Berufliches Schulzentrum für Bau und Technik Dresden
Fachoberschule
Fachrichtung Technik

Facharbeit im Fach Informatik

Merkmale und Konzepte einer Programmiersprache der nächsten Generation

Verfasser: Simon Sommer
Klasse: FOS21B
Betreuer: Herr Rottmann
Ort, Datum: Dresden, 26.03.2023

Inhaltsverzeichnis

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. TODO: Mention Typst as a successor to Latex. 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 umstritten, da es unterschiedliche Meinungen über den Wert jeder bestimmten Spracheigenschaft im Vergleich zu anderen gibt. Obwohl es auch aufgrund der Unterschiedlichen Anwendungsbereiche einer Programmiersprache keine allgemeingültigen Vorteile gibt, werden die folgenden 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. TODO: Cite source proving that code is more often read than written. Da die Wartungsfreundlichkeit zu einem großen Teil durch die Lesbarkeit von Programmen bestimmt wird, ist sie eine wichtige Maßnahme zur Verbesserung der Qualität von Programmen und Programmiersprachen. In den folgenden Unterabschnitten werden Merkmale beschrieben, die zur Lesbarkeit einer Programmiersprache beitragen. [2, S. 42f]

2.1.1 Einfachheit

Die 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 Überladungen die Sprache, da sie die Anzahl der verschiedenen Operatoren reduzieren. [2, S. 43f] [2, S. 664]

TODO: write more?. Eine dritte Charakteristik, welche die Einfachheit einer Programmiersprache negativ beeinträchtigt, ist das Vorhandensein zu vieler 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 Programmiersprache. 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 Programmiersprache bedeutet, dass eine kleine Menge von primitiven Konstrukten auf wenige Arten kombiniert werden können, um die Kontroll- und Datenstrukturen der Sprache aufzubauen. Außerdem sollte jede mögliche Kombination von Primitiven legal und sinnvoll sein. Ein Mangel an Orthogonalität führt 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:

```
let use_timeout = 1
```

