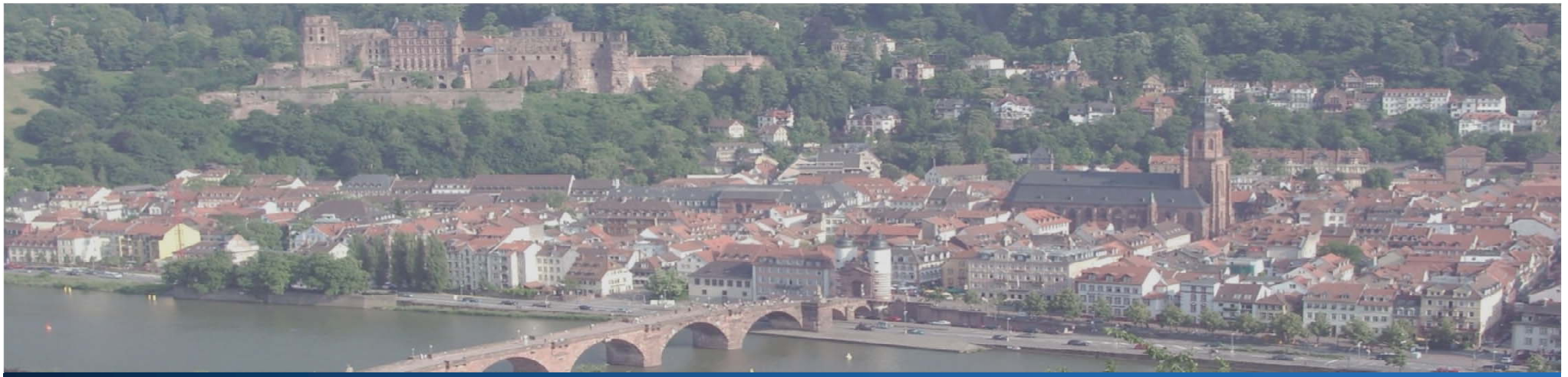
Einführung in die Praktische Informatik

Prof. Björn Ommer **HCI, IWR**

Computer Vision Group



Q & A

Fragetypen in der Klausur

- Code lesen, verstehen und schreiben können
- Fehler finden & Ergänzen von Programmen
- Algorithmische & theoretische Grundlagen
- Verständnisfragen, Definitionen, Begründungen
- Transfer von Konzepten auf neue Probleme
- Unterschiede von & Transfer zwischen Programmierkonzepten

