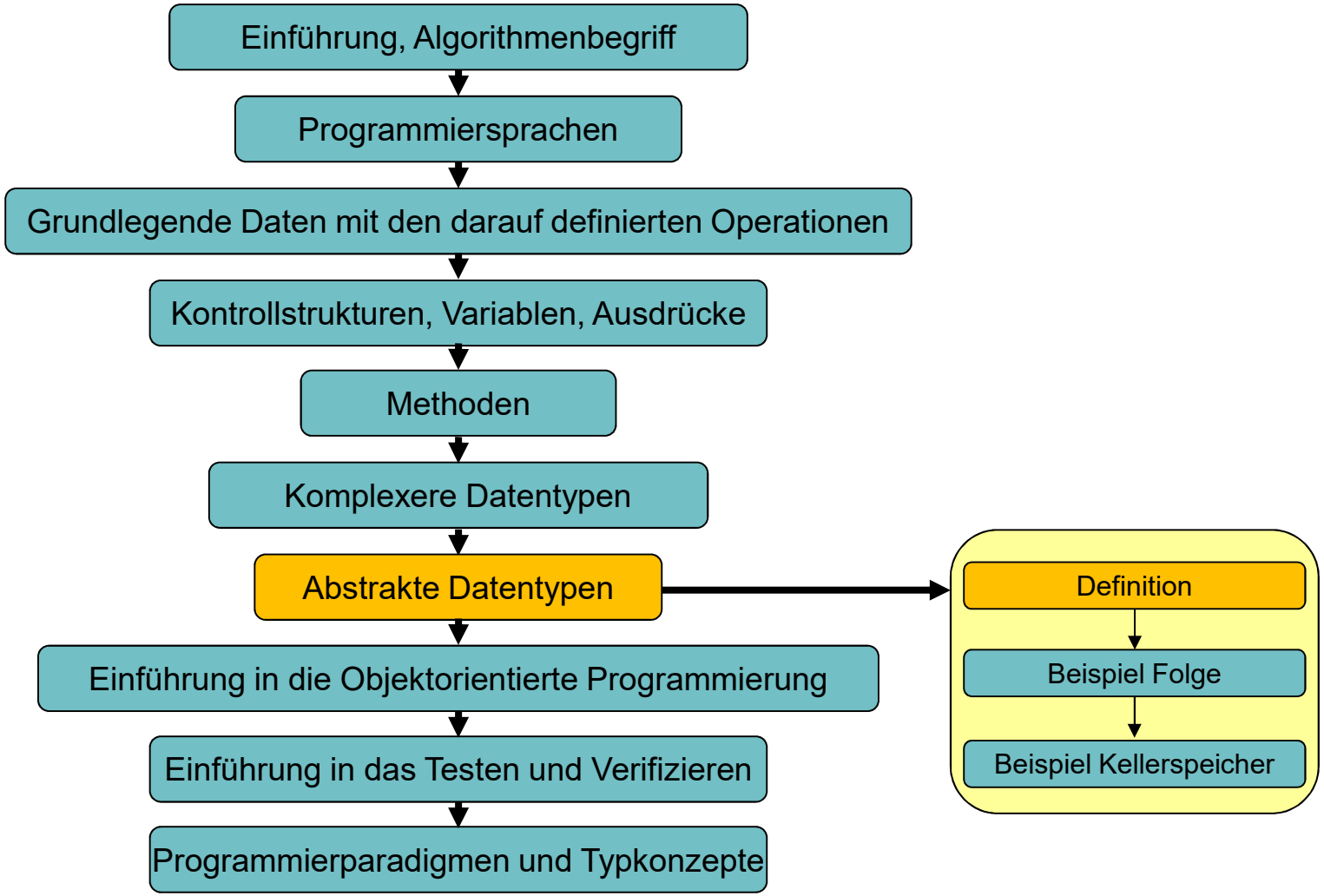
Inhalt dieser Veranstaltung



Was sind Daten?

- Als Daten haben wir bis jetzt überwiegend mit einfachen / primitiven Werten wie den natürlichen Zahlen mit ihrer üblichen Bedeutung und den gewohnten Operationen (Addition, Subtraktion) genommen.

- Spätestens, wenn wir auch komplexere Informationen verarbeiten wollen, stellt sich eine Reihe von Fragen:

  – Was sind überhaupt Daten? Welche sind möglich?

  – Wie kann man Daten (z.B. in einem Rechner) darstellen?

  – Welche Operationen sind auf den Daten erlaubt?

- Diese Fragen führen uns zur näheren Betrachtung des Datentyp-Begriffs.

Datentyp (1. Annäherung)

- 1. Annäherung: Ein Datentyp ist eine Menge von Werten mit den darauf definierten Operationen.

- Beispiel: Menge der Boolschen Werte B = {wahr, falsch} mit den Operationen

  $\neg : B \rightarrow B$

  $\wedge : B \times B \rightarrow B$

  $\vee : B \times B \rightarrow B$

- Wir wollen auch hier zwischen der Spezifikation und der konkreten Implementierung (Programmierung) unterscheiden.

- Deshalb:
  - Abstrakter Datentyp: Spezifikation von Eigenschaften
  - Konkreter Datentyp: zusätzlich die Realisierung dieser Eigenschaften

Abstrakter Datentyp

- Definition:
  Sei S eine Menge von Sorten und Ops eine Menge von Operationssymbolen.
  Ein abstrakter Datentyp $T = <S^T, Ops^T>$ besteht aus einer Familie von Trägermengen $\{ s_i^T \mid s_i \in S\}$ und einer Familie von Operationen $\{Ops_i^T \mid Ops_i \in Ops\}$ auf diesen Trägermengen.