Die Bedeutung dieser Anweisung ist unklar, TODO: Explain why it is unclear (time in ms/s vs truthy value). und kann in einer Sprache mit Boolean-Typen wesentlich klarer dargestellt werden:

```
let 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 den 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 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 sollte die Syntax dem Programmierer dabei helfen, so wenig wie möglich im Programm herumzuspringen. Das macht es einfacher, sich auf das Schreiben des Codes zu konzentrieren. Dies kann durch eine Syntax erreicht werden, die wie im Englischen TODO: Missing Footnote., auf einer Leserichtung von links nach rechts basiert. [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. Dadurch verringert sich auch die Fehleranfälligkeit. In anderen Worten: Eine Sprache sollte die Anzahl der Dinge minimieren, über die der Nutzer gleichzeitig nachdenken muss. [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 Ergebnisse können schädlich sein, da die Implementierungen nicht unbedingt über alle Ausführungen, Versionen oder Optimierungsstufen konsistent bleiben. [4, S. 21] [5, S. 190]

Verhaltensweisen, die in der Spezifikation nicht erwähnt werden, nennt man *unspezifiziertes Verhalten*. *Undefiniertes Verhalten* sind Verhaltensweisen, die zwar erwähnt werden, deren Verhalten aber nicht festgelegt ist. Beide 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 möglichst wenige solcher Verhaltensweisen aufzeigen. [4, S. 21]

2.3.2 Sicherheit

Zusätzlich zu den Sicherheitslücken, die durch unspezifiziertes oder undefiniertes Verhalten entstehen, gibt es weitere Probleme. Diese sollten angegangen werden, 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- und Typ-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 Rückwärtskompatibilität 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, 7, 8]

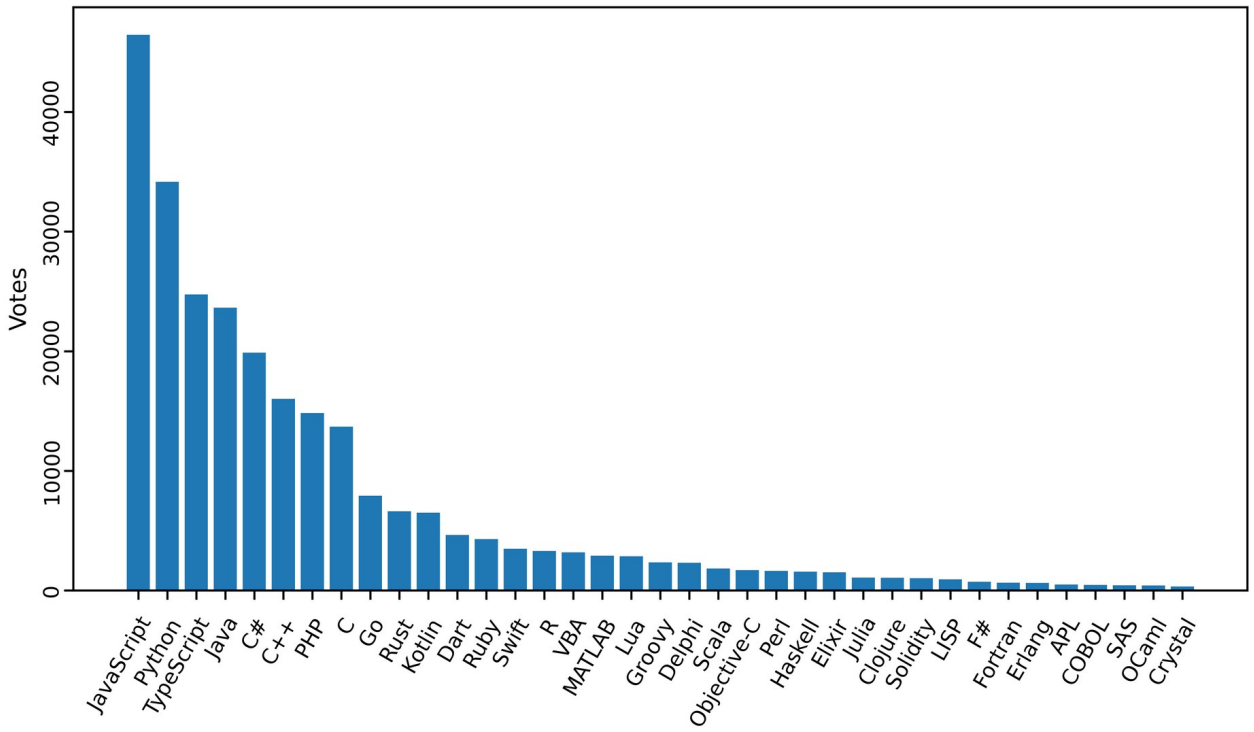

Abbildung 1: Die beliebtesten Programmiersprachen im Jahr 2022

The original data also mentions HTML, CSS, SQL, Bash/Shell, and PowerShell. These are not considered here because they are highly specialised and in part not Turing-complete.

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 sind.

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 Erweiterung von JavaScript handelt, können Entwickler TypeScript schrittweise in bestehende JavaScript Projekte integrieren. [9]

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. [10] [3, S.867f] [11] [12]

Kotlin von JetBrains 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 , 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. [13] [14] [15] [16] [17] [18] [19] [3, S.867f] [20] [21] [22] [23]

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++. Es wurde erst Ende 2022 veröffentlicht und ist noch lange nicht ausgereift. [24] [25] [26] [27]

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-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 Ver-

wendung von Swift für Backend-Systeme ermöglicht, auch in der serverseitigen Entwicklung an Popularität gewonnen. [28] [29]

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. [30, S. 7] Es gibt sogar einen Entwurf, JavaScript mit Typ-Hinweisen zu versehen, so dass Werkzeuge wie TypeScript große Teile des Kompilierungsschrittes überspringen können. [31] [32] 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. [33] [34] [35] [25] [36] [28]

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. [37] [25] [38] [39] [22, S. 236] [40, S. 12] [28] [41] [42] [43, S. 13]

Typ-Klassen sind ein weiteres Konzept aus dem Bereich der funktionalen Sprachen, insbesondere Haskell. Sie werden zur Validierung von Verbindungen zwischen benutzer-definierten oder externen Datenstrukturen verwendet, und zur Erweiterung ihrer Funktion-alität. 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 heißen oft „interfaces“. [44, S. 44f] [45] [46, S. 526] [47] [48] [49]

3.3.2 Null Sicherheit

Tony Hoare betrachtet die von ihm 1965 zu ALGOL W hinzugefügten Nullreferenzen als seinen „Milliarden-Dollar-Fehler“ [50]. 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. [46, S. 622-684] TypeScript, Rust, Go, Dart, Kotlin und Swift unterstützen alle Merkmale der funktionalen Programmierung wie z.B. Funktionen höherer Ordnung^[Funktionen höherer Ordnung sind Funktionen, welche andere Funktionen als Parameter verwenden oder eine Funktion zurüchgeben.] und unveränderliche Datenstrukturen. Obwohl sie es einfacher machen, zu sehen, wo Seiteneffekte auftreten können, könnte die Unterstützung für deren Kontrolle verbessert werden. [39] [51] [40, S. 126fff] [22, S. 331f] [52]

