
Folgen durchlaufen

Schlagwörter	lineares Suchen, Rekursion, Iteration, strukturiertes Programmieren, Verifikation
Motivation	Eine oft vorkommende Programmieraufgabe ist das Suchen von Elementen, sei es in Behältern wie einfache Reihungen, allgemeinere Daten- oder Objektstrukturen, Datenbanken oder das Web, oder in Folgen dynamisch berechneter Elemente. Jeder Informatiker sollte elementare effiziente Suchalgorithmen beherrschen und in Programmiersprachen umsetzen können. Der Text beschränkt sich auf Reihungen und Folgen.
Lernziele	Der Leser lernt Standardalgorithmen zum Durchlaufen von und Suchen in Reihungen und Folgen in verschiedenen Varianten und Implementationen kennen und übt, diese Algorithmen zu konstruieren und ihre Korrektheit zu zeigen.

Inhalt

1	Elemente summieren	2
1.1	Funktionaler rekursiver Pseudocode	2
1.2	Strukturierter iterativer Pseudocode	3
1.3	Component Pascal	4
1.4	Eiffel	5
1.5	C++	6
1.6	Java	8
1.7	C#	9
1.8	Fazit	10
1.9	Aufgaben	10
2	Element mit einer Eigenschaft suchen	11
2.1	Strukturierter iterativer Pseudocode	13
2.2	Component Pascal	16
2.3	Eiffel	17
2.4	C++	18
2.5	Java	18
2.6	C#	19
2.7	Fazit	19
2.8	Aufgaben	19
3	Ausblick	23
3.1	Effizienz verbessern	23
3.2	Kontext verbessern	23
3.3	Behälter verallgemeinern	24
3.4	Aufgaben	25
A	Glossar	26
B	Literatur	27

1 Elemente summieren

Bevor wir das Suchproblem untersuchen und Algorithmen dazu konstruieren, lösen wir ein ähnliches Problem, um daraus Ideen für einen Suchalgorithmus zu gewinnen.

Summenproblem

Gegeben: Eine Reihung von numerischen Elementen, d.h. die Elemente gehören zu einer Grundmenge mit einer Addition, etwa einer Gruppe.

Gesucht: Die Summe der Elemente der Reihung.

Einfacherweise dient als Grundmenge die Gruppe der reellen Zahlen mit der Addition, $\langle \mathbb{R}, 0, -, + \rangle$. Mathematisch handelt es sich um die Abbildung

$$\text{Sum} : \mathbb{R}^n \rightarrow \mathbb{R},$$

$$(x_1, \dots, x_n) \mapsto \text{Sum}(x_1, \dots, x_n) := \sum_{i=1}^n x_i := x_1 + x_2 + \dots + x_{n-1} + x_n$$

mit $n \in \mathbb{N}_0$ für die Anzahl der Elemente $x_1, \dots, x_n \in \mathbb{R}$ und dem aus der Mathematik bekannten Summenzeichen Σ ; $(x_1, \dots, x_n)$ heißt Folge oder Vektor. Zur Definition der Summe ist „...“ intuitiv verständlich, aber vage. Genauer ist die **rekursive Definition**

$$\sum_{i=1}^n x_i := \begin{cases} 0 & \text{falls } n = 0 \\ \sum_{i=1}^{n-1} x_i + x_n & \text{sonst} \end{cases}.$$

Lösung

Auch um für solch eine Berechnung mit wiederholter Operation eine algorithmische Lösung zu erhalten, ist zuerst eine **Rekursionsformel** aufzustellen, die den i -ten Berechnungsschritt auf den $(i-1)$ -ten zurückführt. Sie lautet hier:

Rekursionsformel 1
für Summe

$$\begin{aligned} s_0 &= 0, \\ s_i &= s_{i-1} + x_i \quad \text{für } i = 1, \dots, n. \end{aligned}$$

Von s_n zu s_0 zurücklaufend gilt

$$\begin{aligned} s_n &= s_{n-1} + x_n \\ &= s_{n-2} + x_{n-1} + x_n \\ &\dots \\ &= s_1 + x_2 + \dots + x_{n-1} + x_n \\ &= s_0 + x_1 + x_2 + \dots + x_{n-1} + x_n \\ &= x_1 + x_2 + \dots + x_{n-1} + x_n \\ &= \sum_{i=1}^n x_i \\ &= \text{Sum}(x_1, \dots, x_n). \end{aligned}$$

Also ist mit der Rekursionsformel der wesentliche Teil der Arbeit geleistet.

1.1 Funktionaler rekursiver Pseudocode

Die Rekursionsformel 1 lässt sich als rekursive Funktion im funktionalen Programmierstil darstellen, der in *Informatik 3 Skript Teil 1*, 1.2.2 S. 1-9 beschrieben ist. Benötigt wird nur das Konstrukt des **bedingten Ausdrucks** (*conditional expression*)

Bedingter Ausdruck

if *Ausdruck1* **then** *Ausdruck2* **else** *Ausdruck3* **end**

mit der Semantik: Ist der boolesche *Ausdruck1* wahr, so wird *Ausdruck2*, sonst *Ausdruck3* ausgewertet und der erhaltene Wert als Wert des bedingten Ausdrucks eingesetzt. Dazu müssen (in typisiertem Kontext) *Ausdruck2* und *Ausdruck3* vom selben Typ sein.

Programm 1

Funktionaler
Pseudocode: Summe
einer Reihung, rekursiv

```

sum ( $x_1, \dots, x_n$  : REAL) : REAL is
  if  $n = 0$  then
    0
  else
    sum ( $x_1, \dots, x_{n-1}$ ) +  $x_n$ 
  end

```

1.2**Strukturierter iterativer Pseudocode**

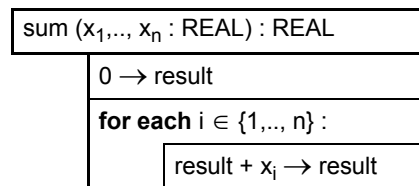
Doch interessieren hier iterative Lösungen. Was ist beim Umsetzen der Rekursionsformel 1 in einen strukturierten iterativen Algorithmus in Pseudocode, wie er sich etwa mit einem Struktogramm ohne Kenntnis einer speziellen Programmiersprachensyntax darstellen lässt, zu beachten?

Von der
Rekursionsformel zum
iterativen Algorithmus

- (1) Die Abbildung *Sum* in eine Funktion *sum* umformen.
- (2) Die Rekursionsformel in eine Zählschleife umformen, denn offenbar sind zur Berechnung der Summe alle Reihungselemente auszuwerten, also die Reihung vom ersten bis zum letzten Index zu traversieren.
- (3) Die indizierten Variablen $s_0, s_1, \dots, s_n$ werden zu einer lokalen Schleifenvariablen *result*, die auch das Funktionsergebnis enthält. Die Indizes verschwinden.
- (4) Die Indexvariable *i* wird zu einer Zählvariablen *i*, die von 1 bis *n* zählt.
- (5) Die erste Gleichung der Rekursionsformel ergibt die Initialisierung von *result*.
- (6) Die zweite Gleichung der Rekursionsformel ergibt den Schleifenrumpf, wobei aus Gleichungen Zuweisungen werden.

Programm 2

Struktogramm: Summe
einer Reihung, iterativ



→
result

Im Pseudocode ist „→“ das Zuweisungssymbol, wobei die Zuweisung in der in diesem Kulturkreis üblichen Leserichtung von links nach rechts zu lesen ist. Das Funktionsergebnis ist der letzte Wert der implizit vereinbarten Variablen *result*.

Was ist beim Umsetzen des Pseudocodes in eine imperative Implementationssprache zu beachten? Nur die konkreten Sprachkonstrukte sind zu berücksichtigen:

Vom Pseudocode zur
Implementations-
sprache

- (7) Wie stellen sich Reihungen dar? Gibt es offene Reihungen? Lassen sie sich per Referenz übergeben? Die angemessene Variante wählen!
- (8) Den Reihungsnamen *x* an die Sprachkonvention anpassen!
- (9) Wie ist der Indexbereich einer Reihung festgelegt? Wie sind die Indexgrenzen benannt? Den Indexbereich $\{1, \dots, n\}$ ggf. an den in der Sprache üblichen Indexbereich anpassen!
- (10) Wie wird auf Reihungselemente zugegriffen? Die Schreibweise anpassen!
- (11) Gibt es ein Zählschleifenkonstrukt? Oder nur Bedingungs-schleifen? Die angemessene Variante wählen!
- (12) Kann der Indexbereich leer sein? Falls nein, lässt sich eine Iteration einsparen.

Während die Schritte (1) bis (6) die Essenz der algorithmischen Lösung erfassen, ergänzen die Schritte (7) bis (12) unwesentliche sprachspezifische Details, syntaktischen Zucker und semantischen Pfeffer.

1.3**Component Pascal**

Zu 1.2(7)

Component Pascal bietet für Reihungen das Sprachkonstrukt `ARRAY-OF`. Es gibt statisch vereinbarte Reihungen, die im globalen Speicher oder im Laufzeitkeller liegen, und dynamisch auf der Halde erzeugte Reihungen, auf die mit Zeigern zugegriffen wird. Der Elementtyp ist statisch, die Reihungslänge statisch oder dynamisch festgelegt. Als Formalparameter können Reihungen offen sein, d.h. ihre Länge ist statisch offen gelassen und dynamisch mit der `LEN`-Funktion abfragbar. Möglich sind Wert- und Referenzübergabe, letztere als Ein-, Aus- und Ein-/Ausgabe. Für die Funktion `Sum` wählen wir als Parameter eine offene Reihung mit Eingabereferenzübergabe. Damit sind statische und dynamische Reihungen beliebiger Länge als Aktualparameter einsetzbar, die Funktion ist flexibel nutzbar. Die Referenzübergabe ist effizient. Wegen der Eingaberichtung ist die aktuelle Reihung schreibgeschützt.

Zu 1.2(8)

Eine Konvention gibt es nicht, wir behalten den Namen `x` bei. Der Indexbereich einer Reihung `x` ist stets

Zu 1.2(9)

`0 .. LEN (x) - 1`.

Für Lese- und Schreibzugriffe auf ein Reihungselement wird sein Index `i` in eckigen Klammern rechts vom Reihungsnamen notiert:

Zu 1.2(10)

`x [i]`.

Programm 3

Component Pascal:

Summe einer Reihung¹

```

TYPE
  Numeric = one of BYTE, SHORTINT, INTEGER, LONGINT, SHORTREAL, REAL;
  Index   = INTEGER;

PROCEDURE Sum (IN x : ARRAY OF Numeric) : Numeric;
  (*!
    Sum of all elements of x.
    Postcondition: result = Sum { x [i] : 0 <= i < LEN (x) }.
  !*)
  VAR
    result : Numeric;
    i      : Index;
BEGIN
  result := x [0];
  FOR i := 1 TO LEN (x) - 1 DO
    result := result + x [i];
  END;
  RETURN result;
END Sum;

```

Zu 1.2(11)

Die `FOR`-Zählschleife eignet sich gut zum Traversieren von Reihungen. Die Indexgrenzen lassen sich direkt lesbar und verständlich in das Sprachkonstrukt einsetzen:

`FOR i := 0 TO LEN (x) - 1 DO ...`

Zu 1.2(12)

Der Indexbereich einer Reihung `x` ist nie leer. Da das erste Reihungselement `x [0]` stets existiert, initialisieren wir `result` damit und starten die Zählschleife erst beim Index 1.

