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 demnach sehr sinnvoll, da diese Teile bereits vielen bekannt sind. [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]

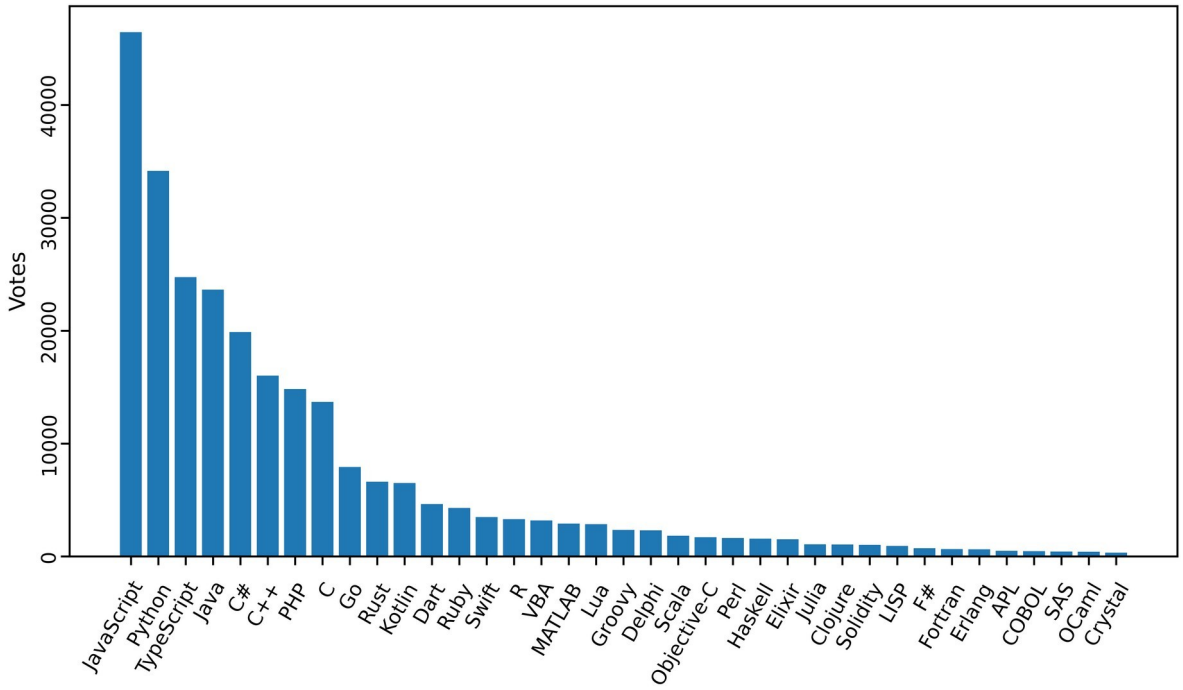

Abbildung 3.1: Die beliebtesten Programmiersprachen im Jahr 2022

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 Neuerungen

Diese neueren Sprachen unterscheiden sich von ihren Vorgängern dadurch, dass Lösungen für viele Probleme und Unannehmlichkeiten gefunden wurden, die von ihren Vorgängern nicht gelöst werden konnten. Neue Funktionen, die zur Verbesserung der Sicherheit und der Benutzerfreundlichkeit beitragen, werden jedoch – soweit dies möglich ist – in die älteren Sprachen übernommen.

3.3.1 Typ-Syteme

Eine gängige Änderung ist die Einführung der statischen Typisierung oder zumindest der Typ-Hinweise *(engl. type annotations)*. Diese Funktion ermöglicht es dem Compiler oder Tools von Drittanbietern, während der Kompilierung auf Typfehler zu prüfen, Dadurch wird es einfacher, Fehler zu einem frühen Zeitpunkt im Entwicklungsprozess zu erkennen. TypeScript ist ein sehr gutes Beispiel dafür, da es ein komplettes Typisierungssystem für JavaScript bereitstellt, welches etwa 15% der häufig auftretenden Fehler verhindern kann. [32, S. 7] Es gibt sogar einen Entwurf, JavaScript mit Typ-Hinweisen zu versehen, so dass Werkzeuge wie TypeScript große Teile des Kompilierungsschrittes überspringen können. [33], [34] Andere dynamisch typisierte Sprachen wie Python und PHP unterstützen bereits Typ-Hinweise, und Lua verfügt über Werkzeuge, die TypeScript sehr ähnlich sind. Alle anderen gerade genannten Sprachen (Rust, Kotlin, Go, Dart, Swift) erfordern statische Typen. [27], [30], [35]–[38]

