1. Einleitung

Jedes Jahr werden zahlreiche neue Programmiersprachen veröffentlicht. Viele von ihnen sind ursprünglich Teil einer wissenschaftlichen Arbeit, und sowohl Privatpersonen als auch Studierende arbeiten in ihrer Freizeit häufig an ihren eigenen Sprachen, um verschiedene Aspekte der Implementierung kennenzulernen. In manchen Fällen möchten Unternehmen oder Organisationen ein bestimmtes Problem angehen, aber es fehlt noch die richtige Technologie. Auch wenn die meisten von ihnen vom durchschnittlichen Entwickler nie benutzt werden, können Sie durch das Realisieren neuer Ideen alle Programmiersprachen voranbringen.

Eine neue Sprache hat den Vorteil, dass ihre Entwicklung nicht durch interne Uneinigkeiten behindert wird, wie dies bei vielen älteren Programmiersprachen der Fall ist. Da man bei der Implementierung von Null anfangen kann, aber konzeptionell alles Wissen aus früheren Versuchen genutzt wird, können völlig neue Ideen viel einfacher und schneller ausprobiert werden. Bei einer Sprache, die von Tausenden von Menschen benutzt wird und möglichst stabil bleiben soll, ist dieser Prozess schwieriger.

Ein weiterer Vorteil kleiner neuer Projekte ist, dass es weniger Bürokratie und Koordination gibt, was es ermöglicht, viele Konzepte schnell und iterativ auszuprobieren. Die Koordination ist bei größeren Projekten essentiell, da viele Leute gleichzeitig an ihnen arbeiten und es sonst kaum Fortschritte geben würde.

Wenn die daraus resultierenden Ansätze der „Neulinge” sinnvoll erscheinen und an anderen Stellen übernommen werden, führt dies oft zu einer Integration in etabliertere Sprachen. Genau eine solche Sprache wird im Rahmen dieser Arbeit entworfen und umgesetzt, indem allgemein vorteilhafte Eigenschaften und bewährte Technologien analysiert und mit neuen Ideen kombiniert werden.

Diese Arbeit befasst sich daher mit der Analyse verschiedener Programmiersprachen, um herauszufinden, in welche Richtung sie sich entwickeln. Aus den daraus gewonnenen Erkenntnissen soll eine neue Sprache entworfen werden, die eine mögliche Antwort auf die Frage geben soll: *Wie könnte eine Programmiersprache der nächsten Generation aussehen?*

2. Vorteilhafte Eigenschaften von Programmiersprachen

Entscheidend für das Erreichen dieses Ziels ist es, herauszufinden, welche Eigenschaften einer Programmiersprache vorteilhaft sind und welche nicht. Diese werden dann bewertet, indem man sich darauf konzentriert, wie sie den Softwareentwicklungsprozess, einschließlich der Wartung, beeinflussen. Eine solche Liste von Merkmalen ist zwangsläufig umstritten, da es schwierig ist, alle Beteiligten dazu zu bringen, sich über den Wert einer bestimmten Spracheigenschaft im Vergleich mit anderen einig zu sein. Trotz dieser Unterschiede würden die meisten wahrscheinlich zustimmen, dass die folgenden Kriterien zumindest wichtig sind. [1, S. 60], [2, S. 41]

2.1 Lesbarkeit

Ein wichtiges Kriterium für die Beurteilung einer Programmiersprache ist die Leichtigkeit, mit der Programme gelesen und verstanden werden können. Der in unserer neuen Sprache geschriebene Quellcode wird möglicherweise öfter gelesen als geschrieben und die Wartung, bei der viel gelesen wird, ist ein wichtiger Teil des Entwicklungszyklus. Da die Wartungsfreundlichkeit zu einem großen Teil durch die Lesbarkeit von Programmen bestimmt wird, ist sie eine wichtige Maßnahme zur Verbesserung der Qualität von Programmen und Programmiersprachen. In den folgenden Unterabschnitten werden Merkmale beschrieben, die zur Lesbarkeit einer Programmiersprache beitragen. [2, S. 42f]

2.1.1 Einfachheit

Die allgemeine Einfachheit einer Programmiersprache wirkt sich stark auf ihre Lesbarkeit aus. Eine Sprache mit einer großen Anzahl von Grundkonstrukten ist schwieriger zu verstehen als eine einfachere Sprache. Programmierer, die eine komplexe Sprache verwenden müssen, lernen oft nur eine Teilmenge der Sprache und ignorieren die anderen Merkmale. Dieses Lernmuster wird manchmal als Entschuldigung für die große Anzahl von Strukturen herangezogen, aber dieses Argument ist nicht stichhaltig. Probleme mit der Lesbarkeit treten immer dann auf, wenn der Programmverfasser eine andere Teilmenge gelernt hat als die, mit der der Leser vertraut ist. [1, S. 61], [2, S. 43]

Ein zweites Merkmal, das die Lesbarkeit einer Programmiersprache beeinträchtigt, ist die Existenz von mehr als einem Weg, um dieselbe Operation auszuführen. In Kapitel 4.7 wird ein Beispiel dafür näher erläutert. Dieses Beispiel erfordert die Überladung von Operatoren in Rym, bei dem ein einzelner Operator mehr als eine Implementierung haben kann. Obwohl dies oft nützlich ist, kann es zu einer geringeren Lesbarkeit führen, wenn es den Benutzern erlaubt ist, ihre eigenen Überladungen zu erstellen und sie dies nicht vernünftig tun. So ist es beispielsweise durchaus akzeptabel, + zu überladen, um es sowohl für die Ganzzahl- als

auch für die Gleitkommaaddition zu verwenden. In der Tat vereinfacht diese Überladung eine Sprache, indem sie die Anzahl der verschiedenen Operatoren reduziert. Wie das Überladen in Rym funktioniert, wird in Kapitel 8 erklärt. [2, S. 43f]

Andererseits kann man es mit der Einfachheit auch übertreiben. Zum Beispiel sind Form und Bedeutung der meisten Assembler-Anweisungen sehr einfach, aber da komplexere Steueranweisungen fehlen, ist die Programmstruktur weniger offensichtlich und es sind mehr Anweisungen erforderlich als in entsprechenden Programmen in einer höheren Sprache, ist die Programmstruktur weniger offensichtlich und es werden mehr Anweisungen benötigt als in entsprechenden Programmen in einer höheren Sprache. Dieselben Argumente gelten auch für den weniger extremen Fall von Hochsprachen mit unzureichenden Kontroll- und Datenstrukturen. [2, S. 44]