- Stoff: Vorlesungen, Übungen

Polymorphismus

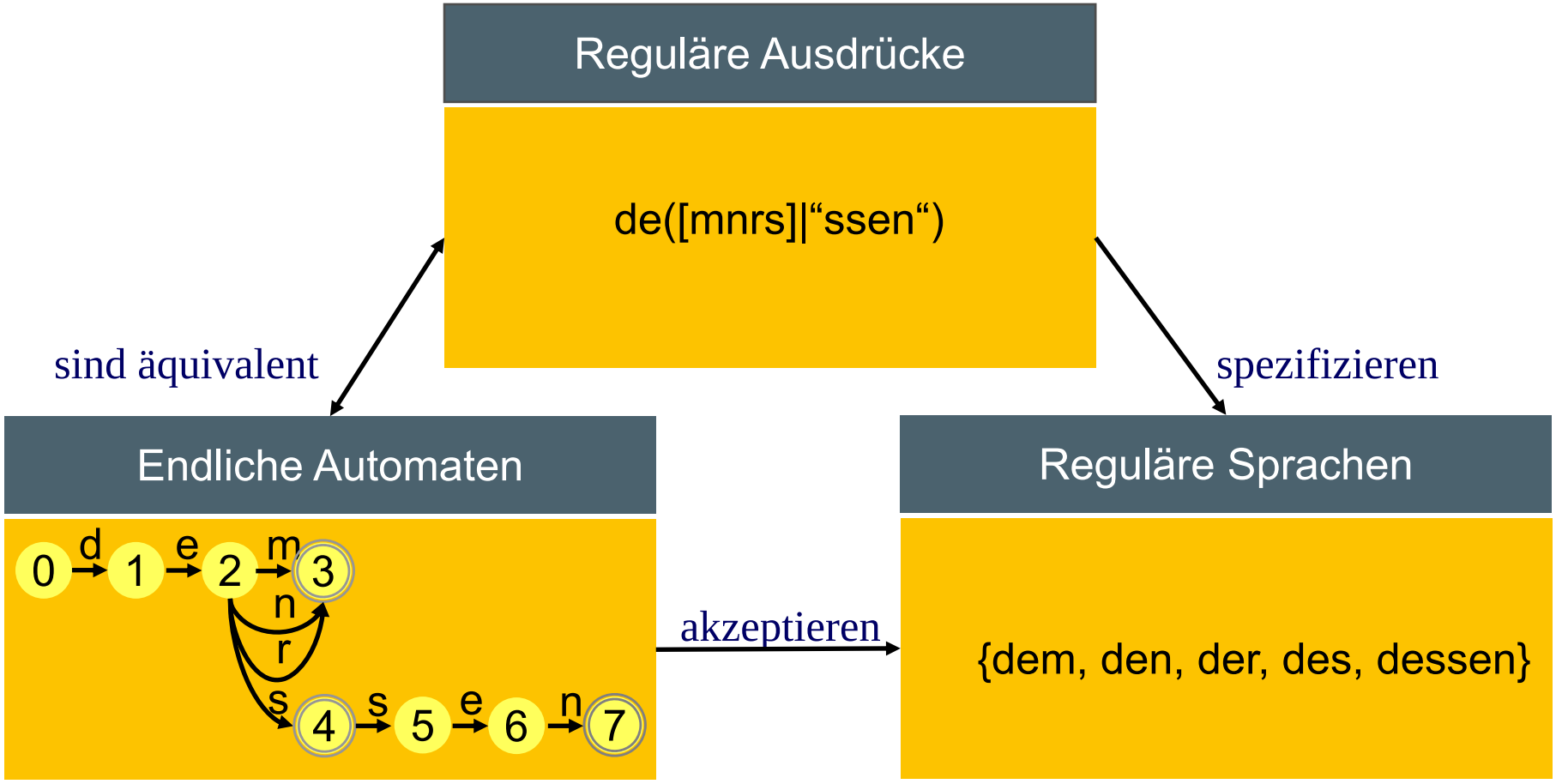
```
#include <iostream>
using namespace std;

class Shape {
   protected:
      int width, height;

   public:
      Shape( int a = 0, int b = 0){
         width = a;
         height = b;
      }
      int area() {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};
class Rectangle: public Shape {
   public:
      Rectangle( int a = 0, int b = 0):Shape(a, b) { }

      int area () {
         cout << "Rectangle class area :" <<endl;
         return (width * height);
      }
};
class Triangle: public Shape {
   public:
      Triangle( int a = 0, int b = 0):Shape(a, b) { }

      int area () {
         cout << "Triangle class area :" <<endl;
         return (width * height / 2);
      }
};
```

virtual

=0

```
// Main function for the program
int main() {
   Shape *shape;
   Rectangle rec(10,7);
   Triangle  tri(10,5);

   // store the address of Rectangle
   shape = &rec;

   // call rectangle area.
   shape->area();

   // store the address of Triangle
   shape = &tri;

   // call triangle area.
   shape->area();

   return 0;
}
```

Äquivalenzen: Endliche Automaten, reguläre Sprachen, reguläre Ausdrücke



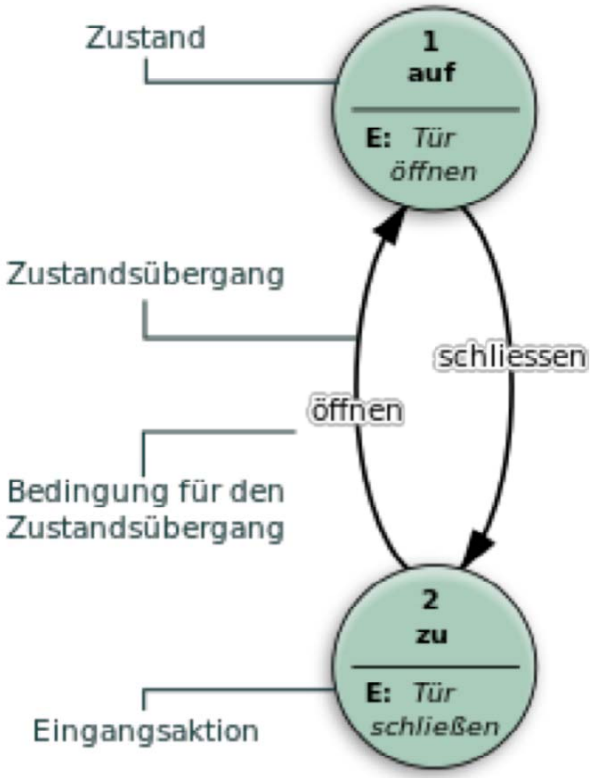
Von regulären Sprachen zu kontextfreien

- **Endliche Automaten (endl. Anzahl Zustände) erzeugen reguläre Sprachen:**

$A \rightarrow aB$ (rechtsregulär) oder
$A \rightarrow Ba$ (linksregulär)
$A \rightarrow a$
$A \rightarrow \varepsilon$
$A, B \in N, a \in T$

Nur links- oder nur rechtsreguläre Produktionen

T: Terminalsymbol
N: Nichterminalsymbole



Grenzen von endlichen Automaten

Der Automat akzeptiert folglich auch Wörter, die nicht zu $L = \{a^n b^n \mid n = 1, 2, 3, ...\}$ gehören. Das steht aber im Widerspruch zur Annahme, dass der Automat A die Sprache $L = \{a^n b^n \mid n = 1, 2, 3, ...\}$ erkennt.

