LMU LUDWIG-MAXIMILIANS-UNIVERSITY MUNICH
DEPARTMENT INSTITUTE FOR INFORMATICS
DATABASE SYSTEMS GROUP

Skript zur Vorlesung:

Einführung in die Informatik: Systeme und Anwendungen

Sommersemester 2013

Kapitel 1: Informationsverarbeitung durch Programme

Vorlesung: PD Dr. Peer Kröger
Übungen: Johannes Niedermayer

http://www.dbs.ifi.lmu.de/cms/ Einführung_in_die_Informatik_Systeme_und_Anwendungen

1.1 Informationen und Daten

- Was ist Informatik?
  - Wissenschaft, Technik und Anwendung der maschinellen Verarbeitung und Übermittlung von Daten

    [Gesellschaft für Informatik: Studien- und Forschungsführer Informatik, Springer-Verlag]

- Algorithmus
  - Grundlage jeglicher maschineller Informationsverarbeitung
  - Zentraler Begriff der Informatik
  - Systematische, „schematisch“ („automatisch“, „mechanisch“) ausführbare Verarbeitungsvorschrift
  - Beispiele aus dem Alltag:
    - Kochrezepte, Bedienungsanleitungen, Aufbauvorschriften
    - Mathematische Berechnungsverfahren (z.B. Summe 1 + 2 + … + n für ein beliebiges n)

1.1 Informationen und Daten

- Beispiel:
  - Berechne zu einer gegebenen Abfahrts- und Ankunftszeit eines Zuges seine Fahrtzeit (zur Vereinfachung: keine Fahrt dauert länger als 24 Stunden)
  - *Was* ist zu verarbeiten?
    - Abfahrts-/Ankunftszeit, Fahrtzeit: Dinge der realen Welt

      z.B. „dreizehn Uhr zehn“
  - *Wie* wird es verarbeitet?
    - Abstraktion: mathematische Modelle der realen Dinge

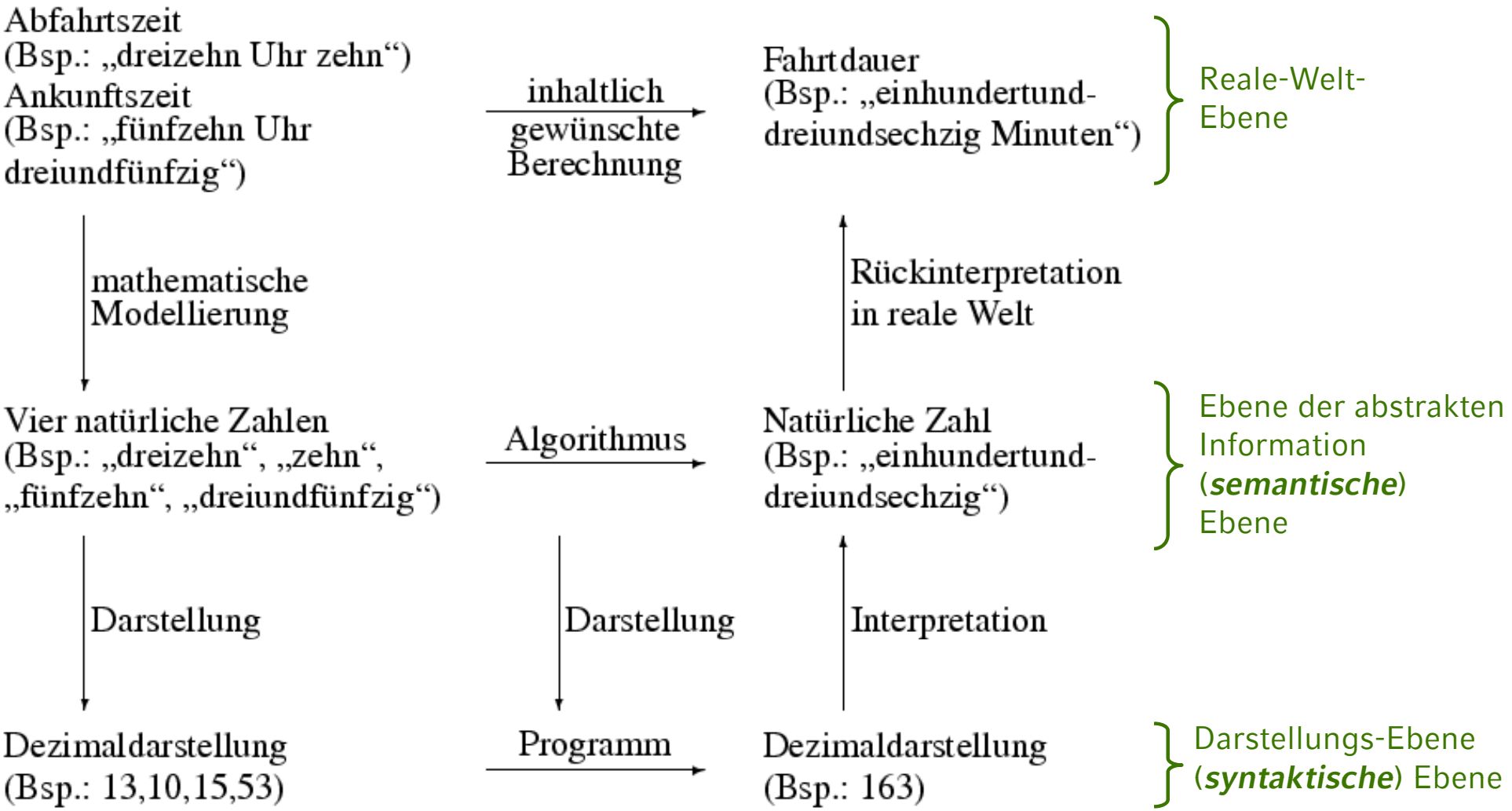
      z.B. die Zahlen „dreizehn“ und „zehn“
    - Verarbeitung der abstrakten „Eingabe“-Objekte liefert ein abstraktes „Ausgabe“-Objekt