- Erläuterungen zu den verwandten Begriffen:
  - Sorten: Namen für Wertemengen
  - Ops: Namen von Operationen
  - Trägermenge: Werte
  - Operationen: Funktionen zu den Operationssymbolen, arbeiten auf den Werten

Namen

Bedeutung

Beispiel 1: Boolsche Werte mit Operationen

- Wir wollen einen abstrakten Datentyp boolean definieren.

  $S = \{ bool \}$ Sorten

  $Ops_{Bool} = \{ not, and, or\}$ Operationssymbole

- Trägermenge zu bool: B = {wahr, falsch}
- Operationen in $Ops_{Bool}{}^{boolean}$:

  $not^{boolean} : B \rightarrow B$ mit $not^{boolean}(x) := \neg x$

  $and^{boolean} : B \times B \rightarrow B$ mit $and^{boolean}(x,y) := x \wedge y$

  $or^{boolean} : B \times B \rightarrow B$ mit $or^{boolean}(x,y) := x \vee y$

- Der abstrakte Datentyp kann dann angegeben werden durch

  $boolean = < \{B\}, Ops_{Bool}{}^{boolean} >$

Beispiel 2: Natürliche Zahlen

- Wir wollen einen abstrakten Datentyp nat definieren.

  S = { bool, NZahlen } Sorten

  Ops = { add, sub, equal, not, and,...} Operationssymbole

- Trägermenge zu bool: B = {wahr, falsch}
- Trägermenge zu NZahlen: N = {0,1,2,3,...}

- Operationen in Ops^{nat}:

  $add^{nat} : N \times N \rightarrow N$ mit $add^{nat}(x,y) := x+y$

  $sub^{nat} : N \times N \rightarrow N$ mit $sub^{nat}(x,y) := x-y$ (max. 0)

  $equal^{nat} : N \times N \rightarrow B$ mit $equal^{nat}(x,y) := (x = y)$

  $not^{nat} : B \rightarrow B$ mit $not^{nat}(x) := \neg x$

  $and^{nat} : B \times B \rightarrow B$ mit $and^{nat}(x,y) := x \wedge y$

  ...

- Der abstrakte Datentyp kann dann angegeben werden durch

  $nat = < \{B,N\}, Ops^{nat} >$

Abstrakter / konkreter Datentyp

- Bei einem Datentyp ist zu unterscheiden zwischen der konkreten Implementierung (z.B. auf einem Rechner) und dem abstrakten Datentyp.

- Das Beispiel der natürlichen Zahlen war als abstrakter Datentyp angegeben: Wir haben keine Aussagen darüber gemacht, wie z.B. der (mathematische) Wert 4 dargestellt werden soll. Mögliche Darstellungen für den Wert 4:

4	(Ziffer 4)
||||	(Strichdarstellung)
IV	(römische Ziffern)
0100	(Bitmuster in einem Rechner)

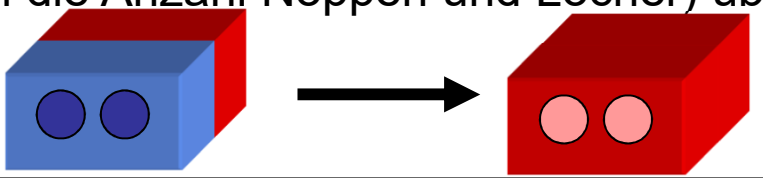
- Ein abstrakter Datentyp abstrahiert also von vielen möglichen Darstellungsmöglichkeiten und enthält nur die wesentlichen Eigenschaften. Den abstrakten Datentyp N können wir auf einem Großrechner genauso wie in einer Waschmaschinensteuerung konkret implementieren. Die Operationen add, sub,... sind in allen Implementierungen gleich verfügbar, die **internen Details** sind bei einem abstrakten Datentyp nicht sichtbar.

Verfeinerung

- Abstrakte Datentypen sind sehr hilfsreich im Software-Entwurfsprozess.
- Man lässt die konkreten Details der Realisierung zuerst außen vor und konzentriert sich auf die wesentlichen Aspekte des Datentyps (Wertemenge, abstrakte Operationen).

- Kommt man später zu einer Stufe, wo eine konkrete Implementierung vorgenommen werden muss, so setzt man den abstrakten Datentypen in einem konkreten Datentypen um.

- Es ist sogar möglich, mehrere austauschbare konkrete Datentypen bereitzustellen, die alle die gleiche Schnittstelle besitzen (den abstrakten Datentyp!)
- Dies ist wiederum vergleichbar mit Lego-Bausteinen: Wir können einen roten Stein durch einen blauen Stein ersetzen, solange die Schnittstelle (im Fall Lego-Baustein die Anzahl Noppen und Löcher) übereinstimmt.



Beispiel

- Problem: Gegeben seien 4 Punkte $P_1,...,P_4$ im zweidimensionalen euklidischen Raum. Es soll die Weglänge l des Weges $P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4$ berechnet werden.

- Lösung: Führe die Lösung auf die Berechnung der Entfernung zwischen zwei Punkten zurück: $l=d(P_1,P_2) + d(P_2,P_3) + d(P_3,P_4)$

  Diese Formel bleibt gleich, egal welche konkrete Realisierung es für d() gibt!

