1. Einleitung

Jedes Jahr werden zahlreiche neue Programmiersprachen veröffentlicht. Viele von ihnen starten als Teil einer wissenschaftlichen Arbeit, und sowohl Privatpersonen als auch Studierende arbeiten in ihrer Freizeit häufig an eigenen Sprachen, um verschiedene Aspekte der Umsetzung kennenzulernen. In manchen Fällen möchten Unternehmen oder Organisationen ein bestimmtes Problem angehen, aber ihnen fehlt 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. Bei der Umsetzung kann man von Null anfangen und konzeptionell alle Erkenntnisse aus früheren Versuchen nutzen. Völlig neue Ideen können so viel einfacher und schneller ausprobiert werden. Bei einer Sprache, die von Tausenden von Menschen benutzt wird und möglichst stabil bleiben soll, gestaltet sich 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. Eigenschaften von Programmiersprachen

Eine Liste dieser Art ist stets umstritten, da es unterschiedliche Meinungen über den Wert einer bestimmten Spracheigenschaft im Vergleich zu anderen gibt. Obwohl es auch aufgrund der Unterschiedlichen Anwendungsbereiche einer Programmiersprache keine allgemeingültigen positiven Aspekte gibt, werden die folgende Kriterien häufig genannt. [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 öfter gelesen als geschrieben und Wartungen, bei welchen viel gelesen wird, sind ein wichtiger Teil des Entwicklungsprozesses. Da die Wartungs-freundlichkeit 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 vielen Grundstrukturen ist schwerer zu verstehen als eine mit wenigen. Programmierer, die eine komplexe Sprache verwenden müssen, lernen oft nur eine Teilmenge der Sprache und ignorieren andere. 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]

Weiterhin kann das Überladen von Operatoren, bei dem ein einzelner Operator viele verschiedene Implementierung haben kann, die Einfachheit beinflussen. Das ist zwar oft sinnvoll, kann aber das Lesen erschweren, wenn Benutzern erlaubt wird, ihre eigenen Überladungen zu erstellen und sie dies nicht in einer vernünftigen Art und Weise tun. Es ist jedoch durchaus akzeptabel, „+” zu überladen, um es sowohl für die Ganzzahl- als auch für die Gleitkommaaddition zu verwenden. In diesem Fall vereinfachen Überladung die Sprache sogar, da sie die Anzahl der verschiedenen Operatoren reduzieren. [2, S. 43f], [2, S. 664]

Eine dritte Charakteristik, welches die Einfachheit einer Programmiersprache beeinträchtigt, ist das Vorhandensein mehrerer Möglichkeiten, dieselbe Operation auszuführen. [2, S. 43]

Andererseits kann man es mit der Einfachheit auch übertreiben. Beispielsweise sind Form und Bedeutung der meisten Assembler-Anweisungen sehr einfach. Da komplexere Steueranweisungen fehlen, ist die Programmstruktur jedoch weniger offensichtlich. Es werden mehr Anweisungen benötigt als in entsprechenden Programmen einer höheren Programmier-sprache. Die gleichen Argumente gelten auch für den weniger extremen Fall von Sprachen mit unzureichenden Kontroll- und Datenstrukturen. [2, S. 44]

2.1.2 Orthogonalität

Orthogonalität *(griech. orthos „gerade, aufrecht, richtig, rechtwinklig“)* in einer Programmier-sprache 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 also zu Ausnahmen an den Regeln der Sprache. Je weniger Ausnahmen es gibt, desto leichter ist die Sprache zu lesen, zu verstehen und zu erlernen. [2, S. 44ff]

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

2.1.3 Datentypen

Die Lesbarkeit kann noch weiter verbessert werden, indem den Benutzern die Möglichkeit gegeben wird, geeignete Datentypen und Datenstrukturen zu definieren. Dies gilt insbesondere für Booleans und Enumerationen. [2, S. 47] In einigen Sprachen, die keinen Boolean-Typen bieten, könnte z.B. der folgende Code verwendet werden:

```
const use_timeout = 1
```