      z.B. die Zahl „einhundertdreiundsechzig“
    - Rückinterpretation liefert: Antwort auf die gestellte Aufgabe in der realen Welt

      z.B. „Die Fahrtzeit beträgt einhundertdreiundsechzig Minuten“

1.1 Informationen und Daten

- Schematisch



1.1 Informationen und Daten

- Darstellung der Informationen
  - Zu verarbeiten (nach Abstraktion): Zahlen (mathematische Objekte), z.B. „dreizehn“
  - Notwendig: ***Darstellung*** der Zahlen
    - Typische Darstellung von „dreizehn“:

      13 (***Dezimaldarstellung***)
    - Andere Möglichkeiten:

      IIIIIIIIIIIII

      1101 (***Binärdarstellung***)

      XIII

      DREIZEHN

      (usw.)
  - Neben natürlichen Zahlen werden auch andere Typen von Daten (Sorten) verarbeitet:
    - Reelle Zahlen, ganze Zahlen
    - Zeichen, Zeichenketten (Strings)

1.2 Algorithmen

- Algorithmus für die Fahrtzeitberechnung
  - Eingabe: vier natürliche Zahlen, ***Parameter*** hierfür: h_{AB}, m_{AB}, h_{AN}, m_{AN}
  - Ergebnis: natürliche Zahl
  - Berechnung:
    - Falls Fahrt nicht über Mitternacht hinausgeht:

      Das Ergebnis ergibt sich aus $(h_{AN} - h_{AB}) \cdot 60 + m_{AN} - m_{AB}$
    - Falls Fahrt über Mitternacht geht

      Das Ergebnis ergibt sich aus $(23 - h_{AB}) \cdot 60 + (60 - m_{AB}) + h_{AN} \cdot 60 + m_{AN}$

- Bestandteile des Algorithmus
  - Daten, meist durch ***Variablen*** repräsentiert (z.B. h_{AB} repräsentiert die Stundenzahl der Abfahrtszeit)
  - Operationen auf den Daten („elementare Verarbeitungsschritte“)
  - „Zusammensetzung“ des Berechnungsvorgangs

1.2 Algorithmen

- Algorithmus
  - löst (typischerweise) eine Klasse von Aufgaben, die durch ihre ***Parameter*** (Eingabe-Variablen) bestimmt ist (in unserem Beispiel für beliebige An-/Ab-Zeiten). Eine ***Eingabe*** besteht aus konkreten (aktuell zu verarbeitenden) Daten für die Parameter (z.B. „13“ für h_{AB}).
  - seine ***Ausführung*** wird durch eine Eingabe erzeugt und liefert i.d.R. ***Ergebnisse***. Diese können Daten oder Steuersignale sein.