Da die Annahme, dass es einen endlichen Automaten gibt, der die Sprache $L = \{a^n b^n \mid n = 1, 2, 3, ...\}$ erkennt, zu einem Widerspruch führt, muss die Annahme falsch sein.

Satz (über die Grenzen von endlichen Automaten):

Die Sprache $L = \{a^n b^n \mid n = 1, 2, 3, ...\}$ kann nicht von einem endlichen Automaten erkannt werden. Sie ist also nicht regulär.

Automatentheorie

- **klassifiziert Algorithmen nach der Art des Speichers, der für die Implementierung zum Merken von Zwischergebnissen gebraucht wird**

Spezialisierungen ↓

Automat	Speicher
Turingmaschine	unendlich großer Speicher
linear beschränkter Automat	endlich großer Speicher
Kellerautomat (Push Down Automaton)	Kellerspeicher (Stack)
endlicher Automat	kein zusätzlicher Speicher

Chomsky-Hierarchie

Typ	Grammatik	äquivalenter Automat
0	allgemein	(nicht)deterministische Turingmaschine
1	kontextsensitiv	nichtdet. linear beschr. Turingmaschine
2	kontextfrei	nichtdeterministischer Kellerautomat
3	regulär	(nicht)deterministischer endl. Automat

KFG – formale Definition

Eine kontextfreie Grammatik

$$G = (\Sigma, V, S, P)$$

besteht aus

- einer endlichen Menge Σ von **Terminalen** und einer endlichen Menge V von **Nichtterminalen** (oder **Variablen**).
  - Die Mengen Σ und V sind disjunkt, d.h. $\Sigma \cap V = \emptyset$ gilt.
  - Die Menge $W := \Sigma \cup V$ heißt **Vokabular**, die Elemente in W nennt man auch Symbole,
- einem Symbol $S \in V$, dem **Startsymbol** und
- einer endlichen Menge

$$P \subseteq V \times W^*$$

von **Produktionen**.
Für eine Produktion $(A, x) \in P$ schreiben wir meistens $A \to x$.

Eine Sprache $L \subseteq \Sigma^*$ heißt genau dann

kontextfrei,

wenn es eine kontextfreie Grammatik $G = (\Sigma, V, S, P)$ gibt mit

$$L = L(G).$$

KFG - Zusammenfassung

KFGs werden eingesetzt, um

rekursiv definierte Strukturen

zu modellieren.

Bsp.: Wohlgeformte Klammerausdrücke

Sei D die Sprache aller wohl-geformten Klammerausdrücke über $\Sigma = \{ (,) \}$.

1. **Wohl-geformte Klammerausdrücke** sind
   - (), ((())), (())()()((())), (()())
2. **Nicht wohl-geformt** sind
   - (()(), (((()

Es ist $D = L(G)$ für die kontextfreie Grammatik $G = (\Sigma, \{S\}, S, P)$ mit den Produktionen

$$S \to SS \mid (S) \mid \epsilon.$$

KFG und Programmiersprachen: Bsp. Pascal

Wir beschreiben einen (allerdings sehr kleinen) Ausschnitt von Pascal durch eine kontextfreie Grammatik.

- Wir benutzen das Alphabet $\Sigma = \{\texttt{a}, \ldots, \texttt{z}, \texttt{;}, :=, \texttt{begin}, \texttt{end}, \texttt{while}, \texttt{do}\}$ und
- die Variablen S, statements, statement, assign-statement, while-statement, variable, boolean, expression.
- variable, boolean und expression sind im Folgenden nicht weiter ausgeführt.

$$
\begin{aligned}
S &\rightarrow \texttt{begin}\ \text{statements}\ \texttt{end} \\
\text{statements} &\rightarrow \text{statement} \mid \text{statement ; statements} \\
\text{statement} &\rightarrow \text{assign-statement} \mid \text{while-statement} \\
\text{assign-statement} &\rightarrow \text{variable} := \text{expression} \\
\text{while-statement} &\rightarrow \texttt{while}\ \text{boolean}\ \texttt{do}\ \text{statements}
\end{aligned}
$$

KFG und Programmiersprachen

Lassen sich die syntaktisch korrekten Programme einer modernen Programmiersprache durch eine kontextfreie Sprache definieren?