Die Bedeutung dieser Anweisung ist unklar, und kann in einer Sprache mit Boolean-Typen wesentlich klarer dargestellt werden:

```
const use_timeout = true
```

2.1.4 Syntax

Die Lesbarkeit eines Programms wird außerdem stark von seiner Syntax beeinflusst. Syntax bezieht sich auf die Regeln und die Struktur, die vorgeben, wie ein Programm in einer bestimmten Programmiersprache geschrieben werden muss. Diese Regeln definieren die Reihenfolge, den Aufbau und die Verwendung von Anweisungen, Ausdrücken, Schlüsselwörtern und Zeichensetzung innerhalb der Sprache. Auch hier ist es sinnvoll, die Grundsätze der Einfachheit und Orthogonalität anzuwenden. Daher sollten Elemente mit ähnlicher Bedeutung ähnlich geschrieben werden. Zu viele Alternativen, um Code mit der gleichen Bedeutung zu schreiben, verstoßen ebenfalls gegen die Orthogonalität. Letztlich hängt die Lesbarkeit einer Sprache jedoch von individuellen Vorlieben und Vorkenntnissen ab. Die Übernahme von Teilen der Syntax bekannter Sprachen in eine neue Sprache ist daher sehr sinnvoll. [1, S. 61ff], [2, S. 48f]

2.2 Schreibbarkeit

Die Leichtigkeit, mit der eine Programmiersprache verwendet werden kann, um Programme für eine bestimmte Aufgabe zu schreiben, wird als ihre Schreibfähigkeit bezeichnet. Die Sprachmerkmale, die die Lesbarkeit verbessern, tragen auch zu ihrer Schreibbarkeit bei, denn das Schreiben eines Programms erfordert häufig das erneute Lesen von Teilen des Codes. Um die Schreibbarkeit zu optimieren, ist es wünschenswert, dass die Syntax die Notwendigkeit für den Programmierer minimiert, herumzuspringen, Das macht es einfacher, sich auf das Schreiben von Code zu konzentrieren. Dies kann durch eine Syntax erreicht werden, die auf einer Leserichtung von links nach rechts basiert, wie im Englischen. [2, S. 49ff]

Neben einer leicht verständlichen Syntax ist es für eine Programmiersprache wichtig, dass sie eine klare und einfache Struktur hat. So kann sich ein Programmierer auf eine Aufgabe konzentrieren, anstatt mehrere Konzepte gleichzeitig im Kopf behalten zu müssen. Eine Sprache mit einer einfachen und intuitiven Struktur verringert die geistige Anstrengung, die zum Schreiben von Code erforderlich ist, Das macht es einfacher, sich auf die Aufgabe zu konzentrieren, und verringert die Fehleranfälligkeit. Mit anderen Worten: Eine Sprache, die die Anzahl der Dinge, über die ein Programmierer gleichzeitig nachdenken muss, minimiert, ist ein Schlüsselfaktor für die Verbesserung der Schreibfähigkeit. [2, S. 49ff]

2.3 Verlässlichkeit

Ein Programm gilt als zuverlässig, wenn es unter allen Bedingungen die erwartete Leistung erbringt und dies auch in Zukunft tun wird. [2, S. 51]

2.3.1 Spezifikation

Eine Programmiersprache ohne eine standardisierte Spezifikation kann kaum als zuverlässig angesehen werden. Bei dieser Spezifikation handelt es sich um eine Reihe von Regeln, die eine Implementierung der Sprache befolgen muss, um sicherzustellen, dass sich die Programme wie vorgesehen verhalten. Sie sollte alle möglichen Kombinationen an Merkmalen der Programmiersprache abdecken. [3, S. 615]

In einigen Fällen kann das Verhalten bestimmter Merkmale unzureichend definiert sein. Dies kann zu verschiedenen Implementierungen führen, die unterschiedliche Ergebnisse liefern. Einige der möglichen Ergebnisse können schädlich sein, und die Implementierungen müssen nicht über alle Ausführungen hinweg konsistent bleiben, Versionen oder Optimierungsstufen konsistent bleiben. [4, S. 21], [5, S. 190]