- Ausführungsbeispiel Fahrtzeit-Algorithmus
  - Eingabe 13, 10, 15, 53 für h_{AB}, m_{AB}, h_{AN}, m_{AN} liefert das Ergebnis 163
  - Eingabe 22, 15, 1, 30 für h_{AB}, m_{AB}, h_{AN}, m_{AN} liefert das Ergebnis 195

1.2 Algorithmen

- Grundanforderungen an Algorithmen
  - Präzise Darstellung:

    die zu verarbeitenden Daten und die Verarbeitungsvorschrift müssen unmissverständlich aufgeschrieben sein
  - Effektivität:

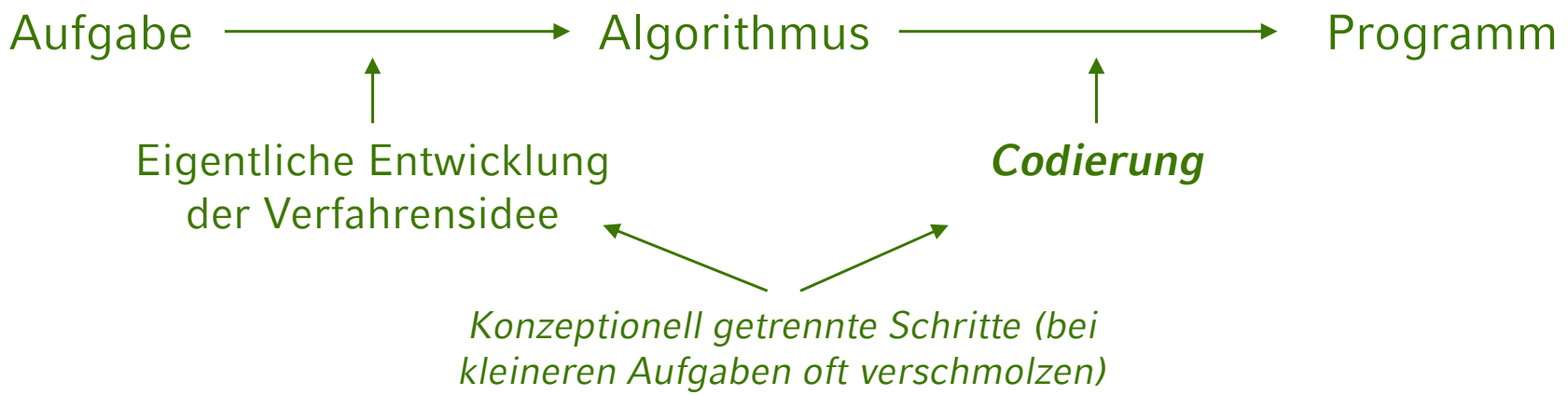
    jeder elementare Verarbeitungsschritt muss von der zugrunde liegenden „Verarbeitungseinheit“ (***Prozessor***) ausführbar sein

1.3 Programme

- Darstellung von Algorithmen
  - Zur Ausführung auf einem Computer muss ein Algorithmus, d.h. seine Daten, elementaren Verarbeitungsschritte und zusammengesetzte Verarbeitungsvorschrift

    formal in einer Programmiersprache (als ***Programm***) dargestellt werden

- Entwicklung von Algorithmen
  - Zentrale Aufgabe des Informatikers



1.3 Programme

- Pseudo-Code
  - Aufschreiben von Algorithmen in der Entwicklungsphase
  - Keine konkrete Programmiersprache
  - Programmiersprachenähnliche Darstellung
    - Verwendung algorithmischer Konzepte und mathematischer Schreibweisen
    - Meist verbale Zusätze
  - ***Gut für Menschen lesbar!!!***
  - Beispiel:

algorithmus Fahrtzeit
input h_{AB}, m_{AB}, h_{AN}, m_{AN} : natürliche Zahlen
output Fahrtzeit : natürliche Zahl
begin
if Fahrt geht nicht über Mitternacht
then Fahrzeit = $(h_{AN} - h_{AB}) \cdot 60 + m_{AN} - m_{AB}$;
else Fahrzeit = $(23 - h_{AB}) \cdot 60 + (60 - m_{AB}) + h_{AN} \cdot 60 + m_{AN}$;
end

1.3 Programme

- Bestandteile/Konzepte
  - Variablen
    - Intuitiv: Zettel, auf die ein Wert geschrieben werden kann
      - Der Wert kann abgelesen werden
      - Der Wert kann verändert werden (radieren und neu schreiben)
    - Formal: ein Abschnitt im Speicher
    - Typischerweise haben Variablen Typen (Sorten), d.h. der Zettel kann nur Werte eines speziellen Typs aufnehmen (z.B. Typ „natürliche Zahl“)
    - Wir gehen davon aus, dass es vordefinierte Typen gibt, die wir verwenden können