Alternativen

Gibt es von Programm 3 verschiedene Lösungen des Problems? Ja, aber...:

⊗ Die Zählrichtung umkehren

```

result := x [LEN (x) - 1];
FOR i := LEN (x) - 2 TO 0 BY -1 DO ...

```

ist umständlicher und schwerer verständlich.

¹ Programm 3 ist in den Informatik-Hug-Subsystemen in den Modulen `MathVectorsOf[Integer|LongInt|Real]` enthalten, die unter `[Hug112]` zu finden sind.

- ⊗ Die kopfgesteuerte (abweisende) Bedingungsschleife verwenden

```
result := x [0];
i := 1;
WHILE i < LEN (x) DO
    result := result + x [i];
    INC (i);
END; ...
```

ist auch umständlicher, weil der Index explizit zu inkrementieren ist.

- ⊗ Die fußgesteuerte (nicht abweisende) Bedingungsschleife verwenden

```
result := 0;
i := 0;
REPEAT
    result := result + x [i];
    INC (i);
UNTIL i >= LEN (x); ...
```

ist noch schwächer, weil es die gesparte Iteration wieder erfordert.

1.4

Eiffel

Zu 1.2(7)

Eiffel bietet für Reihungen eine generische Standardbibliotheksklasse namens *ARRAY*. Der Elementtyp einer Reihung wird statisch bei der Vereinbarung einer Reihungsreferenz festgelegt. Der Indexbereich eines Reihungsobjekts wird dynamisch bei seiner Erzeugung festgelegt, das Reihungsobjekt liegt auf der Halde. *Eiffel* kennt nur die Wertparameterübergabeart. Das einzig Mögliche ist, die Funktion *sum* mit einer Reihungsreferenz per Wertübergabe zu parametrisieren. Damit sind Referenzen auf dynamisch erzeugte Reihungsobjekte mit beliebigem Indexbereich als Aktualparameter einsetzbar. Die Übergabe einer Referenz ist effizient. Die Referenz ist schreibgeschützt, aber nicht das referenzierte Reihungsobjekt.

Zu 1.2(8)

Eine Konvention gibt es nicht, wir wählen den Namen *a* für *ARRAY*. Den minimalen und den maximalen Reihungsindex liefern die Abfragen

lower, upper : *INTEGER*

der *ARRAY*-Klasse. Ist also *a* eine Reihungsreferenz, die sich auf ein Reihungsobjekt bezieht (d.h. gilt *a* ≠ *Void*), so ist sein Indexbereich

Zu 1.2(9)

a.lower .. a.upper.

Programm 4

Eiffel: Summe einer
Reihung

```
class TOOLSET_ARRAY_NUMERIC [E -> NUMERIC]
feature
    sum (a : ARRAY [E]) : E
        -- Sum of all elements of a.
    require
        object_attached:    a /= Void
    local
        i : INTEGER
    do
        from
            i := a.lower
        until i > a.upper loop
            Result := Result + a.item (i)
            i := i + 1
        end
    ensure
        object_unchanged:    equal (a, old a)
    end -- sum
end -- class TOOLSET_ARRAY_NUMERIC
```

	Lesezugriff auf das Reihungselement mit dem Index i ermöglicht die mehrfach benannte Abfrage
Zu 1.2(10)	<i>item</i>, infix "@" (i : <i>INTEGER</i>) : <i>G</i> der <i>ARRAY</i>-Klasse. <i>G</i> bezeichnet den generischen Elementtyp. <i>Eiffel</i> bietet neben der üblichen objektorientierten Punktnotation <i>a.item</i> (i) auch die Infixnotation $a @ i$; neuere Implementationen von Eiffel Software bieten auch die spezielle Klammernotation $a [i]$.
Zu 1.2(11)	<i>Eiffel</i> kennt nur eine Schleifenart, nämlich eine kopfgesteuerte Bedingungsschleife mit Initialisierungsteil (from) und Abbruchbedingung (until). Aus dem Indexbereich ergibt sich die Initialisierung des Index i mit der Untergrenze <i>a.lower</i> , seine Inkrementierung am Schleifenrumpfende, und, da i bei der letzten Iteration die Obergrenze <i>a.upper</i> einnehmen muss, die Abbruchbedingung $i > a.upper$.
Zu 1.2(12)	Da der Indexbereich einer Reihung leer sein kann, lässt sich keine Iteration einsparen.
E -> NUMERIC	<i>sum</i> ist in einer Klasse mit eingeschränkter Generizität formuliert. Der formale generische Parameter <i>E</i> für den Elementtyp ist ein Nachfolger der Konzeptklasse <i>NUMERIC</i> der <i>Eiffel</i> -Standardbibliothek, die die geforderte Addition garantiert.
require	Die Vorbedingung $a \neq \text{Void}$ garantiert, dass sich <i>a</i> auf ein Reihungsobjekt bezieht.
Result	Die lokale Variable <i>Result</i> ist implizit vom Ergebnistyp der Funktion vereinbart und standardmäßig mit dem Nullwert dieses Typs initialisiert.
ensure	In der Nachbedingung <i>equal</i> (<i>a</i> , old <i>a</i>) bezeichnet old <i>a</i> das referenzierte Objekt im Zustand vor dem Aufruf von <i>sum</i> . Während $a = \text{old } a$ nur die Referenzen vergleicht, vergleicht <i>equal</i> die referenzierten Objekte. Die Nachbedingung garantiert, dass das Reihungsobjekt seinen Zustand behält, also <i>sum</i> keinen Nebeneffekt darauf bewirkt. In <i>Eiffel</i> programmiert man üblicherweise nebeneffektfrei, sodass man dazu neigt, diese als selbstverständlich betrachtete Nachbedingung nicht zu dokumentieren. Aber hier steht sie aus didaktischem Grund.
Alternativen	Gibt es von Programm 4 verschiedene Lösungsvarianten? Ja, aber nur die Zählrichtung umkehren: <pre> from $i := a.upper$ until $i < a.lower$ loop $Result := Result + a.item(i)$ $i := i - 1$ end </pre> Der Schreibaufwand ist genau gleich, die Verständlichkeit durch die unübliche Richtung vielleicht etwas reduziert.
1.5	C++
Zu 1.2(7)	<i>C++</i> hat von <i>C</i> ein Sprachkonstrukt für Reihungen geerbt, das jedoch eher eine Reihungsnotation für konstante Zeiger ist. <i>C</i>-Reihungen sind nicht speichersicher, weshalb Stroustrup abrät, sie noch zu benutzen, und stattdessen das <i>C++</i>-Konstrukt für Reihungen empfiehlt, nämlich die generische Klasse <i>vector</i> aus der <i>Standard Template Library</i> (<i>STL</i>). Deshalb handelt dieser Abschnitt von Vektoren statt Reihungen. Der Elementtyp eines <i>vector</i>-Objekts ist statisch festgelegt, sein Indexbereich dynamisch beim Aufruf eines Konstruktors. <i>C++</i> kennt nur die Wertparameterübergabeart. Wir wählen die Option, die Funktion <i>sum</i> mit einer Referenz auf ein konstantes <i>vector</i>-Objekt zu parametrisieren. Damit sind statische und dynamische Vektoren beliebiger Größe als Aktualparameter einsetzbar. Die Übergabe einer Referenz ist effizient. Die Referenz und das referenzierte <i>vector</i>-Objekt sind schreibgeschützt.

Zu 1.2(8)	Eine Konvention gibt es nicht, wir wählen den Namen <i>v</i> für <i>vector</i>. Die Abfrage <pre>int size()</pre> der <i>vector</i>-Klasse liefert die Elementanzahl. Der Indexbereich eines Vektors <i>v</i> ist stets
Zu 1.2(9)	<pre>0 .. v.size() - 1.</pre> Für Lese- und Schreibzugriffe auf das <i>i</i>-te Vektorelement dient die gewohnte Klammernotation, ermöglicht durch die C++-Spezialität des Überladens von Operatoren:
Zu 1.2(10)	<pre>v[i].</pre>
Zu 1.2(11)	C++ kennt drei Arten von Bedingungsschleifen, von denen die for-Schleife am ehesten dem Zweck einer Zählschleife dient. Sie fasst einen Initialisierungsteil, eine Fortsetzungsbedingung und eine Schleifenrumpfendausdrucksfolge syntaktisch zusammen. Aus dem Indexbereich ergibt sich die Initialisierung des Index <i>i</i> mit der Untergrenze 0, seine Inkrementierung als Schleifenrumpfendausdruck <i>++i</i>, und, da <i>i</i> bei der letzten Iteration die Obergrenze <i>v.size() - 1</i> einnehmen muss, die Fortsetzungsbedingung <i>i < v.size()</i>.
Zu 1.2(12)	Da der Indexbereich eines Vektors leer sein kann, lässt sich keine Iteration einsparen.
Programm 5 C++: Summe einer Reihung	<pre>template <typename T> T sum (const std::vector<T> & v) { T result = 0; for (int i = 0; i < v.size(); ++i) { result += v[i]; } return result; }</pre>
template T	<i>sum</i> erscheint als generische Funktion (<i>template function</i>). Zum formalen generischen Parameter <i>T</i> für den Elementtyp sind keine Eigenschaften spezifiziert, aber die Implementation von <i>sum</i> zeigt, dass <i>T</i> den Wert 0 und den Additionsoperator + kennen muss.
+=	Im Schleifenrumpf lässt sich die Addition mit anschließender Zuweisung <pre>result = result + v[i]</pre> abkürzen zu dem Ausdruck <pre>result += v[i].</pre>
++	In Programm 5 ist irrelevant, ob zum Inkrementieren des Index <i>i</i> der Präinkrementoperator (<i>++i</i>) oder der Postinkrementoperator (<i>i++</i>) benutzt wird, da es nur auf den Nebeneffekt auf <i>i</i> , nicht den Wert des Ausdrucks ankommt. Weniger dieser Nebeneffekt, mehr die spärliche Syntax der for -Schleife beeinträchtigt ihre Verständlichkeit.
Alternativen	Ähnlich wie in <i>Component Pascal</i> gibt es zu Programm 5 Lösungsvarianten: ⊗ Die Zählrichtung umkehren ist etwas schwerer verständlich: <pre>for (int i = v.size() - 1; i >= 0; --i) ...</pre> ⊕ Die kopfgesteuerte Bedingungsschleife verwenden <pre>int i = 0; while (i < v.size()) { result += v[i]; ++i; } ...</pre> bedeutet, dieselben semantischen Teile in anderer syntaktischer Reihenfolge hinzuschreiben. Die while-Schleife ist nicht umständlicher als die for-Schleife. Warum sollte man eine der beiden Schleifenarten vorziehen? Man kann etwa jede Schleifenart für einen bestimmten Zweck verwenden.

Leitlinie 1

C*: Wähle
angemessene
Schleifenart

Verwende bei

- Bedingungsschleifen die *while*-Schleife,
- Zählschleifen die *for*-Schleife!

Die fußgesteuerte Bedingungsschleife, die stets „einmal gratis iteriert“, eignet sich hier nicht, da Vektoren leer sein können und dann 0 Iterationen brauchen. Es gibt jedoch weitere Varianten. Typisch für die C*-Sprachen mit ihrem Konzept nebeneffektbehafteter Operatoren und Ausdrücke ist die Möglichkeit, Programme durch Ausnutzen von Nebenwirkungen zu verkürzen.