Verhaltensweisen, die in der Spezifikation nicht erwähnt werden, nennt man *unspezifiziertes Verhalten*, und *undefiniertes Verhalten*, wenn sie zwar erwähnt werden, aber nicht definiert ist, wie sie sich verhalten. Sie stellen eine große Herausforderung für die Benutzer der Sprache dar, da sie nicht sicher sein können, wie und ob ihr Programm ausgeführt wird. Aus diesem Grund sollte eine Sprache so wenige von ihnen wie möglich haben. [4, S. 21]

2.3.2 Sicherheit

Zusätzlich zu den Sicherheitslücken, die durch nicht spezifiziertes oder undefiniertes Verhalten entstehen, gibt es weitere Probleme, die angegangen werden sollten, um die Sicherheit des Programms zu gewährleisten. Eines dieser Probleme ist das Aliasing, welches auftritt, wenn mehrere Variablen auf dieselbe Speicherstelle verweisen. Dies kann zu unerwarteten Änderungen an den Daten führen und Fehler verursachen, die nur schwer zu erkennen sind. Um diese Risiken zu mindern, kann die Programmiersprache mit robusten statischen Analysemechanismen entworfen werden, wie z.B. Daten-/Kontrollflussanalyse, Bounds-Checking und Type Checking. Dadurch wird sichergestellt, dass die Annahmen des Benutzers über die zu verarbeitenden Daten korrekt sind, was zur Vermeidung von Fehlern beiträgt und die Zuverlässigkeit von Programmen verbessert. [2, S. 51ff] [4, S. 81f] [4, S. 37f] [4, S. 16]

2.3.3 Zukunftssicherheit

Eine Programmiersprache kann zukunftssicher gemacht werden, indem sie mit Blick auf Skalierbarkeit und Wartungsfreundlichkeit entwickelt wird. Technische Schulden können unter Kontrolle gehalten werden, indem übermäßig komplexe Konstrukte vermieden werden, und das Feedback von Benutzern und Entwicklern berücksichtigt wird. Grundlegende Änderungen an einer Sprache können sinnvoll sein, um grundlegende Designprobleme zu beheben oder neue, bahnbrechende Technologien einzubinden. Solche Änderungen müssen jedoch sorgfältig durchgeführt werden, da sie zu Problemen mit der Rückwärts-kompatibilität führen können. Die Auswirkungen von Änderungen können abgeschwächt werden, indem man die langfristigen Auswirkungen von Entscheidungen im Sprachdesign berücksichtigt. Ein Beispiel hierfür sind reservierte Schlüsselwörter, die Programmierer nicht zur Namensgebung verwenden dürfen. Viele Sprachen fügen dieser Liste Schlüsselwörter hinzu, die in der Zukunft verwendet werden könnten, um spätere grundlegende Änderungen zu vermeiden. [2, S. 344]

3. Analyse bestehender Programmiersprachen

Der nächste Schritt auf der Suche nach einer Antwort besteht darin, die derzeit verwendeten Programmiersprachen zu untersuchen, insbesondere diejenigen, die erst vor kurzem erschienen sind. Ziel ist es, herauszufinden, was diese im Vergleich zu den älteren Sprachen verändert 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 kann man eine Statistik aus der Umfrage sehen, 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. [6]–[8]