- TODO
  - as few things to think about at once as possible

2.1.2 Orthogonalität

Orthogonalität in einer Programmiersprache bedeutet, dass eine kleine Menge von primitiven Konstrukten auf wenige Arten kombiniert werden können, um die Kontroll- und Datenstrukturen der Sprache aufzubauen. Außerdem sollte jede mögliche Kombination von Primitiven legal und sinnvoll sein. Ein Mangel an Orthogonalität führt daher zu Ausnahmen von den Regeln der Sprache. Je orthogonaler der Entwurf einer Sprache ist, desto weniger Ausnahmen erfordern die Sprachregeln. Weniger Ausnahmen bedeuten einen höheren Grad an Regelmäßigkeit im Design, wodurch die Sprache leichter zu lernen, zu lesen und zu verstehen ist. [2, S. 44ff]

Die Bedeutung eines orthogonalen[1] Sprachelements ist unabhängig von dem Kontext, in dem es in einem Programm auftritt. Aber alles kontextunabhängig zu machen, kann auch zu unnötiger Komplexität führen. Da Sprachen eine große Anzahl von Primitiven benötigen, führt ein hoher Grad an Orthogonalität zu einer Explosion an Kombinationen. Selbst wenn die Kombinationen einfach sind, führt ihre schiere Anzahl zu Komplexität. Die Einfachheit einer Sprache ist daher zumindest teilweise das Ergebnis einer Kombination aus einer relativ kleinen Anzahl von primitiven Konstrukten und einer begrenzten Verwendung des Konzepts der Orthogonalität. [1, S. 60f], [2, S. 46f]

2.1.3 Datentypen

Readability can be increased even more by allowing users to define adequate data types and data structures for specific use cases. This especially applies to Booleans and

[1] Das Wort orthogonal stammt aus dem mathematischen Konzept der orthogonalen Vektoren, die voneinander unabhängig sind.

Enumerations covered in Kapitel 5.1 and Kapitel 6.2.1 respectively. For example, in some languages that do not provide a Boolean type the following code could be used:

```
const timeout = 1
```

The meaning of this statement is unclear, whereas in a language that includes Boolean types, the following would be used:

```
const timeout = true
```

Which is is a clearer then the first one. [2, S. 47]

2.1.4 Syntax

[2, S. 48f]

- TODO
  - simple
  - expressive
  - flat learning curve
  - familiar syntax
  - `{}` is a block (vs. `end`/`end if`)
  - might not make as much sense for non programmers
  - comparable with the way that + and other math notation could be considered less readable then `plus`

2.2 Schreibbarkeit

Writability is a measure of how easily a language can be used to create programs for a chosen problem domain. Most of the language characteristics that affect readability also affect writability. This follows directly from the fact that the process of writing a program requires the programmer frequently to reread the part of the program that is already written. As is the case with readability, writability must be considered in the context of the target problem domain of a language.

2.3 Verlässlichkeit

A program is said to be reliable if it performs to its specifications under all conditions and will continue to in the future. [2, S. 51]

2.3.1 Spezifikation

A programming language without a standardised specification can hardly be considered reliable. This specification is a set of rules that an implementation of the language must obey

to ensure that programs behave as intended. It should cover all possible combinations of programming language features.

In some cases, the behaviour of certain features may be poorly defined, either to make programs run optimally on certain platforms, or because no one has thought of the exact combination. This can lead to different implementations that produce different results. Some of the possible outcomes can be harmful and implementations are not required to remain consistent across executions, implementation versions, or levels of optimisation. [3, S. 21], [4, S. 190]

Behaviours that are not mentioned in the specification are called *unspecified behaviour*, and *undefined behaviour* if they are mentioned, but it is not defined how they behave. They pose a significant challenge to users of the language, as they cannot be sure how and whether their program will run. For this reason, a language should have as few of them as possible. [3, S. 21]

2.3.2 Sicherheit

- TODO
  - aliasing
  - type checking

2.3.3 Zukunftssicherheit

- TODO
  - allow breaking changes
  - reserved keywords

3. Analyse bestehender Programmiersprachen

Der nächste Schritt auf der Suche nach einer Antwort besteht darin, die derzeit verwendeten Programmiersprachen zu untersuchen, insbesondere solche, die erst vor kurzem erschienen sind, und herauszufinden, was sie im Vergleich zu älteren Sprachen geändert haben.

3.1 Zu analysierende Daten

Um die Zahl der Sprachen in einem vernünftigen Rahmen zu halten, wurden nur die beliebtesten Programmiersprachen des Jahres 2022 analysiert. Die Hauptquelle für diese Daten ist der *StackOverflow Developer Survey 2022*, eine weltweite Umfrage, die seit 2011 jährlich durchgeführt wird und sich an alle richtet, die programmieren. Unterhalb dieses Absatzes sehen Sie eine Statistik aus der Umfrage, in der die Programmiersprachen nach ihrer Beliebtheit geordnet sind. Die Teilnehmer wurden gebeten, für alle Sprachen zu stimmen, die sie im letzten Jahr verwendet haben und wieder verwenden würden. Die Höhe eines Balkens entspricht der Anzahl der Stimmen, die diese Sprache erhalten hat. [5]–[7]