Programm 6

C++: Summe einer
Reihung,
nebenwirkungbehaftet
☹

```
template <typename T> T sum (const std::vector<T> & v) {
    T result = 0;
    for (int i = 0; i < v.size(); result += v[i++]);
    return result;
}
```

Programm 6 ist zwar drei Zeichen, zwei Zeilen kürzer als Programm 5, aber keineswegs verständlicher. Der Postinkrementoperator bei *i++* ist wesentlich. Ein Vielfaches der Zeit, die der Schreiber beim Tippen spart, investiert der Leser beim Studieren der Nebenwirkungen des Schleifenrumpfendausdrucks.

1.6

Zu 1.2(7)

Java

Java hat ein eingebautes Sprachkonstrukt für Reihungen, die gewisse Eigenschaften mit Klassen teilen, insbesondere die Referenzsemantik. Der Elementtyp einer Reihung ist statisch festgelegt, ihr Indexbereich dynamisch beim Aufruf eines Konstruktors. *Java* kennt nur die Wertparameterübergabeart. Das einzig Mögliche ist, die Funktion *sum* mit einer Reihungsreferenz per Wertübergabe zu parametrisieren. Damit sind dynamisch erzeugte Reihungen beliebiger Größe als Aktualparameter einsetzbar. Die Übergabe einer Referenz ist effizient. Die Referenz ist durch *final* schreibgeschützt, das referenzierte Reihungsobjekt nicht.

Zu 1.2(8)

Da es keine Konvention gibt, wählen wir den Namen *a* für *array*. Die Abfrage

```
int length
```

ist auf Reihungen anwendbar und liefert die Anzahl der Elemente. Damit ist

Zu 1.2(9)

```
0 .. a.length - 1
```

der Indexbereich der Reihung *a*. Auf das *i*-te Element von *a* wird mit gewohnter Klammernotation zugegriffen:

Zu 1.2(10)

```
a[i].
```

Zu 1.2(11)

Java ergänzt die von C/C++ übernommenen drei Arten von Bedingungsschleifen um eine Für-alle-Schleife zum Traversieren von Behältern, insbesondere Reihungen.

Zu 1.2(12)

Da der Indexbereich einer Reihung leer sein kann, lässt sich keine Iteration einsparen.

Programm 7

Java: Summe einer
Reihung

```
public static float sum (final float[] a) {
    assert a != null : "precondition";
    float result = 0;
    for (int i = 0; i < a.length; ++i) {
        result += a[i];
    }
    return result;
}
```

Lässt sich die Funktion `sum` als Methode einer Klasse mit eingeschränkter Generizität formulieren? Vermutlich nicht, da Zahlentypen in *Java* keine Klassen sind. Deshalb wählen wir einfacherweise als Elementtyp den Gleitpunktzahlentyp `float`.

`assert`

Die Vorbedingung `a != null` garantiert, dass sich `a` auf ein Reihungsobjekt bezieht. Dass `sum` keinen Nebeneffekt auf das Reihungsobjekt bewirkt, lässt sich schwer als Nachbedingung spezifizieren.

Alternativen

Schwächere Lösungsvarianten mit C++-Schleifen sind auf S. 7 vorgestellt. Eine neue, interessante Variante liefert die Für-alle-Schleife. Sie bietet eine echte Abstraktion. Der Programmierer muss keine Zählrichtung wählen, er braucht sich nicht einmal um einen Index zu kümmern.

Programm 8

Java: Summe einer

Reihung,

Für-alle-Schleife



```
public static float sum (final float[] a) {
    assert a != null : "precondition";
    float result = 0;
    for (float x : a) {
        result += x;
    }
    return result;
}
```

1.7

Zu 1.2(7) bis 1.2(12)

C#

C# unterscheidet sich oft nur unwesentlich von *Java*, so auch bei Reihungen und Schleifen. Weder die Referenz noch das referenzierte Reihungsobjekt ist schreibgeschützt (`final` gibt es nicht). Von der Form der Zusicherung abgesehen differiert das Folgende in der Konvention für Klein-Großschreibung. Für die Abfrage der Elementanzahl und den Indexbereich einer Reihung `a` gilt:

```
int Length
0 .. a.Length - 1.
```

Programm 9

C#: Summe einer

Reihung

```
public static float Sum (float[] a) {
    System.Diagnostics.Debug.Assert(a != null, "precondition");
    float result = 0;
    for (int i = 0; i < a.Length; ++i) {
        result += a[i];
    }
    return result;
}
```

Lässt sich die Funktion `sum` als Methode einer Klasse mit eingeschränkter Generizität formulieren? So lange wir das nicht wissen, wählen wir einfacherweise als Elementtyp den Gleitpunktzahlentyp `float`.

Alternativen

Schwächere Lösungsvarianten mit C++-Schleifen sind von S. 7 bekannt. Die interessante Variante mit der Für-alle-Schleife ist durch zwei syntaktische Verbesserungen verständlicher als ihr *Java*-Pendant Programm 8.

Programm 10

C#: Summe einer

Reihung,

Für-alle-Schleife



```
public static float sum (float[] a) {
    System.Diagnostics.Debug.Assert(a != null, "precondition");
    float result = 0;
    foreach (float x in a) {
        result += x;
    }
    return result;
}
```

1.8

Fazit

- Zum Summieren der Elemente einer Reihung mit n numerischen Elementen sind n Additionen auszuführen, und zwar in einem sequenziellen Algorithmus eine nach der anderen, praktischerweise vom ersten zum letzten Element.
- Diese Berechnung lässt sich mit einer Rekursionsformel beschreiben.
- Aus der Rekursionsformel ergeben sich durch einfache Umformungsregeln
 - eine rekursive Funktion,
 - ein iterativer Algorithmus.
- Da alle Reihungselemente auszuwerten sind, eignen sich Zählschleifen mit impliziten Anteilen am besten.
 - Bei der FOR-Zählschleife von *Component Pascal* sind eine Zählvariable, der Indexbereich und die Zählrichtung zu spezifizieren; das Schleifenkonstrukt erledigt das Zählen.
 - Bei den Für-alle-Schleifen von *Java* (`for`) und *C#* (`foreach`) sind nur eine Elementvariable und die Reihung zu spezifizieren; das Schleifenkonstrukt erledigt das Traversieren der Reihung.
- Bei Sprachen, die kein Zählschleifenkonstrukt bieten, ist der Algorithmus mit einer kopfgesteuerten Bedingungsschleife mit expliziter Zählung zu formulieren.
 - In *Eiffel* gibt es abgesehen von der Zählrichtung nur eine Lösung mit der **from-until**-Schleife.
 - In *C++* ist die **for**-Schleife für Zählzwecke einzusetzen. *C++* erlaubt viele, oft schwächere Lösungsvarianten.

1.9

Aufgaben

Aufgabe 1

Norm eines
Zahlenvektors

Gegeben ist ein Zahlenvektor $(x_1, \dots, x_n) \in \mathbb{R}^n$, gesucht seine durch

$$\text{norm}(x_1, \dots, x_n) := \sqrt{\sum_{i=1}^n x_i^2}$$

definierte **Norm**. Entwickeln Sie dazu

- (1) eine Rekursionsformel,
- (2) eine iterative Lösung in Pseudocode,
- (3) iterative Lösungen in *Eiffel*, *C++*, *Java*, *C#*!

Eine Lösung in *Component Pascal* mit *BlackBox* finden Sie in den Informatik-Hug-Subsystemen im Modul *MathVectorsOfReal* [HugI12]. Dieses Modul enthält auch statistische Funktionen wie Variationsbreite, arithmetischer, geometrischer und harmonischer Mittelwert, Varianz und durchschnittliche Abweichung, sowie Prozeduren zum Initialisieren, Kopieren, Addieren, Subtrahieren, Invertieren und skalar Multiplizieren von Vektoren, deren Implementationen auf demselben Algorithmus beruhen.

Aufgabe 2

Minimum einer Folge

Zu Folgen $(x_1, \dots, x_n) \in G^n$ in einer vollgeordneten Grundmenge $\langle G, \leq \rangle$ ist das durch

$$x = \min(x_1, \dots, x_n) \quad :\Leftrightarrow \quad x \in \{x_1, \dots, x_n\} \wedge \forall i = 1, \dots, n : (x \leq x_i)$$

definierte **Minimum** ihrer Elemente gesucht. Entwickeln Sie dazu

- (1) eine Rekursionsformel,
- (2) eine iterative Lösung in Pseudocode,
- (3) iterative Lösungen in *Eiffel*, *C++*, *Java*, *C#*!

Eine Rekursionsformel und eine Konstruktion eines iterativen Algorithmus finden Sie im Skript *Theoretische Grundlagen*, Abschnitt 4.5.6, S. 4-54ff [HugTG]. Das Modul *MathVectorsOfReal* enthält eine *Component-Pascal*-Lösung und weitere, mit demselben Algorithmus implementierte Funktionen [HugI12].

2 Element mit einer Eigenschaft suchen

Zuerst formulieren wir das Suchproblem verbal, ohne jeglichen Formalismus.

Vorkommensproblem
verbal

Gegeben: Eine Reihung oder Folge von Elementen und eine auf solche Elemente anwendbare Eigenschaft.

Gesucht: (a) Die Aussage, ob alle Elemente in der Reihung oder Folge die Eigenschaft besitzen.
(b) Die Aussage, ob wenigstens ein Element in der Reihung oder Folge die Eigenschaft besitzt.

Wege zu algorithmischen Lösungen eines Problems führen über seine Formalisierung. Deshalb formalisieren wir das Suchproblem.

Vorkommensproblem
formal

Gegeben: Reihungen

$$x = (x_1, \dots, x_n) \in G^n$$

von $n \in \mathbb{N}_0$ Elementen $x_1, \dots, x_n \in G$ einer beliebigen Grundmenge G und eine auf dieser Grundmenge definierte Eigenschaft, also ein einstelliges Prädikat P , das als Teilmenge $P \subseteq G$ oder deren Indikatorfunktion

$$P : G \rightarrow \mathbb{B}$$

in die Menge $\mathbb{B} = \{\text{false}, \text{true}\}$ der logischen Konstanten dargestellt wird.

Allform

Gesucht: (a) Das n -stellige Prädikat auf G , ob alle Elemente in $x \in G^n$ die Eigenschaft P besitzen:

$$\begin{aligned} All_P : G^n &\rightarrow \mathbb{B}, \\ (x_1, \dots, x_n) \mapsto All_P(x_1, \dots, x_n) &:= \bigwedge_{i \in \{1, \dots, n\}} P(x_i) \\ &= P(x_1) \wedge P(x_2) \wedge \dots \wedge P(x_{n-1}) \wedge P(x_n) \\ &= \forall i = 1, \dots, n : P(x_i). \end{aligned}$$

Existenzform

(b) Das n -stellige Prädikat auf G , ob wenigstens ein Element in $x \in G^n$ die Eigenschaft P besitzt:

$$\begin{aligned} Exists_P : G^n &\rightarrow \mathbb{B}, \\ (x_1, \dots, x_n) \mapsto Exists_P(x_1, \dots, x_n) &:= \bigvee_{i \in \{1, \dots, n\}} P(x_i) \\ &= P(x_1) \vee P(x_2) \vee \dots \vee P(x_{n-1}) \vee P(x_n) \\ &= \exists i = 1, \dots, n : P(x_i). \end{aligned}$$

Die Analogie zum auf S. 2 formulierten Summenproblem ist offensichtlich: Auf jedes Reihungselement x_i wird zunächst das Prädikat P angewandt, die so erhaltenen Werte werden dann (statt addiert) konjunktiv oder disjunktiv verknüpft. Bevor wir wie beim Summenproblem eine Rekursionsformel aufstellen, klären wir, wie die Prädikate All_P und $Exists_P$ zusammenhängen. Bekanntlich gelten auch in der Prädikatenlogik de Morgansche Regeln, siehe etwa Satz 2.56, S. 2-59 in [HugTG].