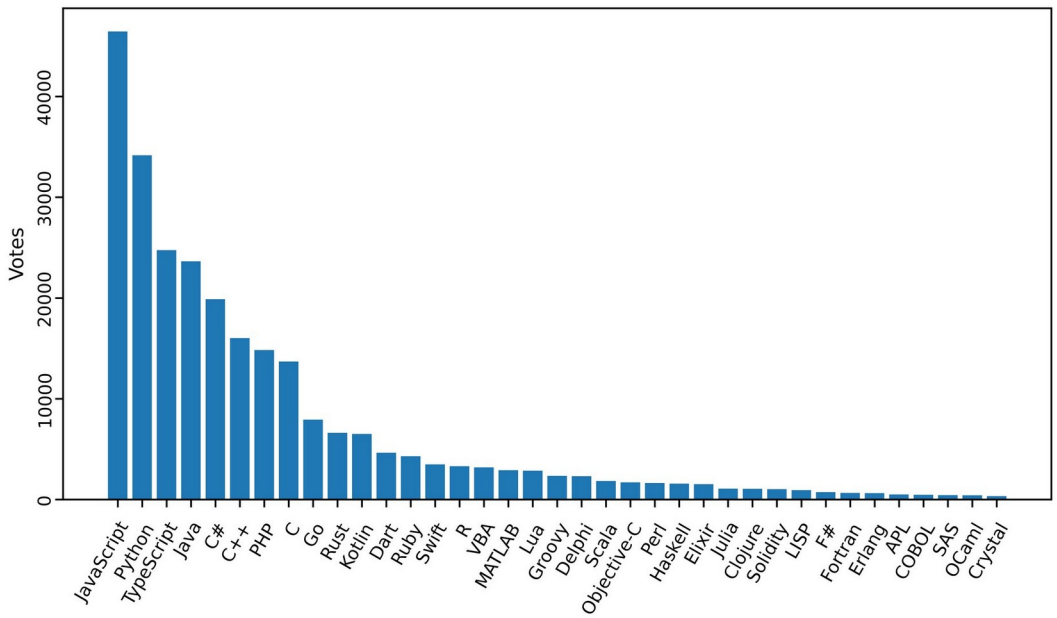

Abbildung 3.1: Die beliebtesten Programmiersprachen im Jahr 2022

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[1] *(engl. Turing-complete)* sind.

[1] Eine Turing-vollständige Sprache kann zur Umsetzung jedes beliebigen Algorithmus benutzt werden. [9], [3]

3.2 Neuere Programmiersprachen

Alle neueren Programmiersprachen, die in der Statistik aufgeführt sind, wurden von anderen Sprachen inspiriert oder sind sogar die Nachfolger dieser Sprachen. Eine dieser Sprachen ist TypeScript, eine statisch typisierte Sprache, die JavaScript um optionale Typ-Annotationen und andere Funktionen erweitert. Sie wird von Microsoft entwickelt und hat in den letzten Jahren vor allem bei großen Webentwicklungsprojekten stark an Popularität gewonnen. Da es sich nur um ein Super-Set von JavaScript handelt, können Entwickler TypeScript schrittweise in bestehende JavaScript-Projekte integrieren. [10]

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. [3, S. 867f], [11]–[13]

Kotlin von JetBrains[2] kann als Nachfolger von Java angesehen werden, der übersichtlicher, klarer und sicherer ist. Die Interoperabilität mit Java ist durch die Verwendung der Java Virtual Machine (JVM) weiterhin gewährleistet. Groovy von Apache[3], Closure und Scala sind weitere Sprachen, die auf der JVM basieren, aber nicht so populär geworden sind wie Kotlin. Einer der größten Faktoren für den Erfolg war die offizielle Unterstützung von Google für Android-Entwicklung mit Kotlin. [3, S. 867f], [15]–[25]

Google hat auch einige eigene Programmiersprachen entwickelt, darunter Go, Dart und Carbon. Go wurde entwickelt, um die Effizienz von kompilierten Sprachen mit der Einfachheit von Skriptsprachen zu kombinieren. Es wird für die Entwicklung von vernetzten Diensten, groß angelegten Webanwendungen und anderer parallel laufender Software verwendet. Dart wurde entwickelt, um eine schnelle Entwicklung von Benutzeroberflächen für viele Plattformen zu ermöglichen. Mit dem Flutter-Framework lassen sich plattform-übergreifende Anwendungen für Android, IOS, Web, Windows, macOS und Linux erstellen. Das jüngste Projekt ist Carbon, ein experimenteller Nachfolger für C++, welcher Ende 2022 veröffentlicht wurde und ist noch lange nicht ausgereift ist. [26]–[29]