- Kartesische Koordinaten: Darstellung eines Punktes P_i: (x_i,y_i)

$$d(P_1,P_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

- Polarkoordinaten: Darstellung eines Punktes P_i: (r_i,φ_i)

$$d(P_1,P_2) = \sqrt{r_1^2 + r_2^2 - 2r_1r_2\cos(\varphi_2 - \varphi_1)}$$

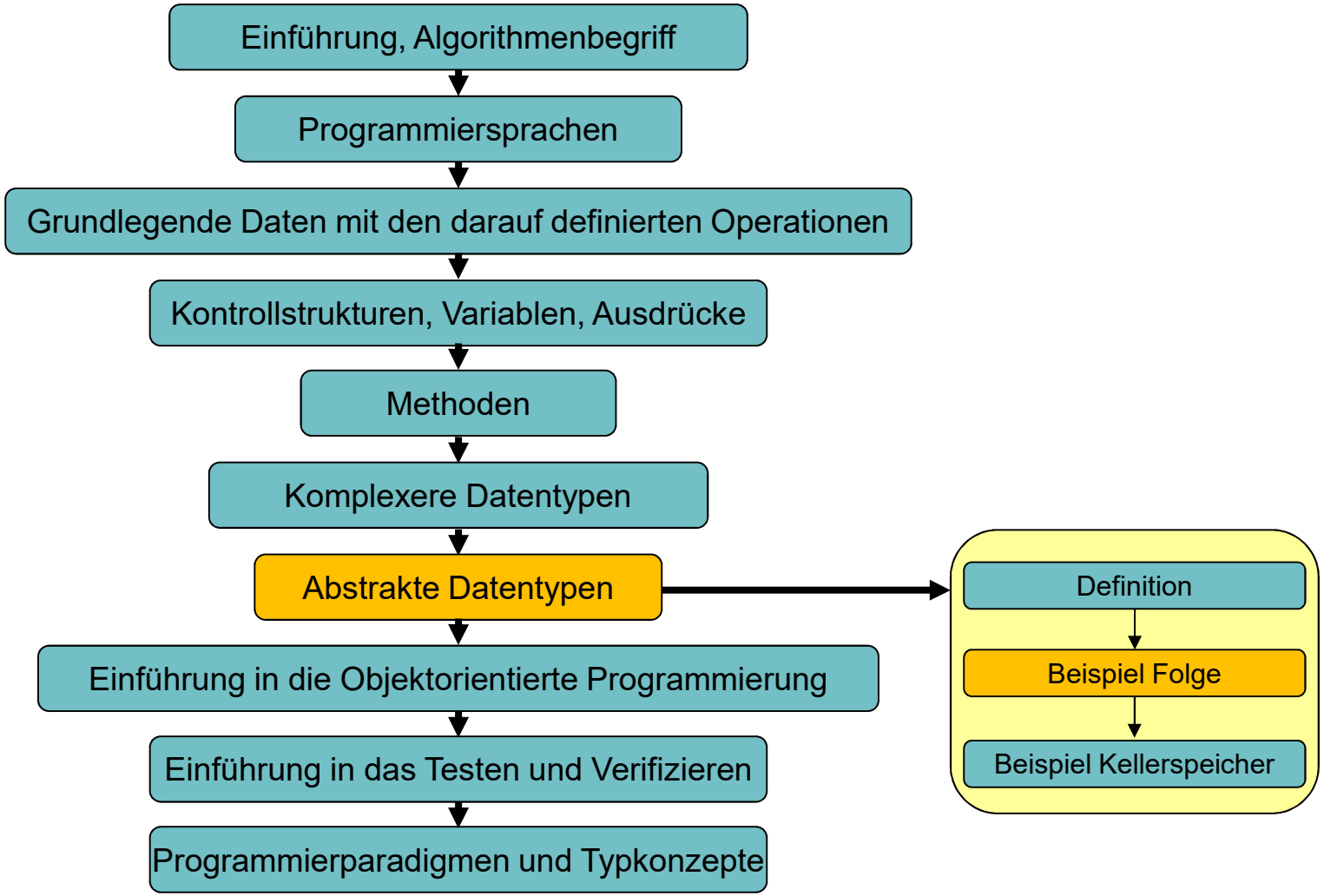
Zwischenstand

- Man unterscheidet zwischen einem abstrakten und konkreten Datentyp
- Bei einem abstrakten Datentyp stehen die verfügbaren Operationen im Vordergrund, ohne sich um die konkrete Realisierung Gedanken machen zu müssen
- Zu einem abstrakten Datentyp kann es mehrere konkrete Datentypen geben, die (intern) die Operationen unterschiedlich umsetzen können
- Nutzt man beim Programmieren nur die abstrakten Operationen, so ist die konkrete Realisierung austauschbar

Reflektion

- Geben Sie einen abstrakten Datentyp an für Brüche.

Inhalt dieser Veranstaltung



Beispiele für abstrakte Datentypen

- Im Folgenden einige Beispiele für abstrakte Datentypen:

  1. Folge
  2. Kellerspeicher

- Wir werden sehen,
  1. wie man neue Operationen über vorhandenen Operationen definieren kann
  2. wie man zur Definition neuer Datentypen auf vorhandene Datentypen zurückgreifen kann

Vorbemerkung: Mathematische Folge

- Definition:
  - Seien $M_1,...,M_n$ Mengen ($n \geq 0$). Dann heißt $M_1 \times ... \times M_n$ ein kartesisches Produkt.
  - Jedes $(x_1,...,x_n)$ mit $x_1 \in M_1, ..., x_n \in M_n$ heißt Tupel der Länge n über $M_1,...,M_n$.
  - x_i heißt i-te Komponente des Tupels $(x_1,...,x_n)$

- Definition: Ein Tupel der Länge n über $M_1,...,M_n$, für das gilt $M_1=...=M_n =: M$, heißt Folge (über M).

