such as ListType, ReferenceType and DeclaredType. Expressions are also typed, however their type can not be accessed directly via a type or a similar reference and must be inferred, which is covered in the implementation of the type provider in section 8.4.3.

The four elementary types integer, rational, boolean and string are covered by BasicType, which is instantiated each time one of these four keywords is used to specify the type of a typed construct. The used keyword is stored as typeName in the instance of BasicType. Instances of ListType and ReferenceType are created similarly, when the corresponding keyword is used, however as mentioned, they also reference another type, e.g. list<integer> is an instance of ListType which references a BasicType object with the value of the typeName attribute being integer. The OperationType does not represent the return type of an Operation (although it can be certainly used as one), it represents the signature of an operation combining the types of its parameters, their order and the return type.

The DeclaredType is probably the most interesting one, since it is referencing user defined types which are represented by the super type TypeDeclaration. Users can define Enumerations, Compositions and TypeParameters of Compositions. Technically, the grammar allows to declare type parameters in the context of compilation units at the same level as enumerations and compositions, since TypeParameter is a subtype of TypeDeclaration. This is, however, pro-

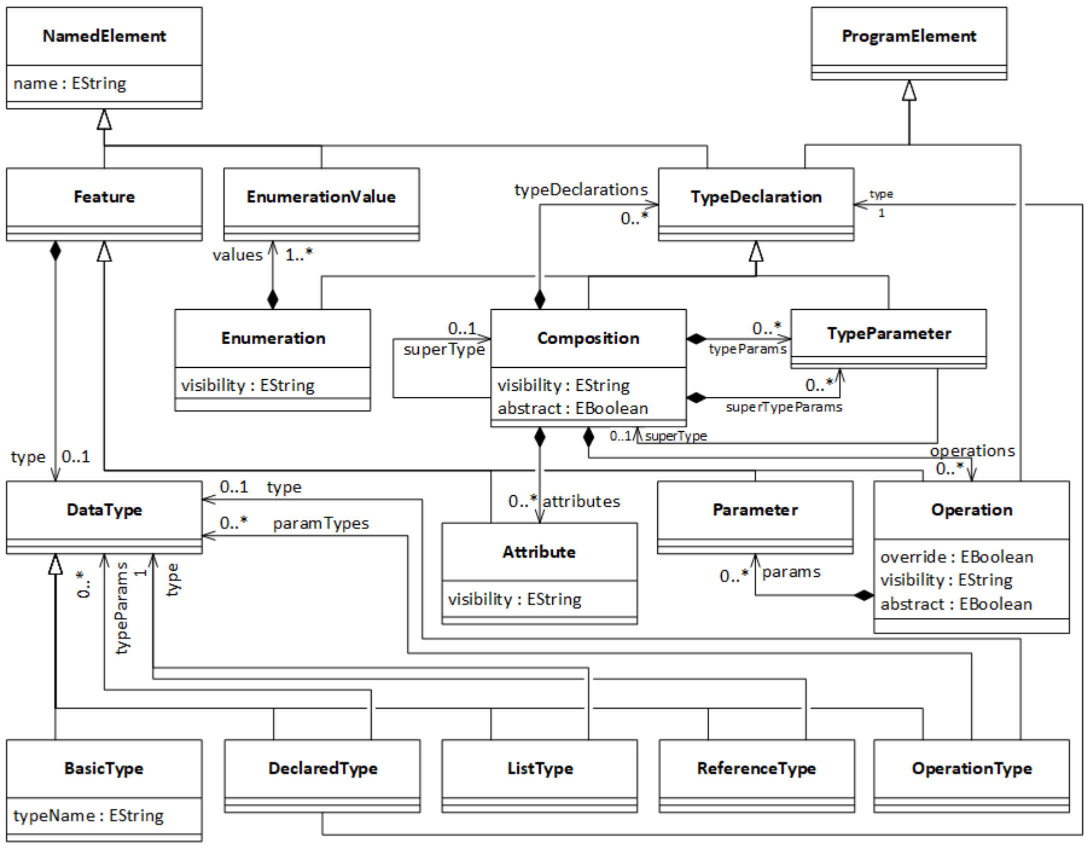


Figure 37: A class diagram depicting the elements of the metamodel related to the type system.

hibited by validations rules. Same applies to type parameters declared within the body of a composition instead of its head, i.e. when it is contained in the typeDeclarations collection instead of typeParams. Both Composition and TypeParameter can specify one other Composition as a its superType, which is then used to implement inheritance relations in the code generation, validation and scoping mechanisms.

It is explicitly stated in the grammar, that declared types can be referenced by their qualified names (lines 4, 28 and 35 in listing 122), which allows to reference imported type declaration. The standard cross-referencing mechanism, i.e. without an explicit specification which kind of identifier should be allowed, relies on simple names.

```
1   DataType: BasicType | DeclaredType | ReferenceType | ListType | OperationType;
2
3   DeclaredType:
4       type=[TypeDeclaration|QualifiedName]
5       ('<' typeParams+=DataType (',' typeParams+=DataType)* '>')?;
```

```
BasicType: typeName=('integer' | 'rational' | 'string' | 'boolean');

ReferenceType: 'reference' '<' type=DataType '>';

ListType: 'list' '<' type=DataType '>';

OperationType:
    {OperationType} 'operation' '(' (paramTypes+=DataType
    (',' paramTypes+=DataType)*)? ')' (':' type=DataType)?;

TypeDeclaration: Composition | Enumeration | TypeParameter;

Enumeration:
    visibility=VisibilityModifier? 'type' name=ID ':' 'enumeration'
    values+=EnumerationValue (',' values+=EnumerationValue)* 'endtype';

EnumerationValue: name=ID;

Composition:
    visibility=VisibilityModifier? abstract?=('abstract')? 'type' name=ID
    ('<' typeParams+=TypeParameter (',' typeParams+=TypeParameter)* '>')?
    ':' 'composition' ('extends' superType=[Composition|QualifiedName]
    ('<' superTypeParams+=TypeParameter
            (',' superTypeParams+=TypeParameter)* '>')? )?
        typeDeclarations+=TypeDeclaration*
        attributes+=Attribute*
        operations+=Operation*
    'endtype';

TypeParameter: name=ID ('extends' superType=[Composition|QualifiedName])?;
```

Listing 122: An excerpt of the MuLE grammar containing the rules defining the type system.

Statements

As mentioned earlier, some constructs, such as operations, the main procedure or some statements, define a `Block`, which is basically a wrapper for a collection of statements gathered under the umbrella of their super type `Statement`. Listing 123 demonstrates an excerpt of the grammar including a selection of statements. To keep it short, the definitions of the `ExitStatement`, the `ReturnStatement` and the `LetStatement` are not included in listing 123.

The statements that include blocks are control flow defining statements `Loop-Statement`, `IfStatement` and `LetStatement`. Since the latter two can define more that one execution path, depending on their conditions, these statements can have more than one block. Therefore, both `IfStatement` and `LetStatement` reference `Block` directly as `block` and an optional `elseBlock`. Furthermore, both statements reference a collection of ElseIfs and ElseLets respectively, each of them defining their own instance of Block. All instances of these four classes define and reference an Expression, which acts as the condition for the execution of their blocks.

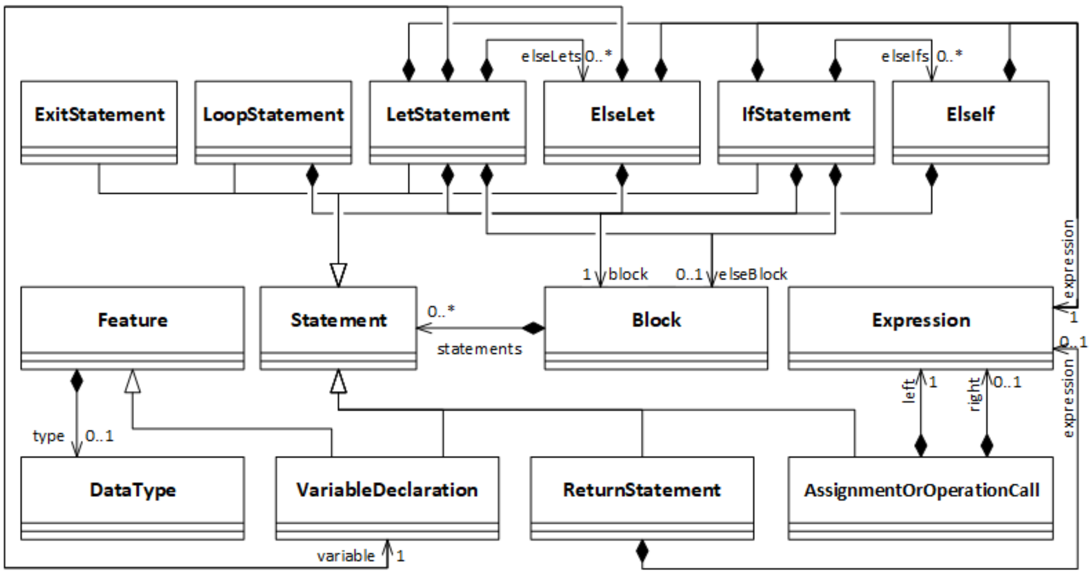


Figure 38: A class diagram representing the elements of the metamodel related to the statements of MuLE.

Another statement that references Expression is ReturnStatement, however in this case the reference may be null if the return statement is empty. However, the most interesting statement is AssignmentOrOperationCall. We have mentioned earlier, that Xtext uses a generated top down parser which has trouble processing left recursive or ambiguous grammars. For example, if we have two separate statement rules OperationInvocation and Assignment, which both start with an identifier, we have a case of ambiguity. Furthermore, operations can be invoked both as an expression (with a return type) and as a statement (without a return type), meaning that there must be rules which would allow both cases, but at the same time not collide with assignment statements or references to data containers in expressions. We have solved it by defining a single rule AssignmentOrOperationCall, which starts with a SymbolReference expression or SuperExpression, both of which can be used to invoke an operation (see next subsequence) and the corresponding types are used to instantiate the respective objects if the statement is in fact an operation invocation. However, as soon as an assignment operator is present after these expressions, the language construct is interpreted as an instance of AssignmentOrOperationCall with a left side, which is an instance of either of the two aforementioned Expressions, and a right side which can be any Expression. Validation checks prevent assignments to operation calls and ensure type conformability in valid assignment statements.

```
Block: {Block} statements+=Statement*;

Statement:
    VariableDeclaration | AssignmentOrOperationCall | IfStatement | LoopStatement |
    LetStatement | ReturnStatement | ExitStatement;

VariableDeclaration: 'variable' name=ID ':' type=DataType;

AssignmentOrOperationCall:
    (SymbolReference | SuperExpression)
    ({AssignmentOrOperationCall.left=current} ':=' right=Expression)?;

LoopStatement:
    {LoopStatement} 'loop' block=Block 'endloop';

IfStatement:
    'if' expression=Expression 'then' thenBlock=Block
    elseIfs+=ElseIf* (=> 'else' elseBlock=Block)? 'endif';

ElseIf: 'elseif' expression=Expression 'then' block=Block;
```
Listing 123: An excerpt of the MuLE grammar containing a selection of statement rules.

Expressions

Since we have already mentioned `SymbolReference`s as a part of the statement rule `AssignmentOrOperationCall`, let us begin by discussing its implementation. Due to the aforementioned concerns about ambiguity, this rule is used to invoke operations, refer to data containers or as a part of the composition value constructor.

Some elements, for example composition attributes, must be referenced by their qualified name. As a reminder, the corresponding rule is defined as `ID ('.' ID)*`, meaning that we cannot use it if the first identifier, for example, refers to a reference to a value with a composition type in which case we have to dereference it first, before we can access the attribute.

Therefore, the expression rule `SymbolReference` is defined recursively, i.e. it references itself as `memberCall` when members of compositions need to be referred. Instances of this rule have a cross reference `symbol` to declared named element. They can reference `SymbolRefCompositionInit` thus resulting in a composition value constructor and an `SymbolRefAccessModifier`, which acts as a super type for `OperationInvocation`, `ListAccess` and `Dereference` and is also defined recursively to allow chaining of access modifiers. Validation rules prevent using access modifiers if the `compositionInit` reference is not `null`.

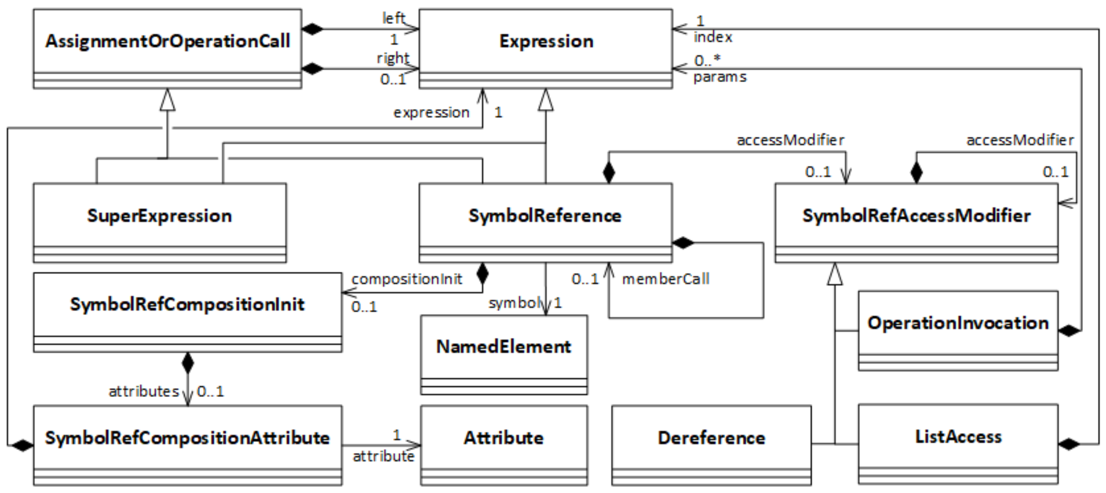


Figure 39: A class diagram representing the elements mentioned in the grammar in listing 122.

```
AssignmentOrOperationCall:
    (SymbolReference | SuperExpression)
    ({AssignmentOrOperationCall.left=current} ':=' right=Expression)?;

AtomicExpression returns Expression:
    SymbolReference | SuperExpression | ... ;

SuperExpression: {SuperExpression} 'super' '.' memberCall=SymbolReference;

SymbolReference:
    symbol=[NamedElement]
    compositionInit=SymbolRefCompositionInit?
    accessModifier=SymbolRefAccessModifier?
    ('.' memberCall=SymbolReference)?;

SymbolRefAccessModifier:
    {OperationInvocation} '(' (params+=Expression (',' params+=Expression)*)? ')'
                            accessModifier=SymbolRefAccessModifier? |
    {ListAccess} '[' index=Expression ']' accessModifier=SymbolRefAccessModifier? |
    {Dereference} '@' accessModifier=SymbolRefAccessModifier? ;

SymbolRefCompositionInit:
    {SymbolRefCompositionInit} '{' (attributes+=SymbolRefCompositionAttribute
        (',' attributes+=SymbolRefCompositionAttribute)*)? '}';

SymbolRefCompositionAttribute: attribute=[Attribute] '=' expression=Expression;
```

Listing 124: An excerpt of the MuLE grammar with expression rules tasked with referring to named elements.

Let us finally take a look at the remaining expressions. By its nature, the expression grammar is recursively defined. The simplest way to define a binary expression would be, for example, as `Expr: Expr Operator Expr;`. However, this would result in a left recursive grammar, which we cannot use with the Xtext parser. Furthermore, we would still need to implement operator precedence. Both of these issues can be solved by left factoring the grammar. As we can see, the first `Expression` rule is `OrExpression`, which is evaluated as `AndExpression`, unless the operator `or` with the corresponding right side is present, similar to the approach we have used to define the statement rule `AssignmentOrOperationCall` earlier.

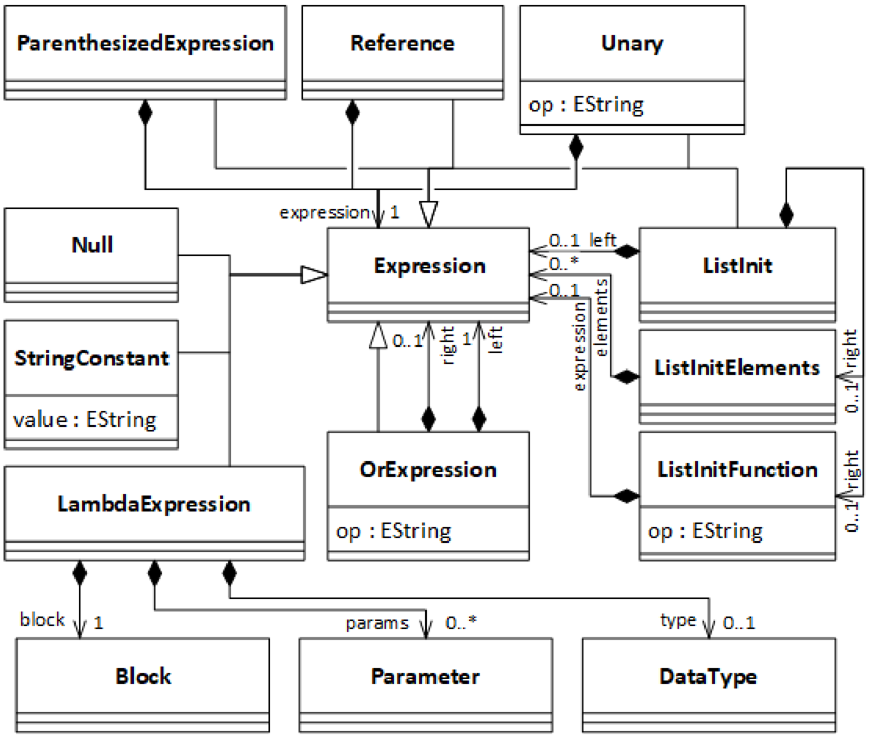


Figure 40: A class diagram representing most of the elements mentioned in the grammar in listing 122.

Same approach is used with the `AndExpression` rule, as well as all other rules which represent binary expressions. This resolves the issue of left recursion, since the parser attempts to resolve the left side as an expression of another type initially. Furthermore, this approach enforces the intended operator precedence, since the left and right sides of the `OrExpression` are typed `AndExpression`, the `AndExpression` is evaluated first resulting in the operator `and` having higher precedence than `or`. For the sake of reducing redundancy, the class diagram in figure 40 does not fully represent this behaviour and depicts only the `OrExpression` class.

```
Expression:
    OrExpression;

OrExpression returns Expression:
    AndExpression ({OrExpression.left=current} op=('or') right=AndExpression)*;

AndExpression returns Expression:
    EqualityExpression ({AndExpression.left=current}
    op=('and') right=EqualityExpression)*;

// other binary expressions

ExponentExpression returns Expression:
    AtomicExpression ({ExponentExpression.left=current}
        op=('exp') right=AtomicExpression)*;

AtomicExpression returns Expression:
    ... | {StringConstant} value=STRING | {Null} 'null'| ListInit |
    LambdaExpression | {Unary} op=('+'|'-'|'not') expression=AtomicExpression |
    {Reference} 'reference' expression=AtomicExpression |
    {ParenthesizedExpression} '(' expression=Expression ')' ;

LambdaExpression returns Expression:
    {LambdaExpression} 'operation'
    '(' (parameters+=Parameter (',' parameters+=Parameter)*)? ')'
    (':' type=DataType)? block=Block 'endoperation';

ListInit:
    {ListInit} "[" (left=Expression
                    right=(ListInitFunction | ListInitElements))? "]";

ListInitFunction:
    {ListInitFunction} op=("**" | "..") expression=Expression;

ListInitElements:
    {ListInitElements} ("," elements+=Expression)*;
```

Listing 125: An excerpt of the MuLE grammar with expression rules tasked with referring to named elements.

The left and right sides of the binary `ExponentExpression` are defined as `AtomicExpression`, i.e. all non-binary expressions that either represent a value literal, a value constructor, unary expressions or the already mentioned `Symbol-References` and `SuperExpressions`. Although a `ParenthesizedExpression` is hardly atomic if we are being precise, we have nevertheless placed it under the same super type for the sake of simplicity. As we can see in the grammar, we can put any `Expression` between parentheses, but can only reference other `AtomicExpressions` in a `Reference` or a `Unary` expression, meaning that the keyword `reference` and unary operators will be applied only to the left side of a binary expression, unless it is put into parentheses.

To define list value constructors, we have used the same approach of resolving ambiguity. The problem is that we have several slightly different notations for list value constructors, i.e. an empty list, a list with values separated by commas, a list defined as a range of integers and a list with a set number of the same value, which all begin with the same symbol. Thus, if the `ListInit` lacks a left side (and therefore the right side as well), it is interpreted as an empty list. If not, the right side can either be a `ListInitFunction`, which represents the last two notations, or `ListInitElements` which references a collection of other `Expression`s each representing a value.

Finally, the implementation of `LambdaExpression` is somewhat similar to that of `Operation`, however it lacks a name, cannot be abstract and has therefore always a block in a valid program. Furthermore it is not a subtype of the class `Feature`, therefore it must reference a `DataType` by itself.

Without validation checks, users may and will do a lot of errors, that are not discovered by the lexer and the parser. Typing errors are a good example here, without proper checks we can declare an integer variable and assign it a string value, which would represent grammatically correct, yet still erroneous code. Similarly, errors related to the scoping mechanism are not discovered by the parser. Once the source code of a MuLE program has been written and parsed, we have access to an AST as an instance of the EMF metamodel presented in the previous subsection. We can now use this model to perform these checks, and if the program is validated, compile it into executable code. The implementation of these checks as well as the code generator will be discussed in subsequent sections.

8.4.2. Scope Provider

The resolution of visible named elements is implemented in the scope provider, which is a generated artefact which relies on a default implementation and can be further customized. The general approach of the scope provider is to gather all named elements that are visible at a certain reference in the source code. The default implementation is not optimal, for example it does not show operations of imported libraries, e.g. `IO.writeLine()` will not work with the default implementation, while the identifier `IO` is correctly resolved if the library is imported, `writeLine` is not visible. A similar problem appears when referring to members of compositions in symbol reference expressions or to composition attributes in composition value constructors. The solution is to customize the scope provider.

The general implementation of the scope provider is demonstrated in listing 126, the method `getScope` returns an instance of `IScope`, which contains a set of all visible elements depending on the parameters `context` and `reference`. The latter is used to identify the referenced node in the AST while the context is used to compute visible elements. In the first case included in listing 126, the `reference` points to a named element as a `symbol` reference of a `SymbolReference` node, the scope is computed by a set of methods called `getVisibleSymbols` based on the `context` and its `container`. For example, if the container of the context is a block of statements, we first need to gather all variables which are declared in this block before the statement which contains the context. Then, we have to add all the named elements, that are declared as a part of the construct which acts as the immediate container of the block, e.g. if it is an operation, we add all parameters. We continue this process by navigating trough the encompassing containers, e.g. if the container is a compilation unit, the visible elements are operations, type declarations, import instructions and finally the compilation unit itself. If the container is a composition, the members of its super type are added to the scope, unless they are marked as private. If the container is another `SymbolReference` and the referenced element is a `Feature`, i.e. it represents a value, we must compute its type by using the type provider, and if it is a composition, gather the elements visible from the context of an instance of that composition, this time also taking the visibility modifier `protected` into consideration. These methods return all visible named elements, regardless whether they are data containers, operations, declared types, etc.

```
class MuLEScopeProvider extends AbstractMuLEScopeProvider {
    @Inject MuLETypeProvider typeProvider

    override IScope getScope(EObject context, EReference reference) {
        var defaultScope = super.getScope(context, reference)
        if (reference == MuLEPackage::eINSTANCE.symbolReference_Symbol) {
            return getVisibleSymbols(context.eContainer, context)
        }
        else if (reference == MuLEPackage::eINSTANCE.declaredType_Type) {
            var outerScope = scopeForDataTypeReferences(context, reference)
            var list = newArrayList
            var contextContainer = context.eContainer
            while (!(contextContainer instanceof CompilationUnit)) {
                if (contextContainer instanceof Composition) {
                    list.addAll(contextContainer.typeParams)
                }
                contextContainer = contextContainer.eContainer
            }
            return Scopes.scopeFor(list, outerScope)
        }
        // other cases
        return defaultScope
    }
}
```

Listing 126: An excerpt of the `MuLEScopeProvider` class.

In cases when we reference a declared type, e.g. as a data type of a variable declaration, we do not need all visible elements which would include other variables, operations, etc. This would mean, that we would have to implement additional checks, whether an element is allowed to be used as a type. Instead, we use another method `scopeForDataTypeReferences`, which computes the scope only for the declared types from the outer context similarly to the approach we have described earlier. If the context is contained in a composition, we add all type parameters of this composition to the scope.

8.4.3. Type Provider

Before we can perform any type checks, we must first determine the types of the respective language constructs. In many cases, we can get the type of the element directly from the corresponding node in the AST, for example a `VariableDeclaration` object has a type reference to a `DataType` object. However, this becomes more difficult as soon as we are confronted with expressions, for example we cannot simply say that the expression `a + b * c` yields a rational value, it could also be an integer, or one of the values might not be a numerical type at all and we have a type error on our hands.

```
class MuLETypeProvider {
    def DataType typeFor(EObject obj, EObject context) {
        switch (obj) {
            BooleanConstant:
                BOOLEAN_TYPE
            OrExpression:
                BOOLEAN_TYPE
            AdditiveExpression: {
                val left = typeFor(obj.left, context)
                val right = typeFor(obj.right, context)
                if (obj.op.equals("&")) {
                    return STRING_TYPE
                } else if (left == RATIONAL_TYPE || right == RATIONAL_TYPE) {
                    return RATIONAL_TYPE
                } else {
                    return INTEGER_TYPE
                }
            }
            LambdaExpression: {
                var paramTypes = newArrayList
                for (param : obj.parameters) {
                    var type = typeFor(param, context)
                    var typeCpy = MuLEObjectCopyProvider.copyMuLEObject(type) as DataType
                    paramTypes.add(typeCpy)
                }
                var returnType = MuLEObjectCopyProvider.copyMuLEObject(obj.type) as DataType
                var opType = MuLEFactory.eINSTANCE.createOperationType()
                opType.type = returnType
                opType.paramTypes.addAll(paramTypes)
                return opType
            }
            // further cases
        }
    }
}
```

Listing 127: An excerpt of the `MuLETypeProvider` class.

The type provider (an excerpt is shown in listing 127) is tasked with computing the type of expressions based on their actual values, provide the expected type for binary expressions, as well as some additional functionality related to providing types for MuLE's language constructs. The type provider is not generated automatically when generating Xtext artefacts from a grammar, however we felt it should be kept separate from the general validation module. The type provider is reused in other modules, for example in the aforementioned validation module or in the generator to compute the type of an expression to generate a type cast (see section 8.4.5). The core of the type provider is the method `typeFor`, which accepts an `EObject` which we want to know the type of, as well as its context, which may be the encompassing statement, if additional information to compute the type is required.

The simplest example is the case of basic type value literals. For example, the literals `true` and `false` are represented by the `BooleanConstant` grammar rule and, therefore, the AST node with the same name. If the `EObject` that was passed to `typeFor` happens to be a `BooleanConstant`, the returned type is an instance of `BasicType` with `typeName` set to `"boolean"`. Same applies to binary expressions which yield a boolean value, such as the `OrExpression`. However, some arithmetical binary expressions may yield different types, depending on the used operator, or the types of the respective operands. For example, if the operator of the `AdditiveExpression` is `&`, then we have a string concatenation, meaning that the expression yields a string value. If the operator is either `+` or `-`, we must check if one or both operands are typed rational, in which case the entire expression is typed as rational. If not, the returned type is integer.

When computing types of value constructors, for example those specified by the `Reference` expression, we compute the type of the referenced expression, wrap it in an instance of `ReferenceType` and return it as a result. The type of the referenced expression is copied, which is necessary due to the EMF rule stating that an object may be contained only in one container at any time. If we would not copy the type of, let's say, a variable declaration, the original element node in the AST would lose its type. The types of `ListInit` expressions and `LambdaExpressions` (shown in listing 127) are computed in a similar way.

The type computation of `SymbolReference` expressions is the most interesting one. The general approach is similar to the control flow that we have already described when discussing the generation of this construct, i.e. we can have a composition value constructor, wherein we would simply create and return an instance of `DeclaredType`, or passing an operation as data, in which case we create an `OperationType` similar to the approach used when typing `LambdaExpressions`, or it can also be an operation invocation or a reference to a data container. Since the `SymbolReference` expression can be simply described as a qualified name with some additions, the type of the entire expression is that of the last referred named element. For example, the type of the expression `point2d.x` would be the type of the attribute `x` which is defined in the composition `Point2D`. We still have to consider access modifiers, let us assume that we have a variable `lst` of type `list<reference<integer>>`, then the expression `lst[0]@` is typed `integer`. The parameter of `typeFor` is in this case a `SymbolReference` pointing to the start of the expression, i.e. the named element `lst`. We first compute the type of `lst`, i.e. `list<reference<integer>>`, then continue to unwrap the type for each present access modifier ultimately resulting in the `BasicType` integer.

```
type Container<T> : composition
   attribute value : reference<T>
endtype

main
   variable c : Container<integer>
   c.value := reference 42
endmain
```

Listing 128: An example used to demonstrate cases related to type parameters in the implementation of the type provider.

Finally, sometimes it is not enough to compute the type simply by navigating to the referred element. Let us take a look at the program in listing 128, we have a composition `Container<T>` which stores an attribute `value` typed `reference<T>`, i.e. a formal type parameter which is later replaced by an actual type parameter, e.g. `Container<integer>`. If the computed type includes formal type parameters, the actual type parameter is inferred from the declared `Feature` object. For example, we declare a variable `c` typed `Container<integer>` and attempt to initialize the stored `value` using the assignment `c.value := reference 42`. The initially computed type of `value` is `reference<T>`, thus we would have a typing error here (expected type is `reference<T>`, actual type is `reference<integer>`). The type provider resolves the formal `T` to actual `integer` from the type of the variable `c`.