Swift hingegen wurde 2014 von Apple als Ersatz für Objective-C eingeführt. Swift soll lesbarer, sicherer und schneller als Objective-C sein und hat bei iOS- und macOS-

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

[3] Die Apache Software Foundation ist eine gemeinnützige Organisation, die Open-Source-Softwareprojekte unterstützt. [14]

Entwicklern schnell an Popularität gewonnen. Swift bietet moderne Features wie funktionale Programmierung, Generics und Typinferenz, die das Schreiben komplexer Softwaresysteme erleichtern. Swift hat mit der Einführung von Swift on the Server, einem Projekt, das die Verwendung von Swift für Backend-Systeme ermöglicht, auch in der serverseitigen Entwicklung an Popularität gewonnen. [30], [31]

3.3 Common Recent Changes

These new languages improve on their predecessors by fixing problems and inconveniences that they encountered and could not remove. Sometimes the new features that improve safety and usability are ported back to the older languages.

3.3.1 Type Systems

A common change is the introduction of static typing, or at least type annotations. This feature allows the compiler or third party tools to check for type errors at compile time, making it easier to catch bugs early in the development process. TypeScript is a very good example of this, as it adds an entire type system on top of JavaScript, which can prevent about 15% of bugs. [32] There is even a proposal to add type annotations to JavaScript, so that tools like TypeScript can skip large parts of the compilation step. [33], [34] Other dynamically typed languages, like Python and PHP already support type hints and Lua has tools that are similar to TypeScript. All the other languages just mentioned (Rust, Kotlin, Go, Dart, Swift) require static types.

- getting more powerful
- type inference
- Type Classes have been adapted
  - easily extend the functionality of uncontrollable 3rd parties
  - Swift protocols
  - C++23 concepts
  - Rust traits
  - Java, C# Interfaces?
- where?
  - Rust, Go, Swift
  - auto in C++
  - state of C, C++, C#, Java, php types?

3.3.2 Null Safety

Tony Hoare considers calling null references, which he added to ALGOL W in 1965, his „billion dollar mistake” [35]. They can represent the absence of a value, but can also cause hard to find bugs if not properly handled. To address this, many programming languages

have introduced null safety features that make it easier for developers to handle and prevent null pointers. These include optional classes, nullable type hints, and optional enums. The null coalescing operator and optional chaining operator are also commonly used to handle these issues and make code more readable.

3.3.3 Functional/Declarative 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 subjective 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.

- TODO
  - naming: [2] S. 344
  - 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`

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!” script shows what this transformation looks like in practice *(main.rys)* :

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

It is just as simple as a Python version that does the same thing, but actually corresponds to this module file *(main.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. [36]

Because functions are a data type

- closures: [2, S. 665]

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. [37], [38]

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. [27], [30], [38]–[40]

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. [41, S. 0], [42, S. 0]

- Python [4]
  - size as large as needed
- PHP [5]
  - int
  - size platform dependent, 32bit|64bit
- Java [6]
- C, C++:
  - char, short, int, long are not always the same size
  - cstdint header: c++ standard S. 493f.
- Go [7]
- Zig (special case) [8]
  - arbitrary size
  - size 1–65535
  - i{Size} eg. i333
  - u{Size} eg. u8
- Rust [9]

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

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

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

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

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

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

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

5.3.2 Float

Generally languages provide floating point data types that adhere to the „IEEE 754 - Floating-Point" arithmetic standard [43] or its ISO adoption „ISO/IEC 60559" [44]. 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 [43]
  - https://ieeexplore.ieee.org/document/8766229
  - same as ISO/IEC 60559
- Accessed: 02.01.2023
  - 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?
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 [45]

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

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') [47, S. 760] languages in which some values may have more than one type. [48, 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 [47, 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

Quellenverzeichnis

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

[4] *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

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

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

[7] „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).

[8] „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).

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

[10] „TypeScript is JavaScript with syntax for types.“ https://www.typescriptlang.org (zugegriffen 1. Februar 2023).