Das Problem ist, dass ihre Syntax sehr umfangreich werden kann, weshalb viele Typsysteme Typinferenz ermöglichen. Dadurch kann die Sprache automatisch den Typ eines Wertes oder Ausdrucks auf der Grundlage seiner Verwendung bestimmen, ohne dass der Benutzer ihn explizit angeben muss. Ursprünglich kommt dieses System von funktionellen Sprachen wie Haskell, F# und OCaml und hat seinen Weg zu TypeScript, Go, Rust, Kotlin, Dart, Swift und vielen anderen gefunden. Selbst C++, Java und C# haben an einigen Stellen entsprechende Funktionen hinzugefügt. [24, S. 236], [27], [30], [39]–[41], [42, S. 12] [43], [44], [45, S. 13]

Typ-Klassen sind ein weiteres Konzept aus dem Bereich der funktionalen Sprachen, insbesondere Haskell. Sie werden zur Validierung von Verbindungen zwischen benutzerdefinierten oder externen Datenstrukturen verwendet, und zur Erweiterung ihrer Funktionalität. Typ-Klassen wurden in verschiedene Programmiersprachen übernommen, darunter Swift, wo sie als „protocols" bekannt sind, C++, wo sie „concepts" heißen und Rust, wo sie „traits" genannt werden. Java, C# und viele andere Sprachen haben ähnliche, wenn auch weniger leistungsfähige Konstrukte und heißens oft „interfaces". [46, S. 44f], [47], [48, S. 526], [49]–[51]

3.3.2 Null Sicherheit

Tony Hoare betrachtet die von ihm 1965 zu ALGOL W hinzugefügten Nullreferenzen als seinen „Milliarden-Dollar-Fehler" [52]. Sie können das Nichtvorhandensein eines Wertes darstellen, aber auch schwer zu findende Fehler verursachen, wenn sie nicht richtig behandelt werden. Um dieses Problem zu lösen, haben viele Programmiersprachen Sicherheitsfunktionen eingeführt, die es den Entwicklern erleichtern, mit Null-Pointern umzugehen und sie zu vermeiden. Dazu gehören optionale Klassen, Typen, Enums und Typ-Hinweise. Der Null-Coalescing-Operator und der optionale Verkettungsoperator werden ebenfalls häufig verwendet, um diese Probleme zu umgehen und den Quellcode zu verkürzen.

3.3.3 Deklarative und funktionale Programmierung

Die meisten der soeben erwähnten Änderungen haben ihren Ursprung im Bereich der deklarativen und insbesondere der funktionalen Sprachen. Ihre zunehmende Popularität hat dazu geführt, dass ihre Besonderheiten zunehmend in allgemeinen Programmiersprachen eingesetzt werden. Dies ist eine Reaktion auf den Bedarf an intuitivem, lesbarem und effizientem Code sowie an einer besseren Handhabung komplexer Probleme und Datenstrukturen. C++ hat zum Beispiel Funktionen zur Unterstützung der funktionalen Programmierung hinzugefügt. Dies geschah durch die Hinzufügung des *functional* Moduls und die Implementierung von Algorithmen zur Arbeit mit Iteratoren. [48, S. 622–684] TypeScript, Rust, Go, Dart, Kotlin und Swift unterstützen alle Merkmale der funktionalen Programmierung wie z.B. Funktionen höherer Ordnung[4] und unveränderliche Datenstrukturen. Obwohl sie es einfacher machen, zu sehen, wo Seiteneffekte auftreten können, könnte die Unterstützung für deren Kontrolle verbessert werden. [24, S. 331f], [41], [42, S. 126fff], [53], [54]

[4] Funktionen höherer Ordnung sind Funktionen, welche andere Funktionen als Parameter verwenden oder eine Funktion zurüchgeben.

Die Integration dieser Paradigmen in allgemeine Programmiersprachen ermöglicht es Entwicklern, die Stärken sowohl des imperativen als auch des deklarativen Programmierstils zu nutzen, was zu einer flexibleren und ausdrucksstärkeren Programmierung führt. Dieser Trend wird sich in Zukunft wahrscheinlich fortsetzen, da die Nachfrage nach anspruchsvolleren und effizienteren Programmierlösungen weiter steigt. [55]

4. Konzept einer modernen Programmiersprache

Rym ist eine Sprache für allgemeine Zwecke, die auf einer höheren Ebene als die System Programmierung arbeitet, aber dennoch die Verwendung von tieferliegenden Funktionen erlaubt, wenn dies erforderlich ist. Obwohl sowohl ein Interpreter als auch ein Kompiler zur Umsetzung der Sprache verwendet werden können, ist die derzeitige Implementierung ein Interpreter. Diese Beschreibung macht jedoch keine Annahmen über die Implementierung und ermöglicht eine zukünftige kompilierte Version.