      Nat : „natürliche Zahlen“ **Real** : „reelle Zahlen“

      Int : „ganze Zahlen“

      Char : „druckbare Zeichen“ **String** : „Zeichenketten“

      Bool : „Wahrheitswerte“ (Wertemenge {*wahr*, *falsch*})

1.3 Programme

- Beispiel
  - h_{AB}, m_{AB}, h_{AN}, m_{AN} sind Variablen, die die Eingabewerte des Fahrzeitalgorithmus enthalten; die Variablen sind vom Typ „natürliche Zahl“ (**Nat**), d.h. sie können nur Werte von natürliche Zahlen speichern
    - » Dadurch ist der Algorithmus generell einsetzbar für beliebige An-/Abfahrtzeiten
    - » Beim Aufruf des Algorithmus mit konkreten Werten werden diese Werte auf den entsprechenden Zettel geschrieben
  - Fahrzeit ist eine Variable, die das Resultat des Algorithmus verwaltet; Typ: „natürliche Zahl“ (**Nat**)
  - Zudem können beliebig viele Variablen vereinbart werden, z.B. um Zwischenergebnisse zu speichern
- Die vordefinierten Datentypen stellen grundlegende Operationen zur Verfügung, z.B. die natürlichen Zahlen **Nat** die Multiplikation

1.3 Programme

- Anweisungen
  - Fahrzeit = $(h_{AN} - h_{AB}) \cdot 60 + m_{AN} - m_{AB}$;

    ist eine ***Anweisung***, der Wert der linke Seite wird in der Variablen Fahrzeit gespeichert (auf den „Zettel“ mit Namen Fahrzeit geschrieben)

    Jede Anweisung wird mit einem „;“ beendet!!!
  - Bedingte Anweisung

    if <Bedingung> **then** <Anweisung01> **else** <Anweisung02>

    führt je nachdem, ob die <Bedingung> wahr oder falsch ist, <Anweisung01> oder <Anweisung02> aus (dabei kann es sich jeweils um mehr als eine Anweisung handeln, einem sog. ***Block*** von Anweisungen innerhalb { } Klammern; der **else**-Fall kann fehlen)
  - Wiederholungsanweisungen

    while <Bedingung> **do** <Anweisung>

    testet, ob <Bedingung> wahr ist, wenn ja, wird <Anweisung> so oft ausgeführt bis <Bedingung> falsch ist; ist <Bedingung> falsch, wird mit den Anweisungen nach <Anweisung> fortgefahren

1.3 Programme

- Beispiel

  Programm zur Berechnung der Fakultät *n*! einer natürlichen Zahl *n*

```
algorithmus Fakultaet
input       n : Nat
output      Fakultaet : Nat
variables   i : Nat
begin
   i = 1 ;
   Fakultaet = 1 ;
   while i <= n
   do {
            Fakultaet = Fakultaet * i ;
            i = i + 1;
      }
end
```

Name des Algorithmus

Eingabevariable(n) (inkl. Typ)

Resultatvariable (inkl. Typ)

Vereinbarung der im Algorithmus verwendeten Variablen (inkl. Typ)

Anweisung (Wert-Zuweisung)

Anweisung (Wert-Zuweisung)

Anweisungsblock "Schleifenrumpf"

Wiederholungs-anweisung

Ende des Algorithms

1.3 Programme

- Beispiel

  Berechnung des Absolutbetrags $|n|$ einer natürlichen Zahl n

```
algorithmus Absolutbetrag
input       n : Nat
output      betrag : Nat
begin
   if   n <= 0
   then      betrag = -n;
   else      betrag = n;
end
```

1.3 Programme

- Abstraktion
  - Wir werden einen Block an Anweisungen abstrahieren, wenn die Details der Anweisungen keinen Rolle spielen
  - Beispiel

    if Details unwichtig **then** <Anweisungsfolge, die uns nicht interessiert>

- Kommentare
  - Kommentare könne an jeder Stelle platziert werden und dienen dazu, den Algorithmus zu erklären
  - Kommentare beginnen mit „//“ und enden am Zeilenende
  - Beispiel

```
i = k + 1;          // berechne k+1 und weise den Wert i zu        (einzeiliger Kommentar)
...
i = x + y;          // berechne die Summe aus den Werten der Variablen x und y
                    // und weise diesen Wert i zu        (Kommentar ueber mehrere Zeilen)
...
```