Die Integration dieser Paradigmen in allgemeine Programmiersprachen ermöglicht es Entwicklern, die Stärken der imperativen und deklarativen Programmierstile gleichzeitig zu nutzen, was zu einer flexibleren und ausdrucksstärkeren Programmierung führt. Dieser Trend wird sich in Zukunft wahrscheinlich fortsetzen, da die Nachfrage nach anspruchs-volleren und effizienteren Programmierlösungen weiter steigt. [53]

3.3.4 Modulare Projektstruktur

Abgesehen von den Unterschieden zwischen deklarativen und imperativen Programmiersprachen haben sie eine Veränderung gemeinsam. Die Systeme zur Modularisierung von Projekten und zum Importieren eines Elements aus einem Modul in ein anderes haben sich stark verbessert. In älteren Sprachen geschah dies traditionell durch einfaches Einfügen des Inhalts einer anderen Datei. Heute sind eingebaute Paketsysteme der De-facto-Standard. C+

+20 bietet Unterstützung für Module, die auf diese Weise funktionieren, und JavaScript hat 2015 ein ähnliches System eingeführt. Rust, Go, Dart, Kotlin und Swift unterstützen diese Funktionalität seit ihrer Entwicklung. [46, S. 232-242] [54, S. 390-425] [55] [25] [22, S. 295f] [56]

4 Konzept einer Programmiersprache der nächsten Generation

Rym ist eine Sprache für allgemeine Zwecke, die auf einer höheren Ebene als die System Programmierung arbeitet, aber dennoch die Verwendung von tieferliegenden Funktionen gestattet. Die aktuelle Implementierung ist nur ein Interpreter. Rym kann jedoch auch durch einem Kompiler umgesetzt werden.

Die Git-Repository für dieses Projekt ist unter https://github.com/CreatorSiSo/rym zu finden.

4.1 Syntax

Die Syntax folgt den in sowie genannten Aspekten und ist so aufgebaut, dass das Prinzip des Spiralcurriculums angewendet werden kann. Rym ermöglicht es, zunächst sehr simplen Code zu schreiben, ohne alle Details der Sprache kennen zu müssen. Später kann der Benutzer durch kleine Änderungen mehr Kontrolle erlangen.

Rym kann als Teil der C-ähnlichen Sprachen betrachtet werden, da die Syntax ähnlich der von C, Go, Rust, TypeScript, F# ist. 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 TypeScript sowie F# sehr ähnlich. [25, 57] [43, S. 39f]

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 und die Namen von komplexen Typen müssen in *PascalCase* geschrieben werden. Snake-Case wird verwendet, da der Unterstrich die Lesbarkeit verbessert. Pascal-Case vereinfacht die Unterscheidung von komplexen Datentypen und Werten. [2, S. 344fff]

4.2 Kommentare

Die typischen Kommentare von Sprachen im C-Stil sind auch Teil von Rym. Um den Rest einer Zeile auszukommentieren wird `//` benutzt. Wenn er über mehrere Zeilen reichen soll sind `/*` und `*/` die Begrenzungen für den Kommentar. Außerdem kann die zweite Art in einnander verschachtelt werden, solange die Anzahl der öffnenden `/*` den der schließenden `*/` entspricht.

4.3 Programmstruktur

Das Ausführungsmodell von Rym basiert auf Paketen, die entweder eine Bibliothek oder eine ausführbares Programm sein können. Pakete, die eine Funktion namens `main` enthalten, können als eigenständige Programme ausgeführt werden, während andere wiederverwendbare Bibliotheken sind. Diese Pakete bestehen aus Modulen, Funktionen, Konstanten und anderen Definitionen, die eine baumartige Struktur bilden und für die Organisation des Codes sorgen.

Quelldateien, die ein Top-Level-Modul darstellen, verwenden die Erweiterung `.rym`. Rym erlaubt auch die Ausführung von `.rys` Skriptdateien, um schnelle Tests zu ermöglichen und Personen, die die Sprache lernen, den Einstieg zu erleichtern. Diese Skripte funktionieren ähnlich wie eine JavaScript- oder Python-Datei, wo alle Anweisungen sofort ausgeführt werden, ohne dass eine Eingangsfunktion definiert werden muss. Rym erreicht dies, indem es den Inhalt des Skripts einfach in eine Hauptfunktion einbettet.

Ein Blick auf das typische „Hello World!“ Skript zeigt, wie diese Umwandlung in der Praxis aussieht (`main.rys`) :

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

Es ist genauso simpel wie ein Python-Skript, welches `"Hello World!\n"` ausgibt. Tatsächlich wird es aber in diese Moduldatei umgewandelt (`main.rym`) :