4.1 Syntax

Die Syntax folgt den in Kapitel 2.1 genannten Aspekten, insbesondere aus Kapitel 2.1.4. Rym kann als Teil der C-ähnlichen Sprachen betrachtet werden, da es die Syntax von C, Go, Rust, F# und anderen verbindet. Die typischen geschweiften Klammern – „{" und „}" – werden für Struktur-Typen und zum Umschließen von Anweisungsblöcken verwendet. Funktionsdefinitionen beginnen mit „func", genau wie in Go oder Swift, und die Typ-Deklarationen sind denen in F# sehr ähnlich. [27], [45, S. 39f], [56]

In einem Punkt gibt es jedoch große Unterschiede zur typischen C-Syntax: Rym ist, wie Rust und F#, primär ausdrucksbasiert. Das bedeutet, dass die meisten Formen der wertbildenden oder effektverursachenden Auswertung durch die Syntaxkategorie der Ausdrücke gesteuert werden. Im Gegensatz dazu werden Anweisungen meist dazu verwendet, eine spezifische Reihe an Ausdrücken zu enthalten.

Rym hat auch strikte Regeln für die Namensgebung. Namen dürfen nur mit den ASCII-Zeichen „a-z", „A-Z" oder „_" beginnen. Anschließend sind auch die Zahlen „0-9" erlaubt. Dies erleichtert den Benutzern die Zusammenarbeit, da die Sprache sie veranlasst, ähnlichen Code zu schreiben. Variablen-, Funktions- und Modulnamen werden in *snake_case* geschrieben, benutzerdefinierte Typen müssen in *PascalCase* geschrieben werden und die eingebauten Standardtypen wie „bool" bestehen aus einem kleingeschriebenen Wort. Snake-Case wird verwendet, da der Unterstrich die Lesbarkeit verbessert und Pascal-Case Typen von Werten unterscheidet. [2, S. 344fff]

4.2 Programmstruktur

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 script that prints „Hello World!", but actually gets transformed 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. [57]

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

Rym provides a set of unary and binary operators. The unary operators include the „!" operator used to invert the value of a boolean, the „-" operator used to negate numbers and the „?" operator. The last operator unwraps types like *Option* or *Result* and returns the value out of the current function if an unsuccessfull variant (*None*, *Err*) is met.[5] Rym does not support the „++" and „--" operators, which are often used increment or decrement an integer, as it is often unclear what these operators exactly do. The same

The assignment operators can only be used with variables that are marked as mutable (via mut keyword).

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.

[5] Beispiel siehe Kapitel A.2.5

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. Fazit

Zusammenfassend lässt sich sagen, dass eine Programmiersprache der nächsten Generation wahrscheinlich so konzipiert sein wird, dass sie den sich ändernden Bedürfnissen von Entwicklern und der wachsenden Komplexität moderner Softwaresysteme gerecht wird. Mit Fortschritten in Bereichen wie künstlicher Intelligenz, Cloud Computing und Data Science werden sie wahrscheinlich einen starken Fokus darauf haben, in diesen Bereichen effizienter zu sein. Rym und ähnliche moderne Programmiersprachen erreichen dies durch eine intuitivere und aussagekräftigere Syntax, erhöhte Sicherheit und bessere Unterstützung für Parallelisierung. Unveränderliche Datenstrukturen, solide Typ-Systeme und vor allem funktionale Programmierung scheinen gute Werkzeuge zu sein, um diese Ziele zu erreichen. Letztendlich wird die Zukunft der Programmiersprachen von den aktuellen Technologien geprägt sein, die wir und andere zur Unterstützung unserer Arbeit verwenden. Es wird interessant sein, zu sehen, welche neuen Innovationen und Fortschritte in diesem Bereich entstehen werden.

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] „typing. Support for type hints“. https://docs.python.org/3/library/typing.html?highlight=typ#module-typing (zugegriffen 7. Februar 2023).

[36] „PEP 484. Type Hints“. https://peps.python.org/pep-0484 (zugegriffen 7. Februar 2023).

[37] „PHP Language Reference. Types. Type declarations“. https://www.php.net/manual/en/language.types.declarations.php (zugegriffen 7. Februar 2023).

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

[39] „Handbook. Type Inference“. https://www.typescriptlang.org/docs/handbook/type-inference.html (zugegriffen 7. Februar 2023).