- 1. Antwort: Nein. In Pascal muss zum Beispiel sichergestellt werden, dass Anzahl und Typen der formalen und aktuellen Parameter übereinstimmen.
  - Die Sprache $\{ww \ : \ w \in \Sigma^*\}$ wird sich als nicht kontextfrei herausstellen.
- 2. Antwort: Im Wesentlichen ja, wenn man „Details“ wie Typ-Deklarationen und Typ-Überprüfungen ausklammert:
  - Man beschreibt die Syntax durch eine kontextfreie Grammatik, die alle syntaktisch korrekten Programme erzeugt.
  - Allerdings werden auch syntaktisch inkorrekte Progamme (z.B. aufgrund von Typ-Inkonsistenzen) erzeugt.

Infix/Postfix Schreibweise, Auswertung

Der Compiler übersetzt den Ausdruck aus der Infix-Schreibweise in die äquivalente Präfixschreibweise. Die Auswertung des Ausdrucks, d. h. die Berechnung der Funktionen, erfolgt dann von innen nach aussen:

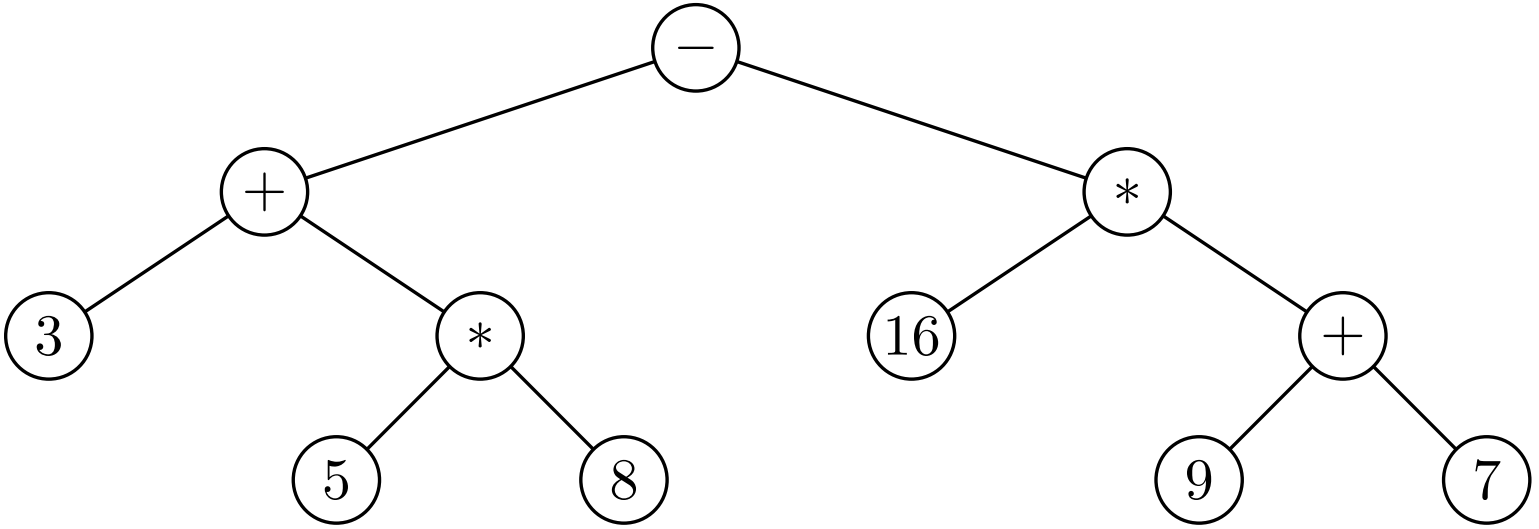
```
  -(+(3,*(5,8)),*(16,+(7,9)))
= -(+(3,  40  ),*(16,+(7,9)))
= -(    43     ,*(16,+(7,9)))
= -(    43     ,*(16,  16  ))
= -(    43     ,    256     )
=           -213
```

Bemerkung: Dies ist nicht die einzig mögliche Reihenfolge der Auswertung der Teiloperationen, alle Reihenfolgen führen jedoch zum gleichen Ergebnis! (Zumindest bei nicht zu großen, ganzen Zahlen).

Bemerkung: C++ kennt die Punkt-vor-Strich-Regel und das Assoziativgesetz. Überflüssige Klammern können also weggelassen werden.

Ausdrücke als Bäume

Jeder arithmetische Ausdruck kann als binärer Baum dargestellt werden. Die Auswertung des Ausdruckes erfolgt dann von den Blättern zur Wurzel. In dieser Darstellung erkennt man welche Ausführungsreihenfolgen möglich sind bzw. welche Teilausdrück gleichzeitig ausgewertet werden können (Datenflussgraph).