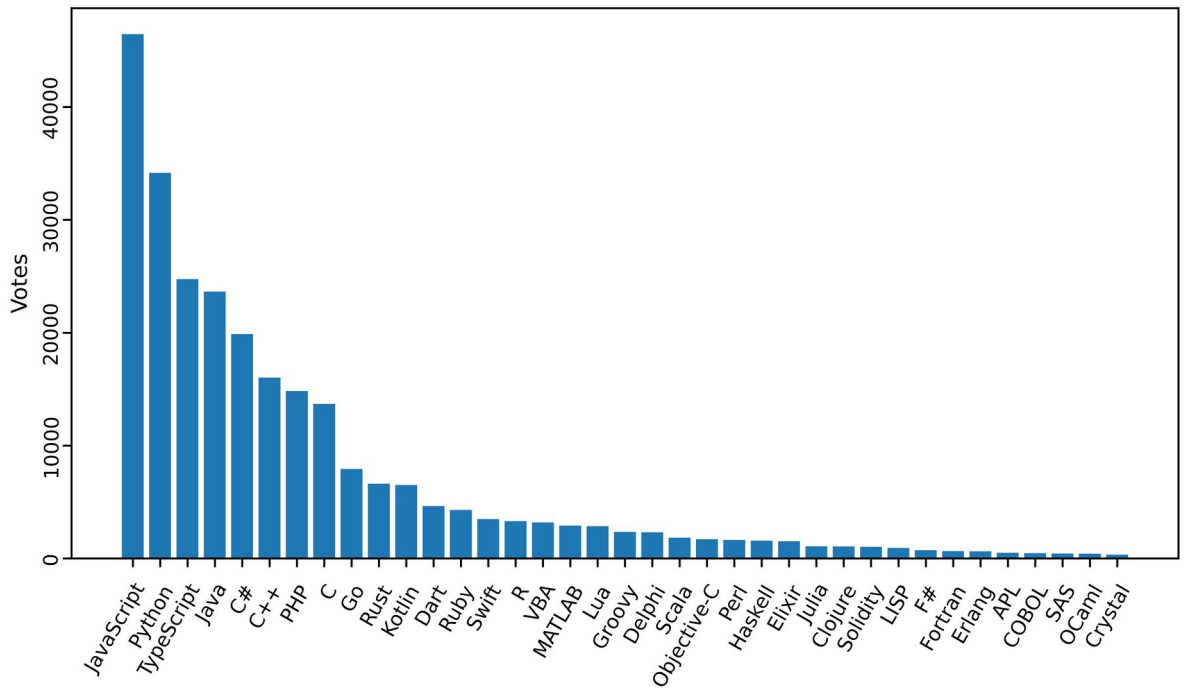


Abbildung 3.1: Die beliebtesten Programmiersprachen im Jahr 2022

In den ursprünglichen Daten werden auch HTML, CSS, SQL, Bash/Shell und PowerShell erwähnt. Diese werden hier nicht berücksichtigt, da sie hochspezialisiert und teilweise nicht Turing-vollständig *(engl. Turing-complete)* sind.[2]

3.2 Neuere Programmiersprachen

Alle neueren Programmiersprachen, die in den Statistiken aufgeführt werden, sind von anderen Sprachen inspiriert oder sogar deren Nachfolger. Zu diesen Sprachen gehört z.B. Kotlin von JetBrains[3] als Nachfolger von Java. Kotlin soll kompakter, ausdrucksstärker und sicherer sein, aber weiterhin auf der Java Virtual Machine laufen und Interoperabilität mit Java bieten. Groovy von Apache[4], Closure und Scala sind weitere Sprachen, die auf der JVM basieren, aber nicht so populär geworden sind wie Kotlin. Der vielleicht größte Faktor für den Erfolg war die offizielle Unterstützung von Google für Android-Entwicklung mit Kotlin. [9, S. 867f], [12]–[22]

In den letzten Jahren hat sich Rust zu einer sehr beliebten Programmiersprache entwickelt. Ursprünglich von Graydon Hoare bei Mozilla entwickelt, ist Rust heute ein unabhängiges Gemeinschaftsprojekt. Die Sprache ist für die Systemprogrammierung gedacht und kombiniert Sicherheit und Geschwindigkeit. Der Schwerpunkt liegt auf Typsicherheit, Speichersicherheit (ohne automatische Garbage Collection) und Parallelität. Rust verbietet sowohl Null- als auch Dangling-Pointer. Diese sind bekannt dafür, schwer zu behebende Fehler zu verursachen, die laut Google, für über 50% aller Bugs in Android verantwortlich sind. [9, S. 867f], [23]–[25]

Google has also created some programming languages of their own, including Go, Dart and Carbon. Go was created to combine the efficiency of compiled languages with the simplicity of scripting languages. It is used for building networked services, large-scale web applications and other concurrent software. On the other hand, Dart was developed to provide fast user interface development across multiple platforms. Using the flutter framework one can create cross-platform apps for Android, IOS, Web, Windows, macOS and Linux. [26], [27], **go_web?**

TODO Google: Go, Dart Go and Dart are both programming languages developed by Google that have been gaining popularity in recent years. Both languages were created to address specific issues in the programming world and to provide a solution for developers. Go, often referred to as „Go", was developed to address the need for a language that is simple, efficient and easy to learn, while still providing a high level of performance and

[2] Eine Turing-vollständige Sprache kann zur Umsetzung jedes beliebigen Algorithmus benutzt werden. [8], [9]

[3] JetBrains ist ein Unternehmen, das integrierte Entwicklungsumgebungen entwickelt

[4] Die Apache Software Foundation ist eine gemeinnützige Organisation, die Open-Source-Softwareprojekte unterstützt. [11]

scalability. Go is particularly well-suited for building networked services and large-scale web applications, it aims to make it easy to write concurrent and parallel systems. On the other hand, Dart was created to address the issue of building fast and high-performance apps on multiple platforms. Dart is designed to be a client-optimized language for building fast apps on any platform, it allows for a single codebase to run on different platforms and provides features such as Just-In-Time (JIT) and Ahead-of-Time (AOT) compilation.

- TODO
  - Carbon?
  - Swift: Developed by Apple, Swift is a general-purpose, compiled programming language that is designed to be easy to learn and use. It is particularly well-suited for developing iOS and macOS applications, and has been gaining popularity among developers for its modern syntax, strong type system, and improved performance over Objective-C.
  - TypeScript: Developed by Microsoft, TypeScript is a typed superset of JavaScript that compiles to plain JavaScript. It was designed to make it easier to write and maintain large-scale JavaScript applications by adding features such as static typing, interfaces, and classes. It is widely used in Angular and React projects.
  - Julia: Developed to be used in scientific, engineering, and technical computing, Julia is a high-performance, high-level dynamic programming language. It is designed to be easy to use and has a syntax similar to that of MATLAB or Python. Julia is particularly well-suited for data science and numerical computing and it's been gaining popularity among researchers and data scientists.
  - Scala: Developed by Martin Odersky, Scala is a general-purpose programming language that runs on the Java Virtual Machine (JVM). It is designed to be an object-oriented and functional language, and it's been gaining popularity among developers for its ability to write concurrent and parallel systems. It's also widely used for big data processing using Apache Spark.

3.3 Results

3.3.1 Problems of current and past programming languages

- Parallelization
- (Uncontrolled) Undefined Behaviour
  - https://blog.sigplan.org/2021/11/18/undefined-behavior-deserves-a-better-reputation
- Side Effect Safety, Non local reasoning, Sand Boxing
  - https://www.reddit.com/r/ProgrammingLanguages/comments/9eqrfy/comment/e5s3vhb/?utm_source=share&utm_medium=web2x&context=3
  - Code that we read is not understandable in isolation. For example, taking C++: `call(foo);`, is `foo` modified? Dunno.
    - https://www.reddit.com/r/ProgrammingLanguages/comments/9eqrfy/comment/e5v3vj5/?utm_source=share&utm_medium=web2x&context=3

  - Solved by: Koka
- Cannot rewind time while debugging
  - Reversible Computation: Extending Horizons of Computing
- Rust solves most of these, but a watered-down version which only loses a bit of efficiency could probably help a lot. Source => Basically what Rym tries to be :)

