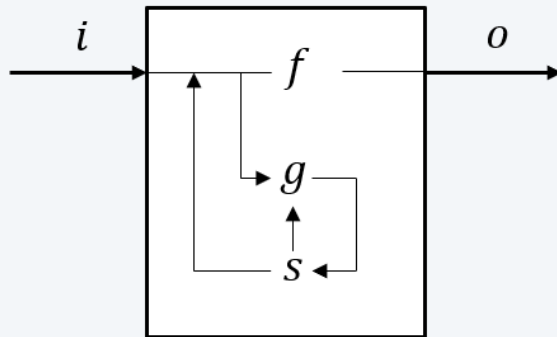


5 Algorithmen mit Gedächtnis



5.1 Begriff und Grundkonzept

5.2 Realisierungsformen

5.3 Anwendungsbeispiel

5.1 Begriff und Grundkonzept

Bisher haben wir Algorithmen der Form $o = f(i)$ betrachtet, d.h.

- In Analogie zu Funktionen im mathematischen Sinn
- Ausgangsgrößen (Ergebnisse, o) hängen nur von Eingangsgrößen (i) ab
- Frühere Aufrufe wirken nicht auf spätere Aufrufe
- Bei gleicher Eingabe liefert ein Algorithmus immer die gleiche Ausgabe

Algorithmus im mathematischen Sinn

```
var  
  a: array [1:10] of int  
  index: int
```

```
a := (1, 1, 2, 3, 5, 8, 13, 21, 34, 55)  
Search(↓a ↓13 ↑index)
```



liefert immer $\text{index} = 7$, egal wie oft Search aufgerufen wird

Begriff und Grundkonzept

Beispiel: Fortlaufende Mittelwertberechnung

Gesucht sei ein Algorithmus $\text{Average}(\downarrow \text{value: real } \uparrow \text{avg: real})$, der fortlaufend mit Messwerten $\text{value}_1, \text{value}_2, \text{value}_3, \text{value}_4, \dots$ aufgerufen wird und bei jedem Aufruf soll der arithmetische Mittelwert aller bisherigen Messwerte (Aufrufe) geliefert werden (avg).

streng genommen also kein Algorithmus,
weil gleiches rein müsste gleiches rauskommen

i	value _i	avg _i	
1	2.0	2.0	= 2.0 / 1
2	1.0	1.5	= (2.0 + 1.0) / 2
3	3.0	2.0	= (2.0 + 1.0 + 3.0) / 3
4	1.0	1.75	= (2.0 + 1.0 + 3.0 + 1.0) / 4
5	3.0	2.0	= (2.0 + 1.0 + 3.0 + 1.0 + 3.0) / 5

Ergebnisse hängen von früheren Aufrufen ab

⇒ erfordert Algorithmus mit Gedächtnis

Algorithmus mit Gedächtnis

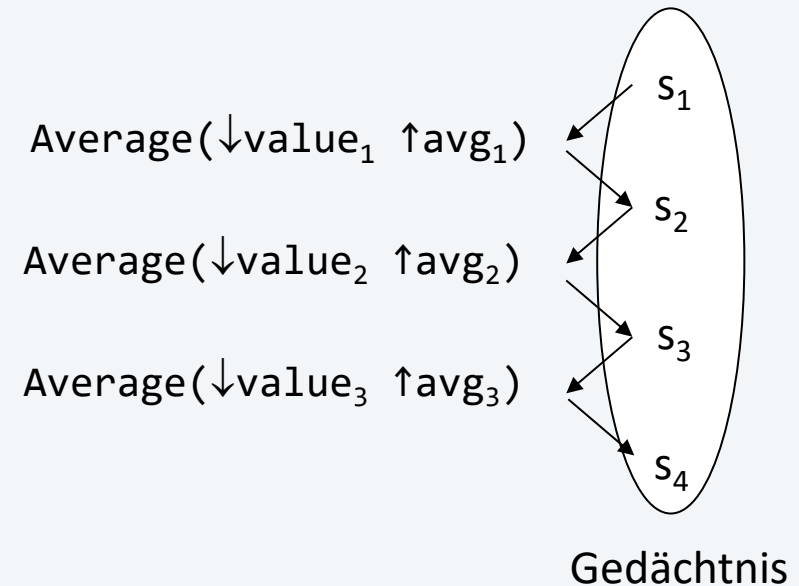
Um einen Algorithmus $A(\downarrow i: \dots \uparrow o: \dots)$ mit Gedächtnis beschreiben zu können, benötigen wir zwei Funktionen, eine zur Ergebnisberechnung und eine zur Fortschreibung des inneren Zustandes (= Gedächtnis)