Prädikatenlogische
Regeln von de Morgan,
allgemein

Für beliebige Variablen x und prädikatenlogische Formeln φ gilt:

- | | | | |
|-----|---------------------------------|----------|-----------------------------|
| (1) | $\neg \forall x : \varphi$ | $\equiv$ | $\exists x : \neg \varphi.$ |
| (2) | $\neg \forall x : \neg \varphi$ | $\equiv$ | $\exists x : \varphi.$ |
| (3) | $\neg \exists x : \varphi$ | $\equiv$ | $\forall x : \neg \varphi.$ |
| (4) | $\neg \exists x : \neg \varphi$ | $\equiv$ | $\forall x : \varphi.$ |

Aufgrund dieser Negierungssätze lässt sich der Allquantor mittels Negation und Existenzquantor, der Existenzquantor mittels Negation und Allquantor ausdrücken. Als Formel φ das obige Prädikat P eingesetzt liefert diese Äquivalenzen:

Prädikatenlogische
Regeln von de Morgan,
speziell

- | | | | |
|-----|--|----------|--|
| (5) | $\neg \forall i = 1, \dots, n : P(x_i)$ | $\equiv$ | $\exists i = 1, \dots, n : \neg P(x_i).$ |
| (6) | $\neg \forall i = 1, \dots, n : \neg P(x_i)$ | $\equiv$ | $\exists i = 1, \dots, n : P(x_i).$ |
| (7) | $\neg \exists i = 1, \dots, n : P(x_i)$ | $\equiv$ | $\forall i = 1, \dots, n : \neg P(x_i).$ |
| (8) | $\neg \exists i = 1, \dots, n : \neg P(x_i)$ | $\equiv$ | $\forall i = 1, \dots, n : P(x_i).$ |

Die Prädikatnamen All_P und $Exists_P$ eingesetzt liefert dies:

Prädikatenlogische
Regeln von de Morgan,
mit Namen

- | | | | |
|------|---|----------|-------------------------------------|
| (9) | $\neg All_P(x_1, \dots, x_n)$ | $\equiv$ | $Exists_{\neg P}(x_1, \dots, x_n).$ |
| (10) | $\neg All_{\neg P}(x_1, \dots, x_n)$ | $\equiv$ | $Exists_P(x_1, \dots, x_n).$ |
| (11) | $\neg Exists_P(x_1, \dots, x_n)$ | $\equiv$ | $All_{\neg P}(x_1, \dots, x_n).$ |
| (12) | $\neg Exists_{\neg P}(x_1, \dots, x_n)$ | $\equiv$ | $All_P(x_1, \dots, x_n).$ |

Also kurz dies:

Prädikatenlogische
Regeln von de Morgan,
kompakt

- | | | | |
|------|------------------------|----------|-------------------|
| (13) | $\neg All_P$ | $\equiv$ | $Exists_{\neg P}$ |
| (14) | $\neg All_{\neg P}$ | $\equiv$ | $Exists_P$ |
| (15) | $\neg Exists_P$ | $\equiv$ | $All_{\neg P}$ |
| (16) | $\neg Exists_{\neg P}$ | $\equiv$ | All_P |

Wozu?

Warum diese Regeln so breittreten? Weil sie elementar wichtig sind! Was bedeuten sie für die Konstruktion einer algorithmischen Lösung des Suchproblems? Dass es genügt, eines der beiden Teilprobleme (a), (b) zu lösen. (14) und (16) zeigen, dass sich aus der Lösung des einen Problems die Lösung des anderen durch zweimaliges Negieren ergibt. Im Folgenden beschränken wir uns deshalb auf das Teilproblem (b), die Existenzform: Kommt in der Reihung ein Element mit der gegebenen Eigenschaft vor? (Teilproblem (a), die Allform, bleibt als I3-Prüfungsstoff erhalten.)

Lösung zu (b)

Die wiederholte Operation ist hier die Disjunktion, die Oder-Verknüpfung. Mit dem Variablennamen r für *result* erhält man analog zum Summenproblem auf S. 2 die Rekursionsformel:

Rekursionsformel 2
für Disjunktion

$$r_0 = false,$$

$$r_i = r_{i-1} \vee P(x_i) \quad \text{für } i = 1, \dots, n.$$

Damit gilt

$$r_n = r_{n-1} \vee P(x_n)$$

$$= r_{n-2} \vee P(x_{n-1}) \vee P(x_n)$$

$$\dots$$

$$= r_1 \vee P(x_2) \vee \dots \vee P(x_{n-1}) \vee P(x_n)$$

$$\begin{aligned}
&= r_0 \vee P(x_1) \vee P(x_2) \vee \dots \vee P(x_{n-1}) \vee P(x_n) \\
&= P(x_1) \vee P(x_2) \vee \dots \vee P(x_{n-1}) \vee P(x_n) \\
&= \bigvee_{i \in \{1, \dots, n\}} P(x_i) \\
&= \text{Exists } P(x_1, \dots, x_n).
\end{aligned}$$

Also leistet die Rekursionsformel wieder das Wesentliche.

Anforderungen an
Lösungen

An Implementationen, also Umsetzungen der Rekursionsformel in Programmiersprachen, stellen wir Qualitätsanforderungen: Algorithmische Lösungen des Problems müssen korrekt und verständlich sein. Darüber hinaus sollen sie möglichst effizient sein, ohne an Klarheit zu verlieren. Soll eine Lösung korrekt sein, so sollte ihre Korrektheit leicht nachzuweisen sein, denn was nützt Korrektheit, die keiner sieht? Da zur Verständlichkeit einer Lösung ihre Strukturiertheit beiträgt, fordern wir Algorithmen, die der Ein-Eingang-Ein-Ausgang-Regel des strukturierten Programmierens folgen.

2.1

Strukturierter iterativer Pseudocode

Entwickelt man den Suchalgorithmus analog zum Summenalgorithmus mit den Regeln 1.2(1) bis 1.2(6), so erhält man Programm 11.

Programm 11
Struktogramm: Suchen
in einer Reihung,
Zählschleife
☹

existsP ($x_1, \dots, x_n : G$; P : function ($x : G$) : BOOLEAN) : BOOLEAN	
	false $\rightarrow$ result
	for each $i \in \{1, \dots, n\}$:
	result or P(x_i) $\rightarrow$ result

Think twice
before you code!

Doch halt! Dieser Algorithmus rechnet unnötig viel: Er prüft jedes Element vom ersten bis zum letzten. Sein Laufzeitaufwand ist proportional zur Anzahl n der Elemente. Da nur interessiert, ob es ein Element mit der Eigenschaft P gibt, kann die Suche nach Elementen mit der Eigenschaft P enden, sobald das erste davon gefunden ist. Der so verbesserte Algorithmus hat einen mittleren Laufzeitaufwand proportional zu $n / 2$; er gehört zur selben Komplexitätsklasse wie die Zählvariante, ist aber im Mittel immerhin doppelt so schnell wie jene.

Bedingungsschleife

Die Bedingung „Element mit der Eigenschaft P gefunden“ ist eine Abbruchbedingung für die Schleife. Also ist für das Suchen keine Zählschleife, sondern eine Bedingungsschleife angebracht.

Lange und kurze
Auswertung

Eine andere Sicht ist: Bei Implementationen der logischen Operationen Konjunktion, Disjunktion und Implikation in Programmiersprachen ist zwischen langer und kurzer Auswertung zu unterscheiden.

- Bei **langer Auswertung** werden alle Operanden ausgewertet und ihre Werte verknüpft. Sobald ein Operand undefiniert ist, ist der ganze Ausdruck undefiniert.
- Bei **kurzer Auswertung** werden Operanden von links nach rechts nur so lange ausgewertet, bis das Ergebnis nach zweiwertiger Logik feststeht. Der Ausdruck kann definiert sein, ohne dass jeder Operand definiert sein muss.

Bei der Disjunktion gilt in zweiwertiger Logik: Sobald ein Operand true ist, ist das Ergebnis true, unabhängig davon, ob folgende Operanden true oder false sind. Die Berechnung kann also enden, wenn zum ersten Mal ein Operand true liefert.

Ansatz 1

Um effiziente Lösungen des Suchproblems zu konstruieren, betrachten wir drei methodische Ansätze. Beim ersten Ansatz gehen wir von Programm 11 aus, transformieren die Zählschleife in eine kopfgesteuerte Bedingungsschleife mit Abbruchbedingung:

i : NATURAL is init 1
until i > n :
result or P(x _i) → result
i + 1 → i

und fügen die zweite Abbruchbedingung hinzu:

i : NATURAL is init 1
until i > n or result :
assert i <= n and not result
result or P(x _i) → result
i + 1 → i

Die im Schleifenrumpf als Zusicherung ergänzte Fortsetzungsbedingung zeigt, dass **not result** gilt, also result = false. Damit ist result **or** P(x_i) durch false **or** P(x_i), also P(x_i) ersetzbar. Damit erhalten wir Programm 12.

Programm 12

Struktogramm: Lineare
Suche in einer
Reihung,
Bedingungsschleife mit
Ergebnisvariable

existsP (x₁, ..., x_n : G; P : function (x : G) : BOOLEAN) : BOOLEAN
false → result
i : NATURAL is init 1
until i > n or result :
P(x _i) → result
i + 1 → i

Da die Reihung sequenziell vom ersten bis zum gesuchten oder zum letzten Element durchlaufen wird, heißt der Algorithmus **lineare Suche**. Der **Suchraum**, zu Beginn alle Elemente der Reihung, wird bei jeder Iteration (nur) um ein Element kleiner. Die Schleife terminiert spätestens, wenn der Suchraum erschöpft ist.

Ansatz 2

Programm 12 ergibt sich auch mit dem zweiten Konstruktionsansatz:

- Die Traversierung als Rohbau hinschreiben (**for each** i ∈ {1, ..., n} : *loopbody* mit Bedingungsschleife).
- Eine lokale Variable result vorsehen, deren letzter Wert das Funktionsergebnis ist.
- result geeignet initialisieren (hier: false, d.h. noch nicht gefunden).
- result in der Schleifenbedingung berücksichtigen (hier: fertig wenn gefunden, sonst weiter suchen).
- result im Schleifenrumpf geeignete Werte zuweisen (hier: Hat aktuelles Element gesuchte Eigenschaft?).

So wird die Schleife eher von oben nach unten konstruiert. Nützlich an Ansatz 2 ist, dass das Ergebnis in der Variablen result klar ersichtlich ist. Eine Analyse von Programm 12 ergibt, dass der Wert von P(x_i) in result geschrieben und danach in der Schleifenbedingung wieder gelesen wird. Einen nur einmal benötigten Wert zu speichern ist unnötig. Also ist result in der Schleife von Programm 12 eine verzichtbare Steuervariable. Dies gilt für viele nach Ansatz 2 entwickelte Algorithmen. Nachteilig an ihnen ist also, dass sie u.U. eine überflüssige Variable enthalten, also nicht möglichst effizient sind.

or else
and then

Eliminieren wir das Zwischenspeichern, so erhalten wir Programm 13. Zu beachten ist, dass hier in den Schleifenbedingungen kurze Auswertung – hier ausgedrückt durch **or else** und **and then** – wesentlich ist: Zuerst ist zu prüfen, ob der Index i gültig ist, bevor er zum Zugriff auf ein Reihungselement x_i benutzt wird.