3.3.2 Problems of new programming languages

- https://www.quora.com/What-are-the-biggest-problems-with-modern-programming-languages?share=1
- There are too many of them
- No innovation
  - Too conservative, offer no real improvement to what came before
  - No clear gain to switch to this language
- Being specific to one problem
  - do interesting thing X but do not advance in all other places

3.3.3 Qualities of a good programming language

- (progressivly having to learn new parts / not having to learn everything at once) makes the language easier to learn
  - see talk about type classes, concept is from Swift
  - begin with a simple script like Python and be able to makke it complex over time
- future proof
  - reserved keywords: try, catch, throw, ..
  - allow breaking changes (semantic versioning), Rust good, Python ok, Js/C++ bad, C does not really extend language anymore but rather core libs right?
- Good Package Manager
  - https://www.reddit.com/r/ProgrammingLanguages/comments/zqjf47/a_good_dependency_manager_for_a_new_programming/
  - https://futhark-lang.org/blog/2018-07-20-the-future-futhark-package-manager.html

3.3.4 Type Systems

- getting more powerful
- type inference
- Type Classes have been adapted
  - easily extend functionality of uncontrollable 3rd parties
  - Swift protocols
  - C++23 concepts
  - Rust traits
  - Java, C# Interfaces?
- where?
  - JavaScript
    - TypeScript, other one from Facebook
    - Proposal to add type annotations
  - Python

        - added type annotations
    - Rust, Go, Swift
    - auto in C++
    - state of C, C++, C#, Java, php types?

3.3.5 Null Safety

- solutions based on powerful type systems
    - Rust, Swift, Dart?
    - TypeScript checks for null/undefined
- what are the others up to?

3.3.6 Functional programming

- C++
    - addition and work on the *functional* module
    - algorithms to work with lists and iterators
- Python

4. Rym Programming Language

Rym is a general purpose language designed for working at a level above systems programming, while still allowing the use of lower level features when required. While both an interpreter and a compiler can be used to implement Rym, the current implementation is an interpreter. This description, however, does not assume anything about the implementaion and makes a future compiled version possible.

4.1 Syntax

Which syntax the user of a language prefers is their objective opinion and it is not possible to give a subjective best answer. But most programmers are used to a C-style syntax which is why Rym, like Kotlin, Rust, Dart, and others simply adapts it. Actual functional changes happen at the semantic level.

4.2 General Structure

Rym's execution model is based on packages, which can be either a library or an executable. Packages containing a „main" function can be run as standalone programs, while others are reusable libraries. These packages are made up of modules, functions, constants and other definitions that form a tree-like structure, that provides organisation for the code. **TODO Mention where the constructs are explained in detail**

Source files that represent a top-level module use the .rym extension. To better adhere to the **TODO** principle, Rym also allows the execution of „.rys" script files. These scripts work much like a JavaScript or Python file, where all statements are executed immediately without the need to define an entry function. Rym achieves this by simply wrapping the contents of the script in a main function.

A look at the typical „Hello World!" shows what this transformation looks like in practice:

```
// hello_world.rys

print("Hello World!")
```

It is just as simple as a Python version that does the same thing, but actually corresponds to:

```
// hello_world.rym

func main() → () {
```

```
  print("Hello World!")
}
```

4.3 Modules

In Rym, modules are the building blocks of a package and provide a way to organize code into logical units. Each module can contain its own functions, constants, and other definitions. Modules can be imported into other modules, allowing for the reuse of code.

4.4 Data Types

Rym provides a rich set of data types, including primitive types (such as integers, floats, and strings) and composite types (such as arrays and tuples). Rym also provides support for algebraic data types, which allow for the creation of custom, named data types.

4.4.1 Primitive Data Types

Rym provides a set of basic data types such as integers, floating-point numbers, strings, and booleans. These types can be used to build more complex data structures.

4.4.2 Algebraic Data Types

Algebraic data types (ADTs) in Rym provide a way to create custom, named data types that can be used in the same way as other built-in types. ADTs can be constructed from primitive types or other ADTs, and can be used to model complex data structures in a type-safe manner.

4.5 Functions

Functions in Rym are first-class citizens, just like values and data types. They are blocks of code that can be passed as parameters, assigned to variables, and used in expressions. Functions can return values and accept parameters, which makes them powerful tools for creating modular, reusable code.

4.6 Higher Order Functions

Functions in Rym can be used as values, just like booleans, numbers and strings. They are also higher order functions, meaning that they can be used as arguments to another function, or they can be a function’s result. [28]

Because functions are a data type

To allow for easier declaration of functions inline

```
func twice(f: func(int) → int) → func(int) → int {
  x → f(f(x))
}

func plus_three(i: int) → int {
  i + 3
}

func main() {
  const specific_twice = twice(plus_three)

  print(f"{specific_twice(7)}") // 13
}
```

4.7 Bindings and Scope

Bindings in Rym are the association of a name with a value, and scope determines the accessibility of these bindings. Rym has both global and local scopes, and bindings declared in a local scope are only accessible within that scope.

Immutable

```
const example_1 = 99
example_1 = 100     // error: Cannot assing to immutable variable
example_1
print(example_1)
```

Mutable

```
mut example_1 = 99
example_1 = 100
print(example_1) // prints "100"
```

TODO

4.8 Operators

For example, in Java, a user can increment a simple integer variable in four different ways:

```
count = count + 1
count += 1
count++
++count
```

Although the last two statements have slightly different meanings from each other and from the others in some contexts, all of them have the same meaning when used as stand-alone expressions. These variations are discussed in Chapter 7.

4.9 Control Flow

Control flow in Rym is achieved through the use of conditional statements (if/else) and loops (for/while). Rym also provides a mechanism for early exits from loops or functions through the use of return statements.

4.10 Tooling and Ecosystem

Rym has a growing ecosystem of tools and libraries that make it easier to develop, test, and deploy applications. These tools include IDEs, text editors with Rym plugins, build tools, package managers, and libraries for various domains. Rym's tooling and ecosystem are designed to be flexible and allow for easy integration with other systems and technologies.