- Berechnung des Ergebnisses

$$o = f(i, s_n)$$

- Berechnung des Folgezustands

$$s_{n+1} = g(i, s_n)$$

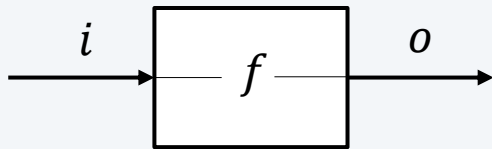


- Zustand (das Gedächtnis) s ist „versteckte“ Ein- und Ausgangsgröße, d.h. scheint nicht in Schnittstelle auf

Automatengleichungen

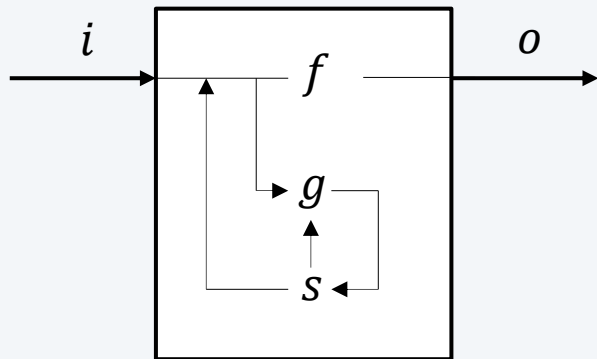
Einen Algorithmus mit Gedächtnis können wir mit Hilfe der Automaten-
gleichungen aus der Automatentheorie beschreiben

Für gewöhnliche Algorithmen gilt



$$o = f(i)$$

Für Algorithmen mit Gedächtnis gilt



$$o = f(i, s_n)$$

$$s_{n+1} = g(i, s_n)$$

Anwendungsbeispiel Mittelwertberechnung

Beispiel: Fortlaufende Mittelwertberechnung mit Zustandsmenge $s = (\text{sum}, n)$ und Eingangsobjekt $i = (v)$

```
Average(↓v: real ↑avg: real)
begin
  avg := (sum + v) / Real(↓n + 1)
  n := n + 1
  sum := sum + v
end Average
```

Typumwandlung von int --> real

$\text{avg} = f(v, n, \text{sum})$
 $n = g_1(v, n, \text{sum})$
 $\text{sum} = g_2(v, n, \text{sum})$

in Pascal: einer der beiden ist real, dann ist das Ergebnis auch real, td kann mans schreiben --> bessere Lesbarkeit

oder einfacher

```
Average(↓v: real ↑avg: real)
begin
  n := n + 1
  sum := sum + v
  avg := sum / Real(↓n)
end Average
```

Welche Werte haben n und sum beim ersten Aufruf?

Wo und wie wird der Zustand definiert und initialisiert und damit das Gedächtnis realisiert?

5.2 Realisierungsformen von Algorithmen mit Gedächtnis

Wie wird die Zustandsmenge (Gedächtnis) initialisiert (Anfangszustand)?

Wie bleiben die aktuellen Werte der Zustandsmenge von Aufruf zu Aufruf erhalten?

Die bisher eingeführten Konzepte für Datenobjekte reichen nicht aus:

```
Average(↓v: real ↑avg: real)
  var
    n: int
    sum: real
begin
  n := n + 1
  sum := sum + v
  avg := sum / Real(↓n)
end Average
```

Lokale Variablen: sind nur innerhalb der Prozedur / Funktion existent
--> Lebensdauer: Über welchen Zeitraum der Algorithmen/Programmausführung existieren Datenobjekte

==> kein Gedächtnis

Globale Variablen? haben schlechte Eigenschaften
deshalb gibt es statische Datenobjekte

- undefinierte Werte für sum und n beim ersten Aufruf
- Werte von sum und n gehen nach Aufruf verloren