8.4.4. Compile Time Validation

Performing additional constraint checks is required to enforce the rules of a programming language, which are not already covered by the grammar or the scoping mechanism, like type checking, naming or checking for cyclic relations. Xtext provides a mechanism to implement validation checks in form of a generated validator class which may include methods with an annotation `@Check`. These methods accept AST nodes as a parameter and are invoked for every object in the AST that matches the type of the parameter. For example, we can implement a check method which accepts instances of `NamedElement` and performs checks related to naming conflicts within. Then, when we write our DSL code, this method will be automatically invoked for every parsed named element in the background, giving immediate feedback if one or more elements fail to validate.

Since we have already introduced the example of named elements, let us first focus on validation rules specified for these elements. Xtext offers an option to use the default name validator, that performs checks for unique names in the same namespace. Since we needed to perform some additional checks and there were cases, where the offered validator was not sufficient, we have implemented our own name validator, which also happens to be the simplest validator which is why we have included the entire class in listing 129.

```
class MuLENamesValidator extends AbstractMuLEValidator {
   @Inject MuLEScopeProvider scopeProvider
   override void register(EValidatorRegistrar registrar) {}

   @Check
   def checkDuplicateNamesForFeatures(NamedElement nElem) {
      var reservedNames = newArrayList
      reservedNames.addAll(ReservedNames.reservedJavaKeywords)
      reservedNames.addAll(ReservedNames.reservedJavaNames)
      reservedNames.addAll(ReservedNames.reservedMuLEKeywords)
      if (reservedNames.contains(nElem.name)) {
         error("Use of this name is not allowed. You are attempting to use a reserved
            word as an identifier.", MuLEPackage.Literals.NAMED_ELEMENT__NAME);
      }
      var container = nElem.eContainer
      if (container !== null) {
         var scope = scopeProvider.getVisibleSymbols(container, nElem)
         for (elem : scope.allElements) {
            var currentElement = (elem as IEObjectDescription).EObjectOrProxy
            if (nElem.name.toString().equals(elem.name.toString())
                     && currentElement !== nElem) {
               error("An element with such name already exists. Use a different name for
                  this element.", MuLEPackage.Literals.NAMED_ELEMENT__NAME)
            }
         }
      }
   }

   @Check
   def checkUnitNameSameAsImportedLibrary(CompilationUnit unit) {
      for (_import : unit.imports) {
         if (unit.name.equals(_import.importedNamespace.name)) {
            error("Naming conflict, the names of the importing and imported compilation
               units must not be equal.", MuLEPackage.Literals.NAMED_ELEMENT__NAME)
         }
      }
   }
}
```

Listing 129: The `MuLENamesValidator` class.

The naming validator relies on the scope provider, which computes visible named elements at specific places in the source code (more in section 8.4.2). The naming validator contains just two check methods. The first one is invoked for every instance of `NamedElement` and its first task is to check if the name of the element is equal to one of the reserved names. The lists of reserved names include MuLE keywords for obvious reasons, as well as Java keywords and other reserved names, like commonly used Java types, to prevent naming conflicts in the generated Java code. If the name of the checked element is included in one of these lists, an error message is displayed in the editor marking the name of the element. Additionally, we have to check if there are no two elements with the same name in the same namespace. For this task, we use the injected scope provider to get all visible elements at the position of our named element. If one of these elements has the same name as our current named element, a corresponding error message is displayed.

The second method checks instances of `CompilationUnit` explicitly. Its task is to ensure, that the import instructions do not state the same name as that of the compilation unit. The purpose is to prevent confusion, for example if the task of the students is to test the `Turtle` library, they might name their program also `Turtle`, which will cause the `import Turtle` instruction to attempt to import the program itself.

In section 8.4.1, we have spoken about the necessity of type checks. These checks are performed by the type validator, which relies heavily on the type provider discussed in section 8.4.3. The type validator works by the same principle as the name validator, i.e. the constraint checking is performed by a set of `@Check` annotated methods. However it also includes utility methods, such as `checkExpectedType`, which is used by most of the type check methods, as well as the methods `isCompatibleType` and `isEqualType`, the first one checks if one type if compatible to another according to the rules defined in section 5.5.12 and the other one checks if two types are equal. The method `checkExpectedType` relies on `isCompatibleType`, and displays an error message at the offending expression if the types are not compatible. Apart from the expected type and the actual type, this method accepts the reference to the source code location in the editor, which is used to accurately display the error message if the types are not compatible, as well as the context, which is used in specific cases. Furthermore, the type provider stores error codes, which are not the actual error message, but are used identify the errors in the quick fix provider (see section 8.5.5).

An excerpt of the type provider is demonstrated in listing 130, we can see the injection of the type provider, an example of an error code and two `checkType` methods, one for statements and one for expressions. Let us start with the validation check of the `OrExpression`, which is a binary expression consisting of a left-hand and a right-hand side. Both sides must contain truth values, i.e. they must be typed as `boolean`. We use the type provider to get the types of both sides and invoke the `checkExpectedType` method to check, whether these types are actually instances of `BasicType` with `typeName` `boolean`.

The other example is that of the `ReturnStatement`, which may or may not have an expression depending on whether the containing operation has a return type or not. What we have to do first, is get the type of the containing operation. Since the immediate container of the return statement must not necessarily be an operation, we must navigate through the containers of the statement until we find one, which may be either an instance of `Operation` or `LambdaExpression`, and then use the type provider to get its return type. Next, we check if the return statement is actually allowed to be empty or not, e.g. if the return type of the operation is not null but the expression reference of the return statement is null, we have an empty return statement in a function and show a corresponding error message. This error message refers to an error code, which is stored in line 5 in listing 130, the quick fix provider can use this error code to generate a placeholder expression. We also have to check the inverse case, i.e. if a non-empty return statement is contained in a procedure. And finally, we must check if the type of the expression in our return statement is compatible to the return type of our operation, or if both are `null`.

```
1   class MuLETypeValidator extends AbstractMuLEValidator {
2      @Inject extension MuLETypeProvider
3      override void register(EValidatorRegistrar registrar) {}
4   
5      public static val ERROR_ILLEGAL_RETURN_NO_VALUE = "IllegalReturnStatementNoValue"
6   
7      @Check
8      def checkType(Statement s) {
9         switch (s) {
10           ReturnStatement: {
11              var opType = null as DataType
12              // get the type using the injected type provider
13              if (opType !== null && s.expression === null)
14                 error("An empty return statement is not allowed in an operation with
15                 a return type.", MuLEPackage.Literals.RETURN_STATEMENT__EXPRESSION,
16                 ERROR_ILLEGAL_RETURN_NO_VALUE)
17              // check for non-empty return statements in operations without a return type
18              else   checkExpectedType(typeFor(s.expression, s), opType,
19                       MuLEPackage.Literals.RETURN_STATEMENT__EXPRESSION, s)
20           }
21           // further cases
22        }
23     }
24  
25     @Check
26     def checkType(Expression s) {
27        switch (s) {
28           OrExpression: {
29              checkExpectedType(typeFor(s.left, s), MuLETypeProvider::BOOLEAN_TYPE,
30                    MuLEPackage.Literals.OR_EXPRESSION__LEFT, s)
31              checkExpectedType(typeFor(s.right, s), MuLETypeProvider::BOOLEAN_TYPE,
32                    MuLEPackage.Literals.OR_EXPRESSION__RIGHT, s)
33           }
34           // further cases
35        }
36     }
37     // further check methods, utility methods and error codes
38  }
```

Listing 130: An excerpt of the `MuLETypeValidator` class.

Finally, a lot of validation checks have nothing to do with naming or the type system, for example checking for cyclic inheritance, if a program unit has a main procedure, or if the integer literal starts with a zero. These rules are combined in the `MuLEValidator` class, which also acts as the main validator, i.e. it references both previously discussed validators. In other aspects, it is implemented in a similar way as what we have already discussed.

8.4.5. Code Generation to Java

Once the AST has been successfully validated we can use it to generate Java code, which can then be executed on the Java virtual machine, which allows to use MuLE on any operating system that supports Java. The Xtext generator is integrated in the Eclipse building infrastructure [110], i.e. it is executed automatically each time the source file is changed and saved.

Mapping of MuLE Constructs to Java Constructs

Before we discuss the implementation of the generator, let us take a look at the mapping of MuLE's language constructs to the corresponding Java constructs, thus demonstrating which artefacts are generated from the MuLE source code.

Each MuLE compilation unit is mapped to a separate Java class, which has the same name as the compilation unit. Import instructions are mapped directly to Java import statements. MuLE enumerations and compositions are mapped to nested static enums and classes respectively. Operations are generated as static methods of the base class. The main procedure is mapped to the main method.

MuLE type	Java type	Generated default value
integer	Integer	new Integer(0)
rational	Double	new Double(0.0)
string	String	new String("")
boolean	Boolean	new Boolean(false)
list<T>	ArrayList<T>	new ArrayList<>()
reference<T>	MuLEReferenceType<T>	null
enumeration	enum	EnumName.FirstLiteral
composition	class	new ClassName()
operation<...>	Function<T,R> Supplier<R>	A lambda expression with default functionality based on its signature.

Table 3: MuLE types with the corresponding Java types and the generated default values.

Let us now take a look at the mapping of MuLE types to Java types, which is summarized in table 3. We use `Integer` and `Double` to represent `integer` and `rational` respectively, as well as `Boolean` for MuLE's `boolean`. This allows us to use these types as type parameters of parameterized types such as `list` (`java.util.ArrayList`) and `reference` (`MuLEReferenceType`). The `MuLEReferenceType` contains a single field `value`, which is accessed when a reference value is dereferenced. As already mentioned earlier, type declarations are generated as nested types in the classes either representing the compilation unit or the containing composition.

The operation type is generated either as a `Supplier` or as `Function`. The type `Supplier` is used to represent anonymous operations with neither parameters nor a return type (the generated supplier returns `null`, but can be only be invoked as a procedure), or those with a return type but still lacking any parameters. Although there are other Java types that could be more suitable in some cases, e.g. `Consumer`, we have decided against using them to keep the number of cases in the generator at the minimum. The type `Function` is used to represent anonymous operations with at least one parameter but without a return type (similarly to supplier, the generated function returns null and can be only invoked as a procedure), or when both parameters and the return type are present. When more than one parameter is present, we generate curried lambda expressions to simulate anonymous operations with multiple parameters.

MuLE statements are generated the their Java counterparts as follows:

- `VariableDeclaration` – a variable declaration with the generated Java type and a generated standard value, the corresponding mapping will be given below.
- `AssignmentOrOperationInvocation` – if the statement is an operation invocation, i.e. its type is either `SymbolReference` of `SuperExpression`, the corresponding expression generation methods are invoked. Otherwise, both left and right sides are generated as expressions. The invocation of the utility method `copyObject` (section 8.4.7) is generated around the right side in order to enforce the value copying semantics of MuLE.
- `LoopStatement` – is generated as a `while(true)` loop.
- `IfStatement` – as described in listing 133, it is generated simply as a Java `if`-statement.
- `LetStatement` – is generated as a Java `if`-statement where the condition of the generated `if`-statement is an `instanceof` check against the type of the variable declared in the head of the `let`-statement. This variable is generated as the first statement of the generated `if`-statement.
- `ExitStatement` – a Java `break` statement.
- `ReturnStatement` – a Java `return` statement with or without the return value.

Finally, the expressions are mostly mapped to the corresponding Java expressions. MuLE binary expressions are generated as Java binary expressions with the corresponding operator, same applies to most of the unary expressions. For example, let us take a look at the binary `MultiplicativeExpression`, if the operator is `mod`, the generated operator is `%`, if the MuLE operator is `div`, the generated operator is `/` which acts as integer division in Java as long as integers are used on both sides (validations checks are implemented to enforce this in MuLE), and if the MuLE operator is `/`, then both sides are cast to `double` simulating rational division. Some expressions lack an appropriate direct counterpart in Java. These expressions include:

- `EqualityExpression` – invocation of the `Util.MuLEEquals` method (see section 8.4.7), which is additionally negated if the used MuLE operator is `/=`.
- `ExponentialExpression` – invocation of the `java.lang.Math.pow` method.
- `ListInit` – generates a new `ArrayList`, depending on the used MuLE list value, an invocation of dedicated utility methods (see section 8.4.7) may be necessary.
- `Reference` – generates a new `MuLEReferenceType` with the value of referenced expression copied (using `Util.copyObject`, see section 8.4.7) and passed as the constructor parameter.
- `LambdaExpression` – by default, Java does not support lambda expressions with more than one parameter. As previously mentioned, MuLE lambda expressions with multiple parameters are mapped to curried lambda expressions in Java.

The Generator Stub

The generator stub is one of the artefacts created from the grammar, however unlike the lexer, parser and the metamodel, the generator stub must be implemented manually to be of any actual use. The generator class is written in Xtend, which we have also used to generate strings for most single line language constructs, e.g. expressions, while Xpand templates are used to generate multi-line language constructs, such as the compilation unit itself, compositions, operations or block-based statements. A newly created generator class initially overrides the inherited method `doGenerate`, which accepts an EMF `Resource` object, which is basically our AST, a `IFileSystemAccess2` with framework related file access operations (creation, deletion, path management, etc.) and the also framework related `IGeneratorContext` object, which tracks the cancellation indicator.

```
class MuLEGenerator extends AbstractGenerator {
   @TracedAccessors(MuLEFactory)
   static class MuLETraceExtensions {}
   @Inject extension MuLETraceExtensions
   @Inject extension MuLETypeProvider

   override void doGenerate(Resource resource, IFileSystemAccess2 fsa,
   IGeneratorContext context) {
      val program = resource.contents.head as CompilationUnit
      if (program !== null) {
         val programName = resource.URI.lastSegment.nameWithoutExtension
         generateTracedFile(fsa, "generated/" + programName + ".java", program,
         '''package generated;
            import mule.util.*;
            // further imports of required Java libraries
            «FOR i : program.imports»
               «var n = i.importedNamespace.name»
               «IF #["IO", "Mathematics", "Strings", "Lists"].contains(n)»
                  import mule.lang.«n»;
               «ELSEIF n.equals("Turtle")»
                  import mule.turtle.Turtle;
                  import mule.turtle.Turtle.*;
                  // further imports of MuLE standard libraries
               «ELSE»
                  import generated.«n».*;
               «ENDIF»
            «ENDFOR»

            public class «programName» {
               «FOR element : program.programElements»
                  «IF element instanceof TypeDeclaration»
                     «generateTypeDeclaration(element)»
                  «ELSEIF element instanceof Operation»
                     public static «generateOperation(element)»
                  «ENDIF»
               «ENDFOR»

               «IF program.main !== null»
                  public static void main(String[] args){
                  «FOR s : program.main.block.statements»
                     «generateStatement(s)»
                  «ENDFOR»
                  }
               «ENDIF»
         } ''')
      }
   }  // further generator methods
}
```

Listing 131: The core of the generator class.

The default approach when using the Xtext code generator is to call the `IFileSystemAccess2` `generateFile` method in the overridden method `doGenerate` passing it the file name and the generated text, which can be written using an Xpand template. However, one of the requirements for the language was debugging support, thus we had to make slight adjustments to the generator by reusing the tracing mechanism which was initially implemented for Xtend, which is seen in lines 2-4 in the shortened source code of the generator class in listing 131.

Although the topic of this section is code generation and we have a dedicated section for the implementation of the debugging support (section 8.5.1), the generator lays the groundwork for the debugging support. What we also see in line 5, is the injection of the `MuLETypeProvider`, which is discussed in section 8.4.3. The injection mechanism allows to easily specify a dependency directly and thus use the functionality of the injected component. The internal mechanisms of the framework take over the task of instantiating objects for stated dependencies.

Let us take a look at the overridden `doGenerate` method. Instead of calling the `generateFile` method of the `IFileSystemAccess2` object, we invoke the method `generateTracedFile` which we have got by injecting the `MuLETraceExtensions`. We still need to pass the `IFileSystemAccess2` object to it as a parameter, along with the path of the generated file, the root of the AST (`program`), as well as the content of the generated file, which is defined by an Xpand template.

Generation of Compilation Units and their Elements

In the template, we see textual content coloured blue which is generated as it is, as well as Xpand constructs which we use to iterate over specific nodes of our AST to generate the corresponding strings which depend the contents of the AST. For example, every generated class requires a specific set of imports, e.g. the contents of the `mule.util` package which is distributed with MuLE and contains, among others, utility operations, such as the implementation of the MuLE value copying semantics (see section 8.4.7), meaning that this import is always generated. However, not every library is required, both standard and user defined libraries are manually imported in MuLE compilation units, meaning that the import instructions are present as nodes in our AST. We iterate over these nodes and generate the corresponding strings depending on whether it is a standard library or one defined by the user. The name of the program itself is also the name of the generated Java file (thus, also the name of the generated class), the difference is in the path. While the MuLE files are located in the source folder *src*, the generated files are located in the package *generated* in the source folder *src-gen*. Since we are using the `generateTracedFile`, in addition to the Java file, a hidden trace file is generated in the same folder with tracing information between the original MuLE file and the generated Java file, which can be used to implement debugging support (more in section 8.5.1). This allows us to simply generate the import instructions of user defined libraries using the name of the library with the prefix `import generated.`, however, this also means that as soon as the users make changes to the generated files (which they should not do), the generated code will stop working.

In a similar fashion, we iterate over `ProgramElement` nodes of our AST to generate type declarations and operations, if there are any, and ultimately generate the main procedure, again if it is present in our source compilation unit. Types are generated as static nested types in the generated class while operations are generated as static methods, unless they are member operations of compositions. The generated visibility modifier of all Java elements is always `public`, regardless of the actual visibility modifier of the MuLE element. This results in a simpler generator code, while the visibility checks performed by the scope provider (section 8.4.2 ensure that the scoping rules (section 5.1) are enforced in the MuLE source code.

```
@Traced def private generateOperation(Operation op) {
    '''
    «op.generateOperationSignature»«IF op.abstract»;«ELSE»{
        «FOR s : op.block.statements»
            «s.generateStatement»
        «ENDFOR»
    }
    «ENDIF»
    '''
}

def private generateOperationSignature(Operation op) {
    var n = ""
    if (op.abstract)         n += "abstract "
    if (op.type === null)    n += "void"
    else                     n += generateType(op.type)
    n += " " + op.name + "("
    for (p : op.params) {
        n += generateType(p.type) + " " + p.name
        if (!op.params.get(op.params.size() - 1).equals(p))
        n += ", "
    }
    n += ")"
    return n;
}
```

Listing 132: Methods tasked with generating operations.

We will use operations to demonstrate further generator functionality. The general approach depicted in listing 132 is to generate the method signature, and if it is not abstract, the method body too. To generate the method body, we must iterate over all statements of the corresponding block and generate them too, which ultimately requires generating expressions in most cases. This approach applies more or less to the generation of type declarations and the main method too, thus we will not look into them. The `generateOperation` method is invoked from within the core Xpand template in listing 131 and provides another template tasked with generating Java code from `Operation` nodes in the AST. This method is marked by the annotation `@Traced`, meaning that it is used as a part of the traced code generation procedure initiated with the invocation of the `generateTracedFile` method. As mentioned earlier, in addition to the generated Java code we get a trace model in a separate file. The method `generateOperationSignature`, which is written in pure Xtend is meant to simplify the template in `generateOperation`, it provides a simple string based on the information stored in the passed `Operation` object.

Generation of Statements

Statements are generated in a similar way (see listing 133), however, the annotation `@Traced` additionally specifies, that the value `useForDebugging` is set to true. This means that the nodes that representing statements in the trace model are intended to be used for the purpose of debugging, i.e. the program will stop at the line of code represented by this node if a breakpoint is placed at this line.

```
@Traced(useForDebugging=true) def private generateStatement(Statement s) {
    switch s {
        VariableDeclaration: { '''«generateVariableDeclaration(s)»;''' }
        IfStatement: { '''«generateIfStatement(s)»''' }
        // further statements
    }
}

def private String generateVariableDeclaration(VariableDeclaration varDec) {
    var type = varDec.type
    var staticType = generateType(type)
    var result = staticType + " " + varDec.name
                + " " + generateInitialValueForType(varDec.type)
    return result
}

@Traced def private String generateIfStatement(IfStatement c) {
    '''
    if(«generateExpression(c.expression)»){
        «FOR s : c.block.statements»
            «generateStatement(s)»
        «ENDFOR»
    }
    «FOR elif : c.elseIfs»
    else if («generateExpression(elif.expression)»){
        «FOR s : elif.block.statements»
            «generateStatement(s)»
        «ENDFOR»
    }
    «ENDFOR»
    «IF c.elseBlock !== null»
    else {
        «FOR s : c.elseBlock.statements»
            «generateStatement(s)»
        «ENDFOR»
    }
    «ENDIF»
    '''
}
```

Listing 133: Methods tasked with the generation of statements.

This does not mean that we can now place breakpoints in our MuLE file, the actual logic for breakpoint placement in MuLE files is discussed in section 8.5.1. The method `generateStatement` checks the actual type of the passed `Statement` object and executes the corresponding generation method.

The variable declaration is a single line statement, we use pure Xtend to generate this statement. The type of a variable as well as its initial value are generated by separate methods. The `if`-statement is a multiline construct, therefore the corresponding method is providing another Xpand template. The MuLE `if`-statement is generated simply as a Java `if`-statement, the conditional expression is generated by its own method. Since the `if`-statements may have several paths defined by additional `ElseIf` objects and an optional `elseBlock`, we have to check whether they are present, generate the corresponding Java code representing these constructs as well call `generateStatement` for each statement within their respective blocks. Single line statements are generated with a trailing semicolon, the multiline statements are not.

Generation of Expressions

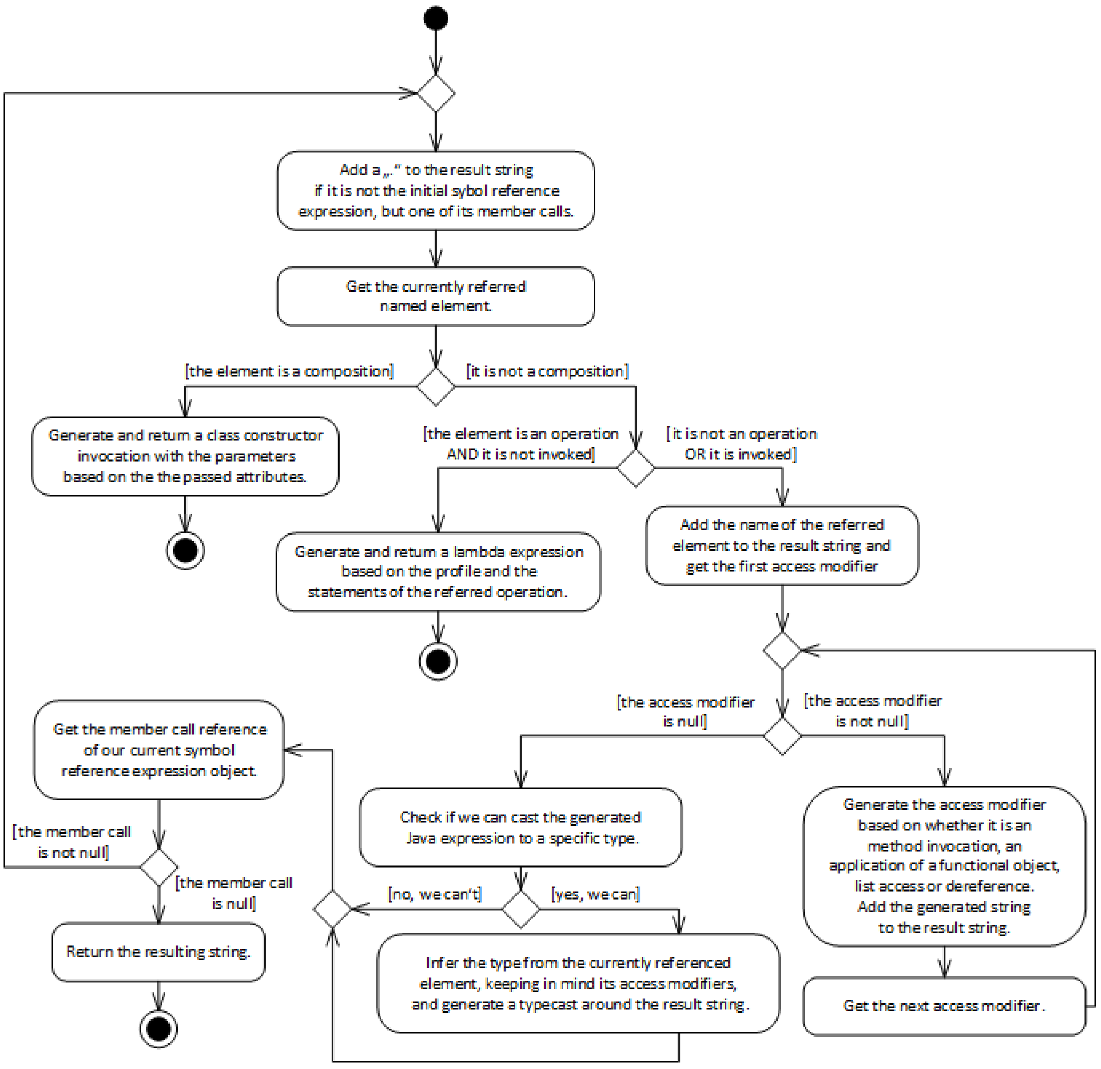
The expressions range from single literals to complex constructs like lambda expressions, we have selected a few examples for various cases (listing 134). Literals, like the `IntegerConstant`, are the simplest expressions, we must simply return the value stored in the AST node. String literals are a bit more complex, we have to watch for escaped characters during the generation process. In case of the `ListInit` expression, depending on the used notation we either generate an empty `ArrayList`, an `ArrayList` filled with elements, wherein each element is an expression which also must be generated, or invoke the list creation functions of the utility class (section 8.4.7).

Binary expressions, like the `OrExpression` consist of two expressions and a corresponding operator. Some expression rules allow several different operators, e.g. the `MultiplicativeExpression`, which must be considered in the corresponding cases.

The aforementioned lambda expressions, represented by instances of `Lambda-Expression`, are a special case of multiline expression. However, since all other expressions are generated as single strings and the entire process is invoked from within the statement generator, which expects string values when generating expressions, we do not use a template in this case.

The issue with the templates is that when using the `@Traced` mode, the templates are actually typed as `GeneratorNode` instead of `String`. The generation of statements expects strings when invoking the expression generator, which means that we would need to convert the result back to string if we would have used a template losing any tracing information in the process. Therefore, we have decided to generate these expressions without a traced template. This also means, that it is not possible to place breakpoints within lambda expressions.

The generation of the `SymbolReference` expression is probably the most complex of all expressions, the implemented method is too large to be rationally placed here, thus its heavily abstracted behaviour is depicted in a diagram in figure 41. As mentioned in section 8.4.1, the instances of `SymbolReference` are used to refer to named elements in expressions, when necessary, via its qualified name. Since the named element may be a list variable, a composition declaration or an operation, we have to distinguish between these cases. When a composition is referenced, we generate a Java constructor call based on whether the composition value constructor has attributes or not. Any declared composition is generated as a Java class with two constructors, one that is empty and another one with all fields including those inherited from its super types.



Add a „.“ to the result string if it is not the initial sybol reference expression, but one of its member calls.
Get the currently referred named element.
[the element is a composition]
[it is not a composition]
Generate and return a class constructor invocation with the parameters based on the the passed attributes.
[the element is an operation AND it is not invoked]
[it is not an operation OR it is invoked]
Generate and return a lambda expression based on the profile and the statements of the referred operation.
Add the name of the referred element to the result string and get the first access modifier
[the access modifier is null]
[the access modifier is not null]
Get the member call reference of our current symbol reference expression object.
Check if we can cast the generated Java expression to a specific type.
Generate the access modifier based on whether it is an method invocation, an application of a functional object, list access or dereference. Add the generated string to the result string.
[the member call is not null]
[the member call is null]
[no, we can't]
[yes, we can]
Return the resulting string.
Infer the type from the currently referenced element, keeping in mind its access modifiers, and generate a typecast around the result string.
Get the next access modifier.

Figure 41: An activity diagram depicting the behaviour of the `generateSymbol-Reference` method.

```
def private String generateExpression(Expression e) {
    switch e {
        IntegerConstant: return e.value
        ListInit: return generateListInit(e)
        SymbolReference: return generateSymbolReference(e)
        OrExpression:
        return generateExpression(e.left) + " || " + generateExpression(e.right)
        LambdaExpression: {
            var result = ""
            if (e.parameters.empty) result += "()" + " -> "
            for (Parameter p : e.parameters)
            result += p.name + " -> "
            result += "{\n"
                for (Statement s : e.block.statements) {
                    result += traceNodeToString(generateStatement(s)
                    as CompositeGeneratorNode) + "\n"
                }
                if (e.type === null) result += "\treturn null;\n"
                result += "}"
            return result }}
}

def private String generateListInit(Expression expr) {
    var result = ""
    if (expr instanceof ListInit) {
        var right = (expr as ListInit).right
        if (right === null) { result = "new ArrayList(Arrays.asList())" }
        else {
            switch right {
                ListInitElements: {
                    result = "new ArrayList(Arrays.asList("
                    if ((expr as ListInit).left !== null)
                        result += (expr as ListInit).left.generateExpression
                    for (e : right.elements)
                        result += "," + e.generateListInit
                    result += "))" }
                ListInitFunction: {
                    if (right.op.equals("**")) {
                        var rightType = typeFor(right.expression, null)
                        result = "mule.util.Util.fillListRepetition("
                            + (expr as ListInit).left.generateExpression
                            + ", " + right.expression.generateExpression
                            + ", \"" + copyObjectHelper(rightType, expr) + "\")"
                    } else if (right.op.equals("..")) {
                        result = "mule.util.Util.fillListRange("
                            + (expr as ListInit).left.generateExpression
                            + "," + right.expression.generateExpression + ")" }}}}}
    else { result += expr.generateExpression }
    return result
}
```

Listing 134: A selection of rules tasked with the generation of expressions.