- Schreibweisen:
- $x = <x_1,...,x_n>$, $x_i \in M$, $1 \leq i \leq n$, $n \geq 0$
- Leere Folge (n=0): $x = <> =: \varepsilon$

- Für zwei Folgen $f_1=<x_1,...,x_n>$ und $f_2=<y_1,...,y_m>$ ($n,m \geq 0$) ist der $\circ$-Operator definiert durch $<x_1,...,x_n> \circ <y_1,...,y_m> = < x_1,...,x_n, y_1,...,y_m>$.

Beispiele

- M = { A,...,Z, a...,z}

- Folgen (über M):
  1. <>
  2. ε
  3. <a>
  4. <a,b,c>
  5. <A,a,C>
  6. <a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z>
  7. <z,y,x,w,v,u,t,s,r,q,p,o,n,m,l,k,j,i,h,g,f,e,d,c,b,s>

- Konkatenation von Folgen:
  1. <> ° <> = <>
  2. <a,b> ° <c,d> = <a,b,c,d>
  3. <> ° <c,d> = <c,d>

M^n

- Definition: Sei M eine Menge. Dann ist M^n für $n \geq 0$ rekursiv definiert durch:
  - $M^0 = \{ \varepsilon \}$
  - $M^1 = M$
  - $M^{n+1} = M^n \times M$

- Weiterhin ist definiert:
  - $M^* = M^0 \cup M^1 \cup M^2 \cup \ldots$ Menge aller Folgen über M
  - $M^+ = M^1 \cup M^2 \cup \ldots$ Menge aller nichtleeren Folgen über M
    $= M^* - \{ \varepsilon \}$

Beispiel

Sei M = { a,b }.

M^0 = { ε }
M^1 = { <a>, <b> }
M^2 = { <a,a>, <a,b>, <b,a>, <b,b> }
M^3 = { <a,a,a>, <a,a,b>, <a,b,a>, <a,b,b>, ... , <b,b,b> }

In M^*: <>, <a>, <a,b,a,b>, <b,b,a,a>, <a,a,a,a,a,...>
In M^+: <a>, <a,b,a,b>, <b,b,a,a>, <a,a,a,a,a,...>

Nicht in M^*: <a,c>
Nicht in M^+: <>, <a,c>

Abstrakter Datentyp Folge

- Sei M eine beliebige, nicht-leere Menge. Wir betrachten den Datentyp

  $folge = < W^{folge} , Ops^{folge} >$

mit

$W^{folge} = \{ M, M^*, B \}$

$Ops^{folge} = \{emptyfolge, makefolge, first, rest, concat, isemptyfolge\}$

wobei B für die Menge der Boolschen Werte steht. (Ops^{folge} auf nächster Folie)

- Bemerkung:
  Mit spitzen Klammern <> werden allgemein in der Mathematik algebraische Strukturen notiert. Da sowohl ein abstrakter Datentyp als auch eine Folge eine algebraische Struktur ist, darf man den abstrakten Datentyp $< W^{folge} , Ops^{folge} >$ nicht mit einer Folge $<x_1,...,x_n>$ durcheinander bringen.

Ops^{folge} (2)

emptyfolge : $\varnothing \rightarrow M^*$ mit

emptyfolge() = ε

makefolge : $M \rightarrow M^*$ mit

makefolge(m) = < m >

first : $M^* \text{ -} \rightarrow M$ mit

$$\text{first}(f) = \begin{cases} \bot & \text{falls } f = \varepsilon \\ x_1 & \text{falls } f = \langle x_1,\ldots,x_n \rangle \text{ mit } n > 0 \end{cases}$$

rest : $M^* \text{ -} \rightarrow M^*$ mit

$$\text{rest}(f) = \begin{cases} \bot & \text{falls } f = \varepsilon \\ \langle x_2,\ldots,x_n \rangle & \text{falls } f = \langle x_1,\ldots,x_n \rangle \text{ mit } n > 0 \end{cases}$$

Ops^{folge} (2)

concat : $M^* \times M^* \rightarrow M^*$ mit

$$concat(f_1, f_2) = f_1 \circ f_2$$

isemptyfolge : $M^* \rightarrow B$ mit

$$isemptyfolge(f) = \begin{cases} wahr & falls\ f = \varepsilon \\ falsch & sonst \end{cases}$$

isequal : $M^* \times M^* \rightarrow B$ mit

$$isequal(f_1,f_2) = \begin{cases} wahr & falls\ (f_1 = \varepsilon) \wedge (f_2 = \varepsilon) \\ falsch & falls\ (f_i = \varepsilon) \wedge (f_j \neq \varepsilon)\ mit\ i,j \in \{1,2\},\ i \neq j \\ (first(f_1) = first(f_2)) \wedge isequal(rest(f_1), rest(f_2)) & \\ & sonst \end{cases}$$

Anwendungsbeispiele

Sei M = { a,b }.

(1) Die Folge f = < a,b > lässt sich erzeugen mit:

f = concat(makefolge(a), makefolge(b))

(2) Der Test, ob die beiden Folgen f_1 = < a,b,c > und f_2 = < a,b,a > gleich sind, geschieht folgendermaßen:

isequal(<a,b,c>,<a,b,a>)

= (first(<a,b,c) = first(<a,b,a>) $\wedge$ isequal(rest(<a,b,c>), rest(<a,b,a>))

= (a=a) $\wedge$ isequal(<b,c>, <b,a>)

= (first(<b,c>) = first(<b,a>)) $\wedge$ isequal(rest(<b,c>), rest(<b,a>))

= (b= b) $\wedge$ isequal(<c>, <a>)

= (first(<c>) = first(<a>)) $\wedge$ isequal(rest(<c>), rest(<a>))

= (c = a) $\wedge$ isequal(ε, ε)

= falsch

"Programmieren" mit dem Abstr. Datentyp *folge*

Definition: Ein Palindrom ist ein Wort, im dem der i-te Buchstabe von vorne gleich dem i-ten Buchstaben von hinten ist.