Ein neues Konzept: statische Datenobjekte

Wir unterscheiden **lokale** und **statische** Datenobjekte

- **Lokale Datenobjekte** werden beim Aufruf eines Algorithmus angelegt, ihre Lebensdauer ist an die Lebensdauer des Algorithmus, in dem sie definiert sind gebunden, d.h. sie existieren nach Beendigung der Algorithmenausführung nicht mehr
- **Statische Datenobjekte** werden beim **ersten** Aufruf eines Algorithmus angelegt und mit einem Wert **initialisiert**, ihre Lebensdauer ist **unbegrenzt**, d.h. an die Lebensdauer des Algorithmus-Universums, in dem der Algorithmus sich befindet, gebunden existiert so lange, bis das Programm beendet wird

Zur Deklaration von statischen Variablen führen wir das Konstrukt `static` ein und deklarieren statische Variablen folgendermaßen:

```
static var  
  x: int := 0
```

Neben dem Attribut `static` enthält die Deklaration den Initialisierungswert

Ein neues Konzept: statische (persistente) Datenobjekte

Beispiel zur Illustration der Verwendung von statischen Variablen

```
F(↓a: int)
  var
    b: int
  static var
    c: int := 1
begin
  b := a + 1
  c := a + b + c
end F
```

Zeitpunkt	a	b	c
vor 1. Aufruf	-	-	1
F(↓4)	4	5	10
F(↓2)	2	3	15
F(↓3)	3	4	22

Wirkung

- Variable c wird angelegt, wenn der Algorithmus zum ersten Mal aufgerufen wird, und dann genau einmal initialisiert
- Gültigkeitsbereich/Sichtbarkeitsbereich von c ist auf F beschränkt
- Wert von c bleibt von Aufruf zu Aufruf erhalten, d.h. die Lebensdauer von c ist unbegrenzt

Anwendungsbeispiel Mittelwertberechnung

Realisierung mit statischen Variablen und interner Initialisierung

```
Average(↓v: real ↑avg: real)
  static var
    n: int := 0
    sum: real := 0.0
begin
  n := n + 1
  sum := sum + v
  avg := sum / Real(↓n)
end Average
```

- Zustandsvariablen sind nur innerhalb von Algorithmus sichtbar
- Zustandsvariablen können genau einmal initialisiert werden
- Zustandsvariable behalten ihren Wert von Aufruf zu Aufruf
- Nicht in jeder Programmiersprache möglich (C, C++: ja, Pascal: nein)

Anwendungsbeispiel Mittelwertberechnung

Realisierung mit globalen Variablen mit externer Initialisierung

Globale Zustandsvariablen sind auch außerhalb des Algorithmus jedem anderen Algorithmus im entsprechenden Universum bekannt und werden extern initialisiert

Algorithmus u. Zustandsvariablen

```
var
  n: int
  sum: real
```

```
n := 0
sum := 0.0
```

```
Average(↓v: real ↑avg: real)
begin
  n := n + 1
  sum := sum + v
  avg := sum / Real(↓n)
end Average
```

Verwendung

```
Average(↓3.0 ↑avg)
Average(↓2.0 ↑avg)

n := 0
sum := 0.0

Average(↓4.0 ↑avg)
```

liefert avg = 3.0

liefert avg = 2.5

liefert avg = 4.0

Anwendungsbeispiel Mittelwertberechnung

Realisierung mit globalen Variablen mit **internal** Initialisierung

Globale Zustandsvariablen werden durch speziellen Parameterwert (z.B. $v < 0$) initialisiert

```
var
  n: int
  sum: real
```

```
Average(↓v: real ↑avg: real)
begin
  if v < 0.0 then
    n := 0
    sum := 0.0
    avg := 0.0
  else
    n := n + 1
    sum := sum + v
    avg := sum / Real(↓n)
  end -- if
end Average
```

Average(↓3.0 ↑avg)

Average(↓2.0 ↑avg)

Average(↓-1.0 ↑avg)

Average(↓4.0 ↑avg)

← liefert avg = 3.0

← liefert avg = 2.5

← liefert avg = 4.0

← Initialisierung ist Teil
des Algorithmus.