Die Nachbedingungen der Schleife in Programm 13 verdeutlichen, was die Schleife leistet: Sie findet ein Element mit der Eigenschaft P oder nicht. Abzulesen ist das am Zustand des Index nach dem Schleifenende: Gilt $i > n$, so wurde der ganze Indexbereich durchlaufen, also der Suchraum erschöpft, ohne ein Element mit P zu finden. Gilt $i \leq n$, so hat das Element x_i als einziges unter den Elementen $x_1, \dots, x_i$ die Eigenschaft P. Der Rückgabewert ist entsprechend zu differenzieren. Die Variable result wird erst hier benötigt. Zu zeigen bleibt, dass die Nachbedingungen korrekt sind.

Programm 13

Struktogramm: Lineare
Suche in einer
Reihung,
Bedingungsschleife mit
Zusicherungen



existsP ($x_1, \dots, x_n : G$; function ($x : G$) : BOOLEAN) : BOOLEAN	
i : NATURAL is init 1	
loopinvariant $1 \leq i \leq n + 1$ and then for all $k = 1 \dots i - 1$: not $P(x_k)$	
until $i > n$ or else $P(x_i)$:	-- termination condition
assert $i \leq n$ and then not $P(x_i)$	-- continuation condition
$i + 1 \rightarrow i$	
assert $i > n$ implies for all $k = 1 \dots n$: not $P(x_k)$	-- not found, loop postcondition
assert $i \leq n$ implies ($P(x_i)$ and for all $k = 1 \dots i - 1$: not $P(x_k)$)	-- found, loop postcondition
$i \leq n \rightarrow \text{result}$	

Nachweis der
Korrektheit des
Algorithmus

Behauptung. Nach der Schleife gilt:

$(i > n \text{ implies for all } k = 1 \dots n : \text{not } P(x_k)) \text{ and } (i \leq n \text{ implies } (P(x_i) \text{ and for all } k = 1 \dots i - 1 : \text{not } P(x_k)))$.

Beweis. Die Schleifeninvariante

$1 \leq i \leq n + 1 \text{ and then for all } k = 1 \dots i - 1 : \text{not } P(x_k)$

muss vor und nach jedem Schleifendurchlauf gelten. Der Teil

$1 \leq i \leq n + 1$

besagt, dass i entweder im Indexbereich $1 \dots n$ liegt oder ungültig ist, und zwar mit $n + 1$ als einzig möglichem Wert. Dies folgt daraus, dass i mit 1 initialisiert und dann solange $i \leq n$ nur um 1 erhöht wird. Der Teil

for all $k = 1 \dots i - 1 : \text{not } P(x_k)$

besagt, dass alle vor dem aktuellen Index i liegenden Elemente x_k nicht die Eigenschaft P haben, also keines davon das gesuchte ist. Für $i = 1$ vor dem ersten Schleifendurchlauf gilt das trivialerweise, für $i > 1$ folgt es aus der wiederholten Fortsetzungsbedingung. Also ist die Schleifeninvariante korrekt. Nach der Schleife gelten die Abbruchbedingung und die Schleifeninvariante, also

$(i > n \text{ or else } P(x_i)) \text{ and } (1 \leq i \leq n + 1 \text{ and then for all } k = 1 \dots i - 1 : \text{not } P(x_k))$.

Wir schwächen diese Bedingung durch Weglassen von $1 \leq i \leq n + 1$ ab, eliminieren das **or else** und erhalten

$(i > n \text{ or } (i \leq n \text{ and } P(x_i))) \text{ and for all } k = 1 \dots i - 1 : \text{not } P(x_k)$.

Mit den Abkürzungen

$a \equiv i > n$,

$b \equiv P(x_i)$,

$c \equiv \text{for all } k = 1 \dots i - 1 : \text{not } P(x_k)$

gilt:

$(i > n \text{ or } (i \leq n \text{ and } P(x_i))) \text{ and for all } k = 1 \dots i - 1 : \text{not } P(x_k) \equiv (a \text{ or } (\text{not } a \text{ and } b)) \text{ and } c \equiv$

$(a \text{ and } c) \text{ or } (\text{not } a \text{ and } b \text{ and } c) \equiv$
 $(a \text{ or } \text{not } a) \text{ and } (a \text{ or } (b \text{ and } c)) \text{ and } (c \text{ or } \text{not } a) \text{ and } (c \text{ or } (b \text{ and } c)) \equiv$
 $(a \text{ or } (b \text{ and } c)) \text{ and } (\text{not } a \text{ or } c) \text{ and } c \equiv$
 $(\text{not } a \text{ implies } (b \text{ and } c)) \text{ and } (a \text{ implies } c) \text{ and } c.$

Wir schwächen die Bedingung durch Weglassen von c am Ende weiter ab, ersetzen die Abkürzungen und erhalten

$(\text{not } i > n \text{ implies } (P(x_i) \text{ and for all } k = 1 \dots i - 1 : \text{not } P(x_k))) \text{ and}$
 $(i > n \text{ implies for all } k = 1 \dots i - 1 : \text{not } P(x_k)).$

Berücksichtigen wir, dass $\text{not } i > n \equiv i \leq n$ und dass aus $i > n$ folgt $i = n + 1$, so folgt die behauptete Nachbedingung. $\square$

Ansatz 3

Der dritte methodische Ansatz zur Konstruktion des linearen Suchalgorithmus lernt aus den vorigen Ansätzen: Am besten ist, mit den Nachbedingungen der Schleife zu beginnen, diese also eher von unten nach oben zu entwickeln:

- Die Nachbedingung der Schleife formulieren und
- daraus die Abbruchbedingung extrahieren:
Suchraum erschöpft, d.h. Indexbereich ganz durchlaufen, d.h. Index i ungültig (und gesuchtes Element nicht gefunden)
oder sonst
gesuchtes Element gefunden, und zwar beim Index i , d.h. es gilt $P(x_i)$.
- Die Bedingung *Suchraum nicht erschöpft* liefert das Ergebnis.
- Im Schleifenrumpf nur den Index weiterzählen.
- Vor der Schleife den Index initialisieren.

Gute Programmierer (und I3-Prüflinge) schreiben den strukturierten linearen Suchalgorithmus ohne Umweg und Beweis in der korrekten Kurzform (ohne Sprunganweisungen, ohne Steuervariablen) nieder:

Programm 14

Struktogramm: Lineare
Suche in einer
Reihung,
Bedingungsschleife,
Existenzform



existsP ($x_1, \dots, x_n : G; P : \text{function } (x : G) : \text{BOOLEAN}$) : BOOLEAN					
<table> <tr> <td>i : NATURAL is init 1</td></tr> <tr> <td>until i > n or else P(x_i) :</td></tr> <tr> <td> <table> <tr> <td>i + 1 $\rightarrow$ i</td></tr> </table> </td></tr> <tr> <td>i $\leq$ n $\rightarrow$ result</td></tr> </table>	i : NATURAL is init 1	until i > n or else P(x_i) :	<table> <tr> <td>i + 1 $\rightarrow$ i</td></tr> </table>	i + 1 $\rightarrow$ i	i $\leq$ n $\rightarrow$ result
i : NATURAL is init 1					
until i > n or else P(x_i) :					
<table> <tr> <td>i + 1 $\rightarrow$ i</td></tr> </table>	i + 1 $\rightarrow$ i				
i + 1 $\rightarrow$ i					
i $\leq$ n $\rightarrow$ result					

Um Programmierpraxis (und I3-Prüfungsvorbereitung) zu erleichtern folgt auf die Existenzform die Allform:

Programm 15

Struktogramm: Lineare
Suche in einer
Reihung,
Bedingungsschleife,
Allform



allP ($x_1, \dots, x_n : G; P : \text{function } (x : G) : \text{BOOLEAN}$) : BOOLEAN					
<table> <tr> <td>i : NATURAL is init 1</td></tr> <tr> <td>until i > n or else not P(x_i) :</td></tr> <tr> <td> <table> <tr> <td>i + 1 $\rightarrow$ i</td></tr> </table> </td></tr> <tr> <td>i > n $\rightarrow$ result</td></tr> </table>	i : NATURAL is init 1	until i > n or else not P(x_i) :	<table> <tr> <td>i + 1 $\rightarrow$ i</td></tr> </table>	i + 1 $\rightarrow$ i	i > n $\rightarrow$ result
i : NATURAL is init 1					
until i > n or else not P(x_i) :					
<table> <tr> <td>i + 1 $\rightarrow$ i</td></tr> </table>	i + 1 $\rightarrow$ i				
i + 1 $\rightarrow$ i					
i > n $\rightarrow$ result					

2.2

Component Pascal

Aus der Grundmenge G wird ein beliebiger Typ *SomeType*. Für die Eigenschaft lässt sich in *Component Pascal* ein Funktionstyp *Property* definieren. Die aktuelle Eigenschaft P ist eine Funktion dieses Typs, die als Parameter der Suchfunktion *ExistsP* übergeben wird. Die kopfgesteuerte Bedingungsschleife ist die *WHILE*-Schleife, sie hat eine Fortsetzungsbedingung. Die Variable *result* ist wegen der **RETURN**-Anweisung unnötig.

Programm 16

Component Pascal:
Lineare Suche in einer
Reihung

```

TYPE
  SomeType = some type;
  Property = PROCEDURE (IN e : SomeType) : BOOLEAN;
  Index = INTEGER;

PROCEDURE ExistsP (IN x : ARRAY OF SomeType; P : Property) : BOOLEAN;
  (*!
    Does x contain an element e with P (e)?
    Postcondition: result = ( $\exists i = 0 \dots \text{LEN}(x) - 1 : P(x[i])$ ).
  !*)
  VAR
    i : Index;
BEGIN
  i := 0;
  WHILE (i < LEN(x)) & ~P(x[i]) DO
    INC(i);
  END;
  ASSERT ((i >= LEN(x)) OR P(x[i]), BEC.invariant);
  RETURN i < LEN(x);
END ExistsP;

```

2.3**Eiffel**

In *Eiffel* lösen wir das Problem mit einer abstrakten generischen Klasse und dem Entwurfsmuster Schablonenmethode (*template method*). Die Klasse *SEARCHER* enthält die unbekannte Eigenschaft und die komplett implementierte Suchfunktion.

Programm 17

Eiffel: Lineare Suche in
einer Reihung

```

deferred class SEARCHER [E]
feature
  property (e : E) : BOOLEAN
    require
      object_attached: e /= Void
    deferred
    ensure
      object_unchanged: equal(e, old e)
    end -- property

  exists_property (a : ARRAY [E]) : BOOLEAN
    -- Does a contain an element e with property (e)?
    -- Design pattern: template method.
    require
      object_attached: a /= Void
    local
      i : INTEGER
    do
      from
        i := a.lower
      until i > a.upper or else a.item(i) /= Void and then property(a.item(i)) loop
        i := i + 1
      end
      Result := i <= a.upper
    ensure
      object_unchanged: equal(a, old a)
    end -- exists_property
end -- class SEARCHER

```

Die Grundmenge G setzen wir in den formalen generischen Parameter E von *SEARCHER* um. Die Eigenschaft P setzen wir in die abstrakte (eiffelisch: aufgeschobene, *deferred*) Funktion *property* um. Die Suchfunktion *existsP* wird zur Funktion

exists_property; sie ist eine Schablonenmethode, da sie mit dem unfertigen *property* und dem Suchalgorithmus fertig implementiert ist. Der Suchalgorithmus ergibt sich direkt aus Programm 14.