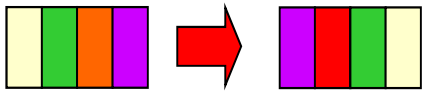
Beispiel für ein Palindrom:

otto

Mit Hilfe der bekannten Funktionen eines Abstrakten Datentyps lassen sich neue Funktionen definieren.

Dazu definieren wir uns zuerst einige Hilfsfunktionen (neue Operationen) aufbauend auf den bereits bekannten Funktionen, bevor wir die eigentliche Funktion palindrom definieren.

Hilfsfunktion spiegel



spiegel : M* → M* mit:

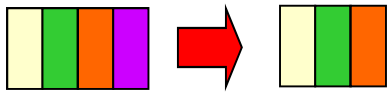
$$\text{spiegel}(f) = \begin{cases} \varepsilon & \text{falls } f = \varepsilon \\ \text{concat}(\text{spiegel}(\text{rest}(f)), \text{makefolge}(\text{first}(f))) & \text{sonst} \end{cases}$$

spiegel liefert für eine Folge $<x_1,...,x_n>$ die Folge $<x_n,...,x_1)$.

Beispiel:

spiegel(<a,b,c>)
= concat (spiegel (rest (<a,b,c>)), makefolge (first (<a,b,c>)))
= concat (spiegel (<b,c>), <a>)
= concat (concat (spiegel (rest (<b,c>)), makefolge (first (<b,c>))), <a>)
= concat (concat (spiegel (<c>), <b>), <a>)
= concat(concat(concat(spiegel(rest(<c>),makefolge(first(<c>)), <b>), <a>)
= concat(concat(concat(spiegel(ε), <c>), <b>), <a>)
= concat(concat(concat(ε, <c>), <b>), <a>)
= concat(concat(<c>, <b>), <a>)
= concat(<c,b>, <a>)
= <c,b,a>

Hilfsfunktion head



head : M^* - $\rightarrow$ M^* mit:

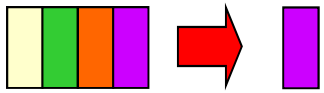
$$\text{head}(f) = \begin{cases} \perp \text{ falls } f = \varepsilon \\ \text{spiegel}(\text{rest}(\text{spiegel}(f))) \end{cases}$$

head liefert für eine Folge $<x_1,...,x_n>$ die Folge $<x_1,...,x_{n-1}>$.

Beispiel:

head(<a,b,c>)
= spiegel (rest (spiegel (<a,b,c>)))
= spiegel (rest (<c,b,a>))
= spiegel (<b,a>)
= <a,b>

Hilfsfunktion last



last : M^* - $\rightarrow$ M mit:

$$\text{last}(f) = \begin{cases} \perp & \text{falls } f = \varepsilon \\ \text{first}(f) & \text{falls rest}(f) = \varepsilon \\ \text{last}(\text{rest}(f)) & \text{sonst} \end{cases}$$

last liefert für eine Folge $<x_1,...,x_n>$ das letzte Element x_n.

Beispiel:

last(<a,b,c>)
= last (rest (<a,b,c>))
= last (<b,c>)
= last (rest (<b,c>))
= last (<c>)
= first (<c>)
= c

Und jetzt die eigentliche Funktion…

palindrom : $M^* \rightarrow B$ mit:

$$\text{palindrom}(f) = \begin{cases} \text{wahr} & \text{falls } (f = \varepsilon) \vee (\text{rest}(f) = \varepsilon) \\ \text{falsch} & \text{falls } \text{first}(f) \neq \text{last}(f) \\ \text{palindrom}(\text{rest}(\text{head}(f)) & \text{sonst} \end{cases}$$

1. Falls die Folge leer ist oder nur ein Element enthält, so ist sie ein Palindrom.
2. Falls die Folge zu Beginn einen anderen Wert hat als am Ende, kann sie kein Palindrom sein
3. Ansonsten teste die Folge ohne erstes und letztes Element auf die Palindromeigenschaft.

Beispiel 1

palindrom : M* $\rightarrow$ B mit:

$$\text{palindrom}(f) = \begin{cases} \text{wahr} & \text{falls } (f = \varepsilon) \vee (\text{rest}(f) = \varepsilon) \\ \text{falsch} & \text{falls } \text{first}(f) \neq \text{last}(f) \\ \text{palindrom}(\text{rest}(\text{head}(f)) & \text{sonst} \end{cases}$$

Beispiel 1:

palindrom(<o,t,t,o>)

= palindrom (rest (head (<o,t,t,o>)))
= palindrom (rest (<o,t, t>))
= palindrom (<t, t>)
= palindrom (rest (head (<t, t>)))
= palindrom (rest (<t>))
= palindrom (ε)
= wahr

Beispiel 2

palindrom : M* $\rightarrow$ B mit:

$$\text{palindrom}(f) = \begin{cases} \text{wahr} & \text{falls } (f = \varepsilon) \vee (\text{rest}(f) = \varepsilon) \\ \text{falsch} & \text{falls } \text{first}(f) \neq \text{last}(f) \\ \text{palindrom}(\text{rest}(\text{head}(f)) & \text{sonst} \end{cases}$$