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

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

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

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

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

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

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

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

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

[20] B. Venners und F. Sommers, „The Origins of Scala. A Conversation with Martin Odersky, Part I“. https://www.artima.com/articles/the-origins-of-scala (zugegriffen 15. Januar 2023).

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

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

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

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

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

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

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

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

[29] „Carbon Language. An experimental successor to C++“. https://github.com/carbon-language/carbon-lang (zugegriffen 1. Februar 2023).

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

[31] https://www.swift.org (zugegriffen 1. Februar 2023).

[32] Z. Gao, C. Bird, und E. T. Barr, „To Type or Not to Type: Quantifying Detectable Bugs in JavaScript“, in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, S. 758–769. doi: 10.1109/ICSE.2017.75.

[33] „State of Js 2022. What do you feel is currently missing from JavaScript?“, 2022. https://2022.stateofjs.com/en-US/opinions/#top_currently_missing_from_js (zugegriffen 4. Februar 2023).

[34] „ECMAScript proposal. Type Annotations“, 2022. https://github.com/tc39/proposal-type-annotations (zugegriffen 4. Februar 2023).

[35] „Null References. The Billion Dollar Mistake“. https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/ (zugegriffen 4. Februar 2023).

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

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

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

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

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

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

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

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

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

[45] 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 1. Februar 2023).

[46] 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.

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

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

Anhang A — Rym Quellcode Beispiele

A.1 Data Types

A.1.1 Booleans

```
enum bool {
  true,
  false,
}
use bool.true
use bool.false
```

A.1.2 Numbers

```
type int
type uint

type float
```

A.1.3 Text

```
type char
type string
```

A.1.4 Other

```
type Option<T> = None | Some(T)
use Option.Some
use Option.None

// TODO
// If you want access to the ordinal value of an enum, you
// can specify the tag type.
const Value = enum(u2) {
  Zero,
  One,
  Two,
}
// Now you can cast between u2 and Value.
// The ordinal value starts from 0, counting up by 1 from the
previous member.
test "enum ordinal value" {
  assert_eq(@enumToInt(Value.Zero), 0)
  assert_eq(@enumToInt(Value.One), 1)
```

```
  assert_eq(@enumToInt(Value.Two), 2)
}
```

You can override the ordinal value for an `enum`.

```
enum Value2 {
  Hundred = 100,
  Thousand = 1_000,
  Million = 1_000_000,
}
test "set enum ordinal value" {
  // TODO: How to convert enum to int/uint?
  assert_eq(@enumToInt(Value2.Hundred), 100)
  assert_eq(@enumToInt(Value2.Thousand), 1000)
  assert_eq(@enumToInt(Value2.Million), 1000000)
}
```

You can also override only some values.

```
enum Value3 {
  A,
  B = 8,
  C,
  D = 4,
  E,
}
test "enum implicit ordinal values and overridden values" {
  assert_eq(@enumToInt(Value3.A), 0)
  assert_eq(@enumToInt(Value3.B), 8)
  assert_eq(@enumToInt(Value3.C), 9)
  assert_eq(@enumToInt(Value3.D), 4)
  assert_eq(@enumToInt(Value3.E), 5)
}
```

`Enums` can have methods, the same as `structs`. `Enum` methods are not special, they are only namespaced functions that you can call with dot syntax.

```
enum Suit {
  Clubs,
  Spades,
  Diamonds,
  Hearts,

  pub fn isClubs(self: Suit) bool {
    self == Suit.Clubs
  }
}
test "enum method" {
```

```
  const suit = Suit.Spades
  assert_eq(suit.isClubs(), false)
}
```

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")
}

// TODO
// `@typeInfo` can be used to access the integer tag type of an enum.
enum Small {
  One,
  Two,
  Three,
  Four,
}
test "std.meta.Tag" {
  assert_eq(@typeInfo(Small).Enum.tag_type, u2)
}

// `@typeInfo` tells us the field count and the fields names:
test "@typeInfo" {
  assert_eq(@typeInfo(Small).Enum.fields.len, 4)
  assert_eq(@typeInfo(Small).Enum.fields[1].name, "Two")
}

// `@tagName` gives a `string` representation of an enum value:
test "@tagName" {
  assert_eq(@tagName(Small.Three), "Three")
}
```

A.1.5 Non-exhaustive enum

A non-exhaustive enum can be created by adding an underscore as the last field. Non-exhaustive means the enum might gain additional variants in the future, so when unpacking the enum it is required to add a fall through case.