Man beachte: *property* ist streng und verbietet Argumente, die sich auf nichts beziehen. Das verursacht *exists_property* mehr Aufwand, es muss vor jedem Aufruf von *property* prüfen, ob sich das aktuelle Argument auf ein Objekt bezieht. Der Leser formuliere eine Variante mit liberalem *property*!

2.4

C++

In C++ hat der Programmierer die Wahl zwischen prozeduralen Optionen (ähnlich *Component Pascal*) und objektorientierten Optionen (ähnlich *Eiffel*). Da mit *Java* und *C#* objektorientierte Varianten folgen, begnügen wir uns mit prozeduralen – aber generischen – Varianten. Aus der Grundmenge *G* wird ein generischer Typ *T*. Die Suchfunktion *exists_p* ist generisch. Als Argumente hat sie

- für die Reihung *x* eine Referenz (&) auf ein konstantes *vector*-Objekt mit dem Elementtyp *T*:

`const std::vector<T> & v`

- für die Eigenschaft *P* einen Zeiger (*) auf eine Funktion, die eine Referenz auf ein konstantes *T*-Objekt erhält und *bool* liefert:

`bool (*property) (const T &)`

Die kopfgesteuerte Bedingungsschleife ist die *while*-Schleife, sie hat eine Fortsetzungsbedingung. Die Variable *result* ist wegen der *return*-Anweisung unnötig.

Programm 18

C++: Lineare Suche in einer Reihung

```
template <typename T>
bool exists_p (const std::vector<T> & v, bool (*property) (const T &)) {
    int i = 0;
    while (i < v.size() && !property(v[i])) {
        ++i;
    }
    assert(i <= v.size() || property(v[i]));
    return i < v.size();
}
```

Alternativen

C*-Programmierer finden schnell die Variante Programm 19. Dogmatische Puristen verpönen sie, da sie entgegen Leitlinie 1 die *for*-Schleife nicht als Zähl-, sondern als Bedingungsschleife benutzt missbraucht.

Programm 19

C++: Lineare Suche in einer Reihung



```
template <typename T>
bool exists_p (const std::vector<T> & v, bool (*property) (const T &)) {
    int i;
    for (i = 0; i < v.size() && !property(v[i]); ++i);
    return i < v.size();
}
```

2.5

Java

In *Java* lösen wir das Problem ähnlich wie in *Eiffel*. Die abstrakte generische Klasse *Searcher* hat den formalen generischen Parameter *E* für die Grundmenge *G*. Die Eigenschaft *P* erscheint als abstrakte Funktion *property*, die Suchfunktion *existsP* als Schablonenmethode *existsProperty*, die mit dem unfertigen *property* und dem Suchalgorithmus aus Programm 14 fertig implementiert ist.

Man beachte die verglichen mit dem *Eiffel*-Programm 17 geänderte Arbeitsteilung: *property* ist liberal und erlaubt Argumente, die sich auf nichts beziehen. Es soll aber

zusichern, dass es für null-Argumente false liefert. Das erleichtert existsProperty die Arbeit, es kann auf die null-Prüfung der Argumente für property verzichten. Der Leser formuliere eine Variante mit strengem property!

Programm 20

Java: Lineare Suche in einer Reihung

```
public abstract class Searcher<E> {
    public abstract boolean property (E e);
    // Postcondition: e == null returns false.
    public boolean existsProperty (E[] a) {
        assert a != null : "precondition";
        int i = 0;
        while (i < a.length && !property(a[i])) {
            i++;
        }
        assert i >= a.length || property(a[i]) : "termination condition";
        return i < a.length;
    }
}
```

2.6**C#**

Copy-Paste-Replace-Some-Characters erzeugt aus dem Java-Programm 20 eine C#-Lösung, über die kaum Neues zu berichten ist.

Programm 21

C#: Lineare Suche in einer Reihung

```
public abstract class Searcher<E> {
    public abstract bool Property (E e);
    // Postcondition: e == null returns false.
    public virtual bool ExistsProperty (E[] a) {
        System.Diagnostics.Debug.Assert(a != null, "precondition");
        int i = 0;
        while (i < a.Length && !Property(a[i])) {
            i++;
        }
        System.Diagnostics.Debug.Assert
            (i >= a.Length || Property(a[i]), "termination condition");
        return i < a.Length;
    }
}
```

2.7**Fazit**

- Zur Suche eines Elements mit einer Eigenschaft in einer Reihung mit n Elementen sind im Mittel $n / 2$ Schritte auszuführen, da die Suche mit dem Finden enden kann.
- Auch diese Berechnung lässt sich mit einer Rekursionsformel beschreiben.
- Aus der Rekursionsformel ergibt sich durch einfache Umformungsregeln ein iterativer Algorithmus, der durch Ausnutzen kurzer Auswertung zu beschleunigen ist.
- Da die Reihungselemente nur auszuwerten sind, bis die Suche erfolgreich ist, eignen sich Bedingungsschleifen am besten.
- Der systematisch konstruierte Suchalgorithmus ist strukturiert. Es gibt kaum einen rationalen Grund, dafür Zählschleifen zu missbrauchen oder Sprunganweisungen zu verwenden.

2.8**Aufgaben****Aufgabe 3**

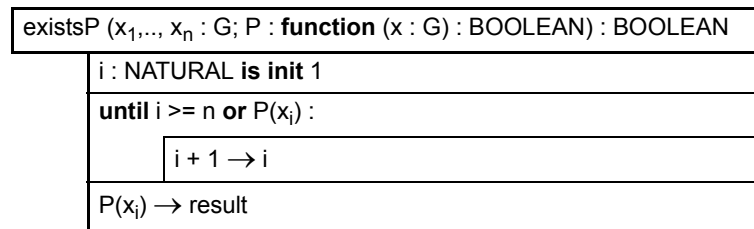
Fehler bei linearer Suche?

Untersuchen Sie die folgende Variante von Programm 14, in der **or** der Disjunktionsoperator mit langer Auswertung ist! Begründen Sie, ob oder unter welcher Bedingung der Algorithmus korrekt ist! Falls der Algorithmus nicht immer korrekt ist, geben Sie

- (1) für den Fehlerfall die Anweisung an, bei der sich die Fehlerursache auswirkt;
- (2) eine Vorbedingung für existsP an, die die Fehlerursache verhindert!

Programm 22

Struktogramm: Lineare
Suche in einer
Reihung, variiert



Ersetzt man in Programm 22 **or** durch das kurz auswertende **or else**, ist der Algorithmus dann stets korrekt? Falls nein, wie lässt er sich zu einer korrekten Variante ohne Vorbedingung verbessern?

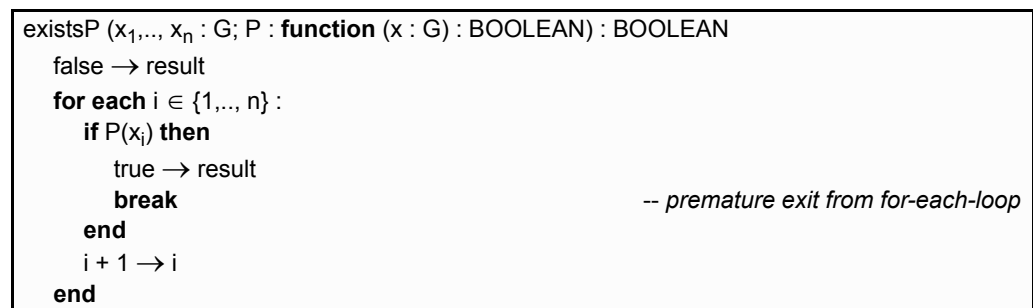
Aufgabe 4

Lineare Suche –
unstrukturiert

Sprachen, die noch Sprunganweisungen wie **return**, **break**, **continue** oder gar **goto** enthalten, erlauben noch unstrukturierte Varianten linearer Suche ähnlich Programm 23.

Programm 23

Pseudocode: Lineare
Suche in einer
Reihung, missbrauchte
Zählschleife mit
Raussprunganweisung
☹



Vorteilhaft daran mag sein, dass der Suchbereich im Zählschleifenkopf und die Suchbedingung beim **if** erkennbar sind und keine Bedingung mehrfach notiert und ausgewertet wird. Nachteilig sind die Unstrukturiertheit des Algorithmus durch die **break**-Anweisung, der Missbrauch der Zählschleife durch den Sprung aus dem Schleifenrumpf, und die durch **if-break** verschleierte zweiseitige Auswahl.

Transformieren Sie Programm 23 in nachvollziehbar korrekten Schritten in eine äquivalente strukturierte Variante ohne Sprunganweisungen, missbrauchte Zählschleifen und verschleierte Mehrfachauswahlen, und ohne eine Steuervariable einzuführen!

Aufgabe 5

Lineare Suche nach
Position

Gefragt ist oft nicht nur, ob ein Element mit einer definierten Eigenschaft in einem Suchraum vorkommt, sondern ggf. auch an welcher Position. Mit den Voraussetzungen und Bezeichnungen des Vorkommensproblems auf S. 11 und dem Prädikat $Exists_P$ lässt sich das Positionsproblem so formalisieren:

Positionsproblem
formal

Gesucht: Die n -stellige Funktion auf G , die die erste Position eines Elements in $x \in G^n$ mit der Eigenschaft P liefert, oder *notFound*, wenn es kein solches Element gibt:

$$\begin{aligned}
 MinIndex_P : G^n &\rightarrow \{1, \dots, n, notFound\}, \\
 (x_1, \dots, x_n) &\mapsto MinIndex_P(x_1, \dots, x_n) := \\
 &\begin{cases} \min\{i \in \{1, \dots, n\} \mid P(x_i)\} & \text{falls } Exists_P(x_1, \dots, x_n) \\ notFound & \text{sonst} \end{cases}
 \end{aligned}$$

Die Algorithmen des Positionsproblems ähneln denen des Vorkommensproblems; es sind lediglich die Rückgabewerte anzupassen: Statt *false* ist *notFound*, statt *true* die Position i zurückzugeben. Reformulieren Sie alle Programme zum Vorkommensproblem in diesem Text als Programme zum Positionsproblem!

Aufgabe 6

Kleinsten Index
suchen, Lehrbuch A

Ein einführendes Lehrbuch der Informatik schlägt in seiner dritten Auflage für eine Funktion `linearSearch`, die den Index des ersten Vorkommens eines Objekts `a` in einer Folge `f` von Objekten liefern soll, die folgende polymorphe *Java*-Lösung vor.

Programm 24

Java: Kleinsten Index
eines Objekts in einer
Reihung aus [KüW00]
S. 270, Format original,
aber Kommentar
entfernt



```
public static int linearSearch(Object[] f,
                                Object a) {
    int res = -1;
    for(int i = 0; i < f.length; i++) {
        if (f[i].equals(a))
            return (res = i);
    }
    return res;
}
```

Begründen Sie, unter welchen Bedingungen die Implementation korrekt ist und ergänzen Sie die Bedingungen als Zusicherungen! Finden und erläutern Sie mindestens vier weitere Schwächen der Implementation und transformieren Sie sie in eine äquivalente Implementation ohne die gefundenen Schwächen!