Beispiel 2:

palindrom(o,r, t,o)

= palindrom (rest (head (<o,r,t,o>)))

= palindrom (rest (<o,r, t>))

= palindrom (<r, t>)

= falsch

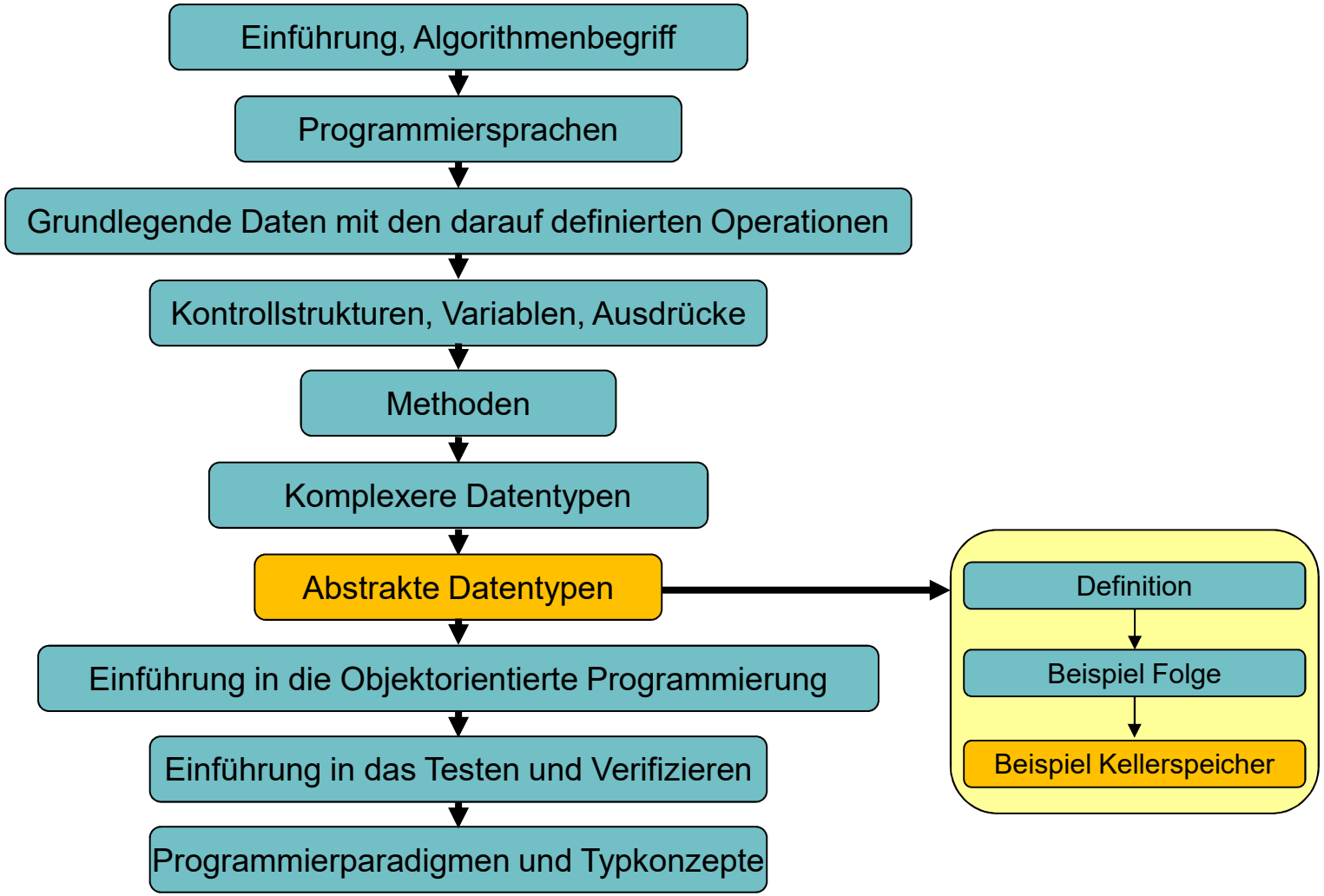
Zwischenstand

- Der abstrakte Datentyp einer Folge hat mehrere Operationen, die erzeugen, extrahieren und testen
- Ein komplexes Problem lässt sich lösen über die Aufteilung in Teilprobleme und deren Nutzung für die Gesamtlösung

Reflektion

- Geben Sie eine Funktion paargleich: M* $\rightarrow$ B an, die testet, ob alle Paare in einer Folge identisch sind.
- In der Folge <a,a,b,b,c,c> sind Paare <a,a>, <b,b> und <c,c>. Diese sind alle gleich, also wäre das Ergebnis hier wahr.