If the currently referred element is an operation and the expression lacks an access modifier, meaning that the operation is not invoked in the source code, we generate a lambda expression which copies the functionality of the referred operation. This allows us, for example, to pass named operations as parameters to other operations, with some exceptions: it is not possible to pass named operations which are members of compositions or are imported from a library, since they could reference outer context which could be not available.

However, if the referred element is neither a composition nor an operation, or it is an operation with an operation invocation access modifier, we add the name of the referred element to the result and generate all access modifiers depending on their type, i.e. an operation invocation with actual parameters, a `.get(INDEX)` in case of a list access or a `.value` if it is a dereference.

The next step is to check if we can cast the generated expression to its type. Why do we need to do that? For example, let us assume we have two variables `a : list<reference<A>>` and `b : list<reference<B>>`, with `B` being a subtype of `A`, the assignment `a := b` is valid in MuLE. However, if we generate the Java types for these variables directly as they are, i.e. `ArrayList<MuLEReferenceType<A>>` and same for `B`, we can no longer perform this assignment in Java. Since lists are reference types in Java, this assignment would create two aliases with different type parameters of the same list object, allowing to store instances of `A` in the same list in which we can except to access all features defined in `B`. This is not a problem in MuLE, since the list is copied. The corresponding compilation errors in Java are circumvented by avoiding the generation of the type parameter of the `MuLEReferenceType` and casting of the generated expressions instead. We generate a type cast as long as it can be done according to Java rules, i.e. it is not a procedure invocation, not a super expression and the referenced element is not the left side of an assignment (its parent symbol references are valid for casting though).

Example of Code Generation

Let us take a look at a simple example of generated code, we have two small compilation units `demoProgram` and `demoLibrary`. The generated Java code can be seen in listings 135 and 136 respectively. It should be noted, that we have shortened the examples of the generated code considerably, e.g. by removing empty lines, unnecessary imports, comments, etc. Furthermore, the generated code is generally not well formatted, and is thus not easy to read compared to hand written code. If we would make these changes to the generated code in our project, we would at very least break debugging support, which depends on line information which was actual at the time of code generation.

As we see, both Java classes have the same names as the original MuLE units, both are located in the package generated. Note that the import of the `demoLibrary` unit is generated as `import generated.demoLibrary.*;`, while the import of the standard library `IO` has a different qualified name. The import of the `util` package is generated automatically. Line 8 in listing 135 shows a generated variable declaration with the corresponding default value. We can also see examples of generated code for pretty much every statement, except for the `let`-statement. As we can see, when a value is passed to a data container, the `copyObject` method is invoked. Similarly, the equality is checked by the `MuLEEquals` utility method (more in section 8.4.7).

```
program demoProgram
import demoLibrary
main
    variable i : integer
    loop
        if i = 5 then
            exit
        endif
        demoLibrary.printIntLine(i)
        i := i + 1
    endloop
endmain
```

```
library demoLibrary
import IO

operation printIntLine(parameter i : integer)
    IO.writeInteger(i)
    IO.writeLine()
endoperation
```

```
package generated;

import mule.util.*;
import generated.demoLibrary.*;

public class demoProgram {
    public static void main(String[] args){
        Integer i = new Integer(0);
        while(true){
            if(mule.util.Util.MuLEEquals(((Integer)i), 5)){
                break;
            }
            demoLibrary.printIntLine(mule.util.Util.copyObject(((Integer)i), "Integer"));
            i = mule.util.Util.copyObject(((Integer)i) + 1, "Integer");
        }
    }
}
```

Listing 135: The shortened generated code of the `demoProgram` compilation unit.

```
package generated;

import mule.util.*;
import mule.lang.IO;

public class demoLibrary {
    public static void printIntLine(Integer i){
        IO.writeInteger(mule.util.Util.copyObject(((Integer)i), "Integer"));
        IO.writeLine();
    }
}
```

Listing 136: The shortened generated code of the `demoLibrary` compilation unit.

8.4.6. Implementation of Standard Libraries

The standard libraries are implemented using Java. For each standard library, a `.mule` file, which acts as an interface to the Java implementation, is provided and can be imported by the user. The `.mule` file is basically a MuLE library unit with type declarations and operations, the latter are not implemented and may even lack a return statement. The library units representing standard libraries are parsed, but not validated. This means that when importing them, we get an AST and can refer to their elements, but we get no errors if an operation with a return type lacks a return statement in this unit. As we have said, they are meant to be used as interfaces for the actual Java implementations, which are imported in the generated code. This approach allows to implement functionality, which is otherwise not possible with the constructs offered by MuLE.

A short example of this approach is demonstrated in listings 137 and 138. The former contains an excerpt of the standard library `IO` (section 6.1) defined by its importable MuLE unit, while the latter demonstrates its actual Java implementation. As we can see, the operations in the MuLE unit do not specify any functionality, moreover, the library would not even compile if we would write it on our own since the operation `readString` lacks a return statement. However, if we import this library in our program, the parser will be able to create an AST from it, meaning that we can then invoke the operations regardless whether they are implemented or not. The generator (see section 8.4.5) will generate the instruction `import mule.lang.IO;` in the Java code, resulting in the invocation of Java implemented method.

```
library IO

operation writeString(parameter arg : string) endoperation

operation readString() : string endoperation
```

Listing 137: An excerpt of the `IO` MuLE library unit.

```
package mule.lang;
public class IO {
    static Scanner sc = new Scanner(System.in);

    public static void writeString(Object arg) {
        System.out.print(arg);
    }

    public static String readString() {
        String result = sc.next();
        sc.nextLine();
        return result;
    }
}
```

Listing 138: An excerpt of the Java implementation of the `IO` library.

As we have mentioned in chapter 6, some of the standard libraries were implemented by students of the University of Bayreuth, i.e. `Turtle`, `UBTMicroworld`, and `GUIFactory`. They have provided the Java implementation of these libraries, which was later integrated in MuLE using the approach described above. The implementation of the Turtle library was designed from the very beginning with MuLE in mind resulting in a very simple integration process, i.e. the implementation already included a single Java interface, for which we had to provide a corresponding MuLE interface. This was not the case with the other two libraries resulting in a more complex integration process. Among other actions, we had to wrap the return types of the factory methods as `MuLEReferenceType`s to make it compatible to the type system of MuLE. On the other hand, this served as a proof, that even more complex Java libraries can be integrated into MuLE, using the interface and the adapter patterns if necessary.

8.4.7. Utility Package

When discussing the implementation of various modules, we were constantly confronted with the utility methods, such as the `copyObject` method. As we have seen in the implementation of the generator, the contents of the package `mule.util` are imported into any generated Java class. This package contains the classes `MuLEReferenceType`, `Util`, and `MuLEObjectCopyProvider`.

The class `MuLEReferenceType` serves as the wrapper type for reference types. In MuLE, we can wrap any type as a reference type and any expression as a reference expression, which creates an instance of `MuLEReferenceType` with the expression providing its referenced value. The `toString` method of this class provides the string representation of the values of this type, that are seen in the debugger or when printed on the console.

The class `MuLEObjectCopyProvider` creates copies of AST nodes, when simply passing the reference to a node would break the AST since an EMF object may only be contained within one container at any time. For example, if we pass a named operation as data, we create a lambda expression with the same semantics. Each parameter, the return type as well as each statement of the original operation is copied using this class, otherwise the AST node representing the source operation would be emptied of its structural features afterwards.

And finally, we have the `Util` class, which provides utility methods mostly invoked in generated Java classes. For instance, this class provides the methods which implement MuLE's equality and value copying semantics (see section 5.5.13). The implementation of the equality semantics for most of the types is fairly simple, e.g. if both values are of numerical but different types, they are converted to `Double` and the results are compared. However, if we have two compositions, we have to rely on reflection to decide, which object is instance of the subtype of another in order to invoke the correct `equals` method. For example, we have two objects `p2d` (`Point2D{x = 2, y = 3}`) and `p3d` (`Point3D{x = 2,` y = 3, z = 5}), with Point3D being a subclass of Point2D. If the classes were not related, we would return false in the MuLEEquals method. However, since they are, we need to invoke the generated equals method. Since these are value types, we can only compare the shared attributes. The generated equals method for each composition contains an instance check against the current type as well as comparisons of all owned and inherited attributes, i.e. invoking equals on p2d would result in a instanceof Point2D check and subsequent comparison of the respective values of x and y, meaning that p2d.equals(p3d) would result in true while p3d.equals(p2d) in false. Since we can write both p2d = p3d and p3d = p2d in MuLE and expect true in each case, the MuLEEquals method checks which way the comparison should be actually performed and returns the result.

A similar issue has to be handled in the copyObject method, which implements the value copying semantics. If the copied object is null, a reference, a value of a basic type or a function, we simply return this object. If it is a list, we create a new list and copy the entries of the source list recursively. However, if we have a composition, we use reflection to create an instance of the target class, the class name is passed as string to the method and then used to search for the class in the workspace, and then iterate over the declared fields of the target class to copy values of these fields (again recursively) into the target object, thus ensuring that only shared attributes are copied. For example, let us take a look at the assignment p2d := p3d. The source object is p3d with three attributes, while the target class Point2D specifies only two. We first use the type provider to get the type of the left side of the assignment, which is Point2D, then generate the corresponding qualified name, e.g. CompilationUnitName$Point2D, and then invoke copyObject with p3d and the qualified name of the target class as parameters. Inside of the method, we locate the target class by its qualified name and create an instance of it. We continue by iterating over the fields declared in class Point2D, apart from sone internal fields there are x and y, which are then searched in the source object and the values are copied (by invoking copyObject with the corresponding parameters) into the new instance of Point2D.

8.5. UI Modules

The focus of this section lies on the implementation of the user interface modules contained in the de.ubt.ai1.mule.ui plugin. These modules provide tool support within the Eclipse IDE and are thus less relevant should MuLE be ported to another IDE one day. The general approach for most of the modules is similar to what we have already seen in the previous section, however, in addition to the AST we will also rely on the parse tree for some of the UI modules.

8.5.1. Launch Shortcuts and Debug Support

Initially, the DSL created with Xtext acts as a source from which runnable code, e.g. Java, is generated. The users were then meant to execute the generated Java code directly, i.e. although the program was written using the DSL, the users were still confronted directly with the generated code. This was not acceptable in case of MuLE, we wanted to create a simple tool for education by reducing the overall overhead, not increasing it. Therefore, we had to implement shortcuts which would allow users to execute MuLE files directly, without navigating to the generated Java files first.

To achieve this, we took inspiration from Xtext projects which relied on the Xbase grammar, as we have mentioned in section 8.4.1 this is an expression grammar which, when used as a basis for the implemented DSL, also automatically provides debug support, among other artefacts. By reverse engineering (with the help of Xtext developers) the generated Xtext infrastructure as well as the general launch process of Eclipse projects, we were able to provide launch shortcuts and debug support for our own language without using the Xbase grammar.

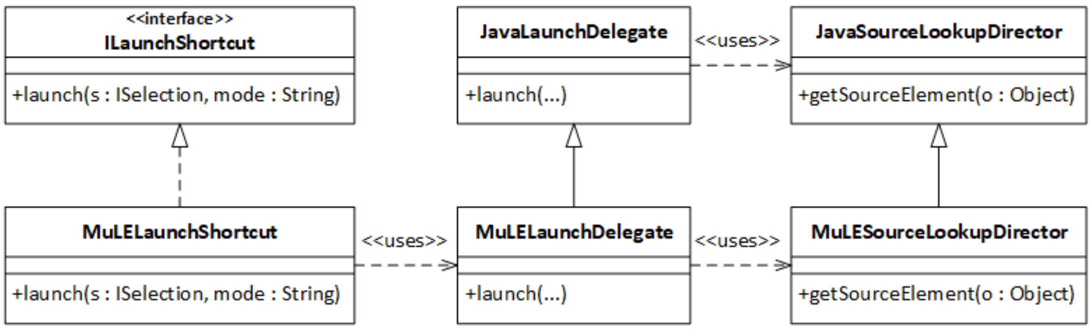


Figure 42: A simplified diagram demonstrating the relations of the classes responsible for the execution of MuLE files.

By using plug-in extensions as well as Xtext components registration mechanism, we have registered two launch modes, *run* and *debug*, both of which rely on the class `MuLELaunchShortcut` which implements the Eclipse interface `ILaunchShortcut` responsible for launching the selected file or the contents of the active editor in the workbench (see figure 42). In the implemented `launch` method, we can get the project name and, ultimately, the path to the generated Java file from the editor input, i.e. the MuLE file. From this data we create the launch configuration and pass it along to the launching procedure of Eclipse. Additionally, we had to redefine the `JavaSourceLookupDirector` (as `MuLESourceLookupDirector`), which is responsible for the location of the MuLE source file at a later stage, when the correct line in the debugging process has to be displayed. We also had to provide our own extension of the `JavaLaunchDelegate` as `MuLELauchDelegate`, which uses the `MuLESourceLookupDirector` instead of its Java variant.

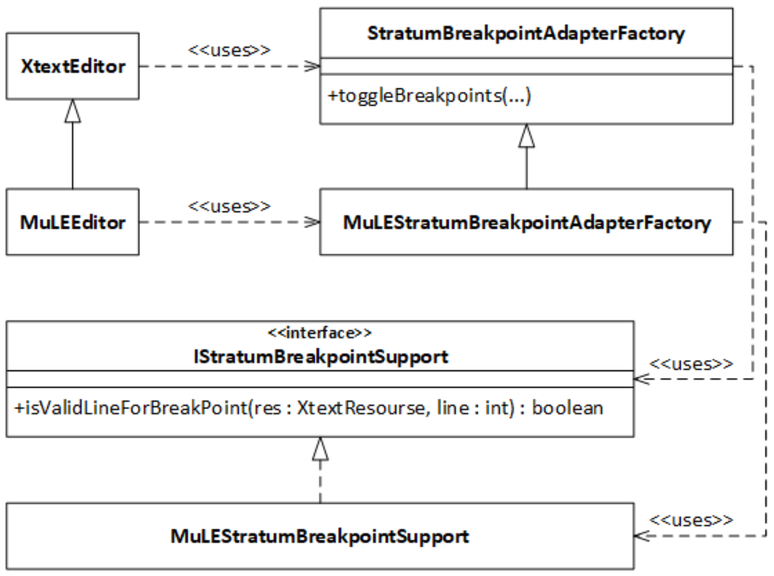
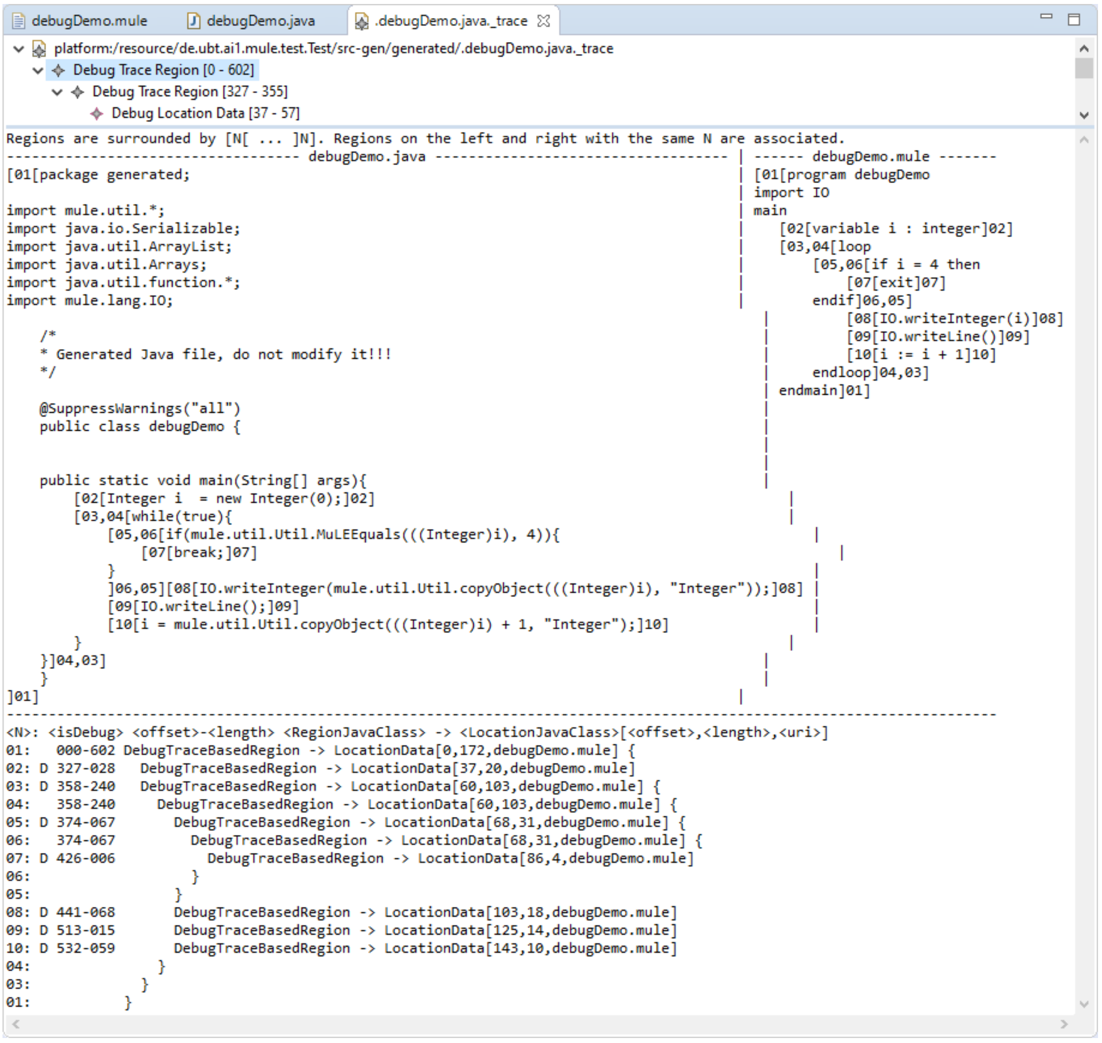

XtextEditor
<<uses>>
StratumBreakpointAdapterFactory
+toggleBreakpoints(...)
MuLEEditor
<<uses>>
MuLEStratumBreakpointAdapterFactory
<<interface>>
IStratumBreakpointSupport
+isValidLineForBreakPoint(res : XtextResourse, line : int) : boolean
<<uses>>
MuLEStratumBreakpointSupport
<<uses>>

Figure 43: A simplified diagram demonstrating the relations of the classes responsible for the placement of breakpoints within MuLE files.

The launch shortcut alone is not sufficient to actually enable the creation of breakpoints. To implement debug support for Xtend and Xbase based DSLs, the developers of Xtext have provided the class `StratumBreakpointAdapterFactory`, which links the logic related to the creation, removal and toggling of breakpoints to the `XtextEditor`. In order to fully provide debug support for MuLE, we had to specify and register our own text editor class `MuLEEditor` by extending the default `XtextEditor` as well as an extension of `StratumBreakpointAdapterFactory` (see figure 43). We register our own `MuLEStratumBreakpointAdapterFactory` as the adapter for `MuLEEditor`. The breakpoint adapter factory uses an implementation of `IStratumBreakpointSupport` which provides the logic for decision making, whether the selected line is valid for placing a breakpoint, or not. In our implementation of the method `isValidLineForBreakpoint`, we iterate through the parse tree to locate the text region which corresponds to the passed line. Then, we get the corresponding AST node and return true, if the node is an instance of one of the state altering single line statements, i.e. a variable declaration, an assignment, an invocation of a procedure, or a return statement, and the line in the text region is equal to the line parameter. If the AST node is a block, a block based statement, a composition or a compilation unit, we invoke `isValidLineForBreakpoint` again to search within these constructs for statements, where we can place a breakpoint. In other cases we return false signalizing that we can not place a breakpoint in this line. Finally, we have to register the double click action and a pop-up action on the ruler in our text editor using the Eclipse plug-in extensions.

In the implementation of the generator (section 8.4.5), we have used a trace



```
platform:/resource/de.ubt.ai1.mule.test.Test/src-gen/generated/.debugDemo.java._trace
  Debug Trace Region [0 - 602]
    Debug Trace Region [327 - 355]
      Debug Location Data [37 - 57]
```

```
Regions are surrounded by [N[ ... ]N]. Regions on the left and right with the same N are associated.
----------------------------------- debugDemo.java ----------------------------------- | ------ debugDemo.mule -------
[01[package generated;                                                                  | [01[program debugDemo
                                                                                        | import IO
import mule.util.*;                                                                     | main
import java.io.Serializable;                                                            |    [02[variable i : integer]02]
import java.util.ArrayList;                                                             |    [03,04[loop
import java.util.Arrays;                                                                |        [05,06[if i = 4 then
import java.util.function.*;                                                            |            [07[exit]07]
import mule.lang.IO;                                                                    |        endif]06,05]
                                                                                           |         [08[IO.writeInteger(i)]08]
    /*                                                                                     |         [09[IO.writeLine()]09]
    * Generated Java file, do not modify it!!!                                             |         [10[i := i + 1]10]
    */                                                                                     |     endloop]04,03]
                                                                                           | endmain]01]
    @SuppressWarnings("all")                                                               |
    public class debugDemo {                                                               |
                                                                                           |
                                                                                           |
    public static void main(String[] args){                                                |
        [02[Integer i  = new Integer(0);]02]                                                  |
        [03,04[while(true){                                                                   |
            [05,06[if(mule.util.Util.MuLEEquals(((Integer)i), 4)){                               |
                [07[break;]07]                                                                      |
            }                                                                                    |
            ]06,05][08[IO.writeInteger(mule.util.Util.copyObject(((Integer)i), "Integer"));]08] |
            [09[IO.writeLine();]09]                                                              |
            [10[i = mule.util.Util.copyObject(((Integer)i) + 1, "Integer");]10]                  |
        }                                                                                     |
    }]04,03]                                                                               |
    }                                                                                      |
]01]                                                                                    |
------------------------------------------------------------------------------------------------------------------------
<N>: <isDebug> <offset>-<length> <RegionJavaClass> -> <LocationJavaClass>[<offset>,<length>,<uri>]
01:   000-602 DebugTraceBasedRegion -> LocationData[0,172,debugDemo.mule] {
02: D 327-028   DebugTraceBasedRegion -> LocationData[37,20,debugDemo.mule]
03: D 358-240   DebugTraceBasedRegion -> LocationData[60,103,debugDemo.mule] {
04:   358-240     DebugTraceBasedRegion -> LocationData[60,103,debugDemo.mule] {
05: D 374-067       DebugTraceBasedRegion -> LocationData[68,31,debugDemo.mule] {
06:   374-067         DebugTraceBasedRegion -> LocationData[68,31,debugDemo.mule] {
07: D 426-006           DebugTraceBasedRegion -> LocationData[86,4,debugDemo.mule]
06:                   }
05:                 }
08: D 441-068       DebugTraceBasedRegion -> LocationData[103,18,debugDemo.mule]
09: D 513-015       DebugTraceBasedRegion -> LocationData[125,14,debugDemo.mule]
10: D 532-059       DebugTraceBasedRegion -> LocationData[143,10,debugDemo.mule]
04:               }
03:             }
01:           }
```

Figure 44: Content of the hidden trace file generated along the executable Java file.

based generator, which, as we have said, generates an additional trace model based on the parse tree, which is then used to link regions in the source code of the DSL file to the corresponding regions in the generated file. Let us take a look at such a trace file demonstrated in figure 44, which was generated from a simple program. In the screenshot we see both the trace model (shortened in the screenshot) and the textual representation of the currently selected node. In the textual representation, we see both the source MuLE file on the right, the generated Java file on the left and the textual representation of the trace regions at the bottom. The trace model consists of a tree of debug trace regions, we have selected the root node which represents the entire program unit. The trace regions are enumerated, i.e. we can quickly identify which regions are linked, e.g. the variable declaration is marked as region 02 in both the source and the target file. Furthermore, if we look at the textual representation of the regions at the bottom of the screenshot, we see that this region is additionally marked as `D`, which means that it can be used for the purpose of debugging. The marker `D` is placed on regions that were generated using the generator annotation `@Traced(useForDebugging=true)`, meaning that if we place a breakpoint at this line, it will be actually registered and the program will halt. As a reminder, only the statements are generated using this annotation and only specific statements can return true in the method `isValidLineForBreakpoint`. This means, that we can place a breakpoint in the line with the variable declaration, when debugging, the trace model is checked whether we can actually use this breakpoint, and if it is the case, the execution of the Java program will be halted and the corresponding line will be displayed in the MuLE editor by tracing the line from the Java file to the MuLE file.

In a nutshell, this allows us to place breakpoints in our MuLE text editor as well as execute MuLE files both in *run* and *debug* mode. In both cases, we select the MuLE file and the underlying implementation locates the corresponding generated Java file and executes it. In case of the *run* mode, the program is executed normally and all breakpoints are ignored. When launched in debug mode, the tracing model is used to display the correct line in the MuLE editor based on the line in the executed Java program. The process is as follows, we place a breakpoint in the MuLE editor, then execute the program in debug mode. The Java code is executed with the framework keeping track between the executed Java code and the MuLE source code and halting the program at the appropriate Java line at the same time displaying the corresponding MuLE line in the editor. We do not need to provide an implementation for the actual halting of the executed Java code, this is taken over by the Xtext framework and Eclipse itself.

8.5.2. Syntax Highlighting

The default syntax highlighting already does a decent job, e.g. it marks keywords and string literals by distinct colours while everything else remains black. Additionally, we wanted to mark references to data types in our source code by a different colour. To achieve that, we have extended the `DefaultHighlightingConfiguration` and implemented the interface `ISemanticHighlightingCalculator`. The class `MuLEHighlightingConfiguration` defines and registers the text style (orange and bold) for data type references, while `MuLESemanticHighlightingCalculator` iterates over the parse tree and if the grammar element in the current node is either a `DataType` reference, which, for example, occurs in a variable declaration, or the identifier of a type declaration, we get the location and the length of the textual region and use the previously defined custom text style to mark this region. All other textual nodes are covered by the default implementation. Since both of these classes are not automatically generated artefacts, we had to manually register them in the specific class reserved exactly for this task.

8.5.3. Outline Tree Provider

The outline tree (section 7.1) offers a quick overview over the source code. The default implementation is insufficient, it generates nodes for every element in the AST including expressions or keywords resulting a cluttered tree. Moreover, since some of these elements lack specific information required during the creation of the outline tree, e.g. an identifier, a lot of the nodes are generated as `<unnamed>`, resulting in a clearly unfinished look.

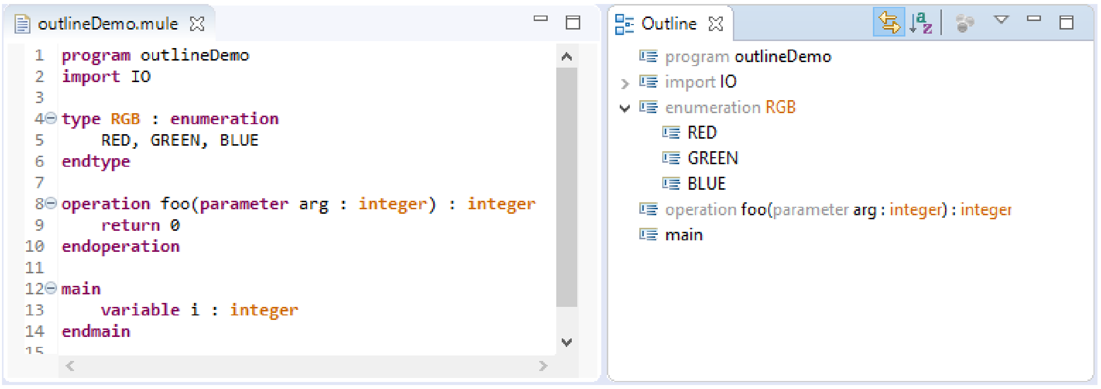
To alleviate this, Xtext automatically generates an outline provider stub, which must be customized by the developer of a DSL in order to implement a proper outline tree. An excerpt of the implemented stub is displayed in listing 139. First, we have to define which elements of our outline tree are leaves, i.e. they will have no further children. For example, we have specified that nodes created for instances of compilation units are leaves, which might sound like a problem at first, after all, we want to create an entire tree for our compilation unit. Other leaves are for example the main procedure or all operations.

```
class MuLEOutlineTreeProvider extends DefaultOutlineTreeProvider {
    @Inject extension MuLETypeProvider
    @Inject StylerFactory stylerFactory

    def _isLeaf(CompilationUnit a) { true }
    // further leaf definitions, e.g. MainProcedure, Operation, etc.

    def void _createChildren(DocumentRootNode outlineNode, CompilationUnit program) {
        createNode(outlineNode, program)
        for (importStatement : program.imports) {
            createNode(outlineNode, importStatement)
            // Create nodes for visible content of the imported library.
        }
        for (programElement : program.programElements)
            createNode(outlineNode, programElement)
        if (program.main !== null)
            createNode(outlineNode, program.main)
    }

    def Object _text(CompilationUnit program) {
        var result = new StyledString()
        if (program.isLibrary)
            result.append("library ",
                stylerFactory.createXtextStyleAdapterStyler(getNodeTypeTextStyle()))
        else
            result.append("program ",
                stylerFactory.createXtextStyleAdapterStyler(getNodeTypeTextStyle()))
        if (program.name !== null)
            result.append(program.name)
        return result
    }
    // further _text methods, as well as methods defining text styles
}
```

Listing 139: An excerpt of the `MuLEOutlineTreeProvider` class.