Aufgabe 7

Kleinsten Index
suchen, Lehrbuch B

Für eine Funktion `linearSearch`, die den Index des ersten Vorkommens einer Ganzzahl `key` in einer Reihung `numbers` von Ganzzahlen liefern soll, schlägt ein einführendes *Java*-Programmierkurs-Buch die folgende Lösung vor.

Programm 25

Java: Kleinsten Index
einer Ganzzahl in einer
Reihung aus [MHR07]
S. 252, Format original



```
static int linearSearch(int[] numbers, int key) {
    boolean found = false;
    int i = 0;
    // Repeat while more elements in array:
    while (i < numbers.length && !found) {
        // If the current element contains the key:
        // Return the index of the current element
        if (numbers[i] == key) {
            found = true;        // Found!
        } else {
            i++;                // Move to next element
        }
    }
    assert (i >= numbers.length || found) : "Error in linear search.";
    if (found)
        return i;              // The key is in element numbers[i]
    else
        return -1;             // The key is not found
}
```

- (1) Damit `linearSearch` korrekt wird, ergänzen Sie eine als Zusicherung implementierte Vorbedingung!



Programm 25 ist anfängergerecht üppig kommentiert, im Wesentlichen strukturiert, missbraucht keine Zählschleife und verzichtet auf unnötige Nebeneffekte. Dennoch enthält es zwei Schwächen und Überflüssiges: Eine Steuervariable mit Initialisierung, eine Zuweisung, eine strukturierte Anweisung.



- (2) Transformieren Sie Programm 25 in eine professionell kompakte und dennoch verständliche Variante ohne Schwächen!

Eine Lösung in *Component Pascal* mit *BlackBox* finden Sie in den Informatik-Hug-Subsystemen im Modul `MathVectorsOfReal` unter dem Namen `MinIndexOf` [Hug12].

Aufgabe 8

All-Prädikat

Abschnitt 2 beschränkt sich auf das Teilproblem (b) von S. 11, die Existenzform „Kommt in der Reihung ein Element mit der gegebenen Eigenschaft vor?“. Zum Teilproblem (a) von S. 11, die Allform „Haben alle Elemente in der Reihung die gegebene Eigenschaft?“, sollen Sie nun

- (1) alle Überlegungen analog durchführen,
- (2) die entsprechenden Programme entwickeln!

Aufgabe 9
Sortiertsein

Bei Folgen $(x_1, \dots, x_n) \in G^n$ in einer vollgeordneten Grundmenge $\langle G, \leq \rangle$ interessiert das Prädikat

$$\begin{aligned} \text{sortiert}(x_1, \dots, x_n) &: \Leftrightarrow \forall i, k = 1, \dots, n : (i \leq k \Rightarrow x_i \leq x_k) \\ &\Leftrightarrow \forall i = 1, \dots, n-1 : x_i \leq x_{i+1}. \end{aligned}$$

Was hat das mit Suchen zu tun? Wonach wird gesucht? Entwickeln Sie dazu

- (1) eine Rekursionsformel,
- (2) eine iterative strukturierte Lösung in Pseudocode,
- (3) iterative strukturierte Lösungen in *Eiffel*, *C++*, *Java*, *C#*!

Eine Lösung in *Component Pascal* mit *BlackBox* finden Sie in den Informatik-Hug-Subsystemen im Modul *MathSorterOfArrayOfInteger* unter dem Namen *Sorted* [HugI12].

Aufgabe 10
Einserfolgen gerader Länge

Ben Wegbreit präsentiert diese von Alan Perlis stammende Aufgabe: Gegeben ist eine Reihung von Bits (0 und 1). Gesucht ist die Aussage, ob alle Folgen von 1en in der Reihung gerade Länge haben [Weg76] S. 77–78. Entwickeln Sie dazu

- (1) eine iterative strukturierte Lösung in Pseudocode,
- (2) iterative strukturierte Lösungen in *Eiffel*, *C++*, *Java*, *C#*!

Aufgabe 11
Palindrom

Palindrome sind Zeichenfolgen, die sich vorwärts und rückwärts gleich lesen, wie

Ada
Otto
Reliefpfeiler
Ein Esel lese nie
Ein Neger mit Gazelle zagt im Regen nie

Leerzeichen und Klein-Großschreibung werden ignoriert. Formalisieren Sie die Palindrom-Eigenschaft! Was hat sie mit Suchen zu tun? Wonach wird gesucht? Entwickeln Sie dazu

- (1) eine Rekursionsformel,
- (2) eine iterative strukturierte Lösung in Pseudocode,
- (3) iterative strukturierte Lösungen in *Eiffel*, *C++*, *Java*, *C#*!

Eine Lösung für Zahlenpalindrome (z.B. 12321) in *Component Pascal* mit *BlackBox* steht in den Informatik-Hug-Subsystemen im Modul *11Naturals* unter *IsPalindrome* [HugI12].

Aufgabe 12
Exponentialfunktion

Eine Folge $x_1, \dots, x_n$ muss nicht in einer Reihung gespeichert sein. Auch wenn die Glieder x_i nach einer Rekursionsformel der Art

$$\begin{aligned} x_0 &= \text{Anfangswert,} \\ x_i &= \text{Funktion}(x_{i-1}) \quad \text{für } i = 1, \dots, n \end{aligned}$$

berechnet werden, lässt sich mit der vorgestellten Methode ein iterativer Algorithmus konstruieren. Die mathematische reelle **Exponentialfunktion** ist durch eine unendliche Reihe definiert:

$$\exp(x) := \sum_{i=0}^{\infty} \frac{x^i}{i!} \quad \text{für } x \in \mathbb{R}.$$

Die entsprechende Programmfunktion

`exp (x : REAL) : REAL`

soll Näherungswerte mit einem iterativen Algorithmus berechnen, der die Reihe bis zu einem Glied $x^i / i!$ summiert, dessen Betrag kleinergleich einem kleinen vorgegebenen $\epsilon > 0$ ist. Stellen Sie Rekursionsformeln für

- (1) eine endliche Anfangsreihe,
- (2) die Glieder der Reihe

auf! Teil (2) ist wichtig, weil die wiederholte Berechnung der Potenzen x^i und Fakultäten $i!$ bei jeder Iteration ineffizient wäre. Beachten Sie die Beziehungen

$$x^i = x^{i-1} * x,$$

$$i! = (i - 1)! * i.$$

Entwickeln Sie

- (3) eine iterative strukturierte Lösung in Pseudocode,
- (4) iterative strukturierte Lösungen in *Eiffel*, *C++*, *Java*, *C#*!

Eine Lösung für komplexe Zahlen in *Component Pascal* mit *BlackBox* steht in den Informatik-Hug-Subsystemen im Modul MathComplexFunctions unter SetExp [HugI12].

3

Ausblick

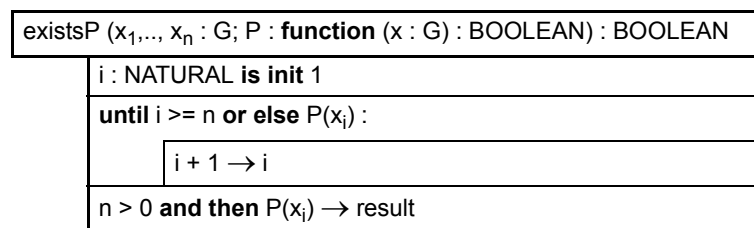
3.1

Effizienz verbessern

Lässt sich der Algorithmus in Programm 14, S. 16 beschleunigen? Ja, das letzte Reihungselement erst nach der Schleife zu prüfen spart eine Iteration. Das gilt aber nur, wenn es ein letztes, also mindestens ein Element gibt. Diese Bedingung vergessen zu haben ist der Fehler in Programm 22, S. 20. Eine korrigierte Variante ist Programm 26:

Programm 26

Struktogramm: Lineare
Suche in einer
Reihung, verbessert



Mit einer Iteration weniger ist aber kaum etwas gewonnen. Wo bieten sich weitere Sparchancen? Nachteilig für den Laufzeitaufwand ist in Programm 14, dass sich die Abbruchbedingung aus zwei Bedingungen disjunktiv zusammensetzt:

$$i > n \text{ or else } P(x_i)$$

So sind bei den meisten Iterationen beide Bedingungen auszuwerten. Wäre die zweite Abbruchbedingung $P(x_i)$ für wenigstens ein i erfüllt, so wäre die erste Abbruchbedingung $i > n$ verzichtbar und die Laufzeit reduzierbar.

Stoppelement

Ist das Prädikat P bekannt, so kann man temporär als letztes Reihungselement ein x mit $P(x)$ einsetzen. Solch ein Element, bei dem die Suche sicher stoppt, heißt **Stoppelement** (*Stopper*, *Wächter*; *sentinel*). Niklaus Wirth hat das Konzept erfunden und in seinem immer noch lesenswerten Buch *Algorithmen und Datenstrukturen* dargestellt [Wir86] S. 56, [Wir04] S. 34. Ein Beispiel dazu bringt [Hug01] in Programm 8.11 auf S. 223.

3.2

Kontext verbessern

Die Programme 17, 20 und 21 sollen vor allem den Suchalgorithmus demonstrieren. Für einen praktischen Einsatz bietet sich eine günstigere Organisation in einer Bibliothek etwa so an: Die generische Klasse SEARCHER [generic E] mutiert zu einer generischen Klasse TOOLSET_OF_ARRAY [generic E], die nicht nur die Suchfunktionen existsP

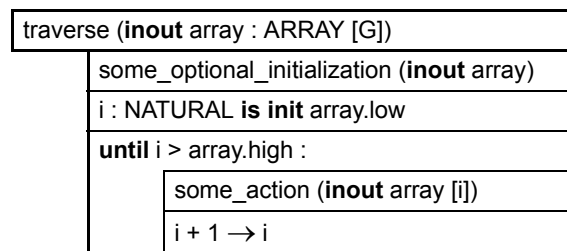
und allP enthält, sondern auch weitere Abfragen und Aktionen, die E-Reihungen bearbeiten. Die abstrakte generische Eigenschaft `property` ist erstens unabhängig von Reihungen existenzberechtigt, zweitens macht sie, wenn sie in `TOOLSET_OF_ARRAY` deklariert ist, diese Klasse abstrakt, also nicht direkt benutzbar. Deshalb ist es günstiger, für `property` eine eigene abstrakte generische Klasse (in *Java* oder *C#* ein generisches Interface) `PROPERTY [generic E]` vorzusehen. Die Suchfunktionen `existsP` und `allP` erhalten dann einen zusätzlichen Parameter des Typs `PROPERTY [E]`. Bei der `property`-Funktion ist der Einfachheit halber die liberale Variante von Programm 20 der strengen Variante von Programm 17 vorzuziehen.

3.3

Behälter verallgemeinern

Extrahiert man aus Programm 2 und Programm 14 das algorithmische Muster zum Traversieren einer Reihung, so erhält man etwa dies:

Programm 27
Struktogramm:
Traversieren einer
Reihung



Statt Reihungen können allgemeinere Arten von Behältern vorkommen, beispielsweise **lineare Strukturen mit einem Cursor**, der im Unterschied zu einem Reihungsindex außen nicht direkt sichtbar ist.¹ Der Cursor hat eine aktuelle Position, er steht entweder bei einem der Elemente in der Struktur oder bei nichts.