Inhalt dieser Veranstaltung



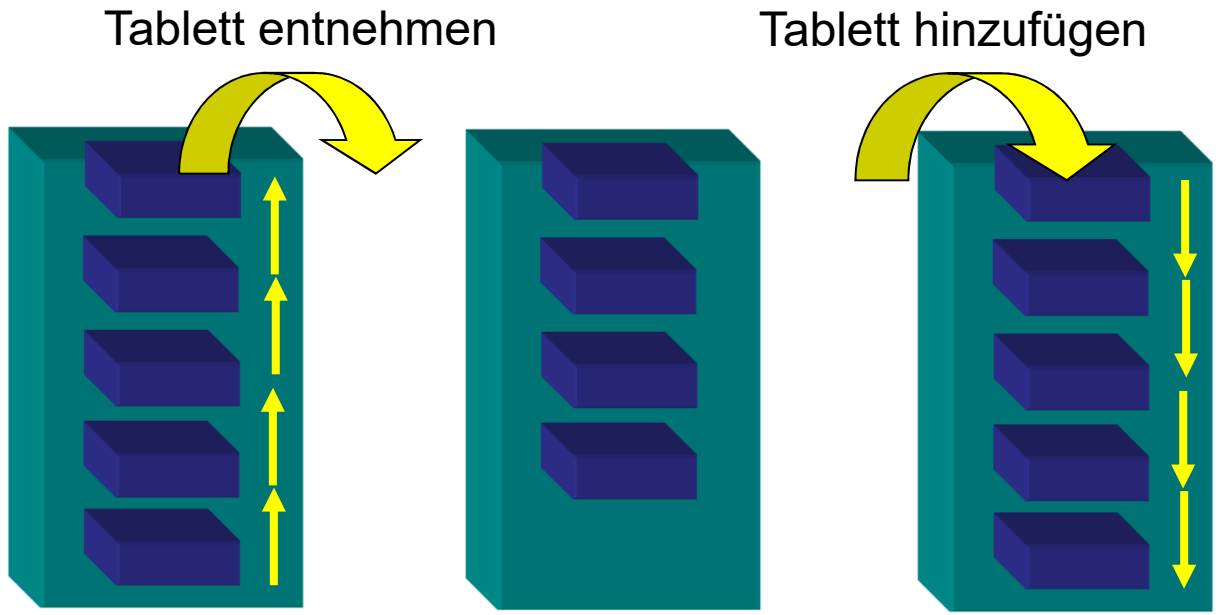
Kellerspeicher (Stack)

Tablettspender in der Mensa:

- Tablett von oben entnehmen
- Tablett oben drauflegen

Bekanntes Beispiel: Laufzeitstack mit Aufrufbereichen für Methodenaufrufe

Namen für dieses Schema: Kellerspeicher, Stack, Stapel



Abstrakter Datentyp stack

Sei M eine beliebige, nicht-leere Menge. Wir betrachten den Datentyp

$$stack = < W^{stack} , Ops^{stack} >$$

mit

$$W^{stack} = \{ M, M^*, B \}$$
$$Ops^{stack} = \{emptystack, isemptystack, push, top, pop\}$$

wobei B für die Menge der Boolschen Werte steht.

Beispiel:
Für M = {Tablett | Tablett ist grau} bekommt man einen Stack von grauen Tabletts.

Ops_{stack}

emptystack : $\varnothing \rightarrow M^*$ mit

emptystack() = <>

isemptystack : $M^* \rightarrow B$ mit

$$\text{isemptystack}(s) = \begin{cases} \text{wahr} & \text{falls } s=<> \\ \text{falsch} & \text{sonst} \end{cases}$$

Ops_{stack}

push : $M^* \times M \rightarrow M^*$ mit

$$push(s,x) = \langle x, x_1,\ldots,x_n \rangle \quad \text{falls } s = \langle x_1,\ldots,x_n \rangle, n \geq 0$$

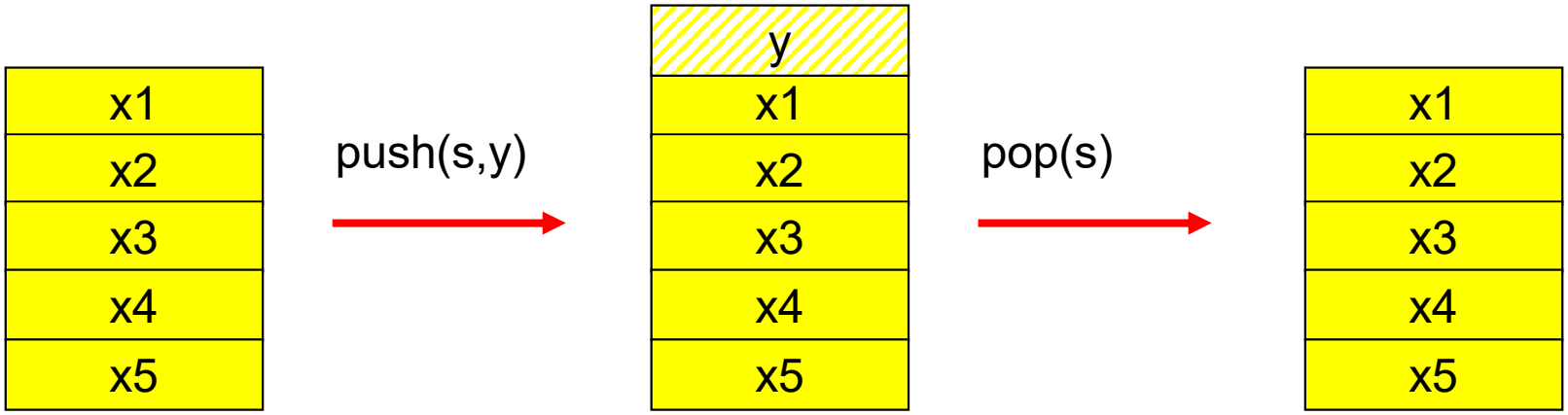
top : $M^* \text{ -} \rightarrow M$ mit

$$top(s) = \begin{cases} x_1 & \text{falls } s = \langle x_1,\ldots,x_n \rangle, n > 0 \\ \text{undef.} & \text{sonst} \end{cases}$$

pop : $M^* \text{ -} \rightarrow M^*$ mit

$$pop(s) = \begin{cases} \langle x_2,\ldots,x_n \rangle & \text{falls } s = \langle x_1,\ldots,x_n \rangle, n > 0 \\ \text{undef.} & \text{sonst} \end{cases}$$

push und pop



Anwendung 1

Problem:

Spiegeln eines Wortes w = $<x_1,...,x_n>$ soll $< x_n,...,x_1 >$ ergeben.