5. Primitive Data Types

Data types that are not defined in terms of other types are called primitive data types. Nearly all programming languages provide a set of primitive data types. Some of the primitive types are merely reflections of the hardware for example, most integer types and others require only a little nonhardware support for their implementation. More complex types can be created by combining primitive types as we will see in Kapitel 6. [2, S. 400]

5.1 Boolean

Boolean types are perhaps the simplest of all types and have been included in most general-purpose languages designed since 1960. They only have two possible values one for true and one for false. Boolean types are often used to represent switches or flags in programs. Although other types, such as integers, can be used for these purposes, the use of boolean types is more readable. C and C++ still allow numeric expressions to be used as if they were boolean. This is not the case in the subsequent languages, Java and C# which is why Rym will disallow this as well. [29], [30]

A boolean value could be represented by a single bit, but because a single bit of memory cannot be accessed efficiently on many machines, they are often stored in the smallest efficiently addressable cell of memory, typically a byte. As this detail is trivial Rym does not specify how to store boolean values. [2, S. 404f]

The boolean type in Rym is called „bool” like in Python, PHP, C#, Go, Rust, Swift and many others. It is named after *George Boole* who pioneered the field of mathematical logic. [26], [30]–[33]

5.2 Boolean Operations

A Boolean value may be created using the *true* or *false* literals

```
const var_1 = true
const var_2 = false
```

and is always the result for the comparison binary operators ==, <, <=, >= and >. The comparision operations actually use the literals in their implementation as well, which can be seen in Kapitel 8. There is also the unary not prefix operator represented by ! which allow one to invert a boolens value . The prefix already suggests that this operator must come before the expression it operates on, as the ! can be used as a unary postfix operator to unwrap a value and **TODO: INSERT CHAPTER REF** explains why that is useful.

In Rym the control-flow expressions *if* and *while* use booleans to decide whether some code should be executed or not. How they work and why their syntax tooks like this will be covered in **TODO: INSERT CHAPTER REF**.

```
const condition = true
if condition { /* do something once */ }
while condition { /* do something forever */ }
```

5.3 Numeric Types

5.3.1 Integer

Another very common primitive numeric data type is the integer. The hardware of many computers supports several sizes of integers. These sizes of integers, and often a few others, are supported by some programming languages. As seen in Tabelle 5.1, Java includes four signed integer sizes: byte, short, int, and long. Some languages, for example C, C++ and C# include unsigned integer types, which are types for integer values without signs. Unsigned types are often used for binary data. 8 bit large unsigned integers can for example represent exactly one byte. [2, S. 400]

The types for C and C++ that are represented in Tabelle 5.1 are using the „cstdint” header as the language standards do not specify the sizes for default integer types and leave them up to the implementation. Types from this standard header file are however required to that exact size. [34, S. 0], [35, S. 0]

- Python [5]
  - size as large as needed
- PHP [6]
  - int
  - size platform dependent, 32bit|64bit
- Java [7]
- C, C++:
  - char, short, int, long are not always the same size
  - cstdint header: c++ standard S. 493f.
- Go [8]
- Zig (special case) [9]
  - arbitrary size
  - size 1–65535
  - i{Size} eg. i333
  - u{Size} eg. u8
- Rust [10]

Tabelle 5.1: Supported integer formats

Size [Bits]	Java	C#	C, C++	Go	Rust
8	byte	sbyte, byte	int8_t, uint8_t	int8, uint8	i8, u8
16	short	short, ushort	int16_t, uint16_t	int16, uint16	i16, u16
32	int	int, uint	int32_t, uint32_t	int32, uint32	i32, u32
64	long	long, ulong	int64_t, uint64_t	int64, uint64	i64, u64
32|64	—	nint, nuint	—	int, uint	—
128	—	—	—	—	i128, u128
pointer	—	—	intptr_t, uintptr_t	intptr, uintptr	isize, usize

5.3.2 Float

Generally languages provide floating point data types that adhere to the „IEEE 754 - Floating-Point” arithmetic standard [36] or its ISO adoption „ISO/IEC 60559” [37]. How wide that support is can be seen in Tabelle 5.2.

- IEEE 754
  - https://en.wikipedia.org/wiki/IEEE_754
  - active version is from 2019 [36]
  - https://ieeexplore.ieee.org/document/8766229
  - same as ISO/IEC 60559
- Accessed: 02.01.2023

[5] https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex

[6] https://www.php.net/manual/en/language.types.integer.php

[7] https://docs.oracle.com/javase/specs/jls/se19/html/jls-4.html#jls-4.2

[8] https://go.dev/ref/spec#Numeric_types

[9] https://ziglang.org/documentation/master/#Integers

[10] https://doc.rust-lang.org/reference/types/numeric.html#integer-types

  - Js: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number
  - Python: https://docs.python.org/3/library/stdtypes.html#typesnumeric
  - PHP: https://www.php.net/manual/en/language.types.float.php
  - Java: https://docs.oracle.com/javase/specs/jls/se19/html/jls-4.html#jls-4.2.3
  - Go: https://go.dev/ref/spec#Numeric_types
  - Rust: https://doc.rust-lang.org/reference/types/numeric.html#floating-point-types
- Accessed: 09.01.2023
  - C#
    - https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/types#837-floating-point-types
    - https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/floating-point-numeric-types
  - C, C++
    - C++ „The value representation of floating-point types is implementation-defined.” S. 75
    - standards do not specific float format to use, they just mandate the minimum range and precision
    - https://en.cppreference.com/w/cpp/language/types

Tabelle 5.2: Supported IEEE-754 floating point formats

Size [Bits]	Js/Ts	Python	PHP	Java	C#	C, C++	Go	Rust
32	Number	—	—	float	float	?	float32	f32
64	—	—	—	double	double	?	float64	f64
32|64	—	float	float	—	—	—	—	—

5.3.3 Decimal

- Pythony 09.01.2023
  - decimal.Decimal
  - The decimal module provides support for fast correctly rounded decimal floating point arithmetic.
  - Once constructed, Decimal objects are immutable.
  - https://docs.python.org/3/library/decimal.html

5.4 Character

- Rym:
  - Name: char
  - valid utf-8 character
  - space: 1 byte?

Tabelle 5.3: Character data types

Language	Formats
Js/Ts	not supported
Python	not supported?

Language	Formats
PHP	not supported?
Java	char: 16bit unsinged int
C#	?
C++	?
C	?
Go	?
Rust	`char`

- Java: 09.01.2023 https://docs.oracle.com/javase/specs/jls/se19/html/jls-4.html#jls-4.2

5.5 String

- Rym:
  - characters array: [char]
  - dynamic characters vector: String

```
const const_string: [char; 12] = "Hello World!"

impl Add for [char] {
  fn add(move self, move rhs: Self) -> Self {
    [..self, ..rhs]
    // or
    mut new_array = ['\0'; self.length + rhs.length]
    new_array[0..self.length] = self
    new_array[self.length..] = rhs
  }
}
```

6. Algebraic Data Types

Definintion [38]

6.1 Product Types

- explain product types
  - https://jrsinclair.com/articles/2019/algebraic-data-types-what-i-wish-someone-had-explained-about-functional-programming
- exist in:
  - Js/Ts (Arrays, Objects, Maps, WeakMaps)
  - Python (Lists, Records, ..)
  - PHP
  - Java
  - C#
  - C++ (Classes, Structs, ..)
  - C (Structs, ..)
  - Go
  - Rust (Structs, ..)