If we look at the method _createChildren, we notice why making a compilation unit a leaf is not an issue, but rather a necessity. This method creates our outline tree by invoking the inherited method createNode. The created nodes are then added to the passed root node. Another parameter is an instance of our compilation unit, for which we instantly create a node. The root node is not actually displayed in the outline view, creating a leaf for our compilation unit creates a corresponding element in the view as the very first entry, so that we can actually see the type and the identifier of our compilation unit in the tree. The strings, that are displayed in the view, are provided by the _text methods. For example, for our compilation unit, we append the string library or program as a grey styled string and the normally styled name of the program, and return the result.

We then create the nodes for import instructions, which are not leaves, i.e. we can unwrap them to see the imported elements. Same applies to type declarations, meaning that we can see literals of enumerations and members of compositions. Operations are, however, leaves similar to the main procedure. This also explains, why the CompilationUnit entry must be leaf, otherwise, we would have the created children not only in the immediate tree, but also for every compilation unit node resulting in duplicated entries in our tree. We use orange styled strings for data type references and identifiers of type declarations.



```
program outlineDemo
import IO

type RGB : enumeration
    RED, GREEN, BLUE
endtype

operation foo(parameter arg : integer) : integer
    return 0
endoperation

main
    variable i : integer
endmain
```

```
program outlineDemo
import IO
enumeration RGB
    RED
    GREEN
    BLUE
operation foo(parameter arg : integer) : integer
main
```

Figure 45: An example of a source program and the resulting outline tree.

Figure 45 demonstrates an example of an outline tree created from a simple program. As we can see, the first child of our tree is a program outlineDemo element, recognizable as a leaf since we cannot expand it. Again, if we could, we would get the same content that is already displayed here. The next child is the import IO node, we can expand it to get an overview over imported operations. The identifier of the node representing the type declaration RGB is coloured orange, its literals are visible if we expand the node. The children representing the operation and the main procedure are both leaves, we cannot see their statements in the outline tree. However, we still can see the parameter of the operation, the parameters are included in the styled strings which represent the operations in the tree.

8.5.4. Proposal Provider

The proposal provider generates content proposals when the content assist function of the IDE is invoked, e.g. by pressing the *Ctrl+Space* key combination. The default implementation displays allowed keywords as well as visible identifiers. However, we wanted to provide more information when displaying the identifiers. For example, instead of a simple operation name, which is the case when using the default implementation, we wanted the proposal to be marked with the keyword operation followed by its name and the parameter profile.

Conveniently, Xtext generates a proposal provider class stub which extends the default implementation. We have redefined the implementation of the method tasked with creation of proposals for symbol references. We use the injected scope provider to get all visible elements at the current position of our cursor, and then create proposals for each visible element. The proposals consist of two string values, a label, which is displayed in the content assist pop-up window, as well as the generated proposal string in the source code once one of the labels was selected. The approach of creation of these strings is similar to that used in the generator, i.e. create a string based on the information stored in the AST node. The injected type provider is used to get the types of typed elements.

For example, if we invoke the content assist in the main procedure of the program in figure 45, one of the proposed elements will be the operation `foo`, the corresponding label in the content assist view is `operation foo(parameter arg : integer) : integer`, while the generated proposal is `foo(___ARG__INTEGER___)`.

8.5.5. Quickfix Provider

During the discussion of the implementation of the validator in section 8.4.4, we have encountered error codes, which were passed when displaying an error message. We have also mentioned, that these error codes are used in the implementation of the quick fix provider, an excerpt of which is displayed in listing 140.

Generally, when one or more quick fixes are implemented for a validation error, which is identified by its error code, Eclipse will show available quick fixes when hovering over the location of the error with the mouse. For example, if a return statement is contained in an operation with a return type, but lacks an expression, we will get an error message and the program will not compile. This particular error message is identified by the error code `ERROR_ILLEGAL_RETURN_NO_VALUE`, the implemented quick fix appends a placeholder string to the return statement and replaces the erroneous text fragment in the source code with the new string. Although it does not fully solve the problem, the placeholder must also be replaced by a meaningful expression, this should still help programmer novices to solve the problem. Finally, as already mentioned in sections 7.1 and 8.4.4, it is not possible to implement useful quick fixes for all validation errors.

```
class MuLEQuickfixProvider extends DefaultQuickfixProvider {
   @Fix(MuLETypeValidator.ERROR_ILLEGAL_RETURN_NO_VALUE)
   def returnAddValue(Issue issue, IssueResolutionAcceptor acceptor) {
      acceptor.accept(issue, 'Add a value placeholder to the return statement.',
         'Add a value placeholder to the return statement.', null) [ element, context |
            val xtextDocument = context.xtextDocument
            val issueString = xtextDocument.get(issue.offset, issue.length)
            val newString = issueString + " __INSERT_VALUE_HERE__"
            xtextDocument.replace(issue.offset, issue.length, newString)
      ]
   }
   // further methods
}
```

Listing 140: An excerpt of the `MuLEQuickfixProvider` class.

8.5.6. Automatic Formatting

By default, the formatter stub is not generated, although, there is an option to activate its generation. Once generated, the auto formatter must be implemented, because otherwise invoking the auto formatting process will format the source code into a single line, which is probably the reason why the stub is not generated by default.

```
class MuLEFormatter extends AbstractFormatter2 {
   @Inject extension MuLEGrammarAccess
   override protected void _format(XtextResource resource, IFormattableDocument document) {
      if (resource.errors.size > 0) return
      else super._format(resource, document)
   }

   def dispatch void format(CompilationUnit program,
                            extension IFormattableDocument document) {
      append(prepend(regionFor(program).keyword('library'), [noSpace]), [oneSpace])
      append(prepend(regionFor(program).keyword('program'), [noSpace]), [oneSpace])
      for (importStatement : program.imports) {
         prepend(importStatement, [newLine])
         format(importStatement, document)
      }
      for (programElement : program.programElements) {
         format(programElement, document)
         prepend(programElement, [newLines = 2])
      }
      prepend(program.main, [newLines = 2])
      append(program.main, [newLines = 3])
      format(program.main, document)
   }
   // further methods
}
```

Listing 141: An excerpt of the `MuLEFormatter` class.

The general approach is to manipulate the parse tree directly, for example by replacing the currently present white spaces around certain tokens with a specified number of space characters, indentations or line breaks. An example is demonstrated in listing 141, the injected `MuLEGrammarAccess` is used to access the parse tree. If compilation errors are present in the source code, the formatting process is not initiated. Otherwise, format methods are invoked for various nodes in our AST. For example, in the case of a compilation unit node, we ensure that there are no white spaces before the keyword `program`, if one is present in the corresponding text region in the parse tree, and exactly one space character after it. Same applies to the keyword `library`. Furthermore, every `import` instruction is preceded by exactly one line break, and additionally formatted by its own `format` method. We apply a similar pattern to all other elements of the compilation unit, with a varying amount of line breaks in between. The methods `append` and `prepend` do not add the specified number of white spaces to the already present ones, but replace them instead.

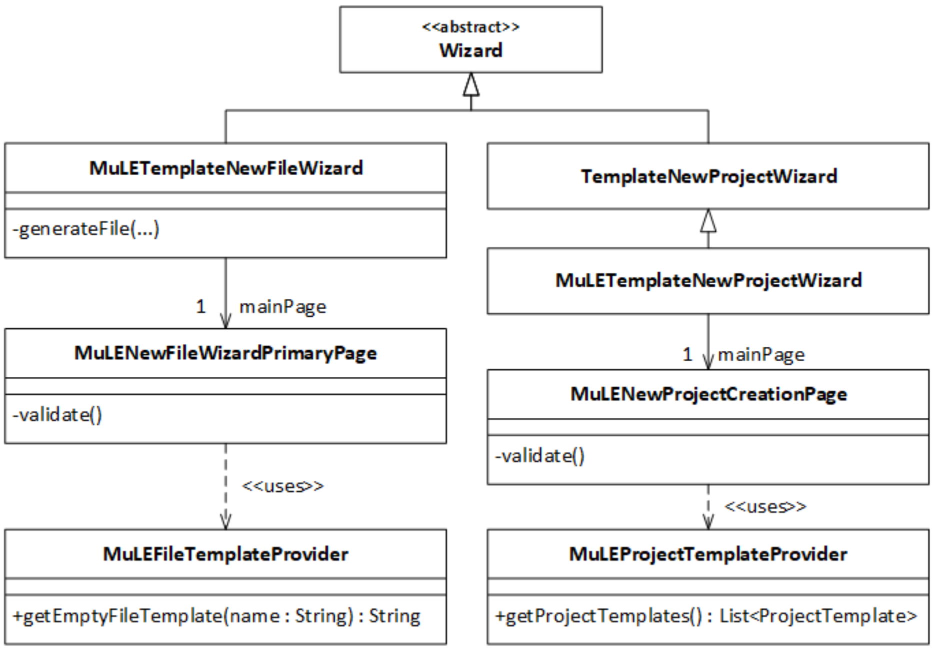
8.5.7. Project and File Creation Wizards

Both the file and the project creation wizards are implemented as subtypes of the abstract class `Wizard`, which acts as the standard base for wizards in Eclipse. At some point, Xtext introduced template provider class stubs, which simplified the implementation of wizards for Xtext DSLs. However, at this point we had our own implementation, which was by far not optimal. Furthermore, we had encountered issues when using the new Xtext approach. As a result, we ended up with a mix of both our own implementation and a large part of the current Xtext approach, which is based around specifying templates for files and projects in the template provider and validation rules in the creation page (see figure 46). Everything else is taken over by the framework.

```
class MuLEFileTemplateProvider implements IFileTemplateProvider {
   def static String getEmptyFileTemplate(String name) {
      '''
      program «name»
      import IO

      main
      endmain
      '''
   }
}
```

Listing 142: The `MuLEFileTemplateProvider` class.

This approach worked well for the project creation wizard, but, as we have already mentioned, we had issues getting it to work with the file creation wizard. As a solution, we have created a workaround by copying the functionality of the default `TemplateNewFileWizard` into our `MuLETemplateNewFileWizard` and altering it at key places. For example, by adding a `generateFile` method, which creates a new file at the specified path with a standard template provided by the method `getEmptyFileTemplate` of the class `MuLEFileTemplateProvider` which is displayed in listing 142.



<<abstract>>
Wizard
MuLETemplateNewFileWizard
-generateFile(...)
1
mainPage
MuLENewFileWizardPrimaryPage
-validate()
<<uses>>
MuLEFileTemplateProvider
+getEmptyFileTemplate(name : String) : String
TemplateNewProjectWizard
MuLETemplateNewProjectWizard
1
mainPage
MuLENewProjectCreationPage
-validate()
<<uses>>
MuLEProjectTemplateProvider
+getProjectTemplates() : List<ProjectTemplate>

Figure 46: A heavily abstracted diagram demonstrating the relations of the classes responsible for the project and file creation wizards.

Although the project template provider in figure 46 states the possibility of several templates, we have implemented only one. Since an eclipse project is a bit more complex than a mere text file, in addition to the project name and its path, we have to specify the project natures (Java, Xtext, Plugin), the builders (Java and Xtext), the dependency on the MuLE plugin, the source folders (*src* and *src-gen*), etc. Furthermore, as mentioned in chapter 7, each MuLE project contains a help folder with text files filled with examples and short documentation of the standard libraries. These help files are generated using templates similar to the one displayed in listing 142.

8.6. Testing Procedure

Initially, we have used the *JUnit* framework to test the implementation during the development. However, since we have also used a runtime Eclipse environment to test the plug-ins directly by implementing example programs, we have quickly noticed that we need more time to maintain both the unit tests and the example programs at the same time. Since the latter variant produced better testing results in a large picture, especially when testing the execution semantics of specific language constructs, we have decided to omit unit testing entirely. Building all test projects after introducing some changes to the language provided a similar experience to the automated tests created with JUnit, i.e. if something is wrong, either we will see it as a compile time error in the MuLE code or in the generated Java code, unless it is a very specific case, which we would usually notice when executing the program. This has also demonstrated another advantage of this approach, we have noticed far more errors compared to when using unit tests, which required to design a test case for a very specific scenario.

For the sake of testing, we have created thematically oriented projects. For example, one of the projects would contain programs tasked with testing purely procedural concepts, e.g. variables, input/output, control flow, etc., while others would concentrate on object-oriented or functional language constructs. Specific projects were tasked with testing the more complex standard libraries, such as `GUIFactory` or `UBTMicroworld`. Nevertheless, the actual deployment of the language in an introductory course (chapter 9) was extremely helpful to find additional bugs, although, we were relieved that no absolutely breaking bugs were encountered during the course. The errors were mostly caused by students new to the world of programming trying to write programs in a way, that was not meant to be done but neither were thought of as possible by the developers of this language. For this, we are very grateful to our students, who had contributed to the process of improvement of MuLE, and had to have the patience when doing so.

8.7. Conclusion

As we have seen in this chapter, Xtext provides a very generous set of tools to create a programming language, even offering various options, which approach to choose. Without heavily relying on the tools provided by this framework and reusing existing infrastructure, it would not be possible to implement a fully functional programming language and provide it with tool support within the context of a single PhD thesis. Even with these tools, the developers of a new programming language must prepare themselves to invest a lot of time into the process.

The current implementation is meant to be used with Eclipse, however, it is possible to create an independent execution environment. With a few additional steps [110], the generator can be invoked from the command line terminal which would allow to use it completely separated from Eclipse. This would however also mean, that the entire implementation of the supported tools discussed in section 8.5 cannot be used.

9. Evaluation and Related Work

Over the course of this thesis, we have already given an ample amount of example programs written with MuLE, thus this chapter will not focus on the demonstration of this language in action. Instead, we will mostly discuss its practical application in a programming course, which acts as a preliminary course prior to the start of the first semester with the intention to give students a basic understanding of programming concepts. Furthermore, we discuss students' feedback that we were able to gather over several iterations of this course as well as their performance immediately at the end of the course and at the end of the semester. Additionally, this chapter includes a comparison to other programming languages used in the context of education.

9.1. Practical Application and Students' Feedback

As mentioned in the introduction of this thesis, MuLE was initially developed as a second part of a two step plan meant to facilitate the entry into programming education at the University of Bayreuth. The first part of this plan actually consisted of a preliminary programming course, which is conducted shortly before the start of the CS1 lecture at our university. The language of choice is Java in this lecture, our initial plan was to use a simpler language such as Python [59] in the preliminary course as a medium used to convey the programming concepts in a simpler manner and thus to prepare the students for the lecture. As soon as MuLE was mature enough to be used for this task, we have started to use it instead of Python in our course, which has also helped us to test the language with the target audience, gather feedback and improve the language. Ultimately, MuLE is supposed to be used in the CS1 lecture as a replacement for Java. By now, it supports all the necessary concepts, however, we have not been able to use it for this task yet.

9.1.1. Preliminary Programming Course

Initially, the course took place in a computer lab with room for roughly 40 participants. Since we had around 90 participants at the start of each year, we had to separate them into two equally sized groups. This in turn meant that we could not separate them based on their background knowledge, since we had assumed that we would have more participants without prior programming experience. This turned out to be mostly correct, almost each year around two thirds of the participants had either very basic programming experience or none at all. However, this has changed since the start of the Covid-19 pandemic and the subsequent transition to education via online live conferences. In the last two iterations of this course, we could separate the students based on their background knowledge, since we were no longer limited by the available physical space.

This allowed us to tailor the educational approach and the speed of presentation specifically to these distinct groups, for example, by limiting the time spent on the trivial introductory topics and discussing assignments only if necessary in case of the experienced group.

However, the online conference approach has a significant drawback. The course was initially designed around the *active learning* approach [117], wherein the students are encouraged to actively participate in the process of education. Simply described, an educational unit consists of a new programming concept, which is usually introduced by a real world algorithmic scenario or a previously known problem, a corresponding new language construct, an example program and, finally, one or two assignments centered around the new concept as well as those introduced earlier. The scenario used to introduce the new concept acts as an analogy which is meant to create a mental model among participants and facilitate the transfer of the newly introduced concept [118].

During all of these phases, we encourage active discussions with the students, and especially in the last part, the assignment, the students are encouraged to work in pairs to discuss the solution. Students with prior programming knowledge, who have no difficulties solving these assignments in short time, were encouraged to take a role of a tutor by helping other students. During this phase, the instructor would observe the process and provide help when needed.

This has been transformed more or less back into frontal lecture styled lessons since the start of the pandemic, since the students are apparently much less willing to actively participate in such activities (those that are possible at all) in an online conference. Thus the resulting approach was turned into discussing a real world scenario, followed by the introduction of the related programming concept and implementation of an example program. The corresponding student assignments were either solved together (beginner group) or by the students separately from the course time at their own leisure (advanced group).

Another core approach that we have chosen when designing our course was *problem-based learning* [119], which can be easily combined with the aforementioned active learning approach. During the assignment phase, the students were given a task which was formulated in a rather abstract way, i.e. in most cases students were given no concrete algorithmic steps which they merely had to translate into a program. This means that first they had to find an algorithmic solution and then implement it as a program, a very difficult task for beginner programmers, which was somewhat alleviated by working in pairs and with the aforementioned tutoring by experienced students and the instructor [120]. However, this way of working is not viable in an online conference, most of the students had shown no inclination to work together on a solution or give any signs of active participation at all. As mentioned earlier, the participants of the beginner group wished that the instructor would demonstrate them the solution, which resulted in a stepwise implementation of the assignment by the instructor with minimal input by the participants, while the participants of the group with prior experience were solving the tasks on their own and required help only in a few non-trivial cases. To facilitate this, the assignments were formulated in a more detailed way, with steps outlining which constructs to choose and how to use them.

As we can see, depending on the conditions in which the course is performed, its design and chosen approaches may change drastically out of necessity. Due to the transition to online conferences, we have moved from a less formal and dynamic learning environment to a more classic lecture-style approach. However, this way we could separate the groups by their knowledge, which seems to suit our students according to their feedback after the course.

Course Structure

Since the support for procedural programming was implemented first, initial iterations of the course were focused entirely on this paradigm. With the subsequent support for other paradigms added to MuLE, the course was expanded by additional chapters covering these concepts and restructured to ensure more or less fluent transition to new chapters. Thus, the current version of the course is built around the *procedural-first* approach [10], i.e. we start by teaching the concepts of procedural programming and move on to object-oriented and functional programming at later stages in the course.

Regardless whether the course was performed in presence in the computer lab or entirely via online meetings, we have used the *Elearning* educational platform of the University of Bayreuth as a central hub to organize the course, host all related information and files as well as gather feedback. Since the start of the Covid-19 pandemic, the students were informed about the separation into two groups based on their previous knowledge prior to the beginning of the course. In order to estimate which group they should visit, they had to answer a set of self-assessment questions with answers given in less readable form next to the questions. The questions included MuLE code snippets where, for example, the resulting value of a variable was asked, or purely conceptual questions, e.g. what are the truth values. The students had to answer these questions on their own and if they felt that they knew the correct answer to most of the questions, they were advised to participate in the group with prior knowledge. The students were not restricted to a specific group after their decision, if they felt that they had chosen poorly, they were free to change their group.

After the initial organisational chapter, we begin with the actual contents of the course. The course is structured into six main chapters, each chapter focuses onto specific related concepts. Initially, algorithms were part of their separate chapter with more tasks and content, including flow charts and assignments based around programming using executable flow charts with the RAPTOR tool [70]. However, due to the subsequent implementation of new programming concepts in MuLE and the necessity to discuss them in the course, some contents were reduced or removed entirely due to the present time constraints.

The course is structured into the following chapters:

1. **Variables, data types, I/O** – We begin by discussing a money withdrawal algorithm at an automated teller machine, thus introducing the concept of algorithms and their relevance in the world of computer science. We continue by formulating the algorithm for the computation of the *greatest common divisor* together with the students, introducing the concepts of *data* (variables, input/output) and *data manipulation* (assignment instructions and control flow). This leads us to the implementation of our very first `"Hello, world!"` program and the discussion of its elements. Subsequently, we introduce and discuss variables, elementary data types (except for boolean), user input, assignments, arithmetic operators, string concatenation, type conversion as well as error types. Of all chapters, this one covers the most concepts, which are comparably fairly easy to comprehend and are required to understand topics featured in the subsequent chapters.

2. **Control structures** – We start by analysing the already discussed money withdrawal algorithm and take a specific interest at steps, in which certain conditions are examined, in turn leading to skipping or repeating other steps. We continue by introducing relational operators, the elementary type `boolean` with the corresponding values as well as operators and, ultimately, the `if`-statement. Another new concept is iteration via the `loop`-statement as well as its termination via the `exit`-statement. Among other examples, the money withdrawal algorithm is implemented and incrementally expanded utilizing these concepts.

3. **Complex data types** – This chapter covers the user defined enumerations and composition as well as one- and two-dimensional lists. Real world examples are used as a motivation for the introduction for each of these constructs, e.g. weekdays or a menu for enumerations, points in a coordinate system or cars (wherein the previously introduced enumerations are reused) for compositions and shopping lists or a parking-lot (again reusing previous types) for lists. Finally, we finish this chapter by discussing multidimensional lists and introduce a chessboard example, which relies on all previously learned language constructs and is further expanded in subsequent chapters.

4. **Operations** – As an analogy to loops and their goal to reduce redundancy, we introduce the concept of named subroutines with the ultimate goal to increase reusability. Therefore, operation parameters, procedures, functions and return statements are also discussed in this chapter. The topic of recursion is, however, not part of this chapter and is covered in the last chapter instead.

5. **Reference types and object-oriented programming** – The previously implemented chessboard example is used as a motivation for reference types. The issue with the previous implementation is that figures are handled as value types, i.e. a figure is placed on each field of the board due to MuLE's policy that each variable and attribute is initialized with a default value after its declaration. References and their default value `null` are used to alleviate this problem. The differences between stack allocated and heap allocated values are explained on a simplified memory model. We continue with a *car sale* scenario, where one `Person` is meant to sell a `Car` to another `Person`. This scenario is used as a running example for the rest of this chapter. We start by analysing static and dynamic features of a car relevant to our scenario and implement it in a procedural way initially. We then introduce object-oriented programming by moving the operations related to the type `Car` into the corresponding type declaration. Same is done with the type `Person`. We continue to introduce other concepts of this paradigm, such as inheritance, redefinition of operations, data abstraction via visibility modifiers and abstract types, etc., and incrementally expand the *car sale* scenario.

6. **Recursion and functional programming** – Both topics are rather difficult to understand for beginner programmers, which is why we have put them into the final chapter, which may or may not be covered due to lack of time at the end of the course. The topic of recursion is introduced by examples of recursion in the nature and use the classic Fibonacci function as our first example. We learn how to translate loops into recursive operations and vice versa, implement both recursive functions and procedures (one of the tasks is to draw a tree using the `Turtle` library) as well as a simple recursive data structure. The topic of functional programming is then introduced by the discussion of the lambda calculus as well as the lambda expressions and how they compare to named operations. As a motivation for this topic, we demonstrate the use of a `forEach` operation of the `Lists` library, which applies a lambda expression to a list of values, with different results depending on the passed lambda expression. Concepts of operations as data, higher order functions and currying are covered in this chapter. Among other tasks, the students have to implement a `filter` operation, which returns a filtered list depending on the passed lambda expression.

In addition to the regular chapters and the corresponding assignments, we have given a set of additional tasks, which are not covered by the instructor during the course and are meant as an additional challenge for experienced students. These tasks are more difficult compared to standard assignments. Furthermore, their formulation is rather abstract, usually, we just give the expected result and the students have to figure out everything else, i.e. the algorithm as well as which constructs to use, on their own.

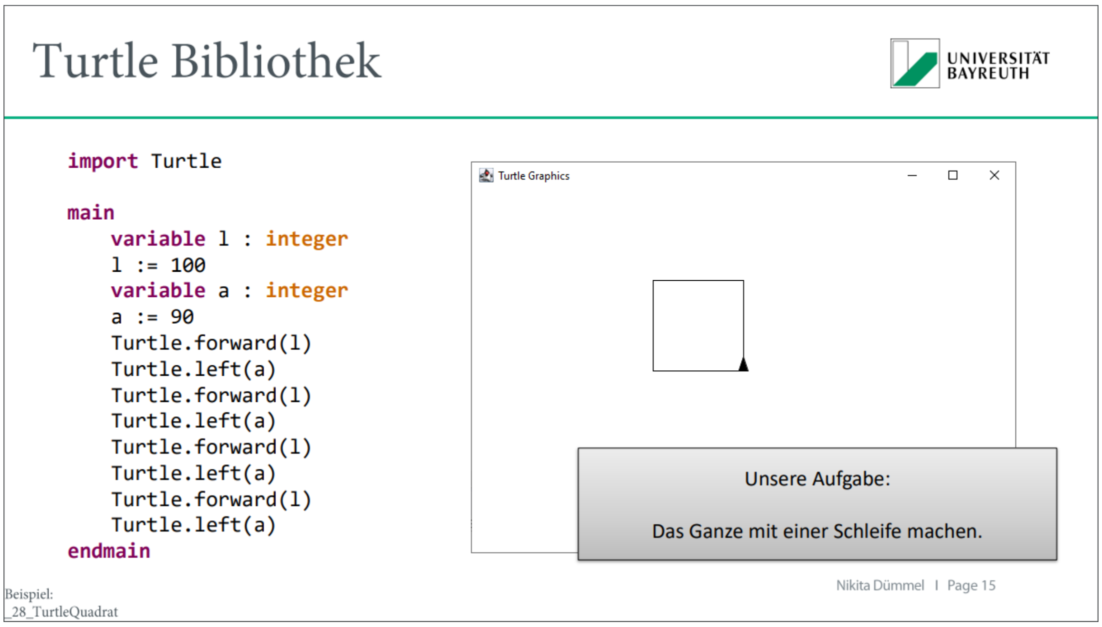
An Example of an Educational Unit

As already mentioned, an educational unit usually starts with a real world example or a program, which was implemented as a part of a previous topic but sill has some unresolved issues. Let us take a look at the topic of loops, which is a part of the second chapter. By now, we have discussed the money withdrawal algorithm and implemented a simple variant of it using `if`-statements to check if we have enough money on our account to make a withdrawal. As a motivation we discuss that after a transaction the ATM does not shut down but returns to its initial state and waits for further input allowing to repeat the process without restarting it manually.

We start by demonstrating a simple counting loop, which prints a set of incrementing integers until the termination condition is fulfilled. The program is executed once normally, and then step-by-step using debugging tools. Afterwards, we return to our money withdrawal algorithm, and implement a loop together with the students which waits for user input. The user must enter numbers, which represent options either to withdraw money or to terminate the program. This way, we demonstrate the concept of potentially infinite loops as well as its reliance on previously known concepts by using a familiar scenario.

As an additional motivation to use loops, we take another program implemented during the chapter one. The task of this program is to keep the students motivated and interested in future content by using the `Turtle` library to draw simple shapes. Experience has shown, that using this library can make the students more relaxed and increase their interest in the topic. Since control flow is not yet part of chapter one, initially the square shape is drawn simply by copy-pasting the instructions responsible for the drawing as well as for the rotation of the cursor, as can be seen in the screenshot in figure 47. Now that the concept of loops is presented in chapter two, the students are tasked to alter this implementation using a `loop`-statement. Here, it is important to check whether they actually make a good use of this construct and remove the redundant lines. Some students would simply put all the present copy-pasted instructions into a loop effectively drawing four squares on top of each other, which necessitates an explanation why we are actually using loops.

The screenshot also demonstrates, that the slides are designed in such a way that they can be used by students who prefer to work on their own or can not participate in the live sessions for various reasons. Since this is a rather simple task, not much explanation is given on the slide. In case of more complex tasks, we give an explanation of the underlying algorithm as well as which constructs should be used to implement it. Nevertheless, in the lower-left corner of the slide we see the name of the corresponding compilation unit, which can be downloaded as a part of an Eclipse project associated with this chapter. This is meant to help students to quickly find the corresponding example program or assignment in the respective projects. Furthermore, the example programs in the projects are usually more extensive than what can be normally covered on a presentation slide. Comments are additionally used to explain the functionality of the example program or to formulate the steps that must be implemented in an assignment.



Turtle Bibliothek

```
import Turtle

main
    variable l : integer
    l := 100
    variable a : integer
    a := 90
    Turtle.forward(l)
    Turtle.left(a)
    Turtle.forward(l)
    Turtle.left(a)
    Turtle.forward(l)
    Turtle.left(a)
    Turtle.forward(l)
    Turtle.left(a)
endmain
```

Unsere Aufgabe:

Das Ganze mit einer Schleife machen.

Beispiel:
_28_TurtleQuadrat

Figure 47: A screenshot of one of the tasks covering the topic of loops.

9.1.2. Students' Feedback and Performance

The course is usually performed over a duration of two weeks prior to the start of the semester. In the latest iteration at the time of writing, each session took 120 minutes, with a 10 minute break in the middle. Each student visits one session per day, making it ten sessions in total. Each year, we have an average of 80 to 90 participants at the start of the course, with approximately 50 remaining at the end. Most of the students drop out right after the first day, probably realizing that they are not interested in the course. Presumably, some of them realize that they do not have to actively participate in the course and can become acquainted with the concepts and solve the assignments at their own leisure. However, we have not attempted to interview these students to find out the real reasons why they stop participating. The average participation numbers and the attrition rate have not changed when we have switched from Python to MuLE, leading to the assumption that the choice of the programming language is not an issue for most of the students.