Korrektheit von Schleifen mittels Schleifeninvariante

Wir betrachten nun eine Variante des Hoare-Kalküls, mit der sich die Korrektheit von Schleifenkonstrukten nachweisen lässt. Dazu betrachten wir eine `while`-Schleife in der kanonischen Form

while (B(v)) { v = H(v); }

mit $\Rightarrow Q(v^k)$

- $v = (v_1, \ldots, v_m)$ dem Vektor von Variablen, die im Rumpf modifiziert werden,
- $B(v)$, der Schleifenbedingung und
- $H(v) = (H_1(v_1, \ldots, v_m), \ldots, H_m(v_1, \ldots, v_m))$ dem **Schleifentransformator**.

$$H^j(v) = \underbrace{H(H(\ldots H(}_{j \text{ mal}} v) \ldots))$$

Definition: (Schleifeninvariante)
Sei $v^j = H^j(v^0)$, $j \in \mathbb{N}_0$, die Belegung der Variablen nach j-maligem Durchlaufen der Schleife. Eine Schleifeninvariante INV(v) erfüllt:

1. INV(v^0) ist wahr.

2. INV$(v^j) \wedge B(v^j) \Rightarrow$ INV(v^{j+1}).

Gilt die Invariante vor Ausführung der Schleife und ist die Schleifenbedingung erfüllt, dann gilt die Invariante nach Ausführung des Schleifenrumpfes.

Angenommen, die Schleife wird nach k-maligem Durchlaufen verlassen, d. h. es gilt $\neg B(v^k)$. Ziel ist es nun zu zeigen, dass

$$\text{INV}(v^k) \wedge \neg B(v^k) \Rightarrow Q(v^k)$$

wobei $Q(v^k)$ die geforderte Nachbedingung ist.

Beispiel: Betrachte das Programm zur Berechnung der Fakultät von n:

```
t = 1; i = 2;
while ( i <= n ) { t = t*i; i = i+1; }
```

Behauptung: Sei $n \geq 1$, dann lautet die Schleifeninvariante:

$$\text{INV}(t, i) \equiv t = (i-1)! \wedge i - 1 \leq n.$$

1. Für $v^0 = (t^0, i^0) = (1, 2)$ gilt $\text{INV}(1, 2) \equiv 1 = (2-1)! \wedge (2-1) \leq n$. Wegen $n \geq 1$ ist das immer wahr.

2. Es gelte nun $\text{INV}(v^j) \equiv t^j = (i^j - 1)! \wedge i^j - 1 \leq n$ sowie $B(v^j) = i^j \leq n$. (Vorsicht v^j bedeutet **nicht** v hoch j!). Dann gilt
   - $t^{j+1} = t^j \cdot i^j = (i^j - 1)! \cdot i^j = i^j!$
   - $i^{j+1} = i^j + 1$, somit gilt wegen $i^j = i^{j+1} - 1$ auch $t^{j+1} = (i^{j+1} - 1)!$.
   - Schließlich folgt aus $B(i^j, t^j) \equiv i^j = i^{j+1} - 1 \leq n$ dass $\text{INV}(i^{j+1}, t^{j+1})$ wahr.

3. Am Schleifenende gilt $\neg(i \leq n)$, also $i > n$, also $i = n + 1$ da i immer um 1 erhöht wird. Damit gilt dann also

$$
\begin{aligned}
& \overbrace{\mathsf{INV}(i,t) \wedge \neg B(i,t)}^{t = (i-1)! \wedge i - 1 \leq n} \wedge i = n + 1 \\
\Leftrightarrow \quad & t = (i-1)! \wedge i - 1 = n \\
\Rightarrow \quad & t = n! \equiv Q(n)
\end{aligned}
$$

Für den Fall $n \geq 0$ muss man den Fall $0! = 1$ als Sonderfall hinzunehmen. Das Programm ist auch für diesen Fall korrekt und die Schleifeninvariante lautet $\mathsf{INV}(i,t) \equiv (t = (i-1)! \wedge i - 1 \leq n) \vee (n = 0 \wedge t = 1 \wedge i = 2)$.

Beispiel: Smart Pointer

Problem: Dynamisch erzeugte Objekte können ausschließlich über Zeiger verwaltet werden. Wie bereits diskutiert, ist die konsistente Verwaltung des Zeigers (bzw. der Zeiger) und des Objekts nicht einfach.

Abhilfe: Entwurf mit einem neuen Datentyp, der anstatt eines Zeigers verwendet wird. Mittels Definition von **operator**$*$ und **operator**$->$ kann man erreichen, dass sich der neue Datentyp wie ein eingebauter Zeiger benutzen lässt. In Copy-Konstruktor und Zuweisungsoperator wird dann *reference counting* eingebaut.

Bezeichnung: Ein Datentyp mit dieser Eigenschaft wird intelligenter Zeiger (*smart pointer*) genannt.