6.2 Sum Types

- explain sum types
  - https://jrsinclair.com/articles/2019/algebraic-data-types-what-i-wish-someone-had-explained-about-functional-programming/
- exist in: (https://en.wikipedia.org/wiki/Algebraic_data_type)
  - C++, Java 15, Rust, TypeScript
  - F#, Haskell, Idris, Kotlin, Nim, Swift

6.2.1 Enums

```
// Declare an enum.
enum Type {
  Ok,
  NotOk,
}

// Declare a specific enum field.
const c = Type.Ok
```

An `enum` can be `matched` upon.

```
enum Foo {
  String,
  Number,
  None,
```

```
}
test "enum match" {
  const foo = Foo.Number
  const what_is_it = match foo {
    Foo.String ⇒ "this is a string",
    Foo.Number ⇒ "this is a number",
    Foo.None ⇒ "this is a none",
  }
  assert_eq(what_is_it, "this is a number")
}
```

6.3 Optional Values

- used
  - Initial values
  - Return values for functions that are not defined over their entire input range (partial functions)
  - Return value for otherwise reporting simple errors, where None is returned on error
  - Optional struct fields
  - Struct fields that can be loaned or „taken”
  - Optional function arguments
  - Nullable pointers
  - Swapping things out of difficult situations
- The *Option* type represents an optional value: every *Option* is either *Some* and contains a value, or *None*, and does not. *Option* types are very common in Rym code, as they have a number of uses:
- *Options* are commonly paired with pattern matching to query the presence of a value and take action, always accounting for the *None* case.
- null references, the billion dollar mistake
  - https://www.lucidchart.com/techblog/2015/08/31/the-worst-mistake-of-computer-science/
  - https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/
  - https://en.wikipedia.org/wiki/Tony_Hoare
- Ts
  - `null`, `undefined`
  - can be detected by Ts compiler
- Java, C, C++, ..
- Rust
  - `Option` enum
  - must be unwrapped to use the value
- `Options` replace null references
  - is a sum type / enum

[39]

7. Functions

TODO refer to characteristics and analysis; TODO compare function keywords: https://twitter.com/code_report/status/1325472952750665728/photo/1

Functions are one of the most important building blocks of many programming languages.

7.1 Basic functionality

In Rym it will be possible to define a function by using the *func* keyword followed by a name for the function, a comma separated list of parameters and their respective types as well as the return type of the function.

This means that a function with multiple parameters will generally look like this:

```
func name(parameter0: Type0, parameter1: Type1) → ReturnType {
  // ..
}
```

Just like in any *C-style* language this function may be called by specifying the name of the function followed by a list of arguments within parentheses separated by commas. The arguments must match the types of the parameters defined in the function definition. The previous example function would be called like this:

```
const result = name(argument0, argument1)
```

This would assign the return value to the variable result.

7.2 Scope

When a function is called, the variables and values in the current scope are temporarily suspended and a new scope is created for the function. This new scope contains the parameters of the function as variables, initialized with the values of the arguments passed to the function. The function body is then executed within this new scope, and any variables or values defined within the function are only accessible within this scope. When the function execution is complete, the function scope is destroyed and the original scope is restored.

For example, consider the following code:

```
const x = 5

func changeX(x: Int) → Int {
  const y = 10
  return x + y
```

```
}

const result = changeX(3) // result = 13
```

When the function changeX is called, a new scope is created containing the parameter newX initialized with the value 3. The variable y is also defined within this scope and initialized with the value 10. The function body is executed, and the value of newX + y (13) is returned. The function scope is then destroyed and the original scope is restored, resulting in the value 13 being assigned to the variable result. The variable y is not accessible outside of the function scope and therefore does not affect the value of x.

7.3 Returning

In the Rym programming language, a function may be defined with a return type, which specifies the type of value that will be returned by the function. The return type is specified after the arrow symbol (->) in the function definition, followed by the function body in curly braces ({}). The function body may contain any number of statements and expressions, and the value of the final expression in the function body will be returned as the result of the function.

For example, consider the following function definition:

```
func multiply(x: Int, y: Int) → Int {
  let result = x * y
  return result
}
```

In this example, the function multiply takes two integer arguments, x and y, and returns the result of their multiplication as an integer. The function body contains a single statement that defines a local variable result and initializes it with the value of x * y. The return statement then specifies that the value of result should be returned as the result of the function.

However, it is not necessary to specify a return statement in every function. If the return type of the function is specified, the value of the final expression in the function body will be returned automatically. For example, the following function is equivalent to the one above:

```
func multiply(x: Int, y: Int) → Int {
  x * y
}
```

In this case, the value of the expression x * y is returned as the result of the function, without the need for a separate return statement. This allows for more concise and readable code, as the return statement is often unnecessary when the function body contains only a single expression.

7.4 Default Parameter Values

TODO Add explaination

```
func example(required: String, optional: String = "default") {
  print(required, optional)
}

// Type of `optional` can be inferred
func example(required: String, optional = "default") {
  print(required, optional)
}

example("A")                    // prints "A default"
example("A", "specific value")  // prints "A specific value"
```

7.5 Closures

Rym is trying to combine imperative and declarative approaches of programming and already allows functions to used like data. But function declarartions are statements and cannot be used as expressions. This can become very cumbersome if functions should be passed into a function call as they first have to be defined and then referenced via their name. Closure solve this problem, as they are expressions and allow functions to be created inline. Their syntax is based on the function declarations of Rym and can be generalized like this:

```
(param_1, param_2, param_n) -> ReturnType { /*  closure body */ }
```

This form is still very long, which is why several parts of it can be omitted if they are not needed. The return type can be omitted if it is inferreable from context.

```
(param_1, param_2) -> { /* closure body */ }
```

A body with only one expression does not require a block surrounding it.

```
(param_1, param_2) -> /* single expression */
```

If there is only one parameter the parentheses can be omitted and a closure without parameters just starts with the thin arrow `->`.

```
param -> /* single expression */
-> /* single expression */
```

TODO Explsin why a closure without parameters makes sense

```
const outer = 9

```

```
const closure_1 = -> 9
const closure_2 = -> outer
const closure_3 = other -> outer + other

func add(lhs: uint, rhs: uint) -> uint { lhs + rhs }
const add = (lhs, rhs) -> lhs + rhs
const add = (lhs, rhs) -> uint { lhs + rhs }
const add = (lhs: uint, rhs: uint) -> uint { lhs + rhs }

const fraction = n: uint -> (1..=n).fold(1, (accum, n) -> accum * n)

{
  mut counter = 0
  const increment = -> counter += 1
  increment()
  assert_eq(counter, 1)
}
{
  mut counter = 0
  const increment = n -> counter += n
  increment(20)
  assert_eq(counter, 20)
}
```

8. Polymorphism

Some much older languages are based on the idea that functions, procedures and their respective parameters have unique types. These languages are said to be *monomorphic* (from Greek ‚one shape'), in the sense that every value and variable can be interpreted to be of one and only one type. Monomorphic programming languages may be contrasted with *polymorphic* (from Greek ‚many shapes') [40, S. 760] languages in which some values may have more than one type. [41, S. 4]

Polymorphic functions are functions whose parameters can have more than one type. Polymorphic types are types whose operations are applicable to values of multiple types.