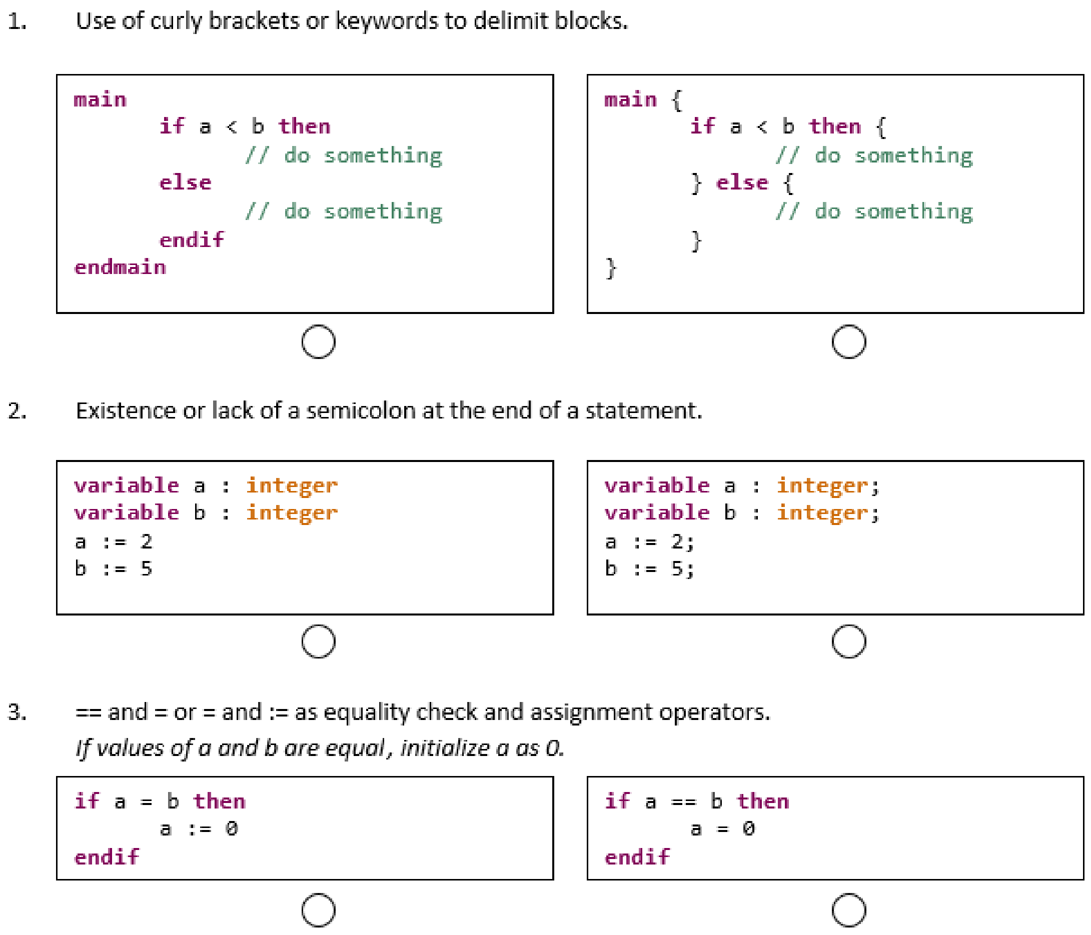
Initially, in computer lab held iterations, the students were asked to fill out two anonymous questionnaires, one at the beginning of the course meant for us to assess the students background knowledge, and one at the end of the course with the intention to gather feedback. In the online held iterations, the questionnaires were hosted in the central course platform. The number of filled out questionnaires has dropped since then, probably since the students are not given an actual printed out form and are thus feeling less obliged to fill it out. The questionnaires changed slightly over time, but their main goals remained the same.

The First Questionnaire

The first questionnaire was performed right at the beginning of the course, before the students were confronted with MuLE. It included questions regarding prior programming experience as well as code snippets which helped us to make some syntax related decisions. Initially, the snippets included an assignment statement, a loop and an `if`-statement written with MuLE (which had a quite different syntax at that time), Java and Quorum [79] with slight semantic differences among the examples. The students had to assign the correct semantics to these statements. The main outcome of this questionnaire was the change of the assignment operator in MuLE from `<--` to `:=`. Our assumption that using the same operator for assignment and equality check as it is done in Quorum might confuse the students was confirmed by this questionnaire.

The latest iteration of this question was intended to compare some syntactical choices we made for MuLE to the concrete syntax of C-based languages (see figure 48). The initial results did not look promising, for all three questions over the half of the participants preferred C-based syntax. As it turned out, in that year, we had a higher than usual number of participants with prior experience. Out of 70 total participants in the questionnaire, 45 had prior programming experience, mostly with Java. Once we had filtered them out, we had another picture which is demonstrated in figure 49. Students without prior experience clearly preferred keywords over brackets to denote blocks. As for the other two questions, the results were inconclusive. And even though students with prior experience tended to choose familiar syntax, around 25% of them preferred keywords to denote blocks and a lack of a semicolon and around 33% preferred MuLE's choice of assignment and equality operators (see figure 50).

As a result of this questionnaire, we assume that our choice of syntax is mostly correct. Even though the majority of students that were exposed to other languages preferred the syntax that was familiar to them, a recognizable part of this cohort would rather prefer the unfamiliar syntax of MuLE. One third of experienced students seem to be unhappy with the choice of the equality and the assignment operators in languages with C-based syntax and would rather prefer the variant present in MuLE. Students without prior experience were more in favour of MuLE's syntax, or at least neutral. Especially in regards to the use of



1. Use of curly brackets or keywords to delimit blocks.

```
main
        if a < b then
                // do something
        else
                // do something
        endif
endmain
```

○

```
main {
        if a < b then {
                // do something
        } else {
                // do something
        }
}
```

○

2. Existence or lack of a semicolon at the end of a statement.

```
variable a : integer
variable b : integer
a := 2
b := 5
```

○

```
variable a : integer;
variable b : integer;
a := 2;
b := 5;
```

○

3. == and = or = and := as equality check and assignment operators.
If values of a and b are equal, initialize a as 0.

```
if a = b then
        a := 0
endif
```

○

```
if a == b then
        a = 0
endif
```

○

Figure 48: Questions targeted at the choice of the concrete syntax.

keywords to denote blocks, inexperienced students are preferring keywords.

The issue with such questionnaires is that only very specific choices regarding the concrete syntax can be analysed using such questions. The more complex decisions, such as whether or not to include a `foreach`-loop as a separate construct and if it is allowed to manipulate the list it is iterating over, are difficult to wrap in an appropriate question for the target audience, i.e. programmer beginners who lack an understanding of such concepts. A couple of more threats to the validity of these observations are:

- As we have seen, experienced students tend to prefer options familiar to them.

- Inexperienced students may have chosen options at random. Due to their lack of experience, they can not know which advantages one option has over the other. For example, some of them might prefer semicolons to separate or terminate statements assuming that it might increase readability without

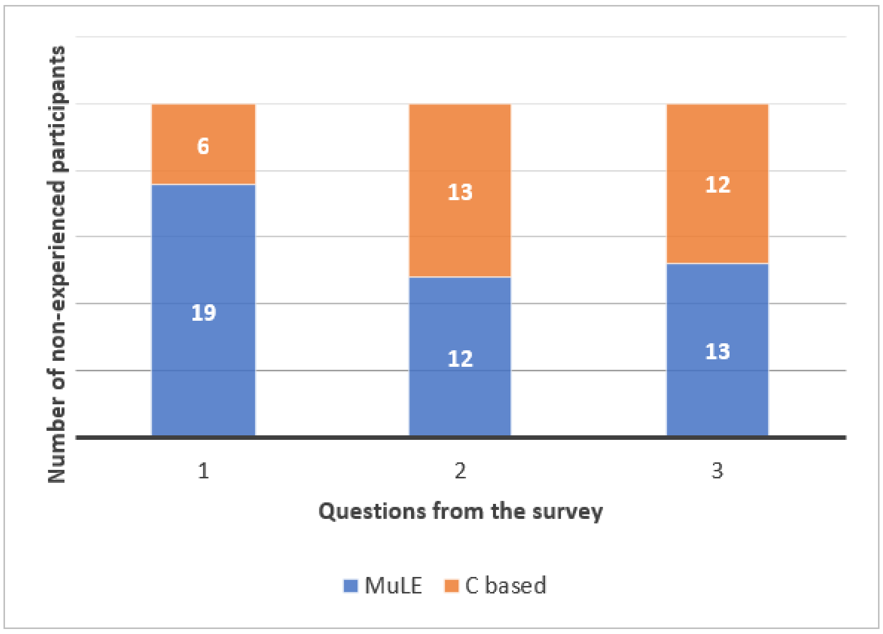


Figure 49: Results of the questionnaire in figure 48, participants with prior experience filtered out.

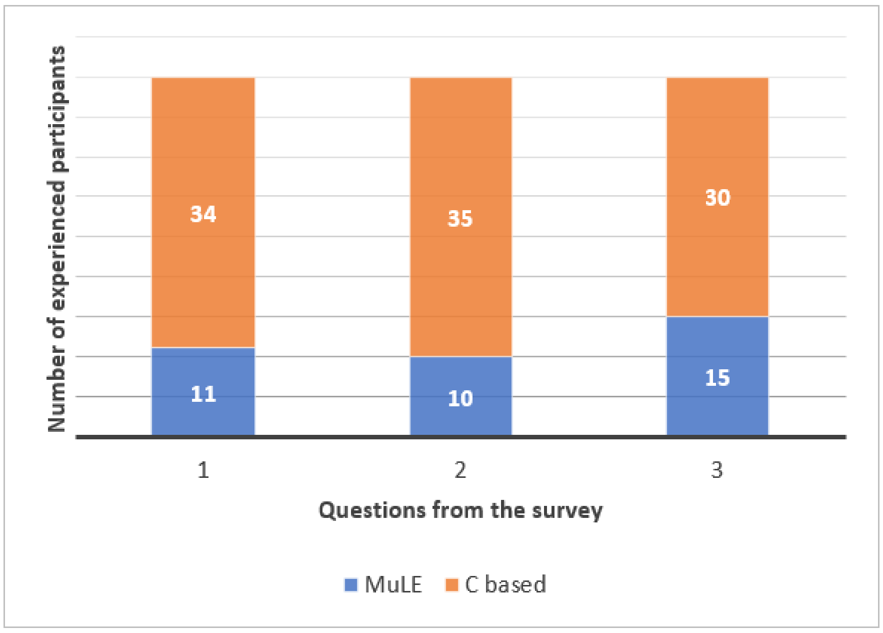


Figure 50: Results of the questionnaire in figure 48, includes only participants with prior experience.

ever having experienced semicolon related errors.

- In general, the participants had little time to answer the questions meaning that not much thought could have been put into their decisions.

- Option three could have been made more expressive by changing the assignment to `a = a + 1` which could potentially lead to different results.
- The questionnaire has changed over time, this particular variant was performed only once. Therefore, we lack data gathered over several years which we could use to make more reliable statements.

With the transition to the online mode, we have removed this questionnaire entirely and replaced it with the self-assessment form which we have described earlier.

The Second Questionnaire

The second questionnaire was performed at the end of the course with the intention to gather feedback both for the course and for MuLE. The questions regarding the course are mostly multiple-choice questions, e.g. *You had enough time to complete the exercises. (not at all, rather not, rather yes, absolutely).* Students were also asked to give written feedback and suggestions, which were used to improve subsequent iterations of the course. A small number of students wished that another language was used in the course, e.g. either Python or Java instead of MuLE. However, when we were using Python, some were wishing Java was used instead. The majority of the students did not mention that they wished another language while another minority explicitly stated that they were glad that an easier language was used. This leads us to believe, that, although we can never make everyone happy, in its current state MuLE is in fact appropriate for the task it was designed for.

The latest two iterations of the course were performed online, with the result that we had less participants in this questionnaire. Less than a half of participants remaining at the last day of the course filled out the questionnaire. The overall feedback was, however, rather similar to the feedback gained from previous years. The major difference was that students were explicitly happy about the separation into groups based on prior knowledge, which was wished for in previous iterations. In all iterations, some students were positively mentioning the use of the `Turtle` library. Each year, some complaints were issued regarding the increase of complexity starting with chapter three.

In addition to course feedback, this questionnaire included an ungraded test, intended to assess the state of the knowledge after the course. Initially, this test included ten written conceptional questions as well as four code snippets with multiple-choice questions. In the latest iteration, the test was separated into two tests with six questions for the beginner group and four additional questions for the experienced group. All of them were multiple-choice questions based around a program with multiple correct answers. The answers included the output of the program, potential error messages, explanations of concepts, etc. The questions of the beginner group were based on topics covered by the first four chapters, the additional four questions of the experienced group were based on the topics of the other two chapters.

```
operation foo(parameter x : integer)
    x := x * 2
    IO.writeInteger(x)
endoperation

main
    variable x : integer
    x := 2
    foo(x)
    IO.writeString(", ")
    IO.writeInteger(x)
endmain
```

Listing 143: The program example of the question six of the second questionnaire.

As an example, the question six is built around the program in listing 143, the participants are notified that multiple answers are correct. The possible answers are:

a. The parameters keep their default values, if users don't pass them explicitly.

b. The output of the program is `4, 4`.

c. Users must pass values for all parameters when invoking an operation.

d. Operations without a return type must be terminated with an empty `return`-statement.

e. Operations with a return type must be terminated with one or more `return`-statement with the corresponding value.

f. The output of the program is `4, 2`.

The correct answers are *c*, *e*, and *f*, choosing all three results in one point for this question, i.e. each option yields 0.33 of the point for this question. Selecting incorrect options does not lead to total point reduction. This question was present in both the beginner test and in the advanced test.

In all iterations of the course, the majority of participants managed to get over the half of the points, meaning that in a graded test roughly 90% of the participants would pass the course. However, we should keep in mind, that the participation was non-mandatory, i.e. we do not know if the non-participants would produce similar or different results. In the latest iteration, only half of the

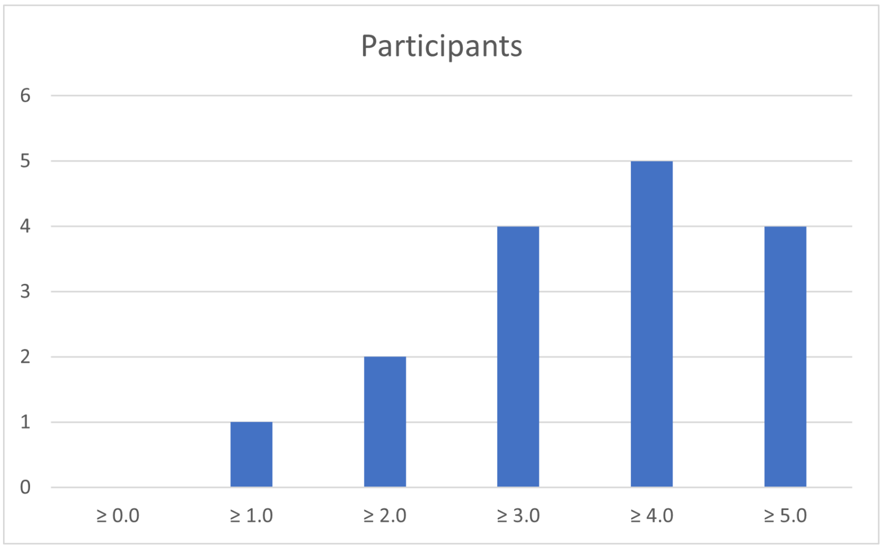


Figure 51: Results of the test in the second questionnaire taken by participants without prior experience.

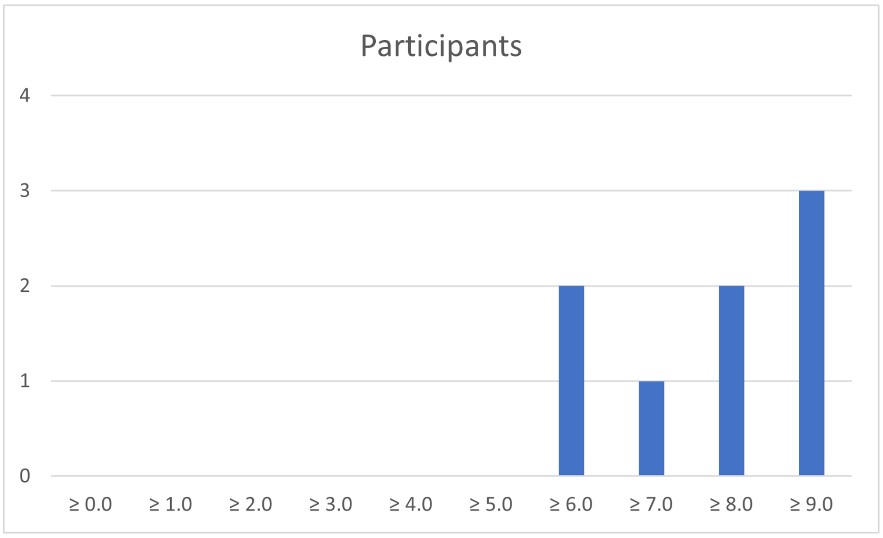


Figure 52: Results of the test in the second questionnaire taken by participants with prior experience.

students remaining at the end of the course have taken the test, and even less filled out the feedback questionnaire.

The results of participants with prior programming experience were overall better compared to students without experience. All experienced students would pass the test, while roughly 19% of those without prior experience have gathered less than 50% of the points. On average, the cohort of the experienced students got around 83% of the total points while inexperienced students got roughly 66%, meaning that at the end of the course the knowledge heterogeneity among the students is still present.

Among others, the threats to the validity of this questionnaire are:

- Lack of a control group using another language, e.g. Python, to compare results at the end of the course.
- Small sample size, which is especially evident in the number of experienced participants who have taken the test, which is eight.
- The attrition over the course is not taken into the account. The performance of the students who have dropped out could be different altering the entire result.
- The previous point leads to the issue that we have only compared students within the course. It would be interesting to compare inexperienced students who have participated in the course with students with prior experience who did not. This would allow us to make more concrete statements about the effect of the course on the knowledge heterogeneity among the students.

Performance of the Students at the End of the Semester

Over the years, we have compared the performance in the final exam of the CS1 lecture at the end of the first semester of those students who have participated in the course to those who have not. The results are displayed in table 4 and in figures 53 and 54. The failure rate represents the percentage of the students of each group who have failed the final exam. As a reminder, the students are required to use Java in this exam, same as in the corresponding CS1 lecture. The lesser the average score, the better the results, the best score is 1.0 while the lowest is 5.0, in which case the student fails the exam.

The first two iterations were performed using Python. As we can see, in the very first iteration the participants of the course had a better average score and a lesser failure rate compared to non-participants. In 2017, the overall results were slightly different, the participants had still performed better, however, the difference of the failure rates was no longer that remarkable. Overall, in both years the participants had better results in the final exam. The difference between both years can be explained by different compositions of the students, unlike in 2016, in the second iteration most of the students were enrolled in their first semester.

However, when we look at 2018, the year when we have used MuLE for the first time, we get a rather bleak picture. The participants had yielded overall worse results compared to non-participants. One explanation for this could be that neither MuLE nor its supporting tools were mature enough at that time, in fact, the syntax of the language looked quite differently at that time. The course was

	2016	2017	2018	2019	2020	2021
Language	Python	Python	MuLE	MuLE	MuLE	MuLE
Mode	lab	lab	lab	lab	online	online
Number of P	44	48	48	36	65	48
Number of NP	74	62	77	78	80	60
Total number	118	110	125	114	145	108
P avg. score	3.01	3.20	3.15	2.88	3.09	2.78
NP avg. score	3.11	3.40	2.62	3.10	3.55	3.85
Total avg. score	3.07	3.31	2.83	3.03	3.35	3.38
P failure rate	16%	30%	27%	28%	23%	17%
NP failure rate	24%	32%	19%	26%	40%	48%
Total failure rate	21%	31%	22%	26%	32%	34%

Table 4: Comparison of course participants (P) and non-participants (NP) in the final exam at the end of the semester.

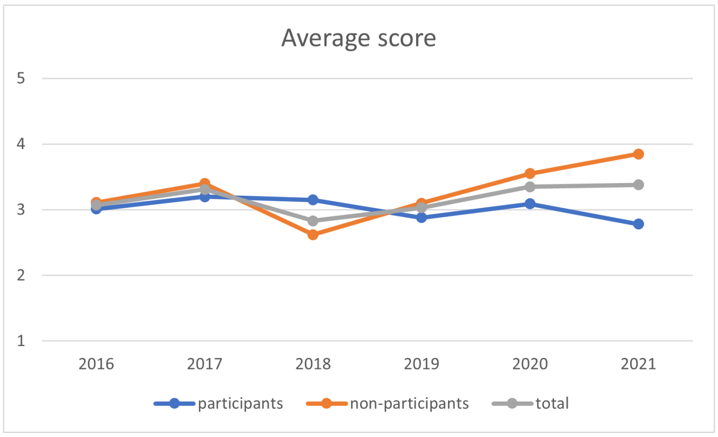


Figure 53: Average scores of the students in the final exam over the years.

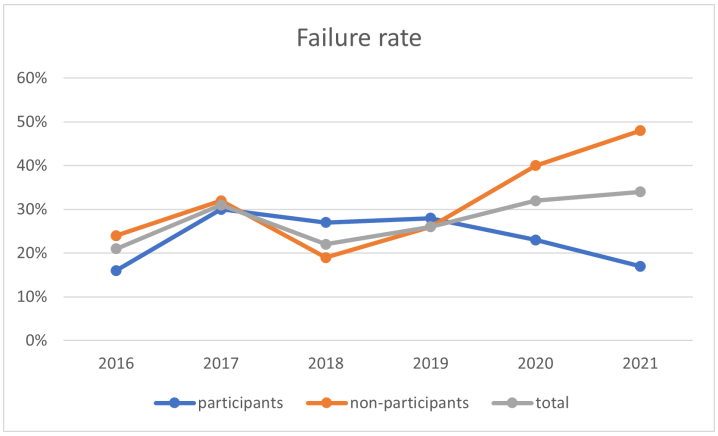


Figure 54: Failure rates of the students in the final exam over the years.

designed anew for this year, which means that its design and structure included mistakes which could be ironed out for later iterations. We have used the feedback of the students to improve both MuLE and its tools, as well as made slight changes to the design of the course. In the next year, the performance shifted. Although the participants still have a slightly higher failure rate, their average score is better than that of non-participants, meaning that in this year the group of course participants included both a high number of very good students, as well as those who have failed the exam. Some errors in the implementation of the language were still encountered by the students during this iteration, however, overall, the syntax of the language represented the state described in this thesis.

The iteration in year 2020 had again shown different results. First, we had an overall higher number of students. Furthermore, for the first time, the course was performed online. This might explain a generally higher participation rate, since the students were not forced to be physically present at the university two weeks before the actual start of the lectures. Additionally, this was also the first iteration where we have covered object-oriented programming, which represents at least a third of the final exam. This, as well as the, by now, improved state of MuLE, the supporting tools and the design of the course, might explain the overall better performance among the participants.

In the final iteration in 2021, which was also performed online, we had overall less students compared to the previous iteration. However, the participation rate remained roughly the same 44%. This was also the first iteration, in which we had discussed functional programming, although it should not have any significant impact on the results since this topic was not covered in the final exam. As we can see, the performance of the participants is even better in this iteration compared to 2020, especially noticeable in the failure rate diagram in figure 54. Only 17% of course participants have failed the final exam, while about 48% of non-participants failed which in our opinion implies, that there is a certain correlation between students' participation in the course and their performance in the final exam.

Of course, we have to consider additional factors when looking at these numbers. Thus the issues with these observations are as follows:

- Each year, the composition of the students changes slightly compared to the previous year, for example, in the year 2016 non-computer-science students who wrote the exam were enrolled in higher semesters and had demonstrated better results compared to computer-science students enrolled in their first-semester. A large number of these students had also participated in the course. This has changed in subsequent years, most of our students are now enrolled in the first semester.

- Furthermore, we assume that students who participate in a non-mandatory course are by their nature more motivated, which might also explain the, usually, better results of course participants.

- The change of used language, the environment, the fact that the final exam is changing in each subsequent year, all of these circumstances make these observations highly inconsequential.
- Due to the anonymous nature of these analyses, we do not know which students had actual prior programming experience and who did not. We only know, who had participated in the course.
- We do not know what impact on the performance had the language itself, and what was caused by the course design. The results might be similar if the students had participated in the same course but with a different language.

Thus, the only concrete statement that we can make based on this data, is that the performance of course participants has improved over the last three iterations, leading us to assume that both the language and the course have matured over this time.

9.1.3. Conclusion and Threats to Validity

Based on the more or less favourable students' feedback and a lack of major critique towards MuLE, we can assume that our target audience is, at large, not deterred by the use of a language that is otherwise not used in the industry. In fact, in recent years some students have explicitly stated that MuLE is a good choice for an introductory course. Of course, there are also those who would rather prefer a “real” language.

The results of the ungraded tests taken directly at the end of the preliminary course are very good. Most students would pass if the test would have been graded. This may be explained by the intensity of the course, the students are actively practising programming over the course of two weeks and take a test immediately after it, without a longer pause in which they might forget previously learned concepts. Nevertheless, we are very happy with these results and assume that, at least in the short term, the course has a positive effect on the students.

To asses the long term effects, we have compared the exam results of those who have participated in the course who those who have not. The exam in question is the final exam of the CS1 lecture which takes place approximately half a year after the preliminary course. Here, we have had some rather mixed results. Nevertheless, except for one case when we have started using MuLE which may have not yet been fully ready for the task, the performance of participants was better than the performance of non-participants. Based on this data, we assume that the participation in the course may have a long term positive effect on the students' performance. Based on the results of the latest iterations of the course, we also assume that MuLE does not have a negative effect, and may even have a positive effect on students' future learning process.

In the previous subsection, we have listed a number of problems for each performed analysis. To sum it up, the general threats to validity of our evaluation of students' performance are as follows:

- The composition of the students, the course structure, chosen methods and tools, even the language itself has evolved over each iteration. The version of MuLE which we have presented in this thesis with all its features was used only in the latest iteration, the procedural part of the language has remained roughly unchanged for the last three iterations. Based on these constant changes it is difficult to make solid assumptions.
- Lack of control groups using other languages, such as Python or Java, while teaching the same contents to directly asses the impact of the language.
- Small sample sizes in certain questionnaires.
- The attrition rate was not considered, neither during the analysis at the end of the course, nor over the semester. In the latter case, it would also be interesting to compare the attrition rate of course participants to non-participants.
- No separation between experienced and inexperienced students when analysing their performance in the final exam which makes it hard to asses which impact the course and the used language had on the actual problem of knowledge heterogeneity among our first semester students.

9.2. Related Work

Professionally used programming languages follow a *more is more* approach [90] regarding included language constructs, data types, libraries, etc. MuLE, as an educational programming language, follows a *less is more* approach, for each language construct in MuLE, professional languages offer a selection of similar constructs. The reasons are discussed in chapter 4. In the same chapter, we have specified a set of requirements for an educational language, we will use these requirements to compare MuLE to professional languages currently used in education, as well as other educational programming languages.

Comparison to Java and Python

As discussed in section 3.2.1, since the advent of object-oriented programming educational institutions have stopped using Pascal, which was specifically designed for education but lacked support for the new paradigm, and moved on to using professional programming languages. The initially used C++ is in our opinion by far one of the most difficult languages to be used in education, which is why Java and Python are rather used today for this task. Thus, in this section we will compare MuLE to these two languages.

```java
class HelloWorldExample {
   public static void main (String[] args) {
      System.out.println("Hello, world!");
   }
}
```

Listing 144: A *"Hello, world!"* program written with Java.

```python
print("Hello, world!")
```

Listing 145: A *"Hello, world!"* program written with Python 3.

```
program HelloWorldExample
import IO

main
   IO.writeString("Hello, world!")
endmain
```

Listing 146: A *"Hello, world!"* program written with MuLE.

Let us first take a look at the implementation of the classic `"Hello, world!"` program using Java (listing 144), Python (listing 145), and MuLE (listing 146). As already mentioned in section 3.2.1, Java is an object-centric programming language, i.e. it is not possible to program in a purely procedural way using this language. This is evident in the `"Hello, world!"` example. When explaining this simple program we have to either explain such object-oriented concepts as classes, visibility modifiers as well as static/non-static context from the beginning overwhelming the students with the amount of information, or save them for later. The latter variant is not optimal either, since the students will adapt to using specific constructs or keywords simply because they are told to do so and not because they understand their meaning.

The Python implementation is the shortest of the three, producing the quickest results. But, in our opinion, this is also one of the weaknesses of Python when used in the context of education. It tends to hide important concepts and rely on implicit behaviour.

Finally, the MuLE variant of this program is more similar to the Java implementation, at least at the first glance. MuLE compilation units are conceptually not equal to classes, like in Java, thus when introducing a program we merely need to differentiate between a program and a library and can follow up by discussing the entry into a program via the main procedure. Differently to the other two languages, we actually have to import the standard library `IO` to be able to use the output producing operations. Both in Java and in Python the respective library is implicitly imported, however other specific libraries have to be imported manually, which represents less consequent behaviour possibly increasing the confusion among the students.

Requirement **1 – Easy to Learn** states that the language should be easily understood by students without prior programming knowledge while at the same time be an effective educational tool, i.e. it should demonstrate important concepts (requirement **9 – Clear execution semantics**) instead of hiding them under a layer of implicity. Although Python has the shorter syntax compared to MuLE, its reliance of implicit behaviour, dynamic typing (which violates the requirement **10 – Explicit static type system**), use of indentations to denote blocks, etc., makes it less suitable to demonstrate these specific concepts. Furthermore, errors caused by dynamic typing or wrong indentations are not possible in MuLE. Both Python and Java have a mix of mutable, e.g. objects, and immutable types, e.g. primitive types and strings. Without knowing which is which, inexperienced students may encounter unexpected behaviour when passing values of these types to operations. MuLE makes a clear differentiation between value and reference types, thus forcing the students to understand the differences but also making it less confusing by using different notations.