Anwendungsbeispiel Mittelwertberechnung

Realisierung mittels (Übergangs-)Parameter

Zustandsvariablen werden als Parameter übergeben

```
Average(↓v: real ↑avg: real ↓↑n: int ↓↑sum: real)
begin
  n := n + 1
  sum := sum + v
  avg := sum / Real(↓n)
end Average
```

- Der Algorithmus selbst hat kein „Gedächtnis“, das Gedächtnis wird durch die Umgebung (den Rufer oder dessen Rufer) realisiert
- Zustandsvariablen müssen extern initialisiert werden
- Rufer muss Zustandsvariablen explizit übergeben
- Verschiedene Sätze von Zustandsvariablen möglich

Realisierung in Form von Modulen

Wir benötigen dazu ein neues Konzept und Konstrukt

Definition (*Modul*):

Ein Modul ist eine **Sammlung von Algorithmen und Datenobjekten** zur Bearbeitung einer in sich abgeschlossenen Aufgabe. Die Verwendung eines Moduls (d.h. seine Integration in ein Programmsystem) darf keine Kenntnis seines inneren Aufbaues und der konkreten Realisierung der in ihm gekapselten Algorithmen und Datenobjekte erfordern und seine Korrektheit muss ohne Kenntnis seiner Einbettung in ein bestimmtes Programmsystem nachprüfbar sein.

- Module werden deshalb auch Datenkapseln genannt und die Technik ihrer Verwendung **information hiding**
- Modulkonzept und Information-Hiding-Prinzip gehen auf D. L. Parnas und N. Wirth zurück
- Wir unterscheiden (wegen des Information-Hiding-Prinzips) zwischen der Modul-Schnittstelle und der Modul-Implementierung

Realisierung in Form von Modulen

Beispiel: Mittelwertberechnung realisiert mittels Modulkonzept

```
interface of MAverage  
  Init()  
  Average(↓v: real ↑avg: real)  
end MAverage
```

← Für Rufer sichtbar.

Wertzuweisungen aus dem Modul bleiben auch im Hauptprogramm gültig

Realisierung in Form von Modulen

Beispiel: Mittelwertberechnung realisiert mittels Modulkonzept

```
implementation of MAverage
var
  sum: real
  n: int
Init()
begin
  sum := 0.0
  n := 0
end Init
Average(↓v: real ↑avg: real)
begin
  n := n + 1
  sum := sum + v
  avg := sum / Real(↓n)
end Average
begin
  n := 0
  sum := 0.0
end MAverage
```

Für Rufer unsichtbar.

Gedächtnis, Zustandsmenge.

- Beliebige Initialisierung des Gedächtnisses zur Laufzeit ist möglich
- Die im Modul gekapselten Datenobjekte verhalten sich wie statische Variable, sie werden beim „Laden des Moduls“ angelegt, entsprechend den Anweisungen im Modulrumpf initialisiert, ihre Lebensdauer ist identisch mit der des Moduls und ihre Sichtbarkeit erstreckt sich auf alle Algorithmen der Modul-Implementierung

Modul-Verwendung

- Module werden nicht (wie Algorithmen) aufgerufen, sie werden vor der Ausführung eines Programmsystems „geladen“
- Auf die Datenobjekte eines Moduls kann nicht direkt zugegriffen werden; sie können nur über die im Interface angegebenen Algorithmen (auch Zugriffsprozeduren genannt) manipuliert werden
- Ein Algorithmus oder Modul kann ein Modul als Ganzes oder einzelne Algorithmen, die über das Interface zur Verfügung gestellt werden importieren und dann benutzen

```
from MAverage import Init, Average
```

```
Init()  
Average(↓10.0 ↑a)  -- a = 10.0  
Average(↓6.0 ↑a)  -- a =  8.0  
Init()  
Average(↓1.0 ↑a)   -- a =  1.0  
Average(↓3.0 ↑a)   -- a =  2.0  
Average(↓7.0 ↑a)   -- a =  3.66666666666667
```

```
var  
  a: real
```

Vorteile der Verwendung von Modulen (Datenkapseln)

Sicherheit

- Die Klienten-Algorithmen können die (gekapselten) Datenobjekte nicht missbräuchlich verwenden

Änderungsfreundlichkeit

- Da die konkrete Realisierung der gekapselten Datenobjekte (des Gedächtnisses) den Klienten-Algorithmen verborgen ist, kann sie geändert werden, ohne dass davon die Klienten-Algorithmen betroffen sind (sofern die Schnittstellen der Zugriffsprozeduren nicht verändert werden)

Mini-Aufgabe

Bei einem industriellen Prozess werden in unregelmäßigen Zeitabständen (alle paar Sekunden) Temperaturen gemessen. Dabei wird der Zeitpunkt der Messung (Zeitstempel des aktuellen Tages in Sekunden) sowie die Temperatur (Grad Celsius, auf 0.1 Grad genau) aufgezeichnet.