Programm: (Zeigerklasse zum *reference counting*, Ptr.hh)

```
template <class T>
class Ptr
{
  struct RefCntObj
  {
    int count;
    T* obj;
    RefCntObj( T* q ) { count = 1; obj = q; }
  };
  RefCntObj* p;

  void report()
  {
    std::cout << "refcnt = " << p->count << std::endl;
  }
```

```
void increment()
{
  p->count = p->count + 1;
  report();
}

void decrement()
{
  p->count = p->count - 1;
  report();
  if ( p->count == 0 )
  {
    delete p->obj; // Geht nicht fuer Felder!
    delete p;
  }
}
```

```
public:
  Ptr() { p = 0; }

  Ptr( T* q )
  {
    p = new RefCntObj( q );
    report();
  }

  Ptr( const Ptr<T>& y )
  {
    p = y.p;
    if ( p != 0 ) increment();
  }

  ~Ptr()
```

```
    {
      if ( p != 0 ) decrement();
    }

    Ptr<T>& operator=( const Ptr<T>& y )
    {
      if ( p != y.p )
      {
        if ( p != 0 ) decrement();
        p = y.p;
        if ( p != 0 ) increment();
      }
      return *this;
    }

    T& operator*() { return *(p->obj); }
    T* operator->() { return p->obj; }
};
```

Bemerkung:

- Man beachte die sehr einfache Verwendung durch Ersetzen der eingebauten Zeiger (die natürlich nicht weiterverwendet werden sollten!).
- Nachteil: mehr Speicher wird benötigt (das `RefCntObj`)
- Es gibt verschiedene Möglichkeiten, *reference counting* zu implementieren, die sich bezüglich Speicher- und Rechenaufwand unterscheiden.
- Die hier vorgestellte Zeigerklasse funktioniert (wegen **delete**[]) nicht für Felder!
- *Reference counting* funktioniert nicht für Datenstrukturen mit Zykeln $\rightsquigarrow$ andere Techniken zur automatischen Speicherverwaltung notwendig.

Funktionale Programmierung

Berechnungsmodell aus folgenden Bestandteilen:

1. Auswertung von zusammengesetzten Ausdrücken aus Zahlen, Funktionsaufrufen und Selektionen.

2. Konstruktion neuer Funktionen.

Namen treten dabei in genau zwei Zusammenhängen auf:

1. Als Symbole für Funktionen.

2. Als formale Parameter in Funktionen.

Im Substitutionsmodell werden bei der Auswertung einer Funktion die formalen Parameter im Rumpf durch die aktuellen Werte ersetzt und dann der Rumpf ausgewertet.

Zur prozeduralen Programmierung...

Unter Vernachlässigung von Resourcenbeschränkungen (endlich große Zahlen, endlich viele rekursive Funktionsauswertungen) kann man zeigen, dass dieses Berechnungsmodell äquivalent zur Turingmaschine ist.

In der Praxis erweist es sich als nützlich, weitere Konstruktionselemente einzuführen um einfachere, übersichtlichere und wartbarere Programme schreiben zu können.

Der Preis dafür ist, dass wir uns von unserem einfachen Substitutionsmodell verabschieden müssen!

Zur prozeduralen Programmierung...

Bemerkung:

- Wir können uns vorstellen, dass einem Namen ein Wert zugeordnet wird.
- Einem formalen Parameter wird der Wert bei Auswertung der Funktion zugeordnet.
- Einer Konstanten kann nur einmal ein Wert zugeordnet werden.
- Die Auswertung solcher Funktionsrümpfe erfordert eine Erweiterung des Substitutionsmodells:
  - Ersetze formale Parameter durch aktuelle Parameter.
  - Erzeuge (der Reihe nach !) die durch die Zuweisungen gegebenen Name-Wert Zuordnungen und ersetze neue Namen im Rest des Rumpfes durch den Wert.

Anweisungsfolgen (Sequenz)

- Funktionale Programmierung bestand in der Auswertung von Ausdrücken.
- Jede Funktion hatte nur eine einzige Anweisung (`return`).
- Mit Einführung von Zuweisung (oder allgemeiner Nebeneffekten) macht es Sinn, die *Ausführung mehrer Anweisungen* innerhalb von Funktionen zu erlauben. Diesen Programmierstil nennt man auch *imperative Programmierung*.

Bedingte Anweisung (Selektion)

Anstelle des `cond`-Operators wird in der imperativen Programmierung die bedingte Anweisung verwendet.

Syntax: **(Bedingte Anweisung, Selektion)**

$$<\text{Selektion}> \quad ::= \quad \underline{\text{if}}\ \underline{(}\ <\text{BoolAusdr}>\ \underline{)}\ <\text{Anweisung}> \\ [\ \underline{\text{else}}\ <\text{Anweisung}>\ ]$$

Ist die Bedingung in runden Klammern wahr, so wird die erste Anweisung ausgeführt, ansonsten die zweite Anweisung nach dem `else` (falls vorhanden).