```
func main() {
  print("Hello World!")
}
```

4.4 Variablen

Variablen in Rym sind entweder unveränderlich, was vorzuziehen ist, oder veränderbar. Das Schlüsselwort `let` kann verwendet werden, um eine neue Variable zu erstellen. Beispiele für die Erstellung von Variablen sind in Abschnitt A.3.1 zu finden.

4.5 Datentypen

Typen in Rym werden im Allgemeinen mit dem Schlüsselwort `type` deklariert. Dem Schlüsselwort kann eine Syntax folgen, die definiert, welche Werte dieser Typ enthält. Alle Typen in Rym können durch das Konzept der algebraischen Datentypen *(ADT/ADTs)* beschrieben werden. Dieses Modell verknüpft jeden Typ mit der Menge der Werte, die er darstellen kann.

4.5.1 Unit

Der Unit-Typ hat seinen Ursprung in funktionalen Sprachen, ist aber dem void-Typ von C, C++, JavaScript usw. sehr ähnlich. Er hat genau einen Wert und wird von Funktionen und Ausdrücken ohne Ergebnis zurückgeben.

4.5.2 Booleans

Boolean-Typen gehören zu den einfachsten aller Typen und sind in den meisten seit 1960 entwickelten allgemeinen Programmiersprachen enthalten. Der Boolean-Typ in Rym heißt `Bool`, wie in Python, PHP, C#, Go, Rust, Swift und vielen anderen. Er wurde nach *George Boole* benannt, einem Pionier der mathematischen Logik. [58, 59, 25, 36, 28]

Booleans haben nur zwei mögliche Werte, `True` und `False`. Deshalb verwendet man sie häufig, um Schalter oder Marker in Programmen darzustellen. Obwohl auch andere Typen, wie z.B. Integer, für diese Zwecke verwendet werden können, ist die Verwendung von Boolescher Werten besser lesbar. In C und C++ können numerische Ausdrücke immer noch so verwendet werden, als wären sie Booleans. In den nachfolgenden Sprachen, Java und C#, ist dies nicht der Fall. Daher wird Rym dies auch nicht erlauben. [60, 59] [2, S. 404]

Ein Boolescher Wert könnte durch ein einziges Bit dargestellt werden. Da aber auf vielen Rechnern nicht effizient auf ein einzelnes Bit im Speicher zugegriffen werden kann, werden sie oft in der kleinsten effizient adressierbaren Speicherzelle gespeichert, in der Regel einem Byte. Da dieses Detail unwichtig ist, spezifiziert Rym nicht, wie Boolesche Werte zu speichern sind. [2, S. 404f]

4.5.3 Verwedung von Booleans

Ein Boolescher Wert kann mit den Begriffen `True` oder `False` erstellt werden.