Lösung:

1. Ein Stack von Buchstaben
2. Algorithmus *spiegeln*

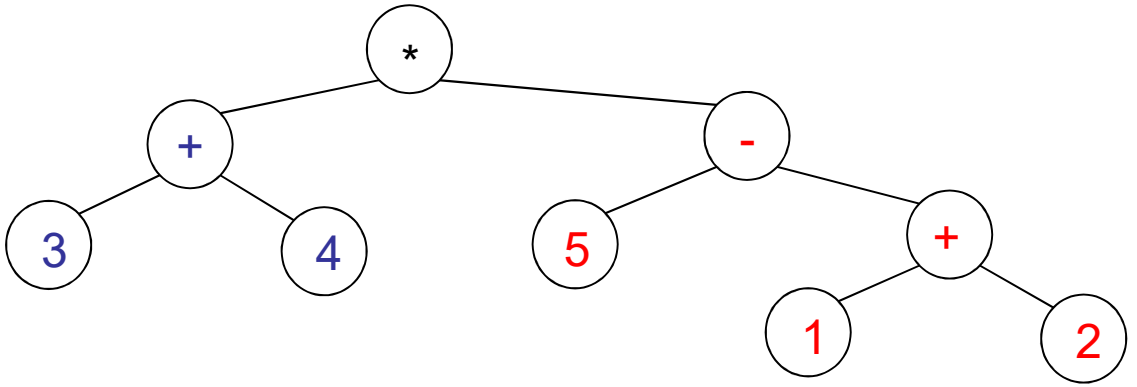
Anwendung 1

```
          Modul spiegeln (wort)
S1          s := emptystack()
S2          solange nicht isemptyfolge(wort)
S3            s := push (s, first(wort))
S4            wort := rest(wort)
S5          end
S6          spiegelwort := <>
S7          solange nicht isemptystack(s)
S8            spiegelwort := concat (spiegelwort, makefolge(top(s)))
S9            s := pop(s)
S10         end
S11         spiegeln := spiegelwort
          Modulende
```

Stelle im Algorithmus	wort	spiegelwort	s
S_1	<a,b,c>	Undef.	Undef.
S_5 (1. mal)	<b,c>	Undef.	<a>
S_5 (2. mal)	<c>	Undef.	<b,a>
S_5 (3. Mal)	<>	Undef.	<c,b,a>
S_{10} (1. Mal)	<>	<c>	<b,a>
S_{10} (2. Mal)	<>	<c,b>	<a>
S_{10} (3. Mal)	<>	<c,b,a>	<>

Anwendung 2

Problem:
Auswerten arithmetischer Ausdrücke in Postfixnotation / UPN,
z.B. 3 4 + 5 1 2 + - *



Lösung:
Abspeichern von Operanden und Zwischenergebnissen auf einem Stack

Algorithmus

```
Funktion auswerten (audruck)
s := emptystack()
solange nicht isemptyfolge(ausdruck)
  x := first(ausdruck)
S4Fallunterscheidung nach x:
    Falls x ∈ Z:
      s := push(s,x)
    Falls x = +:
      x1 := top(s); s = pop(s)
      x2 := top(s); s = pop(s)
      x := x2 + x1
      s := push(s,x)
    Falls x = -:
      x1 := top(s); s = pop(s)
      x2 := top(s); s = pop(s)
      x := x2 - x1
      s := push(s,x)
    analog *, /
  Ende Fallunterscheidung
  ausdruck := rest(ausdruck)
End
S30 Ergebnis auswerten := top(s)
Ende Funktion
```

Stelle	ausdruck	x	s
S_4 (1. mal)	<3 4 + 5 1 2 + - *>	3	<>
S_4 (2. mal)	<4 + 5 1 2 + - *>	4	< 3 >
S_4 (3. mal)	<+ 5 1 2 + - *>	+	< 4 3 >
S_4 (4. mal)	< 5 1 2 + - *>	5	< 7 >
S_4 (5. mal)	< 1 2 + - *>	1	< 5 7 >
S_4 (6. mal)	< 2 + - *>	2	< 1 5 7 >
S_4 (7. mal)	< + - *>	+	< 2 1 5 7 >
S_4 (8. mal)	< - *>	-	< 3 5 7 >
S_4 (9. mal)	< *>	*	< 2 7 >
S_{30}	<>	*	< 14 >

Zwischenstand

- Der abstrakte Datentyp eines Stacks hat mehrere Operation, die erzeugen, extrahieren und testen
- Die Organisation der Daten in dem Stack und die Operationen auf dem Stack unterscheiden sich von der Folge

Reflektion

- Geben Sie ein möglichst kurzes Beispiel für eine Folge von Stackoperationen, die zu einem Problem führen wird.
- Kann man für das Beispiel des Auswertens von UPN Ausdrücken beim Programmieren eine maximale Größe des Stacks angeben?

Zusammenfassung

- Abstrakte Datentypen konzentrieren sich ausschließlich auf die Bedeutung der Operationen und Daten und nicht auf deren konkrete Realisierung
- In einem abstrakten Datentyp werden alle grundlegenden Operationen zusammengefasst
- Die Operationen führen immer zu konsistenten Zuständen des ADS
- Mit den Operationen lassen sich dann Anwendungen entwickeln