Genauer bezeichnet man die Variante **ohne** `else` als bedingte Anweisung, die Variante **mit** `else` als Selektion.

Objektorientiertes Programmieren

Motivation

Bisher:

- Funktionen bzw. Prozeduren (Funktion, bei welcher der Seiteneffekt wesentlich ist) als *aktive* Entitäten
- Daten als *passive* Entitäten.

Beispiel:

```
int konto1 = 100;
int konto2 = 200;
int abheben( int& konto, int betrag )
{
  konto = konto - betrag;
  return konto;
}
```

Kritik:

- Auf welchen Daten operiert `abheben`? Es könnte mit jeder **int**-Variablen arbeiten.
- Wir könnten `konto1` auch ohne die Funktion `abheben` manipulieren.
- Nirgends ist der Zusammenhang zwischen den globalen Variablen `konto1`, `konto2` und der Funktion `abheben` erkennbar.

Idee: Verbinde Daten und Funktionen zu einer Einheit!

Objektdefinition

Die Klasse kann man sich als Bauplan vorstellen. Nach diesem Bauplan werden Objekte (*objects*) erstellt, die dann im Rechner existieren. Objekte heißen auch Instanzen (*instances*) einer Klasse.

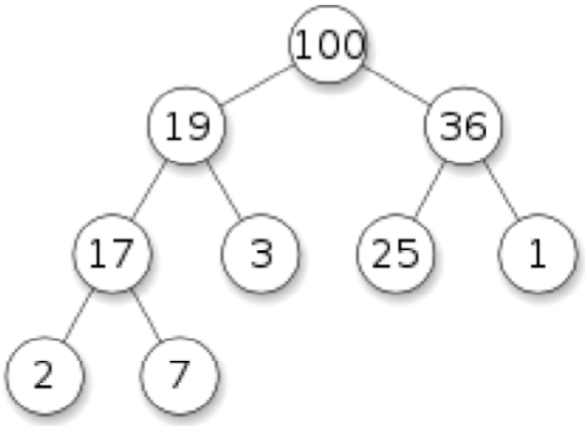
Objektdefinitionen sehen aus wie Variablendefinitionen, wobei die Klasse wie ein neuer Datentyp erscheint. Methoden werden wie Komponenten eines zusammengesetzten Datentyps selektiert und mit Argumenten wie eine Funktion versehen.

Bemerkung: Objekte haben einen internen Zustand, der durch die Datenmitglieder repräsentiert wird. Objekte haben ein Gedächtnis!

Heaps

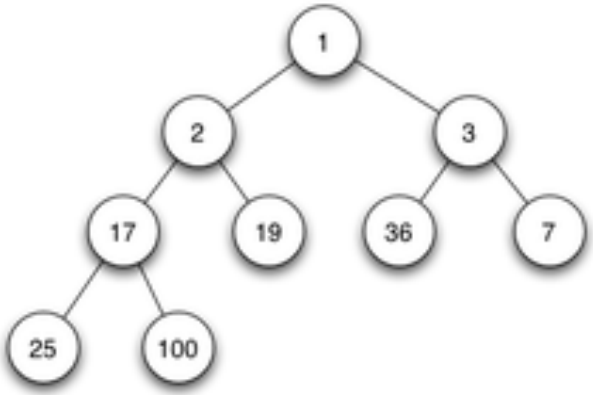
max-heap

Elter >= Kinder



min-heap

Elter <= Kinder



Umgang mit negativen Zahlen

In der Mathematik ist das Vorzeichen eine separate Information welche 1 Bit zur Darstellung benötigt.

Im Rechner wird bei ganzen Zahlen zur Basis $\beta = 2$ eine andere Darstellung gewählt, die Zweierkomplementdarstellung.

Früher war auch das Einerkomplement gebräuchlich.

Einer- und Zweierkomplement

Definition: Sei $(a_{n-1}a_{n-2}\dots a_1a_0)_2$ die Binärdarstellung von $a \in [0, 2^n - 1]$. Dann heisst

$$e_n(a) = e_n((a_{n-1}a_{n-2}\dots a_1a_0)_2) = (\bar{a}_{n-1}\bar{a}_{n-2}\dots \bar{a}_1\bar{a}_0)_2$$

das Einerkomplement von a, wobei $\bar{a}_i = 1 - a_i$.

Definition: Sei $a \in [0, 2^n - 1]$. Dann heisst

$$z_n(a) = 2^n - a$$

das Zweierkomplement von a.

Es gilt: $e_n(e_n(a)) = a$ und $z_n(z_n(a)) = a$!

Das Zweierkomplement einer Zahl kann sehr einfach und effizient berechnet werden.