Polymorphism is supported by

Most often, it is a good idea to give different functions different names, but when different functions conceptually perform the same task on objects of different types, it can be more convenient to give them the same name [40, S. 326]

- *polymorphic*
- compile-time vs run-time polymorphism
  - Rym will only support compile-time as it is simpler to design and implement
  - run-time polymorphism could be added later on

9. Tooling and Ecosystem

- no time to implement them for Rym

9.1 Package Manager

- examples:
  - Python: pip
  - Js/Ts: npm, yarn, deno
  - C: ?
  - C++: ?
  - Rust: cargo

=> rympkg?

9.2 Formatting and Styleguide

9.2.1 Naming Conventions

- checked by compiler => error that stops compilation
  - variable: `variable_name`
  - function: `function_name`
  - struct: `StructName`
  - enum: `EnumName`
  - trait: `TraitName`

9.2.2 Linters

- examples:
  - Python: pep8, ?
  - Js/Ts: not official?, eslint, prettier, ..
  - PHP: ?
  - ..: ?
  - Go: ?, gofmt
  - Rust: builtin, clippy, rustfmt

Precommit scripts can ensure that only correctly formatted code gets comitted.

=> rymfmt, rym fmt

10. Fazit

TODO

Bibliografie

[1] R. Stansifer, *Theorie und Entwicklung von Programmiersprachen. Eine Einführung*. München: Prentice Hall, 1995.

[2] R. W. Sebasta, *Concepts of Programming Languages. 12th Edition*. Essex: Pearson Education, 2019.

[3] *ISO/IEC 24772-1. Programming languages – Avoiding vulnerabilities in programming languages – Part 1: Language independent catalogue of vulnerabilities*. 2019. Verfügbar unter: https://open-std.org/jtc1/sc22/wg23/docs/ISO-IECJTC1-SC22-WG23_N1218-wd24772-1-international-standard-seed-document-for-DIS-ballot-20221023.pdf

[4] S. Michell, „Ada and Programming Language Vulnerabilities“, *Ada User Journal*, Bd. 30, Nr. 3, S. 180, 2009.

[5] „Stack Overflow Annual Developer Survey“. https://insights.stackoverflow.com/survey (zugegriffen 15. Januar 2023).

[6] „2022 StackOverflow Developer Survey. Programming, scripting, and markup languages“. https://survey.stackoverflow.co/2022/#section-most-popular-technologies-programming-scripting-and-markup-languages (zugegriffen 14. Januar 2023).

[7] „2022 StackOverflow Developer Survey. Full Data Set (CSV)“. https://info.stackoverflowsolutions.com/rs/719-EMH-566/images/stack-overflow-developer-survey-2022.zip (zugegriffen 15. Januar 2023).

[8] M. L. Scott, *Programming language pragmatics. 3rd Edition*. Burlington: Morgan Kaufmann, 2009.

[9] M. L. Scott, *Programming language pragmatics. 4th Edition*. Burlington: Morgan Kaufmann, 2016.

[10] B. QoChuk, „Age of All Programming Languages“. https://iq.opengenus.org/age-of-programming-languages (zugegriffen 27. September 2023).

[11] „Apache Software Foundation“. https://apache.org/ (zugegriffen 31. Januar 2023).

[12] „Apache Groovy. A multi-faceted language for the Java platform“. https://groovy-lang.org/index.html (zugegriffen 15. Januar 2023).

[13] „Groovy Language Documentation. Version 4.0.7“. http://www.groovy-lang.org/single-page-documentation.html (zugegriffen 15. Januar 2023).

[14] „The Clojure Programming Language“. https://clojure.org/index (zugegriffen 15. Januar 2023).

[15] „Closure Reference“. https://clojure.org/reference (zugegriffen 15. Januar 2023).

[16] „The Scala Programming Language“. https://scala-lang.org (zugegriffen 15. Januar 2023).

[17] „Go. Build simple, secure, scalable systems with Go“. https://go.dev (zugegriffen 31. Januar 2023).

[18] „Scala Language Specification. Version 2.13“. https://scala-lang.org/files/archive/spec/2.13 (zugegriffen 15. Januar 2023).

[19] „JetBrains. Essential tools for software developers and teams“. https://www.jetbrains.com (zugegriffen 14. Januar 2023).

[20] „Kotlin. A modern programming language that makes developers happier“. https://kotlinlang.org (zugegriffen 14. Januar 2023).

[21] „Kotlin Language Documentation 1.8.0“. https://kotlinlang.org/docs/kotlin-reference.pdf (zugegriffen 14. Januar 2023).

[22] „Develop Android apps with Kotlin“. https://developer.android.com/kotlin (zugegriffen 15. Januar 2023).

[23] „Interview on Rust, a Systems Programming Language Developed by Mozilla“. https://www.infoq.com/news/2012/08/Interview-Rust (zugegriffen 14. Januar 2023).

[24] „Rust. A language empowering everyone to build reliable and efficient software“. https://www.rust-lang.org (zugegriffen 14. Januar 2023).

[25] L. Bergstrom, „Google joins the Rust Foundation“, 2021. https://opensource.googleblog.com/2021/02/google-joins-rust-foundation.html (zugegriffen 29. Januar 2023).

[26] „The Go Programming Language Specification“. https://go.dev/ref/spec (zugegriffen 10. Januar 2023).

[27] „Flutter. Multi Platform“. https://flutter.dev/multi-platform (zugegriffen 31. Januar 2023).

[28] „Higher order function“. https://wiki.haskell.org/Higher_order_function (zugegriffen 30. Januar 2023).

[29] „Java Language Specification. Types, Values, and Variables.“ https://docs.oracle.com/javase/specs/jls/se19/html/jls-4.html (zugegriffen 10. Januar 2023).

[30] „C# 7.0 draft specification. Types.“ https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/types (zugegriffen 10. Januar 2023).