Requirement **2 – Non-cryptic syntax** states that the language should provide a readable syntax. In comparison to Java, the syntax of MuLE is more readable due to its reliance on keywords to introduce various declarations (e.g. `type`, `operation` or `variable`) and denote blocks. Various shortcuts, such as `i++`, although convenient, are not necessarily practical in the context of education. Even if the instructor avoids using these constructs, sooner or later experienced students will start asking if they can use them instead of, for example, the longer `i = i + 1`, which is conceptually easier to explain and to understand. At this point, the instructor has to explain what these constructs mean to other students wasting valuable time. Python's syntax is very readable, the use of indentations is one of the reasons for that, which have however other disadvantages, which we have mentioned earlier.

Compared to the other two languages, MuLE offers far less language constructs fulfilling the requirement **3 – Minimal number of language constructs**. For example, if we look at numeric types, MuLE offers `integer` and `rational`, while Java offers `byte`, `short`, `int`, `long`, `float` and `double` as primitive types alone as well as corresponding wrapper types and other class based implementations of numeric types such as `BigDecimal`. While it is useful when used professionally, especially when performance and optimization are involved, such complexity is overwhelming in education. Python, on the other hand, also offers a small amount of primitive types, e.g. `int`, `float` and `complex` for numeric types. Both Java and Python offer various non-orthogonal constructs, such as `while` and `for` loops (or abstract classes and interfaces in Java), making MuLE a better candidate according to the requirement **4 – Orthogonality**. As for the requirement **6 – Build upon present knowledge**, all three languages are rather similar in this context, we can say that MuLE has a slight advantage by using the `:=` operator for assignment and `=` for equality checks as well as strict value copying semantics.

Both Java and Python support the three required paradigms stated in the requirement **7 – Multi-paradigm language**. However, as we have discussed earlier, Java is an object-oriented language first and foremost violating the requirement **8 – No forced object-orientation**, which in turn violates the requirement **5 – Incremental introduction**, at least when attempting to use Java in a course built around a procedural-first approach. Same is evident when attempting to program in a functional way using Java. Python's lambda expressions are conceptually similar to those in MuLE. Compared to MuLE, Python offers more functionality related to functional programming on syntactical level, e.g. `for` loops (simulated as a `forEach` library operation in MuLE) and powerful list comprehension (MuLE merely offers several list creation notations). Then again, this functionality is basically syntactical sugar which violates the requirement **4 – Orthogonality**.

In the context of object-oriented programming regarding the requirement **11 – Data abstraction**, MuLE displays certain similarities to Java, with some simplifications. Java offers two non-orthogonal variants of abstract types, i.e. abstract classes (represented by abstract compositions in MuLE) and interfaces, allowing to simulate multiple inheritance in Java. However, our experience shows, that students have trouble differentiating between these two constructs, furthermore, at least in the context of CS1, such scenarios where both constructs are used for their intended purpose can be simulated by other means. Python follows a rather unusual way of declaring abstract classes, for example by importing an `AbstractBaseClass` from the `abc` module and inheriting from it, or by simulating abstract methods by raising the `NotImplementedError`, which is in our opinion rather confusing. Just like in MuLE, attributes in Python are by default accessible. To restrict the visibility one must type underscores before the member identifier, one underscore representing `protected`, and two underscores `private` visibility, which is in our opinion a less meaningful notation (requirement **2 – Non-cryptic syntax**) and can result in errors caused by the wrong number of typed underscores. Unlike Java and Python, MuLE does not differentiate between static and non-static context within compositions, all members are handled as instance attributes and operations. Static context is either managed by the procedural part of the language or simulated by separate objects.

Although MuLE does support parametric and subtype polymorphism fulfilling the requirement **12 – Polymorphism**, Java and Python have an advantage over MuLE in this regard. MuLE does not support overloading of operations and lacks the ability to pass type parameters to operations, meaning that parametric polymorphism is only supported by operations declared within compositions.

All three languages fulfil the requirement **13 – Modularity**. Since Java and Python are professionally used languages with a long history, they are supported by a far larger range of standard libraries (requirement **14 – Standard libraries**). MuLE offers a rather limited set of types and functions provided by standard libraries, which are nevertheless more than sufficient in the context of CS1. Moreover, MuLE is distributed with a set of built-in educational libraries.

Finally, all three languages fulfil the requirement **15 – Tool support**, however, Java and Python have a clear advantage in this regard due to their widespread professional use. Since Java has also been used in education for quite some time, a number of educational IDE's such as BlueJ [85] have been implemented for this language, which is currently not the case for MuLE.

To sum it up, while Java and Python certainly still have some advantages over MuLE, in case of most of the formulated requirements, MuLE seems to be better suited for education.

Comparison to other Educational Languages

Since Pascal is no longer used in education, supports only procedural programming in its basic version and its tool support is considered obsolete by modern standards, we will not compare it to MuLE. However, it remains to asses how MuLE compares to other textual educational languages, which we have discussed in section 3.2.2, i.e. Grace [76], Quorum [79], Pyret [81], and RESOLVE [82]. The specific shortcomings of each language are discussed in section 3.2.2, here we are going to make a broad comparison between the languages. All being said, by no means do we intend to degrade the contribution of the developers of these languages.

Since all of these languages are implemented with education in mind, they are designed to be easy to learn, at least compared to professional programming languages (requirement **1 – Easy to Learn**). Except for Grace, all languages use keywords to denote blocks. Similarly to MuLE, all languages attempt to reduce the number of language constructs compared to professional languages. Unlike MuLE, they still offer non-orthogonal constructs, e.g. different kinds of loops or conditional statements thus not fulfilling the requirement **4 – Orthogonality** and only partly fulfilling the requirement **3 – Minimal number of language constructs**. Comparing the clarity of the syntax (requirement **2 – Non-cryptic syntax**) and the semantics (requirement **9 – Clear execution semantics**), we would rate MuLE on par with Pyret (which is inspired by Python with certain improvements) and better than Grace (use of brackets to denote blocks, returning a value without an explicit return statement, various notations for method declarations), Quorum (same operator used for assignment statements and equality checks), and RESOLVE (confusing parameter modes and additional conditions which can be represented by other means, shortcuts for keywords).

Most of these languages, except for RESOLVE, support dynamic (violating the requirement **10 – Explicit static type system**) or a mix of static and dynamic typing (violating the requirement **4 – Orthogonality**). None of these languages offer a clear separation between value and reference types the way it is implemented in MuLE (requirement **9 – Clear execution semantics**).

Most of these languages support procedural, object-oriented and functional programming (requirement **7 – Multi-paradigm language**), except for Quorum and RESOLVE, which lack support for functional programming. Usually, one of these paradigms is better represented compared to the others, e.g. Grace has a focus on object-oriented programming, Pyret on functional, while MuLE is focused on procedural. That being said, none of these languages forces to use a specific paradigm, fulfilling the requirements **8 – No forced object-orientation** and **5 – Incremental introduction**.

Concerning the requirement **11 – Data abstraction** in the context of object-oriented programming, Grace offers the highest flexibility, including a way to simulate multiple inheritance similarly to Java, which inherently results in less orthogonality. Quorum supports multiple inheritance but apparently lacks abstract types relying onto the keywords `private` and `public` as the sole means of encapsulation. Neither Pyret, nor RESOLVE support inheritance.

Similarly to MuLE, Grace and Quorum support both subtype and parametric polymorphism (requirement **12 – Polymorphism**). Pyret provides support for parametric polymorphism. RESOLVE seems to support neither of them.

All of these languages fulfil the requirement **13 – Modularity** as well as the requirement **14 – Standard libraries**. Similarly to MuLE, Quorum and Pyret offer standard libraries which are specifically designed to be used in the programming education, such as the Turtle library.

As for the requirement **15 – Tool support**, both Pyret and Qurom are supported by web based interpreters, which help to reduce the effort to start programming with these languages, however lack the tool support provided by a dedicated IDE, such as debugging tools with stepwise execution. Grace is supported by several desktop based interpreters. According to the RESOLVE documentation, it is supported by an Eclipse plug-in similarly to MuLE, however at the time of writing, the provided link was not accessible.

Although all of them are educational programming languages, in the end all of these languages, including MuLE, were developed with a certain focus in mind. MuLE was basically built around procedural programming with the intention to be used in CS1 with a procedural-first approach. Other languages are either focused on another paradigm, thus being better suited in a course built around this way of programming, or are designed around other specific features, such as a built-in verification or testing mechanism. Next to MuLE, of all of these languages Grace seems to fulfil most of the requirements that we have set up for MuLE, however, it also comes with most of the issues which are also present in Java. All of these languages have certain drawbacks or limitations, at least from our point of view, such as lack of support of a certain paradigm, non-orthogonal language constructs, unnecessary syntactical shortcuts, dynamic type system, etc., which makes MuLE more capable for our goals. At the same time, these languages were designed with other requirements in mind and we are certain, that the developers of these languages will find a number of shortcomings within MuLE according to their preferences.

10. Conclusion

At last, we have reached the final chapter of this thesis. To recapitulate, our intention was to facilitate students' entry into programming education by using a simpler programming language, at least compared to Java or C++. The language should still be capable to convey relevant programming concepts. Other existing languages like Python had various shortcomings which, in our opinion, made them less adequate in the context of programming education. Thus, we have implemented our own language called MuLE which represents the main contribution of this thesis. First, we have analysed existing professional and educational languages. Based on the information gained from this research we have summarized a set of requirements for MuLE, which we have used to specify and implement the language.

To sum it up, MuLE provides following features:

- Support of procedural, object-oriented and functional programming. The implementations of the object-oriented and functional paradigms are characterized by their reliance on procedural constructs, due to the initial implementation of procedural programming and its subsequent use as a basis for the implementation of other paradigms. It is not possible to program entirely in an object-oriented or functional manner the way it would be possible using a mono-paradigm language, such as Smalltalk or Haskell.

- A minimalistic set of orthogonal language constructs, sufficient to support the aforementioned paradigms. This reduces the overhead of information confronted by the students setting the focus on understanding the algorithmic solution by removing divergences caused by a multitude of implementations possible by additional non-orthogonal constructs. Instead of introducing a class construct, we have reused the composition construct and expanded it by object-oriented concepts when implementing this paradigm which facilitates the transition from procedural to object-oriented programming in an introductory course. Similarly, the lambda expression is implemented in a very similar way to named operations to increase the number of analogies and reduce confusion, when discussing functional programming.

- Possibility to program in a procedural way without relying onto object-oriented or functional constructs, allowing to easily design a programming course around a *procedural-first* approach without having to worry about handling concepts of other paradigms as black-boxes until they can be reasonably explained at a later stage without overwhelming the students with too much information at once.

- Static explicit type system, which prevents certain typing errors caused by dynamic typing as well as puts data types into focus and thus forces the students to understand this important concept.

- Explicit differentiation between stack and heap allocated values. We have implemented a parameterized reference type, which accepts any other type as a parameter. This makes the use of this type very flexible while at the same time confronting the students with the concept of references at lexical level and forcing them to understand it.

- More or less expressive compile time error messages. Those that are caused by the parser are still rather cryptic, especially for beginner programmers.

- Support by Eclipse and its powerful tools, including background compilation checks with immediate feedback and debugging with stepwise execution.

- Platform independency due to its current Java based implementation. At least version 8 of Java is therefore required to run MuLE.

The language has been used in its intended way, i.e. as an educational tool, in a non-mandatory preliminary programming course which is performed prior to the CS1 lecture at the University of Bayreuth. We have used this as an opportunity to test the language and gather feedback from the target audience, which was used to improve MuLE over time. Based on students' performance immediately at the end of the course, as well as in the final exam of the CS1 lecture roughly half a year after the course, we assume that the current state of MuLE, its tool support and the preliminary course have a rather positive effect on the ability of our students to comprehend the programming concepts taught in CS1.

Future Work

One of the aforementioned requirements was adequate tool support. As mentioned, we are currently using Eclipse as the development environment supporting MuLE due to reasons mentioned in chapter 8. However, Eclipse is by far not a beginner friendly programming environment, the amount of options available to the user can be quite overwhelming, which is exactly the reason why we think that professionally used programming languages and environments are not an appropriate tool in programming education. Therefore, the next priority is to develop a suitable IDE which would correspond to same requirements that apply to the language itself.

A web-browser based programming environment could be beneficial due to its accessibility. Compared to desktop based IDEs, the amount of time and effort required to start programming in such an environment is minimal. Users are not required to follow installation instructions to get an IDE running, a process, in which inexperienced users may encounter errors which they are not yet capable to solve on their own. A web-based environment requires only a web-browser and can be seamlessly integrated into a web-based tutorial or even an entire programming course. Ultimately, this would make the language more attractive for potential users, who may be sitting on the fence about using it and would rather decide not to due to a complex installation process of required tools.

The compile time constraint checking mechanism can also benefit from additional improvements. For example, we could implement checks that would provide meaningful error messages for cases, currently backed by the parser. In such cases, the error messages can be rather cryptic, for example the error message `extraneous input ‘attribute’, expecting RULE_ID` is displayed when attempting to declare an attribute inside of an enumeration. An inexperienced user may rightfully assume that something is wrong with the keyword *attribute*, but will most likely be totally baffled by the expected *RULE_ID*. Furthermore, previous parser errors may cause other errors in the subsequent parts of the encompassing construct, which may add to the overall confusion.

At the time of writing, MuLE suffers from two restrictions in regards to parametric polymorphism, i.e. generic types can only be used wrapped as corresponding reference types and the fact that it is not possible to declare generic operations. These restrictions should be eliminated.

In general, MuLE would certainly benefit from further development. Currently, the language provides support for three programming paradigms, i.e. procedural, object-oriented and functional. However, in chapter 2 we have covered logic programming as another major paradigm. Although this paradigm differs significantly from the other three, it could be implemented as a library, which would for example provide specific types or operations capable to register facts, rules based on these facts as well as the ability to resolve queries. Basically, the implementation of such a library would be similar to the implementation of a logic programming language such as Prolog.

Speaking of which, MuLE could only benefit from additional libraries designed for education. One of the tasks of programming education is not only to teach programming per se, but also to keep students motivated and perhaps even attract others to the world of programming. To achieve this, MuLE should provide opportunities to be used in more different ways, for example in the programming of robots or microcontrollers. This, together with a specifically designed IDE, could also make MuLE more attractive for secondary education and other educational facilities. The language is suited not only to be used in a preliminary programming course, but would also fit into the high school curriculum in the context of the German educational system.

Additionally, the language would benefit from additional statistical analysis performed in better designed controlled experiments and a detailed evaluation of individual language constructs to further assess the performance of the language itself and the specific design decisions met during its development. In chapter 9, we have discussed that the performance of the participants of the preliminary programming course was usually better in the final exam of the CS1 lecture, compared to the performance of non-participants. We have also mentioned, that the gathered data is lacking important details, e.g. we did not separate the course participants in those who had prior programming experience before the start of the semester and those who had none. This comparison (under additional consideration of further variables, such as prior experience) should be continued over the course of the following years to further back the assumption, that MuLE and the course designed around it are actually beneficial for our students.

Finally, it would be interesting to design an alternative course based on an objects-first approach, separate the students into two groups and use the different approaches in the respective groups in order to assess, whether MuLE is flexible enough to be used in a differently designed course, and which approach is more effective at improving the students' ability to apply same concepts in a different language, such as Java, in subsequent courses. Another idea would be to use Java or Python in the control group and compare the performance of these groups to asses whether MuLE is actually beneficial compared to these languages.

A. Installation Instructions

This section covers the installation instructions for MuLE as an Eclipse plug-in.

1. Java version 8 or higher is required to run MuLE, and should therefore be installed first (if not already present on the target machine). MuLE was tested on both *Oracle Java SE* [121] and *OpenJDK* [122], other distributions should work too but we can not guarantee it. Current versions of Java are built only for 64bit systems, however, when installing an older version, such as Java 8, users must pay attention whether they are installing a 32bit or a 64bit version.

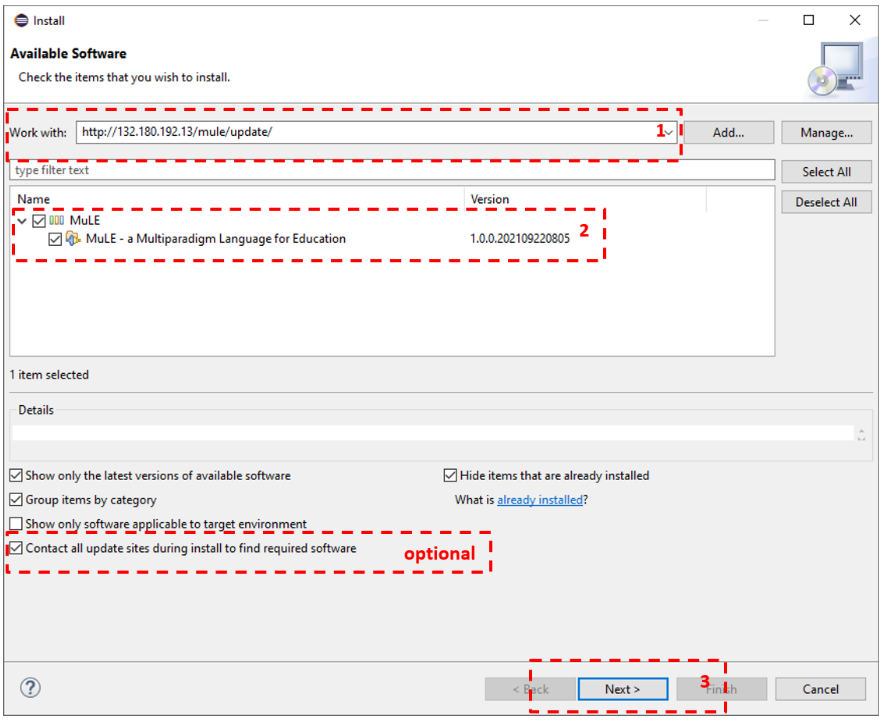
2. The next step is to download and install or simply unpack Eclipse. The standard package *Eclipse IDE for Java Developers* [123] is sufficient to run MuLE, however, any other package can also be used. Similarly to current Java versions, current Eclipse packages are built only for 64bit systems. Older packages were built for both 32bit and 64bit systems and require the corresponding Java version, i.e. a 32bit Eclipse version will require an installation of a 32bit Java version. MuLE was tested with the 2021-9 build, we do not guarantee it working on releases earlier than 2019.

3. When starting Eclipse, the user is prompted to select a path for the workspace folder, where the projects will be located on the system.

4. Once Eclipse has started, the user should navigate to the menu bar (see sector 1 in section 7.1), select the category *Help*, and from there select the entry *Install new Software.* At this point, a new window will open which is displayed in screenshot in figure 55.

   The first step here is to enter the URL to the MuLE update site, which is *http://132.180.192.13/mule/update/*, in the *Work with* entry field. Users may simply place the URL of the MuLE update site directly in the entry field or create a corresponding entry by pressing the *Add* button first. Once the URL is entered, the MuLE plug-in is displayed (step two) and must be selected in order to proceed. Finally, users must press the next button and follow the subsequent instructions. Optionally, users may disable the option in the step marked as *optional* in the screenshot, this will cause an error message in the next steps but MuLE will be installed anyway and will work properly. This step is required, if the normal installation fails without the optional step.

5. In the subsequent steps, the user must accept the licence agreement, ignore any dependency (caused by the optional step mentioned before) or authenticity (caused by unsigned content) related warnings and wait until the installation is complete.


Install

Available Software

Check the items that you wish to install.

Work with: http://132.180.192.13/mule/update/

type filter text

Name	Version
☑ MuLE	
☑ MuLE - a Multiparadigm Language for Education	1.0.0.202109220805

1 item selected

Details

☑ Show only the latest versions of available software

☑ Group items by category

☐ Show only software applicable to target environment

☑ Contact all update sites during install to find required software

☑ Hide items that are already installed

What is already installed?

Figure 55: Screenshot of the plug-in installation user interface.

6. Once the installation is finished, the user is prompted to restart Eclipse. If the installation was successful, the user should be now able to create new MuLE projects and files via their respective wizards.

B. Grammar Definitions

```
grammar org.eclipse.xtext.common.Terminals hidden(WS, ML_COMMENT, SL_COMMENT)

import "http://www.eclipse.org/emf/2002/Ecore" as ecore

terminal ID: ’^’?(’a’..’z’|’A’..’Z’|’_’) (’a’..’z’|’A’..’Z’|’_’|’0’..’9’)*;
terminal INT returns ecore::EInt: (’0’..’9’)+;
terminal STRING:
    ’"’ ( ’\\’ . /* ’b’|’t’|’n’|’f’|’r’|’u’|’"’|"’"|’\\’ */ | !(’\\’|’"’) )* ’"’ |
    "’" ( ’\\’ . /* ’b’|’t’|’n’|’f’|’r’|’u’|’"’|"’"|’\\’ */ | !(’\\’|"’") )* "’"
;
terminal ML_COMMENT : ’/*’ -> ’*/’;
terminal SL_COMMENT : ’//’ !(’\n’|’\r’)* (’\r’? ’\n’)?;

terminal WS        : (’ ’|’\t’|’\r’|’\n’)+;

terminal ANY_OTHER: .;
```

Listing 147: Definition of the Xtext `Terminals` grammar.

```
grammar de.ubt.ai1.mule.MuLE with org.eclipse.xtext.common.Terminals
generate muLE "http://www.ubt.de/ai1/mule/MuLE"

CompilationUnit:
    (isProgram?=’program’ | isLibrary?=’library’) name=ID
    imports += Import*
    programElements += ProgramElement*
    main=MainProcedure?;

QualifiedName: ID (’.’ ID)*;

Import: ’import’ importedNamespace=[CompilationUnit];

MainProcedure:
    {MainProcedure} ’main’ block=Block ’endmain’;

ProgramElement: TypeDeclaration | Operation;

NamedElement: EnumerationValue | TypeDeclaration | Feature | CompilationUnit;

////// DATA TYPES //////
DataType: BasicType | DeclaredType | ReferenceType | ListType | OperationType;

DeclaredType:
    type=[TypeDeclaration|QualifiedName]
    (’<’ typeParams+=DataType (’,’ typeParams+=DataType)* ’>’)?;

```

```
BasicType: typeName=('integer' | 'rational' | 'string' | 'boolean');

ReferenceType: 'reference' '<' type=DataType '>';

ListType: 'list' '<' type=DataType '>';

OperationType:
    {OperationType} 'operation'
    '(' (paramTypes+=DataType (',' paramTypes+=DataType)*)? ')' (':' type=DataType)?;

VisibilityModifier: 'private' | 'protected';

TypeDeclaration: Composition | Enumeration | TypeParameter;

Enumeration:
    visibility=VisibilityModifier? 'type' name=ID ':' 'enumeration'
    values+=EnumerationValue (',' values+=EnumerationValue)* 'endtype';

EnumerationValue: name=ID;

Composition:
    visibility=VisibilityModifier? abstract?=('abstract')? 'type' name=ID
    ('<' typeParams+=TypeParameter (',' typeParams+=TypeParameter)* '>')?
    ':' 'composition' ('extends' superType=[Composition|QualifiedName]
    ('<' superTypeParams+=TypeParameter (',' superTypeParams+=TypeParameter)* '>')? )?
        typeDeclarations+=TypeDeclaration*
        attributes+=Attribute*
        operations+=Operation*
    'endtype';

TypeParameter: name=ID ('extends' superType=[Composition|QualifiedName])?;

////// FEATURES //////
Feature: Attribute | VariableDeclaration | Parameter | Operation ;

Attribute: visibility=VisibilityModifier? 'attribute' name=ID ':' type=DataType;

Parameter: 'parameter' name=ID ':' type=DataType;

Operation:
    override?=('override')? visibility=VisibilityModifier? abstract?=('abstract')?
    'operation' name=ID '(' (params+=Parameter (',' params+=Parameter)*)? ')'
    (':' type=DataType)? (block=Block 'endoperation')?;

Block: {Block} statements+=Statement*;


```

```
////// STATEMENTS //////
Statement:
    VariableDeclaration | AssignmentOrOperationCall | IfStatement | LoopStatement |
    LetStatement | ReturnStatement | ExitStatement;

ReturnStatement: {ReturnStatement} 'return' (=> expression=Expression)?;

ExitStatement: {ExitStatement} 'exit';

VariableDeclaration: 'variable' name=ID ':' type=DataType;

AssignmentOrOperationCall:
    (SymbolReference | SuperExpression)
    ({AssignmentOrOperationCall.left=current} ':=' right=Expression)?;

LoopStatement: {LoopStatement} 'loop' block=Block 'endloop';

IfStatement:
    'if' expression=Expression 'then' block=Block
    elseIfs+=ElseIf*
    (=> 'else' elseBlock=Block)?
    'endif';

ElseIf: 'elseif' expression=Expression 'then' block=Block;

LetStatement:
    'let' variable=VariableDeclaration 'be' expression=Expression 'do' block=Block
    elseLets+=ElseLet*
    (=> 'else' elseBlock=Block)?
    'endlet';

ElseLet:
    'elselet' variable=VariableDeclaration 'be' expression=Expression 'do' block=Block;

////// EXPRESSIONS //////
Expression:
    OrExpression;

OrExpression returns Expression:
    AndExpression ({OrExpression.left=current} op=('or') right=AndExpression)*;

AndExpression returns Expression:
    EqualityExpression ({AndExpression.left=current}
    op=('and') right=EqualityExpression)*;

EqualityExpression returns Expression:
    ComparisonExpression ({EqualityExpression.left=current}
    op=('=' | '/=') right=ComparisonExpression)*;

ComparisonExpression returns Expression:
    AdditiveExpression ({ComparisonExpression.left=current}
    op=('<' | '<=' | '>' | '>=') right=AdditiveExpression)*;

AdditiveExpression returns Expression:
    MultiplicativeExpression ({AdditiveExpression.left=current}
    op=('+' | '-' | '&') right=MultiplicativeExpression)*;

MultiplicativeExpression returns Expression:
    ExponentExpression ({MultiplicativeExpression.left=current}
    op=('*' | '/' | 'div' | 'mod') right=ExponentExpression)*;

ExponentExpression returns Expression:
    AtomicExpression ({ExponentExpression.left=current}
    op=('exp') right=AtomicExpression)*;

AtomicExpression returns Expression:
    SymbolReference                                             |
    SuperExpression                                             |
    {StringConstant} value=STRING                               |
    {IntegerConstant} value=INTEGER                             |
    {RationalConstant} value=RATIONAL                           |
    {BooleanConstant} value=('true' | 'false')                  |
    {Null} 'null'                                               |
    {Unary} op=('+'|'-'|'not') expression=AtomicExpression      |
    {Reference} 'reference' expression=AtomicExpression         |
    {ParenthesizedExpression} '(' expression=Expression ')'     |
    ListInit                                                    |
    LambdaExpression                                            ;

LambdaExpression returns Expression:
    {LambdaExpression} 'operation'
    '(' (parameters+=Parameter (',' parameters+=Parameter)*)? ')'
    (':' type=DataType)? block=Block 'endoperation';

SuperExpression: {SuperExpression} 'super' '.' memberCall=SymbolReference;

SymbolReference:
    symbol=[NamedElement]
    compositionInit=SymbolRefCompositionInit?
    accessModifier=SymbolRefAccessModifier?
    ('.' memberCall=SymbolReference)?;

SymbolRefAccessModifier:
    {OperationInvocation} '(' (params+=Expression (',' params+=Expression)*)? ')'
```

```
                            accessModifier=SymbolRefAccessModifier? |
    {ListAccess} '[' index=Expression ']' accessModifier=SymbolRefAccessModifier? |
    {Dereference} '@' accessModifier=SymbolRefAccessModifier? ;

SymbolRefCompositionInit:
    {SymbolRefCompositionInit} '{' (attributes+=SymbolRefCompositionAttribute
                                (',' attributes+=SymbolRefCompositionAttribute)*)? '}';

SymbolRefCompositionAttribute:
    attribute=[Attribute] '=' expression=Expression;

ListInit:
    {ListInit} "[" (left=Expression right=(ListInitFunction | ListInitElements))? "]";

ListInitFunction:
    {ListInitFunction} op=("**" | "..") expression=Expression;

ListInitElements:
    {ListInitElements} ("," elements+=Expression)*;

@Override
terminal ID: ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;

@Override
terminal STRING: '"' ( '\\' . | !('\\'|'"') )* '"';

terminal INTEGER: INT;

terminal RATIONAL: INT '.' INT ('E' ('+' | '-')? INT)?;
```
Listing 148: Definition of the MuLE grammar with Xtext implementation details.

C. IO Library API

This section provides the specification of the operations provided by the library `IO`, which we have discussed in section 6.1.

- `writeLine()` – creates a line break, other operations in this library do not create a line break after the printed content.

- `writeString(arg : string)` – prints a string value on the console.

- `writeBoolean(arg : boolean)` – prints a boolean value on the console.

- `writeInteger(arg : integer)` – prints an integer value on the console.

- `writeRational(arg : rational)` – prints a rational value on the console.

- `writeFile(path : string, content : string)` – creates a file with the given path and the string content. If a file under the given path already exists, its content is overwritten, meaning that the users should apply this operation with caution. The entered path can be relative to the project, i.e. a simple file name will search for the file in the project folder (or create it if it does not exist), or absolute. In the latter case, all folders in the path must exist, otherwise the file can not be created. In any case, after the execution of the operation the message `File path: [ABSOLUTE PATH]` is displayed on the console.

- `readString() : string` – reads a character sequence from the console and returns it as a string value.

- `readBoolean() : boolean` – reads a character sequence from the console and returns the corresponding boolean value if the entered character sequence is either `true` of `false`. If not, `false` is returned as a default value.