```
let var_1 = True
let var_2 = False
```

In Rym verwenden die Kontrollflussausdrücke `if` und `while` Boolesche Werte, um zu entscheiden, welcher Code ausgeführt werden soll und vice versa.

```
let condition = True
if condition { /* do something once */ }
while condition { /* do something forever */ }
```

4.5.4 Integers

Ein weiterer sehr verbreiteter Datentyp ist der Integer, welcher nur in manchen Sprachen existiert. Wie in zu sehen, wird die Definition einer bestimmten Größe für Integers von noch weniger Programmiersprachen unterstützt. Einige Sprachen unterstützen vorzeichenlose Integer-Typen. Diese Typen werden zur Speicherung von natürlichen Zahlen und häufig auch von binären Daten verwendet. Acht Bit große vorzeichenlose Integers können zum Beispiel genau einen Byte darstellen.

Die Typen für C und C++, welche in dargestellt werden, verwenden den `cstdint` Header, da ihre Standards die Größen für Integers nicht vorgeben und sie der Implementierung überlassen. Die Typen aus der Header-Datei müssen jedoch genau diese Größe haben. [61]

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

Abbildung 2: Unterstützte Integer-Formate

Der `cstdint` Header, Go und Rust haben Namen für diese Typen, die eindeutig angeben, wie groß ein Wert ist. Rym verwendet einfach die Version von Rust, da sie die kürzeste ist. Diese Namen ermöglichen es auch in der Zukunft, Integers beliebiger Größe hinzuzufügen.

4.5.5 Floats

Die meisten Sprachen bieten Fließkommazahl-Typen an, die dem *IEEE-754* [62] Standard für Fließkommaarithmetik oder seiner ISO-Adaption *ISO-60559* [63] entsprechen. Einige Beispiele dafür sind in Abbildung 3 zu sehen. C und C++ sind mit einem Fragezeichen gekennzeichnet, da ihre Standards die Verwendung von *IEEE-754* nicht vorschreiben.

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	—	—	—	—	—

Abbildung 3: Unterstützte *IEEE-754* Fließkommazahl-Formate

Rym bietet – wieder genau wie Rust – die Typen `F32` und `F64"` um Ergänzungen und Änderungen in der Zukunft zu ermöglichen.

4.5.6 Strings

Für die Arbeit mit Text bietet Rym den eingebauten Typ `String`. Dieser kann verwendet werden, um UTF-8 kodierten Text zu speichern. Ein neuer String beginnt und endet mit `"`. Dazwischen können alle mit UTF-8 enkodierbaren Zeichen stehen.

```
"Hello World\n"
```

Für spezielle unsichtbare Zeichen, wie Zeilenumbrüche oder Tabs gibt es spezialle Sequenzen, die mit `\` beginnen. Sie verbessern die Lesbarkeit dieser Zeichen. Beispiels-weise steht `\n` im oben zu sehenden Code für einen Zeilenumbruch.

4.5.7 Komplexe Datentypen

Rym erlaubt neben den bereits erwähnten primitiven Datentypen auch die Verwendung komplexer Typen. Zu diesen Typen gehören Arrays, die eine Liste von Elementen desselben Typs darstellen. Vereinigungstypen oder auch Enumerationen werden verwendet, um darzustellen, dass ein Wert verschiedene Typen haben kann. Strukturen können schließlich verwendet werden, um verschiedene Datentypen in einem Typ zu kombinieren. Beispiele für die Erstellung und Verwendung verschiedener komplexer Datentypen finden sich in Abschnitt A.1.5, Abschnitt A.1.6 und Abschnitt A.3.4.

4.6 Funktionen

Funktionen in Rym werden mit dem Schlüsselwort `func` deklariert, gefolgt von einem Namen für die Funktion, einer durch Kommata getrennten Liste von Parametern und ihren jeweiligen Typen sowie dem Rückgabetyp der Funktion. Eine Funktion mit mehreren Parametern sieht demnach folgendermaßen aus:

```
func name(param_0: Type0, param_1: Type1) -> ReturnType {
  // ..
}
```

Wie in jeder Sprache im *C-Stil* kann diese Funktion aufgerufen werden, indem man den Namen der Funktion angibt, gefolgt von einer Liste an Argumenten in Klammern, getrennt durch Kommata. Die Argumente müssen mit den Typen der in der Funktionsdeklaration definierten Parameter übereinstimmen. Die vorherige Beispielfunktion würde wie folgt aufgerufen werden:

```
name(argument0, argument1)
```

4.6.1 Speicherbereich der Funktion

Wenn eine Funktion aufgerufen wird, werden die Variablen und Werte im aktuellen Speicherbereich vorübergehend ausgesetzt, und es wird ein neuer Speicherbereich für die Funktion erstellt. Dieser neue Bereich enthält die Parameter der Funktion als Variablen, die mit den Werten der an die Funktion übergebenen Argumente initialisiert sind. Der Hauptteil der Funktion wird dann in diesem neuen Bereich ausgeführt, und alle in der Funktion definierten Variablen oder Werte sind außerhalb des Bereichs der Funktion nicht zugänglich. Wenn die Funktion beendet wurde, wird der Bereich der Funktion zerstört und der ursprüngliche Speicherbereich wird wiederhergestellt. Ein Beispiel und eine Erklärung sind in Abschnitt A.3.3.1 zu finden.

4.6.2 Rückgaben

Alle Funktionen in Rym müssen mit einem Rückgabetyp definiert werden. Dies ermöglicht es Tools, eine Funktion zu analysieren, ohne sich ihren Körper ansehen zu müssen. Der Rückgabetyp wird nach dem Pfeilsymbol `→` in der Funktionsdefinition angegeben. Wenn während der Ausführung eine Return-Anweisung auftritt, wird der Wert, der auf die Anweisung folgt, an den Funktionsaufrufenden zurückgegeben.

Im folgenden Beispiel nimmt die Funktion `multiply` zwei Integer-Argumente, `x` und `y`, und gibt das Ergebnis ihrer Multiplikation als Integer zurück.

```
func multiply(x: I32, y: I32) -> I32 {
  let result = x * y
  return x * y
}
```

Der Funktionskörper enthält eine einzige Anweisung. Die lokale Variable `result` wird definiert und mit dem Wert von `x * y` initialisiert. Die Return-Anweisung gibt dann an, dass der Wert von `result` als Ergebnis der Funktion zurückgegeben werden soll. Es ist jedoch nicht notwendig, in jeder Funktion eine Return-Anweisung anzugeben. Der Wert des letzten Ausdrucks im Funktionskörper wird automatisch zurückgegeben.

```
func multiply(x: I32, y: I32) -> I32 {
  x * y
}
```

In diesem Fall wird der Wert des Ausdrucks `x * y` als Ergebnis der Funktion zurückgegeben, ohne dass eine separate Return-Anweisung erforderlich ist. Dies ermöglicht einen kürzeren und besser lesbaren Code, da die Return-Anweisung oft überflüssig ist, wenn der Funktionskörper nur einen einzigen Ausdruck enthält.

Weitere Eigenschaften der Funktionen aus Rym sind in Abschnitt A.3.3 beschrieben.

4.6.3 Closures, Anonyme Funktionen

Rym versucht, imperative und deklarative Ansätze in der Programmierung zu kombinieren. Deshalb ist es möglich Funktionen wie Daten zu behandeln. Funktionsdeklarationen sind jedoch Anweisungen und können nicht als Ausdrücke verwendet werden. Dies macht die Übergabe von Funktionen in einem Funktionsaufruf umständlich. Closures lösen dieses Problem, da sie ein Ausdruck sind und die Erstellung von Funktionen direkt Vorort ermöglichen. [[2] S. 665ff]

Die genauen Typen der Parameter und des Rückgabewerts werden aus dem Kontext abgeleitet. Falls erforderlich, können sie jedoch in der gleichen Weise wie bei Funktionsdeklarationen notiert werden. Ihre Syntax lehnt sich an die Funktionsdeklarationen von Rym an und kann wie folgt geschrieben werden:

```
(param_0, param_1) -> { /* .. */ }
(param_0, param_1) -> { /* .. */ }
(param) -> { /* .. */ }
(param) -> /* .. */
() -> /* .. */
```

Siehe Abschnitt A.2.9 und Abschnitt B.1.2 für Beispiele, wie Closures in Rym verwendet werden.

4.7 Operatoren

Rym bietet eine Reihe von unären und binären Operatoren. Zu den unären Operatoren gehören der `!` Operator, mit dem der Wert eines booleschen Wertes umgekehrt werden kann, der `-` Operator, der zum Negieren von Zahlen verwendet wird und der `?` Operator. Dieser Operator entpackt Typen wie `Option` oder `Result` und gibt bei einer erfolglosen Variante – `None` oder `Err` – den Wert der aktuellen Funktion zurück. *(Siehe Abschnitt A.2.4)* Rym unterstützt die Operatoren `++` und `--` nicht, mit denen häufig ein Integer inkrementiert oder dekrementiert wird. Sie können als Präfix oder Postfix verwendet werden, wodurch es unklar ist, wie genau diese Operatoren funktionieren.

In Rym gibt es zwei Kategorien von binären Operatoren. Diejenigen der ersten Kategorie erzeugen einen neuen Wert , und die anderen ändern eine bestehende Variable. Die Zuweisungsoperatoren können nur mit Variablen verwendet werden, die als veränderbar gekennzeichnet sind.

Die folgende sehr typische Syntax, eine Alternative zu `++` und `--` wird ebenfalls unterstützt:

```
let mut counter = 0

counter = counter + 1
counter += 1

print(counter) // "2\n"
```

4.8 Control Flow

Although Rym can be used to model control flow in a purely functional way it also supports imperative control flow expressions like (`if` , `else` , `match`) and loops (`loop` , `while` , `for`). Examples for how these expressions are used can be found in Abschnitt A.2.

Obwohl Rym zur Beschreibung des Kontrollflusses in einer rein funktionalen Weise verwendet werden kann, unterstützt es auch imperative Kontrollflussausdrücke wie (if, else, match) und Schleifen (loop, while, for). Beispiele für die Verwendung dieser Ausdrücke können in Abschnitt A.2 nachgelesen werden.

Furthermore Rym provides a mechanism for early exits from loops through the use of break statements. The continue statement ends the current iteration of a loop and jumps starts the next one. Return statements are used for early returns out of functions.

Außerdem bietet Rym einen Mechanismus zum vorzeitigen Verlassen von Schleifen durch die Verwendung von break-Anweisungen. Die continue-Anweisung beendet die aktuelle Iteration einer Schleife und springt in die nächste. Return-Anweisungen werden für das vorzeitige Verlassen von Funktionen verwendet.

4.9 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.

Bibliographie

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

[10] "Interview on rust, a systems programming language developed by mozilla." https://www.infoq.com/news/2012/08/Interview-Rust (accessed: Jan. 14, 2023).

[11] "Rust. a language empowering everyone to build reliable and efficient software." https://www.rust-lang.org/ (accessed: Jan. 14, 2023).

[12] L. Bergstrom, "Google joins the rust foundation." https://opensource.googleblog.com/2021/02/google-joins-rust-foundation.html (accessed: Jan. 29, 2023).

[13] "Apache groovy. a multi-faceted language for the java platform." https://groovy-lang.org/index.html (accessed: Jan. 15, 2023).

[14] "Groovy language documentation. version 4.0.7." http://www.groovy-lang.org/single-page-documentation.html (accessed: Jan. 15, 2023).

[15] "The clojure programming language." https://clojure.org/index (accessed: Jan. 15, 2023).

[16] "Closure reference." https://clojure.org/reference (accessed: Jan. 15, 2023).

[17] "The scala programming language." https://scala-lang.org/ (accessed: Jan. 15, 2023).

[18] B. Venners, and F. Sommers, "The origins of scala. a conversation with martin odersky, part i." https://www.artima.com/articles/the-origins-of-scala (accessed: Jan. 15, 2023).

[19] "Scala language specification. version 2.13." https://scala-lang.org/files/archive/spec/2.13 (accessed: Jan. 15, 2023).

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

[20] "Jetbrains. essential tools for software developers and teams." https://www.jetbrains.com/ (accessed: Jan. 14, 2023).

[21] "Kotlin. a modern programming language that makes developers happier." https://kotlinlang.org/ (accessed: Jan. 14, 2023).

[22] "Kotlin language documentation 1.8.0." https://kotlinlang.org/docs/kotlin-reference.pdf (accessed: Jan. 14, 2023).

[23] "Develop android apps with kotlin." https://developer.android.com/kotlin (accessed: Jan. 15, 2023).

[24] "Go. build simple, secure, scalable systems with go." https://go.dev/ (accessed: Jan. 31, 2023).

[25] "The go programming language specification." https://go.dev/ref/spec (accessed: Jan. 10, 2023).

[26] "Flutter. multi platform." https://flutter.dev/multi-platform (accessed: Jan. 31, 2023).

[27] “Carbon language. an experimental successor to c++.” https://github.com/carbon-language/carbon-lang (accessed: Feb. 1, 2023).

[28] “Swift documentation.” https://developer.apple.com/documentation/swift (accessed: Jan. 10, 2023).

[29] https://www.swift.org/ (accessed: Feb. 1, 2023).

[3] M. L. Scott, *Programming Language Pragmatics. 4th Edition*, Burlington: Morgan Kaufmann, 2016.

[30] Z. Gao, C. Bird, and E. T. Barr, “To type or not to type: quantifying detectable bugs in javascript,” in *2017 IEEE/ACM 39th Int. Conf. Softw. Eng. (Icse)*, 2017, pp. 758–769, doi: 10.1109/ICSE.2017.75.

[31] “State of js 2022. what do you feel is currently missing from javascript?.” https://2022.stateofjs.com/en-US/opinions/#top_currently_missing_from_js (accessed: Feb. 4, 2023).

[32] “Ecmascript proposal. type annotations.” https://github.com/tc39/proposal-type-annotations (accessed: Feb. 4, 2023).

[33] “Typing. support for type hints.” https://docs.python.org/3/library/typing.html?highlight=typ#module-typing (accessed: Feb. 7, 2023).

[34] “Pep 484. type hints.” https://peps.python.org/pep-0484 (accessed: Feb. 7, 2023).

[35] “Php language reference. types. type declarations.” https://www.php.net/manual/en/language.types.declarations.php (accessed: Feb. 7, 2023).

[36] “The rust reference. types. https://doc.rust-lang.org/stable/reference/types.html (accessed: Jan. 10, 2023).

[37] “Handbook. type inference.” https://www.typescriptlang.org/docs/handbook/type-inference.html (accessed: Feb. 7, 2023).

[38] “The rust reference. let statements.” https://doc.rust-lang.org/stable/reference/statements.html#let-statements (accessed: Feb. 7, 2023).

[39] “The rust reference. influences.” https://doc.rust-lang.org/stable/reference/influences.html (accessed: Feb. 7, 2023).

[4] *ISO/IEC 24772-1. Avoiding Vulnerabilities in Programming Languages. Part 1: Language Independent Catalogue of Vulnerabilities*, 2019. [Online]. Available: https://open-std.org/jtc1/sc22/wg23/docs/ISO-IECJTC1-SC22-WG23_N1218-wd24772-1-international-standard-seed-document-for-DIS-ballot-20221023.pdf

[40] “Dart programming language specification. 6th edition draft.” https://spec.dart.dev/DartLangSpecDraft.pdf (accessed: Feb. 7, 2023).

[41] “C++ keyword. auto.” https://en.cppreference.com/w/cpp/keyword/auto (accessed: Feb. 7, 2023).

[42] “Java language specification. type inference.” https://docs.oracle.com/javase/specs/jls/se19/html/jls-18.html (accessed: Feb. 7, 2023).

[43] “The f# 4.1 language specification.” https://fsharp.org/specs/language-spec/4.1/FSharpSpec-4.1-latest.pdf (accessed: Feb. 7, 2023).

[44] “Haskell 2010 language report.” https://www.haskell.org/definition/haskell2010.pdf (accessed: Feb. 7, 2023).

[45] “Swift book. protocols.” https://docs.swift.org/swift-book/LanguageGuide/Protocols.html (accessed: Feb. 7, 2023).

[46] *ISO/IEC 14882:2020. Programming Languages -- C++*, 2020. [Online]. Available: https://www.open-std.org/JTC1/SC22/WG21/docs/papers/2022/n4910.pdf

[47] “The rust reference. traits.” https://doc.rust-lang.org/reference/items/traits.html (accessed: Feb. 7, 2023).

[48] “Java language specification. interfaces.” https://docs.oracle.com/javase/specs/jls/se19/html/jls-9.html (accessed: Feb. 7, 2023).

[49] “C# 7.0 draft specification. interfaces.” https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/interfaces (accessed: Feb. 7, 2023).

[5] S. Michell, “Ada and programming language vulnerabilities,” *Ada User J.*, vol. 30, no. 3, p. 180, 2009.

[50] “Null references. the billion dollar mistake.” https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare (accessed: Feb. 4, 2023).

[51] “Functional programming in go.” https://blog.logrocket.com/functional-programming-in-go (accessed: Feb. 8, 2023).

[52] “Swift book. closures.” https://docs.swift.org/swift-book/LanguageGuide/Closures.html (accessed: Feb. 8, 2023).

[53] “Functional programming is finally going mainstream.” https://github.com/readme/featured/functional-programming (accessed: Feb. 8, 2023).

[54] “.” https://www.ecma-international.org/wp-content/uploads/ECMA-262_13th_edition_june_2022.pdf (accessed: Feb. 8, 2023).

[55] “The rust reference. modules.” https://doc.rust-lang.org/reference/items/modules.html (accessed: Feb. 7, 2023).

[56] “Swift book. access control.” https://docs.swift.org/swift-book/LanguageGuide/AccessControl.html (accessed: Feb. 8, 2023).

[57] “Swift book. functions.” https://docs.swift.org/swift-book/LanguageGuide/Functions-.html (accessed: Feb. 8, 2023).

[58] “Php language reference. types. https://www.php.net/manual/en/language.types.php (accessed: Jan. 10, 2023).

[59] "C# 7.0 draft specification. types." https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/types (accessed: Jan. 10, 2023).

[6] "Stack overflow annual developer survey." https://insights.stackoverflow.com/survey (accessed: Jan. 15, 2023).

[60] "Java language specification. types, values, and variables." https://docs.oracle.com/javase/specs/jls/se19/html/jls-4.html (accessed: Jan. 10, 2023).

[61] "<cstdint> stdint.h." https://cplusplus.com/reference/cstdint (accessed: Feb. 8, 2023).

[62] "Ieee standard for floating-point arithmetic," *IEEE Std 754-2019 (Revision IEEE 754-2008)*, 2019, doi: 10.1109/IEEESTD.2019.8766229.

[63] "Ieee sa - iso/iec/ieee international standard - floating-point arithmetic." https://standards.ieee.org/ieee/60559/10226 (accessed: Jan. 2, 2023).

[64] "Factorial. computation." https://en.wikipedia.org/wiki/Factorial#Computation (accessed: Feb. 9, 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 (accessed: Jan. 14, 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 (accessed: Jan. 15, 2023).

[9] "Typescript. typescript is javascript with syntax for types." https://www.typescriptlang.org/ (accessed: Feb. 1, 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 Integers