[40] „The Rust Reference. Let statements“. https://doc.rust-lang.org/stable/reference/statements.html#let-statements (zugegriffen 7. Februar 2023).

[41] „The Rust Reference. Influences“. https://doc.rust-lang.org/stable/reference/influences.html#influences (zugegriffen 7. Februar 2023).

[42] „Dart Programming Language Specification. 6th edition draft“. https://spec.dart.dev/DartLangSpecDraft.pdf (zugegriffen 7. Februar 2023).

[43] „C++ keyword. auto“. https://en.cppreference.com/w/cpp/keyword/auto (zugegriffen 7. Februar 2023).

[44] „Java Language Specification. Type Inference“. https://docs.oracle.com/javase/specs/jls/se19/html/jls-18.html (zugegriffen 7. Februar 2023).

[45] „The F# 4.1 Language Specification“. https://fsharp.org/specs/language-spec/4.1/FSharpSpec-4.1-latest.pdf (zugegriffen 7. Februar 2023).

[46] „Haskell 2010 Language Report“. https://www.haskell.org/definition/haskell2010.pdf (zugegriffen 7. Februar 2023).

[47] „Swift Book. Protocols“. https://docs.swift.org/swift-book/LanguageGuide/Protocols.html (zugegriffen 7. Februar 2023).

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

[49] „The Rust Reference. Traits“. https://doc.rust-lang.org/reference/items/traits.html (zugegriffen 7. Februar 2023).

[50] „Java Language Specification. Interfaces“. https://docs.oracle.com/javase/specs/jls/se19/html/jls-9.html (zugegriffen 7. Februar 2023).

[51] „C# 7.0 draft specification. Interfaces“. https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/interfaces (zugegriffen 7. Februar 2023).

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

[53] „Functional programming in Go“. https://blog.logrocket.com/functional-programming-in-go (zugegriffen 8. Februar 2023).

[54] „Swift Book. Closures“. https://docs.swift.org/swift-book/LanguageGuide/Closures.html (zugegriffen 8. Februar 2023).

[55] „Functional programming is finally going mainstream“. https://github.com/readme/featured/functional-programming (zugegriffen 8. Februar 2023).

[56] „Swift Book. Functions“. https://docs.swift.org/swift-book/LanguageGuide/Functions.html (zugegriffen 8. Februar 2023).

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

Anhang A — Rym Überblick

A.1 Datentypen

A.1.1 Booleans

```
type bool = true | false
use bool.true
use bool.false
```

A.1.2 Ganzzahlen

```
type int
type isize
type uint
type usize

0 44 1834
```

A.1.3 Dezimalzahlen

```
type float

0.1 -1.4
-444. // same as -444.0
```

A.1.4 Text

```
type char

'a' '\n' '\t' '\u{2192}'

type string

"Hello World!\n" "testing"
```

A.1.5 Structures

```
type Vec3 = {
  x: float,
  y: float,
  z: float,
}

const pos = Vec3 { x: 2.5, y: -1.2, z: 0.0, }
```

A.1.6 Enumerationen / Union Typen

```
type EnumName =
  | Variant1
  | Variant2
  | VariantN

const variant = EnumName.VariantN
```

Für kleine Enumerationen gibt es auch eine kürzere Syntax:

```
type SmallEnum = VerySmall | ActuallyLarge
```

Jede Variante kann mit einem bestimmten Wert verknüpft werden. Diese Werte werden automatisch ausgewählt, wenn sie nicht definiert sind, und werden normalerweise durch eine Integerzahl dargestellt:

```
type EnumName =
  | Variant1 = 1
  | Variant2 = 2
  | VariantN = 156

const variant_n = EnumName.VariantN
const repr = variant_n as uint        // 156
const variant_2 = 2 as EnumName       // EnumName.Variant2
```

A.1.7 Eingebaute Enumerationen

Der Option-Typ wird für etwas verwendet, das möglicherweise keinen Wert hat:

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

use Option.Some
use Option.None
```

Der Result-Typ wird verwendet, um anzugeben, dass etwas entweder einen Wert oder einen Fehler enthalten kann:

```
type Result<T, E> =
  | Ok(T)
  | Err(E)

use Result.Ok
use Result.Err
```

A.2 Expressions

A.2.1 Block

Blockausdrücke geben den Wert ihrer letzten Anweisung zurück:

```
const outer_index = 0

// gibt "One" zurück
{
  const array = ["One", "Two", "Three"]
  array[outer_index]
}
```

Sie können, wie jeder andere Ausdruck, als Initialisierer für eine Variable verwendet werden:

```
const outer_index = 0

const str = {
  const array = ["One", "Two", "Three"]
  array[outer_index]
}
print(str) // "One\n"
```

A.2.2 If..Else

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

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

A.2.3 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.2.4 Match

Dient der Destrukturierung komplexer Datentypen wie Enumerationen:

```
match maybe_value {
  Some(value) ⇒ print(value)
  None ⇒ print("None")
}
```

Und funktioniert genauso gut mit Strukturen:

```
match pos {
  Vec3 { x: 0.0, .. } ⇒ print("Position: on ground")
  Vec3 { y: 0.0, z: 0.0, .. } ⇒ print("Position: on x axis")
  Vec3 { x: 0.0, z: 0.0, .. } ⇒ print("Position: on y axis")
  Vec3 { x: 0.0, y: 0.0, .. } ⇒ print("Position: on z axis")
  Vec3 { x: 0.0, y: 0.0, z: 0.0 } ⇒ print("Position: at origin")
  Vec3 { x, y, z } ⇒ print("Position:", x, y, z)
}
```

Es ist auch möglich, “_” als Platzhalter zu verwenden, um alle übrigen Fälle zu erfassen:

```
match pos {
  Vec3 { x: 0.0, y: 0.0, z: 0.0 } ⇒ print("at origin")
  _ ⇒ print("not at origin")
}
```

A.2.5 „?” Operator

```
func read_to_string(path: string) → Result<string, IoError> {
  mut file = File.open(path)?
  mut data = ""
  file.read_to_string(mut data)?
  Ok(data)
}
```

A.3 Statements

A.3.1 Variables

```
const const_name = "Hello World!"

mut mut_name = "Hello "
mut_name += "Universe!"

print(const_name, mut_name) // "Hello World! Hello Universe!/n"
```

Nicht initialisierte Variablen müssen mit einem Wert versehen werden, bevor sie verwendet werden können:

```
const name

if condition { name = "Simon" } else { name = "Robert" }

// erlaubt, da "name" immer einen Wert hat
print(name)
```

A.3.2 Imports / Use

```
use std.fs.{self, File}

const path = "./dad_jokes.txt"
const joke = "What should you do if you meet a giant? Use big words."

const create_result = File.create(path)
const write_result = fs.write(path, joke)
const data = fs.read_to_string(path).unwrap()

print(data) // "What should you do if you meet a giant?
            //  Use big words.\n"
```

A.3.3 Functions

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

const of_return_type = func_name(of_type_1, of_type_2)
```

A.3.3.1 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.3.3.2 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.3.3.3 Variable Arguments

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

const name = "Mr. Walker"

concat("Hello ", name, "!") // "Hello Mr. Walker!"

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.3.3.4 Eingebaute 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.3.4 Implementations

Jeder Typ kann mehrere „impl”-Blöcke haben, welche sich im selben Modul wie der Typ befinden müssen. Sie enthalten statische/nicht-statische Methoden und interne Typen. Standardmäßig sind Methoden privat und können mit dem Schlüsselwort „pub” öffentlich gemacht werden:

```
impl bool {
  pub func then_some<T>(self, value: T) -> Option<T> {
    if self { Some(value) } else { None }
  }
}
```

A.3.5 Traits / Type Classes

Traits werden verwendet, um gemeinsame Funktionen zwischen Typen zu definieren:

```
trait Default {
  func default() -> Self
}
```

Sie werden für bestimmte Typen durch separate Implementierungen umgesetzt:

```
impl Default for bool { func default() -> Self { false } }

impl Default for int { func default() -> Self { 0 } }

impl Default for string { func default() -> Self { "" } }

impl<T> Default for [T] { func default() -> Self { [] } }
```

Auf diese Weise werden auch Iteratoren in Rym definiert:

```
trait Iterator {
  type Item

  func next(mut self) -> Option<Self.Item>
}
```

Diese Iteratoren können verwendet werden, um einen einfachen Zähler wie den folgenden zu implementieren oder um beispielsweise alle Elemente eines Arrays einzeln zu durchlaufen.

```
type Counter = { count: usize, max: usize }

impl Iterator for Counter {
  type Item = usize

  func next(mut self) -> Option<Self.Item> {
    self.count += 1

    if self.count ⩽ self.max { Some(self.count) } else { None }
  }
}

mut counter = Counter { count: 0, max: 3 }

print(counter.next()) // Some(1)
print(counter.next()) // Some(2)
print(counter.next()) // Some(3)
print(counter.next()) // None
```

Anhang B — Rym Quellcode Beispiele

B.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
```

B.1.1 Imperative approach

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

B.1.2 Declarative approach

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

B.2 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")
}
```