1.3 Programme

- Ausführung von Programmen
  - In unseren Formalismen bestehen Programme (Algorithmen) aus einer Reihe von Anweisungen
  - Ein Programm wird aufgerufen durch Belegung der Eingabeparameter mit konkreten Werten (aus dem entspr. Wertebereich)

    Es wird ausgeführt durch schrittweises Ausführen der einzelnen Anweisungen in der festgelegten Reihenfolge
  - Ein Programm (Algorithmus) kann auch andere Programme aufrufen (d.h. der Aufruf eines Programms ist eine gültige Anweisung):
    - Bsp: Aufruf des Algorithmus Fakultaet für eine Variable x

      Fakultaet(x); gibt x! als Wert zurück (abh. vom akt. Wert von x)

      Der Rückgabewert kann einer anderen Variablen zugeordnet werden:

      y = Fakultaet(x);
  - Um einen Algorithmus aufrufen zu können, benötigt man seine Signatur (Name, Eingabetypen, Rückgabetyp), z.B.

    Fakultaet: **Nat** → **Nat**

1.3 Programme

- Beispiel: Berechne die Differenz (als natürliche Zahl) zweier natürlicher Zahlen x und y (ohne zu wissen, welche die größere ist)

```
algorithmus Differenz
input     x,y : Nat
output    differenz : Nat
variables           diff: Nat
begin
        diff = x - y;
        differenz = Absolutbetrag(diff);
end
```

- Auswertung: Aufruf Differenz(2, 6)

⊥ = „leerer Zettel“

Anweisung	x	y	diff	differenz
	2	6	⊥	⊥
diff = x - y	2	6	-4	⊥
differenz = Absolutbetrag(diff)	2	6	-4	⊥
differenz = if diff <= 0 then –diff else …	2	6	-4	4
	2	6	-4	4

1.3 Programme

- Datenstrukturen
  - Oft muss man Mengen von Werten verarbeiten
  - Bisher können Variablen nur einzelne Werte (eines gewissen Typs) aufnehmen
  - Sog. ***Datenstrukturen*** verwalten Mengen von Daten
    - ***Listen*** verwalten Mengen von Daten ***gleichen*** Typs, z.B. eine Liste von **Nat**-Werten
    - ***Records*** (auch: ***Tupel***) verwalten Mengen von Daten beliebigen Typs, z.B. einen **Nat**-Wert, einen **Bool**-Wert und einen **Char**-Wert
  - Eine Datenstruktur bietet typischerweise „Schnittstellen“ (in Form von Operationen) an, die spezifizieren, wie man mit ihr „arbeiten“ kann, z.B.
    - wie man Daten ablegt (z.B. neue Werte an eine bestehende Liste anhängt)
    - auf einzelne Komponenten zugreift (z.B. auf den **Nat**-Wert eines Records)
    - wie viele Werte aktuell verwaltet werden (z.B. Länge der Liste), etc.
  - Viele Programmiersprachen bieten neben den Grundtypen solche Datenstrukturen für Mengen an und/oder ermöglichen die Definition solcher (benutzerdefinierten) Datenstrukturen

1.3 Programme

- Listen
  - verwalten Mengen von Daten ***gleichen*** Typs, z.B. eine Liste von **Nat**-Werten
  - Wir gehen davon aus, dass wir für jeden Grundtyp

    T ∈ { **Nat**, **Real**, **Int**, **Char**, **String**, **Bool** } einen entsprechenden Listentyp

    TList zur Verfügung haben.

    Im folgenden diskutieren wir diese Listentypen anhand des Typs **Nat**
  - Eine Liste von **Nat**-Werten (**NatList**) kann intuitiv als Liste von Zetteln angesehen werden, auf die **Nat**-Werte geschrieben werden können (analog mit jedem anderen Typ)
  - Eine Variable vom Typ **Nat**-Liste ist intuitiv ein „Verweis“ auf diese „Zettel-Liste“

    Variable liste : **NatList** liste → | 2 | 7 | 6 | 3 |

  - Als Operationen stellt eine Liste verschiedene Operationen zur Verfügung:
    - » Länge (Anzahl der Werte in der Liste)

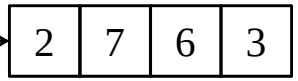
      Signatur: length: **NatList** → **Nat**