Gesucht ist ein Funktionsalgorithmus, der prüft, ob ein Temperaturmesswert plausibel ist. Ein Messwert ist dann plausibel, wenn die Veränderung des Messwerts seit der letzten Messung ± 2.0 Grad Celsius pro Sekunde nicht übersteigt. Der Funktionsalgorithmus soll `true` zurückgeben, wenn der Messwert plausibel ist, und `false`, wenn er das nicht ist.

Schnittstelle:

```
CheckValue(↓value: real ↓timestamp: int): bool
```

5.3 Anwendungsbeispiel

- Für die Lösung von statistischen Aufgabenstellungen etc. aber auch in der betrieblichen Praxis (z. B. für die Planung der Entnahme von Stichproben zur Qualitätssicherung von Produktionsprozessen), werden Zufallszahlen benötigt
- Wir beschäftigen uns damit, wie man mit **deterministischen Mitteln (Algorithmen)** „Zufall erzeugen“ kann
- Es versteht sich von selbst, dass es sich dabei nicht um „echten“ oder „reinen“ Zufall handelt, bei dem Ereignisse völlig unvorhersehbar (kausal nicht erklärbar) eintreten
- Es handelt sich dabei um eine spezielle Art von **berechnetem Zufall**, der sich in Form von Zahlen manifestiert. Man kann deshalb in diesem Zusammenhang auch von „**künstlichem**“, „**quasi**“ oder „**Pseudo-Zufall**“ sprechen

Zufallszahl und Zufallszahlengenerator

Begriff Zufallszahlengenerator

- Darunter verstehen wir einen Algorithmus mit Gedächtnis, der bei jedem Aufruf eine Zahl aus einem bestimmten Wertebereich liefert, die in so unübersichtlicher Weise von früheren Aufrufen abhängt, dass sie als zufällige Größe erscheint

`x := IntRand()` *integer random number*

Begriff Zufallszahl

- Zahl aus einem bestimmten Wertebereich
- deren Wert regellos (= nicht vorhersagbar) ist
- die Teil einer Zahlenfolge ist, die eine bestimmte statistische Verteilung (z. B. Gleichverteilung) aufweist und die bestimmte Tests (z.B. Chi²-Test) erfüllt

Beispiel: Würfel liefert gleichverteilt Zahlen aus dem Bereich 1 .. 6



Lineare-Kongruenz-Methode (*LKM*)

Standardverfahren nach Derrick H. Lehmer (1949)

$$x_{n+1} = (a \cdot x_n + c) \bmod m$$

mit

▪ Modul	m	möglichst groß
▪ Multiplikator	a	$2 \leq a < m$
▪ Inkrement	c	$0 \leq c < m$
▪ vorherige Zufallszahl	x_n	$0 \leq x_n < m$
▪ nächste Zufallszahl	x_{n+1}	$0 \leq x_{n+1} < m$

Güte des Generators hängt von Wahl der Faktoren a , c und m ab

Algorithmus mit Gedächtnis

Nach der Linearen-Kongruenz-Methode lässt sich der folgende (Standard-) Zufallszahlengenerator implementieren:

```
IntRand(): int
  const
    m = 65536
    a = 3421
    c = 1
  static var
    x: int := 0
begin
  x := (a*x + c) mod m
  return x
end IntRand
```

Anwendungen und Tests

```
for i := 1 to 10 do  
  WriteLn(↓IntRand())  
end
```

49914
34515
45480
4617
582
24943
2132
19077
54098
61131

```
cnt := 1  
first := IntRand()  
x := IntRand()  
while x ≠ first do  
  x := IntRand()  
  cnt := cnt + 1  
end  
WriteLn(↓"Periodenlaenge = " ↓cnt)  
Periodenlaenge = 65536
```

Anwendungen und Tests

```
const  
  width = 500  
  height = 300
```

```
SetPenColor(↓50 ↓81 ↓148)  
DrawSize(↓width ↓height)  
for i := 1 to 1000 do  
  x := IntRand() mod width  
  y := IntRand() mod height  
  DrawPoint(↓x ↓y)  
end
```