```
// TODO: Use #NonExhaustive Attribute insted?
#NonExhaustive
enum Number {
  One,
  Two,
  Three,
  _,
}

test "match on non-exhaustive enum" {
  const number = Number.One
  const result = match number {
    .One ⇒ true,
    .Two | .Three ⇒ false,
    _ ⇒ false,
  }
  assert(result)
  const is_one = match number {
      .One ⇒ true,
      else ⇒ false,
  }
  assert(is_one)
}

type bool = true | false
type Testing = One | Other

enum Option<T> = {
  Some(T),
  None,
}

struct Something {
  x: uint,
  y: uint,
}
```

A.2 Statements

A.2.1 Variables

Uninitialized variables must have a value assigned to them before they can be used.

```
const name
const name: OptionalType
```

A.3 Expressions

A.3.1 Block

Block expression evaluate to the value of their last statement:

```
const outer_index = 0

// evaluates to "One"
{
  const array = ["One", "Two", "Three"]
  array[outer_index]
}
```

They, like any other expression, can be used as an initializer for a variable:

```
const outer_index = 0

const one_str = {
  const array = ["One", "Two", "Three"]
  array[outer_index]
}
```

A.3.2 If..Else

```
if expression {
  print("true branch")
}

if expression {
  print("true branch")
} else {
  print("false branch")
}
```

A.3.3 IfVar..Else

```
const maybe_value = Some(2)
if const Some(value) = maybe_value {
  print(value)
}

if const Some(value) = maybe_value {
  print(value)
} else {
  print("None")
}
```

A.4 Functions

```
func func_name(param_1: Type1, param_2: Type2) -> ReturnType {
  ReturnType
}
```

A.4.1 Parameter Default Values

```
func increment(num: int, by = 1) -> int {
  num + by
}

const plus_one = increment(100)        // 101
const plus_50 = increment(100, 50)     // 150
const plus_50 = increment(100, by: 50) // 150
```

A.4.2 Force Named Arguments

```
func testing(pos_or_named: int, .., named: string) { }

testing(2, named: "Hello World!")
testing(2, named: "Hello World!")
testing(pos_or_named: 2, named: "Hello World!")
testing(named: "Hello World!", pos_or_named: 2)
```

A.4.3 Variable Arguments

```
func concat(..strings: [string], sep = "") -> string {
  strings.join(sep)
}

const name = "Mr. Walker"

// all arguments must be of type string
```

```
// they are collected into the strings array
concat("Hello ", name, "!") // "Hello Mr. Walker!"

// arguments defined after the variable argument
// must be passed in by name
concat(2.to_string(), true.to_string(), name, sep: ", ") // "2, true,
Mr. Walker"
concat(sep: ", ", 2.to_string(), true.to_string(), name) // "2, true,
Mr. Walker"
```

A.5 Print Function

```
func print(..args: [impl Display], sep = " ", end = "\n") -> @Io {
  mut output = ""
  for arg in args {
    output.push(arg.fmt())
  }
  /* .. */
}

print("Hello World")               // "Hello World\n"
print("Hello World", end: "")      // "Hello World"

print(true, 2, "three")            // "true 2 three\n"
print(true, 2, "three", sep: ", ") // "true, 2, three\n"
```

A.6 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
```

A.6.1 Imperative approach

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

A.6.2 Declarative approach

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

A.7 Find Summands

```
/// Function for finding two items in a list,
/// that add up to the given sum.
///
/// numbers array must be sorted
func summands(numbers: [int], sum: int) -> Option<[usize; 2]> {
  mut low = 0
  mut high = numbers.len() - 1

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

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

  None
}

const numbers = [-14, 1, 3, 6, 7, 7, 12]
const sum = -13

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