    Bsp.: liste → | 2 | 7 | 6 | 3 | length(liste) = 4

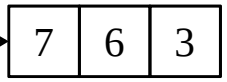
1.3 Programme

» Erster Wert in der Liste ablesen und löschen

Signatur: getAndDeleteFirst: **NatList** → **Nat**

Bsp.: liste → 

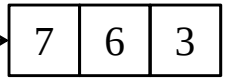
getAndDeleteFirst(liste) = 2

Anschließend gilt: liste → 

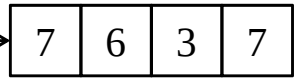
d.h., diese Operation hat einen ***Nebeneffekt***

» Einen Wert am Ende der Liste anhängen

Signatur: append: **NatList** × **Nat** → ∅ (∅ = „kein Rückgabewert“)

Bsp.: liste → 

Aufruf append(liste, 7)

hat folgenden Seiteneffekt: liste → 

1.3 Programme

- Beispiel: Algorithmus zur Suche eines gegebenen **Nat**-Wertes in einer Liste von **Nat**-Werten

```
algorithmus   SucheInListe
input         liste: NatList, wert: Nat
output        enthalten: Bool
begin
   enthalten = falsch;
   while   length(liste) > 0
   do {
      if        getAndDeleteFirst(liste) = wert
      then      enthalten = wahr;
   }
end
```

1.3 Programme

– Ablaufbeispiel

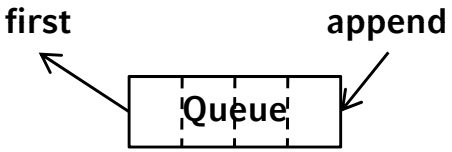
```
begin
enthalten = falsch;
 while length(liste) > 0
 do {
       if   getAndDeleteFirst(liste) = wert   then   enthalten = wahr;
}
end
```

	liste	wert	enthalten	length(liste)	getAndDeleteFirst(liste)
Initial/1. while-Schleife	2 7 6 3	6	*falsch*	4	2
2. while-Schleife	7 6 3	6	*falsch*	3	7
3. while-Schleife	6 3	6	*falsch*	2	6
4. while-Schleife	3	6	*wahr*	1	3
Schleifenabbruch	∅	6	*wahr*	0	⊥

1.3 Programme

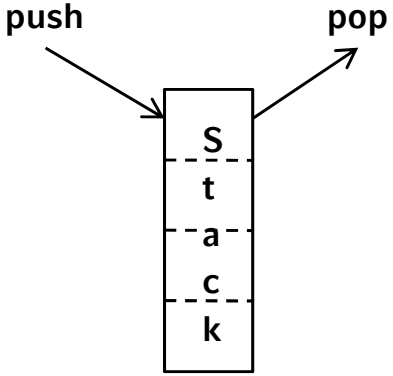
- Spezielle Listen bieten spezielle Schnittstellen mit speziellen Funktionalitäten
  - ***Queue*** (***Schlange***) implementiert eine klassische Warteschlange, d.h. es besteht nur Zugriff auf den Wert, der als ältestes in der Liste ist/als erstes in die Liste kam (Prinzip: First-in-first-out, FIFO)

    Schnittstelle:
    - » append: Wert an (Ende der) Schlange anfügen
    - » first: ersten Wert aus Liste „herausholen“
    - » length: Länge der Schlange



  - ***Stack*** (***Stapel***, ***Keller***) implementiert einen klassischen Stapel, d.h. es besteht nur Zugriff auf den Wert, der als jüngstes in der Liste ist/als letztes in die Liste kam (Prinzip: Last-in-first-out, LIFO)

    Schnittstelle:
    - » push: Wert auf den Stapel legen
    - » pop: obersten Wert des Stapels „abholen“
    - » length: Länge des Stapels



1.3 Programme

- Records: Darstellung von Daten unterschiedlicher Typen
- Motivation: wir wollen Daten über Kunden verwalten (Name, Alter, Kontostand); wie können wir diese Daten in einer Variable verwalten?

```
RECORD Kunde =
(
    name: String,
    alter : Nat,
    kontostand : Real,
)
```

- Ein Record ist intuitiv eine Sammlung von Zettel (beliebigen Typs), hier z.B. drei Zettel, für die einzelnen ***Komponenten*** (***Felder***)
- Eine Variable vom Typ **Kunde** ist intuitiv ein „Verweis“ auf diese drei Zettel. Jeder dieser Zettel kann mit seinem „Namen“ angesprochen werden

1.3 Programme

- Kombination von Listen und Records:
  - Prinzipiell können beide Konzepte Kombiniert werden, d.h. wir können Listen von Record-Typen definieren (z.B. **KundenList**) und Komponenten von Records können Listen-Typen sein.