Programm 28
Pseudocode:
Schnittstelle einer
linearen Cursor-
Struktur

```

interface LINEAR [generic G]

  queries
    is_empty : BOOLEAN
      -- Contains no element?

    off : BOOLEAN
      -- Is no element at cursor position?

    item : G
      -- Element at cursor position
    pre
      not off
    end

  invariants
    is_empty implies off

  actions
    start
      post
        -- If is_empty, no effect.
        -- If not is_empty, then cursor is positioned at first element and not off.
      end

    forth
      pre
        not off
      post
        -- Cursor is positioned at next element or off.
      end

  end LINEAR

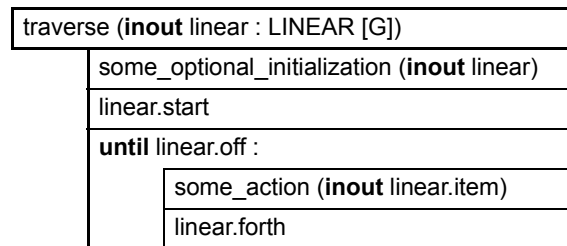
```

¹ Cursor heißt auf Deutsch Schieber. Ausnahmsweise benutze ich die englische Bezeichnung.

Programm 28 zeigt ein an die Klasse *LINEAR* der *Eiffel*-Standardbibliothek angelehntes Schnittstellenfragment. Damit ergibt sich aus Programm 27 durch einige lokale Änderungen, die die algorithmische Grundstruktur belassen, das folgende Traversierungsmuster für lineare Cursor-Strukturen.

Programm 29

Struktogramm:
Traversieren einer
linearen Cursor-
Struktur

**3.4****Aufgaben****Aufgabe 13**

Lineare Suche mit
Stoppelement

Transformieren Sie alle Lösungen von Suchproblemen, bei denen es möglich ist, in Lösungen mit Stoppelementen!

Aufgabe 14

Lineare Suche mit
Ausnahmebehandlung

In ihrer *Einführung in die Informatik* argumentieren Küchlin und Weber etwa so [KüW00] S. 270–271: *Java* prüft bei indizierten Reihungszugriffen zur Laufzeit, ob die Indizes gültig sind. Das Überschreiten einer Indexgrenze löst eine Ausnahme aus, die behandelt werden kann. Bei linearer Suche kann man statt eines Stoppelements eine Ausnahmebehandlung für den Indexüberlauf vorsehen. Damit ergibt sich für die Funktion `linearSearch`, die wie in Aufgabe 6 spezifiziert ist, folgende Implementation (die wie Programm 24 nur bedingt korrekt ist):

Programm 30

Java: Kleinster Index
eines Objekts in einer
Reihung, mit
Ausnahmebehandlung
nach [KüW00] S. 271



```
public static int linearSearch (Object[] f, Object a) {
    try {
        int i = 0;
        while (!f[i].equals(a)) i++;
        return i;
    } catch (IndexOutOfBoundsException ioobe) {
        return -1;
    }
}
```

Bedenken

Ein Stoppelement soll durch Vereinfachung der Schleifenbedingung die Effizienz steigern; die Kosten für das Ein- und Ausfügen des Stoppelements sind unerheblich. Auch die Lösung mit Ausnahmebehandlung hat nur eine einfache Schleifenbedingung, doch fallen zusätzlich die Kosten der Ausnahmebehandlung (der try-catch-Struktur) an. Sie sind aus dem Quelltext nicht ersichtlich, aber nicht zu vernachlässigen. Der Effizienzgewinn konfliktiert mit dem Effizienzverlust. Die theoretische Komplexitätsanalyse hilft hier nicht weiter.

Prüfen

Untersuchen Sie diesen Konflikt empirisch, indem Sie den Laufzeitaufwand der Implementationen der linearen Suche

- (1) ohne Stoppelement,
- (2) mit Stoppelement,
- (3) mit Ausnahmebehandlung

praktisch messen und vergleichen! Führen Sie mehrere Messreihen mit verschiedenen großen Reihungen durch!

Hinterfragen

Prinzipiell stellt sich die Frage: Wozu sind Ausnahmen nütze? Für Ausnahmefälle, die zwischen normalen Fällen und selbst programmierten Fehlern liegen! Auch bei unangenehmen Normalfällen ist Vorbeugen besser als Heilen. Man behandelt sie besser mit bedingten Anweisungen:



```

if müde then
    Auto anhalten
    eine Weile schlafen
end
weiterfahren

```

als mit Ausnahmebehandlungen:



```

try
    weiterfahren
catch (Unfall)
    -- am Steuer eingeschlafen
    Krankenwagen rufen
end

```

Andererseits: Wer in sein Programm Fehler hineinprogrammiert, wie soll der die Ausnahmebehandlungen dieser Fehler korrekt hinkriegen? Zusicherungen prüfen die Korrektheit von Programmezuständen. Eine verletzte Zusicherung löst in *Java* eine Ausnahme des Typs `Error` aus. Diese sollte *nicht* im Programm behandelt werden, sondern der Programmierer sollte das fehlerhafte Programm korrigieren.

Betrachtet man Indexprüfungen als implizite Zusicherungen, ungültige Indizes als Fehler, so wird man sie ungern als Stoppelementersatz für den Normalfall „Element nicht gefunden“ einsetzen. Ausnahmen sind nützlich in den Fällen, wo sie nicht an der Stelle ihrer Auslösung, sondern entlang der Aufrufkette zurück bei einer Kundenmethode behandelt werden, ohne dass sich die dazwischen liegenden Glieder der Aufrufkette darum kümmern müssen.

Aufgabe 15

Summe einer Reihung
mit
Ausnahmebehandlung

Küchlin und Weber meinen, dass beim Traversieren einer Reihung `a` der Test `i < a.length` verzichtbar ist, denn „Java als interpretierte Sprache macht diesen Test automatisch und erzeugt eine Ausnahme“ [KüW00] S. 270–271. Akzeptiert man dieses Argument (und ignoriert ästhetische Aspekte), so erhält man beispielsweise zum Summieren numerischer Reihungselemente statt Programm 7 das angeblich effizientere Programm 31:

Programm 31

Java: Summe einer
Reihung mit
Ausnahmebehandlung



```

public static float sum (float[] a) {
    assert a != null : "precondition";
    float result = 0;
    try {
        for (int i = 0; ; ++i) {
            result += a[i];
        }
    } catch (IndexOutOfBoundsException ioobe) {
    } finally {
        return result;
    }
}

```

Prüfen Sie die Behauptung empirisch, indem Sie die Laufzeitaufwände der Programme 7, 8, 31 praktisch messen und vergleichen! Führen Sie mehrere Messreihen mit verschieden großen Reihungen durch! Welche der drei Programmvarianten ist aus welchen Gründen vorzuziehen?

A

Glossar

Algorithmus, strukturierter Algorithmus, der nur aus den Strukturarten Sequenz, Alternative und Iteration aufgebaut ist und bei dem jede strukturierte Anweisung genau einen Eingang und genau einen Ausgang besitzt (Ein-Eingang-Ein-Ausgangs-Regel). Algorithmen, die Sprunganweisungen wie **goto**, **break**, **continue**, **exit**, aber auch **return** als nicht statisch letzte Anweisung verwenden, sind in diesem Sinn nicht strukturiert.

Bedingungsschleife Schleife, bei der eine wiederholt ausgewertete Bedingung entscheidet, ob noch ein Durchlauf erfolgt oder nicht. Die Anzahl der Durchläufe ergibt sich u.U. erst aus dem Ablauf der Schleife.

Laufvariable Bei einer Zählschleife eine ganzzahlige Variable, die eine vor Start der Schleife festgelegte Wertemenge durchläuft, d.h. bei jedem Schleifendurchlauf einen anderen Wert der Menge annimmt.

Schleifeninvariante Zusicherung bei einer Schleife, die vor und nach jeder Ausführung des Schleifenrumpfs gilt. Sie gilt also nach der Schleifeninitialisierung, bleibt bei jeder Iteration erhalten, und gilt nach der Schleifenterminierung.

Schleifennachbedingung Zusicherung bei einer Schleife, die nach der Schleifenterminierung gilt. Sie folgt aus der Konjunktion der Abbruchbedingung mit der Schleifeninvariante.

Schleifenvariable Bei einer Bedingungsschleife eine Variable, die vor der Schleife initialisiert wird, in der Schleifenbedingung vorkommt, und im Schleifenrumpf ihren Wert ändert.

Zählschleife Schleife, bei der die Anzahl der Durchläufe bei ihrem Start festliegt.

- **missbrauchte Z.** Schleife, die als Zählschleife erscheint, deren Durchlaufanzahl aber von ihrem Ablauf abhängt. Missbrauch einer Zählschleife ist möglich durch
 - ⊗ Wertzuweisungen an Lauf- oder Zählvariable im Schleifenrumpf,
 - ⊗ Sprunganweisungen wie **goto**, **break**, **continue**, **exit**, **return** im Schleifenrumpf.

Zählschleifenkonstrukt Strukturierte Anweisung einer höheren Programmiersprache, die Zählschleifen unterstützt.

Zählvariable Bei einer Zählschleife eine ganzzahlige Variable, die die Schleifendurchläufe zählt.

B

Literatur

- [HugI12] Karlheinz Hug: *Lehrmaterial zu Informatik 1 + 2 für mki-B*. Hochschule Reutlingen (2008)
http://informatik.karlheinz-hug.de/Lehrmaterial/Informatik_1+2/,
ftp://studinf.reutlingen-university.de/MKI/Hug/Lehrmaterial/Informatik_1+2/
- [HugTG] Karlheinz Hug: *Theoretische Grundlagen, Skript für mki-B*. Hochschule Reutlingen (2004)
http://informatik.karlheinz-hug.de/Lehrmaterial/Theoretische_Grundlagen_1+2/Skript/Skript.pdf,
ftp://studinf.reutlingen-university.de/MKI/Hug/Lehrmaterial/Theoretische_Grundlagen_1+2/Skript/Skript.pdf
- [KüW00] Wolfgang Küchlin, Andreas Weber: *Einführung in die Informatik. Objekt-orientiert mit Java*. Springer, Berlin (2000) 383 S.
- [Hug01] Karlheinz Hug: *Module, Klassen, Verträge. Ein Lehrbuch zur komponentenorientierten Softwarekonstruktion*. Vieweg, Braunschweig, Wiesbaden (2001) 2. Aufl., 446 S.
- [MHR07] Khalid A. Mughal, Torill Hamre, Rolf W. Rasmussen: *Java Actually. A First Course in Programming*. Thomson, London (2007) 374 S.
- [Weg76] Ben Wegbreit: *Goal-Directed Program Transformation*. IEEE Transactions on Software Engineering, Vol. SE-2, No. 2 (June 1976) S. 69–80,
http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1702344&tag=1
 (Zugriff 2012-09-25)

- [Wir86] Niklaus Wirth: *Algorithmen und Datenstrukturen mit Modula-2*. Teubner, Stuttgart (1986) 4. neubearbeit. u. erweit. Aufl., 299 S.
- [Wir04] Niklaus Wirth: *Algorithms and Data Structures. Oberon version*. (August 2004) 182 S.,
<http://www.ethoberon.ethz.ch/WirthPubl/AD.pdf> (Zugriff 2012-09-25)