[31] „PHP Language Reference. Types.“ https://www.php.net/manual/en/language.types.php (zugegriffen 10. Januar 2023).

[32] „The Rust Reference. Types.“ https://doc.rust-lang.org/stable/reference/types.html (zugegriffen 10. Januar 2023).

[33] „Swift Documentation“. https://developer.apple.com/documentation/swift (zugegriffen 10. Januar 2023).

[34] *ISO/IEC 9899:2018. Programming languages – C*. 2018. Verfügbar unter: https://www.open-std.org/JTC1/SC22/WG14/www/docs/n2310.pdf

[35] *ISO/IEC 14882:2020. Programming languages – C++*. 2020. Verfügbar unter: https://www.open-std.org/JTC1/SC22/WG21/docs/papers/2022/n4910.pdf

[36] „IEEE Standard for Floating-Point Arithmetic“, *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, 2019, doi: 10.1109/IEEESTD.2019.8766229.

[37] *IEEE Standards Association*, 2020. https://standards.ieee.org/ieee/60559/10226 (zugegriffen 2. Januar 2023).

[38] J. Sinclair, „Algebraic Data Types: Things I wish someone had explained about functional programming“, 2019. https://jrsinclair.com/articles/2019/algebraic-data-types-what-i-wish-someone-had-explained-about-functional-programming (zugegriffen 31. Februar 2023).

[39] P. Wadler und S. Blott, „How to make ad-hoc polymorphism less ad hoc“, in *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1989.

[40] B. Stroustrup, *The C++ Programming Language. Fourth Edition*. Boston: Addison-Wesley, 2013.

[41] L. Cardelli und P. Wegner, „On understanding types, data abstraction, and polymorphism“, *ACM Computing Surveys (CSUR)*, Bd. 17, 1985.

Anhang A — Quellcode Beispiele

A.1 Factorial

Two possible implementations for calculating factorials. Pseudo code from Wikipedia:

```
define factorial(n):
  f := 1
  for i := 1, 2, 3, ... , n:
    f := f × i
  return f
```

Imperative approach

```
func factorial(n: uint) -> uint {
  mut result = 1
  for const i in 1..=n {
    result *= i
  }
  result
}
```

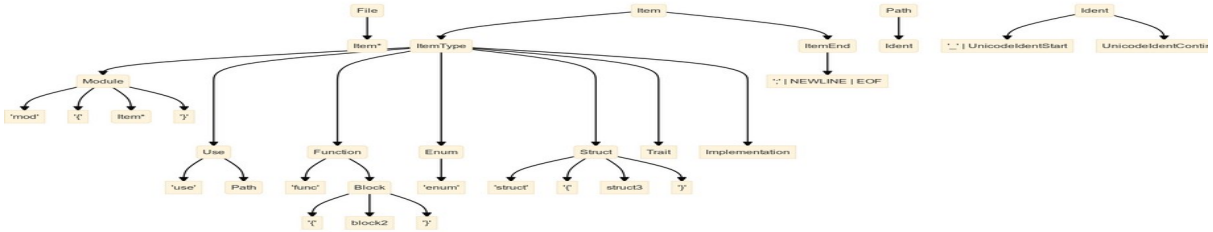
Functional approach

```
func factorial(n: uint) -> uint {
    (1..=n).fold(1, (accum, i) -> accum * i)
}
```

A.2 Summands __TODO Use better name_

```
func main() -> @Io {
  const numbers = [-14, 1, 3, 6, 7, 7, 12]

  const sum = -13
  if const Some([left, right]) = summands(numbers, -13) {
    print(f"Sum of {left} and {right} = {sum}")
  }
  print("Pointers have crossed, no sum found")
}

/// array must be sorted
func summands(array: [i32], sum: i32) -> Option<[usize; 2]> {
  mut low = 0
  mut high = array.len() - 1

  while low < high {
    const current_sum = array[low] + array[high]

    if current_sum == sum {
      return Some([array[low], array[high]])
    } else if current_sum < sum {
      low += 1
    } else {
      high -= 1
    }
  }

  None
}
```

Anhang B — Rym Parser Grammar



File: Item* Item: Module | Use | Function | Enum | Struct | Trait | Implementation

```
Stmt ⇒ Item | VarStmt | ExprStmt
VarStmt ⇒ ("const" | "mut") Ident "=" Expr ItemEnd
ExprStmt ⇒ Expr ItemEnd
Expr ⇒ ExprWithoutBlock | ExprWithBlock
ExprWithoutBlock ⇒ LiteralExpr | PathExpr | OperatorExpr |
GroupedExpr | ArrayExpr
                  | IndexExpr | Tuple | MethodCallExpr | CallExpr |
RangeExpr
                  | ClosureExpr | ContinueExpr | BreakExpr |
ReturnExpr
LiteralExpr ⇒ FloatLit | IntLit | StringLit | CharLit
PathExpr ⇒ Path
OperatorExpr ⇒ UnaryExpr | BinaryExpr
UnaryExpr ⇒ ("!" | "-") Expr
BinaryExpr ⇒ Expr ("+" | "-" | "*" | "/" | "%") Expr
GroupedExpr ⇒ "(" Expr ")"
ArrayExpr ⇒ "[" (Expr ("," Expr)* ","? | Expr ";" Expr) "]"
IndexExpr ⇒ Expr "[" Expr "]"
Tuple ⇒ "(" ((Expr ",")+ Expr)? ")"
MethodCallExpr ⇒ Expr "." Ident "(" CallArgs? ")"
CallExpr ⇒ Expr "(" CallArgs? ")"
CallArgs ⇒ Expr ("," Expr)* ","
RangeExpr ⇒ Expr? ".." "="? Expr?
ClosureExpr ⇒ "|" ClosureParams "|" (Expr | "→" Type BlockExpr)
ClosureParams ⇒ __TODO__
ContinueExpr ⇒ "continue"
BreakExpr ⇒ "break" Expr?
ReturnExpr ⇒ "return" Expr
```

```
ExprWithBlock ⇒ BlockExpr | LoopExpr | IfExpr | IfVarExpr |
MatchExpr
BlockExpr ⇒ "{" Stmt* "}"
LoopExpr ⇒ "loop" BlockExpr
IfExpr ⇒ "if" Expr BlockExpr ("else" (BlockExpr | IfExpr |
IfVarExpr))?
IfVarExpr ⇒ "if" ("const" | "mut") __TODO__ "=" Expr BlockExpr
("else" (BlockExpr | IfExpr | IfVarExpr))?
MatchExpr ⇒ __TODO__
Type ⇒ Path
Ident ⇒ ('_' | UnicodeIdentStart) UnicodeIdentContinue
ItemEnd ⇒ ";" | "\n" | EOF
```