- `readInteger() : integer` – reads a character sequence from the console and returns the corresponding integer value if the entered character sequence can be parsed as integer. If not, `0` is returned as a default value.

- `readRational() : rational` – reads a character sequence from the console and returns the corresponding rational value if the entered character sequence can be parsed either as a floating point number or an integer. If not, `0.0` is returned as a default value.

- `readFile(path : string) : string` – reads a file under the given path and returns its contents as a string. If the file does not exist, the returned string is `"file not found"`.

- `readFileLines(path : string) : list<string>` – reads a file under the given path and returns its contents as a list of strings, wherein each entry of the list is a single line of the file. If the file does not exist, the returned list is `["file not found"]`.
- `fileExists(path : string) : boolean` – returns `true` if a file with the given path does exist, and `false` otherwise.

D. Mathematics Library API

- `sin(a : rational) : rational` – returns the result of the *sine* function of an angle measured in radians.
  `Mathematics.sin(pi / 4.0)` returns `0.7071067811865475`

- `cos(a : rational) : rational` – returns the result of the *cosine* function of an angle measured in radians.
  `Mathematics.cos(pi / 4.0)` returns `0.7071067811865475`

- `tan(a : rational) : rational` – returns the result of the *tangent* function of an angle measured in radians.
  `Mathematics.tan(pi / 4.0)` returns `0.9999999999999999`

- `asin(a : rational) : rational` – returns the result of the *arc sine* function, the resulting angle is measured in radians.
  `Mathematics.asin(0.707)` returns `0.785247163395153` ($\approx 45°$)

- `acos(a : rational) : rational` – returns the result of the *arc cosine* function, the resulting angle is measured in radians.
  `Mathematics.acos(0.707)` returns `0.7855491633997437` ($\approx 45°$)

- `atan(a : rational) : rational` – returns the result of the *arc tangent* function, the resulting angle is measured in radians.
  `Mathematics.atan(1)` returns `0.7853981633974483` ($\approx 45°$)

- `toDegrees(a : rational) : rational` – converts an angle measured in radians to an angle measured in degrees.
  `Mathematics.toDegrees(0.7853981633974483)` returns `45.0`

- `toRadians(a : rational) : rational` – converts an angle measured in degrees to an angle measured in radians.
  `Mathematics.toRadians(45)` returns `0.7853981633974483`

- `log(a : rational) : rational` – returns the logarithm with the basis e.
  `Mathematics.log(0)` returns `-Infinity`
  `Mathematics.log(1)` returns `0.0`

- `log10(a : rational) : rational` – returns the logarithm with the basis 10. Results of `log10(0)` and `log10(1)` are same as above.

- `round(a : rational) : integer` – rounds the floating point number and returns an integer. Only the first decimal is considered.
  `Mathematics.round(2.49)` returns `2`
  `Mathematics.round(2.50)` returns `3`

- `floor(a : rational) : integer` – returns the integer part of a floating point number.

  `Mathematics.floor(2.50)` returns `2`

- `absoluteInteger(a : integer) : integer` – returns the absolute value of an integer.

  `Mathematics.absoluteInteger(-5)` returns `5`

- `absoluteRational(a : rational) : rational` – returns the absolute value of a floating point number.

  `Mathematics.absoluteRational(-5)` returns `5.0`

- `randomInteger(min : integer, max : integer) : integer` – returns a randomly generated integer number between `min` and `max` ($min \leq Number \leq max$). An example is shown in listing 105.
- `randomRational() : rational` – returns a randomly generated floating point number ($0.0 \leq Number < 1.0$). An example is shown in listing 105.
- `pi() : rational` – returns an approximation of the number π, the returned value is `3.141592653589793`.
- `e() : rational` – returns an approximation of the number e, the returned value is `2.718281828459045`.
- `getMaxIntegerValue() : integer` – returns the highest supported integer value, which is `2147483647`.
- `getMinIntegerValue() : integer` – returns the smallest supported integer value, which is `-2147483648`.
- `getMaxRationalValue() : rational` – returns the highest supported floating point value, which is `1.7976931348623157E308`.
- `getMinRationalValue() : rational` – returns the smallest supported floating point value, which is `4.9E-324`.

E. Strings Library API

- `subString(str : string, startPos : integer, endPos : integer) : string` – returns a substring of the passed `str` according to the passed border indices `startPos` and `endPos`. Both borders are inclusive. If `endPos` is lesser than `starPos`, the result is an empty string.

  `Strings.subString("[hi]", 1, 2)` returns `hi`

- `lengthOf(str : string) : integer` – returns the number of characters in a given string.

  `Strings.lengthOf("[hi]")` returns `4`

- `indexOfSubString(str : string, subStr : string) : integer` – returns the position of a substring in a given string. On several occurrences of a substring, the index of the first occurrence is returned. The result is the index of the first character of the substring in the given string. If the substring is not part of the string, `-1` is returned instead.

  `Strings.indexOfSubString("ababa", "ba")` returns `1`

- `replaceAll(str : string, regex : string, repl : string) : string` – replaces all occurrences of the passed substring `regex` in the given string `str` by the replacement string `repl`.

  `Strings.replaceAll("a:b:c:d", ":", "-")` returns `a-b-c-d`

- `replaceFirst(str : string, regex : string, repl : string) : string` – replaces the first occurrence of the passed substring `regex` in the given string `str` by the replacement string `repl`.

  `Strings.replaceFirst("a:b:c:d", ":", "-")` returns `a-b:c:d`

- `split(str : string, regex : string) : list<string>` – splits the given string `str` into a list of substrings at all occurrences of the passed substring `regex`.

  `Strings.split("a:b:c:d", ":")` returns `[a, b, c, d]`

- `toUpperCase(str : string) : string` – replaces all applicable characters with the corresponding upper case variant.

  `Strings.toUpperCase("a:b.C:D")` returns `[A:B.C:D]`

- `toLowerCase(str : string) : string` – replaces all applicable characters with the corresponding lower case variant.

  `Strings.toLowerCase("a:b.C:D")` returns `[a:b.c:d]`

- `integerToString(num : integer) : string` – converts an integer value to a string value and returns it.

- `rationalToString(num : rational) : string` – converts a rational value to a string value and returns it.
- `booleanToString(bool : boolean) : string` – converts a boolean value to a string value and returns it.
- `genericToString(arg : Type) : string` – converts a value of any type to a string value and returns it. This operation can be used to convert values of enumerations, compositions, lists and references into their respective string representations.

F. Lists Library API

- `isEmpty(l : list<Type>) : boolean` – checks if the passed list is empty.

  `Lists.isEmpty([])` returns `true`
  `Lists.isEmpty([1, 2, 3])` returns `false`

- `lengthOf(l : list<Type>) : integer` – returns the number of entries in the passed list.

  `Lists.lengthOf([])` returns `0`
  `Lists.lengthOf([1, 2, 3])` returns `3`

- `indexOf(l : list<Type>, element : Type) : integer` – returns the index of the first occurrence of the passed element. If the element is not present, the result is `-1`.

  `Lists.indexOf([], 2)` returns `-1`
  `Lists.indexOf([1, 2, 2], 2)` returns `1`

- `append(l : list<Type>, element : Type) : list<Type>` – creates a shallow copy of the passed list, appends the passed element at the end of the copy and returns it.

  `Lists.append([1, 2, 3], 4)` returns `[1, 2, 3, 4]`

- `head(l : list<Type>) : Type` – returns the first entry in the list. Passing an empty list will cause a runtime exception.

  `Lists.head([1, 2, 3])` returns `1`

- `tail(l : list<Type>) : list<Type>` – returns a copy of the passed list without its first element. Passing an empty list will cause a runtime exception.

  `Lists.tail([1, 2, 3])` returns `[2, 3]`

- `last(l : list<Type>) : Type` – returns the last entry of the passed list. Passing an empty list will cause a runtime exception.

  `Lists.last([1, 2, 3])` returns `3`

- `subList(l : list<Type>, min : integer, max : integer) : list<Type>` – creates a list filled with copies of the entries of the original list starting at the index `min` and ending at the index `max`. If `min` is greater than `max`, the result is an empty list. Otherwise, if at least one of these values lies outside of the bounds of the list, a runtime exception will occur.

  `Lists.subList([1, 2, 3], 1, 2)` returns `[2, 3]`

- `insert(l : list<Type>, element : Type, pos : integer) : list<Type>` – creates a shallow copy of the original list with the passed element inserted at the passed position. Previous entry at this position as well as all subsequent entries are moved one position to the rear of the list. The new list is returned.

`Lists.insert([1, 2, 3], 42, 1)` returns `[1, 42, 2, 3]`

- `removeElement(l : list<Type>, element : Type) : list<Type>` – returns a shallow copy of the passed list without the first occurrence of the passed value. All entries after this position are moved one position to the front of the list.

  `Lists.removeElement([1, 2, 2], 2)` returns `[1, 2]`

- `removePosition(l : list<Type>, pos : integer) : list<Type>` – returns a copy of the passed list without the entry at the passed position. All entries after this position are moved one position to the front of the list.

  `Lists.removePosition([1, 2, 3], 1)` returns `[1, 3]`

- `filter(l : list<Type>, op : operation(Type) : boolean) : list<Type>` – returns a list with copies of entries of the original list, which fulfil the condition specified by the passed operation.

- `forEach(l : list<Type>, op : operation(Type))` – applies the passed operation to each entry of the passed list in order of their appearance.

G. Turtle Library API

Following enumeration types are specified within the library:

- `Speed` – defines a set of options for the speed of the animation. The values are `SLOW, MEDIUM, FAST` and `INSTANT`.
- `Colors` – specifies a set of predefined colors, which are `WHITE, BLACK, RED, GREEN, BLUE, YELLOW, MAGENTA, CYAN, PINK, ORANGE, LIGHT_GRAY` and `DARK_GRAY`.
- `Orientation` – specifies a set of four orientations for the Turtle pen: `NORTH, SOUTH, EAST` and `WEST`. The standard orientation is `NORTH` or 0°.

Furthermore, following operations are included in the library:

- `forward(length : integer)` – moves the pen in its current direction for the given number of pixels. A line is drawn as long as `penUp` was not invoked before.
- `backward(length : integer)` – moves the pen against its current direction for the given number of pixels. A line is drawn as long as `penUp` was not invoked before.
- `right(degree : rational)` – turns the pen to the right for the given angle.
- `left(degree : rational)` – turns the pen to the left for the given angle.
- `penUp()` – stops drawing if the pen is moved. Can be used to move the pen to another place in the canvas without drawing a line in between.
- `penDown()` – starts drawing again when the pen is moved, if `penUp` was invoked earlier.
- `setColor(color : Colors)` – sets the color of the pen to one of the predefined colours.
- `setColorRGB(r : integer, g : integer, b : integer)` – sets the colour of the pen defined by an RGB value, the values of the parameters must be in range `[0 .. 255]`.
- `setThickness(thickness : integer)` – sets the width of the pen, i.e. the thickness of the drawn lines, for the given number of pixels. The initial value is one pixel.
- `moveTo(x : integer, y : integer)` – moves the pen to a specific coordinate in the canvas and draws a line, if `penUp` was not invoked earlier. The direction of the pen after the movement is that of the direction of the movement.

- `setPosition(x : integer, y : integer)` – places the pen at the given coordinates of the canvas. The pen retains its original orientation after the placement. The initial placement of the pen is at *(300, 200)* pixels.
- `setDirection(arc : rational)` – set the direction of the pen towards a specific angle.
- `setOrientation(o : Orientation)` – sets the direction of the pen towards a predefined orientation.
- `setFrameSize(x : integer, y : integer)` – sets the size of the canvas in pixels.
- `getX() : integer` – returns the value of the x-coordinate of the pen.
- `getY() : integer` – returns the value of the y-coordinate of the pen.
- `getArc() : rational` – returns the current direction of the pen as an angle.
- `startFilledPolygon(color : Colors)` – starts drawing a polygon filled with a predefined colour.
- `startFilledPolygonRGB(r : integer, g : integer, b : integer)` – same as above, however the colour is defined by RGB values.
- `endFilledPolygon()` – terminates the drawing of a polygon.
- `circle(radius : integer)` – draws a circle with a given radius measured in pixels.
- `filledCircle(radius : integer, color : Colors)` – draws a circle filled with a predefined colour.
- `filledCircleRGB(radius : integer, r : integer, g : integer, b : integer)` – draws a circle filled with a color defined by RGB values.
- `setAnimationSpeed(speed : Speed)` – sets the speed of the drawing animation.
- `showCoordinateSystem(bool : boolean)` – is used to toggle the coordinate grid.
- `showCursor(bool : boolean)` – is used to toggle the visual representation of the pen.
- `activateDrawMode(speed : Speed)` – starts the drawing mode wherein the pen is controlled by the arrow keys, which can be used to familiarize users with the behaviour of the Turtle pen.

H. UBTMicroworld Library API

The library provides the abstract composition `Agent`, which is used to interface with the agents in the environment and specifies following operations for this task:

- `getXPosition() : integer` – returns the current x-coordinate of the agent.
- `getYPosition() : integer` – returns the current y-coordinate of the agent.
- `getXstartPosition() : integer` – returns the initial x-coordinate of the agent at the start of the level.
- `getYstartPosition() : integer` – returns the initial y-coordinate of the agent at the start of the level.
- `moveForward()` – attempts to move one step forward.
- `moveBackward()` – attempts to move one step backward.
- `rotateRight()` – rotates the agent 90° to the right.
- `rotateLeft()` – rotates the agent 90° to the left.
- `doNothing()` – the agents waits until next instructions.
- `isTileInFrontOfWalkable() : boolean` – returns `true` if the tile directly in front of the agent is accessible.
- `isTileBehindWalkable() : boolean` – returns `true` if the tile directly behind the agent is accessible
- `getID() : integer` – returns the id of the agent.
- `getStepsMade() : integer` – returns the number of steps made by the agent.
- `getNumberInputs() : integer` – returns the number of attempted actions.
- `getNumberInvalidInputs() : integer` – returns the number of invalid actions.
- `getLastMoveDirection() : MoveDirection` – returns the last `MoveDirection` made by the agent.
- `getMoveDirectionList() : list<MoveDirection>` – returns the list of all `MoveDirections` made by the agent.

- `getLineOfSight()` : `LineOfSight` – returns the current direction faced by the agent as specified by the enumeration type `LineOfSight`.
- `getNumberOfCollectedObjects()` : `integer` – returns the number of collected objects.

Furthermore, the library defines following enumeration types:

- `DefaultLevelType` – specifies literals representing the default levels `DEFAULT_LEVEL_0` up to `DEFAULT_LEVEL_16`.
- `DelayTime` – used to set the delay time between the actions of the agents, the options are `NO_DELAY`, `SHORT_DELAY`, `MEDIUM_DELAY`, `LONG_DELAY`. After an agent receives an instruction, the animation is played after the delay time.
- `LineOfSight` – resembles the direction currently faced by an agent, the values are `NORTH`, `EAST`, `SOUTH`, `WEST`.
- `MoveDirection` – resembles the successfully performed or attempted but failed movements by an agent, the values are `FORWARD`, `BACK`, `RIGHT`, `LEFT`, `NONE`, `INVALID`, `FORWARD_INVALID`, `BACK_INVALID`.
- `ObjectType` – specifies the presence or the lack of an object on the map: `NO_OBJECT`, `KEY`.
- `TerrainType` – specifies the various terrain types: `GRASS`, `SAND`, `PATH`, `SNOW`, `STONE`, `WATER`, `TARGET`, `START`.
- `WinValidatorType` – used to initialize levels with specific level completion rules: `VALIDATOR_1`, `VALIDATOR_2`, `VALIDATOR_3`, `CUSTOM`.

Finally, following operations are specified by the library:

- `initCustomGame1(terrain : list<list<TerrainType>>, validator : WinValidatorType)` – initializes a custom level with a terrain map defined as a 2D list of terrain types and a win validator.
- `initCustomGame2(terrain : list<list<TerrainType>>, objects : list<list<ObjectType>>, validator : WinValidatorType)` – same as the above operation with the addition of the objects map defined by a 2D list of object types.
- `initCustomGame3(terrainMapPath : string, validator : WinValidatorType)` – initializes a custom level with a terrain map defined by a path to an image and a win validator.

- `initCustomGame4(terrainMapPath : string, objectsMapPath : string, validator : WinValidatorType)` – same as the above operation with the addition of the objects map defined by a path to an image.
- `initDefaultGame(value : DefaultLevelType)` – initializes one of the pre-defined levels.
- `getAgentList() : list<reference<Agent>>` – returns the list of agents.
- `getNavMap() : list<list<String>>` – returns a 2D list of strings representing a simple navigational map, which shows accessible and inaccessible tiles as well as the start and target tiles.
- `getTerrainType(x : integer, y : integer) : TerrainType` – returns the `TerrainType` of the tile at the specified coordinates.
- `isGameRunning() : boolean` – returns `true` if the game is not finished.
- `setDelayTime(delayTime : DelayTime)` – sets the delay time between actions performed by the agents.
- `setGameFinished()` – sets the state of the game to game finished.
- `setGameOver()` – sets the state of the game to game over.
- `registerAgentForKeyListener(agent : reference<Agent>)` – registers an agent for a key listener, this agent can now be controlled by the arrow keys on the keyboard. Only one agent can be registered.

I. GUIFactory Library API

This library is separated into five manageable compilation units.

`GUIFactory` Unit

This is the core compilation unit, it contains the composition `Colour`, which does not specify any operations as well as the composition `Window`, which represents the root element of a GUI and specifies following operations:

- `showWindow()` – forces a window to open, can be used to test initial pane placement. As soon as a widget, i.e. a component that is not a pane, is added, the window will open automatically when the program is executed without the need to invoke this operation.
- `setPane(pane : reference<GUIFactoryPanes.Pane>)` – sets the main pane of a window.
- `getPane() : reference<GUIFactoryPanes.Pane>` – returns the main pane of a window.
- `setSize(width : integer, height : integer)` – sets the width and height of a window in pixels.
- `setResizable(resizable : boolean)` – allows or disables the resizing of a window.
- `setTitle(title : string)` – sets the title of a window.

Furthermore, this unit contains the following enumeration types which define values for predefined colours, alignments, etc.:

- `Palette` – specifies a set of predefined colours, which can be used to create an instance of `Colour` by the operation `createColourFromPalette`. The values are: `LIGHT_RED`, `RED`, `DARK_RED`, `CYAN`, `LIGHT_BLUE`, `BLUE`, `DARK_BLUE`, `LIGHT_YELLOW`, `YELLOW`, `DARK_YELLOW`, `LIGHT_GREEN`, `GREEN`, `DARK_GREEN`, `ORANGE`, `GOLD`, `LIGHT_GREY`, `GREY`, `DARK_GREY`, `LIGHT_BROWN`, `BROWN`, `DARK_BROWN`, `PURPLE`, `BLACK`, `WHITE` and `TRANSPARENT`.
- `HorizontalAlignment` – specifies a set of alignments required by specific components, e.g. a `TextField`. The values are: `LEFT`, `CENTER` and `RIGHT`.
- `FontType` – specifies specific font types, the values are: `PLAIN`, `ITALIC`, `BOLD` and `ITALIC_BOLD`.
- `CheckBoxAlignment` – specifies alignments for the `CheckBox` component, the values are: `NONE`, `ICON_LEFT_TEXT_RIGHT`, `ICON_RIGHT_TEXT_LEFT`, `ICON_TEXT_LEFT` and `ICON_TEXT_RIGHT`.

- `Alignment` – specifies alignments for specific components, e.g. `VerticalPane` and `HorizontalPane`. The values are: `NONE`, `TOP_LEFT`, `TOP_CENTER`, `TOP_RIGHT`, `CENTER_LEFT`, `CENTER`, `CENTER_RIGHT`, `BOTTOM_LEFT`, `BOTTOM_CENTER` and `BOTTOM_RIGHT`.

Finally, the unit contains operations which create and return instances of `Window` and `Colour`:

- `createWindow(title : string, width : integer, height : integer) : reference<Window>` – creates a `Window` instance and returns the corresponding reference.

- `createColour(r : integer, g : integer, b : integer, a : integer) : Colour` – creates a `Colour` instance based on the RGBA values.

- `createColourFromPalette(palette : Palette) : Colour` – creates a `Colour` instance from one of the predefined literals in the enumeration type `Palette`.

`GUIFactoryPanes` Unit

There are four pane types specified in this library unit, as well as the corresponding creation operations:

- `VerticalPane` – components are arranged vertically when added as children to this pane. The corresponding creation operation is `createVerticalPane(alignment : GUIFactory.Alignment, spacing : integer) : reference<VerticalPane>`. Children components are displayed in the same order as they are added via the `addComponent(c : reference<Component>)` operation to the pane, i.e. the first added child will be displayed at the top while the last at the bottom of the pane.

- `HorizontalPane` – components are arranged horizontally when added as children to this pane. The corresponding creation operation is `createHorizontalPane(alignment : GUIFactory.Alignment, spacing : integer) : reference<HorizontalPane>`. Similarly to the `VerticalPane`, the children are displayed in the same order as they are added, starting from left to right.

- `GridPane` – components are arranged in a grid pattern, they have to be added at specific coordinates as children. The upper left corner has the coordinates *(0, 0)*. Unlike the previous panes, the `addComponent` operation, which adds a component to the pane, requires coordinates, where the added component is placed. The corresponding creation operation is `createGridPane(columnSpace : integer, rowSpace : integer) : reference<GridPane>`.

- `BorderPane` – this pane is separated into five sectors: top, bottom, center, left and right, thus representing the common arrangement inside of a GUI separated into a header, menu bar, content page, contributions bar, etc. Unlike with the previous panes, children components are not added via the simple `addComponent` operation, but via setter operations, e.g. `setTop` or `setCenter`. Furthermore, the size if the components contained directly in the border pane is overridden so that they take up all available space defined by the corresponding sector. The corresponding creation operation is `createBorderPane(hgap : integer, vgap : integer) : reference<BorderPane>`.

`GUIFactoryBorders` Unit

This library unit contains an abstract type `Border`, an enumeration `BorderType` with the values `RAISED` and `LOWERED`, three non-abstract border types which are subtypes of `Border` as well as the corresponding creation operations. The border types are:

- `TitledBorder` – this border type has a title in the upper left corner of the border which is given upon the creation of the border. Users must also specify the colour and the thickness of the border. The corresponding creation operation is `createTitledBorder(colour : GUIFactory.Colour, thickness : integer, title : string) : reference<TitleBorder>`.

- `LineBorder` – this is a simple line border, users must specify the colour, the thickness and whether the corners are rounded or not. The corresponding creation operation is `createLineBorder(colour : GUIFactory.Colour, thickness : integer, rounded : boolean) : reference<LineBorder>`.

- `BevelBorder` – This border has a shadow effect and appears either as raised or lowered against the background. The users must therefore specify whether the border is raised or lowered as well as the colours of the highlight and shadow effects.The corresponding creation operation is `createBevel-Border(borderType : BorderType, highlight : GUIFactory.Colour, shadow : GUIFactory.Colour) : reference<BevelBorder>`.

`GUIFactoryComponents` Unit

This unit contains `Components` which are not subtypes of `Pane`. The included components are:

- `Component` – this is a super type for all components of this library unit as well as all panes, specifies self-explanatory operations `setVisible(boo-lean)` and `getParent() : reference<Component>`. No specific creation operation exists for this type. All components of this unit have the attributes `width` and `height` which can be accessed by the corresponding getter operations and edited with the operation `setSize(width : integer, height : integer)`.

- `Shape` – this is a subtype of `Component` and a super type for the compositions `Rectangle`, `Polygon` and `Ellipse`. It includes additional operations that can set the colour of the shape or assign an image to it.

- `Rectangle` – This type represents rectangular shapes and offers no additional operations. The corresponding creation operation is `createRectangle(width : integer, height : integer) : reference<Rectangle>`.

- `Ellipse` – The corresponding creation operation is `createEllipse(width : integer, height : integer) : reference<Ellipse>`.

- `Polygon` – This type represents polygonal shapes defined by a set of points with x and y pixel coordinates which are relative to the space assigned to the polygon with the coordinates *(0, 0)* representing the upper-left corner of the polygon space. Additional points can be added via the operation `addPoint(x : integer, y : integer)`. The corresponding creation operation is `createPolygon(xPoints : list<integer>, yPoints : list<integer>, points : integer) : reference<Polygon>`.

- `Image` – This type allows to load images into a GUI via its path. In addition to the size attributes, users can access and change the transparency of the image. The corresponding creation operation is `createImage(path : string) : reference<Image>`.

- `Label` – This type represents simple text that can not be edited directly in the GUI without using setter operations, which allow to set the text itself, the text alignment, size and font. The stored string can be accessed by a getter operation. The corresponding creation operation is `createLabel(text : string) : reference<Label>`.

- `TextField` – This type represents a single line text field. Users have getter operations for the text and whether the text field is editable. Users can rely on setter operations to change the text, its font, the background colour and disable or enable the ability to edit the text in the GUI. The corresponding creation operation is `createTextField(alignment : GUIFactory.HorizontalAlignment, text : string) : reference<TextField>`.

- `TextArea` – This type is similar to `TextField`, the only difference is that the text can be displayed over several lines. The corresponding creation operation is `createTextArea(text : string) : reference<TextArea>`.

- `Button` – The button is the only control element that can perform actions when activated. The performed action is defined as redefined operation in a user defined subtype of `ActionTask` (see next subsection), which is passed to a button via the operation `handleActionTask`. Other than that, users can access and change the text of a button, activate or deactivate the button as well as check if it is active. The corresponding creation operation is `createButton(text : string) : reference<Button>`.

- `Checkbox` – Users may check if the check box is selected via the operation `isSelected`. The corresponding creation operation is `createCheckBox(alignment : GUIFactory.CheckBoxAlignment, text : string, selected : boolean) : reference<CheckBox>`.

- `DropDownMenu` – Users may access the selected string in the drop down menu via the operation `getSelectedItem`. The corresponding creation operation is `createDropDownMenu(itemNames : list<string>) : reference<DropDownMenu>`.

- `Slider` – Users may access the selected value in the slider via the operation `getValue`. The corresponding creation operation is `createSlider(minimum : integer, maximum : integer, value : integer) : reference<Slider>`.

`GUIFactoryTasks` Unit

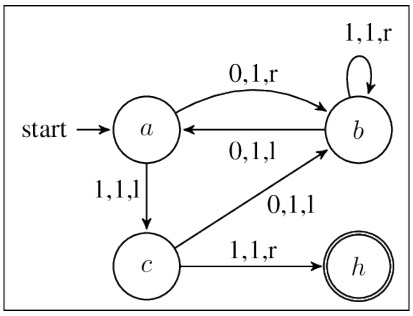
This unit includes the single abstract composition `ActionTask` which contains a single abstract operation `actionPerformed`. Users are meant to extend this type, provide their own implementation of the operation `actionPerformed` and pass instances of this type to `Buttons`.

J. Implementation of the Universal Turing Machine

In 1936, Alan Turing [124] has described theoretical computing machines, which were capable to solve any computable task. Such machines, which we now call *Turing Machines*, consist of a potentially unlimited tape divided into squares and a finite number of states q_1, ..., q_n. At any time, the machine can read from or write on a single square. The behaviour of the machine is defined by the conditions q_1, ..., q_n, which basically represent a finite state automaton. Depending on the symbol on the tape and the condition, the machine can move its head one square at a time to the left or right or stay in place. Furthermore it can replace or erase the current symbol on the tape. If a problem is computable, the *Turing Machine* will solve it given enough time. A computing machine which can solve any given computable task is called a Universal Turing Machine (UTM), any computational device that can solve any task that can be solved by such a machine is called Turing complete.

According to [7] any machine that is capable of data manipulation and supports conditional transfers is basically Turing complete. In practice, one has to consider hardware limitations. *Turing completeness* is used to demonstrate the expressive power of a computational machine or a programming language. Some examples are [125], [126] and [127]. Most general purpose programming languages like Java or Python are Turing complete while most markup languages like HTML or XML are not. This still does not tell us anything about the actual practicality of a language.

