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. 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. Da die Wartungsfreundlichkeit zu einem großen Teil durch die Lesbarkeit von Programmen bestimmt wird, ist sie eine wichtige Maßnahme zur Verbesserung der Qualität von Programmen und Programmiersprachen. In den folgenden Unterabschnitten werden Merkmale beschrieben, die zur Lesbarkeit einer Programmiersprache beitragen. [2, S. 42f]

2.1.1 Einfachheit

Die allgemeine Einfachheit einer Programmiersprache wirkt sich stark auf ihre Lesbarkeit aus. Eine Sprache mit 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]

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, 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 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]

Bibliographie

[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. [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

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

[6] "Factorial. computation." https://en.wikipedia.org/wiki/Factorial#Computation (accessed: Feb. 9, 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 wiederher-gestellt, 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 [6]:

```
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 {
  mut result = 1
  for const 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]> {
  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")
}
```