Sei $a \in [0, 2^n - 1]$. Dann folgt aus der Identität

$$a + e_n(a) = 2^n - 1$$

die Formel

$$z_n(a) = 2^n - a = e_n(a) + 1.$$

Somit wird keine Subtraktion benötigt sondern es genügt das Einerkomplement und eine Addition von 1. (Und das kann noch weiter vereinfacht werden).

Definition: Die Zweierkomplementdarstellung einer Zahl ist eine bijektive Abbildung

$$d_n : [-2^{n-1}, 2^{n-1} - 1] \to [0, 2^n - 1]$$

welche definiert ist als

$$d_n(a) = \begin{cases} a & 0 \leq a < 2^{n-1} \\ 2^n - |a| & -2^{n-1} \leq a < 0 \end{cases} .$$

Die negativen Zahlen $[-2^{n-1}, -1]$ werden damit auf den Bereich $[2^{n-1}, 2^n - 1]$ positiver Zahlen abgebildet.

Sei $d_n(a) = (x_{n-1}x_{n-2}\ldots x_1x_0)_2$ dann ist a positiv falls $x_{n-1} = 0$ und a negativ falls $x_{n-1} = 1$.

Es gilt $d_n(-1) = 2^n - 1 = (1, \ldots, 1)_2$ und $d_n(-2^{n-1}) = 2^n - 2^{n-1} = 2^{n-1}(2-1) = 2^{n-1} = (1, 0, \ldots, 0)_2$.

Operationen mit der Zweierkomplementdarstellung

Die Ein-/Ausgabe von ganzen Zahlen erfolgt (1) im Zehnersystem und (2) mittels separatem Vorzeichen.

Bei der Eingabe einer ganzen Zahl wird diese in das Zweierkomplement umgewandelt:

- Lese Betrag und Vorzeichen ein und teste auf erlaubten Bereich
- Wandle Betrag in das Zweiersystem
- Für negative Zahlen berechne das Zweierkomplement

Bei der Ausgabe gehe umgekehrt vor.

Im folgenden benötigen wir noch eine weitere Operation.

Definition: Sei $x = (x_{m-1}x_{m-2}\dots x_1x_0)_2$ eine m-stellige Binärzahl. Dann ist

$$s_n((x_{m-1}x_{m-2}\dots x_1x_0)_2) = \begin{cases} (x_{m-1}x_{m-2}\dots x_1x_0)_2 & m \le n \\ (x_{n-1}x_{n-2}\dots x_1x_0)_2 & m > n \end{cases}$$

die Beschneidung auf n-stellige Binärzahlen.

Bsp.: 10110111
m=8, n=5

Die Addition von Zahlen in der Zweierkomplementdarstellung gelingt mit

Satz: Sei $n \in \mathbb{N}$ und $a, b, a+b \in [-2^{n-1}, 2^{n-1}-1]$. Dann gilt

$$d_n(a+b) = s_n(d_n(a) + d_n(b)).$$

Es genügt eine einfache Addition. Beachtung der Vorzeichen entfällt!

Beispiel: (Zweierkomplement) Für $n = 3$ setze

0	=	000	-1	=	111
1	=	001	-2	=	110
2	=	010	-3	=	101
3	=	011	-4	=	100

Solange der Zahlenbereich nicht verlassen wird, klappt die normale Arithmetik ohne Beachtung des Vorzeichens:

3	$\rightarrow$	011
-1	$\rightarrow$	111
2	$\rightarrow$	[1]010

Komplexität

Idee: Der Algorithmus Bubblesort ist folgendermaßen definiert:

- Gegeben sei ein Feld $a = (a_0, a_1, a_2, \ldots, a_{n-1})$ der Länge n.
- Durchlaufe die Indizes $i = 0, 1, \ldots, n-2$ und vergleiche jeweils a_i und a_{i+1}. Ist $a_i > a_{i+1}$ so vertausche die beiden. Beispiel:

$$
\begin{array}{lllll}
 & 17 & 10 & 8 & 16 \\
i=0 & 10 & 17 & 8 & 16 \\
i=1 & 10 & 8 & 17 & 16 \\
i=2 & 10 & 8 & 16 & 17
\end{array}
$$

  Am Ende eines solchen Durchlaufes steht die größte der Zahlen sicher ganz rechts und ist damit an der richtigen Position.

- Damit bleibt noch ein Feld der Länge $n-1$ zu sortieren.

Satz: t_{cs} sei eine obere Schranke der Kosten für einen Vergleich und einen swap (Tausch), und n bezeichne die Länge des Felds. Falls t_{cs} nicht von n abhängt, so hat Bubblesort eine asymptotische Laufzeit von $O(n^2)$.

Beweis:

$$\sum_{i=0}^{n-1}\sum_{j=0}^{i-1} t_{cs} = t_{cs}\sum_{i=0}^{n-1} i = t_{cs}\frac{(n-1)n}{2} = O(n^2)$$