MuLE offers variables and assignments to store data, lists as a built-in data structure, conditional statements, loops, and supports recursion. Therefore, it offers more than enough language constructs to be considered Turing Complete. As a further proof we have implemented a UTM with MuLE heavily inspired by the Java implementation found at [128].This implementation of the *Universal Turing Machine* serves as a proof of Turing completeness of MuLE. Listing 150 shows a three state busy beaver machine running on our MuLE UTM. The initial tape is empty, 0 represents blank space, the goal is to fill the tape with as many 1s as possible and, eventually, halt. The corresponding state machine as well as the output of the program is seen in figure 56. In the output, the position of the machine is represented by `[H]` standing left from the current square on the tape.



```
[ [H] 0 ] --- (a, 0)=>(b, 1)/1
[ 1 [H] 0 ] --- (b, 0)=>(a, 1)/-1
[ [H] 1 1 ] --- (a, 1)=>(c, 1)/-1
[ [H] 0 1 1 ] --- (c, 0)=>(b, 1)/-1
[ [H] 0 1 1 1 ] --- (b, 0)=>(a, 1)/-1
[ [H] 0 1 1 1 1 ] --- (a, 0)=>(b, 1)/1
[ 1 [H] 1 1 1 1 ] --- (b, 1)=>(b, 1)/1
[ 1 1 [H] 1 1 1 ] --- (b, 1)=>(b, 1)/1
[ 1 1 1 [H] 1 1 ] --- (b, 1)=>(b, 1)/1
[ 1 1 1 1 [H] 1 ] --- (b, 1)=>(b, 1)/1
[ 1 1 1 1 1 [H] 0 ] --- (b, 0)=>(a, 1)/-1
[ 1 1 1 1 [H] 1 1 ] --- (a, 1)=>(c, 1)/-1
[ 1 1 1 [H] 1 1 1 ] --- (c, 1)=>(halt, 1)/0
[ 1 1 1 [H] 1 1 1 ] --- (halt, 1)
Terminal state reached.
Tape: [1, 1, 1, 1, 1, 1]
```

Figure 56: A three state busy beaver machine [127] and the output of the program shown in listing 150.

```
library TuringMachine
import IO
import Strings
import Lists

type Transitions : composition
   attribute stps : list<StateTapeSymbolPair>
   attribute transitions : list<Transition>

   operation add(parameter transition : Transition)
      if (Lists.contains(stps, transition.from)) then
         variable index : integer
         index := Lists.indexOf(stps, transition.from)
         transitions[index] := transition
      else
         stps := Lists.append(stps, transition.from)
         transitions := Lists.append(transitions, transition)
      endif
   endoperation

   operation get(parameter stp : StateTapeSymbolPair) : Transition
      variable index : integer
      index := Lists.indexOf(stps, stp)
      return transitions[index]
   endoperation

   operation containsStateSymbolPair(
         parameter stp : StateTapeSymbolPair) : boolean
      return Lists.contains(stps, stp)
   endoperation
endtype

type StateTapeSymbolPair : composition
   attribute state : string
   attribute tapeSymbol : string

   operation print()
      IO.writeString("(" & state & ", " & tapeSymbol & ")")
   endoperation
endtype

type Transition : composition
   attribute from : StateTapeSymbolPair
   attribute to : StateTapeSymbolPair
   attribute direction : integer // -1 left, 0 neutral, 1 right.

   operation print()
```

```
            from.print() IO.writeString("=>") to.print()
            IO.writeString("/") IO.writeInteger(direction)
        endoperation
    endtype

    type UniversalTuringMachine : composition
        attribute tape : list<string>
        attribute blankSymbol : string
        attribute headPosition : integer
        attribute transitions : Transitions
        attribute terminalStates : list<string>
        attribute initialState : string

        operation run()
            if Lists.lengthOf(tape) = 0 then
                tape := Lists.append(tape, blankSymbol)
            endif

            variable tsp : StateTapeSymbolPair
            tsp := StateTapeSymbolPair{state = initialState,
                                       tapeSymbol = tape[headPosition]}

            loop
                if transitions.containsStateSymbolPair(tsp) then
                    print() IO.writeString(" --- ")
                    transitions.get(tsp).print() IO.writeLine()

                    variable t : Transition
                    t := transitions.get(tsp)
                    tape[headPosition] := t.to.tapeSymbol

                    tsp.state := t.to.state

                    if t.direction = -1 then
                        if headPosition = 0 then
                            tape := Lists.prepend(tape, blankSymbol)
                        else
                            headPosition := headPosition - 1
                        endif
                        tsp.tapeSymbol := tape[headPosition]
                    elseif t.direction = 1 then
                        if headPosition = Lists.lengthOf(tape)-1 then
                            tape := Lists.append(tape, blankSymbol)
                        endif
                        headPosition := headPosition + 1
                        tsp.tapeSymbol := tape[headPosition]
                    else
                    tsp.tapeSymbol := t.to.tapeSymbol
                endif
            else
                exit
            endif
        endloop

        print() IO.writeString(" --- ") tsp.print() IO.writeLine()

        if Lists.contains(terminalStates, tsp.state) then
            IO.writeString("Terminal state reached.\n")
            IO.writeString("Tape: " & Strings.genericToString(tape) & "\n")
        endif
    endoperation

    operation print()
        IO.writeString("[ ")
        variable pos : integer
        pos := headPosition - 1

        variable i : integer
        loop
            if i > pos then exit endif
            IO.writeString(tape[i] & " ")
            i := i + 1
        endloop
        IO.writeString("[H] ")
        i := pos + 1
        loop
            if i >= Lists.lengthOf(tape) then exit endif
            IO.writeString(tape[i] & " ")
            i := i + 1
        endloop
        IO.writeString("]")
    endoperation
endtype
```

Listing 149: Universal Turing Machine implemented with MuLE.

```
program BusyBeaverTest
import TuringMachine

main
   variable transitions : TuringMachine.Transitions
   transitions.add(TuringMachine.Transition {
      from = TuringMachine.StateTapeSymbolPair{state = "a", tapeSymbol = "0"},
      to = TuringMachine.StateTapeSymbolPair{state = "b", tapeSymbol = "1"},
      direction = 1
   })
   transitions.add(TuringMachine.Transition {
      from = TuringMachine.StateTapeSymbolPair{state = "a", tapeSymbol = "1"},
      to = TuringMachine.StateTapeSymbolPair{state = "c", tapeSymbol = "1"},
      direction = -1
   })
   transitions.add(TuringMachine.Transition {
      from = TuringMachine.StateTapeSymbolPair{state = "b", tapeSymbol = "0"},
      to = TuringMachine.StateTapeSymbolPair{state = "a", tapeSymbol = "1"},
      direction = -1
   })
   transitions.add(TuringMachine.Transition {
      from = TuringMachine.StateTapeSymbolPair{state = "b", tapeSymbol = "1"},
      to = TuringMachine.StateTapeSymbolPair{state = "b", tapeSymbol = "1"},
      direction = 1
   })
   transitions.add(TuringMachine.Transition {
      from = TuringMachine.StateTapeSymbolPair{state = "c", tapeSymbol = "0"},
      to = TuringMachine.StateTapeSymbolPair{state = "b", tapeSymbol = "1"},
      direction = -1
   })
   transitions.add(TuringMachine.Transition {
      from = TuringMachine.StateTapeSymbolPair{state = "c", tapeSymbol = "1"},
      to = TuringMachine.StateTapeSymbolPair{state = "halt", tapeSymbol = "1"},
      direction = 0
   })
   variable utm : TuringMachine.UniversalTuringMachine
   utm := TuringMachine.UniversalTuringMachine {
      tape          = [],
      blankSymbol   = "0",
      headPosition  = 0,
      transitions   = transitions,
      terminalStates  = ["halt"],
      initialState  = "a"
   }
   utm.run()
endmain
```

Listing 150: Busy beaver running on the UTM implementation in listing 149.

List of Figures

List of Listings

List of Tables

Referenced Books

[4] L Bauer Friedrich and H Wössner. *Algorithmische Sprache und Programmentwicklung*. Germany: Springer-Verlag Berlin Heidelberg GmbH, 1984. ISBN: 978-3-662-05654-7.

[5] David Harel. *Algorithmics: The Spirit of Computing*. USA: Addison-Wesley Longman Publishing Co., Inc., 1987. ISBN: 0201192403.

[6] Michael L. Scott. *Programming Language Pragmatics*. 4th ed. Amsterdam: Morgan Kaufmann, 2016. ISBN: 978-0-12-410409-9.

[11] Jeanne C Adams et al. *Fortran 90 Handbook*. McGraw-Hill New York, 1992. ISBN: 978-0070004061.

[15] Kathleen Jensen and Niklaus Wirth. *PASCAL user manual and report: ISO PASCAL standard*. Springer Science & Business Media, 2012. ISBN: 978-0-387-97649-5.

[18] Ole-Johan Dahl and Kristen Nygaard. *Encyclopedia of Computer Science, Simula*. GBR: John Wiley and Sons Ltd., 2003, pp. 1576–1578. ISBN: 0470864125.

[20] Adele Goldberg and Alan Kay. *Smalltalk-72: Instruction Manual*. Xerox Corporation Palo Alto, 1976.

[21] Stanley B. Lippman, Jose Lajoie, and Barbara E. Moo. *C++ Primer*. 5th. Addison-Wesley Professional, 2012. ISBN: 978-0321714114.

[22] Ken Arnold and James Gosling. *The Java programming language*. Addison-Wesley Longman, 2005. ISBN: 978-0321349804.

[23] Robert W. Sebesta. *Concepts of Programming Languages*. 11th. Pearson, 2016. ISBN: 978-0133943023.

[26] Eric Freeman et al. *Head First Design Patterns*. O'Reilly Media, Inc., 2004. ISBN: 9780596007126.

[28] Edmund Callis Berkeley and Daniel Gureasko Bobrow. *The programming language LISP: Its operation and applications*. The MIT Press 1966, 1966. ISBN: 9780262590044.

[29] Alonzo Church. *The calculi of lambda-conversion*. 6. Princeton University Press, 1985. ISBN: 978-0691083940.

[30] Peter Norvig. *Paradigms of artificial intelligence programming: case studies in Common LISP*. Morgan Kaufmann, 1992. ISBN: 1-55860-191-0.

[31] R Kent Dybvig. *The Scheme programming language*. The MIT Press, 2009. ISBN: 978-0262512985.

[32] Stuart Dabbs Halloway and Aaron Bedra. *Programming Clojure*. The Pragmatic Bookshelf, 2012. ISBN: 978-1934356869.

[33] Tomas Petricek and Jon Skeet. *Real World Functional Programming*. 2009. ISBN: 9781933988924.

[37] William F Clocksin and Christopher S Mellish. *Programming in Prolog: Using the ISO standard*. Springer Science & Business Media, 2012. ISBN: 3642554814, 9783642554810.

[43] Rüdiger Baumann. *Didaktik der Informatik*. Klett-Verlag, 1996. ISBN: 978-3129850107.

[44] Peter Hubwieser. *Didaktik der Informatik: Grundlagen, Konzepte, Beispiele*. Springer-Verlag, 2000. ISBN: 978-3-540-65564-0.

[45] Association for Computing Machinery (ACM) Joint Task Force on Computing Curricula and IEEE Computer Society. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. New York, NY, USA: Association for Computing Machinery, 2013. ISBN: 9781450323093.

[52] Peter Gasston. *The modern Web: multi-device Web development with HTML5, CSS3, and JavaScript*. No Starch Press, 2013. ISBN: 978-1593274870.

[71] Martin Hitz et al. *UML@Work, Objektorientierte Modellierung mit UML 2*. dpunkt.verlag, 2005. ISBN: 3-89864-261-5.

[86] Lars Vogel. *Eclipse Rich Client Platform*. 2015. ISBN: 978-3943747133.

[106] Joseph Bergin et al. *Karel J. Robot: A gentle introduction to the art of object-oriented programming in Java*. Dream Songs Redwood City, 2005. ISBN: 0970579519, 9780970579515.

[110] Lorenzo Bettini. *Implementing Domain Specific Languages with Xtext and Xtend - Second Edition*. 2nd. Packt Publishing, 2016. ISBN: 1786464969, 9781786464965.

[111] Markus Voelter et al. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013. ISBN: 978-1-4812-1858-0.

[112] Richard C Gronback. *Eclipse modeling project: a domain-specific language (DSL) toolkit*. Pearson Education, 2009. ISBN: 978-0321534071.

[113] Thomas Stahl, Markus Völter, and Krzysztof Czarnecki. *Model-driven software development: technology, engineering, management*. John Wiley & Sons, Inc., 2006. ISBN: 978-0-470-02570-3.

[114] David Steinberg et al. *EMF: Eclipse Modeling Framework 2.0*. 2nd. Addison-Wesley Professional, 2009. ISBN: 0321331885.

[116] Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013. ISBN: 9781934356999.

Referenced Articles

[1] Yizhou Qian and James Lehman. “Students’ Misconceptions and Other Difficulties in Introductory Programming: A Literature Review”. In: *ACM Trans. Comput. Educ.* 18.1 (Oct. 2017), 1:1–1:24. ISSN: 1946-6226. DOI: 10.1145/3077618.

[2] John Pane, Chotirat Ratanamahatana, and Brad Myers. “Studying the language and structure in non-programmers’ solutions to programming problems”. In: *International Journal of Human Computer Studies* 54 (Oct. 2000), pp. 237–264. DOI: 10.1006/ijhc.2000.0410.

[7] Hao Wang. “A variant to Turing’s theory of computing machines”. In: *Journal of the ACM (JACM)* 4.1 (1957), pp. 63–92.

[8] Brent Hailpern. “Guest editor’s introduction multiparadigm languages and environments”. In: *IEEE Software* 3.1 (1986), p. 6.

[9] Peter Van Roy et al. “Programming paradigms for dummies: What every programmer should know”. In: *New computational paradigms for computer music* 104 (2009), pp. 616–621.

[10] Milena Vujošević-Janičić and Dušan Tošić. “The role of programming paradigms in the first programming courses”. In: *The Teaching of Mathematics* 21 (2008), pp. 63–83.

[12] Jean E Sammet. “Basic elements of COBOL 61”. In: *Communications of the ACM* 5.5 (1962), pp. 237–253.

[13] J. W. Backus et al. “Report on the Algorithmic Language ALGOL 60”. In: *Commun. ACM* 3.5 (May 1960), pp. 299–314. ISSN: 0001-0782. DOI: 10.1145/367236.367262.

[14] Edsger W. Dijkstra. “Letters to the Editor: Go to Statement Considered Harmful”. In: *Commun. ACM* 11.3 (Mar. 1968), pp. 147–148. ISSN: 0001-0782. DOI: 10.1145/362929.362947.

[16] Tim Rentsch. “Object oriented programming”. In: *ACM Sigplan Notices* 17.9 (1982), pp. 51–57.

[17] Ole-Johan Dahl and Kristen Nygaard. “SIMULA: An ALGOL-Based Simulation Language”. In: *Commun. ACM* 9.9 (Sept. 1966), pp. 671–678. ISSN: 0001-0782. DOI: 10.1145/365813.365819.

[24] Ghan Bir Singh. “Single versus Multiple Inheritance in Object Oriented Programming”. In: *SIGPLAN OOPS Mess.* 6.1 (Jan. 1995), pp. 30–39. ISSN: 1055-6400. DOI: 10.1145/209866.209871.

[25] Anders Hejlsberg and Scott Wiltamuth. “C# language reference”. In: (2000).

[35] Simon Marlow et al. “Haskell 2010 language report”. In: *Available on: https://www. haskell. org/onlinereport/haskell2010* (2010).

[38] Timothy A. Budd. “Blending Imperative and Relational Programming”. In: 8.1 (Jan. 1991), pp. 58–65. ISSN: 0740-7459. DOI: 10.1109/52.62933.

[39] P. A. Luker. “Never Mind the Language, What about the Paradigm?” In: 21.1 (Feb. 1989), pp. 252–256. ISSN: 0097-8418. DOI: 10.1145/65294.71442.

[50] Sang Joon Lee and Thomas C Reeves. “A significant contributor to the field of educational technology”. In: *Educational Technology* 47.6 (2007), pp. 56–59.

[51] Diomidis Spinellis. “Choosing a programming language”. In: *IEEE software* 23.4 (2006), pp. 62–63.

[60] Linda Mannila, Mia Peltomäki, and Tapio Salakoski. “What about a simple language? Analyzing the difficulties in learning to program”. In: *Computer Science Education* 16.3 (2006), pp. 211–227. DOI: 10.1080/08993400600912384.

[63] Uolevi Nikula et al. “Python and roles of variables in introductory programming: experiences from three educational institutions”. In: *Journal of Information Technology Education: Research* 6.1 (2007), pp. 199–214.

[64] Majed A. Sahli and Gordon W. Romney. “Agile Teaching: A case study of using Ruby to teach programming language concepts.” In: *Journal of Research in Innovative Teaching* 3 (2010), pp. 63–72.

[65] Michael Kölling. “The Problem of Teaching Object-Oriented Programming, Part I: Languages”. In: *JOOP* 11 (1999), pp. 8–15.

[68] Stephen Cooper, Wanda Dann, and Randy Pausch. “Alice: a 3-D tool for introductory programming concepts”. In: *Journal of computing sciences in colleges* 15.5 (2000), pp. 107–116.

[70] Martin C Carlisle et al. “RAPTOR: a visual programming environment for teaching algorithmic problem solving”. In: *Acm Sigcse Bulletin* 37.1 (2005), pp. 176–180.

[74] Roy D Pea. “Logo programming and problem solving”. In: (1987).

[80] Andreas Stefik and Susanna Siebert. “An Empirical Investigation into Programming Language Syntax”. In: *Trans. Comput. Educ.* 13.4 (Nov. 2013), 19:1–19:40. ISSN: 1946-6226. DOI: 10.1145/2534973.

[84] Michael Kölling. “The Problem of Teaching Object-Oriented Programming, Part II: Environments”. In: *JOOP* 11 (June 1999), pp. 6–12.

[85] Michael Kölling et al. “The BlueJ system and its pedagogy”. In: *Computer Science Education* 13.4 (2003), pp. 249–268.

[87] Zhixiong Chen and Delia Marx. “Experiences with Eclipse IDE in programming courses”. In: *Journal of Computing Sciences in Colleges* 21.2 (2005), pp. 104–112.

[89] BT Denvir. “On orthogonality in programming languages”. In: *ACM SIGPLAN Notices* 14.7 (1979), pp. 18–30.

[91] John F Pane and Brad A Myers. “Usability issues in the design of novice programming systems”. In: (1996).

[93] Brian Hayes. “Computing Science: The Semicolon Wars”. In: *American Scientist* 94.4 (2006), pp. 299–303.

[96] “IEEE Standard for Floating-Point Arithmetic”. In: *IEEE Std 754-2008* (Aug. 2008), pp. 1–70. DOI: 10.1109/IEEESTD.2008.4610935.

[103] David Goldberg. “What every computer scientist should know about floating-point arithmetic”. In: *ACM computing surveys (CSUR)* 23.1 (1991), pp. 5–48.

[117] Tom Briggs. “Techniques for active learning in CS courses”. In: *Journal of Computing Sciences in Colleges* 21.2 (2005), pp. 156–165.

[118] Richard E Mayer. “The psychology of how novices learn computer programming”. In: *ACM Computing Surveys (CSUR)* 13.1 (1981), pp. 121–141.

[120] Brian Hanks et al. “Pair programming in education: a literature review”. In: *Computer Science Education* 21.2 (2011), pp. 135–173. DOI: 10.1080/08993408.2011.579808.

[124] A. M. Turing. “On Computable Numbers, with an Application to the Entscheidungsproblem”. In: *Proceedings of the London Mathematical Society* s2-42.1 (Jan. 1937), pp. 230–265. ISSN: 0024-6115. DOI: 10.1112/plms/s2-42.1.230.

[127] Martin Gebser et al. “1 Potassco: The Potsdam Answer Set Solving Collection”. In: *AI Commun.* 24 (Jan. 2011), pp. 107–124. DOI: 10.3233/AIC-2011-0491.

Referenced Conference Proceedings

[19] Johan Dahl, BJorn Myhrhaug, and Kristen Nygaard. "Some features of the SIMULA 67 language". In: 1968.

[27] Paolo Boldi and Sebastiano Vigna. "Rethinking Java Strings". In: *ACM International Conference Proceeding Series*. Vol. 42. Citeseer. 2003, pp. 27–30.

[36] Seppo Keronen. "Non-procedural logic programming". In: *International Workshop on Extensions of Logic Programming*. Springer. 1993, pp. 183–195.

[41] Leila Goosen. "A brief history of choosing first programming languages". In: *IFIP International Conference on the History of Computing*. Springer. 2008, pp. 167–170.

[42] John E Howland. "IT'S ALL IN THE LANGUAGE (yet another look at the choice of programming language for teaching computer science)". In: *Journal of Computing in Small Colleges, Volume 12, Number 4*. Citeseer. 1997.

[48] Matthew Hertz. "What do CS1 and CS2 mean? Investigating differences in the early courses". In: *Proceedings of the 41st ACM technical symposium on Computer science education*. 2010, pp. 199–203.

[49] Randy M Kaplan. "Choosing a first programming language". In: *Proceedings of the 2010 ACM conference on Information technology education*. 2010, pp. 163–164.

[53] Yuliia Prokop et al. "An Analysis of Criteria for Choosing a First Programming Language in Universities." In: *ICTERI*. 2019, pp. 420–425.

[54] Linda McIver. "The effect of programming language on error rates of novice programmers." In: *PPIG*. Citeseer. 2000, p. 15.

[55] Vladyslav Kruglyk and Michael Lvov. "Choosing the first educational programming language". In: *ICT in Education, Research and Industrial Applications: Integration, Harmonization and Knowledge Transfer: Proceedings of the 8th International Conference ICTERI 2012*. Kherson. 2012, pp. 188–189.

[58] Andrew Black, Kim B. Bruce, and James Noble. "Panel: Designing the Next Educational Programming Language". In: *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*. OOPSLA '10. Reno/-Tahoe, Nevada, USA: ACM, 2010, pp. 201–204. ISBN: 978-1-4503-0240-1. DOI: 10.1145/1869542.1869574.

[59] Linda Grandell et al. “Why complicate things? Introducing programming in high school using Python”. In: *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52.* 2006, pp. 71–80.

[66] F. F. de Vega. “To Be, or Not To Be: That is the Recursive Question”. In: *2019 IEEE Global Engineering Education Conference (EDUCON).* Apr. 2019, pp. 1294–1299. DOI: 10.1109/EDUCON.2019.8725191.

[67] J. Maloney et al. “Scratch: a sneak preview [education]”. In: *Proceedings. Second International Conference on Creating, Connecting and Collaborating through Computing, 2004.* Jan. 2004, pp. 104–109. DOI: 10.1109/C5.2004.1314376.

[69] Neil Fraser. “Ten things we’ve learned from Blockly”. In: *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond).* IEEE. 2015, pp. 49–50.

[72] Paul Gross and Kris Powers. “Evaluating assessments of novice programming environments”. In: *Proceedings of the first international workshop on Computing education research.* 2005, pp. 99–110.

[73] Thomas W Price and Tiffany Barnes. “Comparing textual and block interfaces in a novice programming environment”. In: *Proceedings of the eleventh annual international conference on international computing education research.* 2015, pp. 91–99.

[75] Michael E. Caspersen and Henrik Bærbak Christensen. “Here, there and everywhere - on the recurring use of turtle graphics in CS1”. In: *ACSE.* 2000.

[76] Andrew P. Black et al. “Seeking Grace: A New Object-oriented Language for Novices”. In: *Proceeding of the 44th ACM Technical Symposium on Computer Science Education.* SIGCSE ’13. Denver, Colorado, USA: ACM, 2013, pp. 129–134. ISBN: 978-1-4503-1868-6. DOI: 10.1145/2445196.2445240.

[78] Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. “All syntax errors are not equal”. In: *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education.* 2012, pp. 75–80.

[79] Andreas Stefik and Richard Ladner. “The Quorum Programming Language (Abstract Only)”. In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education.* SIGCSE ’17. Seattle, Washington, USA: ACM, 2017, pp. 641–641. ISBN: 978-1-4503-4698-6. DOI: 10.1145/3017680.3022377.

[90] Linda McIver and Damian Conway. “Seven Deadly Sins of Introductory Programming Language Design”. In: *Proceedings of the 1996 International Conference on Software Engineering: Education and Practice (SE:EP ’96)*. SEEP ’96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 309–. ISBN: 0-8186-7379-6.

[92] Leila Goosen, Elsa Mentz, and Hercules Nieuwoudt. “Choosing the “best” programming language”. In: *Proceedings of the computer science and IT education conference*. 2007, pp. 269–282.

[95] Amjad Altadmri and Neil CC Brown. “37 million compilations: Investigating novice programming mistakes in large-scale student data”. In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. 2015, pp. 522–527.

[97] Lisa C Kaczmarczyk et al. “Identifying student misconceptions of programming”. In: *Proceedings of the 41st ACM technical symposium on Computer science education*. 2010, pp. 107–111.

[119] Roger Duke et al. “Teaching programming to beginners-choosing the language is just the first step”. In: *Proceedings of the Australasian conference on Computing education*. 2000, pp. 79–86.

[125] R. Boyer and J. Moore. “A Mechanical Proof of the Turing Completeness of Pure LISP.” In: 1983.

[126] J Strother Moore. “Proof Pearl: Proving a Simple Von Neumann Machine Turing Complete”. In: *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*. Ed. by Gerwin Klein and Ruben Gamboa. Vol. 8558. Lecture Notes in Computer Science. Springer, 2014, pp. 406–420. DOI: 10.1007/978-3-319-08970-6_26.

Referenced Online Sources

[3] Bundesministerium für Bildung und Forschung. *Qualitätsoffensive Lehrerbildung*. URL: https://www.qualitaetsoffensive-lehrerbildung.de/lehrerbildung/de/home/home_node.html (visited on 11/05/2021).

[34] Graham Hutton. *Frequently Asked Questions for comp.lang.functional*. 2002. URL: http://www.cs.nott.ac.uk/~pszgmh/faq.html (visited on 08/20/2020).

[40] TIOBE. *TIOBE Index*. URL: https://www.tiobe.com/tiobe-index/ (visited on 05/04/2021).

[46] University of Bayreuth. *Computer science module manual*. 2021. URL: https://www.ai.uni-bayreuth.de/pool/dokumente/MHB_Informatik/MHB_Informatik_2021_04_02.pdf (visited on 05/06/2021).

[47] Technical University of Munich. *Computer science module manual*. 2021. URL: https://www.in.tum.de/fuer-studierende/module-und-veranstaltungen/modulkatalog/ (visited on 05/06/2021).

[56] James Gosling et al. *The Java Language Specification, Java SE 12 Edition*. 2019. URL: https://docs.oracle.com/javase/specs/jls/se12/jls12.pdf (visited on 08/20/2020).

[57] Barry D. Bowen. *Educators embrace Java*. 1997. URL: https://www.infoworld.com/article/2076867/educators-embrace-java.html (visited on 05/19/2021).

[61] Tim Peters. *The Zen of Python*. URL: https://www.python.org/dev/peps/pep-0020/ (visited on 05/21/2021).

[77] PSU Grace Team. *Grace Documentation*. URL: http://web.cecs.pdx.edu/~grace/doc/variables/methods/ (visited on 05/27/2021).

[81] Benjamin Lerner and Joe Gibbs Politz. *Pyret Programming Language Documentation*. URL: https://www.pyret.org/docs/latest/ (visited on 05/26/2021).

[82] Svetlana Drachova-Strang. *A RESOLVE Primer*. URL: https://www.cs.clemson.edu/resolve/teaching/tutor/manual/ResolveManual.pdf (visited on 05/27/2021).

[83] Don Ho. *Notepad++*. URL: https://notepad-plus-plus.org/ (visited on 05/31/2021).

[94] Benjamin Hummel. *Save the Semicolon*. URL: https://www.cqse.eu/en/news/blog/save-the-semicolon/ (visited on 06/21/2021).

[100] *Xtext - The Grammar Language.* URL: https://www.eclipse.org/Xtext/documentation/301_grammarlanguage.html (visited on 07/16/2021).

[101] *CP1252 - Windows 1252.* URL: http://www.cp1252.com/ (visited on 07/07/2021).

[102] Ali Dehghani. *Memory Address of Objects in Java.* URL: https://www.baeldung.com/java-object-memory-address (visited on 08/05/2021).

[107] *Package javax.swing.* URL: https://cs.lmu.edu/~ray/notes/paradigms/ (visited on 09/23/2021).

[108] *JavaFX API documentation.* URL: https://openjfx.io/javadoc/17/ (visited on 09/23/2021).

[115] Object Management Group. *About The XML Metadata Interchange Specification Version 2.5.1.* URL: https://www.omg.org/spec/XMI/ (visited on 09/05/2021).

[121] Oracle. *Java SE at a Glance.* URL: https://www.oracle.com/java/technologies/java-se-glance.html (visited on 09/30/2021).

[122] Oracle. *OpenJDK.* URL: https://openjdk.java.net/ (visited on 09/30/2021).

[123] Eclipse Foundation. *Eclipse IDE 2021-09 R Packages.* URL: https://www.eclipse.org/downloads/packages/ (visited on 09/30/2021).

[128] *Universal Turing machine.* URL: https://rosettacode.org/wiki/Universal_Turing_machine (visited on 02/16/2021).

Students' Contributions

[104] Stefan Schill. *Java implementation of a Turtle library*. 2020.

[105] Marco Jantos. *Java implementation of a programming microworlds library*. 2020.

[109] Johannes Glier. *Java implementation of a GUI library*. 2020.

Own Publications

[62] Nikita Dümmel, Bernhard Westfechtel, and Matthias Ehmann. *Effects of a Preliminary Programming Course on Students' Performance*. European Conference of Software Engineering Education (ECSEE), ACM, 2018. DOI: 10.1145/3209087.3209088.

[88] Nikita Dümmel, Bernhard Westfechtel, and Matthias Ehmann. *Work in Progress: Gathering Requirements and Developing an Educational Programming Language*. Global Engineering Education Conference (EDUCON), IEEE, 2019. DOI: 10.1109/EDUCON.2019.8725073.

[98] Nikita Dümmel, Bernhard Westfechtel, and Matthias Ehmann. *MuLE– a Multiparadigm Language for Education. The Procedural Sublanguage*. Global Engineering Education Conference (EDUCON), IEEE, 2020. DOI: 10.1109/EDUCON45650.2020.9125327.

[99] Nikita Dümmel, Bernhard Westfechtel, and Matthias Ehmann. *MuLE: a Multiparadigm Language for Education. The Object-Oriented Part of the Language*. European Conference of Software Engineering Education (ECSEE), ACM, 2020. DOI: 10.1145/3396802.3396806.

[129] Nikita Dümmel, Bernhard Westfechtel, and Matthias Ehmann. *A Multi-Paradigm Programming Language for Education*. Informatics in Education, Vilnius University, ETH Zürich, submitted for publication.

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die von mir angegebenen Quellen und Hilfsmittel verwendet habe.

Weiterhin erkläre ich, dass ich die Hilfe von gewerblichen Promotionsberatern bzw. –vermittlern oder ähnlichen Dienstleistern weder bisher in Anspruch genommen habe, noch künftig in Anspruch nehmen werde.

Zusätzlich erkläre ich hiermit, dass ich keinerlei frühere Promotionsversuche unternommen habe.

Bayreuth, den

Unterschrift