```
0
44
1834
999_999_999
```

A.1.3 Floats

```
0.1
-1.4
```

Ist die Dezimalstelle `0`, dann kann sie weggelassen werden.

```
-444.0
-444.
```

A.1.4 Text

```
type Char
'a' '\n' '\t' '\u{2192}'
```

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

A.1.5 Records

```
type Vec3 = {
  x: F32,
  y: F32,
  z: F32,
}

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

A.1.6 Enumerationen / Vereinigungstypen

```
type EnumName = Variant1 | Variant2 | VariantN

let 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 Integers dargestellt:

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

let variant_n = EnumName.VariantN
let repr = variant_n as Uint       // 156
let 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 und können, wie jeder andere Ausdruck, als Initialisierer für eine Variable verwendet werden:

```
let outer_index = 0

let str = {
  let 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 IfLet..Else

```
let maybe_value = Some(2)
if let Some(value) = maybe_value {
  print(value)
} else {
  print("None")
}
```

A.2.4 `?` Operator

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

A.2.5 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.2.6 While

```
let mut number = 3
while number > 0 {
  print("{number}!")
  number -= 1
}
print("LIFTOFF!!!")
```

```
3
2
1
LIFTOFF!!!
```

A.2.7 Loop

```
let mut counter = 0
let result = loop {
  counter += 1
  if counter == 10 { break counter * 2 }
}
print(f"The result is {result}")
```

```
The result is 20
```

A.2.8 For

```
let numbers = [10, 20, 30, 40, 50]
for number in numbers {
  print(f"the value is: {number}")
}
```

```
the value is: 10
the value is: 20
the value is: 30
the value is: 40
the value is: 50
```

A.2.9 Closures

```
func twice(f: func(i32) -> i32) -> func(i32) -> i32 {
  x -> f(f(x))
}

let plus_three_twice = twice(i -> i + 3)
print(f"{plus_three_twice(10)}")
```

```
16
```

A.3 Statements

A.3.1 Variablen

```
let name = "Hello World!" // unveränderbare variable
let mut mut_name = "Hello " // änderbare variable
mut_name += "Universe!"
print(name, mut_name)
```

```
Hello World! Hello Universe!
```

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

```
let name
if condition { name = "Simon" } else { name = "Robert" }
print(name) // erlaubt, da `name` immer einen Wert hat
```

A.3.2 Use

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

let path = "./dad_jokes.txt"
let joke = "What should you do if you meet a giant? Use big words."
let create_result = File.create(path)
let write_result = fs.write(path, joke)
let data = fs.read_to_string(path).unwrap()

print(data)
```

```
What should you do if you meet a giant? Use big words.
```

A.3.3 Funktionen

A.3.3.1 Scope

Wenn die Funktion `changeBy(3)` aufgerufen wird, wird ein neuer Bereich erstellt, der den Parameter `x` enthält, welcher mit dem Wert `3` initialisiert wird. Die äußere Variable `x` ist innerhalb des Bereichs der Funktion nicht zugänglich und hat daher keinen Einfluss. Die Variable `y` wird ebenfalls in diesem Bereich definiert und mit dem Wert `10` initialisiert. Der Hauptteil der Funktion wird ausgeführt und der Wert von `x + y`, also `13` zurückgegeben. Anschließend wird der Funktionsbereich zerstört und der ursprüngliche Bereich wiederhergestellt, so dass der Wert `13` der Variablen `result` zugewiesen wird.

```
let x = 5
func changeBy(x: i32) -> i32 {
  let y = 10
  return x + y
}
let result = changeBy(3)  // result = 13
```

A.3.3.2 Standardwerte für Parameter

```
func increment(num: i32, by = 1) -> i32 {
  num + by
}

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

A.3.3.3 Erzwingen von benannten Argumenten

```
func testing(pos_or_named: i32, .., 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.3.3.4 Variable Argumente

```
func concat(..strings: [string], sep = "") -> string {
  strings.join(sep)
}
let name = "Mr. Walker"
print(
  concat("Hello ", name, "!"),
  concat(2.to_string(), True.to_string(), name, sep: ", "),
  concat(sep: ", ", 2.to_string(), True.to_string(), name),
  sep: "\n"
)
```

```
Hello Mr. Walker!
2, True, Mr. Walker
2, True, Mr. Walker
```

A.3.4 Implementationen

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:

```
type Bool = True | False

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

A.3.5 Traits / Typ-Klassen

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 I32 { 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 umzusetzen 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 }
  }
}

let mut counter = Counter { count: 0, max: 3 }
print(counter.next())
print(counter.next())
print(counter.next())
print(counter.next())
```

```
Some(1)
Some(2)
Some(3)
None
```

Anhang B — Rym Quellcode Beispiele

B.1 Factorial

Zwei mögliche Implementierungen für die Berechnung von Fakultäten.

Pseudocode von Wikipedia [64]:

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

B.1.1 Imperativer Ansatz

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

B.1.2 Deklarativer Ansatz

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

B.1.3 Nutzung

```
factorial(1)  // 1
factorial(2)  // 2
factorial(3)  // 6
factorial(4)  // 24
factorial(5)  // 120
```

B.2 Eingebaute Print Function

```
func print(..args: [impl Display], sep = " ", end = "\n") -> @Io {
  mut output = ""
  mut first_item = true

  for arg in args {
    if first_item { first_item = false } else { output.push(sep) }
    output.push(arg.fmt())
  }
  output.push(end)
  /* .. */
}
```

B.2.1 Deklarativer Ansatz

```
func print(..args: [impl Display], sep = " ", end = "\n") -> @Io {
  const output = args.iter().map(item -> item.fmt()).join(sep)
  const output = output + end
  /* .. */
}
```

B.2.2 Nutzung

```
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"
```

B.3 Find Summands

Funktion zum Finden von zwei Elementen in einer sortierten Liste, die die angegebene Summe ergeben.

```
func summands(numbers: [I32], sum: I32) -> Option<[Usize; 2]> {
  let mut low = 0
  let 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
}

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

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